Università degli Studi di Genova

PhD Program in Computer Science and Systems Engineering

Large Scale Shallow Learning: Methods and Applications

by

Giacomo Meanti

Thesis submitted for the degree of *Doctor of Philosophy* (XXXV cycle)

March 2023

Lorenzo Rosasco	Supervisor
Giorgio Delzanno	Head of the PhD program

Dibris

Department of Informatics, Bioengineering, Robotics and Systems Engineering

Acknowledgements

I am extremely grateful to prof. Rosasco for teaching me what I know about doing research, and to prof. De Vito and all other MaLGa faculty members for having fostered an amazing research environment in Genova. I would also like to thank prof. Recht and his students for the welcoming me warmly in Berkeley. Special thanks to Luigi, Stefano, Sara, Daniele, Federico, and all the great researchers with whom I have had the honor of collaborating, as well as everyone who has spent time in the labs of the Machine Learning Genova Center for many fruitful discussions, shared meals, and for the friendly community they have helped create.

Abstract

Kernel methods are among the most flexible classes of machine learning models with strong theoretical guarantees. Wide classes of functions can be approximated arbitrarily well with kernels, while fast convergence and learning rates have been formally shown to hold. Exact kernel methods are known to scale poorly with increasing dataset size, and we believe that one of the factors limiting their usage in modern machine learning is the lack of scalable and easy to use algorithms and software. The main goal of this thesis is to study kernel methods from the point of view of efficient learning, with particular emphasis on large-scale data, but also on low-latency training, and user efficiency. We improve the state-of-the-art for scaling kernel solvers to datasets with billions of points using the Falkon algorithm, which combines random projections with fast optimization. Running it on GPUs, we show how to fully utilize available computing power for training kernel machines. To boost the ease-of-use of approximate kernel solvers, we propose an algorithm for automated hyperparameter tuning. By minimizing a penalized loss function, a model can be learned together with its hyperparameters, reducing the time needed for user-driven experimentation. In the setting of multi-class learning, we show that – under stringent but realistic assumptions on the separation between classes – a wide set of algorithms needs much fewer data points than in the more general setting (without assumptions on class separation) to reach the same accuracy. The first part of the thesis develops a framework for efficient and scalable kernel machines. This raises the question of whether our approaches can be used successfully in real-world applications, especially compared to alternatives based on deep learning which are often deemed hard to beat. The second part aims to investigate this question on two main applications, chosen because of the paramount importance of having an efficient algorithm. First, we consider the problem of instance segmentation of images taken from the iCub robot. Here Falkon is used as part of a larger pipeline, but the efficiency afforded by our solver is essential to ensure smooth human-robot interactions. In the second instance, we consider time-series forecasting of wind speed, analysing the relevance of different physical variables on the predictions themselves. We investigate different schemes to adapt i.i.d. learning to the time-series setting. Overall, this work aims to demonstrate, through novel algorithms and examples, that kernel methods are up to computationally demanding tasks, and that there are concrete applications in which their use is warranted and more efficient than that of other, more complex, and less theoretically grounded models.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Machine learning has a wide-ranging impact on academic research, businesses, and even consumer-facing applications, having revolutionized the fields of data science, business intelligence, artificial intelligence, *etc.* One of the major problems that researchers are grappling with, is that of ever-growing model complexity in the face of a lack of theoretical guarantees. A tradeoff is emerging between large, overparameterized models – which need large amounts of dedicated computational resources to run – and small but efficient ones. The former have claimed renowned advances in forecasting accuracy for problems such as object recognition and general scene understanding, in the ability to translate between different data modalities, and in the capability of generative models to fool the human eye. The latter possess theoretical guarantees for convergence to the optimal solution, stability to data perturbations, and the rate at which errors will decrease with training-set size. However, their performance will be best on tabular or unstructured data, low-dimensional problems, or situations of data scarcity. One possible class within this tradeoff is that of *kernel machines*. With kernels, one can model complex non-linear relations within the data but is often limited by their poor scalability. As the number of data samples reaches hundreds of thousands, the time and memory required for training a kernel model quickly go from unreasonable to infeasible. In this thesis, we focus on both the efficiency and accuracy of kernel methods and how to obtain one while not having to do without the other.

Shallow learning models (Ma et al., 2017) are defined in antithesis to deep learning as models composed of two simple layers: first, a non-linear data transformation, followed by a linear step to complete the task of classification, regression, or unsupervised learning. The first layer has the purpose of making the trained model more expressive and capable of representing complex relationships, while the second makes it easy to optimize and characterize theoretically. Possibly the most notable representatives in this class of models are the already cited kernel methods, for which the transformed data lies in a reproducing kernel Hilbert space (Aronszajn, 1950), and which will be the main kind of model considered in this thesis. In particular, the driving motivation behind our work has been to bridge the gap between kernel methods and deep neural networks (at least on those problems where the two can obtain similar accuracy) by improving the usability of the former, addressing a few practical pain points which exist around their usage. Increasing the computational efficiency of training, alongside software usability, are the primary goals, but we also consider efficiency from the point of view of the time a practitioner has to spend on model tuning, which can be even more significant than compute time per se.

One of the biggest challenges hindering the practical usage of kernel machines for learning is connected to the *kernel matrix*, which is at the core of the learning algorithm but is also too large to be processed efficiently when the dataset size increases, as the number of entries it contains scales with the square of the number of data points. For this reason, there exists a rich literature on approximations of kernel-based learning algorithms, whose main goal is to circumvent calculating the full kernel matrix. The two most prominent approaches for kernel approximation are random Fourier features (RFF), and the Nyström method. The first are based on the Fourier decomposition of the kernel function into a set of randomized components which are fewer than the number of data points. We will touch upon RFF briefly in Chapter 2, but will devote more space to the Nyström approximation – also introduced in Chapter 2 and then used throughout the later chapters. It exploits a low-rank assumption on the kernel matrix, by constructing an approximation from a random subsample of the kernel's columns. This allows to greatly reduce the computational burden, while maintaining the same accuracy as if the full kernel had been used (at least in a worst case analysis).

Given the existence of algorithms with the potential to greatly reduce the bottlenecks of kernel methods, the persisting stigma around their inefficiency has to be attributed to the lack of modern, easy to use software tools that implement these improved models. Any attempt to develop such tools must face the increasing prevalence of hardware accelerators. They can greatly speed up computations but may require additional algorithmic considerations to be best utilized. We take an algorithm for approximate kernel regression – the Falkon algorithm introduced in Rudi, Carratino, et al. (2017) – which has optimal theoretical guarantees, and a computational cost that is much lower than that of the full problem. The algorithm is dissected to analyze and resolve performance bottlenecks, especially with regard to memory management. Indeed the latter aspect is especially important when developing algorithms on accelerators such as GPUs and TPUs which have limited memory, and for which the cost of transferring data onto the accelerator itself can be high. The implementation which results from our analysis is evaluated experimentally to confirm the high efficiency and scalability of the solver on problems ranging from small to huge datasets. It has also been packaged into an easy to use software library, which will be at the core of the following chapters of this thesis in which it is used as a general purpose kernel solver.

Another important aspect of modern machine learning (ML) is hyperparameter tuning. As the usage of ML in real-world problems becomes more prevalent, the models employed tend to become more complex due to the need to accommodate different data sources, application requirements, and business needs. Because of such a trend, the number of knobs that can be turned in any given model has been increasing, leading to a consequent increase in time to accurately tune them. AutoML has been proposing alternative systems which can automatically *tune themselves* with minimal user interaction, by searching among the different hyperparameter configurations in clever ways. Classically, one way to do this has been to carefully balance the bias and variance of the generalization error such that the obtained hyperparameters do not overfit the training set. The crucial difficulty here is in approximating the bias and variance terms efficiently. Following this strategy, we propose a novel regularized objective function that can be computed efficiently and is tailored to hyperparameter tuning for approximate kernel methods. Gradient-based optimization of this objective helps find good values for the tuning knobs, which in turn helps compress the kernel approximation even further. The user only needs to choose initial values for the hyperparameters, which can be done using simple heuristics.

Another aspect of the bias/variance decomposition which we tackled, is its usage in deriving learning bounds. Often such decomposition is calculated in a setting with minimal assumptions, which guarantees the generality of the resulting bounds but may not accurately represent specific learning scenarios. In classification, for example, the situation where the different classes are *well separated* (which for binary classification means that the true probability of each class is never close to the decision boundary at 0.5) leads to exponentially fast convergence of the error as the data points increase. This phenomenon has been frequently observed experimentally but can not be captured by default learning bounds. We concentrate on the multi-class problem, where we leverage a simplex encoding of the class labels to derive an appropriate definition of distance from the decision boundary. Assumptions on the distance lead to an error decomposition in which the bias term vanishes, and we recover exponential learning rates. Differently from previous works, our multi-class learning bounds apply not just when minimizing the squared loss (rarely used in practice for classification) but also for a more general class of *margin* losses which includes the logistic and exponential loss. In line with the rest of this thesis, we further connect the general learning bounds to the more specific setting of kernel learning.

One cannot effectively proclaim the advantages of the algorithms for kernel learning introduced up to now without discussing specific case studies in which they were used, thereby showcasing their benefits. Taking a different perspective from the methodological work of the first part of this thesis, in the second part we give priority to applications of kernel methods, and other *shallow* learning models. We wish to show that such methods can be the right fit for the job in a variety of cases, through a mix of expressiveness, interpretability, and crucially computational efficiency.

The first application we tackle is that of *instance segmentation*: given an input image, classify each of its pixels as belonging to an instance of a known object or to the background. This complex task consists of several components: understanding known objects, classifying individual image pixels, detecting the background, *etc.* Therefore it must be solved with a pipeline consisting of multiple interconnected components. Within this pipeline, we identify three tasks where linear and kernel-based models can be employed to improve overall efficiency and one task (that of feature extraction) that is better left to a large pre-trained deep learning model instead. In this way, once the expensive pre-training step is completed, we have a pipeline that can adapt to different environments and even learn new objects on the fly. Harnessing methods developed in the first part of the thesis (namely the Falkon library), we greatly reduced the running time of the proposed instance segmentation pipeline, allowing it to run in real-time on the iCub robot.

In the second instance, we applied shallow machine learning models to help analyze a problem coming from climate science: wind speed forecasting. While the processing speed offered by the Falkon library once again helps train a large number of models in a reasonable amount of time, this is not the main focus of the work since the datasets are relatively small, and there is no real-time component that would have required an especially fast solver. Instead, the main goal is to proceed backward from the forecasts to understand which combination of inputs (corresponding to physical observables) produces better results and to explain the mechanistic reasons which could cause such behavior. It is a commonly held belief that kernel methods and deep neural networks are of similar expressiveness (*i.e.* their accuracies are close) on tabular data, that is data with little structure. We compare our simple model based on kernel ridge regression on time-lagged data against more complex deep learning-based models proposed in the literature to attempt an empirical verification of this claim – at least for this specific setting. While confronting with other works is never trivial, especially since wind-speed forecasting is a small niche and no standard benchmarks exist, a comparison of kernel-based models and recurrent or convolutional neural networks shows that indeed the performance obtained is very similar. On the other hand, our model is easier to understand and much faster to run.

A final example in which shallow learning algorithms have been shown to perform remarkably well is in scene reconstruction or novel-view synthesis, in which 3-dimensional or 4-dimensional models of a real-life scene are reconstructed starting from calibrated photos or videos of the scene itself respectively. We adopt the volumetric rendering formulation introduced in Max (1995) and use an explicit but compressed representation, which unifies the handling of static, dynamic, and variable-appearance scenes that previously had to be handled separately. Setting kernel methods aside we propose two models. The first is fully explicit with no neural-network components at all, and uses a learned view-dependent basis akin to spherical harmonics. The second introduces small multi-layer perceptrons which slightly improve rendering accuracy.

In summary, the contributions of this thesis are structured as follows:

- In Chapter 3 (based on Meanti, Carratino, Rosasco, et al. (2020)) the Falkon algorithm is presented, along its efficient implementation specifically adapted to hardware accelerators. An extensive empirical evaluation containing ablation studies of individual performance improvements, scalability tests to datasets up to 1 billion points, and small data experiments concludes the chapter.

- Chapter 4 concerns a novel objective for gradient-based hyperparameter tuning in Nyström kernel ridge regression. Based on Meanti, Carratino, De Vito, et al. (2022), a short review of existing approaches to the problem is followed by the derivation of our objective through the bias/variance decomposition. The chapter concludes with experiments which characterize the advantages of the proposed method.

- Following Vigogna et al. (2022), in Chapter 5 exponential learning rates are derived under hard margin assumptions in the multiclass classification setting. The bounds are valid for a general class of losses, and reduce naturally to known results in the case of binary classification.

- In Chapter 6 (based on Ceola, Maiettini, Pasquale, Meanti, et al. (2022)), instance segmentation is approached with a pipeline which has the ability to be retrained quickly to handle new objects, and handles predictions with low latency in order to be implementable on the iCub robot.

- Chapter 7 details the application of kernel methods for wind speed forecasting, after Lagomarsino-Oneto et al. (2023). Thousands of models are trained to perform an analysis of variable importance, which is traced back to its physical meaning. Different ways of adapting supervised learning models to temporal data are investigated.

- Finally, in Chapter 8 (see also Fridovich-Keil, Meanti, et al. (2023)), a shallow-learning radiance field model is presented which unifies static and dynamic scenes using a compressed explicit representation named k-planes. The problem is briefly presented, followed by a description of the proposed model, and extensive experiments.

Chapter 2

Learning With Kernels

2.1 Supervised Learning

The supervised learning problem refers to the approximation of a function f which goes from an *input* space $\mathcal{X}$ to an output space $\mathcal{Y}$. As an example, a mail-spam filtering model will require learning a function from the input space of e-mail text and metadata (so $\mathcal{X}$ could be very high-dimensional), to an output space $\mathcal{Y} = \{\text{spam}, \text{not-spam}\}$. In order to learn such function a *training set* of input/output pairs is provided $Z = \{(x_1, y_1), \dots, (x_n, y_n)\} \subset \mathcal{X} \times \mathcal{Y}$, which is generated by a process involving the function f and some noise ϵ:

$$y_i = f(x_i) + \epsilon_i \qquad\qquad i = 1, \dots, n.$$

Let the approximate function learned with the training data be called $\hat{f}$. To evaluate the goodness of $\hat{f}$ we turn to the generalization erro: given a new, unseen input/output pair $(x_{\text{new}}, y_{\text{new}}) \in \mathcal{X} \times \mathcal{Y}$ how far is $\hat{f}(x_{\text{new}})$ from y_{new}, *i.e.* what is the error on this new point? If we measured the error on the training set instead, we would promote models which learn how to reproduce the training data exactly, including the noise which is always present. Such models are said to *overfit* the data, at the expense of learning the true function f^*. Models which generalize well on the other hand, will perform better at approximating the true function f on the new data which will only be available after training.

A more formal way of looking at supervised learning is through the lens of statistical learning theory. In order to get to a formal definition of the generalization error, we must introduce a way of sampling the data pairs, different loss functions to evaluate approximation quality and the concept of hypothesis space.

2.1.1 Data distribution

Consider the space $\mathcal{X} \times \mathcal{Y}$ as a probability space with distribution ρ. We denote with $\rho_{\mathcal{X}}$ the marginal distribution of ρ on $\mathcal{X}$ and with $\rho(\cdot|x)$ the conditional distribution on $\mathcal{Y}$ given $x \in \mathcal{X}$.

Given a pair of random variables $(X, Y) \in \mathcal{X} \times \mathcal{Y}$, we can define the conditional probability of the response variable being equal to y given observation x

$$\rho(y|x) = \mathbb{P}\{Y = y|X = x\}. \tag{2.1}$$

The full distribution is unknown, and can only be observed through the training set sampled independently from the same distribution ρ. While the independence assumption is very common, and will be used throughout this thesis, it is far from universal. There are some particular settings such as for example time-series, where samples must be allowed to depend on one another.

2.1.2 Loss functions

Given a data-point $(x, y) \in \mathcal{X} \times \mathcal{Y}$, to measure the quality of predictions of a model $f : \mathcal{X} \to \mathbb{R}$, we need to compare $f(x)$ with y. This is done via a loss function $\ell : \mathcal{Y} \times \mathbb{R} \to [0, \infty)$, whose formula prominently depends on the type of output space. See Chapter 5 for further discussion on classification problems. For example in continuous value regression where $\mathcal{Y} \subseteq \mathbb{R}$, one often considers the squared loss

$$\ell(y, a) = (y - a)^2 \qquad y, a \in \mathbb{R}$$

which is differentiable everywhere and strongly convex. Another alternative is the L1 loss $\ell(y, a) = |y - a|$ which is known to promote sparse solutions in linear models (Tibshirani, 1996). When $\mathcal{Y} = \{-1, 1\}$ the learning problem is called binary classification (cf. the e-mail spam example). In cases where the output space is very different from the space of real numbers, we can use a deterministic *decoding operator* $D : \mathbb{R} \to \mathcal{Y}$ to connect predictions from the real-valued model $f(x) = a \in \mathbb{R}$ to the labels $y \in \mathcal{Y}$. Putting the two together one obtains a function $c : \mathcal{X} \to \mathcal{Y}$ which is known as a *classifier*. The most natural choice of loss function for binary classification is the 0-1 loss

$$\ell(y, a) = \begin{cases} 1, & \text{if } y = D(a) \\ 0 & \text{otherwise.} \end{cases}, \qquad y \in \mathcal{Y}, a \in \mathbb{R}$$

with link function $D(a) = \operatorname{sgn}(a)$, for $a \in \mathbb{R}$. However this loss is discontinuous and non-convex, which makes it impossible to minimize using gradient-based methods (Feldman et al., 2012). Hence continuous and convex surrogates of the 0-1 loss are commonly used to approximate it while remaining computationally tractable. These surrogates treat the output space $\{-1, 1\}$ as if it were simply a subset of the real numbers. Two common examples are the logistic loss

$$\ell(y, a) = \log(1 + e^{-ya}), \qquad y \in \{-1, 1\}, a \in \mathbb{R} \tag{2.2}$$

and the hinge loss

$$\ell(y,a) = |1 - ya|_+, \qquad y \in \{-1,1\}, a \in \mathbb{R}. \tag{2.3}$$

Another useful setting is that of multiclass classification, in which case can consider $\mathcal{Y}$ to be a discrete set with as many elements as the number of classes. Once again the 0-1 loss cannot be used in practice during optimization. Denote by T the number of classes ($|\mathcal{Y}| = T$), we define continuous relaxations of the 0-1 loss by embedding the output space into $\mathbb{R}^T$ such that each $y \in \mathcal{Y}$ becomes a standard basis vector of $\mathbb{R}^T$ indicating the correct class. This embedding – known as *one-hot* – is used in combination with a multi-valued model $\hat{f} : \mathcal{X} \to \mathbb{R}^T$, and decoder $D(y) = \arg\max_{i=1,\dots,T} y_i, y \in \mathbb{R}^T$. The common losses used are once again the squared loss, or the more natural cross entropy loss

$$\ell(y,p) = -\sum_{i=1}^{T} y_i \log p_i, \qquad y,p \in \mathbb{R}^T \tag{2.4}$$

where y is the one-hot embedding of the true label, and p such that $\sum_{i=1}^T p_i = 1, p_i \geq 0$ is the model's output: a discrete probability distribution over the set of classes.

2.1.3 Risk evaluation

Given a probability distribution over the data and a loss, the ideal metric to evaluate a function f against the data is the *expected risk*

$$\mathcal{E}(f) = \int_{\mathcal{X}\times\mathcal{Y}} \ell(y, f(x))\, \mathrm{d}\rho(x,y). \tag{2.5}$$

To learn a function f, the last object which needs to be defined is the space of functions in which to look for, *i.e.* the *hypothesis space*. The widest possible space is that of all measurable functions $\mathcal{T}$. For example considering the squared loss, the widest possible space such that the expected risk is well defined is the space of squared-integrable functions $L^2(\mathcal{X}, \rho_{\mathcal{X}})$. Then the best solution to the supervised learning problem is

$$\inf_{f \in L^2_{\rho_{\mathcal{X}}}} \mathcal{E}(f). \tag{2.6}$$

Again considering the squared loss, the following well-known result (Cucker et al., 2002) allows us to express the minimizer of (2.6) in closed form:

Theorem 2.1: Cucker et al. (2002)

Assume $\mathcal{Y} \subset \mathbb{R}$ and that there exists a constant $M > 0$ such that $\rho(\mathcal{X} \times [-M, M]) = 1$ (that is, the output space is bounded), and that $\ell(x, x') = \|x - x'\|^2 \; \forall x, x' \in \mathcal{X}$. Define

$$f_\rho(x) = \int_{\mathcal{Y}} y \, \mathrm{d}\rho(y|x) \tag{2.7}$$

which belongs to $L^2_{\rho_\mathcal{X}}$. Then the following holds

$$\mathcal{E}(f) - \mathcal{E}(f_\rho) = \|f - f_\rho\|^2_{\rho_\mathcal{X}}. \tag{2.8}$$

The function f_ρ defined in (2.7) is called the *regression* function. In a similar way for binary classification, the classifier c minimizing the misclassification risk

$$\mathcal{E}(c) = \mathbb{P}\{c(X) \neq Y\}, \qquad \text{for } (X, Y) \sim \rho(x, y) \tag{2.9}$$

is $D(f_\rho(x)) = \operatorname{sgn} f_\rho(x)$ which is known as the *Bayes classifier*. The Bayes classifier is equivalent to the class with higher conditional expectation

$$c_*(x) = D(f_\rho(x)) = \operatorname*{arg\,max}_{y \in \mathcal{Y}} \rho(y|x). \tag{2.10}$$

By its definition, f_ρ is the conditional expectation of y given x: $f_\rho = \mathbb{E}[\, y \mid x \,]$. Since it is the unique minimizer of (2.6), whenever we analyze a learning algorithm we shall compare its risk to the risk of f_ρ:

Definition 2.1 (Excess risk): The excess risk of an estimator f is defined in relation with the regression function:

$$\mathcal{R}_\rho(f) \coloneqq \mathcal{E}(f) - \mathcal{E}(f_\rho) = \|f - f_\rho\|^2_{\rho_\mathcal{X}} \tag{2.11}$$

where the second equality is true when considering the square loss by Theorem 2.1.

Although for certain losses the regression function is known, in practice it cannot be computed as it depends on the unknown data distribution. Instead, we must rely on the training set $Z = \{(x_1, y_1), \dots, (x_n, y_n)\}$ which induces a finite-sample version of Equation (2.5), the *empirical risk*

$$\hat{\mathcal{E}}(f) = \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f(x_i)), \tag{2.12}$$

which can be minimized to find candidate models which are close to the regression function.

To find functions which minimize the empirical risk, searching among all possible functions (in $\mathcal{T}$) would not work since such space allows for elements which approximate the training set perfectly, but behave badly on unseen samples. This behavior - known as *overfitting* - can be prevented in two ways: restricting the size of the hypothesis space from which the model may come from (for example one may only consider linear functions), and modifying the empirical risk criterion by adding a penalty term for “complex” functions.

Empirical Risk Minimization

At its core, ERM takes the empirical risk of (2.12) and minimizes it over some space of functions $\mathcal{F}$:

$$\hat{f} = \operatorname*{arg\,min}_{f \in \mathcal{F}} \hat{\mathcal{E}}(f). \tag{2.13}$$

The final quality of $\hat{f}$ will depend on the chosen space $\mathcal{F}$: if it is very large, including all sorts of complicated functions, it is more likely to contain the true function f^* but also to perfectly match the training set along with its noise. Furthermore, finding the minimum in a larger space can be harder than in a smaller space, leading to additional error if the minimum is not achieved. On the other hand if $\mathcal{F}$ is small, the minimum will be easy to obtain, it will be unlikely to fit the training-set noise, but if the space does not contain the true function the minimizer may still be far from f^*. In ERM balancing between underfitting and overfitting is controlled by the size of the hypothesis space. In practice however, without knowing which space the target function belongs to, one may want to make the search space larger and control overfitting in other ways.

Regularization

Regularization alters the ERM objective by adding a penalty or regularizer. Let $\mathcal{F}$ be a normed space; given a function $\mathcal{F}$ $f \in \mathcal{F}$, *Tikhonov regularization* introduces the following penalty:

$$\hat{f}_\lambda = \operatorname*{arg\,min}_{f \in \mathcal{F}} \hat{\mathcal{E}}(f) + \lambda \|f\|^2_{\mathcal{F}}. \tag{2.14}$$

The hypothesis space $\mathcal{F}$ can be picked to be large (*e.g.* all smooth functions), since the parameter $\lambda \geq 0$ allows to choose the right trade-off between under- and over-fitting in a fine grained manner. Essentially the choice of the correct function space has been reduced to the tuning of a real-valued parameter. While doing so correctly can be hard, there are several possible approaches which are looked into in detail in Chapter 4.

2.2 Reproducing Kernel Hilbert Spaces

In this thesis we will assume $\mathcal{H}$ to be a Hilbert space. One of the simplest Hilbert spaces is that of linear functions. Taking $\mathcal{X} \subseteq \mathbb{R}^d$,

$$\mathcal{H} = \{f : \mathcal{X} \to \mathbb{R} | f(x) = \langle w, x\rangle, w \in \mathbb{R}^d\}. \tag{2.15}$$

However, using this space will not allow a learning algorithm to fit anything but linear relationships between inputs and outputs; this can often be too restrictive in real-world problems. For example, imagine trying to estimate people's height from their age: such relationship would grow quickly at first, and then taper out since 50 year olds will not be higher than 40 year olds. Therefore we need to introduce a larger Hilbert space which allows to represent non-linear functions. In particular we introduce reproducing kernel Hilbert spaces:

Definition 2.2 (Reproducing Kernel Hilbert Space): A space $\mathcal{H}$ is called a reproducing kernel Hilbert space (RKHS) if it satisfies the following properties:

1. $\mathcal{H}$ is a Hilbert space of functions $f : \mathcal{X} \to \mathbb{R}$ with inner product $\langle \cdot, \cdot \rangle_{\mathcal{H}}$.
2. The reproducing property holds: for every $x \in \mathcal{X}$ there exists a function $k_x \in \mathcal{H}$ such that for every $f \in \mathcal{H}$
$$f(x) = \langle f, k_x\rangle_{\mathcal{H}}. \tag{2.16}$$

The reproducing property implies other important properties regarding the evaluation map $e_x : f \mapsto f(x), f \in \mathcal{H}, x \in \mathcal{X}$:

- For every $x \in \mathcal{X}$, the evaluation map is bounded: there exists a constant c_x such that $|e_x(f)| = |f(x)| \leq c_x \|f\|_{\mathcal{H}}$.
- For every $x \in \mathcal{X}$, the evaluation map is continuous.

Since k_x is itself a function, it can be evaluated at another point $x' \in \mathcal{X}$ to get

$$k_x(x') = \langle k_x, k_{x'}\rangle_{\mathcal{H}} =: k(x, x'). \tag{2.17}$$

The two-variate function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ thus defined is known as the reproducing kernel (or just kernel) for $\mathcal{H}$. From the definition it also holds that k is symmetric

$$k(x, x') = \langle k_x, k_{x'}\rangle = k(x', x) \tag{2.18}$$

and positive semi-definite

$$\sum_{i,j=1}^{n} a_i a_j k(x_i, x_j) = \sum_{i=1}^{n} a_i \left\langle k_{x_i}, \sum_{j=1}^{n} a_j k_{x_j} \right\rangle_{\mathcal{H}} = \left\langle \sum_{i=1} a_i k_{x_i}, \sum_{j=1} a_j k_{x_j} \right\rangle_{\mathcal{H}} = \|\sum_{i=1}^{n} a_i k_{x_i}\|_{\mathcal{H}}^2 \tag{2.19}$$

for all $n \in \mathbb{N}$, $a_1, \ldots, a_n \in \mathbb{R}$ and $x_1, \ldots, x_n \in \mathcal{X}$.

The Moore-Aronszajin theorem establishes the other side of the connection between RKHSs and reproducing kernels, allowing to go from a kernel function to its space.

Theorem 2.2: Moore-Aronszajin

Let $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a symmetric, positive definite function. Then there exists a unique RKHS $\mathcal{H}$ such that for every $x \in \mathcal{H}$ $k_x := k(x, \cdot) \in \mathcal{H}$, and for every $f \in \mathcal{H}, x \in \mathcal{X}$ it holds that $f(x) = \langle f, k_x \rangle_{\mathcal{H}}$.

Having introduced the main space of non-linear functions which will be considered in this thesis, we provide a connection to the linear functions space introduced as an example at the beginning of this section: *feature maps*.

Definition 2.3 (Feature map): A feature map for reproducing kernel k is a function $\phi : \mathcal{X} \to \mathcal{W}$ with $\mathcal{W}$ a Hilbert space such that, for every $x, x' \in \mathcal{X}$

$$k(x, x') = \langle \phi(x), \phi(x') \rangle_{\mathcal{W}}. \tag{2.20}$$

We can see how every function k which can be represented as (2.20) is a kernel: as an inner product it is symmetric and positive definite, then by Theorem 2.2 it is a reproducing kernel. Conversely, every reproducing kernel k defines a feature map, by simply choosing $\mathcal{W} = \mathcal{H}$ and letting $\phi(x) = k_x$ for any $x \in \mathcal{X}$. The continuity of ϕ follows from the continuity of k since for any $x, x' \in \mathcal{X}$

$$\|\phi(x) - \phi(x')\|^2_{\mathcal{W}} = k(x, x) + k(x', x') - 2k(x, x'). \tag{2.21}$$

Moreover the feature map associated with a given kernel is not unique (Minh et al., 2006).

Intuitively feature maps can be seen as embedding points from the input space $\mathcal{X}$, into a space with a different number of dimensions - which can even be infinite. We will see that this mapping allows to reuse many tools from linear modeling for learning non-linear functions. As an example of how this can work, imagine estimating a 1-dimensional function $y = 0.5x^2 - 1.0$ from noisy data depicted in Figure 2.1(a), using only linear functions of the form $f(x) = \langle w, x \rangle, w \in \mathbb{R}$. The quality of fit will be necessarily poor (see Figure 2.1(b)) as the model class is too simple for the task, *i.e.* all possible models will underfit the data. However by replacing every data-point x with a vector $\phi(x) = [x^2, x, 1] \in \mathbb{R}^3$, a linear function of $\phi(x)$ can be learned to fit the data precisely as $f(x) = \langle w, \phi(x) \rangle, w \in \mathbb{R}^3$. Quadratic functions in $\mathbb{R}$ can be seen as linear functions in $\mathbb{R}^3$ using feature map ϕ.

Since we have seen how feature maps induce reproducing kernels, the latter can be interpreted as an implicit mapping of the input data into a higher – possibly infinite – dimensional space through ϕ. Crucially for the case of infinite dimensional spaces one cannot compute $\phi(x) \in \mathcal{H}$, but the kernel of (2.20) belongs to the space of real numbers and can be computed explicitly.

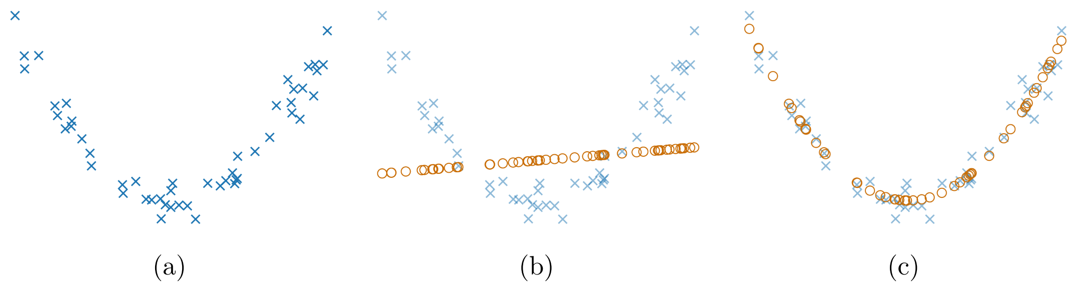


Figure 2.1 Regression on a 1D dataset generated as a parabola, using either linear (b) or quadratic (c) functions.

Designing an appropriate kernel function for a given dataset can be crucial, especially when the data has a well-defined structure. For example there exist kernels on graphs (Borgwardt et al., 2005; Camps-Valls et al., 2006), text (Lodhi et al., 2002; Herbrich, 2001), histograms (F. Li, Ionescu, et al., 2010), images (Cuturi et al., 2006; Barla et al., 2003), *etc.* note that the sampling of references is far from complete. However kernels are frequently applied to tabular datasets, for which a tailored function may not be easily defined. For this use-case, where we assume that $\mathcal{X} \subseteq \mathbb{R}^d$ with d the dimension of the input space, there are some common kernels which can be easily applied to obtain good results:

- The linear kernel
$$k(x, x') = x^\top x'. \tag{2.22}$$
- The polynomial kernel
$$k(x, x') = (1 + x^\top x')^b \qquad b \in \mathbb{N}. \tag{2.23}$$
- The Gaussian kernel
$$k(x, x') = \exp\left(-\frac{\gamma^2}{2}\|x - x'\|^2\right) \qquad \gamma > 0, \tag{2.24}$$
  which is infinitely differentiable and translation invariant ($k(x, x') = k(x + x_0, x' + x_0)$ for all $x_0 \in \mathcal{X}$).
- The Laplacian kernel
$$k(x, x') = \exp(-\gamma\|x - x'\|) \qquad \gamma > 0. \tag{2.25}$$

Some of these options can be thought of as families of kernels, induced by parameters such as b and γ. In particular parameter γ in the Gaussian and Laplacian kernels is generally known as the *bandwidth*, and it controls the distance at which nearby input points can affect each other.

2.3 Kernel Ridge Regression

We now go back to the supervised learning problem, and study an instance of learning in which we make the following three modeling choices:

1. The hypothesis space is a RKHS $\mathcal{H}$, with reproducing kernel k;
2. The loss function is the squared loss $\ell(y, y') = (y - y')^2$;
3. Tikhonov regularization is used to prevent overfitting.

The resulting algorithm, whose explicit solution is given in Equation (2.33) is called kernel ridge regression, or KRR.

Given a training set $Z = \{(x_1, y_1), \dots, (x_n, y_n)\}$ the minimization problem can be written out as

$$\min_{f \in \mathcal{H}} \hat{\mathcal{E}}_\lambda, \quad \hat{\mathcal{E}}_\lambda(f) = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - y_i)^2 + \lambda \|f\|_{\mathcal{H}}^2, \quad \lambda > 0 \tag{2.26}$$

The function $\hat{\mathcal{E}}_\lambda : \mathcal{H} \to \mathbb{R} \cup \{+\infty\}$ is strictly convex thanks to the regularizer, it is proper since $\hat{\mathcal{E}}_\lambda(f) < +\infty$ for at least one $f \in \mathcal{H}$ (and in fact every $f \in \mathcal{H}$ is finite), coercive and continuous. Then the existence and uniqueness of a solution are guaranteed (Boyd et al., 2004).

A hint towards the particular form of the solution is given by the *representer theorem*, which is stated below for a slightly more general setting than the one we will be considering.

Theorem 2.3: Representer theorem (Schölkopf, Herbrich, et al. (2001))

Assuming a strictly monotonic increasing regularizer function Ω and an arbitrary loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$, then each minimizer $f \in \mathcal{H}$ of the regularized empirical risk

$$\frac{1}{n}\sum_{i=1}^{n} \ell(f(x_i), y_i) + \Omega(\|f\|_{\mathcal{H}}) \tag{2.27}$$

can be represented in the form

$$\hat{f}_\lambda(x) = \sum_{i=1}^{n} \alpha_i k(x_i, x) \tag{2.28}$$

for some $\alpha_i \in \mathbb{R}$.

Hence we can represent the unique solution to our problem, which we will denote as $\hat{f}_\lambda \in \mathcal{H}$ to emphasize the dependence on the regularization parameter λ, as a linear combination of the kernel computed on the n training points $\{x_i\}_{i=1}^n$.

Definition 2.4 (Kernel matrix): Given a dataset with n points $\{(x_1, y_1), \dots, (x_n, y_n)\}$ with $x \in \mathcal{X}, y \in \mathcal{Y}$ and a reproducing kernel k, denote the $n \times n$ matrix whose i, j-th entry is given by $k(x_i, x_j)$ as $\boldsymbol{K}$. Such matrix will be referred to as the kernel matrix.

Provided with the definition of the kernel matrix we can write the minimization problem in a more explicit way. First note that from Theorem 2.3,

$$\left\| \hat{f}_\lambda \right\|_{\mathcal{H}}^2 = \left\| \sum_{i=1}^n \alpha_i k(x_i, \cdot) \right\|_{\mathcal{H}}^2 = \sum_{i,j=1}^n \alpha_i \alpha_j k(x_i, x_j) = \langle \alpha, \boldsymbol{K}\alpha \rangle_{\mathbb{R}^n}. \tag{2.29}$$

Then we can rewrite (2.26) as a minimization not over $f \in \mathcal{X}$ but over the *parameter vector* $\alpha = [\alpha_1, \dots, \alpha_n]^\top \in \mathbb{R}^n$:

$$\hat{\alpha}_\lambda = \arg\min_{\alpha \in \mathbb{R}^n} \frac{1}{n} \|\boldsymbol{K}\alpha - \hat{y}\|_{\mathbb{R}^n}^2 + \lambda \langle \alpha, \boldsymbol{K}\alpha \rangle_{\mathbb{R}^n} \tag{2.30}$$

where we denote by $\hat{y}$ the vector of all labels $\hat{y} = [y_1, \dots, y_n]^\top$.

Solving the problem of (2.30) is simple: compute the gradient with respect to α and set it to zero to get

$$\boldsymbol{K}^2\alpha - \boldsymbol{K}\hat{y} + n\lambda\boldsymbol{K}\alpha = 0,$$

which always admits the solution of

$$(\boldsymbol{K} + n\lambda\boldsymbol{I})\alpha = \hat{y}. \tag{2.31}$$

This is a $n \times n$ linear system which can also be written explicitly for α as

$$\hat{\alpha}_\lambda = (\boldsymbol{K} + n\lambda\boldsymbol{I})^{-1}\hat{y}. \tag{2.32}$$

Hence the direct solution to the regularized ERM problem of Equation (2.26) is given by

$$\boxed{\hat{f}_\lambda(x) = \sum_{i=1}^n (\hat{\alpha}_\lambda)_i k(x_i, x), \quad \hat{\alpha}_\lambda = (\boldsymbol{K} + n\lambda\boldsymbol{I})^{-1}\hat{y}.} \tag{2.33}$$

Note that in (2.32) the invertibility of $\boldsymbol{K} + n\lambda\boldsymbol{I}$ is guaranteed by the positive semi-definiteness of $\boldsymbol{K}$ and $\lambda > 0$, so the regularizer improves the solution's stability.

Spectral filtering A useful way of analyzing the improvement to stability provided by regularization, is to look at the spectrum of the kernel matrix without regularization. $\boldsymbol{K}$ can be decomposed as $\boldsymbol{K} = Q\Sigma Q^\top$ with $\Sigma = \text{diag}(\lambda_1, \dots, \lambda_n)$ a diagonal matrix of decreasing eigenvalues ($\lambda_1 \geq \lambda_2 \geq \dots \geq 0$), and Q an orthogonal matrix containing the corresponding eigenvectors $q_1, \dots, q_n$. Then the least squares estimator (with $\lambda = 0$) is

$$\hat{\alpha}_0 = \boldsymbol{K}^{\dagger}\hat{y} = Q\Sigma^{dagger}Q^{\top}\hat{y} = \sum_{\lambda_i \neq 0} \frac{1}{\lambda_i}\langle q_i, \hat{y}\rangle q_i, \tag{2.34}$$

where $A^{\dagger}$ denotes the pseudo-inverse of matrix A. The estimator $\hat{\alpha}_0$ is *ill-conditioned*: tiny perturbations in the data, in correspondence of the small eigenvalues, have an outsized influence on the result. With Tikhonov regularization, Equation (2.34) becomes

$$\hat{\alpha}_{\lambda} = (\boldsymbol{K} + n\lambda\boldsymbol{I})^{-1}\hat{y} = \sum_{i=1}^{n} \frac{1}{\lambda_i + n\lambda}\langle q_i, \hat{y}\rangle q_i \tag{2.35}$$

such that the influence of eigenvalues which are smaller than $n\lambda$ gets essentially filtered out (for $\lambda_i \ll n\lambda$ then $\frac{1}{\lambda_i+n\lambda} \sim \frac{1}{n\lambda}$), while larger eigenvalues are unaffected (for $\lambda_i \gg n\lambda$ then $\frac{1}{\lambda_i+n\lambda} \sim \frac{1}{\lambda_i}$). The point of view of *spectral filtering* is that many different regularization methods can be expressed by defining a suitable filter function of the kernel $G_{\lambda} : \mathbb{R} \to \mathbb{R}$, which acts on its spectrum. Using the eigendecomposition of $\boldsymbol{K}$ then

$$G_{\lambda}(\boldsymbol{K}) = \sum_{i=1}^{n} q_i G_{\lambda}(\lambda_i) q_i^{\top}. \tag{2.36}$$

For Tikhonov regularization we have seen the following filter function

$$G_{\lambda}(\lambda_i) = \frac{1}{\lambda_i + n\lambda}, \tag{2.37}$$

where λ controls the magnitude of the minimum eigenvalues of $G_{\lambda}(\boldsymbol{K})$. In Section 2.4.1 we will see another spectral filter which works on a very different class of algorithms.

Condition number By ensuring that no eigenvalues of the regularized kernel matrix $\boldsymbol{K}_{\lambda} \coloneqq \boldsymbol{K} + n\lambda\boldsymbol{I}$ are lower than a threshold $n\lambda$, small changes in the kernel which inevitably occur due to the noisy empirical data will not alter the solution excessively. A quantitative formulation of this principle is through the condition number of $\boldsymbol{K}_{\lambda}$.

Definition 2.5 (Condition number): The condition number of a symmetric matrix A is

$$\kappa(A) = \frac{|\lambda_{\max}(A)|}{|\lambda_{\min}(A)|} \tag{2.38}$$

where $\lambda_{\max}(A)$ and $\lambda_{\min}(A)$ denote the largest and smallest eigenvalues of A respectively.

A lower condition number implies a lower sensitivity of the linear system solution to small changes in the input. As we will see later, the condition number also has an impact on the time complexity for iterative solutions to the linear system. The following lemma shows how regularization lowers the condition number of the linear system needed for solving KRR.

Lemma 2.1 (Condition number of KRR): Tikhonov regularization improves the condition number:

$$\kappa(\boldsymbol{K} + n\lambda\boldsymbol{I}) \leq \kappa(\boldsymbol{K}) \tag{2.39}$$

for $\lambda > 0$.

Proof of Lemma 2.1: For conciseness let $\lambda_1 := \lambda_{\max}(\boldsymbol{K})$, $\lambda_n := \lambda_{\min}(\boldsymbol{K})$ and $\gamma := n\lambda$. The condition number of the regularized matrix is

$$\kappa(\boldsymbol{K} + n\lambda\boldsymbol{I}) = \frac{\lambda_1 + \gamma}{\lambda_n + \gamma}.$$

Knowing that $\lambda_1 \geq \lambda_n$, $\lambda_1 \geq 0$ and $\gamma > 0$

$$\frac{\lambda_1 + \gamma}{\lambda_n + \gamma} - \frac{\lambda_1}{\lambda_n} = \frac{\lambda_1\lambda_n + \gamma\lambda_n - \lambda_1\lambda_n - \lambda_1\gamma}{\lambda_n(\lambda_n + \lambda_1)} = \frac{\gamma(\lambda_n - \lambda_1)}{\lambda_n(\lambda_n + \lambda_1)} \leq 0$$

where equality is attained only if all eigenvalues of $\boldsymbol{K}$ are equal, in which case the condition number is 1 anyways. $\square$

2.3.1 Statistical considerations

From a statistical point of view the properties of the estimator $\hat{f}_\lambda$ are well studied, see *e.g.* Steinwart, Hush, et al. (2009), Caponnetto et al. (2007), and Shalev-Shwartz et al. (2014). In particular we are interested in the expected risk $\mathcal{E}(\hat{f}_\lambda)$ compared to that of the regression function f_ρ. However, since the hypothesis space $\mathcal{H}$ is not dense in $L^2(\mathcal{X}, \rho_{\mathcal{X}})$, approaching f_ρ will not be possible in general. We therefore have to consider a different version of the excess risk (see Equation (2.11)) $f_{\mathcal{H}}$, which is computed as $f_{\mathcal{H}} = inf_{f\in\mathcal{H}}\mathcal{E}(f)$. Following Caponnetto et al. (2007), the following assumptions need to hold in order to prove the basic statistical bounds comparing the risks of $\hat{f}_\lambda$ and $f_{\mathcal{H}}$.

The first assumption guarantees that the reproducing kernel is bounded

Assumption 2.1. *A constant $\kappa \geq 1$ exists such that $k(x, x) \leq \kappa^2$ for any $x \in \mathcal{X}$.*

Another basic assumption, whose need is clearly necessary from the introduction of this section, guarantees the existence of $f_{\mathcal{H}}$

Assumption 2.2. *Consider RKHS $\mathcal{H}$ with kernel k, we assume there exists $f_{\mathcal{H}} \in \mathcal{H}$ such that*

$$\mathcal{E}(f_{\mathcal{H}}) := \inf_{f\in\mathcal{H}} \mathcal{E}(f). \tag{2.40}$$

We will denote by $\mathcal{R}_{\mathcal{H}}(f) = \mathcal{E}(f) - \mathcal{E}(f_{\mathcal{H}})$ the excess risk of an estimator f with respect to $f_{\mathcal{H}}$.

The last assumption is on the data distribution ρ, and in particular on the conditional probability $\rho(y|x)$

Assumption 2.3. *Constants σ, b such that $0 \leq \sigma \leq b$ exist such that, for any $x \in \mathcal{X}$ the following holds*

$$\int_{\mathcal{Y}} |y - f_{\mathcal{H}}(x)|^p \, \mathrm{d}\rho(y|x) \leq \frac{1}{2} p! \sigma^2 b^{p-2}, \quad \forall p \geq 2. \tag{2.41}$$

This assumption holds for example when y is bounded, sub-Gaussian or sub-exponential. Then, the following proposition holds

Theorem 2.4: from Caponnetto et al. (2007)

Let $\hat{f}_\lambda$ be the KRR estimator of (2.33), and $\delta \in (0, 1]$. Under Assumptions 2.1 to 2.3 outlined above, the following holds with probability at least $1 - \delta$

$$\mathcal{R}_{\mathcal{H}}(\hat{f}_\lambda) \lesssim \lambda + \frac{1}{n\lambda} \log \frac{1}{\delta}, \tag{2.42}$$

where the symbol $\lesssim$ signifies that we ignored constants not depending on n, λ or σ.

From this theorem we can derive the regularizer λ which allows to minimize the risk

Corollary 2.3.1. *Choosing λ_n such that*

$$\lambda_n = \frac{1}{\sqrt{n}},$$

with probability at least $1 - \delta$ the following holds

$$\mathcal{R}_{\mathcal{H}}(\hat{f}_\lambda) \lesssim \frac{1}{\sqrt{n}} \log \frac{1}{\delta}. \tag{2.43}$$

Under additional regularity assumptions this bound can be refined in order to show a dependency on the intrinsic difficulty of the problem. For more details, see Caponnetto et al. (2007) and Steinwart, Hush, et al. (2009).

2.3.2 Computational considerations

Once again we assume to have a training set of n points, and the associated kernel matrix $\boldsymbol{K} \in \mathbb{R}^{n \times n}$. In this section we look at different ways in which one can solve the minimization of (2.30) to get the parameter estimate $\hat{\alpha}_\lambda$, and compare them from the point of view of computational efficiency.

The first possibility is to solve the linear system directly. To do so, one must first calculate and store every entry of $\boldsymbol{K}$. Since there are n^2 entries, assuming the computational cost for one entry is d, computing $\boldsymbol{K}$ requires $\mathcal{O}(dn^2)$ time units and $\mathcal{O}(n^2)$ space units. The cost for a single kernel entry will depend on the choice of kernel, but for the common options of the linear (2.22), polynomial (2.23) or Gaussian (2.24) kernels, it will depend on the dimensionality of $x \in \mathcal{X} \subset \mathbb{R}^d$. Once $\boldsymbol{K}$ has been computed, a linear system solver comes into play which will typically use the Cholesky (which has an asymptotic cost of $\mathcal{O}(1/3n^3)$ operations) or LU (with an asymptotic cost of $\mathcal{O}(2/3n^3)$ operations) decompositions instead of inverting $\boldsymbol{K}$ directly which is slower and less numerically stable. Since this operation has higher time complexity than the kernel computation, the final cost of the direct solution is

$$\mathcal{O}(n^3) \text{ time} \qquad \mathcal{O}(n^2) \text{ space}. \tag{2.44}$$

An alternative to direct inversion is to solve (2.30) using iterative optimization methods, which are the focus of the next section.

2.4 Iterative Optimization

In this section we will provide some background on algorithms which solve a minimization problem by iteratively computing better and better solutions. The first algorithm is gradient descent (GD) which can be applied to all smooth problems (for which a gradient exists). Then we will present the conjugate gradient (CG) method, which is more specialized and can be applied to the solution of symmetric and positive definite linear systems.

For simplicity, we will take as an example the problem of finding x such that

$$Ax = b, \quad \text{with } A^\top = A, A \succ 0. \tag{2.45}$$

for a matrix $A \in \mathbb{R}^{n\times n}$ and $b \in \mathbb{R}^n$. Starting from an estimate of the solution x_0, at each iteration we will compute a new candidate x_{i+1} of the form

$$x_{i+1} = x_i + \alpha_i d_i \tag{2.46}$$

where α_i is a scalar and d_i a vector of the same size as x. Denote the error at iteration i by $e_i = x_i - A^{-1}b$, and the residual by $r_i = b - Ax_i = -Ae_i$.

We can rewrite the problem of Equation (2.45) as an equivalent quadratic problem:

$$\arg\min_x f(x) = \arg\min_x \left(\frac{1}{2}x^\top Ax - b^\top x\right). \tag{2.47}$$

2.4.1 Gradient Descent for KRR

Gradient descent can solve the linear system of Equation (2.45) by following the gradient along the parabola defined in Equation (2.47). The GD algorithm is not limited to such highly structured problems, as it is well-defined for minimizing any smooth function.

Starting from an arbitrary point x_0, which can be taken equal to 0 for simplicity, we will take a series of steps $x_1, \dots, x_t$ by moving along the gradient of f, until some convergence criterion is reached (*e.g.* the last two points are very close to each other).

The gradient of f is

$$f'(x) = Ax - b, \tag{2.48}$$

hence our update steps (in the direction of steepest descent, so against the gradient) should be of the form

$$x_{i+1} = x_i + \gamma_i (Ax_i - b). \tag{2.49}$$

Note that we never need to invert A, and in particular we don't even need to know A itself: it is sufficient to be able to multiply the matrix A with an arbitrary vector. Since the problem is convex, the GD iterations are known to converge to the unique minimizer. However, the speed and accuracy of this convergence crucially depend on the step size γ. If γ is too small then a huge number of iterations will be needed; if it is too large then the algorithm may oscillate around the minimum without every being able to reach it precisely. However, when A is positive definite, the optimal step size can be determined to be

$$\gamma_i = \frac{r_i^\top r_i}{r_i^\top A r_i}. \tag{2.50}$$

Take κ to be the condition number of A:

$$\kappa = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)}, \tag{2.51}$$

and denote the A-norm of a vector v: $\|v\|_A = v^\top A v$. Then it can be proven (Boyd et al., 2004) that the error of GD for this problem decreases with each step as

$$\|e_i\|_A \leq \left(\frac{\kappa - 1}{\kappa + 1}\right)^i \|e_0\|_A. \tag{2.52}$$

Example 2.1: The empirical risk minimization problem of the previous section

$$\min_{f \in \mathcal{H}} \hat{\mathcal{E}}(f), \qquad \hat{\mathcal{E}}(f) = \frac{1}{2n} \sum_{i=1}^{n} (f(x_i) - y_i)^2 \tag{2.53}$$

can be expressed as the minimization of a finite dimensional quadratic problem through the representer theorem

$$\min_{\alpha \in \mathbb{R}^n} \hat{\mathcal{E}}(\alpha), \qquad \hat{\mathcal{E}}(\alpha) = \frac{1}{2n} \|\boldsymbol{K}\alpha - \hat{y}\|^2. \tag{2.54}$$

Note that, since $\boldsymbol{K}$ is positive semi-definite, Equation (2.54) defines an equivalent problem to Equation (2.47). The gradient of $\hat{\mathcal{E}}(\alpha)$ is

$$\nabla\hat{\mathcal{E}}(\alpha) = \frac{1}{n}(\boldsymbol{K}\alpha - \hat{y}) = \frac{1}{n}(\boldsymbol{K}\alpha - \hat{y}) \tag{2.55}$$

and the gradient descent (GD) algorithm gives the following iteration for solving (2.54)

$$\alpha_i = \alpha_{i-1} - \frac{\gamma}{n}(\boldsymbol{K}\alpha_{i-1} - \hat{y}), \tag{2.56}$$

with step size $\gamma > 0$, and $\alpha_0 = 0$. Computationally, the GD algorithm still requires constructing the full kernel matrix. Each iteration (see (2.56)) requires multiplying the kernel matrix with a vector, an operation which has a $\mathcal{O}(n^2)$ asymptotic cost. Similarly, determining γ_i requires a kernel-vector multiplication per Equation (2.50). Assuming we wish to find a solution with a certain accuracy $\epsilon > 0$ (*i.e.* $\|e_i\| \leq \epsilon\|e_0\|$)). From Equation (2.50) we can determine the maximum number of iterations t needed to achieve that accuracy:

$$t = \left\lceil \frac{1}{2}\kappa \ln\frac{1}{\epsilon} \right\rceil \tag{2.57}$$

with κ the condition number of $\boldsymbol{K}$. The final asymptotic complexity of the GD algorithm for KRR is

$$\mathcal{O}(tn^2 + dn^2) \text{ time} \qquad \mathcal{O}(n^2) \text{ space.} \tag{2.58}$$

For large datasets, where $n \gg t$, this is an improvement over the direct solution. In practice the bottleneck becomes the space complexity: for a large but not huge dataset of 500 000 points, the kernel matrix occupies 931 GB of memory!

2.4.2 Implicit regularization

In Section 2.3, we have remarked that the unregularized regression can be ill-conditioned and lead to overfitting. We have thus introduced a Tikhonov regularizer which penalizes functions f with large norm, and have seen how the regularizer also stabilizes the direct solution of a linear system to obtain $\hat{\alpha}_\lambda$. Here we shall see how it is possible to introduce a similar regularization *implicitly* through the optimization algorithm. Intuitively we don't care about reaching the true empirical risk minimizer: we are interested in minimizing the expected risk, and wish to avoid overfitting to the noise in the empirical data. Implicit regularization studies how *early stopping* of the GD iterations regularizes the final solution similarly to Tikhonov regularization. By induction, one can show that the iteration of (2.56), with $\alpha_0 = 0$ is equivalent to

$$\alpha_t = \gamma\sum_{i=0}^{t-1}(\boldsymbol{I} - \gamma\boldsymbol{K})^i\hat{y} \tag{2.59}$$

where we have rescaled the step-size for the sake of brevity. Now recall main property of the Neumann series: supposing A a bounded linear operator, denoting by $\|\cdot\|$ the operator norm for such operator, if $\|A\| < 1$ then

$$\sum_{i=0}^{\infty} (\boldsymbol{I} - A)^i = A^{-1}. \tag{2.60}$$

Importantly, if we consider a truncated version of (2.60) we obtain an approximation of the matrix inverse

$$\sum_{i=0}^{t} (\boldsymbol{I} - A)^i \approx A^{-1}. \tag{2.61}$$

We can view the GD iteration as a spectral filter on the matrix $\boldsymbol{K}$, which now depends on t (the number of iterations) instead of λ. Using spectral calculus we have that

$$\alpha_t = G_t(\boldsymbol{K})\hat{y}, \quad \text{with} \quad G_t(\lambda_i) = \gamma \sum_{j=0}^{t-1} (1 - \gamma\lambda_i)^j. \tag{2.62}$$

and the geometric series

$$G_t(\lambda_i) = \frac{1 - (1 - \gamma\lambda_i)^t}{\lambda_i} \tag{2.63}$$

converges to $1/\lambda_i$ as t goes to ∞. In fact, it can be formally shown that the behavior of the regularization parameter t behaves as $1/\lambda$ at its asymptotes (although a slightly different regularization behavior occurs elsewhere (Yao et al., 2007)).

2.4.3 Conjugate gradient method

The conjugate gradient method (CG) (Hestenes et al., 1952; Shewchuk, 1994) is a well-known efficient iterative algorithm for solving symmetric and positive definite linear systems of the form of Equation (2.45). For the following we must define A-orthogonal or *conjugate* vectors d_i, d_j if $d_i^\top A d_j = 0$.

The conjugate gradient method puts the constraint that once a direction d_i has been used, it should not be used again: the directions must be conjugate to each other. Furthermore the error term at iteration i must be conjugate to the directions used at previous iterations: $d_i^\top A e_{i+1} = 0$. From this we can obtain an explicit form for the coefficients α_i:

$$d_i^\top A_i e_{i+1} = d_i^\top A(e_i + \alpha_i d_i) = 0 \implies \alpha_i = -\frac{d_i^\top A e_i}{d_i^\top A d_i} = \frac{d_i^\top r_i}{d_i^\top A d_i}. \tag{2.64}$$

The following lemmas can be found *e.g.* in Shewchuk (1994).

Proposition 2.4.1

A procedure with α_i defined as in Equation (2.64) and conjugate directions converges to the true value x in n steps.

Proof of Proposition 2.4.1: The error e_0 can be expressed as a linear combination of n directions

$$e_0 = \sum_{i=0}^{n-1} \delta_i d_i.$$

Multiplying both sides by $d_j^\top A$

$$\begin{aligned} d_j^\top A e_0 &= \sum_{i=0}^{n-1} \delta_i d_j^\top A d_i = \delta_j d_j^\top A d_j \\ \delta_j &= \frac{d_j^\top A e_0}{d_j^\top A d_j^\top}. \end{aligned}$$

Note that by the definitions of e_i and x_i we have that

$$e_k = e_0 + \sum_{i=0}^{k-1} \alpha_i d_i,$$

then, taking in mind A-orthogonality of directions

$$\delta_j = \frac{d_j^\top A(e_0 + \sum_{i=0}^{j-1} \alpha_i d_i)}{d_j^\top A d_j^\top} = \frac{d_j^\top A e_j}{d_j^\top A d_j^\top}$$

which is exactly equal to $-\alpha_j$ in Equation (2.64). Hence while we are building up a solution x, we are also decomposing the error term down to zero after n iterations:

$$e_k = e_0 + \sum_{i=0}^{k-1} \alpha_i d_i = \sum_{i=0}^{n-1} \delta_i d_i - \sum_{i=0}^{k-1} \delta_i d_i = \sum_{i=k}^{n-1} \delta_i d_i.$$

□

Finally we must describe how to obtain the conjugate directions. The starting point is taken to be $d_0 = b - Ax_0 = r_0$, and further directions are derived by Gram-Schmidt orthonormalization. For this to work the residuals must form a linearly independent basis, which can be seen by noting that directions and residuals are orthogonal

$$d_j^\top r_i = -d_j^\top A e_i = 0 \quad \text{for } j < i, \tag{2.65}$$

and hence by searching for new directions in the space spanned by the previous residuals we have that every residual is orthogonal to all previous residuals ($r_i^\top r_j = 0, i \neq j$). The construction of new directions follows $d_0 = r_0$,

$$d_i = r_i + \sum_{k=0}^{i-1} \beta_{ik} d_k. \tag{2.66}$$

We can then derive the coefficients by multiplying by Ad_j and using A-orthogonality:

$$d_i^\top Ad_j = r_i^\top Ad_j + \sum_{k=0}^{i-1} \beta_{ik} d_k^\top Ad_j \implies \beta_{ij} = -\frac{r_i^\top Ad_j}{d_j^\top Ad_j} \tag{2.67}$$

where β_{ij} is only defined for $i > j$. Equation (2.67) seems to imply that computing a new direction depends on all the previous ones. We will now see how several terms simplify, and knowledge of previous directions will not be needed. Consider $r_{j+1} = r_j - \alpha_j Ad_j$, then $r_i^\top r_{j+1} = r_i^\top r_k - \alpha_j r_i^\top Ad_j$. We can thus write

$$r_i^\top Ad_j = \frac{1}{\alpha_j}(r_i^\top r_j - r_i^\top r_{j+1}) = \begin{cases} \frac{1}{\alpha_i} r_i^\top r_i, & \text{for } i = j \\ -\frac{1}{\alpha_{i-1}} r_i^\top r_i, & \text{for } i = j+1 \\ 0 & \text{otherwise} \end{cases}$$

Then we can write the direction coefficients as

$$\beta_{ij} = -\frac{r_i^\top Ad_j}{d_j^\top Ad_j} = \begin{cases} \frac{1}{\alpha_{i-1}} \frac{r_i^\top r_i}{d_{i-1}^\top Ad_{i-1}}, & \text{for } i = j+1 \\ 0, & \text{for } i > j+1 \end{cases}$$

where the dependence is only on the last direction. We can simplify even further, denoting $\beta_i = \beta_{i,i-1}$. We first need a simple proposition

Proposition 2.4.2

The following holds

$$d_i^\top r_i = r_i^\top r_i \tag{2.68}$$

Proof of Proposition 2.4.2: Consider the Gram-Schmidt procedure of Equation (2.66), and take the inner product with residual r_j

$$d_i^\top r_j = r_i^\top r_j + \sum_{k=0}^{i-1} \beta_{ik} d_k^\top r_j.$$

Taking $j = i$, the second term becomes zero by Equation (2.65) and we have

$$d_i^\top r_i = r_i^\top r_j.$$

□

Then

$$\beta_i = \frac{d_{i-1}^\top A d_{i-1}}{d_{i-1}^\top r_{i-1}} \frac{r_i^\top r_i}{d_{i-1}^\top A d_{i-1}} = \frac{r_i^\top r_i}{r_{i-1}^\top r_{i-1}}. \tag{2.69}$$

Algorithm 1 CG algorithm.

```
1:  function CG($A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$)
2:      $d_0 \leftarrow b - Ax_0$
3:      $r_0 \leftarrow d_0$
4:      for $i = 0, \dots, n-1$ do
5:          $\alpha_i \leftarrow \frac{r_i^\top r_i}{d_i^\top A d_i}$                ▷ by Equations (2.64) and (2.68)
6:          $x_{i+1} \leftarrow x_i + \alpha_i d_i$
7:          $r_{i+1} \leftarrow r_i - \alpha_i A d_i$
8:          if $\|r_{i+1}\| < \epsilon$ then
9:              exit loop
10:         end if
11:         $\beta_{i+1} \leftarrow \frac{r_{i+1}^\top r_{i+1}}{r_i^\top r_i}$
12:         $d_{i+1} \leftarrow r_{i+1} + \beta_{i+1} d_i$
13:     end for
14:     return $x_{n-1}$
15: end function
```

The complete algorithm is shown in Algorithm 1. Like the gradient descent algorithm, CG requires a single matrix-vector multiplication per iteration (see Line 7). Since this is the most expensive operation the time complexity of CG is $\mathcal{O}(Tn^2)$. On the other hand, the number of iterations T required for convergence can be much smaller. The decrease in error norm as a function of iterations depends now on the square-root of the condition number:

$$\|e_i\|_A \leq 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^i \|e_0\|_A. \tag{2.70}$$

Fixed a desired accuracy $\epsilon > 0$, the maximum number of iterations needed to obtain a solution with that accuracy is

$$T = \left\lceil \frac{1}{2} \sqrt{k} \ln \frac{2}{\epsilon} \right\rceil \tag{2.71}$$

which is a strictly better dependency on the condition number than the GD algorithm (see Equation (2.57)).

2.5 Approximations to KRR

2.5.1 Random Fourier Features

Consider the feature-map view on reproducing kernel Hilbert spaces, where we have seen that, given a reproducing kernel k, there exists a function $\phi : \mathcal{X} \to \mathcal{H}$ such that

$$k(x, x') = \langle \phi(x), \phi(x') \rangle_{\mathcal{H}}. \tag{2.72}$$

Random Fourier features were introduced by Rahimi and Recht in 2007 (Rahimi et al., 2008) as a way to approximate the inner product (whose dimensionality can be infinite), with a finite number of *randomized* feature maps:

$$\langle \phi(x), \phi(x') \rangle_{\mathcal{H}} \approx z(x)^\top z(x') \tag{2.73}$$

for some function $z : \mathcal{X} \to \mathbb{R}^R$. Then the KRR solution of (2.28) becomes

$$\hat{f}(x) = \sum_{i=1}^{n} \alpha_i k(x_i, x) = \sum_{i=1}^{n} \alpha_i \langle \phi(x_i), \phi(x) \rangle_{\mathcal{H}} \approx \sum_{i=1}^{n} \alpha_i z(x_i)^\top z(x) = \beta^\top z(x), \tag{2.74}$$

where the explicit form of β is derived in Equations (2.82) to (2.84). That is, once the has been mapped into this R-dimensional space using z, one simply needs to solve a linear model in R dimensions to obtain $\beta \in \mathbb{R}^R$. If $R \ll n$ this is computationally more convenient than solving the original problem!

The theory of random Fourier features was originally developed for shift-invariant kernels such as the Gaussian and Laplacian kernels, whose function given two points x, y only depends on the difference $x - x' \coloneqq \delta$, *i.e.* $k(x, x') = k_0(x - x')$. More recently there has been development of random features for other types of kernels such as polynomial kernels (Kar et al., 2012; Pham et al., 2013), additive kernels (Vedaldi et al., 2012), histogram kernels (F. Li, Ionescu, et al., 2010), neural tangent kernels (Zandieh et al., 2021) and many more. Here we focus on shift invariant kernels, for which deriving the form of z starts from a theorem by Bochner (Bochner, 1959)

Theorem 2.5: Bochner's Theorem

A continuous kernel function $k(x, x')$ such that k is shift-invariant, is positive definite on $\mathbb{R}^d$ if and only if it is the the Fourier transform of a finite non-negative measure on $\mathbb{R}^d$. This implies that k is the Fourier transform of a non-negative measure p:

$$k(x, x') = k_0(\delta) = \int_{\mathbb{R}^d} p(\omega) \exp(-i\langle \omega, \delta \rangle) \, \mathrm{d}\omega. \tag{2.75}$$

Without loss of generality we can assume p to be a probability measure, and hence write (2.75) as an expectation

$$k_0(\delta) = \int_{\mathbb{R}^d} p(\omega)\exp(-i\langle\omega,\delta\rangle)\,\mathrm{d}\omega = \mathbb{E}_\omega[\exp(-i\langle\omega,\delta\rangle)]. \tag{2.76}$$

The first ingredient for defining random Fourier features is the approximation of expectation (2.76) with a Montecarlo sample, which allows to express $\mathbb{E}_\omega[\exp(-i\langle\omega,\delta\rangle)]$ as a finite-dimensional inner product

$$\begin{aligned} k(x,x') &= \mathbb{E}_\omega[\exp(-i\langle\omega,\delta\rangle)] \\ &\approx \sum_{i=1}^{R} \exp(-i\langle\omega_i, x - x'\rangle) \\ &= \begin{bmatrix} \frac{1}{\sqrt{R}}\exp(-i\langle\omega_1,x\rangle) \\ \vdots \\ \frac{1}{\sqrt{R}}\exp(-i\langle\omega_R,x\rangle) \end{bmatrix}^\top \begin{bmatrix} \frac{1}{\sqrt{R}}\exp(i\langle\omega_1,x'\rangle) \\ \vdots \\ \frac{1}{\sqrt{R}}\exp(i\langle\omega_R,x'\rangle) \end{bmatrix}. \end{aligned}$$

The other ingredient is the relationship between specific probability measures and well-known kernels

Lemma 2.2: Let $\omega \sim \mathcal{N}(0, \boldsymbol{I}_d)$, k a Gaussian kernel with bandwidth equal to one. Then $\mathbb{E}_\omega[\exp(-i\langle\omega, x - x'\rangle)] = k(x,x')$. Note that this is an instance of a well-known result, essentially stating that the Fourier transform of the Gaussian is also Gaussian.

Proof of Lemma 2.2:

$$\begin{aligned} \mathbb{E}_\omega[\exp(-i\langle\omega,\delta\rangle)] &= \int_{\mathbb{R}^d} p(\omega)\exp(-i\langle\omega,\delta\rangle)\,\mathrm{d}\omega \\ &= (2\pi)^{-d/2}\int_{\mathbb{R}^d}\exp(-\frac{1}{2}\langle\omega,\omega\rangle)\exp(-i\langle\omega,\delta\rangle)\,\mathrm{d}\omega \\ &= (2\pi)^{-d/2}\int_{\mathbb{R}^d}\exp(-\frac{1}{2}(\langle\omega,\omega\rangle - 2i\langle\omega,\delta\rangle - \langle\delta,\delta\rangle) - \frac{1}{2}\langle\delta,\delta\rangle)\,\mathrm{d}\omega \\ &= \exp(-\frac{1}{2}\langle\delta,\delta\rangle)(2\pi)^{-d/2}\int_{\mathbb{R}^d}\exp(-\frac{1}{2}(\langle\omega - i\delta, \omega - i\delta\rangle))\,\mathrm{d}\omega \\ &= \exp(-\frac{1}{2}\langle\delta,\delta\rangle) \end{aligned}$$

where we have used the definition of the Gaussian probability density in the second step, and noted in the last step noted that the integral's argument is an unnormalized Gaussian density which must amount to $(2\pi)^{d/2}$. □

Finally we turn to the problem of computing (2.76) in practice. To do so, note that we are interested in cases where both the kernel k and the probability distribution p are real-valued.

We can use Euler's formula to get

$$\begin{aligned}\int_{\mathbb{R}^d} p(\omega)\exp(-i\langle\omega,\delta\rangle)\,\mathrm{d}\omega &= \int_{\mathbb{R}^d} p(\omega)[\cos(\langle\omega,\delta\rangle) - i\sin(\langle\omega,\delta\rangle)]\,\mathrm{d}\omega\\ &= \int_{\mathbb{R}^d} p(\omega)\cos(\langle\omega,\delta\rangle)\,\mathrm{d}\omega - i\int_{\mathbb{R}^d} p(\omega)\sin(\langle\omega,\delta\rangle)\,\mathrm{d}\omega,\end{aligned}$$

where the second integrand is odd, so its integration over $\mathbb{R}^d$ is 0. Hence to define the feature map $z:\mathbb{R}^d\to\mathbb{R}^R$ such that $z(x)^\top z(x')\approx k(x,x')$, consider the following lemma.

Lemma 2.3: Let ω and b be two random variables such that $\omega\sim p(\omega)$ and $b\sim\mathcal{U}(0,2\pi)$. Define

$$z_\omega(x) = \sqrt{2}\cos(\omega^\top x + b). \tag{2.77}$$

Then

$$\mathbb{E}_{\omega,b}[z_\omega(x)z_\omega(x')] = \mathbb{E}_\omega[\cos(\omega^\top(x-x'))] = k(x,x'). \tag{2.78}$$

Proof of Lemma 2.3:

$$\mathbb{E}_{\omega,b}[z_\omega(x)z_\omega(x')] = \mathbb{E}_{\omega,b}[\sqrt{2}\cos(\omega^\top x + b)\sqrt{2}\cos(\omega^\top x' + b)]$$

Now let $e=\omega^\top x + b$ and $f=\omega^\top x' + b$, and recall the trigonometric identity $\cos(x+x') = \cos(x)\cos(x') - \sin(x)\sin(x')$. Then

$$\begin{aligned}2\cos(e)\cos(f) &= (\cos(e)\cos(f)-\sin(e)\sin(f)) + (\cos(e)\cos(f)+\sin(e)\sin(f))\\ &= (\cos(e)\cos(f)-\sin(e)\sin(f)) + (\cos(e)\cos(-f)-\sin(e)\sin(-f))\\ &= \cos(e+f)+\cos(e-f).\end{aligned}$$

Substituting back into the original expression

$$\mathbb{E}_{\omega,b}[z_\omega(x)z_\omega(x')] = \mathbb{E}_{\omega,b}[\cos(\omega^\top(x+x')+2b)] + \mathbb{E}_\omega[\cos(\omega^\top(x-x'))].$$

We focus on the first term, which is equal to $\mathbb{E}_\omega\Big[\mathbb{E}_b[\,\cos(\omega^\top(x+x')+2b)\mid\omega\,]\Big]$. Once again letting $g=\omega^\top(x+x')$

$$\mathbb{E}_b[\,\cos(g+2b)\mid\omega\,] = \int_0^{2\pi}\frac{1}{2\pi}\cos(g+2b)\,\mathrm{d}b = \frac{1}{2\pi}\left(\sin(g+2b)\Big|_0^{2\pi}\right) = 0$$

since the sine function is periodic with period 2π. By substituting back we recover the statement of the hypothesis. $\square$

Finally we can define the map z as

$$z(x) = \begin{bmatrix} \frac{1}{\sqrt{R}} z_{\omega_1}(x) \\ \vdots \\ \frac{1}{\sqrt{R}} z_{\omega_R}(x) \end{bmatrix} \tag{2.79}$$

for which

$$\begin{aligned} z(x)^\top z(x') &= \frac{1}{R}\sum_{i=1}^{R} z_{\omega_i}(x) z_{\omega_i}(x') && \text{by (2.79)} \\ &= \frac{1}{R}\sum_{i=1}^{R} 2\cos(\omega_i^\top x + b_i)\cos(\omega_i^\top x' + b_i) && \text{by (2.77)} \\ &\approx \mathbb{E}_\omega[\cos(\omega^\top (x - x'))] && \text{Lemma 2.3} \\ &= k(x, x'). \end{aligned}$$

Random Fourier Features for KRR The computational benefit of random features comes from "undoing" the kernel trick to go from the direct KRR solution of (2.32) to the solution of *linear* ridge regression where all data points x_i are replaced by $z(x_i)$, for $i = 1, \dots, n$. We can do this directly starting from the KRR solution

$$\hat{f}_\lambda(x) = \sum_{i=1}^{n} k(x, x_i)(\hat{\alpha}_\lambda)_i, \quad \hat{\alpha}_\lambda = (\boldsymbol{K} + n\lambda I)^{-1}\hat{y}, \tag{2.80}$$

replacing the kernel function with a dot product over feature maps (where we denote the feature map evaluated at all training points by $\Phi = [\phi(x_1), \dots, \phi(x_n)]^\top$)

$$\hat{f}_\lambda(x) = \sum_{i=1}^{n} \langle \phi(x), \phi(x_i) \rangle_{\mathcal{H}} (\hat{\alpha}_\lambda)_i, \quad \hat{\alpha}_\lambda = (\Phi\Phi^\top + n\lambda I)^{-1}\hat{y}, \tag{2.81}$$

and replacing the true - infinite dimensional - feature maps with their approximate and finite counterparts (as before, denote $Z = [z(x_1), \dots, z(x_n)]^\top \in \mathbb{R}^{n \times R}$)

$$\hat{f}_\lambda(x) \approx \sum_{i=1}^{n} \langle z(x), z(x_i) \rangle (\hat{\alpha}_\lambda)_i, \quad \hat{\alpha}_\lambda = (ZZ^\top + n\lambda I)^{-1}\hat{y} \tag{2.82}$$

$$= z(x)^\top Z(ZZ^\top + n\lambda I)^{-1}\hat{y} \tag{2.83}$$

$$= z(x)^\top (Z^\top Z + n\lambda I)^{-1} Z^\top \hat{y} \tag{2.84}$$

where in the last step we have used the push-through identity. Notice how we have gone from needing to invert a $n \times n$ matrix in (2.82) and (2.83) to a $R \times R$ matrix in (2.84). Since $R \ll n$, this represents a big computational speed-up. The asymptotic computational complexity of solving KRR with random features is

$$\mathcal{O}(R^3) \text{ time} \qquad \mathcal{O}(R^2 + nR) \text{ space} \tag{2.85}$$

for the direct solution of (2.84). To control the tradeoff between fast but approximate and exact but slow solutions, one can use parameter R: the number of random features. In fact it has been shown (Rudi and Rosasco, 2017) that a number of $R = \mathcal{O}(\sqrt{n}\log(n))$ random features suffices to ensure that the generalization error of RFF-KRR decreases in the same way (*i.e.* decreases as $\mathcal{O}(\frac{1}{\sqrt{n}})$) as the generalization error of the full KRR algorithm. If we use this random features budget, we obtain the following asymptotic complexity:

$$\mathcal{O}(n\sqrt{n}\log^3(n)) \text{ time} \qquad \mathcal{O}(n\sqrt{n}\log(n)) \text{ space} \tag{2.86}$$

which is strictly better than that of full KRR.

2.5.2 The Nyström Method

Another approach to approximating the KRR estimator which was introduced by Williams et al. (2001) and Smola et al. (2000) is the Nyström method. It considers a variant of ERM, where instead of minimizing on the whole space $\mathcal{H}$, a subspace $\mathcal{B} \subset \mathcal{H}$ is used.

From the representer theorem (2.3) we know that

$$\hat{f} = \sum_{i=1}^{n} \alpha_i k(x_i, \cdot) = \sum_{i=1}^{n} \alpha_i \phi(x_i) = \Phi^\top \alpha \in \operatorname{span}\{\phi(x_1), \dots, \phi(x_n)\} \tag{2.87}$$

so the regularized ERM solution of (2.33) belongs to the space $\mathcal{H}_n = \operatorname{span}\{\phi(x_1), \dots, \phi(x_n)\}$. Hence choosing $\mathcal{B} = \mathcal{H}_n$ will give the same solution as if considering the whole $\mathcal{H}$. The Nyström estimator comes from considering instead a subset of the training points

$$\{\tilde{x}_1, \dots, \tilde{x}_m\} \subset \{x_1, \dots, x_n\}, \qquad m \ll n, \tag{2.88}$$

defining $\mathcal{B} = \mathcal{H}_m = \operatorname{span}\{\phi(\tilde{x}_1), \dots, \phi(\tilde{x}_m)\}$ and proceeding to minimize over the new space

$$\tilde{f}_\lambda = \operatorname*{arg\,min}_{f \in \mathcal{H}_m} \tilde{\mathcal{E}}_\lambda, \quad \tilde{\mathcal{E}}_\lambda(f) = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - y_i)^2 + \lambda \|f\|^2_{\mathcal{H}_m}. \tag{2.89}$$

The points $\tilde{x}_1, \dots, \tilde{x}_m$ are called Nyström centers, inducing points or landmarks, and they can be picked uniformly at random, or using more complex schemes such as leverage score sampling (Rudi, Calandriello, et al., 2018). By the representer theorem, $f \in \mathcal{H}_m$ implies that it can be written as

$$f = \sum_{i=1}^{m} \beta_i k(\tilde{x}_i, \cdot) \tag{2.90}$$

for any $\beta \in \mathbb{R}^m$, (keep in mind that $k(\widetilde{x}_i, \cdot)$ denotes partial application of the kernel function and is equivalent to $k_{\widetilde{x}_i} \in \mathcal{H}$) and its norm will be $\|f\|^2_{\mathcal{H}_m} = \beta^\top \boldsymbol{K}_{mm}\beta$ where the entries of the kernel matrix $\boldsymbol{K}_{mm} \in \mathbb{R}^{m \times m}$ are $(\boldsymbol{K}_{mm})_{i,j} = k(\widetilde{x}_i, \widetilde{x}_j)$. Finally, we will use the kernel matrix between the whole training set and the Nyström centers $\boldsymbol{K}_{nm} \in \mathbb{R}^{n \times m}$ such that $(\boldsymbol{K}_{nm})_{i,j} = k(x_i, \widetilde{x}_j)$. With these definitions we can rewrite the empirical risk (2.89) as a minimization over β

$$\widetilde{\beta}_\lambda = \operatorname*{arg\,min}_{\beta \in \mathbb{R}^m} \frac{1}{n}\|\boldsymbol{K}_{nm}\beta - \hat{y}\|^2 + \lambda\beta^\top \boldsymbol{K}_{mm}\beta. \tag{2.91}$$

Once again, taking the gradient of (2.91) and setting it to zero the unique solution is obtained:

$$\widetilde{f}_\lambda = \sum_{i=1}^{m} (\widetilde{\beta}_\lambda)_i k(\widetilde{x}_i, \cdot), \quad \widetilde{\beta}_\lambda = (\boldsymbol{K}_{nm}^\top \boldsymbol{K}_{nm} + n\lambda \boldsymbol{K}_{mm})^{-1} \boldsymbol{K}_{nm}^\top \hat{y}, \tag{2.92}$$

where invertibility is guaranteed if $\boldsymbol{K}_{mm}$ is positive definite, otherwise the pseudo-inverse can be used instead. Note how the linear system which needs to be solved is now of dimension m^2 instead of n^2, and the full kernel matrix $\boldsymbol{K}$ does not need to be computed. Instead, one will only compute the matrix $\boldsymbol{K}_{nm}$ which contains only a small subset of columns. Further note that when $m = n$, we recover the full KRR solution. The direct solution of (2.92) has an asymptotic cost of

$$\mathcal{O}(m^3 + nm^2) \text{ time} \qquad \mathcal{O}(mn) \text{ space}. \tag{2.93}$$

Similarly to random Fourier features, the number of Nyström points m controls the tradeoff between speed and accuracy. We will see in Chapter 3 that $m = \mathcal{O}(\sqrt{n})$ centers are sufficient for strong theoretical guarantees, which brings the asymptotic cost of Nyström KRR down to

$$\mathcal{O}(n^2) \text{ time} \qquad \mathcal{O}(n\sqrt{n}) \text{ space}. \tag{2.94}$$

Again in Chapter 3 we will see how these bounds can be lowered even further.

Part I

Algorithms & Theory

Chapter 3

Fast and Scalable Kernel Methods

In this chapter we build upon the foundations introduced in Chapter 2, to describe the algorithmic details, theoretical underpinnings and efficient implementation of a fast kernel ridge regression solver. The algorithm we are going to present here, is named Falkon and was first introduced in Rudi, Carratino, et al. (2017) where learning bounds were derived. Improvements in the computational efficiency and scalability of the algorithm, along with a comprehensive suite of experiments which compare Falkon to other state of the art solvers, were made in the work of the present chapter which has also been published in Meanti, Carratino, Rosasco, et al. (2020).

3.1 Approximate Kernel Methods

As we have seen in Section 2.5, the full kernel ridge regression (KRR) algorithm does not scale to datasets with more than a few hundred thousand points. Therefore many ways to derive approximate solutions have been proposed in the literature. As we have seen in the previous chapter, they broadly fall into the category of random features (Rahimi et al., 2008; Rahimi et al., 2009; Yang et al., 2012; Le et al., 2013; B. Dai et al., 2014; Cutajar, Bonilla, et al., 2017; Z. Li, Ton, et al., 2019), or the Nyström method. Starting from Williams et al. (2001) and Smola et al. (2000) who first proposed it for approximating KRR, there have been several papers analyzing its theoretical performance: Bach (2013) provided sharp bounds in the fixed design setting, Gittens et al. (2016) instead focused on bounding the kernel matrix approximation, while Rudi, Camoriano, et al. (2015) provided error bounds for Nyström KRR in the more general random design setting. An important variant of the low-rank Nyström approximation is the hierarchical low-rank approximation (J. Chen et al., 2017) where the kernel matrix is first split into hierarchical blocks, each of which is approximated using Nyström. Similar decompositions based on partitioning and low-rank approximation have also been studied from both empirical and theoretical points of view in Si et al. (2017), Thomann et al. (2017), Carratino et al. (2021), and Müecke (2019) among others. As was mentioned in the previous chapter, one crucial aspect of the Nyström approximation is the choice of the landmark points. The most studied options are uniform random sampling and leverage score sampling (Rudi, Calandriello, et al., 2018), but a wide variety of heuristic methods – summarized in a paper by Kumar et al. (Kumar et al., 2012) – has been employed over time. The Nyström method has also been used for different learning algorithms such as k-means (Calandriello and Rosasco, 2018; S. Wang et al., 2019), kernel PCA (Sterge et al., 2020), supervised learning with general loss functions (Della Vecchia et al., 2021; Marteau-Ferey, Ostrovskii, et al., 2019; Marteau-Ferey, Bach, et al., 2019), kernelized bandits (Calandriello, Carratino, et al., 2019; Calandriello, Carratino, et al., 2020), *etc.*

The Bayesian equivalent of KRR, namely Gaussian process regression (GPR), has also seen the development of approximate solutions which closely resemble their frequentist counterparts, with the added twist of estimating the variance and optimizing hyperparameters. Broadly speaking, low-rank approximations for GPR fall under the name of *sparse* GP regression (SGPR). Quiñonero-Candela et al. (2005) details the following SGPR algorithms: (SoR) (Silverman, 1985), the deterministic training conditional (DTC) (Seeger et al., 2003), the fully independent training conditional (FITC) (Snelson et al., 2005), which were followed by several algorithms which used variational inference to optimize a Nyström-like model (Titsias, 2009; Hensman, Fusi, et al., 2013; Hensman, Durrande, et al., 2017) (SVGP and its variants). Another line of research which has considered sparse GP regression is structured kernel interpolation (SKI) (Wilson and Nickisch, 2015) which imposes a grid-like (Kronecker and Toeplitz) structure on the inducing points in order to exploit fast matrix vector multiplications which are only possible with such structure. Improvements to this method have been proposed in Gardner, Pleiss, R. Wu, et al. (2018), Gardner, Pleiss, Bindel, et al. (2018), and K. Wang et al. (2019). See H. Liu, Y.-S. Ong, et al. (2020) for a recent review on scalable GPR.

Given recent theoretical results showing that approximate KRR models can provide huge computational gains with no loss of accuracy (see for example Bach (2013), Rudi, Camoriano, et al. (2015), Y. Sun et al. (2018), and Z. Li, Ton, et al. (2019), we take the practical consequences of this fact to the extreme, developing and testing large scale kernel methods that can run efficiently on datasets with *billions* of points.

Following the Falkon algorithm (Rudi, Carratino, et al., 2017) we use a Nyström approach to reduce the problem size and also to derive a preconditioned gradient solver for kernel methods. We focus on smooth loss functions (in particular the squared and logistic losses), and consider iterative solvers based on the conjugate gradient (CG) method (Saad, 2003), see Section 2.4.3 for more details. Making these algorithmic ideas practical and capable of exploiting the GPU, requires developing a number of computational solutions, borrowing ideas not only from optimization and numerical analysis but also from scientific and high performance computing (Ltaief et al., 2011; Anzt et al., 2015; Catanzaro et al., 2008). The preconditioned conjugate gradient solver (Cutajar, Osborne, et al., 2016) used is tailored to take full advantage of both GPU acceleration and parallelization with multiple GPUs. To achieve this, we find it necessary to devise out-of-core variants of common linear algebra operations to guarantee optimal hardware utilization. We further optimize the numerical precision of different operations and investigate ways to perform matrix-vector multiplications most efficiently. The corresponding implementation is then tested extensively on a number of datasets ranging from millions to billions of points. For comparison, we focused on other available large scale kernel implementations that do not require data splitting, or multiple machines. In particular, we consider Eigenpro (Ma et al., 2019) which is an approach similar to the one we propose, GPyTorch (Gardner, Pleiss, Bindel, et al., 2018) and GPflow (Wilk et al., 2020) which come from the Gaussian process literature. While these latter solutions allow also for uncertainty quantification, we limit the comparison to prediction. We perform a systematic empirical evaluation running an extensive series of tests. Empirical results show that indeed our approach can process huge datasets in minutes and obtain state of the art performances, comparing favorably to other solutions, both in terms of efficiency and accuracy. More broadly, these results confirm and extend the observations made in Ma et al. (2017) and Ma et al. (2019), that kernel methods can now be seamlessly and effectively deployed on large scale problems. To make these new solutions readily available, the code used in our experiments is distributed as an easy to use library developed on top of PyTorch (Paszke et al., 2019), available at the following link https://github.com/falkonml/falkon. The rest of this chapter is organized as follows. In Section 3.2, we provide some background on the considered approaches to kernel learning. In Section 3.3, we detail the main algorithmic solutions in our implementation, whereas Section 3.4 is devoted to an empirical evaluation of the algorithm.

3.2 The Falkon Algorithm

Falkon is an algorithm first proposed in Rudi, Carratino, et al. (2017) which combined random projections (*i.e.* the Nyström method), an iterative solver and a carefully designed preconditioner for solving the kernel ridge regression problem. Crucially, the algorithm comes with strong theoretical guarantees: it converges to the correct solution as fast as full KRR, as long as a large enough number of centers are chosen.

3.2.1 Preconditioning

Consider the problem of kernel ridge regression, whose solution entails inverting an $n \times n$ matrix $\boldsymbol{K} + n\lambda\boldsymbol{I}$. In Section 2.3 we have seen that the *condition number* of this matrix is important for (a) determining whether the matrix inverse will be stable, and (b) determining the time complexity for iterative algorithms (*e.g.* gradient descent) to compute the inverse itself. When using an iterative solver, clearly the accuracy of the solution will also influence the statistical properties of the estimator. Indeed to obtain solutions with “good” (*i.e.* whose error decreases as $\mathcal{O}(1/\sqrt{n})$) convergence properties with respect to the size of the training set, it has been proven (Camoriano, Angles, et al., 2016; Raskutti et al., 2014) that a number of $t = \mathcal{O}(\sqrt{n}\log n)$ iterations of gradient descent are needed.

Naively, preconditioning can be thought of as a procedure to convert a linear system

$$(\boldsymbol{K} + n\lambda\boldsymbol{I})\alpha = \hat{y} \tag{3.1}$$

which may have a high condition number, into an equivalent one

$$B^{-1}(\boldsymbol{K} + n\lambda\boldsymbol{I})\alpha = B^{-1}\hat{y} \tag{3.2}$$

which can be solved more easily, as it has a lower condition number than the original problem. Note that, as long as B is invertible, the solution to (3.2) is the same as the solution to the original problem (3.1) since $B^{-1}(\boldsymbol{K} + n\lambda\boldsymbol{I})\alpha = B^{-1}\hat{y} \implies B^{-1}(\boldsymbol{K} + n\lambda\boldsymbol{I} - \hat{y}) = 0$. For preconditioning to be computationally useful, the inversion of B must be much faster than the original problem. On the other hand, the ideal preconditioner would be $B = \boldsymbol{K} + n\lambda\boldsymbol{I}$ which reduces the condition number to 1. These two objectives are at odds with each other, since the ideal preconditioner is as hard to compute as the original problem, so a compromise between them must be reached. As a side remark, note that the preconditioner of (3.2) is a *left* preconditioner which may destroy the symmetric structure of matrix $\boldsymbol{K} + n\lambda\boldsymbol{I}$, thus limiting the applicability of specialized solvers which only work with symmetric matrices. A *two-sided* preconditioner can be used instead, to preserve symmetry:

$$B^\top(\boldsymbol{K} + n\lambda\boldsymbol{I})B(B^{-1}\alpha) = B^\top\hat{y} \tag{3.3}$$

where we first solve with respect to $\beta \coloneqq B^{-1}\alpha$, and then with respect to α. The ideal two-sided preconditioner would then be

$$BB^\top = (\boldsymbol{K} + n\lambda\boldsymbol{I})^{-1}. \tag{3.4}$$

Efficient preconditioner design is an important research topic, and it has been explored for kernel ridge regression in Ma et al. (2017), Gonen et al. (2016), Cutajar, Osborne, et al. (2016), and Avron et al. (2017). Part of the contribution of the Falkon algorithm was to use the Nyström approximation both to approximate the KRR solution, and to approximate the ideal preconditioner.

Starting from the Nyström-KRR solution, which was introduced in Equation (2.92) and requires solving a linear system

$$(\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm} + \lambda n\boldsymbol{K}_{mm})\beta = \boldsymbol{K}_{nm}^\top\hat{y}, \tag{3.5}$$

consider once again the ideal preconditioner

$$BB^\top = \left(\boldsymbol{K}_{nm}^\top \boldsymbol{K}_{nm} + \lambda n \boldsymbol{K}_{mm}\right)^{-1}.$$

Computing $BB^\top$ is as hard as solving the original problem, but it can be approximated by a second round of subsampling applied to the rows of $\boldsymbol{K}_{nm}$. This leads to the Falkon preconditioner

$$\widetilde{B}\widetilde{B}^\top = \left(\frac{n}{m}\boldsymbol{K}_{mm}^2 + \lambda n \boldsymbol{K}_{mm}\right)^{-1}, \tag{3.6}$$

where intuitively $\boldsymbol{K}_{mm}^2 \approx \boldsymbol{K}_{nm}^\top \boldsymbol{K}_{nm}$.

3.2.2 Falkon with the squared loss

The Falkon algorithm with the squared loss corresponds to preconditioned Nyström KRR solved with the conjugate gradient method (see Section 2.4.3). Given a dataset $\{(x_i, y_i)\}_{i=1}^n$, where we denote by $X = [x_1, \dots, x_n]^\top \in \mathbb{R}^{n\times d}$ the data matrix and by $\hat{y} = [y_1, \dots, y_n]$ the labels; regularizer $\lambda > 0$, and a set of $m \ll n$ centers $\{\widetilde{x}_1, \dots, \widetilde{x}_m\} \subset \{x_1, \dots, x_n\}$ sampled uniformly at random from the training set, and denoted by $X_m = [\widetilde{x}_1, \dots, \widetilde{x}_m] \in \mathbb{R}^{m\times d}$, consider functions of the form

$$\widetilde{f}_\lambda(x) = \sum_{i=1}^m (\widetilde{\beta}_\lambda)_i k(x, \widetilde{x}_i). \tag{3.7}$$

We will only consider uniform sampling of the centers in this chapter, although other more involved sampling schemes exist and can give improved statistical guarantees (Rudi, Carratino, et al., 2017). The m coefficients contained in vector $\widetilde{\beta}_\lambda$ can be obtained using the two-sided preconditioner of Equation (3.6) to solve

$$\widetilde{B}^\top H \widetilde{B}\gamma = \widetilde{B}^\top \boldsymbol{K}_{nm}^\top \hat{y}, \quad \text{with } H = \boldsymbol{K}_{nm}^\top \boldsymbol{K}_{nm} + \lambda n \boldsymbol{K}_{mm}, \quad \gamma = \widetilde{B}^{-1}\widetilde{\beta}_\lambda. \tag{3.8}$$

In particular, Equation (3.8) is not solved for directly, but iteratively, using the conjugate gradient method.

The full algorithm, shown in Algorithm 2, comprises three main sections:

1. *Center selection*, where the Nyström centers are chosen either uniformly at random or with more complex strategies;

2. *Preconditioner evaluation*, where the preconditioner and right-hand side $R = \widetilde{B}^\top \boldsymbol{K}_{nm}^\top \hat{y}$ are computed ahead of time;

3. *Conjugate gradient*, where the linear-operation $\widetilde{B}^\top H \widetilde{B}\gamma$ is applied repeatedly with different candidate solutions γ proposed by the CG method.

Some of the details on computing the preconditioner function through the intermediate variables T and A will be explained in detail in Section 3.3.1. The general strategy is as follows

$$\widetilde{B} = \frac{1}{\sqrt{n}} T^{-1} A^{-1}, \quad T = \mathrm{chol}(\boldsymbol{K}_{mm}), \quad A = \mathrm{chol}\Big(\frac{1}{m} T T^\top + \lambda \boldsymbol{I}\Big), \tag{3.9}$$

where we use an upper Cholesky decomposition so that $T^\top T = \boldsymbol{K}_{mm}$ with T upper triangular, and similarly for A. In this way, $\widetilde{B}$ is never computed explicitly, but only through the triangular matrices T, A. The Cholesky decomposition chol applied to a $m \times m$ matrix requires $\mathcal{O}(\frac{1}{3}m^3)$ floating point operations. Hence the total computational complexity of the Preconditioner function is $\mathcal{O}(\frac{4}{3}m^3)$ time to perform two Cholesky decompositions and a triangular matrix multiplication, and $\mathcal{O}(m^2)$ space. Turning to the LinOp function, we have to solve several linear systems involving A and T. Since these are triangular matrices, the cost of a linear solve is just $\mathcal{O}(m^2)$ floating point operations. The kernel-vector multiplications at line 5 of Algorithm 2 dominate the function's cost, which totals in at $\mathcal{O}(4m^2 + 2nm)$. The space complexity is dominated by operations involving the whole dataset X (at lines 5 and 8 of Algorithm 2), which require $\mathcal{O}(nm)$ space when computed naively. However, notice that they always appear in a matrix multiplication with a vector, whose result is also a vector, and hence much smaller than $n \times m$. For this reason it is possible to compute the kernel matrices multiplied by a vector simultaneously, only keeping arbitrarily small blocks of the kernel itself in memory. Hence using blocked computations (detailed in Section 3.3) the total complexity of Falkon is

$$\mathcal{O}(tnm + m^3) \text{ time} \qquad \mathcal{O}(m^2) \text{ space}. \tag{3.10}$$

The question remains of how to choose the number of iterations t and the number of centers m. In the next Section we shall see how statistics can provide some answers.

3.2.3 Statistical considerations

The main difference between Falkon and full Nyström KRR consists in the optimization procedure, with a novel preconditioner. For this reason, the first step in deriving statistical bounds for the estimator defined by Algorithm 2, is to prove a relationship between the estimator after t iterations of CG $\widetilde{f}_{\lambda,t} = \sum_{i=1}^m (\widetilde{\beta}_{\lambda,t})_i k(x, \widetilde{x}_i)$, and the Nyström-KRR estimator $\widetilde{f}_\lambda$. In Rudi, Carratino, et al. (2017, Theorems 1, 2) it is proven that, under the following assumptions which closely resemble the ones for KRR (see Section 2.3): (a) the kernel function bounded by a constant $\kappa \geq 1$ (see Assumption 2.1), (b) the existence of $f_{\mathcal{H}} \in \mathcal{H}$ such that $\mathcal{E}(f_{\mathcal{H}}) = \inf_{f\in\mathcal{H}} \mathcal{E}(f)$ (see Assumption 2.2) and (c) assuming constant $\hat{v}^2 = \frac{1}{n}\sum_{i=1}^n y_i^2$ (see

Algorithm 2 Pseudocode for the Falkon algorithm. We assume without loss of generality that the data is made of real-valued vectors, and that we have a single real-valued output label. The first assumption can be easily relaxed by noting that it is only necessary to have a kernel which can be computed between pairs of data samples, the second can also be relaxed to have matrix-valued $\hat{y}$ and some matrix-vector operations will become matrix-matrix operations. The ConjugateGradient function implements the standard conjugate gradient algorithm for t iterations. LinOp performs the multiplication $\widetilde{B}^\top H \widetilde{B}\beta$ as in Equation (3.19), via matrix-vector products.

```
1:  function Falkon($X \in \mathbb{R}^{n\times d}, \hat{y} \in \mathbb{R}^n, \lambda, m, t$)
2:      $X_m \leftarrow$ RandomSubsample($X, m$)
3:      $T, A \leftarrow$ Preconditioner($X_m, \lambda$)
4:      function LinOp($\gamma$)
5:          $\boldsymbol{v} \leftarrow A^{-1}\gamma$
6:          $\boldsymbol{c} \leftarrow \boldsymbol{K}_{nm}^\top \boldsymbol{K}_{nm} T^{-1}\boldsymbol{v}$
7:          return $A^{-\top}(T^{-\top}\boldsymbol{c} + \lambda n \boldsymbol{v})$
8:      end function
9:      $R \leftarrow A^{-\top}T^{-\top}\boldsymbol{K}_{nm}^\top \hat{y}$
10:     $\gamma \leftarrow$ ConjugateGradient(LinOp, $R, t$)
11:     return $T^{-1}A^{-1}\gamma$
12: end function
13: function Preconditioner($X_m \in \mathbb{R}^{m\times d}, \lambda$)
14:     $\boldsymbol{K}_{mm} \leftarrow k(X_m, X_m)$
15:     $T \leftarrow \mathrm{chol}(\boldsymbol{K}_{mm})$
16:     $\boldsymbol{K}_{mm} \leftarrow {}^1/_m TT^\top + \lambda \boldsymbol{I}$
17:     $A \leftarrow \mathrm{chol}(\boldsymbol{K}_{mm})$
18:     return $T, A$
19: end function
```

Assumption 2.3); for $\delta \in (0, 1]$, it holds with probability at least $1 - \delta$ that

$$\mathcal{R}_{\mathcal{H}}(\widetilde{f}_{\lambda,t}) \leq 2\mathcal{R}_{\mathcal{H}}(\widetilde{f}_\lambda) \quad \text{when} \quad t \geq \log \mathcal{R}_{\mathcal{H}}(\widetilde{f}_\lambda) + \log\left(1 + \frac{9\kappa^2}{\lambda n}\log\frac{n}{\delta}\right) + \log 16\hat{v}^2. \tag{3.11}$$

Under the same assumptions, the authors prove the following excess risk bound

Theorem 3.1: Rudi, Carratino, et al. (2017)

Let $\delta \in (0, 1]$. Assume $y \in [-\frac{a}{2}, \frac{a}{2}]$ almost surely, $a > 0$. There exists $n_0 \in \mathbb{N}$ such that, for any $n > n_0$, if

$$\lambda = \frac{1}{\sqrt{n}}, \quad m \geq 75\sqrt{n}\log\frac{48\kappa^2 n}{\delta}, \quad t \geq \frac{1}{2}\log n + 5 + 2\log(a + 3\kappa),$$

then with probability at least $1-\delta$,

$$\mathcal{R}_{\mathcal{H}}(\widetilde{f}_{\lambda,t}) \leq \frac{c_0 \log^2 \frac{24}{\delta}}{\sqrt{n}} \tag{3.12}$$

where constants n_0, c_0 do not depend on λ, m, n, t and c_0 further does not depend on δ.

Compare the results in this last theorem with the excess risk bounds of full KRR in Equation (2.43) to see that the Falkon algorithm obtains the same learning rate, which is minmax optimal (Caponnetto et al., 2007) and hence not improvable without additional assumptions. Note that the Falkon algorithm only needs $\mathcal{O}(\log n)$ iterations of CG to converge, while the GD method used in (Camoriano, Angles, et al., 2016) required $\mathcal{O}(\sqrt{n}\log n)$ iterations for the same convergence.

Theorem 3.1 provides precise thresholds for m and t in order to achieve the optimal learning rate. In a practical setting some of the constants will remain unknown. We use asymptotic notation to connect the computational cost of Equation (3.10) with the thresholds on m and t to obtain the following space-time complexities:

$$\mathcal{O}(n\sqrt{n}\log n) \text{ time} \qquad \mathcal{O}(n) \text{ space}. \tag{3.13}$$

3.2.4 Falkon with self-concordant losses

The above ideas extend to the logistic loss and more generally to self-concordant loss functions, including the softmax loss (Marteau-Ferey, Ostrovskii, et al., 2019). In this case, iterative solvers are the default option since there is no closed form solution. The Nyström method can be used a first time to reduce the size of the problem, and then a second time to derive an approximate Newton step (Marteau-Ferey, Bach, et al., 2019). The main ideas from a theoretical and algorithmic viewpoint that we are going to recall here are developed in Marteau-Ferey, Ostrovskii, et al. (2019) and Marteau-Ferey, Bach, et al. (2019). The goal of our work is to make these ideas practical, by efficiently implementing and deploying the algorithms and making full use of the available computational architectures. In particular, we will focus on the following set of *generalized self concordant* loss functions:

Definition 3.1 (Generalized self-concordant function (Marteau-Ferey, Ostrovskii, et al., 2019)): Let $\mathcal{H}$ be a Hilbert space and let $z = (x, y) \in \mathcal{X} \times \mathcal{Y}$ be an input-output pair. We say that $\ell_z : \mathcal{H} \to \mathbb{R}$ is a generalized self-concordant function on $\mathcal{G} \subset \mathcal{H}$, when $\mathcal{G}$ is a bounded subset of $\mathcal{H}$ and ℓ_z is a convex and three times differentiable mapping on $\mathcal{H}$ such that for all $f, h, k \in \mathcal{H}$

$$\nabla^{(3)}\ell_z(f)[h,k,k] \leq \sup_{g\in\mathcal{G}} |g \cdot h| \; \nabla^2\ell_z(f)[k,k]. \tag{3.14}$$

Denote by R the quantity $\sup_{g \in \mathcal{G}} \|g\| < \infty$. For many loss functions $\mathcal{G}$ is just the ball in $\mathcal{H}$ centered in zero and with radius $R > 0$, then $\sup_{g \in \mathcal{G}} |g \cdot h| = R\|h\|$. The following loss functions, which are widely used in machine learning, are generalized self-concordant

Example 3.1: The following loss functions are generalized self-concordant functions:

1. Logistic regression: $\ell_z(f) = \log(1 + \exp(-yf(x)))$, where $z = (x, y)$ with $x \in \mathcal{X}$ and $y \in \{-1, 1\}$.
2. Robust regression: $\ell_z(f) = \varphi(f(x) - y)$ with $\varphi(u) = \log(e^u + e^{-u})$. Here $z = (x, y)$ with $x \in \mathcal{X}$ and $y \in \mathbb{R}$
3. Softmax regression: $\ell_z(f) = \log(\sum_{j=1}^k [f(x)]_j) - [f(x)]_y$, where now $f : \mathcal{X} \to \mathbb{R}^k$, $z = (x, y)$, with $y \in \{1, \dots, k\}$ and v_j denotes the j-th column of $v \in \mathbb{R}^k$.

Note, in particular, that the loss functions above are generalized self concordant, but not *self concordant* as discussed in Marteau-Ferey, Ostrovskii, et al. (2019).

For the statistical learning problem (see Equations (2.5) and (2.6)) with generalized self-concordant loss functions, a strong (minmax optimal) theoretical result analogous to the ones for kernel ridge regression (see Theorem 2.4) has been obtained (Marteau-Ferey, Ostrovskii, et al., 2019). In particular, the regularized empirical risk minimization solution (2.26) with generalized self-concordant losses instead of the squared loss achieves the bound

$$\mathcal{E}(\hat{f}_\lambda) - \inf_{f \in \mathcal{H}} \mathcal{E}(f) = \mathcal{O}(n^{-1/2}), \tag{3.15}$$

under standard regularity conditions on the learning problem and achieves fast learning rates similar to kernel ridge regression, considering more refined regularity conditions that are a natural extension of the conditions for kernel ridge regression (Marteau-Ferey, Ostrovskii, et al., 2019).

The paper Marteau-Ferey, Bach, et al. (2019) suggests to solve the regularized empirical risk minimization problem (2.26) for generalized self-concordant losses, by using a set of techniques that are extensions of the Falkon algorithm (Rudi, Carratino, et al., 2017). In particular, the problem is cast in terms of an approximate Newton method, with pseudocode shown in function GSC-Falkon of Algorithm 3. Nyström method is used a first time to reduce the size of the problem, and then a second time to derive an approximate Newton step (Marteau-Ferey, Bach, et al., 2019). Indeed a model of the form

$$f(x) = \sum_{i=1}^m \beta_i k(x, \tilde{x}_i)$$

is considered and the preconditioner of Equation (3.6) now plays the role of approximate Hessian, to perform the approximated Newton method.

Algorithm 3 Pseudocode for approximate Newton method with Falkon, for GSC losses (based on Marteau-Ferey, Bach, et al. (2019)). LinOp performs the multiplications via matrix-vector products as in Algorithm 2.

```
1: function GSC-Falkon($X \in \mathbb{R}^{n\times d}, \hat{y} \in \mathbb{R}^n, \lambda, m, t$)
2:     Set $\beta_0 = 0 \in \mathbb{R}^m$ and $\mu_0 > 0, q > 0$ according to Marteau-Ferey, Bach, et al. (2019).
3:     $X_m, \hat{y}_m \leftarrow$ RandomSubsample($X, \hat{y}, m$)
4:     for $k \in \mathbb{N}$ do
5:         $\beta_{k+1} \leftarrow$ WeightedFalkon($X, \hat{y}, X_m, \hat{y}_m\lambda_k, t, \beta_k$)
6:         $\mu_{k+1} \leftarrow q\mu_k$
7:         Stop when $\mu_{k+1} < \lambda$ and set $\beta_{last} \leftarrow \beta_k$.
8:     end for
9:     return $\widehat{\beta} \leftarrow$ WeightedFalkon($X, \hat{y}, X_m, \hat{y}_m, \lambda, t, \beta_{last}$)
10: end function
1: function WeightedFalkon($X \in \mathbb{R}^{n\times d}, \hat{y} \in \mathbb{R}^n, X_m \in \mathbb{R}^{m\times d}, \hat{y}_m \in \mathbb{R}^m, \lambda, t, \beta_0 \in \mathbb{R}^m$)
2:     $T, A \leftarrow$ WeightedPreconditioner($X_m, \boldsymbol{y}_m, \beta_0, \lambda$)
3:     function LinOp($\gamma \in \mathbb{R}^m$)
4:         $\boldsymbol{v} \leftarrow A^{-1}\gamma$
5:         $z \leftarrow \boldsymbol{K}_{nm}\gamma$                                   ▷ predictions on the dataset
6:         $D \leftarrow \text{DIAG}[(\ell''(z_1, y_1), \dots, \ell''(z_n, y_n))]$
7:         $\boldsymbol{c} \leftarrow \boldsymbol{K}_{nm}^\top D \boldsymbol{K}_{nm} T^{-1}\boldsymbol{v}$
8:         return $A^{-\top}T^{-\top}\boldsymbol{c} + \lambda n \boldsymbol{v}$
9:     end function
10:    $R \leftarrow A^{-\top}T^{-\top}\boldsymbol{K}_{nm}\hat{y}$
11:    $\gamma \leftarrow$ ConjugateGradient(LinOp, $R, t, \beta_0$)            ▷ CG solver starting from $\beta_0$
12:    return $T^{-1}A^{-1}\gamma$
13: end function
1: function WeightedPreconditioner($X_m \in \mathbb{R}^{m\times d}, \hat{y}_m \in \mathbb{R}^m, \beta \in \mathbb{R}^m, \lambda$)
2:     $\boldsymbol{K}_{mm} \leftarrow k(X_m, X_m)$                   ▷ Compute the kernel between inducing points
3:     $z \leftarrow \boldsymbol{K}_{mm}\beta$                                ▷ predictions on the Nyström points
4:     $T \leftarrow \text{chol}(\boldsymbol{K}_{mm})$
5:     $D \leftarrow \text{DIAG}[(\ell''(z_1, (\hat{y}_m)_1), \dots, \ell''(z_m, (\hat{y}_m)_m))]$
6:     $\boldsymbol{K}_{mm} \leftarrow {}^1/_m TDT^\top + \lambda \boldsymbol{I}$
7:     $A \leftarrow \text{chol}(\boldsymbol{K}_{mm})$
8:     return $T, A$
9: end function
```

Given points $(\tilde{x}_j, \tilde{y}_j)_{j=1}^m$ selected uniformly at random from the dataset, the approximate Hessian $\widetilde{H}$ at step k of the Newton method is a weighted version of the Falkon preconditioner and has the form

$$\widetilde{H} = \frac{1}{m} T \widetilde{D}_k T^\top + \lambda_k I, \tag{3.16}$$

where T is such that $T^\top T = \boldsymbol{K}_{mm}$ (*e.g.* it is the Cholesky decomposition of $\boldsymbol{K}_{mm}$, as in Equation (3.9)) and $\widetilde{D}_k \in \mathbb{R}^{m\times m}$ is a diagonal matrix whose j-th element is $\ell''(f_k(\widetilde{x}_j), \widetilde{y}_j)$ where we assume that the loss function is $\ell(f(x), y)$ and the second derivative is taken with respect to the first variable. f_k is the estimator at step k and λ_k is the regularization parameter at step k. λ_k increases at each iteration starting from an initial point λ_0, until it reaches the desired regularization λ (see function GSC-Falkon in Algorithm 3). In the same way as in the Falkon algorithm, the approximate Hessian is never built explicitly, we compute instead its Cholesky decomposition in terms of matrices T, A as $\widetilde{H}^{-1} = \widetilde{B}\widetilde{B}^\top$ with $\widetilde{B} = T^{-1}A^{-1}$, see the function WeightedPreconditioner in Algorithm 3. Then conjugate gradient is applied to the preconditioned problem, to solve the equation

$$\widetilde{B}^\top (\boldsymbol{K}_{nm} D_k \boldsymbol{K}_{nm} + \lambda \boldsymbol{I}) \widetilde{B}\gamma = \widetilde{B}^\top \boldsymbol{K}_{nm}^\top g_k. \tag{3.17}$$

where $D_k \in \mathbb{R}^{n\times n}$ is a diagonal matrix whose j-th element is $\ell''(f_k(x_j), y_j)$ and $g_k \in \mathbb{R}^n$ is such that $(g_k)_j = \ell'(f_k(x_j), y_j)$. To conclude, as proven in Marteau-Ferey, Bach, et al. (2019), to achieve the same learning rate of (3.15) and good practical performances, GSC-Falkon (Algorithm 3) needs to call WeightedFalkon only a small number of times with decreasing regularization parameters λ_k. Moreover, each time WeightedFalkon needs to execute only few iterations t of the CG algorithm. The algorithm presented in Algorithm 3 has an important theoretical appeal as proved in Marteau-Ferey, Bach, et al. (2019) since it is the fastest to date to achieve optimal learning rates for generalized self-concordant loss functions. The goal of our work is to make it also appealing from a practical viewpoint. This requires efficiently implementing and deploying Algorithm 3, making full use of the available computational architectures. Clearly the main bottlenecks here are the same of Falkon for squared loss and they are introduced and discussed in Section 3.3.

3.3 Scalability and GPU Implementation

In this section we discuss Algorithms 2 and 3 from the point of view of a practical implementation, which has to deal with actual memory usage and the details of computer architecture, leaving behind asymptotic notation. In particular, the advent of the general purpose graphics processing unit (GPGPU) has had an extremely positive impact on the deployment of many machine learning algorithms. The benefits of using a specialized computer with extreme parallelism – the GPU – have been reaped mostly by deep learning algorithms, which have adapted well to the constraints of this different programming paradigm. GPU chips have a peculiar architecture with rather different properties than the standard von Neumann computer; they are characterized by highly parallel computational power, relatively small local accelerator memory and slow memory transfers to and from the accelerator compared to their computational speed (Wilt, 2013). Nevertheless, the raw computing power in terms of number of floating point operations per second (FLOPS) which can be completed, is much higher than that of modern CPUs. As an example, a modern (as of 2022) consumer-grade GPU can complete 35 TeraFLOPS in single-precision, while a server-grade, much more expensive CPU can complete around 3 TeraFLOPS – a difference of more than $10\times$.

In their standard definition, kernel methods require large amounts of memory with a low density of operations per byte of memory used. This opens the question of how to adapt methods with low operation density to platforms designed to be extremely efficient with very high density of operations per byte. With this in mind, we started considering Falkon, which is the state of the art kernel solver with minimal computational requirements for optimal guarantees, with the goal to reformulate its computational structure to dramatically increase the density of operations per byte, and reduce as much as possible the required memory use / transfers. To achieve this goal, we use a number of carefully designed computational solutions which systematically reduce the impact of the inherent bottlenecks of multi-core/multi-GPU architectures, while leveraging their intrinsic potential. In particular in the rest of this section we will focus on

(a) minimizing the memory footprint of the solver, which has long been the main bottleneck for kernel methods, and is the main limitation encountered by current kernel solvers,

(b) dealing with limited memory on the GPU,

(c) reaching the highest possible accelerator utilization, parallelizing memory transfers and computation,

(d) using the enhanced capabilities of GPUs with reduced-precision floating point data.

3.3.1 Overcoming RAM memory bottleneck

Kernel solvers that use the Nyström method rely on the matrices $\boldsymbol{K}_{mm}$ and $\boldsymbol{K}_{nm}$. Since $\boldsymbol{K}_{nm}$ is used only in matrix-vector products, we can avoid constructing it explicitly (as we shall see in the following paragraphs) which leaves us to deal with the $\boldsymbol{K}_{mm}$ matrix. When the number of centers m is large, it is crucial to carefully manage the memory needed for this task: in our implementation we only ever allocate a single $m \times m$ matrix, and overwrite it in different steps to calculate the preconditioner. Indeed, with the Falkon preconditioner of Equation (3.6), the matrix $\boldsymbol{K}_{mm}$ itself is not needed in the conjugate gradient iteration.

Figure 3.1 shows the total memory usage, which consists of the preconditioner occupying approximately 90% of the memory (see last paragraph of Section 3.3.1), the weight vector $\widetilde{\beta}_{\lambda,t}$

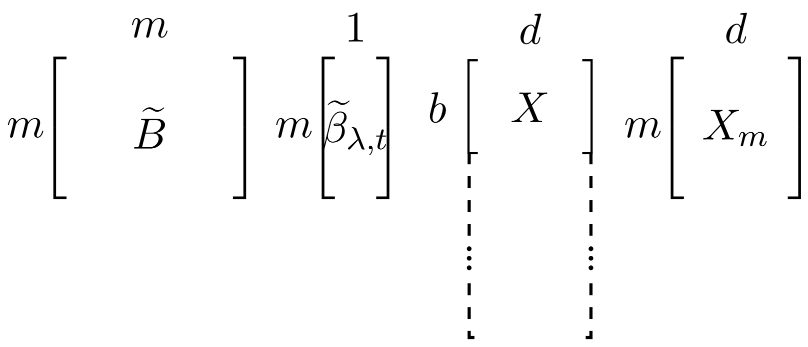


Figure 3.1 Buffers allocated in RAM by the Falkon algorithm: a $m \times m$ matrix for both $\boldsymbol{K}_{mm}$ and the preconditioner, the parameter vector, a batch of the input dataset and the Nyström centers.

and two buffers holding (part of) the m inducing points and a small subset of the dataset X, needed to compute a block of $\boldsymbol{K}_{nm}$.

In-place computation and storage of the preconditioner. We recall the preconditioner decomposition of Equation (3.9), keeping in mind that $\mathrm{chol}(A) = L$ implies $L^\top L = A$ with L lower triangular. $\widetilde{B}$ can be decomposed into two triangular matrices obtained via upper Cholesky decompositions (*i.e.* $TT^\top = \boldsymbol{K}_{mm}$, T upper-triangular),

$$\widetilde{B} = \frac{1}{\sqrt{n}} T^{-1} A^{-1}, \quad T = \mathrm{chol}(\boldsymbol{K}_{mm}), \quad A = \mathrm{chol}\left(\frac{1}{m} TT^\top + \lambda \boldsymbol{I}\right). \tag{3.18}$$

All operations are performed in-place allocating a single $m \times m$ matrix as shown in Figure 3.2 and as described next:

1. A matrix of dimension $m \times m$ is allocated in memory;
2. The $\boldsymbol{K}_{mm}$ kernel is computed in blocks on the GPU and copied to the matrix;
3. Cholesky decomposition of the upper triangle of $\boldsymbol{K}_{mm}$ is performed on the GPU (if the kernel does not fit GPU memory an out-of-core algorithm is used, see later sections). This operation must be run *in place*: the input and output must reside in the same block of memory.
4. The product $TT^\top$ is computed in blocks via GPU and stored in the lower part. This operation can run *out of place* (that is, the output does not overwrite the input).
5. Out-of-core, in-place Cholesky decomposition is performed on the lower triangle to get $A^\top$.

Additional care is needed to take into account the matrix diagonal, not described here for brevity.

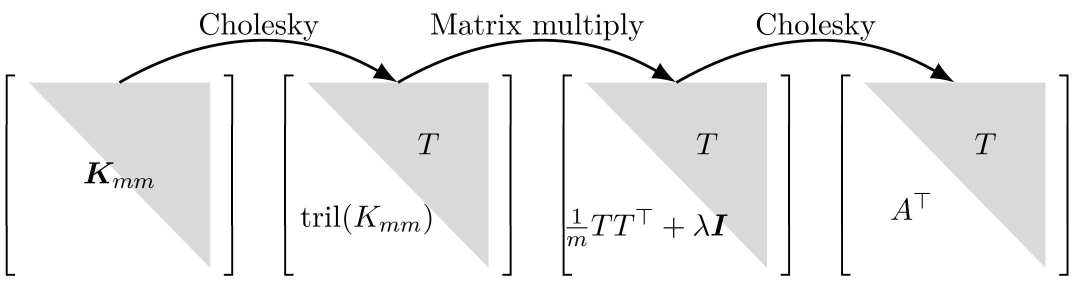


Figure 3.2 In-memory preconditioner computation

Elimination of the storage of $\boldsymbol{K}_{mm}$. Now consider more carefully the left hand side of the linear system solved by Falkon (see also Equation (3.8)):

$$\widetilde{B}^\top(\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm} + \lambda n\boldsymbol{K}_{mm})\widetilde{B}\gamma, \quad \text{with } \gamma = \widetilde{B}^{-1}\widetilde{\beta}_\lambda.$$

By definition of the preconditioner and of the Cholesky decomposition, $(T^{-1})^\top\boldsymbol{K}_{mm}T^{-1} = (T^{-1})^\top T^\top TT^{-1} = \boldsymbol{I}$. Then

$$\widetilde{B}^\top(\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm} + \lambda n\boldsymbol{K}_{mm})\widetilde{B}\gamma = (A^{-1})^\top(T^{-1})^\top(\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm} + \lambda n\boldsymbol{K}_{mm})T^{-1}A^{-1}\gamma \tag{3.19}$$
$$= (A^{-1})^\top[(T^{-1})^\top\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm}T^{-1} + \lambda n\boldsymbol{I}]A^{-1}\gamma. \tag{3.20}$$

This characterization shows that $\boldsymbol{K}_{mm}$ is not needed during CG optimization, and the storage space used for holding it can be repurposed for preconditioner computation with no adverse effects.

Blockwise $\boldsymbol{K}_{nm}$-vector product on GPU. The conjugate gradient algorithm will repeatedly execute Equation (3.20) for different candidate vectors γ. The most expensive operations are the matrix-vector products $\boldsymbol{K}_{nm}^\top(\boldsymbol{K}_{nm}v)$ for an arbitrary vector $v \in \mathbb{R}^{m\times 1}$ which – if computed explicitly – would require $n \times m$ memory. A more efficient way of proceeding is to to split the input data $X \in \mathbb{R}^{n\times d}$ in B batches of b rows each $\{X_{b,:} \in \mathbb{R}^{b\times d}\}_{b=1}^B$, so that matrix-vector products can be accumulated between batches using the formula $\sum_{b=1}^B k(X_{b,:}, X_m)^\top(k(X_{b,:}, X_m)v)$. The matrix blocks to be held in memory are summarized in Figure 3.1 for a total size of $m \times (m + d + 1) + b \times d$ where b can be small under memory pressure, or large for greater performance. It is important to note that $k(X_{b,:}, X_m)$ is never stored in main memory, as all operations on it are done on the GPU. This reduces the required memory bandwidth between CPU and GPU dramatically.

3.3.2 Fitting in GPU memory and dealing with multiple GPUs

While the main RAM might be a bottleneck for the full kernel matrix, GPUs have an even smaller amount of memory, and another level of splitting is needed to exploit their speed.

For example, a typical server may have 256GB of RAM and 4 GPUs with 16GB ram each; a preconditioner with $m = 2 \times 10^5$ occupies 150 GB and $\boldsymbol{K}_{nm}$ with $n = 10^7$ would need 2000 GB of memory if stored.

Therefore we need to deal with both efficiently computing of $\boldsymbol{K}_{nm}$-vector product in chunks that can fit a GPU, and computing the preconditioner which may not fit in GPU memory. Out-of-core (OOC) algorithms are developed specifically for computer architectures where a large storage layer (*e.g.* main RAM) is coupled to a layer which can perform computations but has little memory (*e.g.* the GPU). Often a side goal is to minimize the amount of data transfer between the two layers. Unfortunately, common machine learning libraries such as Tensorflow (Abadi et al., 2015) or PyTorch (Paszke et al., 2019) do not implement OOC versions of the required matrix operations, leaving potentially complex implementations to the users. Hence, in our library, we provide these implementations in easily reusable form. It is important to note that splitting our workload to fit in GPU also provides an easy path to parallelization in a multi-GPU system: new chunks of computation are assigned to the first free GPU, effectively redistributing the workload between multiple accelerators when available.

Optimized block decomposition for out-of-core $\boldsymbol{K}_{nm}$-vector multiplication. As seen in the previous section, matrix-vector products can be split along the dimension n, resulting in independent chunks of work that need to be summed up at the end. The algorithm to compute the product between a kernel matrix and a vector in an out of core way proceeds as follows:

1. transferring a block of data onto the device,
2. computing the kernel on device and multiplying it by the vector,
3. copying the result back to the host.

This sequence of operations minimizes expensive data-transfers between host and device since the kernel matrix is never moved. In particular, the computation is also split along dimensions m and d, to maximize the ratio between computational complexity and transfer time: *i.e.* maximizing $\frac{qrs}{qs+ds}$ subject to $qs + ds \leq G$, where q, r and s are the batch dimensions along n, m and d respectively, and G is the available GPU memory.

Out-of-core multi-GPU Cholesky decomposition. Other operations, such as Cholesky decomposition and triangular matrix multiplication (lines 15, 16, 17 of Algorithm 2), can also benefit from GPU execution. Here we describe, at a high level, our algorithm for multi-GPU OOC Cholesky decomposition inspired by Ltaief et al. (2011) and R. Wu (2018). We leave further details to Chapter B. Consider a symmetric matrix A, split into $B \times B$ tiles $A_{ij} \in \mathbb{R}^{b \times b}, i \in [B], j \in [B]$, assumed of equal size for brevity. We want a factorization $A = LL^\top$,

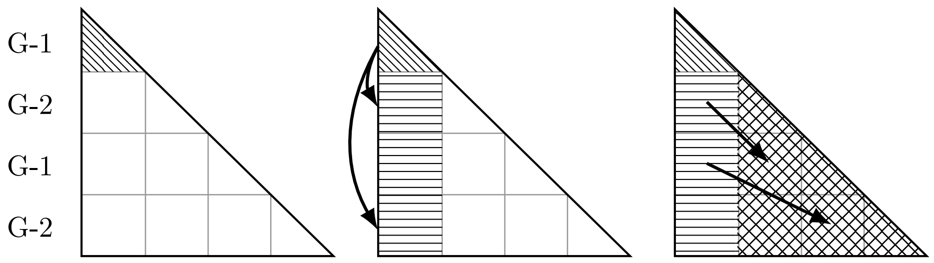


Figure 3.3 Three phases of the block Cholesky decomposition for updating the first column. Arrows indicate inter-GPU memory transfers between accelerators G-1 and G-2.

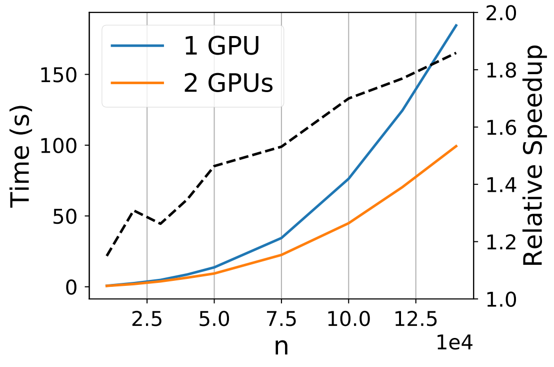
where L is lower triangular, with the formula $A_{i,j} = \sum_{k=1}^{j} L_{i,k} L_{j,k}^\top$.

$$\begin{pmatrix} A_{1,1} & & \\ A_{2,1} & A_{2,2} & \\ \vdots & \ddots & \\ A_{n,1} & \dots & A_{n,n} \end{pmatrix} = \begin{pmatrix} L_{1,1} & & \\ L_{2,1} & L_{2,2} & \\ \vdots & \ddots & \\ L_{n,1} & \dots & L_{n,n} \end{pmatrix} \begin{pmatrix} L_{1,1}^\top & L_{2,1}^\top & \dots & L_{n,1}^\top \\ & L_{2,2}^\top & \dots & L_{n,2}^\top \\ & & \ddots & \vdots \\ & & & L_{n,n}^\top \end{pmatrix}$$

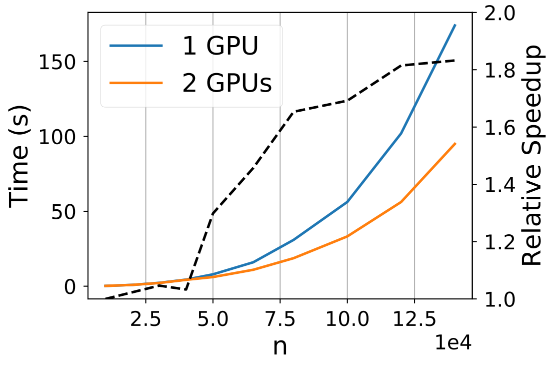
The algorithm runs in-place, updating one column of A at a time. Each column update proceeds in three steps, illustrated in Figure 3.3. Clearly $A_{1,1} = L_{1,1} L_{1,1}^\top$ so we compute $L_{1,1}$ by a Cholesky decomposition on tile $A_{1,1}$ which is small and can be done entirely on the GPU (*e.g.* with cuSOLVER (NVIDIA Corporation, 2020b)). Then we consider the other tiles of the first block column of L for which $A_{j,1} = L_{j,1} L_{1,1}^\top$ with $j > 1$. Since we know $L_{1,1}$ from the first step, we obtain $L_{j,1} = A_{j,1} L_{1,1}^{-\top}$ for all $j > 1$ by solving a triangular system (on the GPU). Finally the first block column of L is used to update the trailing submatrix of A. Note that $A_{i,j} = \sum_{k=1}^{j} L_{i,k} L_{j,k}^\top = L_{i,1} L_{j,1}^\top + \sum_{k=2}^{j} L_{i,k} L_{j,k}^\top$ for $2 \le j \le i$, so we can update the trailing submatrix as $A_{i,j} = A_{i,j} - L_{i,1} L_{j,1}^\top$. We implemented a parallel version of the above algorithm which distributes block-rows between the available processors in a 1D block-cyclic way (*e.g.* Figure 3.3 (left): rows 1 and 3 are assigned to GPU-1, rows 2 and 4 are assigned to GPU-2). For each column update, one processor executes the first step and transfers the result to the others (the arrows in Figure 3.3), which can then execute step 2 in parallel. To update the trailing matrix, further data transfer between devices may be necessary. The tile-size is chosen as a function of GPU memory: each device needs to hold one block column plus a single block at any given time.

The preconditioner contains two Cholesky decompositions, which can be handled as described above, and one triangular matrix multiplication (this is also known as the LAUUM operation in LAPACK (Anderson et al., 1999)). We handle the latter computations similarly to the Cholesky decomposition, as described in Chapter B.

Figure 3.4 shows the results from running the two types of operation we are considering with one and two GPUs. At low matrix sizes the speedup with two GPUs is negligible, especially



(a) Parallel LAUUM.



(b) Parallel Cholesky decomposition.

Figure 3.4 Running time of preconditioner operations with one and two GPUs. The relative speed-up with 2 GPUs is shown in the black dashed line. The LAUUM operation (triangular matrix multiplication) was run out-of-place, which typically leads to better parallelism, while the Cholesky decomposition was run in-place.

for the Cholesky decomposition. In such cases it is best to use a single GPU (especially since for $n \approx 40000$ the whole matrix fits in GPU memory, so an in-core algorithm can be used). With higher matrix sizes, having more than one GPU starts bringing real benefits, with a peak speedup around 1.8× for preconditioners of size 140 000. The factors blocking such speedup from reaching a perfect 2× are different for the two operations. The Cholesky decomposition has many data dependencies, as can be seen by the communication arrows in Figure 3.3. This limits the amount of work which can be parallelized to certain specific parts of the algorithm. Synchronization between threads and processors is necessary at the boundaries of the parallelizable work-items to ensure algorithm correctness, which slows the algorithm down further. Because the LAUUM operation runs out-of-place (which means that input and output reside at two different memory locations, see Chapter B for more details), it does not need any synchronization, since the original data remains available to all processors. The main bottleneck is an operation of the algorithm which must run on the CPU, since an equivalent GPU implementation does not exist in popular linear algebra packages (for the interested readers this is the operation at Line 7 of Algorithm 4 in Chapter B). We left porting this operation to the GPU as future work, but it has the potential to speed up the out-of-core LAUUM algorithm considerably.

3.3.3 Optimizing data transfers and other improvements

The speed of computations on GPUs is such that data transfers to and from the devices become significant bottlenecks. We have described earlier how, for matrix-vector products, the computed blocks of $\boldsymbol{K}_{nm}$ never leave the device. Further, optimization is possible by parallelizing computations and data transfers. Indeed, modern GPUs have an independent and

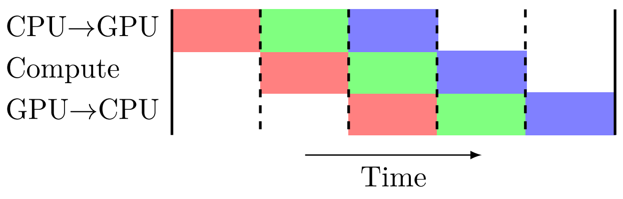


Figure 3.5 Overlapping memory transfers and computation.

parallel control on the following activities: loading from RAM, saving to RAM, performing computations. By controlling the dimensions of the chunk, we control the cost of loading and saving operations, so that they never exceed the computational time. By running three parallel threads for the same GPU and assuming equal duration of each piece of work, we can run t GPU computations in $t + 2$ time units instead of $3t$ time units for a serial implementation (see Figure 3.5, where $t = 3$). This guarantees near optimal usage of the GPU and in practice corresponds to a considerable speed up of matrix-vector products.

Leveraging the trade-off numerical precision / computational power. GPUs are designed to achieve peak performance with low precision floating point numbers, to such a great degree that going from 64 to 32-bit floats can correspond (depending on the exact architecture) to $\approx 10\times$ throughput improvement. However, changing precision can lead to unexpected problems. For example, computing the Gaussian kernel is commonly done by expanding the norm $\|x - x'\|^2 = x^\top x - 2x^\top x' + x'^\top x'$, but in high dimensions $\|x\|$ and $\|x'\|$ and the cross-term can have very high absolute values, so their sum will retain fewer significant digits. Loss of precision may also lead to some of the small eigenvalues of the computed kernel matrix to become negative (by the effect of noise introduced by inexact computations), in turn causing the Cholesky decomposition to fail. To avoid this, we compute $\boldsymbol{K}_{mm}$ in blocks, converting each block to 64-bit precision for the sum, and then back to 32-bits.

Dealing with thin submatrices. As a result of our block division strategies, it may happen that blocks become thin (*i.e.* one dimension is small). In this case, matrix operations, *e.g.* using cuBLAS (NVIDIA Corporation, 2020a), cannot leverage the full computational power. In turn this can reduce performance, breaking the inherent computational symmetry among GPUs which is crucial for the effectiveness of a parallel system like the one proposed in this paper. To guarantee good performance for this case, instead of using standard GPU operations, we perform matrix-vector products using KeOps (Charlier, Feydy, Glaunès, and Durif, 2020; Charlier, Feydy, Glaunès, Collin, et al., 2021): a specialized library to compute kernel matrices very efficiently when one dimension is small, see Table 3.1.

Dealing with sparse datasets. On the other side of the spectrum, sparse datasets with high dimensionality are common in some areas of machine learning. While the kernel computed

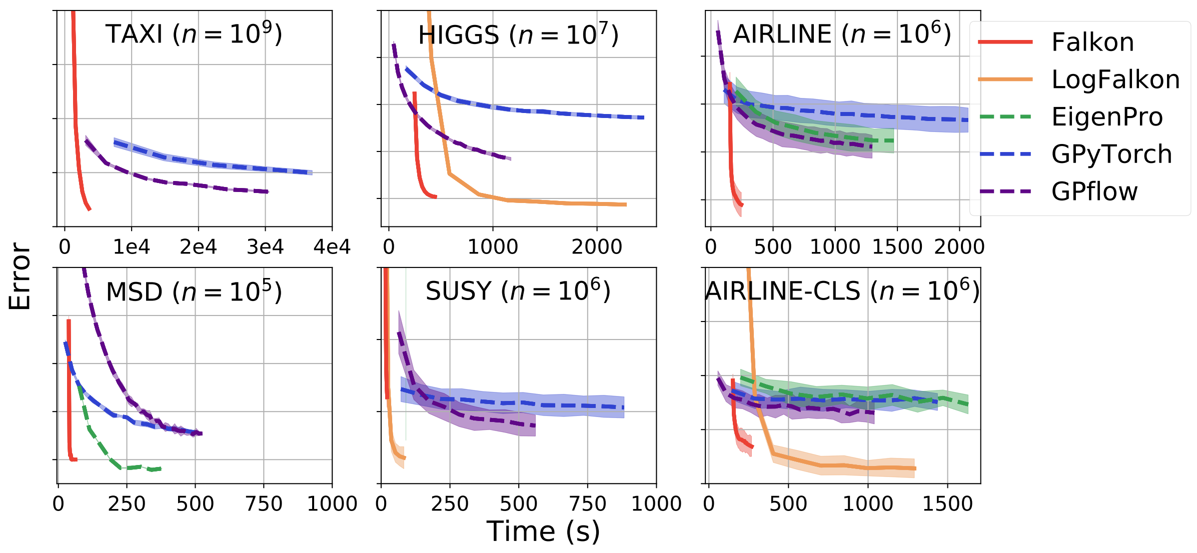


Figure 3.6 Benchmarking kernel solvers on large scale datasets with millions and billions points. Our approach (red and yellow lines) consistently achieves state of the art accuracy in minutes.

on such datasets will be dense, and thus can be handled normally, it is inefficient and in some cases impossible (*e.g.* with $d \sim 10^6$ as is the case for the YELP dataset we used) to convert the inputs to a dense representation. We therefore wrapped specialized sparse linear algebra routines to perform sparse matrix multiplication (NVIDIA Corporation, 2020c), and adapted other operations such as the row-wise norm to sparse matrices. Thus our library handles sparse matrices with no special configuration, both on the GPU and – if a GPU is not available – on the CPU.

3.4 Experiments

We ran a series of tests to evaluate the relative importance of the computational solutions we introduced, and performed extensive comparisons on real-world datasets. Figure 3.6 shows that our implementation of the Falkon algorithm (denoted as **Falkon** for squared loss and **LogFalkon** for logistic loss) converges much faster than competing implementations of kernel-based algorithms, while maintaining the same or better accuracy. We start off this section with an ablation study of the various performance-enhancing features we introduced in Section 3.3, followed by a description of the algorithms used for benchmarking. We then present the benchmark results on large scale datasets (up to one billion points), and a brief survey of results in the literature to back up our experiments. Towards the end of this section, additional experiments are presented to show that the proposed method is state of the art on small datasets as well, and to show how it scales to multiple devices. For a description of the datasets used, and links from where they can be retrieved, please see Chapter A.

Table 3.1 Relative performance improvement of the implemented optimizations w.r.t. Rudi, Carratino, et al. (2017). The experiment was run with the HIGGS dataset, 1×10^5 centers and 10 conjugate gradient iterations.

Experiment	Preconditioner		Iterations	
	Time	Improvement	Time	Improvement
Falkon from Rudi, Carratino, et al. (2017)	2337 s	–	4565 s	–
Float32 precision	1306 s	1.8×	1496 s	3×
GPU preconditioner	179 s	7.3×	1344 s	1.1×
2 GPUs	118 s	1.5×	693 s	1.9×
KeOps	119 s	1×	232 s	3×
Overall improvement		19.7×		18.8×

3.4.1 Relative impact of performance optimizations

We first evaluate the relative importance of the computational solutions introduced in our work. We ran Falkon on the HIGGS dataset several times with the same hyperparameters ($m = 1\times10^5$ and 10 CG iterations), but with different *features* enabled. Each feature roughly corresponds to one of the performance optimizations discussed in Section 3.3. The outcome of these experiments is given in Table 3.1. The baseline model is very similar to the original Falkon implementation (Rudi, Carratino, et al., 2017), where the preconditioner ran on the CPU, float64 (or *double*) precision was being used, but matrix-vector multiplications for the CG algorithm were GPU accelerated. As a first optimization we used float32 (or *single*) precision for all computations, with care taken to avoid accumulation of error when computing the $\boldsymbol{K}_{mm}$ matrix as discussed in Section 3.3. This immediately resulted in a 2× speedup for the preconditioner (which now runs on the CPU), and 3× for the CG iterations. Switching to a GPU preconditioner (using the out-of-core algorithms previously described) gave a huge boost to the preconditioner running time which went from more than 20 min to just under 3 min. Adding a second GPU produced a perfect 2× speedup for the CG iterations, and a more modest 1.5× speedup for the preconditioner which (a) involves operations which are not perfectly parallelizable and (b) incurs in some fixed startup costs. Finally, since the HIGGS dataset has only 9 features (thus the data matrix is thin), we can use KeOps (Charlier, Feydy, Glaunès, Collin, et al., 2021) with great benefits to the speed of matrix-vector multiplications. Overall our implementation boasts a nearly 20× improvement over the baseline, which makes learning on several huge datasets doable in a matter of minutes.

3.4.2 Algorithms & Experimental Settings

Algorithms under test

We compare against the following software packages: EigenPro (Ma et al., 2019), GPflow (A. Matthews et al., 2017) and GPyTorch (Gardner, Pleiss, Bindel, et al., 2018).

The first library implements a KRR solver based on preconditioned block-coordinate gradient descent where the preconditioner is based on a truncated eigendecomposition of a data subsample. EigenPro provides a fully in-core implementation and therefore does not scale to the largest datasets we tried. On some datasets EigenPro required the training data to be subsampled to avoid GPU memory issues. The only hyperparameters in the software package – other than the kernel parameters – are the ones governing the preconditioner's size. On some experiments we found it necessary to manually tune the learning rate (we divided the automatically inferred learning rate by a fixed integer, denoted by $\mu\div$ in Table 3.2).

The other two packages implement several GP approximations and exact solvers, and we had to choose the model which would give a more appropriate comparison: we decided to avoid deep GPs (Damianou et al., 2013; Wilson, Hu, et al., 2016; Cutajar, Bonilla, et al., 2017) since they share more similarities to deep nets than to kernel methods; on the other hand the exact GP – even when implemented on GPU (Gardner, Pleiss, Bindel, et al., 2018; K. Wang et al., 2019) – as well as structured kernel interpolation (Wilson and Nickisch, 2015; Gardner, Pleiss, R. Wu, et al., 2018) approximations do not scale to the size of datasets we are interested in. The only GP models which would scale up to tens of millions of points are stochastic variational GPs (SVGP). The SVGP is trained in minibatches by maximizing the ELBO objective with respect to the variational parameters and the model hyperparameters. Stochastic training effectively constrains GPU memory usage with the minibatch size. Hyperparameters include kernel parameters (such as the length-scale of the RBF kernel) as well as the inducing points which – unlike in Falkon – are modified throughout training using gradient descent. For this reason SVGP works well even with very few inducing points, and all operations can run in-core.

While GP solvers are capable of estimating the full predictive covariance, we ensured that the software did not compute it, and further we did not consider prediction times in our benchmarks. Furthermore we always considered the Gaussian kernel with a single length-scale, due to the high effort of tuning multiple length-scales for Falkon, although for GPs tuning would have been automatic. Both GPyTorch and GPflow implement the same SVGP model, but we found the best settings on the two libraries to be different; the discrepancies in running time and accuracy between the two GP libraries come from implementation and tuning differences. We ran all algorithms under as similar conditions as possible: same hardware, consistent software versions, equal floating-point precision and equal kernels (we always considered the Gaussian kernel with a single length-scale). Hyperparameters were optimized manually by training on a small data subset, to provide a sensible trade off between performance and accuracy: we increased the *complexity* of the different algorithms until they reached high GPU utilization since this is often the knee in the time-accuracy curve.

For GPflow (v2.1.3) we used the SVGP model with Gaussian likelihood for regression, Bernoulli for binary classification and Softmax for multi-class problems. We used Adam for optimization and tuned the learning rate, the number of inducing points, and the constraints on the variational distribution covariance (*i.e.* diagonal or full covariance matrix). We found that using a full covariance matrix was rarely beneficial and increased training times slightly, so all final experiments used a diagonal covariance matrix. The number of parameters optimized by gradient descent is $m \times d + m \times 2 + 3$, which includes the inducing points, the variational parameters, two parameters for the Gaussian kernel (length-scale and variance) and the variance of the likelihood. For multi-class problems separate variational parameters were trained for each class. Since we wished to use single-precision floating point numbers in order to make GPU training more efficient, we found that natural gradient optimization was unstable. It remains to be seen whether the tradeoff between double-precision data and natural gradient optimization could further improve results. We further tested the benefits of using whitening of the inducing points, and found that it decreased per-epoch running times by about $2\times$, while at the same time slowing down convergence by around the same amount. In practice this meant that the difference in global running time was not strongly affected by whitening, and we ended up using it only for the HIGGS dataset.

For GPyTorch (v.1.2.0) we used the SVGP model with Gaussian and Bernoulli likelihoods. We were unable to run GPyTorch's SVGP model on the TIMIT dataset due to problems in dealing with multiple outputs. We used the natural gradient optimizer to learn the variational parameters, and Adam to learn the other hyperparameters. The learning rate of the two optimizers was kept equal and tuned for best performance. We further optimized the number of inducing points, and variational distribution constraints. In practice we found it necessary to use the natural gradient variational distribution for regression problem, and the lower-triangular parameterization for classification problems (which are non-conjugate). We found that using unwhitened inducing points was around $3\times$ faster than performing whitening, and did not hamper convergence. While GPyTorch is theoretically able to run on multi-GPU systems, we noticed that this feature was not available for the SVGP model thus we always used a single GPU; furthermore, while a KeOps integration into GPyTorch is available, we found that for the SVGP model it would increase the running time, so we did not use it. The trained parameters were the same as for GPflow plus an extra scalar for the GP mean.

For Falkon we tuned the kernel length-scale, number of inducing points and regularization amount (λ). We used a coarse to fine approach to tune the length-scale which provided good results with a limited number of validation runs.

For Logistic Falkon, in addition to the previously mentioned hyperparameters, the regularization path had to be tuned. We found the algorithm not to be very sensitive to the exact regularization path: it is sufficient to set the final λ, and many different paths which lead to such value will return the same results.

Experimental setting

All experiments were run on a Dell PowerEdge server with 2 Intel Xeon 4116 CPUs, 2 Titan Xp GPUs and 256GB of RAM. Since out of the analyzed implementations only Falkon could use both GPUs effectively, we ran it both in a 2-GPU configuration (see Table 3.3) and in a single-GPU configuration (see in appendix Table B.1) where Falkon was on average $1.6\times$ slower. Each experiment was run 5 times, varying the random train/test data split and the inducing points. Out of all possible experiments, we failed to run GPyTorch on TIMIT due to difficulties in setting up a multi-class benchmark (this is not a limitation of the software). Other experiments, such as EigenPro on several larger datasets, failed due to memory errors and others yet due to software limitations in handling sparse inputs (none of the examined implementations could run the sparse YELP dataset). Finally, LogFalkon only makes sense on binary classification datasets. Table 3.2 summarizes the most important hyperparameter settings for each algorithm-dataset pair.

3.4.3 Scalability to large-scale datasets

For the second series of experiments we compared our implementation against the three software packages for GPU-accelerated kernel methods presented above, on several large scale datasets. All experiments were run on the same hardware, with comparable amounts of hyperparameter tuning. Accuracy and timings are shown in Table 3.3. In all cases, our library converges in less time than the other implementations: with an average speedup ranging from $6\times$ when compared to EigenPro to $> 10\times$ when compared to GPyTorch. Only on very few datasets such as AIRLINE-CLS, GPflow gets closer to Falkon's running time. Both models had worse accuracy than Falkon. EigenPro has generally high accuracy but can not handle large datasets at all. Finally, LogFalkon provides a small but consistent accuracy boost on binary classification problems, at the expense of higher running time. Compared with the original Falkon library (Rudi, Carratino, et al., 2017) we report slightly higher error on HIGGS; this is attributable to the use of low-precision floating point numbers. We did not find significant performance differences for other datasets.

3.4.4 Extended comparisons

We also compared the results of our library against a comprehensive list of competing kernel methods, by scanning the literature for results which used similar datasets as the ones we considered and reported both accuracy and running times. This allowed us to confirm that the results reported in our benchmarks for competing methods (see Table 3.3) were in-line with

Table 3.2 Summary of the most important hyperparameter settings. η denotes the learning rate, *subsample* the amount of training-set subsampling that was performed (*i.e.* training on a smaller dataset), and by Newton steps the number of separate runs of the main Falkon algorithm for Logistic Falkon (see Section 3.2.4). σ denotes a different parameterization of the Gaussian kernel's length-scale, which follows $\sigma = \frac{1}{\gamma}$ in Equation (2.24).

		AIRLINE	AIRLINE-CLS	MSD	SUSY	TIMIT	YELP	HIGGS	TAXI
	n	5.93×10^{6}	5.93×10^{6}	5.1×10^{5}	5×10^{6}	1.2×10^{6}	1.6×10^{6}	11×10^{7}	1.15×10^{9}
	d	8	8	90	18	440	6.5×10^{7}	28	9
	labels	reg	2-cls	reg	2-cls	144-cls	reg	2-cls	reg
Falkon	m	1×10^{5}	1×10^{5}	5×10^{4}	3×10^{4}	1×10^{5}	5×10^{4}	1.2×10^{5}	1×10^{5}
	σ	0.9	0.9	7	3	14.5	20	3.8	1
	λ	1×10^{-8}	1×10^{-8}	2×10^{-6}	1×10^{-6}	5×10^{-9}	1×10^{-6}	3×10^{-8}	2×10^{-7}
	epochs	20	10	10	5	5	10	10	7
LogFalkon	m	–	1×10^{5}	–	2×10^{4}	–	–	1×10^{5}	–
	σ	–	0.9	–	3	–	–	5	–
	λ	–	1×10^{-9}	–	1×10^{-8}	–	–	1×10^{-9}	–
	Newt. steps	–	9	–	6	–	–	9	–
GPyTorch	m	2000	2000	3000	2000	–	–	2000	1000
	η	5×10^{-3}	2×10^{-3}	2×10^{-3}	1×10^{-3}	–	–	2×10^{-2}	2×10^{-3}
	epochs	20	20	20	20	–	–	15	5
GPflow	m	2000	2000	3000	2000	2000	–	2000	1000
	η	5×10^{-3}	5×10^{-3}	2×10^{-3}	3×10^{-3}	1×10^{-2}	–	2×10^{-2}	3×10^{-3}
	epochs	25	20	45	10	15	–	60	10
	whiten	no	no	no	no	no	–	yes	no
EigenPro	$\eta\div$	10	12	20	1	1	–	–	–
	subsample	1×10^{6}	1×10^{6}	–	6×10^{5}	–	–	–	–
	epochs	9	10	9	1	4	–	–	–

what had been previously reported. The outcome of this comparison is shown in Table 3.4. We do not report results where running time is not mentioned. Some of the numbers in Table 3.4 have higher accuracy than Falkon: this comes from the use of deep GPs which – through a vast number of parameters – can learn better data representations. Such models are intrinsically different in spirit from kernel methods, and we do not aim to compare with them specifically; they are reported in Table 3.4 for the sake of completeness. We wish to highlight the results on the TAXI dataset, where a distributed GP (H. Peng et al., 2017) ran in 6000 s on a system with 28 000 CPUs, while Falkon achieved similar accuracy in less time, with a much smaller computational budget.

3.4.5 Small-scale benchmarks

In Table 3.5 we compare the running times of Falkon and ThunderSVM (Wen et al., 2018) on three popular image datasets. ThunderSVM was chosen among several SVM implementations as it runs entirely on the GPU, and can thus solve the hinge-loss problem quickly for problems of moderate size. Smaller datasets than the ones used for previous experiments were considered, since ThunderSVM solves the full SVM problem and thus suffers from cubic time scaling. The results obtained show that Falkon can work efficiently even on smaller datasets, resulting between 2 and 10 times faster than ThunderSVM (depending on problem size), with comparable

Table 3.3 Accuracy and running-time comparisons on large scale datasets.

	TAXI $n \approx 10^9$		HIGGS $n \approx 10^7$		YELP $n \approx 10^6, d \approx 10^7$	
	RMSE	time	$1 - \text{AUC}$	time	rel. RMSE	time
Falkon	**311.7±0.1**	**3628±2 s**	0.1804±0.0003	**443±2 s**	**0.810±0.001**	**1008±2 s**
LogFalkon	—		**0.1787±0.0002**	2267±5 s	—	
EigenPro	FAIL		FAIL		FAIL	
GPyTorch	315.0±0.2	37 009±42 s	0.1997±0.0004	2451±13 s	FAIL	
GPflow	313.2±0.1	30 536±63 s	0.1884±0.0003	1174±2 s	FAIL	
	TIMIT $n \approx 10^6$		AIRLINE $n \approx 10^6$		MSD $n \approx 10^5$	
	c-error	time	rel. MSE	time	rel. error	time
Falkon	32.27±0.08 %	**288±3 s**	**0.758±0.005**	**245±5 s**	$(4.4834\pm0.0008)\times10^{-3}$	**62±1 s**
EigenPro	**31.91±0.01 %**	1737±8 s	0.785±0.005	1471±11 s[1]	$\mathbf{(4.4778\pm0.0004)\times10^{-3}}$	378±8 s
GPyTorch	—		0.793±0.005	2069±50 s	$(4.5004\pm0.0010)\times10^{-3}$	502±2 s
GPflow	33.78±0.14 %	2672±10 s	0.782±0.005	1297±2 s	$(4.4986\pm0.0005)\times10^{-3}$	525±5 s
	AIRLINE-CLS $n \approx 10^6$		SUSY $n \approx 10^6$			
	c-error	time	c-error	time		
Falkon	31.5±0.2 %	**186±1 s**	19.67±0.02 %	**22±0 s**		
LogFalkon	**31.3±0.2 %**	1291±3 s	**19.58±0.03 %**	83±1 s		
EigenPro	32.5±0.2 %	1629±1 s[1]	20.08±0.55 %	90±0 s[2]		
GPyTorch	32.5±0.2 %	1436±2 s	19.69±0.03 %	882±9 s		
GPflow	32.3±0.2 %	1039±1 s	19.65±0.03 %	560±11 s		

[1]Using a random subset of 1×10^6 points for training. [2]Using a random subset of 6×10^5 points for training.

accuracy. To further shave off some time, we implemented a version of Falkon which runs entirely inside the GPU: we call this **InCoreFalkon**, and it can only be used on smaller datasets which fit inside the GPU, leaving some space to spare which is used for the preconditioner and other computations. Table 3.5 shows that InCoreFalkon gives a further speed-up of – on average – 2× compared to the standard implementation.

3.4.6 Multi-GPU scalability

In this section we look into the scalability of our implementation across multiple GPUs. Scalability results for the full Falkon algorithm on the TAXI dataset are shown in Figure 3.7. This result depends on scaling both the preconditioner and the conjugate gradient iterations. Multi-device scaling of the preconditioner has already been discussed in Section 3.3 (see Figure 3.4): data dependencies and other inefficiencies prevent a linear scaling with more devices.

Every CG iteration consists of two multiplications between the kernel matrix and an arbitrary vector. Hence, to better isolate performance of this part of the algorithm we will analyze different routines for kernel vector multiplication $k(X^{(1)}, X^{(2)})v$. In particular, given the impressive results achieved by KeOps (see Table 3.1) we wished to compare the pure Python implementation (which leverages PyTorch for GPU computations) with the native CUDA implementation from KeOps (Charlier, Feydy, Glaunès, Collin, et al., 2021). Let the problem dimensions be as follows $X^{(1)} \in \mathbb{R}^{n\times d}, X^{(2)} \in \mathbb{R}^{m\times d}, v \in \mathbb{R}^{m\times 1}$ and $k(\cdot,\cdot)$ be a kernel

Table 3.4 Survey of accuracy and run-time results on the considered datasets as reported in the literature. We report the result of our implementation (Falkon) next to other implementations, along with the time taken and the hardware used (where available).

Dataset	Falkon		Other methods		
	error	time	error	time	reference
TAXI (metric: RMSE)	311.7±0.1	3628±2 s	309.7	6000 s 28 000 vCPUs (AWS)	ADVGP (H. Peng et al., 2017)
HIGGS (metric: c-err)	25.78±0.03 %	443±2 s	32.87 %	1392 s on 14 node cluster	liquidSVM (Steinwart and Thomann, 2017)
YELP (metric: RMSE)	0.810±0.001	1008±2 s	0.861	≈ 3500 s	Nyström (Tu et al., 2016)
			0.854	≈ 30 000 s on 128 machines (AWS)	Full linear kernel (Tu et al., 2016)
AIRLINE (metric: MSE)	0.758±0.005	245±5 s	0.827±0.004	265±6 s on a laptop	VFF-GP (Hensman, Durrande, et al., 2017)
			0.791±0.005	18 360±360 s on a cluster	SVIGP (Hensman, Durrande, et al., 2017)
MSD (metric: rel. err.)	4.48×10^{-3}	62±1 s	$\approx 4.55\times10^{-3}$	210 s on IBM POWER8	Hierarchical (J. Chen et al., 2017)
			4.58×10^{-3}	289 s on 8 r3.8xlarge (AWS)	Faster KRR (Avron et al., 2017)
AIRLINE-CLS (metric: AUC)	0.739±0.002	**186±1 s**	0.781±0.001	14 328 s	Varitional Deep GP (Wilson, Hu, et al., 2016)
			0.694	5200 s	TT-GP (Izmailov et al., 2018)
			0.788	1375 s	Deep TT-GP (Izmailov et al., 2018)
			0.665	80 000 s	cVGP(Hensman, A. G. Matthews, et al., 2015)
			0.785	≈ 5000 s	RF Deep GPs (Cutajar, Bonilla, et al., 2017)
SUSY (metric: c-err)	19.67±0.02 %	**22±0 s**	≈ 20%	≈ 2000 s on IBM POWER8	Hierarchical (J. Chen et al., 2017)
			19.8%	58 s on 1 Titan Xp	EigenPro 2.0 (Ma et al., 2019)

Table 3.5 Comparing the running times of Falkon, the in-core version of Falkon and ThunderSVM on three image datasets. Hyperparameters (especially the number of inducing points m) were tuned so that the two algorithms obtained approximately the same accuracy.

	MNIST $n = 6 \cdot 10^4, d = 780$	CIFAR10 $n = 6 \cdot 10^4, d = 1024$	SVHN $n = 7 \cdot 10^4, d = 1024$
Falkon	10.9 s	13.7 s	17.2 s
InCoreFalkon	6.5 s	7.9 s	6.7 s
ThunderSVM	19.6 s	82.9 s	166.4 s

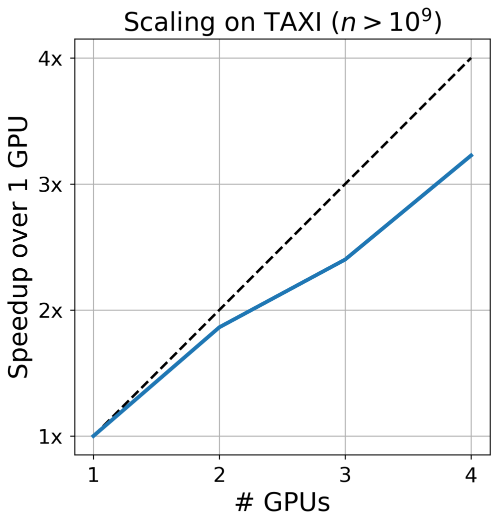
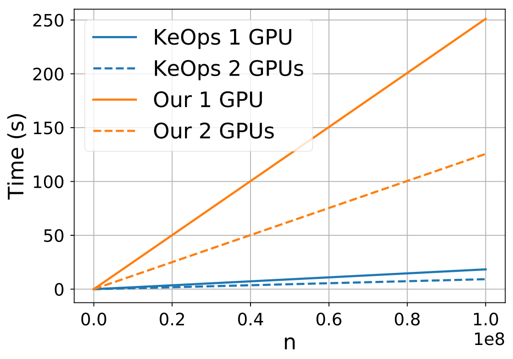


Figure 3.7 Multi-GPU scalability of Falkon on the TAXI dataset (settings are the same as per Table 3.2). Falkon scales remarkably well, with even 4 GPUs.

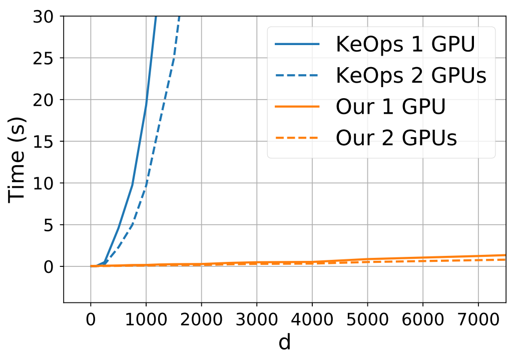
function. In Figure 3.8 we observed two different scaling outcomes: increasing the number of data points n results in linear scaling across both implementations, with KeOps being approximately 10× faster than our implementation (see Figure 3.8(a)). Increasing the data dimension d the PyTorch implementation scales linearly, but KeOps scales polynomially. From Figure 3.8(b) it is clear that KeOps cannot be used with high-dimensional data. In our final algorithm we set a threshold on the data dimensionality and switch implementation based on this. Finally note that this operation scales almost perfectly with multiple GPUs.

3.5 Conclusions

Making flexible and easy to use machine learning libraries available is one of the keys of the recent success of machine learning. Here, we contribute to this effort by developing a library for large scale kernel methods. We translate algorithmic ideas into practical solutions, using a number of carefully designed computational approaches specifically adapted to GPU



(a)



(b)

Figure 3.8 Scaling of matrix-vector implementations where the matrix is the Gaussian kernel. In (a) we have set $m = 20\,000$, $d = 10$ and n is variable; in (b) we set $m = n = 20\,000$ and we vary d. All experiments are run on 1 and 2 GPUs on single precision random data.

architectures. The resulting library achieves excellent performance both in terms of accuracy and computational cost. A number of further developments are possible building on our work. For example, considering other loss functions or optimization approaches, and especially more structured kernels (J. Chen et al., 2017) that could further improve efficiency.

Chapter 4

Hyperparameter Optimization in KRR

Learning from finite data requires fitting models of varying complexity to the available training set. Too complex a model will overfit and fail to generalize to unseen examples, while an exceedingly simple one will not learn the complex relations present in the data. The problem of finding the model with the right complexity is referred to as model selection in statistics and more broadly as hyperparameter tuning in machine learning. It is a classical problem, known to be of utmost importance for machine learning algorithms to perform well in practice. As such, much has been written about it (Hastie et al., 2009), including a number of theoretical results (Tsybakov, 2003; Arlot, 2007; Massart, 2007). Hyperparameter (HP) tuning is also at the core of recent trends such as neural architecture search (Elsken et al., 2019) or AutoML (Hutter et al., 2019). In this paper, we consider the question of hyperparameter tuning in the context of kernel methods and specifically kernel ridge regression (KRR) (Smola et al., 2000). We have shown in Chapter 3 that KRR can be scaled to massive data-sets using approximate solvers, which take advantage of a number of ideas from optimization (Boyd et al., 2004) and randomized algorithms (El Alaoui et al., 2015), and exploit parallel computations with GPUs. While this solution opens up new possibilities for applying kernel methods, hyperparameter tuning is notably missing, ultimately hindering its practical use and efficiency. Indeed, available solutions which provide hyperparameter tuning are either limited to small data, or are restricted to very few hyperparameters (Steinwart and Thomann, 2017; Pedregosa et al., 2011; Suykens et al., 2002).

In this chapter we work to fill in this gap, considering approximate solvers based on the Nyström approximation and working towards the automated tuning of regularization and kernel parameters, as well as of the Nyström centers. On the one hand, we review and compare empirically a number of hyperparameter tuning strategies, while discussing their basic theoretical guarantees. On the other hand we propose, and provide an efficient implementation for, a novel criterion inspired by complexity regularization (Bartlett, Boucheron, et al., 2002) and based on a data-dependent bound. This bound treats separately the sources of variance due to the stochastic nature of the data and to the Nyström approximation. In practice, this results in better stability properties of the corresponding tuning strategy. As a byproduct of our analysis we complement an existing library for large-scale kernel methods with the possibility to adaptively tune a large number of hyperparameters. Code is available at the following address: https://github.com/falkonml/falkon.

In Section 4.1 the basics of hyperparameter tuning are introduced. In Section 4.2 we propose our new criterion, and discuss its efficient implementation in Section 4.3. In Section 4.4 we conduct a thorough experimental study and finally, in Section 4.5 the full derivation of the derived criterion is provided and finally, in Section 4.6 we provide some concluding remarks.

4.1 Background on hyperparameter optimization

We begin by briefly introducing the problem of learning a model's parameters, which naturally leads to learning the hyperparameters, and then discuss various objective functions and optimization algorithms which have been proposed for the task.

4.1.1 Parameter and hyperparameter learning

Assume we are given a set of measurements $\{(x_i, y_i)\}_{i=1}^n \subset \mathcal{X} \times \mathcal{Y}$ related to each other by an unknown function $f^* : \mathcal{X} \to \mathcal{Y}$ and corrupted by some random noise ϵ_i with variance σ^2 for each $i = 1, \dots, n$.

$$y_i = f^*(x_i) + \epsilon_i. \tag{4.1}$$

We wish to approximate the target function f^* using a model $f : \mathcal{X} \to \mathcal{Y}$ defined by a set of *parameters* which must be learned from the limited measurements at our disposal. In order for the learning procedure to succeed, as we have seen in Chapter 2, one often assumes that f belongs to some hypothesis space $\mathcal{F}$, and this space typically depends on additional *hyperparameters* θ. Assume we are given a loss function $\ell : \mathcal{Y} \times \mathbb{R} \to [0, \infty)$; we can learn a model by fixing the hyperparameters θ and minimizing the loss over the available training samples:

$$\hat{f}_\theta = \operatorname*{arg\,min}_{f \in \mathcal{F}_\theta} \sum_{i=1}^n \ell(y_i, f(x_i))$$

In this paper we are concerned with kernel ridge regression: a specific kind of model where the loss function is the squared loss $\ell(y, a) = \|y - a\|^2$ and the hypothesis space is a reproducing kernel Hilbert space (RKHS) $\mathcal{H}$. Associated to $\mathcal{H}$ is a kernel function $k_\gamma : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ which depends on hyperparameters γ. To ensure that the minimization problem is well defined we must add a regularization term controlled by another hyperparameter λ:

$$\hat{f}_{\lambda,\gamma} = \arg\min_{f \in \mathcal{H}} \sum_{i=1}^{n} \|f(x_i) - y_i\|^2 + \lambda \|f\|_{\mathcal{H}}^2.$$

We have previously seen in Chapter 2 that the solution to this minimization problem is unique, but very expensive to compute requiring $\mathcal{O}(n^3)$ operations and $\mathcal{O}(n^2)$ memory. An approximation to KRR considers a lower-dimensional subspace $\mathcal{H}_m \subset \mathcal{H}$ as hypothesis space, where $\mathcal{H}_m$ is defined from $m \ll n$ points $Z = \{z_j\}_{j=1}^{m} \subset \mathcal{X}$ (Williams et al., 2001) such that

$$\mathcal{H}_m = \operatorname{span}\{\phi(z_1), \dots, \phi(z_m)\} \tag{4.2}$$

with ϕ the feature map for which $k_\gamma(x, x') = \langle \phi(x), \phi(x') \rangle$. While in the previous chapters we have only considered taking the inducing points Z (also known as Nyström centers) from the training set, it is also possible to view the points in Z as hyperparameters which determine the hypothesis space. In fact this is common practice in sparse Gaussian Processes (GPs), and in Section 3.4 sparse GPs were shown to achieve similar accuracy to Nyström KRR models with up to one hundred times fewer inducing points (albeit with lower efficiency). This large reduction is possible thanks to the greater flexibility afforded by freeing the inducing points z from being tied to the training points (Titsias, 2009; Hensman, Fusi, et al., 2013; Hensman, A. G. Matthews, et al., 2015).

Following Equation (2.92), but keeping track of dependencies on different hyperparameters, we have the regularized ERM solution

$$\hat{f}_{\lambda,Z,\gamma} = \sum_{i=1}^{m} (\widetilde{\beta}_\lambda)_i k_\gamma(\cdot, z_i), \quad \text{with} \;\; \widetilde{\beta}_\lambda = (\boldsymbol{K}_{nm}^\top \boldsymbol{K}_{nm} + \lambda n \boldsymbol{K}_{mm})^{-1} \boldsymbol{K}_{nm}^\top \hat{y} \tag{4.3}$$

with $(\boldsymbol{K}_{nm})_{i,j} = k_\gamma(x_i, z_j)$ and $(\boldsymbol{K}_{mm})_{i,j} = k_\gamma(z_i, z_j)$. The Nyström KRR (N-KRR) model reduces the computational cost of finding the coefficients to $\mathcal{O}(n\sqrt{n}\log n)$ when using efficient solvers (Rudi, Carratino, et al., 2017; Meanti, Carratino, Rosasco, et al., 2020; Ma et al., 2019), see Chapter 3.

The ideal goal of hyperparameter optimization is to find a set of hyperparameters θ^* (where the space of θ can be, for example, $\theta = (\lambda, Z, \gamma)$) for which $\hat{f}_{\theta^*}$ minimizes the test error (over all unseen samples). By definition we cannot actually evaluate the test error: we can only use the available data points. Naively one could think of minimizing the training error instead, but such a scheme inevitably chooses overly complex models which overfit the training set. Instead it is necessary to minimize a data-dependent criterion $\mathcal{C}$

$$\widehat{\theta} = \arg\min_{\theta} \mathcal{C}(\hat{f}_\theta)$$

such that model complexity is penalized. In practice a $\mathcal{C}$ is commonly chosen in such a way that its expectation (with respect to the sampling of the data) is be equal to, or an upper bound of the test error. In the next section we will look at several instances of penalized objectives $\mathcal{C}$ which appear in the literature and can be readily applied to N-KRR.

4.1.2 Objective functions

Validation error Possibly the most common procedure for HP tuning is to split the available n training samples into two parts: a training set of size n_{tr} and a validation set of size n_{val}. The first is used to learn a model $\hat{f}_\theta$ with fixed hyperparameters θ, while the validation set is used to estimate the performance of different HP configurations.

$$\mathcal{C}^{\text{Val}}(\hat{f}_\theta) = \frac{1}{n_{\text{val}}} \sum_{i=1}^{n_{\text{val}}} \|\hat{f}_\theta(x_i^{\text{val}}) - y_i^{\text{val}}\|^2 \tag{4.4}$$

By using independent datasets for model training and HP selection, $\mathcal{C}^{\text{Val}}$ becomes an unbiased estimator of the test error and it can be proven that its minimizer is close to θ^* under certain assumptions (Arlot and Bach, 2009). However, since $\hat{f}_\theta$ has been trained with $n_{\text{tr}} < n$ samples, there is a small bias in the chosen hyperparameters (Varma et al., 2006). Furthermore the variance of the hold-out estimator is typically very high as it depends on a specific data split. Two popular alternatives which address this latter point are k-fold cross-validation (CV) which takes an average over k HP estimates, each obtained by using a different train/validation split, and leave-one-out CV (LOOCV).

Leave-one-out CV and Generalized CV The LOOCV estimator is an average of the n estimators trained on all $n-1$ sized subsets of the training set and evaluated on the single left out sample. The result is an almost unbiased estimate of the expected risk on the full dataset (Vapnik, 1998). For linear models a computational shortcut allows to compute the LOOCV estimator by training a single model on the whole dataset instead of n different ones (Cawley et al., 2004). In particular in the case of N-KRR we can consider

$$\mathcal{C}^{\text{LOOCV}}(\hat{f}_\theta) = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{y_i - \hat{f}_\theta(x_i)}{1 - H_{ii}} \right)^2, \tag{4.5}$$

where H is the *hat* matrix which maps response values $\hat{y}$ to fitted values (model predictions): $H = \boldsymbol{K}_{nm}(\boldsymbol{K}_{nm}^\top \boldsymbol{K}_{nm} + \lambda n \boldsymbol{K}_{mm})^{-1} \boldsymbol{K}_{nm}$.

GCV is an approach proposed in Golub et al. (1979) to further improve leave-one-out cross-validation's computational efficiency and to make it invariant to data rotations:

$$\mathcal{C}^{\text{GCV}}(\hat{f}_\theta) = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{y_i - \hat{f}_\theta(x_i)}{\frac{1}{n}\operatorname{Tr}(\boldsymbol{I} - H)} \right)^2. \tag{4.6}$$

Cao et al. (2006) proved an oracle inequality which guarantees that the GCV estimator converges to the neighborhood of θ^* when estimating the λ hyerparameter in kernel ridge regression.

Complexity regularization Complexity regularization, or covariance penalties Mallows, 1973; Efron, 2004 are a general framework for expressing objective functions as the empirical error plus a penalty term to avoid overly complex models. For linear models the trace of the hat matrix acts as penalty against complexity. Applying this principle to N-KRR gives the objective

$$\mathcal{C}^{\text{C}-\text{Reg}}(\hat{f}_{\lambda,Z,\gamma}) = \frac{1}{n}\|\hat{f}_{\lambda,Z,\gamma}(X) - \hat{y}\|^2 + \frac{2\sigma^2}{n}\operatorname{Tr}((\widetilde{\boldsymbol{K}} + n\lambda\boldsymbol{I})^{-1}\widetilde{\boldsymbol{K}}) \tag{4.7}$$

where $\widetilde{\boldsymbol{K}} = \boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^{\dagger}\boldsymbol{K}_{nm}^{\top}$ (the Nyström kernel), and $A^{\dagger}$ denotes the Moore-Penrose inverse of matrix A. The first term can be interpreted as a proxy for the bias of the model, and the second as a variance estimate. For estimating λ in (N-)KRR, Arlot and Bach (2009) proved an oracle inequality if a precise estimate of the noise σ^2 is available.

Sparse GP Regression (Titsias, 2009) A different approach comes from the Bayesian perspective, where the equivalent of KRR is Gaussian Process Regression (GPR). Instead of estimating the test error, HP configurations are scored based on the "probability of a model given the data" (Rasmussen et al., 2005). A fully Bayesian treatment of the hyperparameters allows to explicitly write down their posterior distribution, from which the HP likelihood has the same form of the marginal likelihood in the denominator of the model parameters' posterior distribution. Hence maximizing the (log) marginal likelihood (MLL) with gradient-based methods is common practice in GPR.

Like with N-KRR, inducing points are used in GPR to reduce the computational cost, giving rise to models such as SoR, DTC, FiTC (Quiñonero-Candela et al., 2005). Here we consider the SGPR model proposed in Titsias (2009) which treats the inducing points as variational parameters, and optimizes them along with the other HPs by maximizing a lower bound to the MLL. The objective to be minimized is

$$\mathcal{C}^{\text{SGPR}}(\hat{f}_{\lambda,Z,\gamma}) = \log\left|\widetilde{\boldsymbol{K}} + n\lambda\boldsymbol{I}\right| + \hat{y}^{\top}(\widetilde{\boldsymbol{K}} + n\lambda\boldsymbol{I})^{-1}\hat{y} + \frac{1}{n\lambda}\operatorname{Tr}(\boldsymbol{K} - \widetilde{\boldsymbol{K}}). \tag{4.8}$$

The first term of Equation (4.8) penalizes complex models, the second pushes towards fitting the training set well, and the last term measures how well the inducing points approximate the full training set. Recently the approximate MLL was shown to converge to its exact counterpart (Burt et al., 2020), but we note that this does not guarantee convergence to the optimal hyperparameters.

4.1.3 Optimization algorithms

In this section we describe three general approaches which can be used to minimize the objectives introduced above.

Grid search In settings with few hyperparameters the most widely used optimization algorithm is grid-search which tries all possible combinations from a predefined set, choosing the one with the lowest objective value at the end. Random search (Bergstra et al., 2012) and adaptive grid search (used for SVMs in Steinwart and Thomann (2017)) improve on this basic idea, but they also become prohibitively costly with more than ≈ 5 HPs as the number of combinations which need to be tested grows exponentially with the number of dimensions.

Black-box optimization A more sophisticated way to approach the problem is to take advantage of regularities (and in particular smoothness) in the objective. Sequential model-based optimization (SMBO) algorithms (Brochu et al., 2010; Snoek et al., 2012; Shahriari et al., 2016) take evaluations of the objective function as input, and fit a Bayesian *surrogate* model to such values. The surrogate can then be cheaply evaluated on the whole HP space to suggest the most promising HP values to explore. These algorithms do not rely on gradient information so they don't require the objective to be differentiable and can be applied to optimization of discrete HPs. However, while more scalable than grid search, black-box algorithms become very inefficient in high (*i.e.* > 100) dimensions.

Gradient-based methods Scaling up to even larger hyperparameter spaces requires exploiting the objective's local curvature. While the optimization problem is typically non-convex, gradient descent will usually reach a good local minimum. When the objective can be decomposed as a sum over the data-points stochastic gradient descent (SGD) can be used, which may provide computational benefits (*e.g.* the SVGP objective (Hensman, Fusi, et al., 2013) is optimized in mini-batches with SGD-like algorithms). In the context of KRR, gradient-based methods have been successfully used for HP optimization with different objective functions (Seeger, 2008; Keerthi et al., 2007). Recent extensions to gradient-based methods have been proposed for those cases when the trained model cannot be written in closed form. Either by unrolling the iterative optimization algorithm (Maclaurin et al., 2015; Franceschi et al., 2017; Grazzi et al., 2020), or by taking the model at convergence with the help of the implicit function theorem (Pedregosa, 2016; Rajeswaran et al., 2019), it is then possible to differentiate a simple objective (typically a hold-out error) through the implicitly defined trained model. This has proven to be especially useful for deep neural nets (Lorraine et al., 2020), but is unnecessary for N-KRR where the trained model can be easily written in closed form.

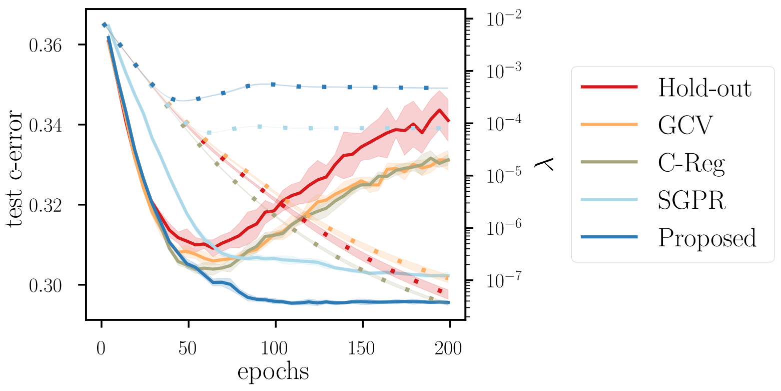


Figure 4.1 Test-error and penalty (λ) as a function of optimization epoch on the small-HIGGS dataset. $m = 100$ centers, d length-scales and λ were optimized with equal initial conditions. The three unbiased proxy functions lead to overfitting, while SGPR and the proposed objective do not.

4.2 Hyperparameter optimization for Nyström-KRR

The objectives introduced in the previous section can be applied to HP tuning for kernel methods. Always keeping in mind efficiency but also usability, our goal is to come up with an objective and associated optimization algorithm which: (a) can be used to tune the hyperparameters of Nyström kernel ridge regression including the inducing points and (b) can be computed efficiently, even for large scale problems.

To satisfy the first point, an algorithm of the first-order is needed since the inducing points are typically between a hundred and a few thousands (each point being of the same dimension as the data). Regarding the second point we found empirically that the unbiased objectives are prone to overfitting on certain datasets. An example of this behavior is shown in Figure 4.1 on a small subset of the HIGGS dataset (see Chapter A for a description of the datasets used) The first three objectives (Hold-out, GCV and C-Reg) are unbiased estimates of the test error, hence it is their variance which causes overfitting. To mitigate such possibility in our objective we may look into the different sources of variance: *hold-out* depends strongly on which part of the training set is picked for validation, *GCV* and *C-Reg* don't rely on data splitting but still suffer from high variance due to the random initial choice of inducing points.

We set out to devise a new objective in the spirit of complexity regularization, which is an upper bound on the test error. A biased estimate – which is therefore over-penalizing – will be more resistant to noise than an unbiased one (as was noted in Arlot (2007)), and we tailor our objective specifically to N-KRR in order to explicitly take into account the variance from inducing point selection.

We base our analysis of the N-KRR error in the fixed design setting, where the points $x_i \in \mathcal{X},\ i = (1, \dots, n)$ are assumed to be fixed, and the stochasticity comes from i.i.d. random noise variables $\epsilon_i, \dots, \epsilon_n$ such that

$$\mathbb{E}[\epsilon_i] = 0 \text{ and } \mathbb{E}[\epsilon_i^\top \epsilon_i] = \sigma^2. \tag{4.9}$$

Denote the empirical error of an estimator $f \in \mathcal{H}$ as $\hat{\mathcal{E}}(f) = n^{-1}\|f(X) - \hat{y}\|^2$ and the "expected" error as $\mathcal{E}(f) = n^{-1}\|f(X) - f^*(X)\|^2$, where f^* is the noiseless target function from Equation (4.1). Note that this is different from the random-design setting introduced in Chapter 2 where the data-points (x_i, y_i) are sampled from an unknown probability distribution. Consider inducing points z_j and a subspace of $\mathcal{H}$:

$$\mathcal{H}_m = \text{span}\{k_\gamma(z_1, \cdot), \dots, k_\gamma(z_m, \cdot)\}, \quad m \ll n, \tag{4.10}$$

and let P be the orthogonal projection with range $\mathcal{H}_m$. Further denote the regularized empirical risk as $\hat{\mathcal{E}}_\lambda(f) = \hat{\mathcal{E}}(f) + \lambda\|f\|^2_{\mathcal{H}}$,

Assessing a particular hyperparameter configuration $\theta = (\lambda, Z, \gamma)$ requires estimating the expected test error at the empirical risk minimizer $\hat{f}_{\lambda,Z,\gamma}$ trained with that configuration; the optimal HPs are then found by $(\lambda, Z, \gamma)^* = \arg\min_{(\lambda,Z,\gamma)} \mathcal{E}(\hat{f}_{\lambda,Z,\gamma})$. The following theorem gives an upper bound on the ideal objective; a full proof is provided in Section 4.5.

Theorem 4.1

Under the assumptions of fixed-design regression we have that,

$$\begin{aligned}\mathbb{E}[\mathcal{E}(\hat{f}_{\lambda,Z,\gamma})] \leq & \frac{2\sigma^2}{n} \operatorname{Tr}((\widetilde{\boldsymbol{K}} + n\lambda\boldsymbol{I})^{-1}\widetilde{\boldsymbol{K}}) \\ & + \frac{2}{n\lambda} \operatorname{Tr}(\boldsymbol{I} - \widetilde{\boldsymbol{K}})\,\mathbb{E}[\hat{\mathcal{E}}_\lambda(f_{\lambda,\gamma})] \\ & + 2\,\mathbb{E}[\hat{\mathcal{E}}_\lambda(f_{\lambda,\gamma})]\end{aligned} \tag{4.11}$$

Proof of Theorem 4.1: *Proof-sketch.*
We decompose the test error expectation in the following manner

$$\begin{aligned}\mathbb{E}[\mathcal{E}(\hat{f}_{\lambda,Z,\gamma})] \leq \mathbb{E}\Big[& \underbrace{\mathcal{E}(\hat{f}_{\lambda,Z,\gamma}) - \hat{\mathcal{E}}(\hat{f}_{\lambda,Z,\gamma})}_{①} \\ & + \underbrace{\hat{\mathcal{E}}(\hat{f}_{\lambda,Z,\gamma}) + \lambda\|\hat{f}_{\lambda,Z,\gamma}\|^2_{\mathcal{H}} - \hat{\mathcal{E}}_\lambda(Pf_{\lambda,\gamma})}_{②} + \underbrace{\hat{\mathcal{E}}_\lambda(f_{\lambda,\gamma})}_{③}\Big]\end{aligned}$$

by adding and subtracting $\hat{\mathcal{E}}(\hat{f}_{\lambda,Z,\gamma})$, $\hat{\mathcal{E}}_\lambda(Pf_{\lambda,\gamma})$ and summing the positive quantity $\lambda\|\hat{f}_{\lambda,Z,\gamma}\|^2_{\mathcal{H}}$. Since $\hat{f}_{\lambda,Z,\gamma}$ is the minimizer of $\hat{\mathcal{E}}(\hat{f}_{\lambda,Z,\gamma}) + \lambda\|(\|^2_{\mathcal{H}}\hat{f}_{\lambda,Z,\gamma})$ in the space $\mathcal{H}_m$ and since $Pf_{\lambda,\gamma} \in \mathcal{H}_m$, the second term is negative and can be discarded.

Term ① is the variance of N-KRR and can be computed exactly by noting that

$$\begin{aligned}\mathbb{E}[\hat{\mathcal{E}}(\hat{f}_{\lambda,Z,\gamma})] &= \mathbb{E}[n^{-1}\|\hat{f}_{\lambda,Z,\gamma}(X) - f^*(X) - \epsilon\|^2] \\ &= \mathbb{E}[\mathcal{E}(\hat{f}_{\lambda,Z,\gamma})] + \sigma^2 - \frac{2}{n}\mathbb{E}[\left\langle \hat{f}_{\lambda,Z,\gamma}(X) - f^*(X), \epsilon \right\rangle]\end{aligned}$$

where the first part cancels and we can ignore σ^2 which is fixed and positive. Expanding the inner product and taking its expectation we are left with

$$\frac{2}{n}\mathbb{E}[\left\langle \hat{f}_{\lambda,Z,\gamma}(X) - f^*(X), \epsilon \right\rangle] = \frac{2\sigma^2}{n}\operatorname{Tr}((\widetilde{\boldsymbol{K}} + n\lambda \boldsymbol{I})^{-1}\widetilde{\boldsymbol{K}})$$

which is the *effective dimension* or the *degrees of freedom* of the hypothesis space $\mathcal{H}_m$, times the noise variance σ^2.

Term ③ takes into account the difference between estimators in $\mathcal{H}$ and in $\mathcal{H}_m$. We begin by upper-bounding the regularized empirical error of $Pf_{\lambda,\gamma}$ with a first part containing the projection operator and a second term without P

$$\mathbb{E}[\hat{\mathcal{E}}(Pf_{\lambda,\gamma}) + \lambda\|Pf_{\lambda,\gamma}\|^2_{\mathcal{H}}] \le \mathbb{E}[\frac{2}{n}\|\boldsymbol{K}^{1/2}(\boldsymbol{I} - P)\|^2\|f_{\lambda,\gamma}\|^2 + 2\hat{\mathcal{E}}_\lambda(f_{\lambda,\gamma})].$$

Now $\|\boldsymbol{K}^{1/2}(\boldsymbol{I} - P)\|^2 \le \operatorname{Tr}\left(\boldsymbol{K} - \widetilde{\boldsymbol{K}}\right)$ the difference between full and approximate kernels, and $\|f_{\lambda,\gamma}\|^2 \le \lambda^{-1}\hat{\mathcal{E}}_\lambda(f_{\lambda,\gamma})$ which leads us to the desired upper bound. □

We now make two remarks on computing Equation (4.11).

Remark 4.1 (Computing $\mathbb{E}[\hat{\mathcal{E}}_\lambda(f_{\lambda,\gamma})]$): In the spirit of complexity regularization we can approximate this bias term by the empirical risk of N-KRR $\hat{\mathcal{E}}_\lambda(\hat{f}_{\lambda,Z,\gamma})$, so that the final objective will consist of a data-fit term plus two complexity terms: the effective dimension and the Nyström approximation error.

Remark 4.2 (Estimating σ^2): Once again following the principle of over-penalizing rather than risking to overfit, we note that in binary classification the variance of $\hat{y}$ is capped at 1 for numerical reasons, while for regression we can preprocess the data dividing $\hat{y}$ by its standard deviation. Then according to Equation (4.1) we must have that the label standard deviation is greater than the noise standard deviation hence $\hat{\sigma}^2 = 1 \ge \sigma^2$.

Our final objective then has a form which we can compute efficiently

$$\begin{aligned}\mathcal{C}^{\text{Prop}} =& \frac{2}{n}\operatorname{Tr}((\widetilde{\boldsymbol{K}} + n\lambda\boldsymbol{I})^{-1}\widetilde{\boldsymbol{K}}) \\ &+ \frac{2}{n\lambda}\operatorname{Tr}(\boldsymbol{K} - \widetilde{\boldsymbol{K}})\hat{\mathcal{E}}_\lambda(\hat{f}_{\lambda,Z,\gamma}) \\ &+ \frac{2}{n}\|\hat{f}_{\lambda,Z,\gamma}(X) - \hat{y}\|^2 + \lambda\|\hat{f}_{\lambda,Z,\gamma}\|^2_{\mathcal{H}}. \end{aligned} \tag{4.12}$$

We make two further remarks on the connections to the objectives of Section 4.1.2.

Remark 4.3 (Similarities with complexity regularization): $\mathcal{C}^{\text{Prop}}$ has a similar form to Equation (4.7) with an extra term which corresponds to the variance introduced by the Nyström centers which we were aiming for (up to multiplication by the KRR bias).

Remark 4.4 (Similarities with SGPR): Equation (4.12) shares many similarities with the SGPR objective: the log-determinant is replaced by the model's effective dimension – another measure of model complexity – and the term $\operatorname{Tr}\left(\boldsymbol{K} - \widetilde{\boldsymbol{K}}\right)$ is present in both objectives. Furthermore the data-fit term in $\mathcal{C}^{\text{SGPR}}$ is

$$\begin{aligned}\hat{y}^\top(\widetilde{\boldsymbol{K}} + n\lambda\boldsymbol{I})^{-1}\hat{y} &= \frac{1}{\lambda}(n^{-1}\|\hat{f}_{\lambda,Z,\gamma}(X) - \hat{y}\|^2 + \lambda\|\hat{f}_{\lambda,Z,\gamma}\|^2_{\mathcal{H}}) \\ &= \frac{1}{\lambda}\hat{\mathcal{E}}_\lambda(\hat{f}_{\lambda,Z,\gamma})\end{aligned}$$

which is the same as in the proposed objective up to a factor λ^{-1}.

4.3 Scalable approximations

Some practical considerations are needed to apply the objective of Equation (4.12) to large-scale datasets – for which direct computation is not possible due to space or time constraints. We examine the terms comprising $\mathcal{C}^{\text{Prop}}$ and discuss their efficient computations. In Figure 4.2, we verify that the resulting approximation is close to the exact objective.

Starting with the last part of the optimization objective (the one which measures data-fit) we have that

$$\|\hat{f}_{\lambda,Z,\gamma}(X) - \hat{y}\|^2 + \lambda\|\hat{f}_{\lambda,Z,\gamma}\|^2_{\mathcal{H}} = Y^\top(I - \underbrace{\boldsymbol{K}_{nm}(\overbrace{\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm} + n\lambda\boldsymbol{K}_{mm}}^{B})^{-1}\boldsymbol{K}_{nm}^\top)Y}_{=\hat{f}_{\lambda,Z,\gamma}(X)}$$

which can be computed quickly using a fast, memory-efficient N-KRR solver such as Falkon (Meanti, Carratino, Rosasco, et al., 2020) or EigenPro (Ma et al., 2019). However we must also compute the objective's gradients with respect to all HPs, and since efficient solvers proceed by iterative minimization, such gradients cannot be trivially computed using automatic differentiation,

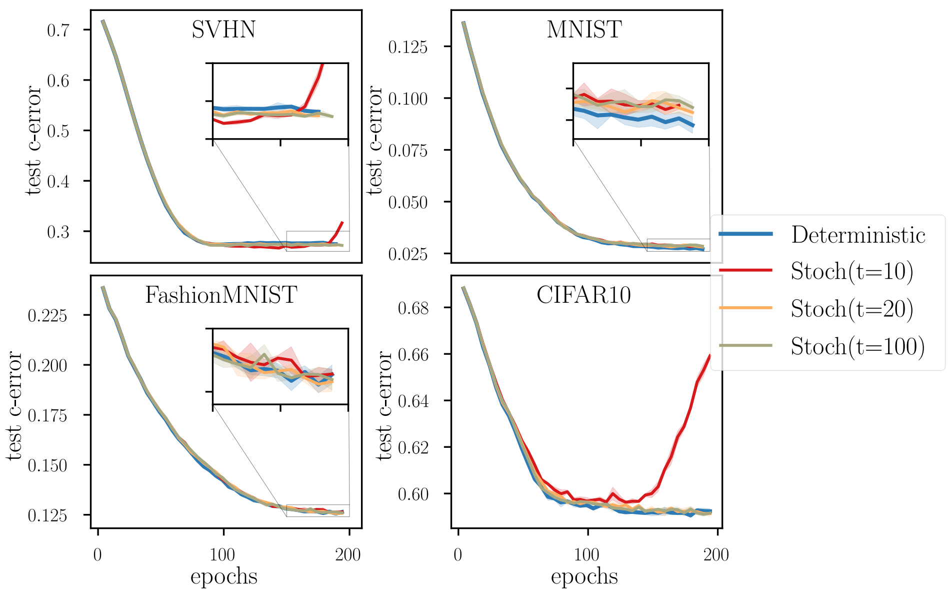


Figure 4.2 The effect of stochastic trace estimation. We plot the optimization curves of the exact objective $\mathcal{C}^{\text{Prop}}$ (*Deterministic*) and the approximated objectives with 10, 20 and 100 STE vectors. On the four datasets we optimized $m = 200$ centers, λ and γ.

indeed, it would be in principle possible to unroll the optimization loops and differentiate through them, the memory requirements for this operation would be too high for large datasets.

Efficient gradients A solution to compute the gradients efficiently is to apply the chain rule by hand until they can be expressed in terms of matrix vector products $(\nabla \ker)\boldsymbol{v}$ with ker any kernel matrix (*i.e.* $\boldsymbol{K}_{nm}$ or $\boldsymbol{K}_{mm}$) and v a vector. As an example the gradient of the data-fit term is

$$\nabla(\hat{y}^\top \boldsymbol{K}_{nm} B^{-1} \boldsymbol{K}_{nm}^\top \hat{y}) = 2\hat{y}^\top (\nabla \boldsymbol{K}_{nm}) B^{-1} \boldsymbol{K}_{nm}^\top \hat{y} - \hat{y}^\top \boldsymbol{K}_{nm} B^{-1} (\nabla B) B^{-1} \boldsymbol{K}_{nm}^\top \hat{y}$$

where we can obtain all $B^{-1}\boldsymbol{K}_{nm}^\top \hat{y}$ vectors via a non-differentiable N-KRR solver, and multiply them by the (differentiable) kernel matrices for which gradients are required. Computing these elementary operations is efficient, with essentially the same cost as the forward pass $\ker v$, and can be done row-wise over ker. Block-wise computations are essential for low memory usage since kernel matrices tend to be huge but kernel-vector products are small, and they allow trivial parallelization across compute units (CPU cores or GPUs). In many cases these operations can also be accelerated using KeOps (Charlier, Feydy, Glaunès, Collin, et al., 2021).

The remaining two terms of Equation (4.12) are harder to compute. Note that in $\text{Tr}\left(\boldsymbol{K} - \widetilde{\boldsymbol{K}}\right)$ we can often ignore $\text{Tr}(\boldsymbol{K})$ since common kernel functions are trivial when computed between a point and itself, but more in general it only requires evaluating the kernel function n times.

We thus focus on

$$\mathrm{Tr}(\widetilde{\boldsymbol{K}}) = \mathrm{Tr}(\boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^{\dagger}\boldsymbol{K}_{nm}^{\top}) \tag{4.13}$$

and on the effective dimension

$$\mathrm{Tr}((\widetilde{\boldsymbol{K}} + \lambda\boldsymbol{I})^{-1}\widetilde{\boldsymbol{K}}) = \mathrm{Tr}(\boldsymbol{K}_{nm}B^{-1}\boldsymbol{K}_{nm}^{\top}). \tag{4.14}$$

Both these terms are traces of huge $n \times n$ matrices. By their symmetry we can express them as squared norms reducing the space requirements to $n \times m$, but they still remain slow to compute: just the $\boldsymbol{K}_{nm}^{\top}\boldsymbol{K}_{nm}$ term costs more than training a N-KRR model with the Falkon solver.

Trace estimation A simple approximation can vastly improve the efficiency of computing Equations (4.13) and (4.14) and their gradients: stochastic trace estimation (STE). The Hutchinson estimator (Hutchinson, 1990) approximates $\mathrm{Tr}(A)$ by $\frac{1}{t}\sum_{i=1}^{t} r_i^{\top}Ar_i$ where r_i are zero mean, unit standard deviation random vectors. We can use this to estimate Equation (4.14) by running the Falkon solver with $R = [r_1, \dots, r_t]$ instead of the labels $\hat{y}$ to obtain $(\boldsymbol{K}_{nm}^{\top}\boldsymbol{K}_{nm} + \lambda\boldsymbol{K}_{mm})^{-1}\boldsymbol{K}_{nm}^{\top}R$, then multiplying the result by $\boldsymbol{K}_{nm}^{\top}R$ and normalizing by the number of stochastic estimators t. The same random vectors R can be used to compute $\boldsymbol{K}_{nm}^{\top}R$ for Equation (4.13), coupled with the Cholesky decomposition of $\boldsymbol{K}_{mm}$. STE reduces the cost for both terms from $\mathcal{O}(nm^2)$ to $\mathcal{O}(nmt)$ which is advantageous since $t < m$. In Figure 4.4 we investigate whether the approximate objective matches the exact one, and how t affects the approximation. The observed behavior is that as few as 10 vectors are enough to approximate the full objective for a large part of the optimization run, but it can happen that such coarse approximation causes the loss to diverge. Increasing t to 20 solves the numerical issues, and on all the datasets tested we found $t = 20$ to be sufficient.

Alternatively, Equation (4.13) can be approximated with a Nyström-like procedure: taking a random subsample of size p from the whole dataset, denote $\boldsymbol{K}_{pm}$ as the kernel matrix between such p points and the m Nyström centers; then

$$\mathrm{Tr}(\boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^{\dagger}\boldsymbol{K}_{nm}^{\top}) \approx \frac{n}{p}\,\mathrm{Tr}(\boldsymbol{K}_{pm}\boldsymbol{K}_{mm}^{\dagger}\boldsymbol{K}_{pm}^{\top})$$

which can be computed in $pm^2 + m^3$ operations. By choosing $p \sim m$ the runtime is then $\mathcal{O}(m^3)$, which does not depend on the dataset size, and is more efficient than the STE approach. Unfortunately, this additional Nyström step cannot be effectively applied for computing Equation (4.14) where the inversion of B is the most time-consuming step.

4.4 Experiments

To validate the objective we are proposing for HP optimization of N-KRR models we ran a series of experiments aimed at answering the following questions:

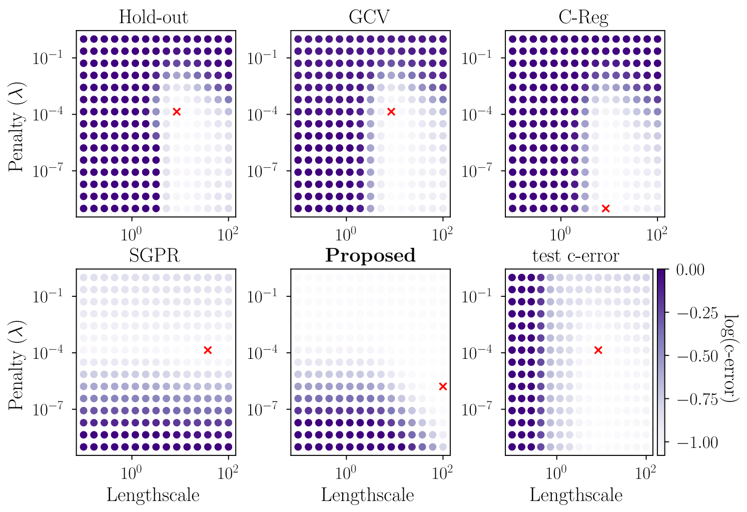


Figure 4.3 Effectiveness of test error proxies on a grid. The objective values (log transformed) are plotted at different λ, γ points for the *small-HIGGS* dataset. Lighter points indicate a smaller objective and hence a better hyperparameter configuration. The minimum of each objective is denoted by a cross.

1. Since our objective is an upper-bound on the test error, is the over-penalization acceptable, and what are its biases?
2. What is its behavior during gradient-based optimization: does it tend to overfit, does it lead to accurate models?
3. Does the approximation of Section 4.3 enable us to actually tune the hyperparameters on large datasets?

The first point is a sanity check: would the objective be a good proxy for the test error in a grid-search scenario over two hyperparameters (λ and γ with the RBF kernel). This doesn't necessarily transfer to larger HP spaces, but gives an indication of its qualitative behavior. In Figure 4.3 we compare 5 objective functions to the test error on such 2D grid. It is clear that the three functions which are unbiased estimators of the test error have very similar landscapes. Both SGPR and the proposed objective instead have the tendency to *over-penalize*: SGPR strongly disfavors low values of λ, while our objective prefers high λ and γ. This latter feature is associated with simpler models: a high γ produces smooth functions and a large λ restricts the size of the hypothesis.

We will see that the subdivision of objective functions into two distinct groups persists during optimization. However, in general it will not be true that the unbiased objectives produce models with lower test error than the over-penalized ones. The best performing method is going to depend on the dataset. The datasets used throughout this section are described in Chapter A.

4.4.1 Small-scale optimization

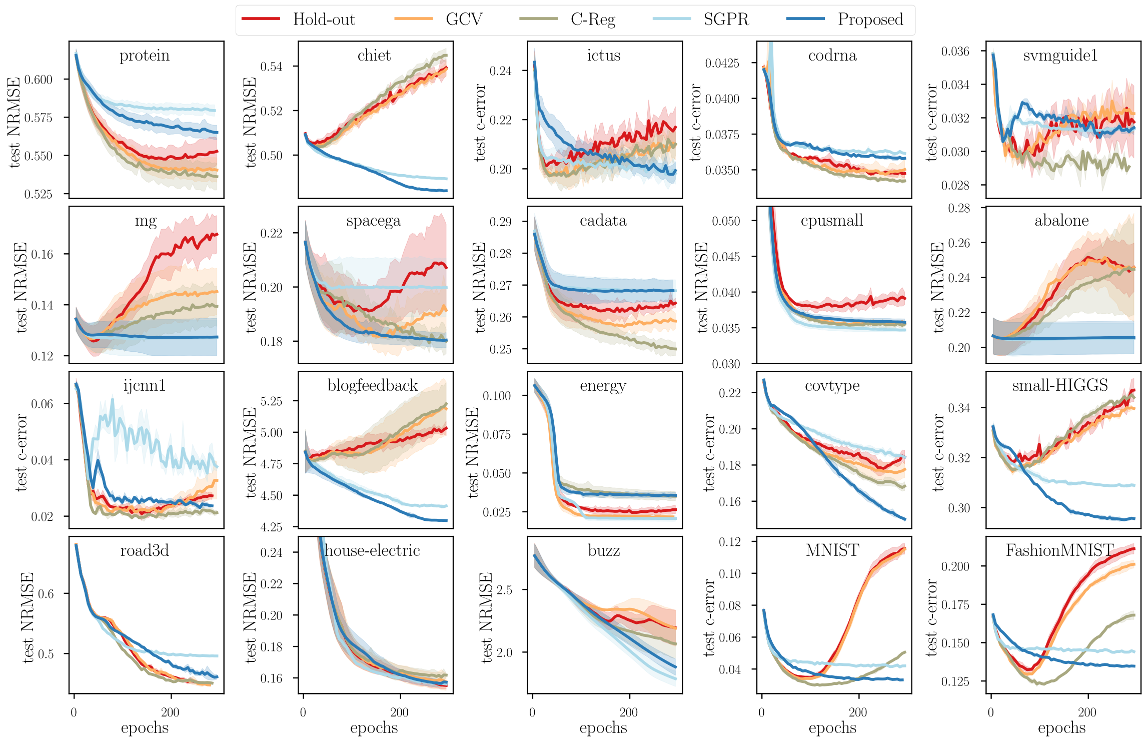


Figure 4.4 Comparing five objective functions for hyperparameter tuning. On each dataset we optimized $m = 100$ Nyström centers, a separate length-scale for each dimension and λ for 200 epochs with a learning rate of 0.05 using the Adam optimizer. Also reported is the standard deviation from 5 runs of the same experiment with a different random seed. Each dataset has its own error metric. Labels of regression datasets were normalized to have unit standard deviation.

We used the exact formulas from Sections 4.1.2 and 4.2, along with automatic differentiation to minimize different objectives on 20 datasets taken from the UCI repository, the LibSVM datasets, or in-house sources. We automated the optimization runs as much as possible to avoid having to set many meta-hyperparameters. The optimized hyperparameters were: $m = 100$ inducing points initialized to a random subset of the training set; a separate Gaussian kernel length-scale for each data dimension, initialized with the median heuristic (Garreau et al., 2017); regularization λ set to $1/n$. We used the Adam optimizer with fixed learning rate (0.05) and default settings for 200 epochs. We used early stopping whenever the objective values started increasing. The validation set size for the *Hold-out* objective was fixed to 60% of the full training data (larger than is common since the size of the considered hyperparameter space was larger than that of the parameter space).

The results – shown in Figure 4.4 – confirm our previous observations: there are some datasets (among which *small-HIGGS*, *buzz*, *house-electric*) on which the unbiased objectives overfit the training set while the proposed proxy function does not. In fact in some cases the hyperparameters found with our objective are much better than the ones found, for example, with the C-Reg objective. On the other hand, there is another group of datasets (*e.g. protein*, *energy* or *codrna*) where the extra bias of the proposed objective becomes detrimental as the optimization gets stuck into a suboptimal configuration with higher test error than what would be attainable with an unbiased objective.

Among the three unbiased objectives, hold-out clearly performs the worst. This is due to its high variance, and could be mitigated (at the expense of a higher computational cost) by using k-fold cross-validation. The GCV and C-Reg objectives perform similarly to each other in many cases. Especially in the image datasets however, GCV overfits more than C-Reg.

SGPR closely matches the proposed objective as it doesn't overfit. However, on several datasets it produces worse HPs than our objective displaying a larger bias. On the other hand there are other datasets for which the ranking is reversed, so there is no one clear winner. We must note however that the SGPR objective cannot be efficiently computed due to the log-determinant term, when datasets are large.

4.4.2 Large-scale optimization

We tested the performance of the proposed objective $\mathcal{C}^{\text{Prop}}$ with stochastic trace estimation on three large-scale datasets, comparing it against two variational sparse GP solvers (A. Matthews et al., 2017; Paszke et al., 2019) – which also learn a compact model with optimized inducing points – and a classic N-KRR model with a large number of centers chosen uniformly at random from the training set learned with Falkon.

For our objective we again used the Adam optimizer. For the Flights and Higgs dataset we trained with learning rate 0.05 for 20 epochs, while we trained Flights-Cls with a smaller learning rate of 0.02 for 10 epochs. We used the Gaussian kernel with a single length-scale, initialized as in (Meanti, Carratino, Rosasco, et al., 2020) (Flights $\gamma_0 = 1$, Flights-Cls $\gamma_0 = 1$, Higgs $\gamma_0 = 4$) and $\lambda_0 = 1/n$. We used $t = 20$ stochastic trace estimation vectors for all three experiments, sampling them from the standard Gaussian distribution. The STE vectors were kept fixed throughout optimization. The conjugate gradient tolerance for the Falkon solver was set to 5×10^{-4} for Flights-Cls, and 1×10^{-3} for Flights and Higgs (a higher tolerance corresponds to longer training time), while we always capped the number of Falkon iterations to 100.

The results in Table 4.1 tell us that we can approach (but not quite reach) the performance – both in terms of speed and accuracy – of a very large model using a small fraction of the inducing points. They also support the conclusion that our objective is effective at optimizing a large number of hyperparameters, at least on par with methods in the GPR framework.

Table 4.1 Error and running time of kernel solvers on large-scale datasets. We compare our objective with two approximate GPR implementations and hand-tuned N-KRR (Falkon).

		$\mathcal{C}^{\text{Prop}}$	GPyTorch	GPFlow	Falkon
Flights $n \approx 10^6$	error	0.794	0.803	0.790	0.758
	time(s)	355	1862	1720	245
	m	5000	1000	2000	10^5
Flights-Cls $n \approx 10^6$	error	32.2	33.0	32.6	31.5
	time(s)	310	1451	627	186
	m	5000	1000	2000	10^5
Higgs $n \approx 10^7$	error	0.191	0.199	0.196	0.180
	time(s)	1244	3171	1457	443
	m	5000	1000	2000	10^5

4.5 Full Derivation of $\mathcal{C}^{\text{Prop}}$

We split the proof of Theorem 4.1 into a few intermediate steps: after introducing the relevant notation and definitions we give a few ways in which the Nyström estimator can be expressed, useful in different parts of the proof. Then we proceed with three more technical lemmas, used later on. We split the main proof into two parts to handle the two terms of the decomposition introduced in the main text of the paper: Lemma 4.6 for the sampling variance and Lemma 4.7 for the inducing point variance. The proof of Theorem 4.1 follows directly from the two variance bounds.

4.5.1 Definitions

Using the same notation as in the main text we are given data $\{(x_i, y_i)\}_{i=1}^n \subset \mathcal{X} \times \mathcal{Y}$ such that

$$y_i = f^*(x_i) + \epsilon_i$$

where $f^* : \mathcal{X} \to \mathcal{Y}$ is an unknown function, and the noise ϵ_i follows Equation (4.9). We let $\mathcal{H}$ be a RKHS and its subspace $\mathcal{H}_m = \text{span}\{k_\gamma(z_1, \cdot), \dots, k_\gamma(z_m, \cdot)\}$ defined using the inducing points $\{z_j\}_{j=1}^m \subset \mathcal{X}$. We define a few useful operators, for vectors $v \in \mathbb{R}^m$ and $w \in \mathbb{R}^n$:

$$\begin{aligned}
&\widetilde{\Phi}_m : \mathcal{H} \to \mathbb{R}^m, && \widetilde{\Phi}_m = (k_\gamma(z_1, \cdot), \dots, k_\gamma(z_m, \cdot)) \\
&\widetilde{\Phi}_m^* : \mathbb{R}^m \to \mathcal{H}, && \widetilde{\Phi}_m^* v = \sum_{j=1}^m v_j k_\gamma(z_j, \cdot) \\
&\Phi : \mathcal{H} \to \mathbb{R}^n, && \Phi = (k_\gamma(x_1, \cdot), \dots, k_\gamma(x_n, \cdot)) \\
&\Phi^* : \mathbb{R}^n \to \mathcal{H}, && \Phi^* w = \sum_{i=1}^n w_j k_\gamma(x_i, \cdot).
\end{aligned}$$

Let $\Sigma : \mathcal{H} \to \mathcal{H} = \Phi^*\Phi$ be the covariance operator, and $\boldsymbol{K} = \Phi\Phi^* \in \mathbb{R}^{n\times n}$ the kernel operator. Further define $\boldsymbol{K}_{nm} = \Phi\widetilde{\Phi}_m^* \in \mathbb{R}^{n\times m}$, $\boldsymbol{K}_{mm} = \widetilde{\Phi}_m\widetilde{\Phi}_m^* \in \mathbb{R}^{m\times m}$, and the approximate kernel $\widetilde{\boldsymbol{K}} = \boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^\dagger\boldsymbol{K}_{nm} \in \mathbb{R}^{n\times n}$. The SVD of the linear operator $\widetilde{\Phi}_m$ is

$$\widetilde{\Phi}_m = U\Lambda V^*$$

with $U : \mathbb{R}^k \to \mathbb{R}^m$, Λ the diagonal matrix of singular values sorted in non-decreasing order, $V : \mathbb{R}^k \to \mathcal{H}$, $k \leq m$ such that $U^*U = \boldsymbol{I}$, $V^*V = \boldsymbol{I}$. The projection operator with range $\mathcal{H}_m$ is given by $P = VV^*$.

The KRR estimator $\hat{f}_{\lambda,\gamma}$ is defined as follows,

$$\hat{f}_{\lambda,\gamma} = \arg\min_{f\in\mathcal{H}} \frac{1}{n}\|f(X) - \hat{y}\|^2 + \lambda\|f\|_{\mathcal{H}}^2. \tag{4.15}$$

It can be shown (Caponnetto et al., 2007) that $\hat{f}_{\lambda,\gamma}$ is unique and can be expressed in closed form as $\hat{f}_{\lambda,\gamma} = \Phi^*(\boldsymbol{K} + n\lambda\boldsymbol{I})^{-1}\hat{y}$. In the proofs, we will also use the noise-less KRR estimator, denoted by $f_{\lambda,\gamma}$ and defined as,

$$flg = \arg\min_{f\in\mathcal{H}} \frac{1}{n}\|f(X) - f^*(X)\|^2 + \lambda\|f\|_{\mathcal{H}}^2. \tag{4.16}$$

This estimator cannot be computed since we don't have access to f^*, but it is easy to see that

$$f_{\lambda,\gamma} = \Phi^*(\boldsymbol{K} + n\lambda\boldsymbol{I})^{-1}f^*(X). \tag{4.17}$$

The N-KRR estimator, found by solving

$$\hat{f}_{\lambda,Z,\gamma} = \arg\min_{f\in\mathcal{H}_m} \frac{1}{n}\|f(X) - \hat{y}\|^2 + \lambda\|f\|_{\mathcal{H}}^2.$$

is unique, and takes the form (see Rudi, Camoriano, et al. (2015), Lemma 1)

$$\hat{f}_{\lambda,Z,\gamma} = (P\Sigma P + n\lambda\boldsymbol{I})^{-1}P\Phi^*\hat{y}$$

where P is the projection operator with range $\mathcal{H}_m$.

The estimator $\hat{f}_{\lambda,Z,\gamma}$ can be characterized in different ways as described next.

4.5.2 Preliminary Results on the Nyström estimator

The following lemma provides three different formulation of the Nyström estimator. We will use the notation $A^\dagger$ to denote the Moore-Penrose pseudo-inverse of a matrix A.

Lemma 4.1 (Alternative forms of the Nyström estimator): The following equalities hold

$$\hat{f}_{\lambda,Z,\gamma} = (P\Sigma P + n\lambda \boldsymbol{I})^{-1} P\Phi^* Y \tag{4.18}$$
$$= V(V^*\Sigma V + n\lambda \boldsymbol{I})^{-1} V^*\Phi^* Y \tag{4.19}$$
$$= \widetilde{\Phi}_m^*(\boldsymbol{K}_{nm}^\top \boldsymbol{K}_{nm} + \lambda n \boldsymbol{K}_{mm})^\dagger \boldsymbol{K}_{nm}^\top Y \tag{4.20}$$

This Lemma is a restatement of results already found in the literature (*e.g.* in Rudi, Carratino, et al. (2017), Lemmas 2 and 3) which are condensed here with slightly different proofs.

Proof of Lemma 4.1: Going from Equation (4.18) to Equation (4.19) consists in expanding $P = VV^*$ and applying the push-through identity

$$\begin{aligned}(P\Sigma P + n\lambda \boldsymbol{I})^{-1} P\Phi^* Y &= (VV^*\Sigma VV^* + n\lambda \boldsymbol{I})^{-1} VV^*\Phi^* Y \\ &= V(V^*\Sigma VV^*V + n\lambda \boldsymbol{I})^{-1} V^*\Phi^* Y \\ &= V(V^*\Sigma V + n\lambda \boldsymbol{I})^{-1} V^*\Phi^* Y.\end{aligned}$$

To go from Equation (4.20) to Equation (4.19) we split the proof into two parts. We first expand Equation (4.20) rewriting the kernel matrices

$$\begin{aligned}\widetilde{\Phi}_m^*(\boldsymbol{K}_{nm}^\top \boldsymbol{K}_{nm} + \lambda n \boldsymbol{K}_{mm})^\dagger \boldsymbol{K}_{nm}^\top Y &= \widetilde{\Phi}_m^*(\widetilde{\Phi}_m \Phi^*\Phi \widetilde{\Phi}_m^* + n\lambda \widetilde{\Phi}_m \widetilde{\Phi}_m^*)^\dagger \boldsymbol{K}_{nm}^\top Y \\ &= \widetilde{\Phi}_m^*(\widetilde{\Phi}_m(\Sigma + n\lambda \boldsymbol{I})\widetilde{\Phi}_m^*)^\dagger \boldsymbol{K}_{nm}^\top Y.\end{aligned}$$

Then, we use some properties of the pseudo-inverse (Ben-Israel et al., 2003) to simplify $(\widetilde{\Phi}_m(\Sigma + n\lambda \boldsymbol{I})\widetilde{\Phi}_m^*)^\dagger$, in particular, using the SVD of $\widetilde{\Phi}_m$, write

$$(\underbrace{U\Lambda}_{F}\, \underbrace{V^*(\Sigma + n\lambda \boldsymbol{I})V}_{H}\, \underbrace{\Lambda U^*}_{F^*})^\dagger.$$

Since U has orthonormal columns, $F^\dagger = (U\Lambda)^\dagger = \Lambda^{-1}U^\dagger = \Lambda^{-1}U^*$. A consequence is that $(F^*)^\dagger = (\Lambda U^*)^\dagger = (\Lambda^{-1}U^*)^* = U\Lambda^{-1}$. Then we split $(FHF^*)^\dagger$ into the pseudo-inverse of its three components in two steps. Firstly $(HF^*)^\dagger = (F^*)^\dagger H^\dagger$ if $H^\dagger H = I$ and $(F^*)(F^*)^\dagger = I$:

1. Since $H = V^*(\Sigma + n\lambda \boldsymbol{I})V$ is invertible, $H^\dagger = H^{-1}$ and the first condition is verified.
2. $F^*(F^*)^\dagger = \Lambda U^* U \Lambda^{-1} = I$.

Also we have $(FHF^*)^\dagger = (HF^*)^\dagger F^\dagger$ if $F^\dagger F = I$ and $HF^*(HF^*)^\dagger = I$:

1. $F^\dagger F = \Lambda^{-1}U^*U\Lambda = I$,
2. $HF^*(HF^*)^\dagger = HF^*(F^*)^\dagger H^\dagger = HH^\dagger = I$.

The end result of this reasoning is that

$$(FHF^*)^\dagger = (F^*)^\dagger H^{-1}F^\dagger = U\Lambda^{-1}(V^*(\Sigma + n\lambda\boldsymbol{I})V)^{-1}\Lambda^{-1}U^*$$

and hence

$$\begin{aligned}\widetilde{\Phi}_m^*(\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm} + \lambda n\boldsymbol{K}_{mm})^\dagger\boldsymbol{K}_{nm}^\top Y &= V\Lambda U^*(U\Lambda V^*(\Sigma + n\lambda\boldsymbol{I})V\Lambda U^*)^\dagger U\Lambda V^*\Phi^*Y\\ &= V\Lambda U^*U\Lambda^{-1}(V^*(\Sigma + n\lambda\boldsymbol{I})V)^{-1}\Lambda^{-1}U^*U\Lambda V^*\Phi^*Y\\ &= V(V^*\Sigma V + n\lambda\boldsymbol{I})^{-1}V^*\Phi^*Y\end{aligned}$$

□

Another useful equivalent form, for the Nyström estimator is given in the following lemma

Lemma 4.2: Given the kernel matrices $\boldsymbol{K}_{nm} \in \mathbb{R}^{n\times m}$, $\boldsymbol{K}_{mm} \in \mathbb{R}^{m\times m}$, and the Nyström kernel $\widetilde{\boldsymbol{K}} = \boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^\dagger\boldsymbol{K}_{nm}^\top \in \mathbb{R}^{n\times n}$, the following holds

$$(\widetilde{\boldsymbol{K}} + n\lambda\boldsymbol{I})^{-1}\widetilde{\boldsymbol{K}} = \boldsymbol{K}_{nm}(\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm} + n\lambda\boldsymbol{K}_{mm})^\dagger\boldsymbol{K}_{nm}^\top \tag{4.21}$$

Proof of Lemma 4.2: We state some facts about the kernel and image of the Nyström feature maps

$$\begin{aligned}(\ker\widetilde{\Phi}_m)^\perp &= \operatorname{span} k(z_1,\cdot),\dots,k(z_m,\cdot) = \operatorname{Im}\widetilde{\Phi}_m^*\\ (\ker\widetilde{\Phi}_m^*)^\perp &= \operatorname{Im}\widetilde{\Phi}_m = \operatorname{Im}\boldsymbol{K}_{mm} = (\ker\boldsymbol{K}_{mm})^\perp = W \subseteq \mathbb{R}^m.\end{aligned}$$

The space $\mathbb{R}^m$ is hence composed of $\mathbb{R}^m = W \oplus \ker\widetilde{\Phi}_m^*$. Take a vector $v \in \ker\widetilde{\Phi}_m^*$. We have that $\widetilde{\Phi}_m^*v = 0$, and $(\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm} + n\lambda\boldsymbol{K}_{mm})v = \widetilde{\Phi}_m(\Phi^*\Phi + n\lambda\boldsymbol{I})\widetilde{\Phi}_m^*v = 0$.
If instead $v \in W$, then $\widetilde{\Phi}_m(\Phi^*\Phi + n\lambda\boldsymbol{I})\widetilde{\Phi}_m^*v \in W$. Hence we have that

$$\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm} + n\lambda\boldsymbol{K}_{mm} : W \to W$$

and that $\boldsymbol{K}_{mm}$ is invertible when restricted to the subspace W, but also $\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm}+n\lambda\boldsymbol{K}_{mm}$ is invertible on W. Furthermore by the properties of the pseudo-inverse, we have that

$$(\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm} + n\lambda\boldsymbol{K}_{mm})(\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm} + n\lambda\boldsymbol{K}_{mm})^\dagger = P_W \tag{4.22}$$

with P_W the projector onto set W.

Furthermore we have the following equalities concerning the projection operator: $\boldsymbol{K}_{mm}^{\dagger}\boldsymbol{K}_{mm} = P_W$, as before; since $\boldsymbol{K}_{nm} = \Phi\widetilde{\Phi}_m^*$, $\boldsymbol{K}_{nm}P_W = \Phi\widetilde{\Phi}_m^* P_W = \boldsymbol{K}_{nm}$ and similarly its transpose $\boldsymbol{K}_{nm}^\top = \widetilde{\Phi}_m\Phi^*$ hence $P_W\boldsymbol{K}_{nm}^\top = \boldsymbol{K}_{nm}^\top$.
Using these properties we can say

$$\begin{aligned}\boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^{\dagger}(\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm} + n\lambda\boldsymbol{K}_{mm}) &= \boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^{\dagger}\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm} + n\lambda\boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^{\dagger}\boldsymbol{K}_{mm}\\ &= \boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^{\dagger}\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm}P_W + n\lambda\boldsymbol{K}_{nm}P_W\\ &= (\boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^{\dagger}\boldsymbol{K}_{nm}^\top + n\lambda\boldsymbol{I})\boldsymbol{K}_{nm}P_W\end{aligned}$$

which implies that

$$(\boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^{\dagger}\boldsymbol{K}_{nm}^\top + n\lambda\boldsymbol{I})^{-1}\boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^{\dagger}(\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm} + n\lambda\boldsymbol{K}_{mm}) = \boldsymbol{K}_{nm}P_W.$$

Multiplying both sides by $(\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm} + n\lambda\boldsymbol{K}_{mm})^{\dagger}$, and using Equation (4.22)

$$(\boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^{\dagger}\boldsymbol{K}_{nm}^\top + n\lambda\boldsymbol{I})^{-1}\boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^{\dagger}P_W = \boldsymbol{K}_{nm}P_W(\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm} + n\lambda\boldsymbol{K}_{mm})^{\dagger} \tag{4.23}$$

Hence we can write the left-hand side of our statement (Equation (4.21)), and use the properties of projection P_W and Equation (4.23) to get

$$\begin{aligned}&(\boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^{\dagger}\boldsymbol{K}_{nm}^\top + n\lambda\boldsymbol{I})^{-1}\boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^{\dagger}\boldsymbol{K}_{nm}^\top\\ &\qquad= (\boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^{\dagger}\boldsymbol{K}_{nm}^\top + n\lambda\boldsymbol{I})^{-1}\boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^{\dagger}P_W\boldsymbol{K}_{nm}^\top\\ &\qquad= \boldsymbol{K}_{nm}P_W(\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm} + n\lambda\boldsymbol{K}_{mm})^{\dagger}\boldsymbol{K}_{nm}^\top\\ &\qquad= \boldsymbol{K}_{nm}(\boldsymbol{K}_{nm}^\top\boldsymbol{K}_{nm} + n\lambda\boldsymbol{K}_{mm})^{\dagger}\boldsymbol{K}_{nm}^\top\end{aligned}$$

which is exactly the right-hand side of our statement. □

Finally, the algebraic transformation given in the following lemma allows to go from a form which frequently appears in proofs involving the Nyström estimator ($\mathrm{Tr}((I-P)\Sigma)$) to a form which can easily be computed: the trace difference between the full and the Nyström kernel.

Lemma 4.3: Let $\widetilde{\Phi}_m : \mathcal{H} \to \mathbb{R}^m$ be the kernel feature-map of the inducing points with SVD $\widetilde{\Phi}_m = U\Lambda V^*$, such that the projection operator onto $\mathcal{H}_m$ can be written $P = VV^*$. Also let $\widetilde{\boldsymbol{K}} = \boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^{\dagger}\boldsymbol{K}_{nm}^\top$ be the Nyström kernel. Then the following equivalence holds

$$\mathrm{Tr}((I-P)\Sigma) = \mathrm{Tr}(\ker - \widetilde{\boldsymbol{K}}). \tag{4.24}$$

Proof of Lemma 4.3: Note that we can write $\boldsymbol{K}_{mm} = \widetilde{\Phi}_m\widetilde{\Phi}_m^* = U\Lambda V^*V\Lambda U^* = U\Lambda^2U^*$, which is a full-rank factorization since both $U\Lambda$ and $\Lambda U^\top$ are full-rank. Then we can use the formula for the full-rank factorization of the pseudoinverse (Ben-Israel et al. (2003), Chapter 1, Theorem 5, Equation 24) to get

$$\begin{aligned} \boldsymbol{K}_{mm}^{\dagger} &= (U\Lambda V^* V \Lambda U^*)^{\dagger} = (U\Lambda\Lambda U^*)^{\dagger} \\ &= U\Lambda(\Lambda U^* U \Lambda^2 U^* U \Lambda)^{-1}\Lambda U^* \\ &= U\Lambda^{-2}U^*. \end{aligned}$$

Now we can prove the statement by expanding the left-hand side, and recalling $U^\top U = I$

$$\begin{aligned} \operatorname{Tr}((I-P)\Sigma) &= \operatorname{Tr}((I - VV^*)\Sigma) \\ &= \operatorname{Tr}((I - V(\Lambda U^* U \Lambda^{-2} U^* U \Lambda)V^*)\Phi^*\Phi) \\ &= \operatorname{Tr}(\Phi(I - V\Lambda U^*(\widetilde{\Phi}_m\widetilde{\Phi}_m^*)^{\dagger} U\Lambda V^*)\Phi^*) \\ &= \operatorname{Tr}(\Phi\Phi^* - \Phi\widetilde{\Phi}_m^*(\widetilde{\Phi}_m\widetilde{\Phi}_m^*)^{\dagger}\widetilde{\Phi}_m\Phi^*) \\ &= \operatorname{Tr}(ker - \boldsymbol{K}_{nm}\boldsymbol{K}_{mm}^{\dagger}\boldsymbol{K}_{nm}^{\top}) = \operatorname{Tr}(ker - \widetilde{\boldsymbol{K}}). \end{aligned}$$

□

The following two lemmas provide some ancillary results which are used in the proof of the main lemmas below.

Lemma 4.4: Let P be the projection operator onto $\mathcal{H}_m$, and $f_{\lambda,\gamma}$ be the noise-less KRR estimator. Then the following bound holds

$$\|Pf_{\lambda,\gamma}\|_{\mathcal{H}}^2 \leq \|f_{\lambda,\gamma}\|^2. \tag{4.25}$$

Proof of Lemma 4.4: This is a simple application of the definition of operator norm, coupled with the fact that orthogonal projection operators have eigenvalues which are either 0 or 1 (hence their norm is at most 1).

$$\begin{aligned} \|Pf_{\lambda,\gamma}\|_{\mathcal{H}}^2 &\leq \|P\|^2\|f_{\lambda,\gamma}\|_{\mathcal{H}}^2 \\ &\leq \|f_{\lambda,\gamma}\|_{\mathcal{H}}^2. \end{aligned}$$

□

Lemma 4.5: Recall the notation $\hat{\mathcal{E}}_\lambda(f) = n^{-1}\|f(X) - Y\|^2 + \lambda\|f\|_{\mathcal{H}}^2$, and let $f_{\lambda,\gamma}$ be the noise-less KRR estimator as before. Then the following statement holds:

$$\|f_{\lambda,\gamma}\|_{\mathcal{H}}^2 \leq \mathbb{E}[\frac{\hat{\mathcal{E}}_\lambda(f_{\lambda,\gamma})}{\lambda}] \tag{4.26}$$

where the expectation is taken with respect to the noise.

Proof of Lemma 4.5: Recall that in the fixed design setting, given a fixed (*i.e.* not dependent on the label-noise) estimator, we always have

$$\mathbb{E}[\hat{\mathcal{E}}(f)] = \mathcal{E}(f) + \sigma^2$$

where σ^2 is the label-noise variance.
In our case, noting that $L(f_{\lambda,\gamma})$ is always non-negative

$$\begin{aligned}\|f_{\lambda,\gamma}\|_{\mathcal{H}}^2 &= \frac{\lambda}{\lambda}\|f_{\lambda,\gamma}\|_{\mathcal{H}}^2 \\ &\leq \frac{\mathcal{E}(f_{\lambda,\gamma}) + \lambda\|f_{\lambda,\gamma}\|_{\mathcal{H}}^2}{\lambda} \\ &\leq \frac{\mathcal{E}(f_{\lambda,\gamma}) + \sigma^2 + \lambda\|f_{\lambda,\gamma}\|_{\mathcal{H}}^2}{\lambda} \\ &= \frac{\mathbb{E}[\hat{\mathcal{E}}_\lambda(f_{\lambda,\gamma})]}{\lambda}.\end{aligned}$$

□

4.5.3 Proof of the main Theorem

The proof of Theorem 4.1 starts from the error decomposition found in Section 4.2 which we report here:

$$\begin{aligned}\mathbb{E}_{\mathcal{E}(\hat{f}_{\lambda,Z,\gamma})}[\leq]\mathbb{E}\Big[&\underbrace{\mathcal{E}(\hat{f}_{\lambda,Z,\gamma}) - \hat{\mathcal{E}}(\hat{f}_{\lambda,Z,\gamma})}_{①} \\ &+ \underbrace{\hat{\mathcal{E}}(\hat{f}_{\lambda,Z,\gamma}) + \lambda\|\hat{f}_{\lambda,Z,\gamma}\|_{\mathcal{H}}^2 - \hat{\mathcal{E}}_\lambda(Pf_{\lambda,\gamma})}_{②} + \underbrace{\hat{\mathcal{E}}_\lambda(Pf_{\lambda,\gamma})}_{③}\Big]\end{aligned}$$

and proceeds by bounding terms ① (see Lemma 4.6) and ③ (see Lemma 4.7). After the two necessary lemmas we restate the proof of the main theorem which now becomes trivial.

Lemma 4.6 (Bounding the data-sampling variance): Denoting by $\hat{f}_{\lambda,Z,\gamma}$ the N-KRR estimator, the expected difference between its empirical and test errors can be calculated exactly:

$$\mathbb{E}[\mathcal{E}(\hat{f}_{\lambda,Z,\gamma}) - \hat{\mathcal{E}}(\hat{f}_{\lambda,Z,\gamma})] = \frac{2\sigma^2}{n}\operatorname{Tr}((\widetilde{\boldsymbol{K}} + n\lambda\boldsymbol{I})^{-1}\widetilde{\boldsymbol{K}})$$

with σ^2 the noise variance and $\widetilde{\boldsymbol{K}}$ the Nyström kernel.

Proof of Lemma 4.6: For the sake of making the proof self-contained we repeat the reasoning of Section 4.2. Starting with the expectation of the empirical error we decompose it into the expectation of the test error minus an inner product term:

$$\begin{aligned}\mathbb{E}[\hat{\mathcal{E}}(\hat{f}_{\lambda,Z,\gamma})] &= \mathbb{E}[\frac{1}{n}\|\hat{f}_{\lambda,Z,\gamma}(X) - f^*(X) - \epsilon\|^2] \\ &= \mathbb{E}[\mathcal{E}(\hat{f}_{\lambda,Z,\gamma})] + \sigma^2 - \frac{2}{n}\mathbb{E}[\langle \hat{f}_{\lambda,Z,\gamma}(X) - f^*(X), \epsilon\rangle].\end{aligned}$$

The σ^2 term is fixed for optimization purposes, so we must deal with the inner-product. We use the form of $\hat{f}_{\lambda,Z,\gamma}$ from Equation (4.20), Lemma 4.1, and $\mathbb{E}[\epsilon] = 0$, and to clean the notation we call $H := \boldsymbol{K}_{nm}(\boldsymbol{K}_{nm}^\top \boldsymbol{K}_{nm} + n\lambda \boldsymbol{K}_{mm})^\dagger \boldsymbol{K}_{nm}^\top$:

$$\begin{aligned}\frac{2}{n}\mathbb{E}[\langle \hat{f}_{\lambda,Z,\gamma}(X) - f^*(X), \epsilon\rangle] &= \frac{2}{n}\mathbb{E}[\langle H(f^*(X) + \epsilon) - f^*(X), \epsilon\rangle] \\ &= \frac{2}{n}\mathbb{E}[\epsilon^\top H \epsilon] = \frac{2\sigma^2}{n}\operatorname{Tr}(H),\end{aligned}$$

and using Lemma 4.2 H can be expressed as $(\widetilde{\boldsymbol{K}} + n\lambda \boldsymbol{I})^{-1}\widetilde{\boldsymbol{K}}$.
Going back to the original statement we have

$$\begin{aligned}\mathbb{E}[\mathcal{E}(\hat{f}_{\lambda,Z,\gamma}) - \hat{\mathcal{E}}(\hat{f}_{\lambda,Z,\gamma})] &= \mathbb{E}[\mathcal{E}(\hat{f}_{\lambda,Z,\gamma}) - \mathcal{E}(\hat{f}_{\lambda,Z,\gamma}) + \frac{2\sigma^2}{n}\operatorname{Tr}((\widetilde{\boldsymbol{K}} + n\lambda \boldsymbol{I})^{-1}\widetilde{\boldsymbol{K}})] \\ &= \frac{2\sigma^2}{n}\operatorname{Tr}((\widetilde{\boldsymbol{K}} + n\lambda \boldsymbol{I})^{-1}\widetilde{\boldsymbol{K}})\end{aligned}$$

□

Lemma 4.7 (Bounding the Nyström variance): Under the fixed-design assumptions, denote by P the orthogonal projector onto space $\mathcal{H}_m$, by $\hat{\mathcal{E}}_\lambda(f)$ the regularized empirical risk of estimator f, and by $f_{\lambda,\gamma} \in \mathcal{H}$ the noise-less KRR estimator. Then the following upper-bound holds

$$\mathbb{E}[\hat{\mathcal{E}}_\lambda(Pf_{\lambda,\gamma})] \leq \frac{2}{n\lambda}\operatorname{Tr}(\boldsymbol{K} - \widetilde{\boldsymbol{K}})\,\mathbb{E}[\hat{\mathcal{E}}(f_{\lambda,\gamma})] + 2\,\mathbb{E}[\hat{\mathcal{E}}_\lambda(f_{\lambda,\gamma})]. \tag{4.27}$$

Proof of Lemma 4.7: Note that for estimators $f \in \mathcal{H}$ we can always write $f(X) = \Phi f$. Hence for the projected KRR estimator we use that $(Pf_{\lambda,\gamma})(X) = \Phi P f_{\lambda,\gamma}$. We start by rewriting the left hand side to obtain a difference between projected and non-projected terms:

$$\begin{aligned}
\mathbb{E}[\hat{\mathcal{E}}(Pf_{\lambda,\gamma}) + \lambda\|Pf_{\lambda,\gamma}\|^2_{\mathcal{H}}] &= \mathbb{E}[\frac{1}{n}\|\Phi Pf_{\lambda,\gamma} - f^*(X) - \epsilon\|^2 + \lambda\|Pf_{\lambda,\gamma}\|^2_{\mathcal{H}}] \\
&= \mathbb{E}[\frac{1}{n}\|\Phi Pf_{\lambda,\gamma} - f^*(X)\|^2 + \frac{1}{n}\|\epsilon\|^2 + \lambda\|Pf_{\lambda,\gamma}\|^2_{\mathcal{H}}] \\
&= \mathbb{E}[\frac{1}{n}\|\Phi Pf_{\lambda,\gamma} - \Phi f_{\lambda,\gamma} + \Phi f_{\lambda,\gamma} - f^*(X)\|^2 \\
&\quad + \frac{1}{n}\|\epsilon\|^2 + \lambda\|Pf_{\lambda,\gamma}\|^2_{\mathcal{H}}] \\
&\leq \mathbb{E}[\frac{2}{n}\|\Phi Pf_{\lambda,\gamma} - \Phi f_{\lambda,\gamma}\|^2 \\
&\quad + \frac{2}{n}\|\Phi f_{\lambda,\gamma} - f^*(X)\|^2 + \frac{2}{n}\|\epsilon\|^2 + 2\lambda\|Pf_{\lambda,\gamma}\|^2_{\mathcal{H}}]
\end{aligned}$$

where we used the fact that $\mathbb{E}[\epsilon] = 0$, and the triangle inequality in the last step. By Lemma 4.4, and the definition of $\mathbb{E}[\hat{\mathcal{E}}(f)]$ we have that

$$\mathbb{E}[\frac{2}{n}\|\Phi f_{\lambda,\gamma} - f^*(X)\|^2 + \frac{2}{n}\|\epsilon\|^2 + 2\lambda\|Pf_{\lambda,\gamma}\|^2_{\mathcal{H}}] \leq 2\,\mathbb{E}[\hat{\mathcal{E}}(f_{\lambda,\gamma})].$$

Next we use again the definition of operator norm to deal with the difference between projected and non-projected noise-less KRR estimators:

$$\begin{aligned}
\mathbb{E}[\frac{2}{n}\|\Phi Pf_{\lambda,\gamma} - \Phi f_{\lambda,\gamma}\|^2] &= \frac{2}{n}\|\Phi(P - \boldsymbol{I})f_{\lambda,\gamma}\|^2 \\
&\leq \frac{2}{n}\|\Phi(\boldsymbol{I} - P)\|^2\|f_{\lambda,\gamma}\|^2.
\end{aligned}$$

The first part of this latter term is

$$\|\Phi(\boldsymbol{I} - P)\|^2 = \|(\boldsymbol{I} - P)\Phi^\top\Phi(\boldsymbol{I} - P)\| \leq \mathrm{Tr}((\boldsymbol{I} - P)\Phi^\top\Phi) = \mathrm{Tr}((\boldsymbol{I} - P)\Sigma)$$

since the trace norm controls the operator norm, and using the cyclic property of the trace and the idempotence of the projection operator $\boldsymbol{I} - P$. By Lemma 4.3 we have that $\|\Phi(\boldsymbol{I} - P)\|^2 \leq \mathrm{Tr}(\boldsymbol{K} - \widetilde{\boldsymbol{K}})$. For the second part we use Lemma 4.5 so that

$$\|f_{\lambda,\gamma}\|^2 \leq \mathbb{E}[\frac{\hat{\mathcal{E}}_\lambda(f_{\lambda,\gamma})}{\lambda}]$$

which concludes the proof. □

We now have all the ingredients to prove Theorem 4.1 which we restate below for the reader.

Theorem: (restated from Section 4.2)

main-restated Under the assumptions of fixed-design regression we have that,

$$\begin{aligned}\mathbb{E}[\mathcal{E}(\hat{f}_{\lambda,Z,\gamma})] \leq & \frac{2\sigma^2}{n}\operatorname{Tr}((\widetilde{\boldsymbol{K}}+\lambda\boldsymbol{I})^{-1}\widetilde{\boldsymbol{K}}) \\ & + \frac{2}{n\lambda}\operatorname{Tr}(\boldsymbol{K}-\widetilde{\boldsymbol{K}})\,\mathbb{E}[\hat{\mathcal{E}}(f_{\lambda,\gamma})] \\ & + 2\,\mathbb{E}[\hat{\mathcal{E}}(f_{\lambda,\gamma})] \end{aligned} \tag{4.28}$$

Proof of Theorem 4.1: The decomposition is the same:

$$\mathbb{E}[\mathcal{E}(\hat{f}_{\lambda,Z,\gamma})] \leq \mathbb{E}\Big[\underbrace{\mathcal{E}(\hat{f}_{\lambda,Z,\gamma}) - \hat{\mathcal{E}}(\hat{f}_{\lambda,Z,\gamma})}_{①}$$
$$+ \underbrace{\hat{\mathcal{E}}(\hat{f}_{\lambda,Z,\gamma}) + \lambda\|\hat{f}_{\lambda,Z,\gamma}\|^2_{\mathcal{H}} - \hat{\mathcal{E}}_\lambda(Pf_{\lambda,\gamma})}_{②} + \underbrace{\hat{\mathcal{E}}_\lambda(Pf_{\lambda,\gamma})}_{③}\Big]$$

where ② ≤ 0. We may then use Lemma 4.6 for term ① and Lemma 4.7 for term ③ to obtain

$$\mathbb{E}[\mathcal{E}(\hat{f}_{\lambda,Z,\gamma})] \leq \frac{2\sigma^2}{n}\operatorname{Tr}((\widetilde{\boldsymbol{K}}+n\lambda\boldsymbol{I})^{-1}\widetilde{\boldsymbol{K}}) + \frac{2}{n\lambda}\operatorname{Tr}(\boldsymbol{K}-\widetilde{\boldsymbol{K}})\,\mathbb{E}[\hat{\mathcal{E}}_\lambda(f_{\lambda,\gamma})] + 2\,\mathbb{E}[\hat{\mathcal{E}}_\lambda(f_{\lambda,\gamma})].$$

□

4.6 Conclusions

In this work, we improved the usability of large scale kernel methods proposing a gradient-based solution for tuning a large number of hyperparameters, on large problems. The developed algorithm is added to an existing library for large scale kernel methods with GPUs (available at https://github.com/FalkonML/Falkon. We showed that it is possible to train compact Nyström-KRR models if the centers are allowed to deviate from the training set, which can speed up inference by orders of magnitude. A future work will be to consider complex parameterized kernels which allow to improve the state of the art of kernel-based models on structured datasets such as those containing images or text.

Chapter 5

Exponential Rates for Multiclass Learning

Consider the *learning curves* obtained by plotting test error as a function of model size (or complexity). The classical hypothesis, backed by theoretical results, is that performance should degrade as models get larger or less constrained (Hastie et al., 2009). However, it was recently remarked that the learning curves observed in practice (in certain settings) can be quite different from those predicted in theory C. Zhang, Bengio, et al., 2021, and the *overfitting* phenomenon does not occur. By the no free lunch theorem (Wolpert, 1996), theoretical results critically depend on the set of assumptions made on the problem. Such assumptions can be hard to verify in practice, hence a possible way to tackle the seeming contradictions in learning theory *vs.* practice is to consider a wider range of assumptions, and check whether the corresponding results can explain empirical observations.

In the context of classification, it is interesting to consider assumptions describing the difficulty of the problem in terms of *margin* (Mammen et al., 1999; Tsybakov, 2004). It is well known that very different learning curves can be obtained depending on the considered margin conditions Bartlett, Jordan, et al., 2006. Further, the behavior of the test error in terms of misclassification can be considerably different from that induced by the surrogate loss function (*e.g.* squared or logistic) used for empirical risk minimization (T. Zhang, 2004b; Bartlett, Jordan, et al., 2006). An extreme case is when there is a hard margin among the classes. Indeed, in this case the misclassification error can decrease *exponentially* fast as the number of points increases, while the surrogate loss displays a polynomial decay. This behavior was first noted in Koltchinskii et al. (2005) and Audibert et al. (2007) for a wide class of estimators (see also Yao et al. (2007)), and reprised more recently in Pillaud-Vivien et al. (2018) and Nitanda et al. (2019) for stochastic gradient descent. The effect of margin conditions has also been considered for *multiclass* learning (T. Zhang, 2004a; D.-R. Chen et al., 2006; Mroueh et al., 2012), but not in the hard-margin case. Interestingly, hard-margin and exponential rates have been studied by

Cabannes et al. (2021) in the context of structured prediction (Nowak et al., 2019). However, these latter results are restricted to least-squares based estimators.

The purpose of this chapter is twofold. On the one hand, we analyze the effect of margin conditions, and in particular hard-margin conditions, for a wide class of multiclass estimators derived from different surrogate losses. On the other hand, we build on ideas in Mroueh et al. (2012), Pillaud-Vivien et al. (2018), and Nitanda et al. (2019) to provide a simplified and self-contained treatment that naturally recovers results for binary classification as a special case. In particular, we note that, in the presence of a hard margin, the misclassification error curve does not exhibit any *bias-variance* trade-off, thus providing a possible explanation to the empirical observations that motivate our study.

The rest of the paper is organized as follows. In Section 5.1 we describe the multiclass classification problem, the surrogate approach and the simplex encoding. In Section 5.2 we analyze the bias-variance decomposition for the misclassification risk, discuss soft and hard-margin conditions, and prove our main results of exponential convergence under assumptions of hard margin. In Section 5.3 we validate the theory with experiments on synthetic data. Some final remarks are provided in Section 5.4.

5.1 Setting

We consider a standard multiclass learning problem. Let $(X, Y) \in \mathcal{X} \times \mathcal{Y}$ be a random pair, where $\mathcal{X} \subset \mathbb{R}^d$ and $\mathcal{Y}$ is a finite set of $T \geq 2$ elements. We call the elements of $\mathcal{Y}$ *classes*, and a (measurable) function $c : \mathcal{X} \to \mathcal{Y}$ a *classifier*. As we have seen in Section 2.1, The misclassification risk of a classifier c is

$$\mathcal{E}(c) = \mathbb{P}\{c(X) \neq Y\}$$

which is minimized by the Bayes classifier c_* (see Equation (2.10)). We denote the minimum risk by $\mathcal{E}_* = \mathcal{E}(c_*)$. Given n independent copies (x_i, y_i) of (X, Y), $i = 1, \dots, n$, the goal is to learn a classifier $\widehat{c}$ such that $\mathcal{E}(\widehat{c}) - \mathcal{E}_* \to 0$ in expectation as $n \to \infty$. More precisely, we are interested in finite-sample bounds of the form

$$\mathbb{E}[\mathcal{E}(\widehat{c})] - \mathcal{E}_* \lesssim a_n,$$

where $a_n \to 0$ gives a rate of convergence.

Empirical risk minimization would prescribe to compute $\widehat{c}$ by minimizing a sample version of $\mathcal{E}$:

$$\frac{1}{n} \sum_{i=1}^{n} \mathbb{1}\{c(x_i) \neq y_i\}$$

However, the 0-1 loss is neither smooth nor convex, and optimizing it is in general an NP-hard combinatorial problem (Feldman et al., 2012). A viable strategy is to replace the 0-1 loss with a convex *surrogate*, and the space of classifiers with a suitable linear space of vector-valued functions. To do this, it is necessary to choose a vector encoding of the classes $\mathcal{Y} \hookrightarrow \mathbb{R}^p$ and a decoding operator $D : \mathbb{R}^p \to \mathcal{Y}$. One possibility, mentioned in Section 2.1.2 is to use the one-hot encoding, where T classes are encoded as T-dimensional standard basis vectors. In this Chapter we follow instead Mroueh et al. (2012), where the classes are encoded as the vertices of a $(T-1)$-simplex embedded in $\mathbb{R}^{T-1}$ (see Figure 5.1). For notational convenience we denote

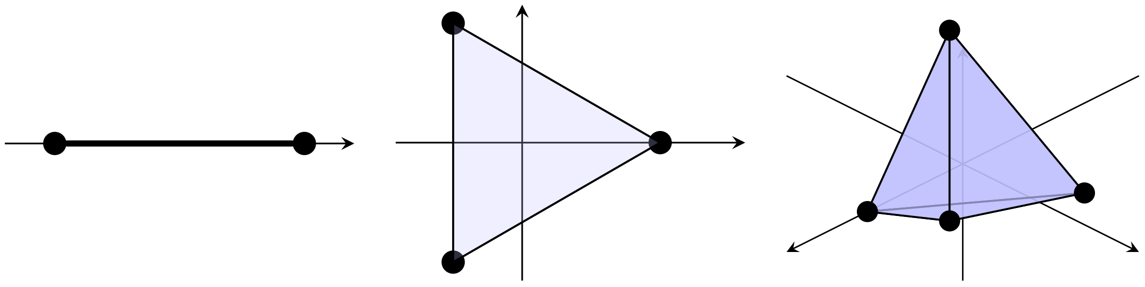


Figure 5.1 Simplex encoding for $T = 2, 3, 4$.

$\mathcal{Y}$ itself with its simplex encoding, that is $\mathcal{Y}$ is the set of points in $\mathbb{R}^{T-1}$ such that

$$\|y\| = 1, \qquad \langle y, y' \rangle = -\frac{1}{T-1}. \tag{5.1}$$

The decoding operator assigns a vector to the class with largest projection, with ties arbitrarily broken (see Figure 5.2):

$$D : \mathbb{R}^{T-1} \to \mathcal{Y}, \qquad D(w) = \arg\max_{y \in \mathcal{Y}} \langle w, y \rangle. \tag{5.2}$$

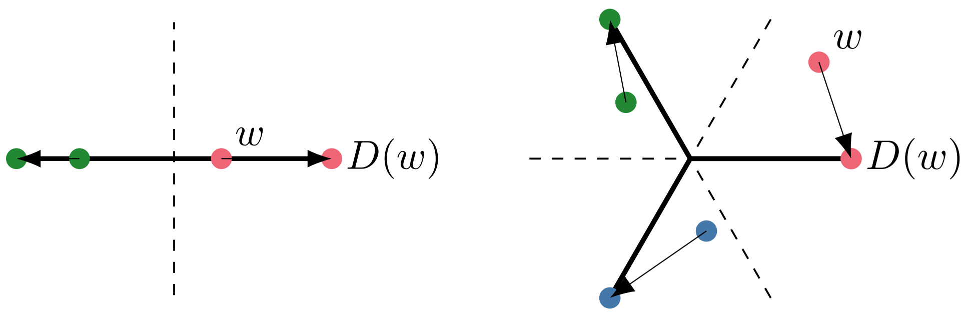


Figure 5.2 Simplex decoding ($T = 2, 3$).

In the case of binary classification ($T = 2$), we have $\mathcal{Y} = \{\pm 1\} \subset \mathbb{R}$ and $D(w) = \operatorname{sgn}(w)$. A *plug-in* classifier $Df(x) = D(f(x))$ can be defined by composing a vector-valued function $f : \mathcal{X} \to \mathbb{R}^{T-1}$ with the decoding operator. The simplex coding offers some advantages over the one-hot. First, as we just saw, it is perfectly consistent with the standard $(\{\pm 1\}, \mathrm{sgn})$ coding of binary classification. Second, it automatically satisfies structural constraints that other codings need to impose additionally on the hypothesis class, such as the so-called sum to zero constraint. This makes both numerical implementation and theoretical analysis more straightforward.

To identify the *target function* to plug into the decoder, we fix a convex surrogate loss $\ell : \mathcal{Y} \times \mathbb{R}^{T-1} \to [0, \infty)$ with corresponding risk $\mathcal{E}_\ell(f)$ defined in Equation (2.5). The risk minimizer is

$$f_\ell = \underset{f \in L^0(\mathcal{Y}, \mathbb{R}^{T-1})}{\arg\min} \ \mathcal{E}_\ell(f). \tag{5.3}$$

We then approximate f_ℓ by a (uniform) approximator f_λ. At the current level of generality, λ simply denotes a generic parameter to be tuned. For instance, f_λ can be the minimizer of a regularized risk, with λ the regularization parameter. Finally, our classifier will be $D\hat{f}_\lambda$, with $\hat{f}_\lambda$ the empirical estimate of f_λ based on the finite samples $\{(x_i, x_i)\}_{i=1}^n$.

We are going to consider two types of loss functions. The first one is the square loss $\ell(w, y) = \|w - y\|^2$, for which $f_\ell(x) = f_\rho(x)$, where

$$f_\rho(x) = \mathbb{E}[Y \mid X = x]$$

is the regression function. The second case is a family of functions which depend on the *margin* $\langle w, y \rangle$, namely losses of the form $\ell_\phi(w, y) = \phi(\langle w, y \rangle)$ for a suitable (differentiable, convex) function $\phi : \mathbb{R} \to [0, \infty)$. Examples of ϕ are

$$\phi(t) = \ln(2)^{-1} \ln(1 + e^{-t}) \tag{5.4}$$

which generalizes the logistic loss (see Equation (2.2)) to the multiclass setting and

$$\phi(t) = e^{-t} \tag{5.5}$$

which generalizes the exponential loss. For margin losses, we will denote the minimizer f_ℓ by f_ϕ. Note that in binary classification the square loss is itself a function of the margin, $\ell(w, y) = \|1 - wy\|^2$, while for $T \geq 3$ this is no longer the case.

5.2 Analysis

We start by analyzing the peculiar structure of the bias-variance decomposition in classification. As we will see, the key point is that the bias can be made zero under suitable margin conditions. When only the variance is left, the misclassification error can be controlled by uniform concentration. These general facts can then be applied to different loss functions, leading to our main results.

5.2.1 Bias-variance for plug-in classifiers

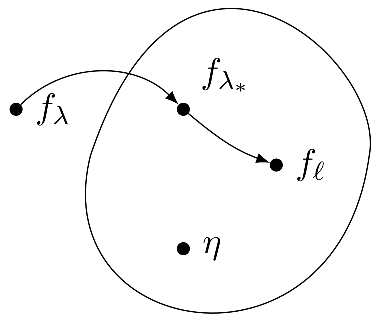
To analyze the performance of a plug-in classifier $D\hat{f}_\lambda$, we decompose the excess misclassification risk as

$$\begin{aligned}\mathcal{E}(D\hat{f}_\lambda) - \mathcal{E}_* &= \mathcal{E}(D\hat{f}_\lambda) - \mathcal{E}(Df_\lambda) && (5.6)\\ &+ \mathcal{E}(Df_\lambda) - \mathcal{E}(Df_\ell) && (5.7)\\ &+ \mathcal{E}(Df_\ell) - \mathcal{E}_*. && (5.8)\end{aligned}$$

The last term results from replacing the 0-1 loss with the surrogate loss ℓ. Loss functions for which $Df_\ell = c_*$, and therefore Equation (5.8) is zero, are called *Fisher consistent* (or *classification calibrated*). Fisher consistency is a common and well characterized property T. Zhang, 2004b. In particular, the square loss is Fisher consistent (see Lemma 5.2). For margin losses, consistency will be assumed in all that follows, and shown in some examples.

The term (5.7) is a bias term. Crucially, it can be set to zero for a wide range of parameters λ. The idea is that we can have $\mathcal{E}(Df_\lambda) = \mathcal{E}(Df_\ell)$ even when $\mathcal{E}_\ell(f_\lambda) \gg \mathcal{E}_\ell(f_\ell)$. Here is a fundamental difference between regression and classification. While in regression f_ℓ is a target point, in classification it is rather a representative of the target class

$$[f_\ell] = \{f \in L^0(\mathcal{X}, \mathbb{R}^{T-1}) | Df = Df_\ell \text{ almost surely}\}. \quad (5.9)$$



Hence, it is enough for f_λ to land in $[f_\ell]$, possibly far from f_ℓ itself. This is easier if the class $[f_\ell]$ is “large”, which can be ensured by imposing special margin conditions. Assuming that ℓ is Fisher consistent, a generic function f lies in $[f_\ell]$ if and only if

$$Df = c_* \quad \text{almost surely.} \qquad (5.10)$$

Chosen a Fisher consistent loss and put the bias to zero, all that’s left is the variance term (5.6):

$$\mathcal{E}(D\hat{f}_\lambda) - \mathcal{E}_* = \mathcal{E}(D\hat{f}_\lambda) - \mathcal{E}(Df_\lambda). \qquad (5.11)$$

At this point, λ is set and needs no trade-off. Fast convergence of the variance, and therefore of the whole excess misclassification risk, can be derived using once again margin conditions.

5.2.2 Margin conditions

In binary classification, the *margin conditions*, also known as Tsybakov’s low-noise assumptions Mammen et al., 1999; Tsybakov, 2004; Koltchinskii et al., 2005; Audibert et al., 2007, are a set of assumptions under which it is possible to obtain fast convergence (up to exponential) for plug-in classifiers. They can be stated as follows: there exists $\alpha \in (0, \infty]$ such that, for every $\delta > 0$,

$$\mathbb{P}\{|f_\rho(x)| \le \delta\} \lesssim \delta^\alpha. \tag{5.12}$$

In the extreme case of $\alpha = \infty$, we get

$$|f_\rho(x)| \ge \delta \quad \text{almost surely,} \tag{5.13}$$

which is sometimes referred to as the *hard-margin* condition.

Following Mroueh et al., 2012; Nowak et al., 2019, we can generalize Equations (5.12) and (5.13) to the multiclass setting. For $w \in \mathbb{R}^{T-1}$, we define the *decision margin*

$$M(w) = \min_{y \neq D(w)} \langle w, D(w) - y \rangle. \tag{5.14}$$

$M(w)$ is the difference between the largest and the second largest projection of w onto $\mathcal{Y}$, namely the confidence gap between first and second guess. For $T = 2$, we have $M(w) = 2|w|$. In general, we say that a function $f : \mathcal{X} \to \mathbb{R}^{T-1}$ satisfies the margin condition with exponent $\alpha \in (0, \infty]$ if, for every $\delta > 0$,

$$\mathbb{P}\{M(f(x)) \le \delta\} \lesssim \delta^\alpha. \tag{5.15}$$

In particular, one can take $f = f_\rho$, which for $T = 2$ gives back Equation (5.12). Again, $\alpha = \infty$ gives the hard-margin condition (see Figure 5.3)

$$M(f(x)) \ge \delta \quad \text{almost surely.} \tag{5.16}$$

Intuitively, these conditions say that the probability of falling in a "runoff zone", where the

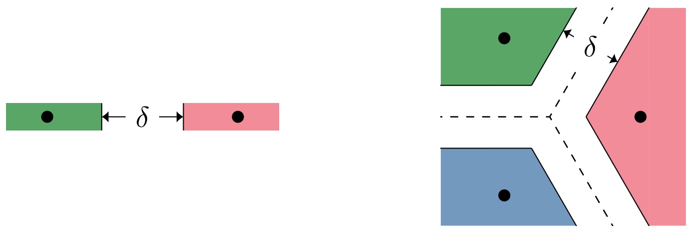


Figure 5.3 Hard-margin condition ($T = 2, 3$).

plugging-in function would be "uncertain", is either (polynomially) small (5.15), or zero (5.16). The reason why we state Equations (5.15) and (5.16) for an arbitrary f is that we will transfer these properties to minimizers f_ℓ (and f_λ) of general (regularized) losses, including but not limited to the square. Combining Fisher consistency and hard margin, we obtain the following stronger condition.

Lemma 5.1: A function $f \in L^0(\mathcal{X}, \mathbb{R}^{T-1})$ satisfies Equations (5.10) and (5.16) if and only if

$$\min_{y \neq c_*(x)} \langle f(x), c_*(x) - y \rangle \geq \delta \quad \text{almost surely.} \tag{5.17}$$

Proof of Lemma 5.1: First note that, if (5.10) holds, then (5.17) is the same as (5.16). Now suppose Equation (5.17) holds. Then

$$\langle f(x), c_*(x) \rangle > \max_{y \neq c_*(x)} \langle f(x), y \rangle,$$

hence $Df(x) = c_*(x)$, that is, Equation (5.10) holds too. □

Beside (5.15) and (5.16), we will also consider another generalization of (5.13) which is independent of any particular classifier, and instead is stated purely in terms of the conditional probabilities. To illustrate such a condition, we note that Equation (5.13) is equivalent to saying that either $\rho(1|x)$ or $\rho(-1|x)$ is no less than $1/2 + \delta/2$ (for almost every x, there is one class with probability bounded away from coin flipping). This in turn is equivalent to

$$\min_{y \neq c_*(x)} \rho(c_*(x) \mid x) - \rho(y \mid x) \geq \delta \quad \text{almost surely,} \tag{5.18}$$

which says that the most probable class has almost always an edge of δ over the second most probable class. Since this inequality makes sense for arbitrary T, we take it as our hard-margin condition for multiclass problems. More generally, one may consider problems where for some $\alpha \in (0, \infty]$ and all $\delta > 0$,

$$\mathbb{P}\left\{ \min_{y \neq c_*(x)} \rho(c_*(x) \mid x) - \rho(y \mid x) \leq \delta \right\} \lesssim \delta^\alpha, \tag{5.19}$$

generalizing Equation (5.12) to $T \geq 2$.

The margin conditions on the conditional probabilities can be related to those expressed on classifiers.

Lemma 5.2: We have $Df_\rho = c_*$ almost surely. Moreover, Equation (5.18) holds if and only if f_ρ satisfies (5.16).

Proof of Lemma 5.2: Recalling the definition of $\mathcal{Y}$ in Equation (5.1), let

$$\Delta = \{p \in \mathcal{Y} : p_y \geq 0, \sum_{y \in \mathcal{Y}} p_y = 1\}$$

be the probability simplex on $\mathcal{Y}$, and let $\operatorname{co}\mathcal{Y}$ be the encoding simplex defined as the convex hull of $\mathcal{Y}$. Then Δ and $\operatorname{co}\mathcal{Y}$ are canonically isomorphic via the barycenter coordinate map

$$\beta : \Delta \to \operatorname{co}\mathcal{Y}, \qquad \beta(p) = \sum_{y\in\mathcal{Y}} p_y y.$$

Now consider the map

$$\rho : \mathcal{X} \to \Delta, \qquad \rho(x) = [\rho(y \mid x)]_{y\in\mathcal{Y}}.$$

Then we have $\beta \circ \rho = f_\rho$. Furthermore

$$\begin{aligned} \arg\max_{y\in\mathcal{Y}} \langle f_\rho(x), y\rangle &= \arg\max_{y\in\mathcal{Y}} \langle \sum_{y'\in\mathcal{Y}} \rho(y' \mid x) y', y\rangle \\ &= \arg\max_{y\in\mathcal{Y}} \sum_{y'\in\mathcal{Y}} \rho(y' \mid x)\langle y, y'\rangle \\ &= \arg\max_{y\in\mathcal{Y}} \rho(y \mid x) \end{aligned}$$

The same holds for maximizing over $y \neq c_*(x)$, whence the second claim of the Theorem.

□

5.2.3 Misclassification comparison

In view of Equation (5.11), we need in fact to compare the misclassification risk of two classifiers. This can be done by introducing a bounding distance. Since the distance will be symmetric, the resulting bound gives a symmetric comparison between any two classifiers, as opposed to the usual comparison of a classifier with respect to a fixed (Bayes) rule. For this reason, the following results may be of independent interest.

We define the *Hamming distance* of $c', c \in L^0(\mathcal{X}, \mathcal{Y})$ as

$$r(c', c) = \mathbb{P}\{c'(x) \neq c(x)\}.$$

The Hamming distance bounds the difference of misclassification risk.

Lemma 5.3: For every $c', c \in L^0(\mathcal{X}, \mathcal{Y})$,

$$|\mathcal{E}(c') - \mathcal{E}(c)| \leq r(c', c). \tag{5.20}$$

Proof of Lemma 5.3: By direct computation,

$$\begin{aligned}|\mathcal{E}(c') - \mathcal{E}(c)| &= |\mathbb{E}[\mathbb{1}\{c'(x) \neq y\} - \mathbb{1}\{c(x) \neq y\}]| \\ &\leq \mathbb{E}[|\mathbb{1}\{c'(x) \neq y\} - \mathbb{1}\{c(y) \neq y\}|] \\ &\leq \mathbb{E}[\mathbb{1}\{c'(x) \neq c(x)\}] = r(c', c).\end{aligned}$$

□

The next step is to bound the Hamming distance between two plug-in classifiers.

Lemma 5.4: For every $f', f \in L^\infty(\mathcal{X}, \mathbb{R}^{T-1})$,

$$r(Df', Df) \leq \mathbb{P}\left\{|f' - f|_\infty \geq \sqrt{\frac{T-1}{2T}} M(f(x))\right\}. \tag{5.21}$$

Proof of Lemma 5.4: Let $Df'(x) = y' \neq y = Df(x)$. Then

$$\begin{aligned}\min_{j \neq y'} \langle y' - j, f'(x)\rangle &= \langle y', f'(x)\rangle - \max_{j \neq y'} \langle j, f'(x)\rangle \\ &\leq \langle y', f'(x)\rangle - \langle y, f'(x)\rangle \\ &\leq \langle y' - y, f'(x)\rangle - \langle y' - y, f(x)\rangle \\ &= \langle y' - y, f'(x) - f(x)\rangle \\ &\leq \|y' - y\| \|f'(x) - f(x)\| \\ &\leq \sqrt{\frac{2T}{T-1}} \|f' - f\|_\infty.\end{aligned}$$

□

Now we let the samples come into play. Let $\hat{f} \in L^\infty(\mathcal{X}, \mathbb{R}^{T-1})$ be a function of (x_i, y_i), $i = 1, \ldots, n$, such that, for every $\epsilon > 0$ and some constant $b > 0$,

$$\mathbb{P}\left\{\|\hat{f} - f\|_\infty > \epsilon\right\} \lesssim \exp(-n\epsilon^2/b^2). \tag{5.22}$$

Then the following polynomial and exponential bounds hold true.

Proposition 5.2.1

Suppose f satisfies the margin condition of Equation (5.15), and let $\hat{f}$ obey the concentration inequality (5.22). Then

$$\mathbb{E}[|\mathcal{E}(D\hat{f}) - \mathcal{E}(Df)|] \lesssim b^\alpha (\tfrac{2T}{T-1})^{\alpha/2} (\tfrac{\log n^{\alpha/2}}{n})^{\alpha/2}. \tag{5.23}$$

If f satisfies the hard-margin condition (5.16), then

$$\mathbb{E}[|\mathcal{E}(D\hat{f}) - \mathcal{E}(Df)|] \lesssim \exp(-n\delta^2/b^2). \tag{5.24}$$

Proof of Proposition 5.2.1: By Lemma 5.3 and Lemma 5.4,

$$\begin{aligned}\mathbb{E}[|\mathcal{E}(D\hat{f}) - \mathcal{E}(Df)|] &\leq \mathbb{E}[r(D\hat{f}, Df)] \\ &\leq \mathbb{E}_{\{(x_i,y_i)\}_{i=1}^n}\left[\mathbb{E}_x\left[\mathbb{1}\left\{\|\hat{f} - f\|_\infty \geq \sqrt{\frac{T-1}{2T}}M(f(x))\right\}\right]\right].\end{aligned}$$

Let $\gamma = \sqrt{\frac{T-1}{2T}}M(f(x))$ and $E = \{M(f(x)) \leq \delta\}$. Then we have

$$\begin{aligned}\mathbb{E}_x\Big[\mathbb{1}\{\|\hat{f} - f\|_\infty \geq \gamma\}\Big] &= \mathbb{E}_x\Big[\mathbb{1}\{\|\hat{f} - f\|_\infty \geq \gamma\} \mid E\ \Big]\ \mathbb{P}\{E\} \\ &\quad + \mathbb{E}_X\Big[\mathbb{1}\{\|\hat{f} - f\|_\infty \geq \gamma\} \mid E^{\complement}\ \Big]\ \mathbb{P}\Big\{E^{\complement}\Big\} \\ &\leq \mathbb{P}\{E\} + \mathbb{1}\{\|\hat{f} - f\|_\infty \geq \sqrt{\frac{T-1}{2T}}\delta\},\end{aligned}$$

where $\mathbb{P}\{E\} \lesssim \delta^\alpha$ by (5.15). Moreover, thanks to Equation (5.22),

$$\begin{aligned}\mathbb{E}_{\{(x_i,y_i)\}_{i=1}^n}\left[\mathbb{1}\left\{\|\hat{f} - f\|_\infty \geq \sqrt{\frac{T-1}{2T}}\delta\right\}\right] &= \mathbb{P}\left\{\|\hat{f} - f\|_\infty \geq \sqrt{\frac{T-1}{2T}}\delta\right\} \\ &\lesssim \exp(-n\frac{T-1}{2T}\delta^2/b^2).\end{aligned}$$

Setting $\delta^2 = b^2\frac{2T}{T-1}(\log(n^{\alpha/2})/n)$, we obtain the first claimed inequality. The second inequality follows similarly using (5.16) in place of (5.15). □

5.2.4 Main results

In this section we establish exponential convergence of plug-in classifiers under assumptions of hard margin. We assume the setting of Section 5.1, and use the arguments of Sections 5.2.1 to 5.2.3. The main results are given for two cases of loss functions, first for the square loss (namely, for the regression function), and then for a general family of margin losses. We will also be making the additional assumptions below.

Assumption 5.1. *Let $f_\ell \in L^\infty(\mathcal{X}, \mathbb{R}^{T-1})$, then*

$$\|f_\lambda - f_\ell\|_\infty \underset{\lambda}{\to} 0. \tag{5.25}$$

Further, let $\hat{f}_\lambda \in L^\infty(\mathcal{X}, \mathbb{R}^{T-1})$ be an estimate of f_λ.

Assumption 5.2. *We assume that, for every $\lambda > 0$ and some $b > 0$, the following concentration bound holds true:*

$$\mathbb{P}\Big\{\|\hat{f}_\lambda - f_\lambda\|_\infty > \epsilon\Big\} \lesssim \exp(-n\epsilon^2/b^2). \tag{5.26}$$

Regularization methods in reproducing kernel Hilbert spaces (RKHS) (Schölkopf and Smola, 2001; Steinwart and Christmann, 2008) provide one framework where the properties 5.1, 5.2 can be satisfied. In particular, one can fix a separable RKHS $\mathcal{H} \subset L^0(\mathcal{X}, \mathbb{R}^{T-1})$ with norm $\|\cdot\|_{\mathcal{H}}$, and define

$$f_\lambda = \operatorname*{arg\,min}_{f\in\mathcal{H}} \mathcal{E}_\ell(f) + \lambda\|f\|_{\mathcal{H}}, \quad \lambda \geq 0.$$

If $\mathcal{H}$ has reproducing kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ such that $\sup_x k(x,x) \leq \kappa^2$ (see Assumption 2.1), $\mathcal{H}$ is continuously embedded in the space of bounded continuous functions on $\mathcal{X}$, with $\|\cdot\|_\infty \leq \kappa\|\cdot\|_{\mathcal{H}}$. Hence, the uniform bounds needed to satisfy Assumptions 5.1 and 5.2 may be derived from bounds in the RKHS norm. The estimate $\hat{f}_\lambda$ can be computed with a variety of methods, such as empirical risk minimization (ERM) (Schölkopf and Smola, 2001), gradient descent (GD) Yao et al., 2007 and stochastic gradient descent (SGD) Robbins et al., 1951. We have seen in detail the first two ways in Chapter 2.

Lemma 5.5: Suppose Equation (5.16) holds true with $f = f_\ell$ and $\delta = \gamma$. Then, under the Assumption 5.1, there is λ_* such that Equation (5.17) holds true with $f = f_\lambda$ and $\delta = \gamma/2$ for every $\lambda \leq \lambda_*$.

Proof of Lemma 5.5: Let $Df_\ell(x) = y_* = c_*(x)$ (recall that ℓ is Fisher consistent), and let

$$a = \langle f_\lambda(x), y_*\rangle - \max_{y\neq y_*}\langle f_\lambda(x), y\rangle.$$

Then

$$a = \langle f_\ell(x), y_*\rangle - \underbrace{\langle f_\ell(x) - f_\lambda(x), y_*\rangle}_{b} - \underbrace{\max_{y\neq y_*}\langle f_\lambda(x), y\rangle}_{c},$$

where $b \leq \|f_\ell - f_\lambda\|_\infty$, and

$$\begin{aligned} c &= \max_{y\neq y_*}(\langle f_\ell(x), y\rangle + \langle f_\lambda(x) - f_\ell(x), y\rangle) \\ &\leq \max_{y\neq y_*}\langle f_\ell(x), y\rangle + \max_{y\neq y_*}\langle f_\lambda(x) - f_\ell(x), y\rangle \\ &\leq \max_{y\neq y_*}\langle f_\ell(x), y\rangle + \|f_\lambda - f_\ell\|_\infty. \end{aligned}$$

In view of Assumption 5.1, there is λ_* such that $\|f_\lambda - f_\ell\|_\infty \leq \gamma/4$. Hence, for every $\lambda \leq \lambda_*$, using (5.16) we obtain

$$\begin{aligned} a &\geq \langle f_\ell(x), y_* \rangle - \gamma/4 - \max_{y \neq y_*} \langle f_\ell(x), y \rangle - \gamma/4 \\ &= M(f_\ell(x)) - \gamma/2 \geq \gamma - \gamma/2 = \gamma/2. \end{aligned}$$

This implies (5.10), and thus (5.16), for $f = f_\lambda$ and $\delta = \gamma/2$. The assertion now follows from Lemma 5.1. $\square$

Square loss. We can now state our first main result.

Theorem 5.1

Suppose the hard-margin condition

$$\min_{y \neq c_*(x)} \rho(c_*(x) \mid x) - \rho(y \mid x) \geq \delta \quad \text{almost surely.}$$

Then, under Assumptions 5.1 and 5.2, there is λ_* such that, for every $\lambda \leq \lambda_*$,

$$\mathbb{E}[|\mathcal{E}(D\hat{f}) - \mathcal{E}(Df')|] \lesssim \exp(-n\delta^2\lambda/b^2).$$

Proof of Theorem 5.1: First, recall that, thanks to Lemma 5.2, $Df_\rho = c_*$ and $M(f_\rho(x)) \geq \delta$ almost surely. Moreover, by Lemma 5.5 (and Lemma 5.1), we have $Df_\lambda = c_*$ and $M(f_\lambda(X)) \geq \delta/2$ almost surely for $\lambda \leq \lambda_*$. Thus, (5.11) holds true, and the claim follows from Proposition 5.2.1. $\square$

Margin losses. We now consider surrogate losses of the form

$$\ell_\phi(w, y) = \phi(\langle w, y \rangle) \tag{5.27}$$

for some scalar function $\phi : \mathbb{R} \to [0, \infty)$. We denote the minimizer of the corresponding risk $\mathcal{E}_\phi(f) = \mathbb{E}[\phi(\langle f(X), Y \rangle)]$ by

$$f_\phi = \operatorname*{arg\,min}_{f \in L^0(\mathcal{X}, \mathbb{R}^{T-1})} \mathcal{E}_\phi(f). \tag{5.28}$$

Following and generalizing the analysis of T. Zhang, 2004b; Nitanda et al., 2019, we want to extract an inner risk from $\mathcal{E}_\phi$. The idea is to expand

$$\mathcal{E}_\phi(f) = \mathbb{E}_x\left[\sum_{y \in \mathcal{Y}} \phi(\langle f(x), y \rangle)\rho(y \mid x)\right]$$

and isolate the argument of $\mathbb{E}_x[\cdot]$ removing the dependence on x. Recalling the definition of Δ in Lemma 5.2, we introduce the inner risk

$$\Phi(p, w) = \sum_{y \in \mathcal{Y}} \phi(\langle w, y \rangle) p_y, \qquad p \in \Delta, w \in \mathbb{R}^{T-1}, \tag{5.29}$$

and the inner risk minimizer

$$h_\phi : \Delta \to \mathbb{R}^{T-1}, \qquad h_\phi(p) = \arg\min_{w \in \mathbb{R}^{T-1}} \Phi(p, w). \tag{5.30}$$

Note that, denoting $p(x)_y = \rho(y \mid x)$, we have

$$f_\phi(x) = h_\phi(p(x)). \tag{5.31}$$

In the following, we will be making two further assumptions:

Assumption 5.3. *ℓ_ϕ is Fisher consistent;*

Assumption 5.4. *$\langle h_\phi(p), y \rangle$ is a non-decreasing function of p_y.*

As previously mentioned, losses satisfying Assumption 5.3 are indeed abundant. For a general characterization of Fisher consistency in the framework of simplex encoded classification, we refer to Mroueh et al., 2012. Assumption 5.4 is easily met by many functions ϕ, as the next lemma shows. Essentially, it is sufficient for the loss to be decreasing and convex. Notable examples of ϕ satisfying both 5.3 and 5.4 are the logistic loss $\phi(t) = \ln(2)^{-1} \ln(1 + e^{-t})$, and the exponential loss $\phi(t) = e^{-t}$.

Lemma 5.6: Suppose ϕ is twice differentiable, non-increasing and convex. Then Assumption 5.4 holds true.

Proof of Lemma 5.6: Let $\Psi(p) = \nabla_{h_\phi} \Phi(p, h_\phi(p))$, such that

$$\Psi(p) = \sum_{i=1}^{T} \mathcal{Y}_i \phi'(\langle h(p), \mathcal{Y}_i \rangle) p_i.$$

By definition of h_ϕ, we have $\Psi(p) = 0$, hence its Jacobian $\mathrm{J}\Psi(p) = 0$ as well. Calculating the derivatives, and denoting by $\mathcal{Y}_i^{(j)}$ the j-th component of the i-th class vector, the

Jacobian matrix's entries are

$$0 = \frac{\partial \Psi(p)_j}{\partial p_k} = \sum_{i=1}^{T} \left(\mathcal{Y}_i^{(j)} p_i \frac{\partial \phi'(\langle h(p), \mathcal{Y}_i \rangle)}{\partial p_k} + \mathcal{Y}_i^{(j)} \phi'(\langle h(p), \mathcal{Y}_i \rangle) \mathbb{1}_{[k=i]} \right)$$
$$= \sum_{i=1}^{T} \left(\mathcal{Y}_i^{(j)} p_i \phi''(\langle h(p), \mathcal{Y}_i \rangle) \left\langle \mathcal{Y}_i, \frac{\partial h(p)}{\partial p_k} \right\rangle \right) + \mathcal{Y}_k^{(j)} \phi'(\langle h(p), \mathcal{Y}_k \rangle),$$

and each column is

$$\sum_{i=1}^{T} \left(p_i \phi''(\langle h(p), \mathcal{Y}_i \rangle) \mathcal{Y}_i \left\langle \mathcal{Y}_i, \frac{\partial h(p)}{\partial p_k} \right\rangle \right) + \phi'(\langle h(p), \mathcal{Y}_k \rangle) \mathcal{Y}_k.$$

For any $k \in [T]$, we can compute the following inner product, noting that it too must be equal to zero

$$0 = \left\langle \frac{\partial h(p)}{\partial p_k}, \frac{\partial \Psi(p)}{\partial p_k} \right\rangle = \sum_{i=1}^{T} \left(p_i \phi''(\langle h(p), \mathcal{Y}_i \rangle) \left\langle \mathcal{Y}_i, \frac{\partial h(p)}{\partial p_k} \right\rangle^2 \right) + \phi'(\langle h(p), \mathcal{Y}_k \rangle) \left\langle \mathcal{Y}_k, \frac{\partial h(p)}{\partial p_k} \right\rangle.$$

Since $\phi'' \geq 0$ and $\phi' \leq 0$, we must have $\langle \frac{\partial h_\phi(p)}{\partial p_k}, \mathcal{Y}_k \rangle \geq 0$, which proves the claim. □

In order to derive exponential rates for margin losses, we need to transfer the hard-margin condition from the conditional probabilities to the minimizer of the margin loss. This is the content of the following lemma.

Lemma 5.7: Suppose that (5.18) holds true with $\delta = \gamma$. Then, under the Assumption 5.4, Equation (5.16) holds true with $f = f_\phi$ and $\delta = m(\gamma)$, where

$$m(\gamma) = \max_{y,j \in \mathcal{Y}} \min\{M(h_\phi(p)) : p \in \Delta, p_y - p_j = 2\gamma\}. \tag{5.32}$$

Proof of Lemma 5.7: Let $p(X)_y = \rho(y \mid X)$. By (5.31) we have

$$M(f_\phi(X)) = M(h_\phi(p(X))).$$

Let $y, j \in \mathcal{Y}$ be such that

$$M(h_\phi(p(X))) = \underbrace{\langle h_\phi(p(X)), y \rangle}_{a} - \underbrace{\langle h_\phi(p(X)), j \rangle}_{b}.$$

In view of Equation (5.18) and Assumption 5.4, there is $p \in \Delta$ with $p_y - p_j = 2\delta$ such that a decreases and b increases, hence

$$M(h_\phi(p(X))) \geq M(h_\phi(p)).$$

Taking the minimum over such a p and the maximum over y and j, we obtain the assertion. □

To visualize the lower bound $m(\gamma)$, note that, for $T = 2$, it corresponds to $\max\{h_\phi(1/2 + \gamma), -h_\phi(1/2 - \gamma)\}$ (cf. with Nitanda et al. (2019)).

We can finally prove our main result for margin losses.

Theorem 5.2

Suppose the hard-margin condition

$$\min_{y \neq c_*(X)} \rho(c_*(X) \mid X) - \rho(y \mid X) \geq \delta \quad \text{almost surely.}$$

Then, under Assumptions 5.1 to 5.4, there is λ_* such that, for every $\lambda \leq \lambda_*$,

$$\mathbb{E}\Big[\big|\mathcal{E}(D\hat{f}) - \mathcal{E}(Df')\big|\Big] \lesssim \exp(-n\, m(\delta)^2 \lambda / b^2), \tag{5.33}$$

where $m(\delta)$ is defined in Lemma 5.7.

Proof of Theorem 5.2: By Assumption 5.3, we have $Df_\phi = c_*$ almost surely. Moreover, thanks to Lemma 5.7, we have $M(f_\phi(X)) \geq m(\delta)$ almost surely. Now, Lemma 5.5 (together with Lemma 5.1) gives that $Df_\lambda = c_*$ and $M(f_\lambda(X)) \geq m(\delta)/2$ almost surely for $\lambda \leq \lambda_*$. Therefore, we have (5.11), and Proposition 5.2.1 yields the result. □

The critical value λ_* in Theorem 5.1 and Theorem 5.2 can be quantified in presence of additional assumptions on the distribution. For example, consider the case of a kernel ridge regression estimator in a separable RKHS $\mathcal{H}$. Suppose that the kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is bounded by κ, and define the covariance operator as

$$T : \mathcal{H} \to \mathcal{H}, \qquad T = \int_{\mathcal{X}} k_x \otimes k_x \, \mathrm{d}\rho_{\mathcal{X}}(x),$$

where $k_x = k(\cdot, x)$ and $\rho_{\mathcal{X}}$ is the marginal distribution on $\mathcal{X}$. Further, suppose there exist $g \in \mathcal{H}$ and $s \in (0, 1/2]$ such that

$$f_\ell = T^s g.$$

This is known as the *source* condition, and it corresponds to assuming Sobolev smoothness of the regression function. Then, it can be proved that (Caponnetto et al., 2007)

$$\|f_\lambda - f_\ell\|_{\mathcal{H}} \leq \lambda^s \|g\|_{\mathcal{H}}.$$

As a consequence, λ_* in Lemma 5.5, and therefore in Theorem 5.1, may be picked as $\lambda_* = (\delta/4\kappa\|g\|_{\mathcal{H}})^{1/s}$.

We finally remark that analogous results to Theorem 5.1 and Theorem 5.2 may be proved under the soft-margin condition (5.19), using the polynomial bound of Proposition 5.2.1.

5.3 Experiments

This section is concerned with empirically verifying the theoretical analysis presented in Section 5.2. We will first consider a classification problem where the true function satisfies the hard-margin condition (defined in Equation (5.16)), and show how – under optimization of a surrogate loss by gradient descent – the misclassification loss decreases more quickly than the surrogate loss. Then we will take into account a different synthetic dataset, where the weaker soft-margin or low noise condition (see Equation (5.15)) is satisfied. We will verify how the rate of change of the misclassification error with the number of points in the dataset adheres to the theoretical rates.

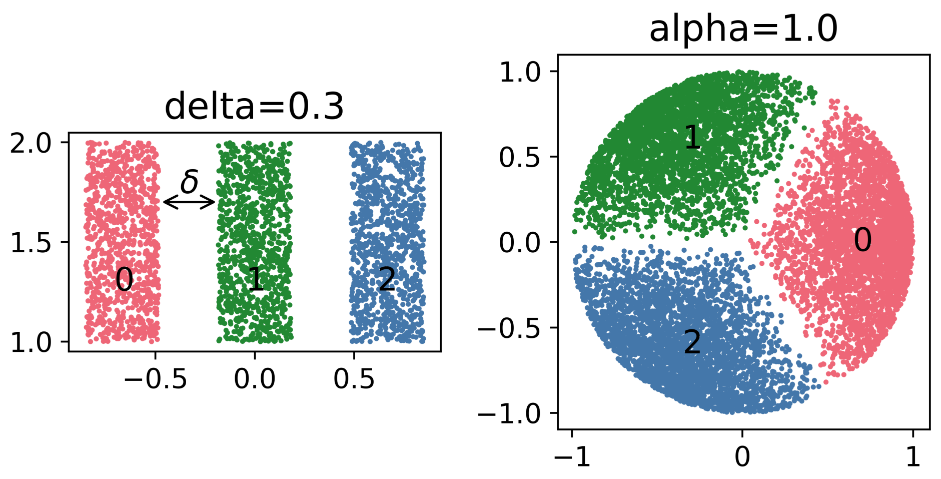


Figure 5.4 Synthetic datasets. In the left panel, the three classes are separated by a hard margin of length δ. In the right panel there is no hard margin, but the probability of a point falling close to the boundary is decreasing (soft-margin).

Initially we compare three different surrogate loss functions: the logistic, the exponential and the square loss. We generated data in two dimensions such that the hard-margin condition holds with margin δ, see Figure 5.4, left panel for a sample dataset. A random Fourier features

(RFF) model Rahimi et al., 2008 approximates potentially infinite dimensional feature maps in a reproducing kernel Hilbert space (RKHS) using finite dimensional randomized maps: given a kernel function $k(x, x') = \langle \psi(x), \psi(x') \rangle_{\mathcal{H}}$ the feature map $\psi \in \mathcal{H}$ can be approximated with function $z : \mathbb{R}^D \to \mathbb{R}^R$ such that $\langle \psi(x), \psi(x') \rangle_{\mathcal{H}} \approx \langle z(x), z(x') \rangle_{\mathbb{R}}$. Finally $z(x)$ can be used instead of the sample itself in a linear model with parameters $w \in \mathbb{R}^R$: $f(x) = w^\top z(x)$. See Section 2.5.1 for more details on the RFF model.

We learn the parameters w by minimizing the regularized surrogate loss with gradient descent. In Figure 5.5 we plot the 0-1 error, as well as the surrogate losses on unseen data as a function of the optimization epoch. A separate model was trained for each of the three surrogates 20 times with a new synthetic dataset. The intuition behind exponential rates in hard-margin classification can be verified by noting how the 0-1 loss converges at a much faster pace than the surrogate: from another perspective, when the 0-1 loss is zero the surrogate loss can still decrease for many epochs. We can further notice how not all surrogates are equal: for both the small ($\delta = 0.1$) and the larger margin ($\delta = 0.2$), the square loss leads to faster convergence of the 0-1 error than both exponential and logistic losses.

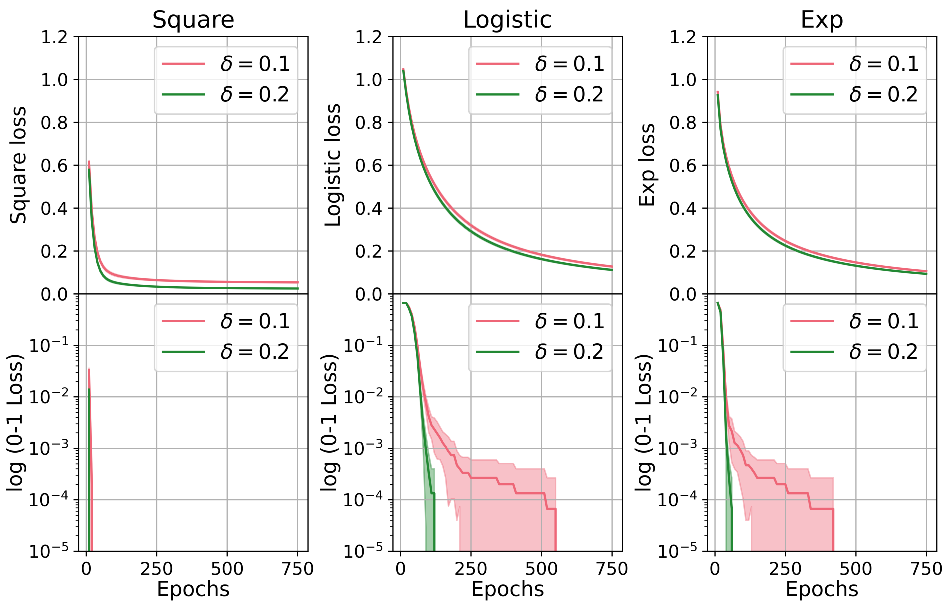


Figure 5.5 Optimization curves on hard-margin classification with different surrogate losses. Each panel contains two curves calculated on datasets with different margins δ. The top row shows the surrogate loss, the bottom row shows the 0-1 loss.

For the second experiment, we generated a synthetic dataset in two dimensions and with three classes such that the probability of a point falling close to the decision boundary decreases with the distance to the boundary itself as M^α for margins M smaller than 1 (see Equation (5.15) and Figure 5.4, right panel). We then used a linear model, trained by minimizing the regularized logistic loss with gradient descent until convergence. We repeated the experiment 100 times for datasets generated with five different values of α (a higher α results in an easier problem), and an increasing number of points, and recorded the average 0-1 loss over unseen data. We then plot the 0-1 loss against the number of points for each value of α, and observe that the trends are approximately linear on a log-log plot (see Figure 5.6). We fit a straight line for each α, and look at how the slope of this line changes with α. From Proposition 5.2.1 we expect the error to drop more rapidly with higher α; in particular the rate of decrease is predicted to be $n^{-\alpha/2}$ ignoring constant and logarithmic factors. By plotting the slopes of the error rates we obtain a straight line with slope -0.35, which is close to the prediction of -0.5 (see the inset on Figure 5.6).

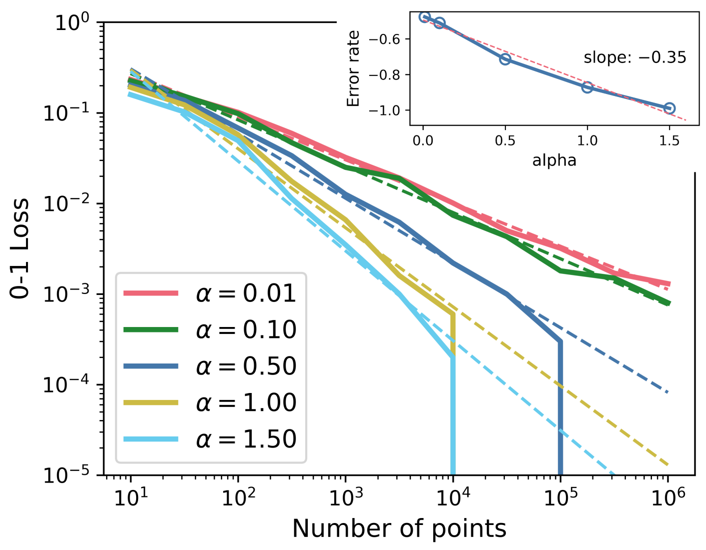


Figure 5.6 *Main figure*: error rates for multiclass classification with polynomial soft-margin with increasing dataset size. *Inset*: Linear rate of convergence of the error with α.

5.4 Conclusions

In this chapter we have shown how, under the hard-margin condition and for a very general framework which encompasses many different models and surrogate losses, the multiclass classification error exhibits exponentially fast convergence. Along the way we have provided an error decomposition where the bias term disappears. This kind of result fits with the recent empirical observations of how even highly overparametrized models do not overfit the training data. Our analysis can be experimentally verified for several losses, and different margin conditions.

Several possible extensions of this work have been left for future work. Beyond the hard-margin and low-noise conditions, robustness with respect to different kinds of noise may be studied. The explicit application of our bounds to specific models – which was sketched in this paper for kernel ridge regression – could be especially interesting for (deep) neural networks, for which fast convergence on classification problems has been ascertained. Indeed, for the latter models, the interplay of exponential convergence and overparameterization is a further topic of great interest.

Part II

Applications

Chapter 6

Fast Object Segmentation on the iCub Robot

Starting from this chapter, until the end of the thesis, we will shift our focus from the methodological contributions and theoretical analyses concerning kernel methods and supervised learning, to practical applications of kernel-based learning and other kinds of models to real-world problems. Some of the techniques developed in Part I will be used here to perform prediction tasks in complex pipelines, with the goal being not the analysis of the pipeline or algorithm itself, but instead the evaluation and interpretation of the model forecasts on unseen data-points. We will describe in some detail the settings where Falkon has been used, the practical goals of the overall system, its design, and how the algorithms we have previously discussed fit within it. For all the applications which are considered in this thesis a comprehensive review of the obtained results will be provided, along with some details on the computational efficiency of the proposed methods compared to alternative approaches. While this is not always the main focus when developing a pipeline for *e.g.* object recognition as in this chapter, we shall see that having an efficient system facilitates both more extensive experimentation, and applications in resource-limited environments.

In this chapter, we shall focus on the topic of interactions of a humanoid robot (iCub) with its surrounding environment, a fundamental problem with applications ranging from object grasping to human-robot interactions, and obstacle avoidance during navigation. Each application may pose different constraints on the robotic vision system. For example, when a robot will only interact with a predefined set of objects, fast learning is not the primary requirement. On the other hand, when a robot is operating in a dynamic environment (for instance a service robot operating in a hospital, a supermarket or a domestic environment), fast adaptation is fundamental. The concept of quickly adapting to new stimuli from the environment is the main motivation behind the work presented in this chapter.

6.1 Introduction

The *computer vision* field is progressing at a fast pace providing algorithms for object detection and segmentation that are remarkably powerful. These methods are mostly based on deep neural networks (DNNs) and are very demanding in terms of training samples and optimization time. For this reason, they are badly suited for applications in robotics that require fast adaptation. Because the dominant trend in computer vision is going towards larger and larger models, comparably little effort is spent to propose methods that are designed to reduce training time. To fill this gap, in this work, we propose a comprehensive analysis in which we study various techniques for adaptation to a novel task. In particular, we consider approaches based on deep neural networks and on a combination of DNNs and kernel methods, focusing on the trade-off between training time and accuracy.

We target the problem of instance segmentation: given an input image, classify each of its pixels as belonging to an instance of a known object or to the background. In particular, we consider the scenario in which the robot encounters new objects during its operation and is required to adapt its vision system to be able to segment them as well, after a learning session that is as short as possible. We observe that this scenario offers opportunities to shorten the training time, for example if we are able to perform some of the training steps (*i.e.*, feature extraction) already during data acquisition, and we propose a new method that is specifically optimized to reduce training time without compromising performance.

Specifically, we propose an instance segmentation pipeline which extends and improves previous work (Ceola, Maiettini, Pasquale, Rosasco, et al., 2021) in which a fast learning method for instance segmentation of novel objects was proposed. One limitation of that method was to rely on a pre-trained region proposal network. In this work, we address this by making the region proposal learning on-line too. While this improves performance, it leads to a more complex and longer training pipeline if addressed naively as it is done in Ceola, Maiettini, Pasquale, Rosasco, et al. (2020). To this aim, we propose an approximated training protocol which can be separated in two steps: 1. feature extraction and 2. fast and simultaneous training of the proposed approaches for region proposal, object detection and mask prediction. We show that this allows to further reduce the training time in the aforementioned robotic scenario.

In addition, we provide an extensive experimental analysis to investigate the training time/accuracy trade-off on two public datasets (*i.e.*, YCB-Video (Xiang et al., 2018) and HO-3D (Hampali et al., 2020)). In particular, we show that our method is much more accurate than Ceola, Maiettini, Pasquale, Rosasco, et al. (2021), while requiring a comparable training time. Moreover, the proposed method allows to obtain accuracy similar to conventional fine-tuning approaches, while being trained much faster.

In summary, the contributions of this work are:

- We propose a new pipeline and training protocol for instance based object segmentation, which is specifically designed for fast, on-line training.
- We benchmark the obtained results on two robotics datasets, namely YCB-Video (Xiang et al., 2018) and HO-3D (Hampali et al., 2020).
- We provide an extensive study to compare our pipeline against conventional fine-tuning techniques, with an in-depth analysis of the trade-off between the required training time and the achieved accuracy.
- We deploy and demonstrate the proposed training pipeline on the iCub (Metta, Natale, et al., 2010) humanoid robot, adapting the algorithm for an incremental setting where target classes are not known a-priori.

This paper is organized as follows. In Section 6.2, we review state-of-the-art approaches for instance segmentation, focusing on methods designed for robotics. Then, in Section 6.3, we describe the proposed training pipeline for fast learning of instance segmentation. In Section 6.4, we report on the experimental setup used to validate our approach. We then benchmark our approach on two benchmark datasets in Section 6.5. In Section 6.6, we specifically quantify the benefit of the adaptation of the region proposal. In Section 6.7, we simulate the robotic scenario in which data come into stream and we discuss various performance trade-offs. Then, in Section 6.8, we describe an incremental version of the proposed pipeline and we deploy it on a robotic platform. Finally, in Section 6.9 we draw conclusions.

6.2 Related Work

In this section, we provide an overview of state-of-the-art methods for instance segmentation (Section 6.2.1), focusing on their application in robotics (Section 6.2.2).

6.2.1 Instance Segmentation

Approaches proposed in the literature to address instance segmentation can be classified in the following three groups.

Detection-based instance segmentation. Methods in this category extend approaches for object detection, by adding a branch for mask prediction within the bounding boxes proposed by the detector. They can be grouped in *(i) multi-stage* (also known as *region-based*) and *(ii) one-stage*. Methods from the first group rely on detectors that first predict a set of candidate regions and then classify and refine each of them (*e.g.* Faster R-CNN (S. Ren et al., 2015) or R-FCN (J. Dai, Y. Li, et al., 2016)). *One-stage* detectors, instead, solve the object detection task in one forward pass of the network. Differently from *multi-stage* approaches, they do not

perform any per-region operation, like *e.g.* per-region feature extraction and classification (see for instance, EfficientDet (Tan et al., 2020) and YOLOv3 (Redmon et al., 2018)).
The representative method among the *multi-stage* approaches is Mask R-CNN (He, Gkioxari, et al., 2017) that builds on top of the detection method Faster R-CNN (S. Ren et al., 2015), by adding a branch for mask prediction (segmentation branch) in parallel to the one for bounding box classification and refinement (detection branch). In Mask R-CNN, input images are initially processed by a convolutional backbone to extract a feature map. This is then used by the Region Proposal Network (RPN) to propose a set of *Regions of Interest* (*RoIs*) that are candidate to contain an object, by associating a class-agnostic objectness score to each region. Then, the *RoI Align* layer associates a convolutional feature map to each *RoI* by *warping* and *cropping* the output of the backbone. These features are finally used for *RoIs* classification, refinement and, subsequently, for mask prediction. In the literature, many other state-of-the-art *multi-stage* approaches for instance segmentation build on top of Mask R-CNN, like Mask Scoring R-CNN (Huang et al., 2019) or PANet (S. Liu et al., 2018).
YOLACT (Bolya et al., 2019) and BlendMask (H. Chen et al., 2020) are representative of *one-stage* methods. YOLACT extends a backbone RetinaNet-like (Lin, Goyal, et al., 2017) detector with a segmentation branch. BlendMask, instead, extends FCOS (Tian et al., 2019) for mask predictions. An alternative paradigm for instance segmentation based on the *one-stage* detector CenterNet (Zhou et al., 2019) is Deep Snake (S. Peng, Jiang, et al., 2020). Differently from the methods mentioned above that predict per-pixel confidence within the proposed bounding boxes, it exploits the circular convolution (S. Peng, Jiang, et al., 2020) to predict an offset for each mask vertex point, starting from an initial coarse contour.
Labeling pixels followed by clustering. Approaches in this group build on methods for semantic segmentation, which is the task of classifying each pixel of an image according to its category (being thus agnostic to different object instances). Building on these methods, approaches in the literature separate the different instances by clustering the predicted pixels. As an example, SSAP (N. Gao et al., 2019) uses the so-called *affinity pyramid* in parallel with a branch for semantic segmentation to predict the probability that two pixels belong to the same instance in a hierarchical manner. This is done with the aim of grouping pixels of the same instance. InstanceCut (Kirillov et al., 2017), instead, exploits an instance-agnostic segmentation and an instance-aware edge predictor to compute the instance-aware segmentation of an image. Finally, the method proposed in Bai et al. (2017) learns the watershed transform with a convolutional neural network, the *Deep Watershed Transform*, given an image and a semantic segmentation. This is done to predict an energy map of the image, where the energy basins represent the object instances. This information is then used, with a cut at a single energy level, to produce connected components corresponding to different object instances.
Dense sliding window. These approaches simultaneously predict mask instances and their class-agnostic or class-specific scores. For instance, DeepMask (Pinheiro et al., 2016) predicts

in parallel a class-agnostic mask and an objectness score for each patch of an input image with a shallow convolutional neural network. InstanceFCN (J. Dai, He, et al., 2016), alternatively, predicts an instance sensitive score map for each window of the considered input image. This method exploits local coherence for class-agnostic masks prediction, and, as DeepMask, per-window class-agnostic scores. Similarly, TensorMask (X. Chen et al., 2019) predicts class-agnostic instance masks, but it leverages on the proposed mask representation as a 4D tensor to preserve the spatial information among pixels. Moreover, the classification branch of the proposed approach outputs a class-specific score, thus improving the class-agnostic predictions provided by DeepMask and InstanceFCN.

6.2.2 Instance Segmentation in Robotics

The instance segmentation task plays a central role in robotics, not only for providing an accurate 2D scene description for a robot, but also to support other tasks like 6D object pose estimation (Xiang et al., 2018) or computation of grasp candidates (Shu et al., 2018). In the literature, the problem is tackled in different ways, depending on the target application. In Wada et al. (2019) and A. Li et al. (2020) the problem is addressed in cluttered scenarios, while S. Li et al. (2020) and Danielczuk et al. (2019) propose adopting synthetic data (both images and depth information) for training. In this work, instead, we focus on learning to segment previously unseen objects. In the following paragraphs, we will cover the main literature on this topic.

Some works propose to generalize to unseen objects in a class-agnostic fashion. However, these methods either focus on particular environments, such as tabletop settings, as in Xie et al. (2020) and Xie et al. (2021), or require some post-processing (Kuo et al., 2019) which may be unfeasible during the robot operation.

Approaches as the ones proposed in Pathak et al. (2018) and Eitel et al. (2019) learn to segment new object instances by interacting with them. Nevertheless, similarly to the class-agnostic approaches, they are constrained to tabletop settings.

The latest literature on *Video Object Segmentation* provides some methods for learning to segment a set of previously unseen objects in videos. They deal with the problem either in a semi-supervised way (P. Zhang et al., 2020), leveraging on the ground-truth masks of the objects in the first frame of the video, or in an unsupervised fashion (S. Garg et al., 2020). They allow to learn to segment new object instances in a shorter time than that required by the fully supervised approaches presented in Section 6.2.1. They typically rely on pretraining a network for instance segmentation and on the subsequent fine-tuning on the target video sequence frames (Voigtlaender et al., 2017). Some of these approaches have been targeted for robotic scenarios. For instance, the method in Siam et al. (2019) proposes to learn to segment novel objects in a *Human-Robot Interaction (HRI)* application, leveraging only on objects motion cues. Nevertheless, these approaches are known to suffer from changes of the objects appearance through the video sequence and error drifts (P. Zhang et al., 2020).

We instead focus on learning to segment novel objects in a class-specific fashion, keeping the performance provided by the state-of-the-art but reducing the required training time. All the approaches mentioned in Section 6.2.1 rely on convolutional neural networks that require to be trained end-to-end via backpropagation and stochastic gradient descent. Despite providing impressive performance, they require long training time and large amounts of labeled images to be optimized. These constraints make the adoption of such approaches in robotics difficult, especially for robots operating in unconstrained environments, that require fast adaptation to new objects.

Incremental learning aims at learning new object instances without degrading performance on the previously known classes. Nevertheless, these approaches rarely focus on speeding-up the training of the models, which may be crucial in robotic applications. Moreover, the current literature in this field mainly focuses on object recognition (Camoriano, Pasquale, et al., 2017; Maltoni et al., 2019), object detection (Shmelkov et al., 2017; Perez-Rua et al., 2020) or semantic segmentation problems (Michieli et al., 2019), while we target an instance segmentation application. As we show in Section 6.8, we deploy the proposed pipeline on the iCub humanoid robot, adapting it to an incremental setting, where the target classes are not known a-priori.

In this work, we propose a pipeline and a training protocol for instance segmentation which is specifically designed to reduce training time, while preserving performance as much as possible. This approach is based on Mask R-CNN (He, Gkioxari, et al., 2017), in which the final layers of the RPN and of the detection and segmentation branches have been replaced with "shallow" classifiers based on a fast kernel-based method optimized for large scale problems (Rudi, Carratino, et al., 2017; Meanti, Carratino, Rosasco, et al., 2020). The backbone of the network is trained off-line, while the kernel-based classifiers are adapted on-line. In this paper, we build on top of Ceola, Maiettini, Pasquale, Rosasco, et al. (2021), in that we include the adaptation of the region proposal network and a novel training protocol which allows to further reduce the training time. This makes the pipeline suitable for on-line implementation.

6.3 Methods

The proposed hybrid pipeline allows to quickly learn to predict segmentation masks of previously unseen objects (TARGET-TASK). We rely on a *pretrained* convolutional neural network, which was trained using a large number of samples coming from a different task (FEATURE-TASK). This larger network is used for feature extraction, on top of which we developed three modules for region proposal, object detection and mask prediction which can be rapidly adapted on the new task. This allows to achieve on-line adaptation on novel objects and visual scenarios.

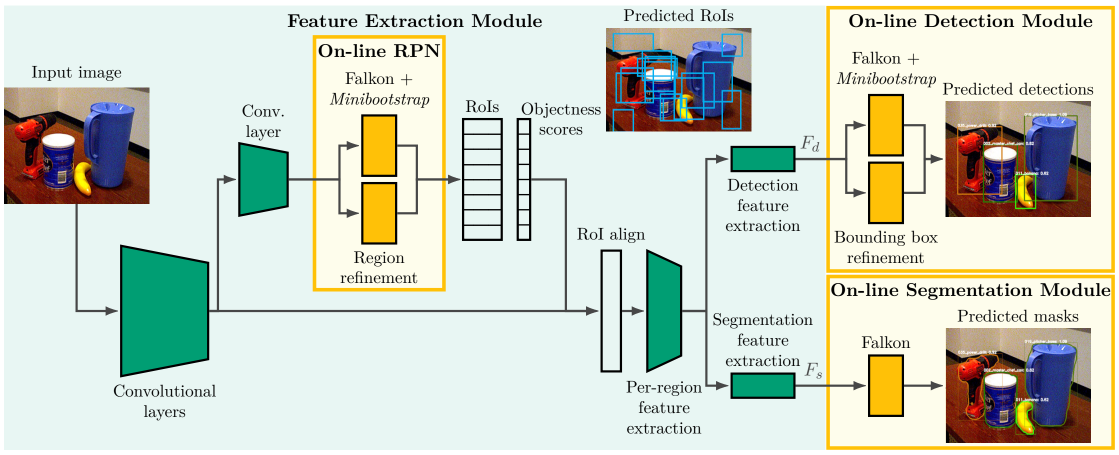


Figure 6.1 The *Feature Extraction Module* is composed of Mask R-CNN's first layers trained off-line on the FEATURE-TASK. The three sets of features for *(i)* region proposal (F_r), *(ii)* object detection (F_d) and *(iii)* instance segmentation (F_s) are fed to *(i)* the *On-line RPN*, *(ii)* the *On-line Detection Module* and *(iii)* the *On-line Segmentation Module*. At inference time, we substitute the final layers of the Mask R-CNN's RPN with the *On-line RPN* trained on the TARGET-TASK and, as in Mask R-CNN, the output of the *On-line Detection Module* is fed as input to the *RoI Align* to compute the objects masks within the proposed bounding boxes.

The proposed pipeline is composed of four modules, which are depicted in Figure 6.1. They are:

- **Feature Extraction Module.** This consists of the first layers of Mask R-CNN, which has been pretrained off-line on the FEATURE-TASK. We use it to extract the convolutional features to train the three on-line modules on the TARGET-TASK. In particular, we use it to extract the features F_r, F_d and F_s from the penultimate layers of the RPN, and of the detection and segmentation branches, respectively.

- **On-line RPN.** This replaces the last layers of the Mask R-CNN's RPN to predict a set of regions that likely contain an object in an image (often called regions of interest or RoIs), given a feature map F_r. We describe the training procedure in Section 6.3.1.

- **On-line Detection Module.** This is used to classify features F_d corresponding to regions proposed by *On-line RPN* into their respective object types (*e.g. banana* vs. *mug*). See Section 6.3.1 for the description of the training procedure.

- **On-line Segmentation Module.** Given a feature map F_s, this last module predicts the masks of objects within the RoIs provided by the *On-line RPN*. It can be trained in parallel to the *On-line Detection Module*, as it doesn't rely on knowledge of the object classes. We describe the training procedure in Section 6.3.3.

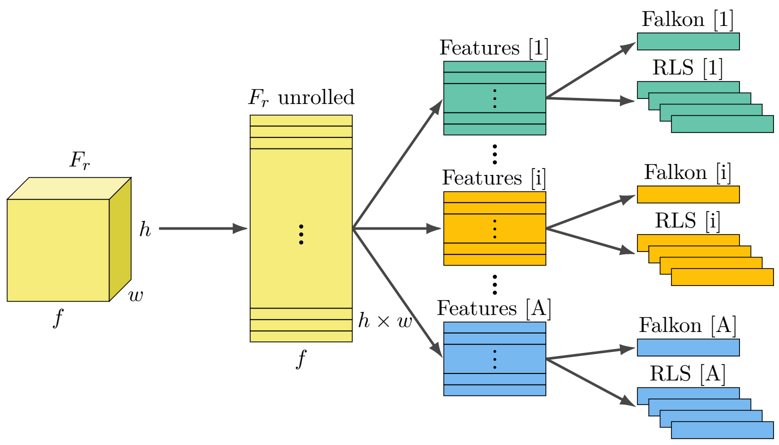


Figure 6.2 On-line RPN. Given the feature map F_r, this is unrolled into $h{\times}w$ tensors of features of size f (F_r *Unrolled*). A subset of these features is chosen to train a Falkon classifier and four RLS regressors for each anchor.

In the three on-line modules described above, we use the Falkon algorithm for classification. This is a kernel-based method optimized for large-scale problems (Rudi, Carratino, et al., 2017). In particular, we use the implementation available in Meanti, Carratino, Rosasco, et al. (2020).

6.3.1 Bounding Box Learning

Given as input the convolutional features computed by Mask R-CNN's backbone, we first process them with a convolutional layer to reduce feature dimensionality. The output of this layer consists of $h \times w$ features of size f representing a whole image (the feature map F_r). Then at each i, j location on this 2-dimensional grid we shall propose k boxes (or *anchors* S. Ren et al., 2015) of predefined sizes and aspect ratios to be the RoIs associated with each pixel. Since this produces a large number of spurious regions by design, the RPN's goal is to *(i)* classify each region as either *object* or *background*, and *(ii)* refine the preset boxes to more accurately bound any object they might contain. Mask R-CNN's RPN uses convolutional layers for both tasks. In contrast, for the first task we adopt k Falkon models trained as binary classifiers on top of the features F_r. The training samples are naturally imbalanced with many more negative (= background) than positive (= object or foreground) samples (Lin, Goyal, et al., 2017), so we adopt the *mini-bootstrap* strategy proposed in Maiettini et al., 2018; Maiettini et al., 2019. As positive examples we adopt those predefined bounding boxes whose *Intersection over Union* (IoU: a measure of overlap between bounding boxes) with a ground-truth bounding box is greater than 0.7, as negative examples instead we take those bounding boxes whose overlap with any ground-truth sample is less than 0.3[1]. We formulate the second task as continuous valued regression with 4 outputs corresponding to relative offsets for the four sides of each

[1]For the *On-line RPN*, we set the positive and negative thresholds for the classifiers as in Mask R-CNN's RPN He, Gkioxari, et al., 2017.

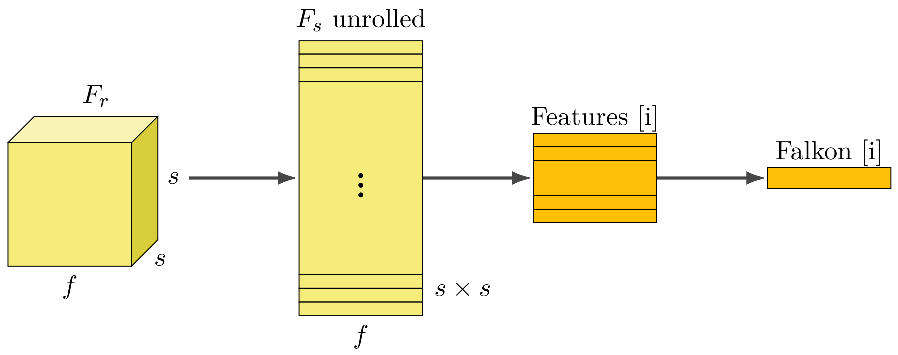


Figure 6.3 On-line Segmentation. Given the feature map F_s associated to a *RoI* of class i, this is unrolled into $s{\times}s$ tensors of features of size f (F_s *Unrolled*) from which positive and negative features are sampled to train the i^{th} Falkon per-pixel classifier. Note that this procedure is performed for each *RoI* of the N classes.

bounding box. Correspondingly we train $4k$ linear regression models (Girshick et al., 2014), using only the proposed boxes with an IoU greater than 0.6 with ground-truth samples. The pipeline is depicted in Figure 6.2.

6.3.2 On-line object detection

For training the *On-line Detection Module*, we consider features produced by the penultimate layer of the Mask R-CNN's detection branch (F_d), associated to each RoI proposed during the region proposal step. Considering a TARGET-TASK with N classes, we will use the Falkon algorithm to train N binary classifiers which learn how to distinguish the n-th class from the rest. For the n-th classifier, we take as positive samples those RoIs whose IoU with a ground-truth box belonging to class n is greater than 0.6[2]. As negative samples we consider RoIs with IoU with any ground-truth box from class n less than 0.3[3].

6.3.3 On-line Segmentation

In Mask R-CNN the segmentation branch is a two-layer fully convolutional network (FCN) that takes as input a feature map of size $s{\times}s{\times}f$ associated to each RoI. The first layer acts another feature-extraction layer, and is preserved in our architecture. The second layer is a convolutional layer with N output channels and the same spatial resolution as the input RoI which provides the model's confidence that any given pixel corresponds to the n-th class. We replace this layer with N Falkon binary classifiers taking f-dimensional features corresponding to each pixel (see Figure 6.3) as input and outputting the probability that such pixel belongs to a specific class. Here we consider as positive samples the features associated with pixels in

[2]We consider as positive samples for the classifiers in the *On-line Detection Module* the training features for region refinement as in He, Gkioxari, et al., 2017.

[3]For the classifiers in the *On-line Detection Module* we define the negative samples as in Girshick et al., 2014.

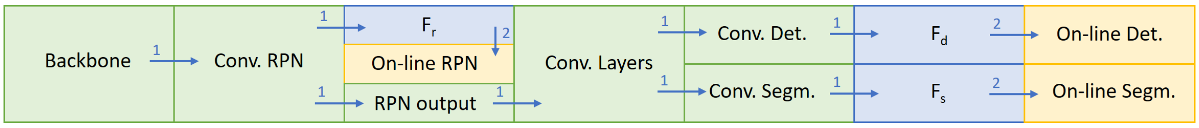


Figure 6.4 **Ours** training protocol. We rely on the feature extraction layers of Mask R-CNN pretrained on the FEATURE-TASK to simultaneously extract F_r, F_d and F_s. We then use these features to train the three on-line modules on the TARGET-TASK. The values on the arrows correspond to the training steps in Section 6.3.4.

the ground-truth masks for each class, and as negative samples the features corresponding to the background pixels. to speed-up the training procedure, we randomly subsample both the positive and the negative features by a factor r. According to the analysis provided in Ceola, Maiettini, Pasquale, Rosasco, et al. (2021), we set r to 0.3.

6.3.4 Training Protocol

In this work, we propose a training protocol that allows to quickly update the *On-line RPN*, the *On-line Detection Module* and the *On-line Segmentation Module*. The proposed method (referred to as **Ours**) starts with the weights of Mask R-CNN pretrained on the FEATURE-TASK and adapts the on-line modules on the TARGET-TASK, as depicted in Figure 6.4:

1. *Feature extraction.* This is done with a forward pass of the pretrained Mask R-CNN feature extractor (in green) to compute F_r, F_d and F_s (depicted in blue).

2. *On-line training.* The set of features F_r, F_d and F_s are used to train the three on-line modules represented in yellow in Figure 6.4 on the TARGET-TASK.

Note that, in order to train the on-line modules in parallel, the training features fed to the *On-line Detection Module* must be those proposed by Mask R-CNN's RPN pretrained on FEATURE-TASK. These are different and sub-optimal compared the ones proposed by the *On-line RPN* which has been adapted on TARGET-TASK. In Section 6.6.2 we perform an ablation experiment comparing this training procedure against *serial training* of the *On-line RPN* followed by the detection module, showing that the price to pay for efficient training is small in terms of accuracy.

At inference time, features fed to the *On-line Detection Module* and to the *On-line Segmentation Module* are those associated to the regions proposed by the *On-line RPN* trained on the TARGET-TASK, as depicted in Figure 6.1.

6.4 Experimental Setup

In this section, we report on the experimental settings that we employ for validating the proposed approach.

Off-line Experiments For our experiments, we compare the proposed method, **Ours**, with two Mask R-CNN (He, Gkioxari, et al., 2017) baselines. In particular, we consider:

- **Mask R-CNN (output layers)**: starting from the Mask R-CNN weights pre-trained on FEATURE-TASK, we fine-tune the output layers of the RPN and of the detection and segmentation branches on the TARGET-TASK, freezing all the other weights of the network.
- **Mask R-CNN (full)**: we use the weights of the pretrained Mask R-CNN as a warm-restart to train Mask R-CNN on the TARGET-TASK.

Specifically we rely on Mask R-CNN (He, Gkioxari, et al., 2017) with ResNet-50 (He, X. Zhang, et al., 2016) as backbone for the feature extraction of **Ours** and for the baselines.

In all cases, we choose hyper-parameters providing the highest performance on a validation set. Specifically, for **Ours** we cross-validate the length-scale of Falkon's Gaussian kernel (γ) and regularization parameter (λ) for each module in which kernel methods were used. Regarding the baselines, we train **Mask R-CNN (output layers)** and **Mask R-CNN (full)** for the number of epochs that provides the highest segmentation accuracy on the validation set. For **Ours**, we set the number of Nyström centers M of the Falkon classifiers composing the *On-line RPN*, the *On-line Detection Module* and the *On-line Segmentation Module* to 1000, 1000 and 500, respectively. Moreover, to train both the *On-line RPN* and the *On-line Detection Module*, we set to 2000 the batch-size (BS) considered in the *Minibootstrap*.

Evaluation metrics. We consider the *mean Average Precision (mAP)* as defined in Everingham et al. (2010) for both object detection and segmentation. Specifically, the accuracy of the predicted bounding boxes will be referred to as **mAP bbox(%)** and the accuracy of the mask instances as **mAP segm(%)**. For each of them, we consider as positive matches the bounding boxes and the masks whose IoU with the ground-truths is greater or equal to a threshold. In our experiments we consider two different thresholds to evaluate different levels of accuracy, namely, 50 % (**mAP50**) and 70 % (**mAP70**). We also evaluate the training time required for each method[4]. For the Mask R-CNN baselines, training time is the time needed for their optimization via stochastic gradient descent. For **Ours** instead, except where differently specified, it is the time necessary for extracting the features and training the on-line modules. For each experiment we run three trials and report average and standard deviation of the accuracy and average training time.

Datasets We used MS COCO (Lin, Maire, et al., 2014) as FEATURE-TASK and YCB-Video (Xiang et al., 2018) and HO-3D (Hampali et al., 2020) as TARGET-TASKs to validate

[4]All the *off-line experiments* have been performed on a machine equipped with Intel(R) Xeon(R) W-2295 CPU @ 3.00GHz, and a single NVIDIA Quadro RTX 6000.

our approach. We opted to validate our system on these datasets, which are composed of streams of frames in tabletop and hand-held settings, to be close to our target application. These datasets are usually considered for the task of 6D object pose estimation, however they are annotated also with object masks. Specifically:

- **MS COCO** (Lin, Maire, et al., 2014) is a general-purpose dataset which contains 80 objects categories for object detection and segmentation.
- **YCB-Video** (Xiang et al., 2018) is a dataset for 6D pose estimation in which 21 objects from the YCB (Calli et al., 2015) dataset are arranged in cluttered tabletop scenarios, therefore presenting strong occlusions. It is composed of video sequences where the tabletop scenes are recorded under different viewpoints. We use as training images a set of 11 320 images, obtained by extracting one image every ten from the total 80 training video sequences. As test set, instead, we consider the 2949 *keyframe* (Xiang et al., 2018) images chosen from the remaining 12 sequences. For hyper-parameters cross-validation, we randomly select a subset of 1000 images from the 12 test sequences, excluding the *keyframe* set.
- **HO-3D** (Hampali et al., 2020) is a dataset for hand-object pose estimation, in which objects are a subset of the ones in **YCB-Video**. It is composed of video sequences in which a moving hand-held object is shown to a fixed camera. For choosing the training and test sets, we split the available annotated sequences in HO-3D[5] such that we gather one and at most four sequences for testing and training, respectively. In particular, we use 20 156 images as training set, which result from the selection of one every two images from 34 sequences. Instead, we consider as test set 2020 images chosen one every five frames taken from other 9 sequences. For validation we consider 2160 frames chosen one every five images, from a subset of 9 sequences taken from the training set.

Robotic Setup We deploy the proposed pipeline for on-line instance segmentation on the humanoid robot iCub[6] (Metta, Natale, et al., 2010). It is equipped with a *Intel(R) RealSense D415* on a headset for the acquisition of RGB images and depth information. We rely on the YARP (Metta, Fitzpatrick, et al., 2006) middleware for the implementation and the communication between the different modules (see Section 6.8). With the exception of the proposed one, we rely on publicly available modules[7]. We set all training hyper-parameters as described in Section 6.4.

[5]Note that in HO-3D the annotations for instance segmentation are not provided for the test set. Therefore, we extract training and test sequences from the original HO-3D training set.

[6]We run the module with the proposed method on a machine equipped with Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz, and a single NVIDIA RTX 2080 Ti.

[7]https://github.com/robotology

Table 6.1 Benchmark on the YCB-Video dataset. We compare the proposed approach **Ours** to the baseline **Mask R-CNN (output layers)** and to the upper bound **Mask R-CNN (full)**.

Method	**mAP50 bbox(%)**	**mAP50 segm(%)**	**mAP70 bbox(%)**	**mAP70 segm(%)**	**train time**
Mask R-CNN (full)	89.66±0.47	91.26±0.56	84.67±0.81	80.26±0.59	96 min
Mask R-CNN (output layers)	84.51±0.40	81.70±0.17	75.81±0.30	70.46±0.24	177 min
Ours	83.66±0.84	83.06±0.92	72.97±1.02	68.11±0.29	14 min

Table 6.2 Benchmark on the HO-3D dataset. We report performance obtained with **Ours** and compare it to **Mask R-CNN (output layers)** and **Mask R-CNN (full)**.

Method	**mAP50 bbox(%)**	**mAP50 segm(%)**	**mAP70 bbox(%)**	**mAP70 segm(%)**	**train time**
Mask R-CNN (full)	92.21±0.88	90.70±0.17	86.73±0.71	77.25±0.62	39 min
Mask R-CNN (output layers)	88.05±0.32	86.11±0.29	74.75±0.19	65.04±0.62	111 min
Ours	83.63±1.64	84.50±1.63	63.33±1.65	61.54±0.33	17 min

6.5 Results

In this section, we benchmark the proposed approach on YCB-Video (Section 6.5.1) and HO-3D (Section 6.5.2).

6.5.1 Benchmark on YCB-Video

We consider the 21 objects from YCB-Video as TARGET-TASK and compare the performance of **Ours** against the baseline **Mask R-CNN (output layers)**. We also report the performance of **Mask R-CNN (full)**, which can be considered as an upper-bound because, differently from the proposed method, it updates both the feature extraction layers and the output layers (*i.e.* the backbone, the RPN, the detection and the segmentation branches). In **Ours**, we empirically set the number of batches in the *Minibootstrap* to 10, which achieves the best time/accuracy trade-off (see Figure 6.6 for details).

Results in Table 6.1 show that **Ours** achieves similar performance as **Mask R-CNN (output layers)** in a fraction ($\approx$ 12.8$\times$ smaller) of the training time. On the other hand, **Ours** is not as accurate as **Mask R-CNN (full)** ($\approx$ 9.0% less precise if we consider the **mAP50 segm(%)**) metric, but is trained $\approx$ 6.9$\times$ faster.

Table 6.3 Comparison between **Ours**, **Mask R-CNN (full)** and **O-OS** trained on YCB-Video. For **O-OS**, we reproduce the experiment of Tab. I in Ceola, Maiettini, Pasquale, Rosasco, et al. (2021), but run the experiment three times (reporting mean and standard deviation) on the same hardware used for this work and setting training hyperparameters as described in Section 6.6.1.

Method	**mAP50 bbox(%)**	**mAP50 segm(%)**	**mAP70 bbox(%)**	**mAP70 segm(%)**	**train time**
Mask R-CNN (full)	89.66±0.47	91.26±0.56	84.67±0.81	80.26±0.59	96 min
O-OS	76.15±0.31	74.44±0.11	68.06±0.34	63.90±0.36	11 min
Ours	83.66±0.84	83.06±0.92	72.97±1.02	68.11±0.29	14 min

Table 6.4 We report on the performance obtained on HO-3D with **Ours** and we compare it to **Mask R-CNN (full)** and **O-OS** for the analysis in Section 6.6.1.

Method	**mAP50 bbox(%)**	**mAP50 segm(%)**	**mAP70 bbox(%)**	**mAP70 segm(%)**	**train time**
Mask R-CNN (full)	92.21±0.88	90.70±0.17	86.73±0.71	77.25±0.62	39 min
O-OS	75.27±0.26	77.42±0.45	57.89±0.24	57.86±0.21	14 min
Ours	83.63±1.64	84.50±1.63	63.33±1.65	61.54±0.33	17 min

6.5.2 Benchmark on HO-3D

On the HO-3D dataset, we empirically set the number of minibootstrap batches of the *On-line RPN* and of the *On-line Detection Module* in **Ours** to 12. Obtained results are in Table 6.2.

Similarly to the experiment on YCB-Video, **Ours** can be trained $\approx 2.3\times$ and $\approx 6.6\times$ faster than **Mask R-CNN (full)** and **Mask R-CNN (output layers)**, respectively. Models obtained with **Ours** are slightly less precise than those provided by **Mask R-CNN (output layers)** for the task of instance segmentation, while they are $\approx 15.3\%$ less accurate if we consider the **mAP70 bbox(%)**. We will show in Section 6.6.2 that this gap can be recovered with a different training protocol. However, **Ours** achieves the best training time with an accuracy that is close to the state-of-the-art.

6.6 Fast Region Proposal Adaptation

In this section, we investigate the impact of region proposal adaptation on the overall performance. In particular, in Section 6.6.1, we show that, with respect to Ceola, Maiettini, Pasquale, Rosasco, et al. (2021), updating the RPN provides a significant gain in accuracy, maintaining a comparable training time. Then, in Section 6.6.2 we analyze the speed/accuracy trade-off achieved with the proposed approximated training protocol.

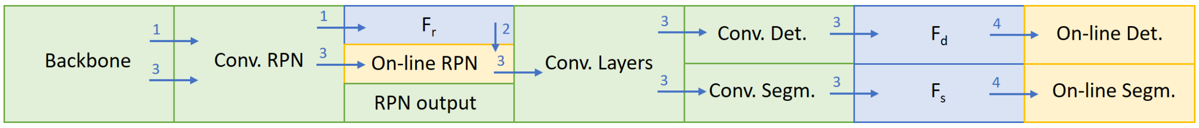


Figure 6.5 **Ours serial** training protocol. We rely on the feature extraction layers of Mask R-CNN pre-trained on the FEATURE-TASK to extract F_r and we train the *On-line RPN* on the TARGET-TASK. Then, we rely on the feature extraction layers of Mask R-CNN and on the *On-line RPN* trained on the TARGET-TASK to extract F_d and F_s. Finally, we train the *On-line Detection Module* and the *On-line Segmentation Module* on the TARGET-TASK. The values on the arrows correspond to the training steps in Section 6.6.2.

6.6.1 Is Region Proposal Adaptation Key to Performance?

Adaptation of the region proposal algorithm on a new task provides a significant gain in accuracy for object detection on the new task itself (in this paper we report some evidence while additional experiments can be found in Ceola, Maiettini, Pasquale, Rosasco, et al. (2020)). In particular it is especially effective when FEATURE-TASK and TARGET-TASK present a significant domain shift (which is a common scenario in robotics). In this section, we show that better region proposals also improve downstream mask estimation.

To test performance under domain shift we consider as FEATURE-TASK the categorization task in MS COCO and as TARGET-TASKs the identification tasks of the YCB-Video and HO-3D datasets, which depict tabletop and in-hand scenarios respectively.

Once again we consider **Mask R-CNN (full)** as the upper bound of achievable performance. We compare **Ours** with the method proposed in Ceola, Maiettini, Pasquale, Rosasco, et al. (2021) (Sec. III), namely **O-OS**[8], in which the RPN remains constant during training on the TARGET-TASK. For a fair comparison, we set **O-OS** hyperparameters according to Section 6.4.

Results in Tables 6.3 and 6.4 show that, as expected, there is an accuracy gap between **Mask R-CNN (full)** and the other considered methods. However, notably, the adaptation of the region proposal on the TARGET-TASK in **Ours** allows to significantly reduce the accuracy gap between **Mask R-CNN (full)** and **O-OS**. Moreover, **Ours** outperforms the accuracy of **O-OS** with a comparable training time. For instance, in the HO-3D experiment, the segmentation **mAP50** obtained with **Ours** is, on average, ≈ 7.1 points greater than **O-OS**, with a difference in training time of only 3 m 20 s.

6.6.2 Approximated On-line Training: Speed/Accuracy Trade-off

To evaluate the impact of training the *On-line Detection* module on RoIs proposed by the non-adapted model, we compare against a different training protocol, referred to as **Ours serial**, where the *On-line RPN* is trained first and the other modules are trained later on top of its proposals. This should allow to train the detection module with better RoIs, improving

[8]In Ceola, Maiettini, Pasquale, Rosasco, et al. (2021), **O-OS** was referred to as **Ours**.

Table 6.5 Comparison between the proposed approach **Ours**, the baseline **Mask R-CNN (output layers)** and **Ours serial** trained on YCB-Video.

Method	**mAP50 bbox(%)**	**mAP50 segm(%)**	**mAP70 bbox(%)**	**mAP70 segm(%)**	**train time**
Mask R-CNN (full)	84.51±0.40	81.70±0.17	75.81±0.30	70.46±0.24	177 min
Ours serial	83.97±0.59	83.00±0.78	75.06±0.88	69.12±0.56	25 min
Ours	83.66±0.84	83.06±0.92	72.97±1.02	68.11±0.29	14 min

the overall pipeline performance. In more detail, **Ours serial** is composed of the four steps depicted in Figure 6.5:

1. Feature extraction for region proposal. This is done to extract F_r (see Section 6.3) on the images of the TARGET-TASK.

2. These features are then used to train the *On-line RPN* on the TARGET-TASK, as described in Section 6.3.1.

3. The new *On-line RPN* is used to extract more precise regions and the corresponding features for detection and segmentation (respectively, F_d and F_s).

4. F_d and F_s are used to train the *On-line Detection Module* and the *On-line Segmentation Module* on the TARGET-TASK, as described in Sections 6.3.1 and 6.3.3 respectively.

We evaluate **Ours** and **Ours serial** in the same setting used for previous experiments (Sections 6.5 and 6.6.1), and using the same procedure for deciding hyperparameters. The number of *minibootstrap* iterations for **Ours serial** is set to 8 and 7 in the experiments on YCB-Video and HO-3D, respectively.

We report results in Tables 6.5 and 6.6. Accuracy of **Ours** in the YCB-Video experiment is comparable to the one of **Ours Serial**, demonstrating that the approximated training procedure substantially does not affect performance in this case. Instead, in the HO-3D experiment, **Ours** is slightly less precise than **Ours serial** for the task of instance segmentation, while being $\approx 11.6\%$ less accurate if we consider the **mAP70 bbox(%)**. However, **Ours** is trained $\approx 1.8\times$ and $\approx 2.2\times$ faster than **Ours serial** on YCB-Video and HO-3D, respectively. Nonetheless **Ours serial** achieves comparable performance to **Mask R-CNN (output layers)** with much shorter training time. However, the approximate training protocol proposed in this paper allows further optimization which is discussed in the next section.

Table 6.6 We report on the performance obtained on HO-3D with **Ours** and we compare it to **Mask R-CNN (output layers)** and **Ours serial**.

Method	**mAP50 bbox(%)**	**mAP50 segm(%)**	**mAP70 bbox(%)**	**mAP70 segm(%)**	**train time**
Mask R-CNN (output layers)	88.05±0.32	86.11±0.29	74.75±0.19	65.04±0.62	111 min
Ours serial	88.70±0.43	87.87±0.37	71.65±0.93	64.76±0.70	37 min
Ours	83.63±1.64	84.50±1.63	63.33±1.65	61.54±0.33	17 min

6.7 Stream-based Instance Segmentation

We now consider a robotic application, in which the robot is tasked to learn new objects on-line, while automatically acquiring training samples. In this case, training data arrive continuously in a stream, and the robot must either use them immediately or store them for later use. We investigate to what extent it is possible to reduce the training time and how this affects segmentation performance.

Because data acquisition takes a considerable amount of time, we exploit the opportunity to perform some of the processing required for training in parallel. In the proposed pipeline, for example, the training protocol **Ours** has been designed to separate feature extraction from training of the kernel-based components. Then the expensive feature extraction step can be performed while images and ground-truth labels are being received by the robot. In this section, we investigate to what extent it is possible to exploit this parallelization with the conventional Mask R-CNN architecture. We compare the proposed **Ours** with three different Mask R-CNN baselines: **Mask R-CNN (full)** and two variations of **Mask R-CNN (output layers)** as presented in Section 6.4.

Because images arrive in a stream, similar views of the same objects are represented in adjacent frames. This requires storing all images and waiting until the end of the data acquisition process, before starting the training process. We hence consider an additional baseline, **Mask R-CNN (store features)**, in which, similarly to **Mask R-CNN (output layers)**, we fine-tune the output layers of the RPN and of the detection and segmentation branches. In this case, however, we compute and store the backbone feature maps for each input image during data acquisition to save time. This can be done because, during the fine-tuning, the weights of the backbone remain unaltered.

Both **Ours** and **Mask R-CNN (store features)** can perform the feature extraction while receiving the stream of images: this allows to further reduce the training time. This is possible because the frame rate for feature extraction in both cases is greater than the frame rate of the stream of incoming data. For instance, with **Ours**, we extract features at 14.7 FPS for YCB-Video while the stream of images that is used for training has a frame rate of 3 FPS

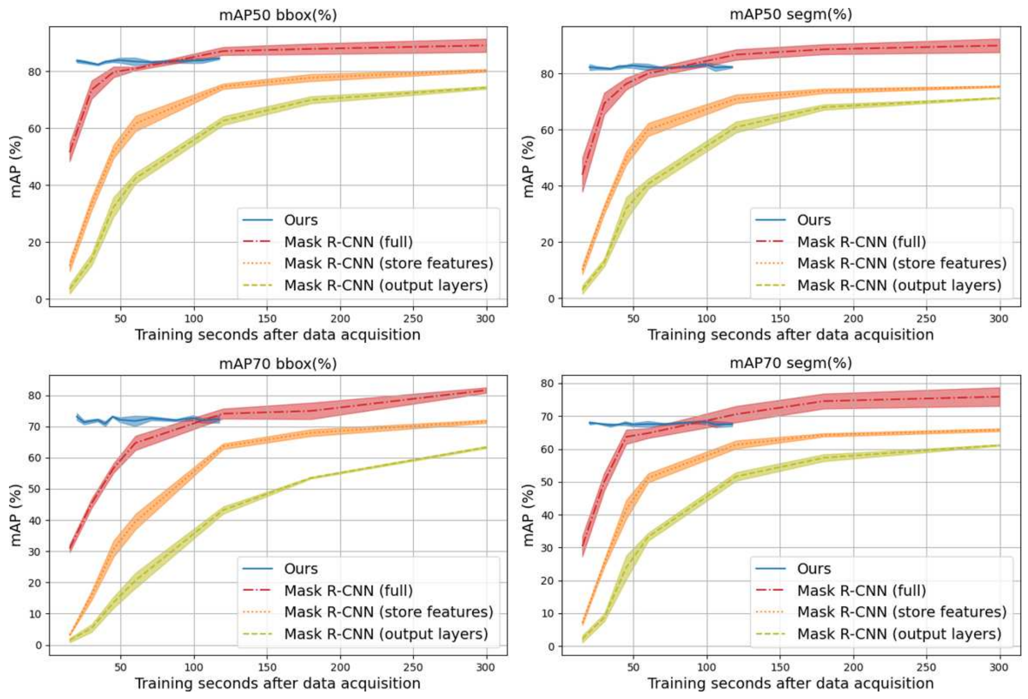


Figure 6.6 Detection and segmentation *mAPs* for increasing number of minibootstrap iterations for **Ours** and for increasing training time of the Mask R-CNN baselines, considering YCB-Video as TARGET-TASK. The plots show the average and the standard deviation of the accuracy obtained over three runs.

(note that the dataset has been collected at 30 FPS, but we use one image over ten to avoid data redundancy). This allows to completely absorb the time for feature extraction in the time for data acquisition for both approaches. Since the time required for the data acquisition is the same for the two compared methods, we remove it from the training time computation, therefore comparing only the processing time that follows this phase. This represents the time to wait for a model to be ready in the target robotic application. As explained above, the time required for feature extraction cannot be removed in the case of **Mask R-CNN (full)** and **Mask R-CNN (output layers)**.

In Figures 6.6 and 6.7 we plot accuracy as a function of training time on the YCB-Video and HO-3D datasets. Note that for **Ours** we increase the number of minibootstrap iterations to increase total training time, while for methods based on Mask R-CNN we can simply train for more epochs. At extremely short training times, **Ours** achieves the best accuracy. For instance in the YCB-Video experiment, considering a training time of ≈ 20 s, (obtained by setting the minimum number of minibootstrap iterations to 2), **Ours** achieves a **mAP** for instance segmentation of *(i)* ≈ 82.2 and *(ii)* ≈ 67.9 for the IoU thresholds set to *(i)* 50% and *(ii)* 70%. With a similar optimization time, **Mask R-CNN (full)** (the best among baselines) reaches a **mAP** of *(i)* ≈ 53.9 and *(ii)* ≈ 39.8 for the IoU thresholds set to *(i)* 50% and *(ii)* 70%.

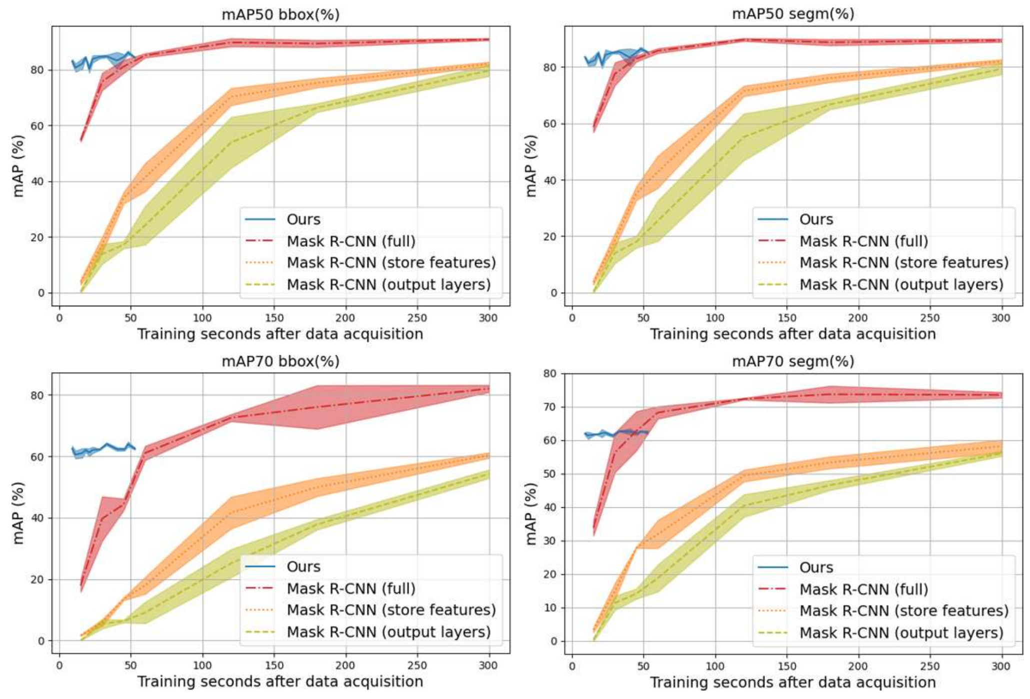


Figure 6.7 We consider HO-3D as TARGET-TASK and we report the average and the standard deviation of the *mAPs* over three training sessions with the same parameters for increasing number of *Minibootstrap* iterations for **Ours**, and for increasing training time of **Mask R-CNN (full)**, **Mask R-CNN (output layers)** and **Mask R-CNN (store features)**.

Moreover, the plots show that, for all the experiments, **Mask R-CNN (output layers)** achieves the worst performance, while **Mask R-CNN (store features)** has a steeper slope, since it has precomputed all features. On the contrary, **Mask R-CNN (full)** presents a better trend than **Mask R-CNN (output layers)** and **Mask R-CNN (store features)**. This might be due to the following reasons. Firstly, **Mask R-CNN (full)** optimizes more parameters of the network. While requiring more time for each training step, this allows to speed-up the optimization process, requiring less iterations on the dataset to achieve comparable accuracy. Secondly, **Mask R-CNN (full)** performs a warm restart of the the output layers of the RPN, while in the other baselines they are re-initialized from scratch. However, to achieve a similar performance to **Ours**, **Mask R-CNN (full)** requires $\approx 75\,\mathrm{s}$ for the YCB-Video experiment and $\approx 50\,\mathrm{s}$ on HO-3D.

Finally, as it can be noticed, the standard deviations of most of the Mask R-CNN baselines are greater than the ones of **Ours**. This derives from the fact that while **Ours** samples features from all the training images, the Mask R-CNN baselines are optimized only on a subset of them due to time constraints (*e.g.* in the YCB-Video experiment **Mask R-CNN (full)** processes images at ≈ 8 FPS when trained for 1 min). Reducing the number of training images increases the variability of the results.

6.8 Robotics Application

In this section, we describe the pipeline based on the proposed method that we developed for the iCub (Metta, Natale, et al., 2010) robot. We set our application in a teacher-learner scenario, in which the robot learns to segment novel objects shown by a human. The proposed application fits in a similar setting to HO-3D showing the effectiveness of the approach to learn new objects also in presence of domain shift.

While in the off-line experiments all input images and object instances are fixed beforehand, in real-world settings this information is not known in advance. New objects may appear in the scene and, while learning to segment them, the robot has to keep and integrate the knowledge of the known classes. We propose a strategy to process incoming images and extract corresponding features such that, for each new class, **Ours** detection model is trained, integrating the knowledge of old and new objects. This is done by first training new classifiers on the new classes, considering also the information from the objects already known. Then, the classifiers previously trained on the old classes are updated using features of the new classes.

The proposed pipeline consists of four main modules (the blocks depicted in Figure 6.8). Updating an instance segmentation model works by *(i)* automatically collecting ground-truth for instance segmentation with an interactive pipeline for incoming training images, *(ii)* extracting corresponding features and aggregating them such that the information of old and new objects are integrated in the *Minibootstrap* and *(iii)* updating the *On-line RPN*, the *On-line Detection*

Module and the *On-line Segmentation Module*. In the next paragraphs, we provide further details for each of the main blocks.

Human-Robot Interaction (HRI) This block allows the human to give commands to the robot with a module for speech recognition (*Speech Recognition* in Figure 6.8), triggering different states of the system. This allows the user to either teach the robot a new object, by presenting and rotating it in front of the camera (*train*) or to perform *inference*, *i.e.* to segment objects already known in the scene.

Automatic Data Acquisition When the state of the system is set to *train*, this block extracts a blob of pixels representing the closest object to the robot (Pasquale et al., 2016). This is used as ground-truth annotation for the new object that is presented by the human. This blob is computed by exploiting the depth information to segment the object from the background (*Automatic GT Extractor*). Moreover, in order to enhance the background variability in the training images, the extracted blob is also used by the robot to follow the object with the gaze (*Gaze Controller*). To deal with noise in the depth image, we post-process the masks to ensure spatiotemporal coherence between consecutive frames. Specifically, we consider as valid ground-truth masks those overlapping over a certain threshold with the ones of previous and subsequent frames.

Feature Extraction Relying on the ground-truth masks provided by the *Automatic GT Extractor* and on the corresponding images collected by the robot, this block implements the *Feature Extraction Module* as described in Section 6.3, with some modifications for the interactive setting. We describe the major differences in Section 6.8.1.

On-line Segmentation. This block is trained with the proposed approach **Ours** (see Section 6.3.4) relying on the features extracted by the *Feature Extraction* block. At inference time, it predicts object masks on a given image. To do this, similarly to **Ours**, it relies on Mask R-CNN pretrained on the MS COCO dataset for feature extraction and on the proposed on-line modules as described in Section 6.3.

6.8.1 Incremental Instance Segmentation Learning

When a new object has to be learned, the three on-line modules need to be updated. For the *On-line RPN* and the *On-line Detection Module*, specific operations are required to integrate knowledge of the old classes with the new one and to re-train the modules. For the *On-line Segmentation Module* only the classifier of the novel class needs to be trained.

Consider a single incremental task scenario, where the robot has been trained on $N-1$ classes and must learn to segment N-th object. To train the new classifiers for the *On-line RPN*

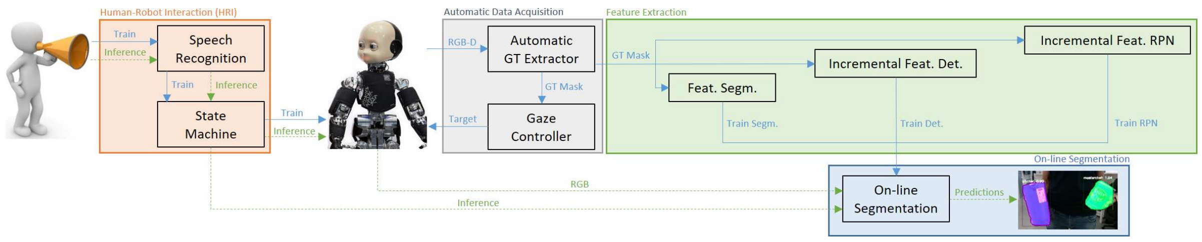


Figure 6.8 Overview of the proposed robotic pipeline for on-line instance segmentation. At training time (solid arrows), a human teacher shows a new object to the robot, which automatically acquires the ground-truth annotations exploiting the depth information. Then, it extracts the features to train the on-line modules. At inference time (dashed arrows), the robot employs such modules to predict the masks of the images acquired by the camera.

and *On-line Detection* modules, we must have access to both positive and negative samples of the new class. While positive samples can be taken from the current image stream, and are not affected by previous objects, obtaining the features for negative samples is more complex. Furthermore, the classifiers from the old classes should learn to distinguish the new object as not belonging to their class. To this end we design two procedures described in the following paragraphs.

On-line RPN When learning the N-th class, we first collect features for *On-line RPN* training for that class by extracting convolutional features from the images of the associated sequence and subsampling them as described in Section 6.3.1. Then we for every class we resample negative features taking into account the whole dataset, including the newly added class. This ensures that the number of negative samples coming from each image is kept balanced.

On-line Detection. To integrate old features with those from the new class, we create a N-th dataset to train a classifier for the new object, which contains the extracted features for the new class (positive examples), and a subset of features from the previous classes. The $N-1$ datasets for training the other classifiers are updated with negative features coming from the new object.

6.8.2 Discussion and Qualitative Results

We designed the incremental feature extraction procedures for the *On-line RPN* and the *On-line Detection Module* to be analogous to the ones used in the off-line experiments (batch procedures), such that, training the on-line modules with *Minibootstrap* batches obtained with the former, provides comparable models to the ones obtained with the batch procedures (and therefore, comparable accuracy). This is due to the fact that a set of negative samples has the same probability to end up in the *Minibootstrap* batches of a classifier (either of the *On-line*

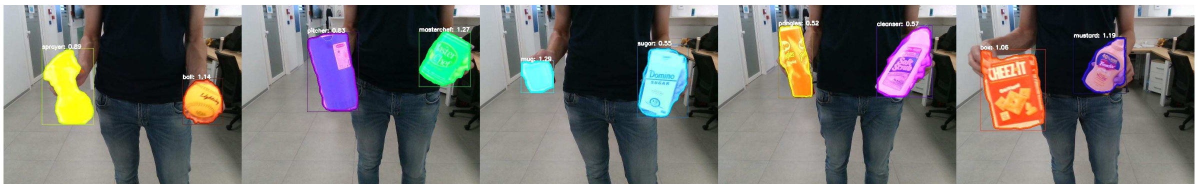


Figure 6.9 Predictions on test images from the incremental application deployed on the iCub.

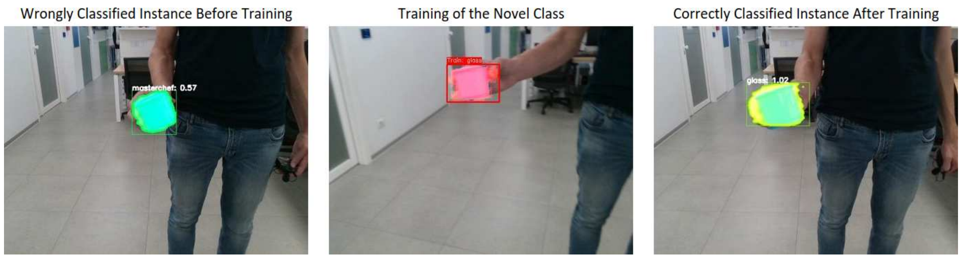

Wrongly Classified Instance Before Training
masterchef: 0.57
Training of the Novel Class
Correctly Classified Instance After Training
glass: 1.02

Figure 6.10 Dealing with false positives. *Left image*: an unknown object (a *glass*) is misclassified (as a *masterchef*). *Center*: training. The robot is provided with the correct label and a demonstration of the object. *Right*: after training the new object is correctly classified.

RPN or of the *On-line Detection Module*), using either of the sample selection pipelines. In particular, it can be shown that the per-image negative selection probabilities with the two procedures are equivalent, because each image is considered independently in both cases. This proves their equivalence.

We qualitatively show the effectiveness of the incremental pipeline by deploying it on the iCub robot. We train ten object instances and report the results of the inference on some exemplar frames in Figure 6.9. In Figure 6.10, we show how the proposed incremental approach allows to deal with false positive predictions. Key to achieve this is the re-training of the $N-1$ classifiers for previous classes when the $N-$th object arrives. Indeed, integrating data from the N-th object when updating the previous $N-1$ allows to strongly reduce the amount of false predictions at inference time.

6.9 Conclusions

The ability of rapidly adapting their visual system to novel tasks is an important requirement for robots operating in dynamic environments. While state-of-the art approaches for visual tasks mainly focus on boosting performance, a relatively small amount of methods are designed to reduce training time. In this perspective, we presented a novel pipeline for fast training of the instance segmentation task. The proposed approach allows to quickly learn to segment novel objects also in presence of domain shifts. We designed a two-stage hybrid pipeline to operate in the typical robotic scenario where streams of data are acquired by the camera of the robot. Indeed, our pipeline allows to shorten the total training time by extracting a set of convolutional features during the data acquisition and to use them in a second step to rapidly train a set of Kernel-based classifiers.

We benchmarked our results on two robotics datasets, namely YCB-Video and HO-3D. On these datasets, we provided an extensive empirical evaluation of the proposed approach to evaluate different training time/accuracy trade-offs, comparing results against previous work (Ceola, Maiettini, Pasquale, Rosasco, et al., 2021) and several Mask R-CNN baselines.

Finally, we demonstrated the application of this work on a real humanoid robot. At this aim, we adapted the fast training pipeline for incremental region proposal adaptation and instance segmentation, showing that the robot is able to learn new objects following a short interactive training session with a human teacher.

Chapter 7

Kernel methods for Wind Prediction

In this second chapter concerning applications of kernel methods, we tackle a challenging problem from the natural sciences: wind speed forecasting. In contrast to the common practice of using mechanistic models (*i.e.* climate simulations) to make forecasts, we adopt a purely data-driven approach based on kernel ridge regression. Starting from a large dataset of wind speed readings, measured from anemometers placed in 32 locations in the Abruzzo and Liguria regions of Italy (central and north west areas of the country), we train supervised models which learn historical wind patterns to predict its future trends. By running many variations on this supervised learning template, changing inputs and outputs, algorithms and datasets, our work focuses on the analysis and interpretation of the data and of the underlying physical processes.

7.1 Introduction

The global consumption of energy produced from wind raised from about 87 TW h annually in the early 2000, to over 3500 TW h per year in 2019. This relative growth rate of about +4000 %, as well as that from solar (about +60 000 %), are order magnitudes larger than that from oil (approximately +25 %), nuclear (−3 %) and other more traditional renewable energy sources like hydroelectric power (approximately +42 %) (Ritchie et al., 2022; BP p.l.c., 2022; Smil, 2017). Thus wind energy will play an increasingly important role in the global economy in the near future. The ability to predict wind is essential for maintenance of wind power plants as well as for energy markets relying on predictions of the energy power produced from wind. Moreover, reliable predictions of wind speed are valuable for private citizens as well as for public administrations concerned with safety in the case of hazard scenarios.

Atmospheric forecasting and weather predictions have traditionally relied on numerical simulations of model equations based on physics. However, these mechanistic models, that form the basis of traditional forecasting, have poor performance close to the ground. Winds near the surface are affected by several processes that occur at spatial and temporal scales that are below the resolution of the numerical simulations. To account for these unresolved mechanisms, alternative approaches use machine learning and predict wind speed close to the ground from time series of measured data. For a complete survey on time series techniques, independent of a particular application, see for example Parmezan et al. (2019).

Following this data-driven approach, several works have been carried out with a growing trend in the use of Deep Learning tools. Among deep architectures, Long-Short Term Memory (LSTM) Neural Networks (Lindemann et al., 2021) and its variants have received increasing attention due to their particular suitability to deal with sequential data like time series. In the context of wind speed prediction a large number of specific strategies have been developed. Many efforts are directed at designing methods to capture the multi-scale nature of atmospheric dynamics, where many decades of spatial scales are dynamically coupled in a highly nonlinear process. The pipeline of these algorithms may combine a multi-scale feature extraction stage with a following regression algorithm. Feature extraction may be accomplished through Wavelet Transforms (F. Li, G. Ren, et al., 2019; Yousuf et al., 2021), Singular Spectrum Analysis (Fu et al., 2020; H. Liu, Mi, et al., 2019), Empirical Mode Decomposition (Fu et al., 2020; Ruiz-Aguilar et al., 2021), CNNs (Lawal et al., 2021). Other authors attempt to embed sensitivity to multi-scale dynamics directly into the architecture of a neural network (Araya et al., 2020). Besides Deep Learning architectures, different algorithms have been developed based on Machine Learning models (for example kernel methods, Support Vector Regression (Mattos Neto et al., 2020) and Gaussian Processes (C. Zhang, Wei, et al., 2016)) as well as classical statistical models like ARIMA and SARIMA (Bivona et al., 2011; Yousuf et al., 2021) and stochastic processes (Bivona et al., 2011; Carpinone et al., 2015). Furthermore several hybrid models that combine techniques from different families have been considered (Mattos Neto et al., 2020; Camelo et al., 2018; Yousuf et al., 2021). Measures from anemometers have been also combined with LIDAR data for wind forecasting at different altitudes (Mohandes et al., 2021a; Mohandes et al., 2021b).

Forecasting methods for wind generally need to address its non-stationarity, *i.e.* that the statistical distribution of wind speed may vary in time. The rolling or moving window approach is a widely adopted solution to tackle non stationarity and it consists in updating the model by periodically retraining the algorithm eliminating obsolete data and adding fresh information given by newly available data. Although this technique proves crucial in certain applications like financial markets, there is no clear evidence in favor or against this method for wind speed forecast.

Predictive models are also classified according to their forecast horizon, ranging from Very Short term (less than 1 hour), Short term (up to about 4 hours) to Medium term (up to 24 hours ahead) but also Long term predictions (more than 1 day). This latter subdivision is somewhat arbitrary and does not immediately connect to a notion of predictability, which may be better captured by other physical time scales (*e.g.* the correlation time of wind speed) that typically change considerably with location. Moreover, the definition of “long term” as longer than one day is peculiar to data driven models, whereas physics-driven models typically forecast several days ahead. Note also that a forecast may be achieved by learning one specific model for the desired horizon, or by inferring directly an array of future values at different horizons either recursively or all together (F. Li, G. Ren, et al., 2019; H. Liu, Mi, et al., 2019; Y. Li et al., 2019). Some works exploit information carried by other meteorological variables, like air pressure or temperature (Trebing et al., 2020), or include spatial correlations among observations from different geographical locations in a network (Xu et al., 2022; Zhu et al., 2018; Messner et al., 2019). Remarkably, wind direction, which is usually available together with wind speed, has been rarely exploited to design features for wind speed forecast, with few exceptions (Trebing et al., 2020; Chitsazan et al., 2017).

Our work takes a shallow approach, *i.e.* we use linear models and kernel methods which benefit from well-known theoretical guarantees, unlike more commonly considered Deep Neural Networks. Our main contribution to the literature in this field is to provide a systematic analysis of the algorithm design and a rationale to understand the optimal algorithm based on principles that are rooted in the physics of the atmosphere. This approach results in design criteria that allow to tailor algorithms to the specific location under study. More specifically, we analyze a massive experimental dataset of wind measured from anemometers placed in 9 locations close to the ground within the Abruzzo region in the central part of Italy and 23 locations in the Liguria region, in north western Italy. These two areas were chosen because of their complex orography and because of the interaction between land and sea circulations, which make wind prediction extremely challenging. On this data, supervised learning algorithms are trained using historical values of the wind to predict its future trends at different temporal horizons. We first analyze two locations and a single time horizon and compare systematically several different algorithms where we vary: the input/output variables; the past history used for training; the linear *vs* non-linear statistical model. Motivated by these results, the analysis is extended across all locations and all forecasting horizons. We find that the optimal design as well as its performance can vary considerably with location; for example, the inclusion of wind direction improves performances in about half of the locations. Furthermore accounting for non-stationarity with a rolling window approach does not improve performance. We demonstrate that where and when the diurnal cycle is robust, the input data should include at least 24h of past history, to take advantage of the regularity of the pattern. This simple design principle is valid for all intermediate forecast horizons, that are most affected by the daily periodicity. Although the optimal algorithms vary with location, we identify a single model that preserves good performances across all datasets. By introducing a measure of performance relative to a widely used standard in atmospheric modeling, we demonstrate that this algorithm is competitive with more complex state-of-the-art algorithms. We further corroborate the result by applying our algorithm to datasets used in state-of-the-art literature and comparing the exact same diagnostic. To further improve results, future studies could build upon this approach by leveraging spatial correlations. Indeed, our predictions are based on data recorded at a single geographical location, thus exploiting temporal information but missing spatial information. To this end, one could either include data from different anemometers as input, or integrate atmospheric modeling which naturally couples different locations through the physics of the atmosphere.

In section 7.2 our data-driven approach to wind speed forecasting is described together with the machine learning algorithms and the datasets used. In Section 7.3 the main results of our experiments are shown. In Section 7.4 the effects of accounting for non-stationarity with a rolling-window approach are discussed. In Section 7.5 the proposed data driven approach is compared to other methods used within the context of wind speed forecasting. In Section 7.6 final remarks and observations that follow from the experimental evidence are discussed.

7.2 Data driven models for wind forecast

In this section the problem of wind speed forecasting is described, along with the data-driven approach used to derive algorithmic solutions. Each time series contains data of wind speed and direction recorded hourly from the start of 2015 to the end of 2019 (more details about the datasets can be found in Section 7.2.3). In particular, for each time t there are three available data: the speed of the wind s_t, its meridional component m_t and its zonal component z_t, which are related through the formula $s_t = \sqrt{m_t^2 + z_t^2}$. This decomposition provides information on wind strength both in the direction parallel to the lines of latitude (zonal component) and in the direction parallel to the meridians (meridional component), thus allowing a more fine-grained characterization of wind patterns. The two components are obtained from measurements of wind direction θ_t and speed s_t as $m_t = s_t \cos\left(\frac{2\pi}{360}\theta_t\right)$ and $z_t = s_t \sin\left(\frac{2\pi}{360}\theta_t\right)$, where the angle θ_t is zero along the North-South direction, grows clockwise and is measured in degrees. Our goal is to use this data to learn a model that predicts the wind speed s_{t+h} at a future time $t+h$, where h defines the forecast *horizon*.

We consider machine learning models, which infer the relation between the future value of the wind speed at time $t+h$ from the past μ measurements, where μ is called *memory*, *i.e.*

$$\widehat{s}_{t+h} = \hat{f}(\eta_{t-\mu+1}, \dots, \eta_t) \tag{7.1}$$

where $\hat{f}$ denotes a learned model, $\widehat{s}_{t+h}$ our prediction at horizon h and the input η_t can be either the sole speed s_t or the pair (z_t, m_t). Figure 7.1 gives a pictorial representation of the wind speed prediction task.

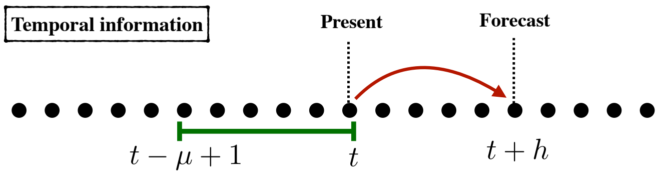


Figure 7.1 For each time t in the time series an input sample is built from the past μ timesteps. The associated output is the value of wind speed measured at $t+h$.

Different combinations of horizon h, memory μ, and input data η_t are studied, in order to understand how they affect the overall prediction performance. In particular hourly horizons $h \in \{1,3,6,12,18,24\}$ and memories up to 3 days in the past $\mu \in \{2,6,24,48,72\}$ are considered. For a fixed horizon h and memory μ the following options for designing the inputs and outputs are taken into account (summarized in Figure 7.2).

`s` $\to$ `s` where both input and output are the wind speed:

$$\widehat{s}_{t+h} = \hat{f}(s_{t-\mu+1}, \dots, s_t)\,; \tag{7.2}$$

`zm` $\to$ `s` where the input is divided in zonal and meridional components and the output is wind speed

$$\widehat{s}_{t+h} = \hat{f}((z_{t-\mu+1}, m_{t-\mu+1}), \dots, (z_t, m_t))\,; \tag{7.3}$$

`zm` $\to$ `zm` where the input is divided in zonal and meridional components, each component of the wind vector is learned separately and the wind speed is computed from the components

$$\widehat{z}_{t+h} = \hat{f}_1((z_{t-\mu+1}, m_{t-\mu+1}), \dots, (z_t, m_t)) \tag{7.4}$$

$$\widehat{m}_{t+h} = \hat{f}_2((z_{t-\mu+1}, m_{t-\mu+1}), \dots, (z_t, m_t)) \tag{7.5}$$

$$\widehat{s}_{t+h} = \sqrt{\widehat{z}^2_{t+h} + \widehat{m}^2_{t+h}}\,. \tag{7.6}$$

More details about the computational procedures to derive the learning model $\hat{f}$ can be found in Section 7.2.1.

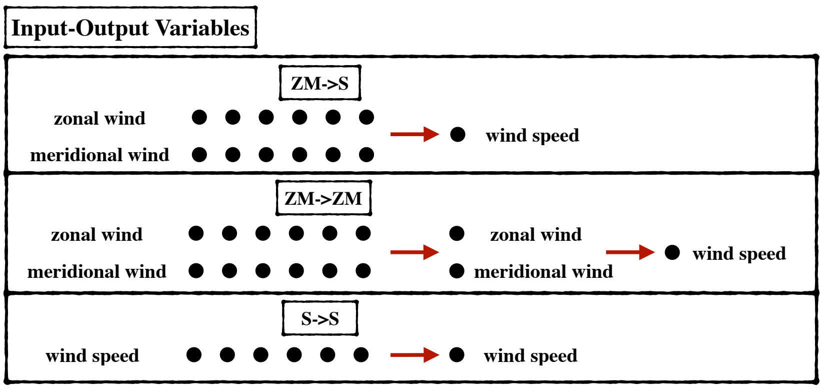


Figure 7.2 For each time t in the time series an input sample is constructed either including both wind components ($\texttt{zm} \to \texttt{s}$ and $\texttt{zm} \to \texttt{zm}$) or considering only the wind speed ($\texttt{s} \to \texttt{s}$). For $\texttt{zm} \to \texttt{s}$ and $\texttt{s} \to \texttt{s}$, the associated output is the value of wind speed measured at $t + h$, where h is the horizon. For $\texttt{zm} \to \texttt{zm}$ the two wind components at time $t + h$ are learned separately and from them the wind speed is reconstructed.

In order to measure the predictive performance of our models, the data from each location is split at a fixed date (January 1st, 2018). Then all data before this date (the training set) is used to train model $\hat{f}$, and the remaining data (test set) is used to test predictive performance of the model. Note that data may be missing at specific dates for technical issues with the anemometers, and the missing data depend on location. The splitting criterion used leads to large variations in the training set size between stations; however it allows for a better comparison among the different sites by making the test sets uniform in size for each region. Throughout the paper, a *static* approach to splitting is used where a model's training set is not updated in time. In Section 7.4 we motivate such choice with a case study to quantify the potential gain from updating the training set continuously with newly available samples.

Accuracy of the predictions is quantified with the normalised root mean squared error (NRMSE). For n predictions it is defined as

$$\text{NRMSE} = \sqrt{\frac{\sum_{t=1}^{n}(s_t - \hat{s}_t)^2}{\sum_{t=1}^{n} s_t^2}}. \tag{7.7}$$

7.2.1 Supervised learning

For a given horizon h, memory μ and input-output for the model $\hat{f}$, n input-output pairs $(\mathbf{x}_t, y_t)_{t=1}^{n}$ comprise our training samples. For example, for the $\texttt{zm} \to \texttt{s}$ model, pairs are defined as:

$$\mathbf{x}_t = [z_{t-\mu+1}, m_{t-\mu+1}, \dots, z_t, m_t] \in \mathbb{R}^d$$
$$y_t = s_{t+h},$$

where $d = \mu \times k$ and k is the number of variables in η_t. Denote by $X \in \mathbb{R}^{n\times d}$ the lag matrix with rows $\{\mathbf{x}_t\}_{t=1}^{n}$, and $\hat{y} \in \mathbb{R}^n$ be the vector of outputs with elements $\hat{y}_t$. In all analyses of this chapter we will be using two models, introduced in Chapters 2 and 3. The first one is linear least-squares regression which assumes that future wind behavior depends linearly on its past trends, and learns functions in the space of Equation (2.15) by minimizing

$$\frac{1}{n}\sum_{i=1}^{n}(\langle x_i, w\rangle - y_i)^2 = \frac{1}{n}\|\mathbf{X}w - \mathbf{y}\|^2, \quad w \in \mathbb{R}^d \tag{7.8}$$

with respect to vector w. As seen in Chapter 2, linear function do not allow to account for complex interactions between past and future behavior of the wind. Kernel ridge regression (KRR) introduces a non-linear transformation of the features via the kernel function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ which intuitively measures the similarity between two data-points. In our experiments we always used the Gaussian kernel (Equation (2.24)) which depends on parameter γ. In particular, we employed the Falkon algorithm (Rudi, Carratino, et al., 2017; Meanti, Carratino, Rosasco, et al., 2020) to solve an approximate version of KRR with greatly improved running time, see Chapter 3 for more details.

7.2.2 Hyperparameter selection

For each location under consideration model hyperparameters (e.g λ and γ for KRR) have been estimated using five-fold cross-validation. A two-step grid search was adopted: in the first step hyperparameters were chosen to maximize the (5-fold cross-validated) R^2 score on a coarse grid, and in the second step a new search was performed on a grid refined around the previously identified optimum. The number of Nyström centers m were set to $10 \cdot \sqrt{n}$ which provided a good trade-off in terms of accuracy versus time.

7.2.3 Datasets

All time series analysed in this work consist of observations recorded by anemometers located at an altitude of 10 m above ground level, except for two stations (A1 and A5) which are 6 m above ground level. Measurements are taken from 32 meteorological stations spread across the Liguria and Abruzzo regions of Italy, both characterized by the presence of a complex orography. These anemometers face diverse dynamical conditions, going from plains to valleys up to the peaks of high mountains (see Figure 7.3 for more details). Proximity to the sea also significantly contributes to enrich the wind dynamics. The data series span a period between 4 and 7 years, depending on the station and have a time step of 1 hour.

Most recent literature uses data coming from wind farms, where the anemometers are typically around 90 meters above ground. Note that within the atmospheric boundary layer, the vortical structures typical of turbulence (eddies) have a size that scales with distance from the ground. Therefore wind at 90 m from the ground is more predictable as it changes over longer timescales, while wind at 10 m from the ground (our datasets) is dominated by small eddies which change on a short timescale and are hard to predict.

7.3 Results

In this section the main results obtained for wind speed prediction with different models are described. We begin by analysing the behavior of our models on two representative stations, comparing different inputs, outputs and model types. The observations made on this subset of locations are then taken into account to inform a full analysis of the whole set of stations.

7.3.1 Analysis of two case studies

The stations considered here are A1 in Abruzzo and L1 in Liguria (see Figure 7.3) as representatives of the whole dataset. They are geographically distant, and diverse regarding both terrain morphology and wind statistics, with L1 being on a mountain peak at 980 m of altitude and A1 in a plain close to the sea.

The forecasting task is fixed to wind speed prediction at a 3 hour horizon, and we wish to identify how the following parameters affect predictive performance.

1. Input and output variables. Determine which input-output variables out of `s` $\to$ `s`, `zm` $\to$ `s` and `zm` $\to$ `zm` results in better predictive performance. `s` $\to$ `s` consists in predicting future wind speed from past wind speed, `zm` $\to$ `s` uses both wind speed and direction in the input through the zonal and meridional components, and `zm` $\to$ `zm` predicts both zonal and meridional components separately, to then reconstruct the wind speed itself.

2. Model class. Distinguish between the performance of *linear* models (with the linear least squares (LLS) algorithm), and *non-linear* models represented by KRR.

3. Memory μ. Investigate the effect of varying the amount of past data considered at each input point. Values between 2 h and 72 h are considered.

The analysis of predictive accuracy as measured by the NRMSE for the different parameters described above is shown in Figure 7.4, and is compared to the performance of a naïve model (the *persistence* model), whose predictions for time $t + h$ are the observed values at time t. From panel (a) a few observations can be made:

- All models outperform the persistence model, reducing the NRMSE by up to 20 % which is quite significant.

- The input design which performs best is the one which takes both wind components as inputs and predicts the wind speed directly, supporting the importance of wind direction

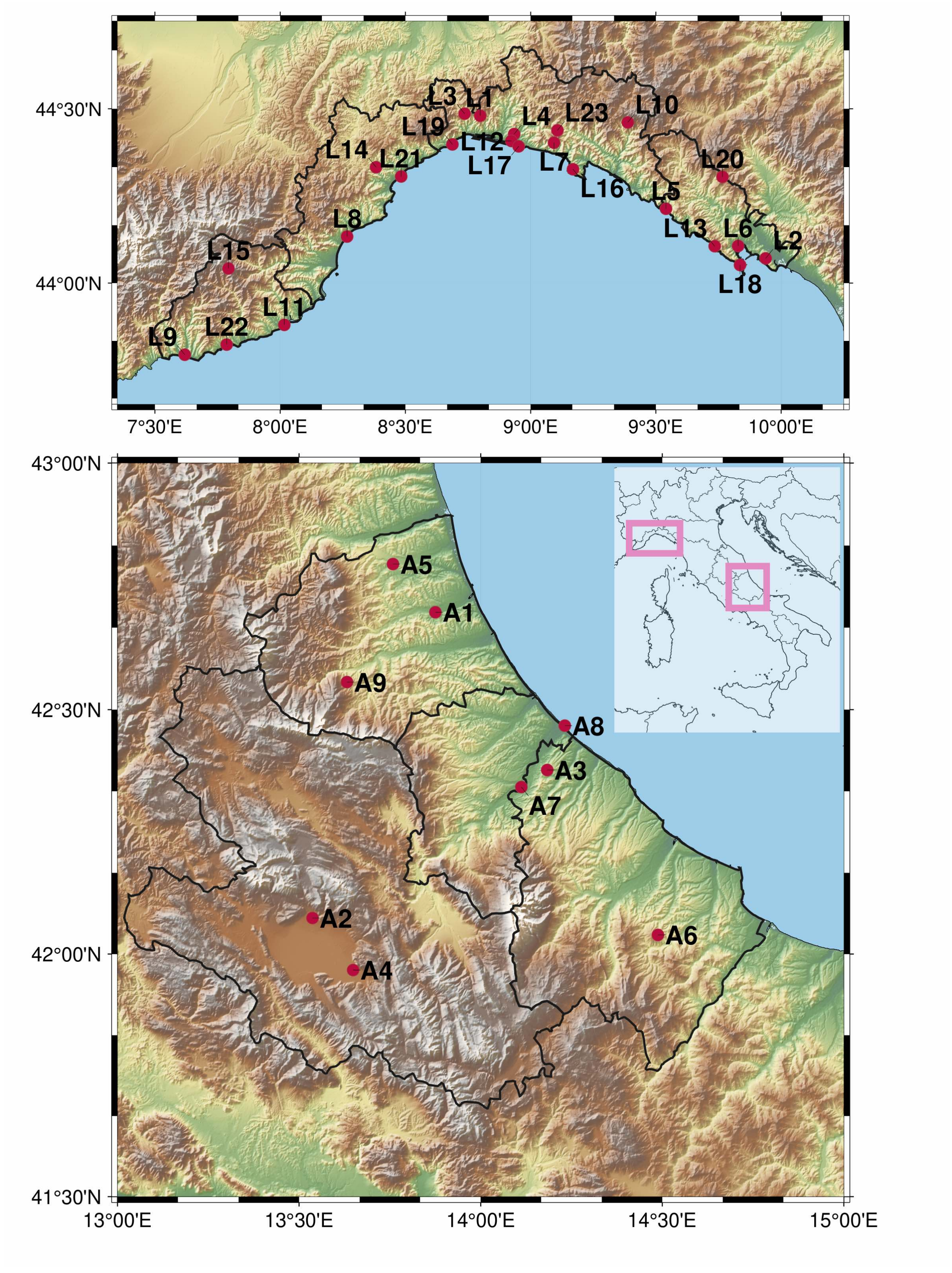


Figure 7.3 Physical location of the 32 anemometers in our dataset.

for forecasting. However, restricting the comparison to the linear models, the $\mathtt{s} \to \mathtt{s}$ input-output design performs better. Hence gains from using wind direction can only be leveraged by non-linear models, due to the non-linear dependency of wind speed from the two components of the wind vector.

- For fixed input and output variables, non-linear algorithms are systematically better than their linear counterparts.
- Overall using 24 h of memory seems to provide the best trade-off between performance and input size. Higher amounts of memory do not seem beneficial, and lower amounts of memory worsen performance noticeably.

To evaluate the robustness of these initial observations, the same comparison is provided for a second location (L1), see Figure 7.4(b). First, improvements in NRMSE over the persistence model drop dramatically to at most 4 %. Second, the use of zonal and meridional components in the input brings no benefit (even for non-linear models). Third, linear and non-linear models achieve the same performance. Fourth, increasing memory above 6 h provides no benefit. How can such discrepancies between two different stations be interpreted? Is there a physical mechanism at the origin of these differences? In order to answer such questions we extend the analysis to a larger number of stations.

7.3.2 Model design

In this section, the analysis of performance for different model types introduced in the previous section (*i.e.* taking into account input-output design, memory, model type) is expanded to the whole set of 32 stations. The forecast horizon is further added into the mix of model parameters under consideration. Short horizons are easier to predict even with simple models as their departure from the current state of the atmosphere is small. As the horizon increases, the chaotic dynamics in the atmosphere causes the wind to decorrelate from its current value more and more, thus the input bears less and less information about the output and predictions become more challenging. Overall a total of 186 model instances are tested for each location, resulting in 5952 trained models. Noting that for each training, the appropriate cross-validation must also be run to chose a model's hyperparameters (as described in Section 7.2.2), the total computational load is very high. To reach the required scale in reasonable times, we rely on the Falkon library (Rudi, Carratino, et al., 2017; Meanti, Carratino, Rosasco, et al., 2020) which implements an approximation of KRR, coupled with clever optimization algorithms, on the GPU.

The aims of this extensive survey are to investigate (a) the role played by the input-output design and the model (*i.e.* linear or non-linear), (b) the effects of memory on predictions, and their physical interpretation, (c) the possibility to identify a single model type which performs well in each geographical location.

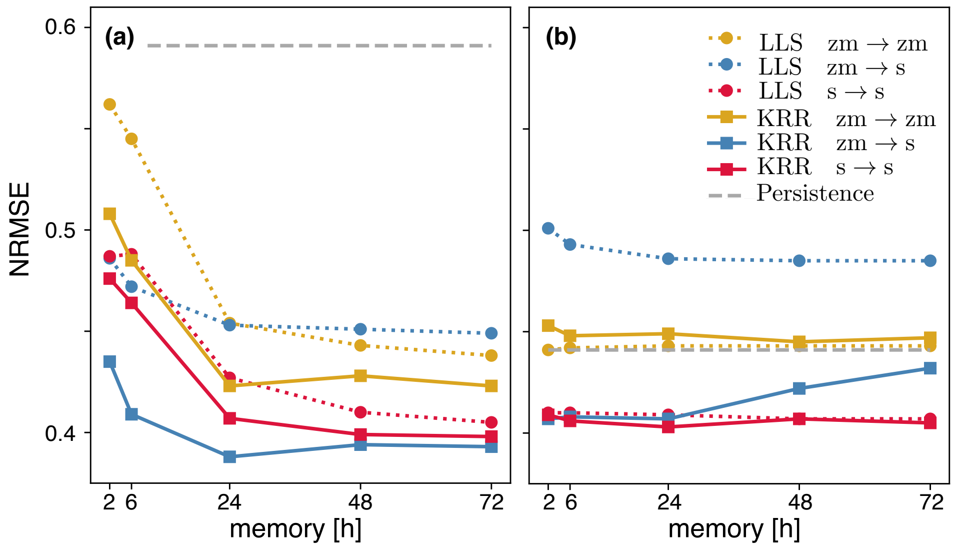


Figure 7.4 Prediction accuracy for different memories on stations A1 and L1. Panel (a) shows the forecast accuracy on station A1. All models improve considerably over the persistence (dashed grey line). The best among non-linear models (solid lines, indicated with KRR) is the `zm` → `s` model (solid blue line) with 24 h memory, which improves on the best linear model (dashed lines, indicated with LLS). Panel (b) displays results on station L1. In this other location the improvement over persistence is lower, and the best linear model is on par with the best non-linear one. All predictions were performed on a 3 hour horizon.

Input-Output Design

As can be seen in Figure 7.5 (b), the first observation of the preliminary analysis can be confirmed: directly predicting the wind leads to much better performance, especially for long-term predictions.

On the input side, it can be observed in Figure 7.5 (a) that the benefits of including the direction in the input depend strongly on the location (note that this experiment used the KRR model). This result is consistent with the results of Section 7.3.1, where two different stations had two different behaviors. In conclusion, the influence of wind direction on its speed is complex, and not always helpful for improving predictions.

Finally, panel (c) shows that nonlinear models remain a better solution, with potentially moderate gains (*e.g.* in the case of 6 hours ahead predictions), but virtually no downside. Another observation is that the performance improvement of KRR over LLS is smallest for horizons of 1 and 24 hours. This can be qualitatively understood by considering the predictability of wind at different forecast horizons. A horizon of 1 hour is always within the correlation time of wind speed, hence for this horizon the time series can be described well by a linear autoregressive process. For 24 hours horizon, the effect of the diurnal cycle (easy to predict)

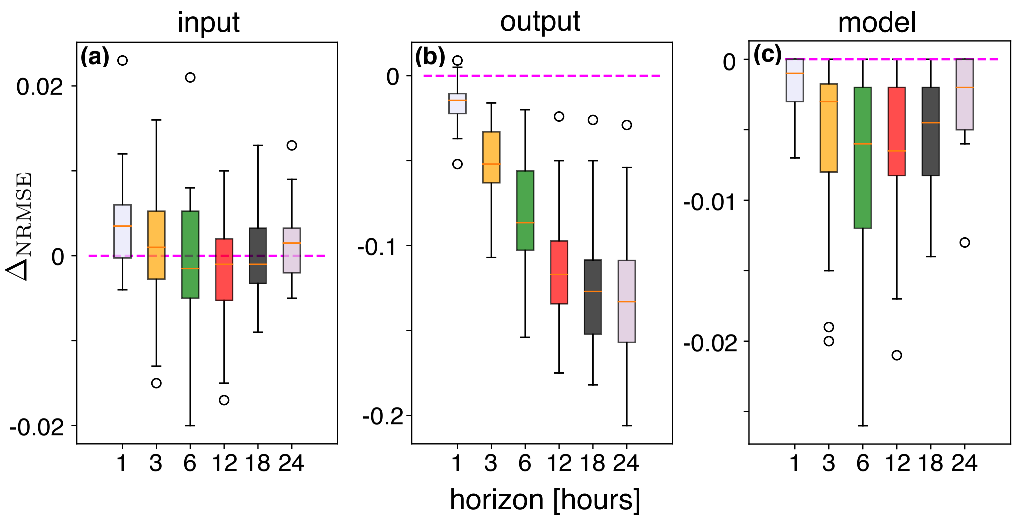


Figure 7.5 Effect of different design choices on accuracy. (a) Variation in the NRMSE ($\Delta_{\text{NRMSE}} = \text{NRMSE}(\texttt{zm} \to \texttt{s}) - \text{NRMSE}(\texttt{s} \to \texttt{s})$) when using both wind components *vs* only wind speed as input variables (in both cases the output is wind speed). (b) Analogous variation in NRMSE ($\Delta_{\text{NRMSE}} = \text{NRMSE}(\texttt{zm} \to \texttt{s}) - \text{NRMSE}(\texttt{zm} \to \texttt{zm})$) when predicting wind speed *vs* predicting separately the two wind components and then reconstructing the speed (in both cases the input includes both components of the wind). (c) Variation in NRMSE ($\Delta_{\text{NRMSE}} = \text{NRMSE}(\text{nonlinear}) - \text{NRMSE}(\text{linear})$) between the locally best model and the best linear one, showing that linear models are competitive with non-linear models when focusing on short (1 hour) and very long (24 hour) horizons. For all experiments the best choice of memory μ for each station was selected. Forecast horizons are indicated by colours and reported on the x axis.

becomes strong, and any variations on top of it are very hard to predict due to the long time scales. Therefore a more complex model has fewer advantages over a simpler one.

The Role of Memory

The next question is to find the optimal amount of past memory to infer the future and how this depends on the forecast horizon. In the preliminary analysis (Figure 7.4) it was observed that $\mu = 24\,\text{h}$ achieved the best performance whereas longer memory would increase the size of the input data without providing benefit for performance. The following experiment is designed to verify whether this result extends to other sites. For each location and horizon, the configuration (in terms of input-output, model type and amount of memory) with the lowest NRMSE is identified. This configuration is referred to as the *locally best model.* Next, the effect of memory on algorithm design is tested. To this end, the input-output variables as well as the model type are kept fixed and equal to those identified by the locally best model. Meanwhile, 5 different memories are compared $(2, 6, 24, 48, 72)$. The difference between the NRMSE with

$\mu = 2$ (short memory), and with other values of μ is plotted in Figure 7.6 for every station and horizon. Each line starts at zero (since $\Delta_{\text{NRMSE}} = \text{NRMSE}(\mu\,\text{h}) - \text{NRMSE}(2\,\text{h})$, and at the first point $\mu = 2$). For longer memories, it either decreases if longer memory is beneficial, or it increases if longer memory is detrimental.

We find that memory affects our models' accuracy in a way that depends on the horizon. For short-term predictions (1 hour, Figure 7.6 (a)), about 50 % of the stations are better predicted using a memory of 2 hours, rather than 24 hours. For most stations (78 %), the optimal memory increases when the horizon is set to $h = 3\,\text{h}$. The number of stations which benefit from longer memory further increases when the horizon is set to 6, 12 and 18 hours and for such medium term scenarios, the optimal memory is 24 hours for most stations (between 88 % and 97 %). When the horizon is set to $h = 24\,\text{h}$, it can be observed that even though most stations (84 %) benefit from a longer memory, the improvement is marginal (an average decrease of 0.002 in NRMSE). Whenever longer memory is beneficial, $\mu = 24\,\text{h}$ is a knee-point, *i.e.* there is a considerable gain in switching from $\mu = 6$ hours to $\mu = 24$ hours, but further increasing memory to $\mu > 24$ hours gives only small improvements, at a substantial computational expense.

We hypothesize that the role of memory laid out above can be traced back to the diurnal cycle in the atmosphere. In a nutshell, the diurnal cycle represents that many environmental quantities in the atmosphere undergo oscillations with a period of 24 hours, caused by periodicity of the sunlight. In the presence of a reliable diurnal cycle, the wind at time t may be well-predicted by the wind at time $t - 24$. At very short horizons however, the wind changes little, thus better predictions may be achieved based on persistence, rather than by exploiting the diurnal cycle. In this case, a memory of 1 hour is optimal because a model which only takes the most recent data as input outperforms a model where the most informative data are combined with less informative data at previous times. At medium term, forecasts become more challenging as persistence is a poor predictor of wind. At these horizons, it is beneficial to include the full 24 hour cycle preceding the target time $t + h$, so that the prediction can benefit from the regularity of the wind. Finally, predictions at $h = 24\,\text{h}$ are even more challenging, and all models incur in significant errors. In this case, the most recent data (at time t) is exactly 24 hours before the target and is expected to be well correlated with wind at the target time. Moreover, there is a full diurnal cycle between the current time and the target time, thus including data prior to t does not provide information about the most recent diurnal cycle, but about the preceding one. This is expected to be less informative, hence the marginal improvement in Δ_{NRMSE}.

Analysis of the Wind's Diurnal Cycle

To test the hypothesis that the diurnal cycle is at the origin of longer optimal memories for intermediate horizons, an additional analysis is carried out to check the correlation between benefits from using 24 h memory, and diurnal cycle strength. The strength of the diurnal

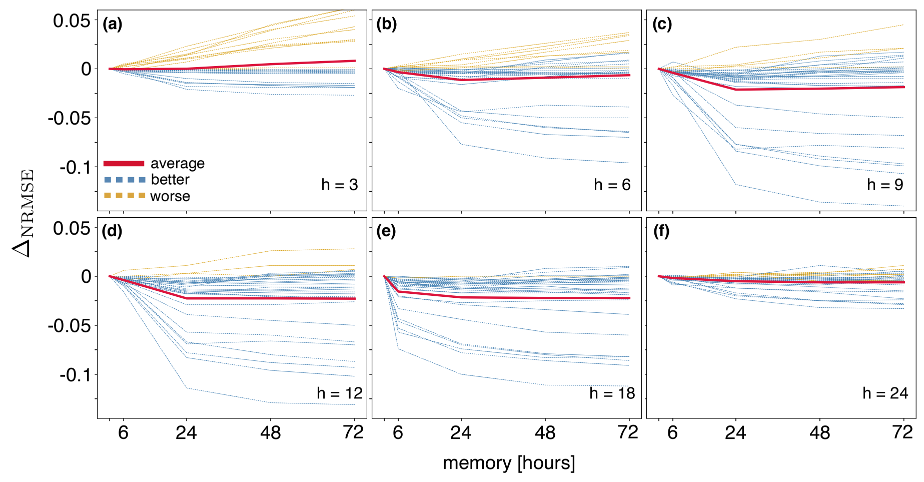


Figure 7.6 Benefits of memory. (a)-(f) Variation in performance $\Delta_{\text{NRMSE}} = \text{NRMSE}(\mu) - \text{NRMSE}(2\,\text{h})$ between the locally best algorithm with memory μ and the same algorithm with reduced memory $\mu = 2$ hours for different time horizons ((a) to (f) correspond to horizons $h = 1$, 3, 6, 8, 12, 24 hours). Each dashed line represents one location; blue and yellow mark locations where a memory of 24 hours improves or deteriorates performance respectively. Red solid lines represent averages over all locations. At intermediate horizons (3 to 18 hours, panels (b)-(e))), an input memory $\mu = 24$ hours is beneficial and longer memories lead to minor improvement.

cycle can be measured by computing the autocorrelation of the wind time series at 24 hours ($R_{ss}(24\,\text{h})$). The gain $\Delta_{\text{NRMSE}}(24\text{ h})$ is quantified by comparing models trained with $\mu = 24\,\text{h}$ and with $\mu = 2\,\text{h}$. Figure 7.7 includes all stations and forecast horizons and confirms the hypothesized relationship between the strength of the diurnal cycle $R_{ss}(24\,\text{h})$ and the benefit of using a 24 h memory in the input. First, at intermediate forecasting times (colored stars), the major benefits of a 24 hour memory are clearly achieved when the diurnal cycle is stronger. Second, this clear trend vanishes at very short and long horizons ($h = 1$ hour and $h = 24$ hours, grey triangles).

Selection of a single model

In the previous analyses, the locally best model was used, which changes for each station. However, it is desirable in practice to select a single model that may perform well over all stations so that it may be used as a default in the absence of better information. For each horizon, we call the model configuration which most frequently performs best the *globally best model.* In Figure 7.8 it can be seen that the globally best model loses little accuracy over the locally best model (a few percentage points of NRMSE). The globally best model features: $\mu = 2\,\text{h}$ for 1-hour-ahead predictions, $\mu = 24\,\text{h}$ for 3, 6, 12 and 24 hour ahead predictions and

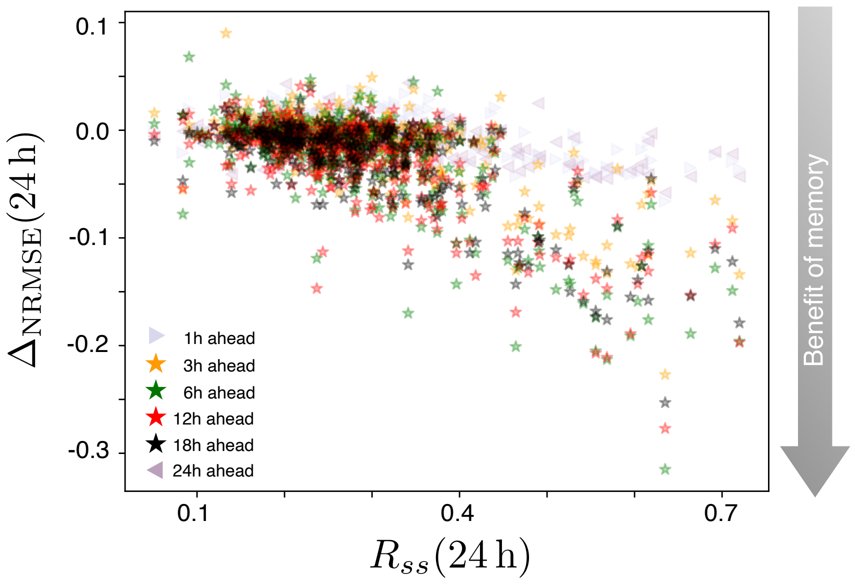


Figure 7.7 At intermediate horizons, a 24-hour memory is beneficial in the presence of a strong diurnal cycle. Benefits of 24-hour memory are defined as $\Delta_{\mathrm{NRMSE}} = \mathrm{NRMSE}(24\,\mathrm{h}) - \mathrm{NRMSE}(2\,\mathrm{h})$, *i.e.* the difference in performance between the locally best algorithm with memory 24 h and the same algorithm with reduced memory $\mu = 2$ hours. The strength of the diurnal cycle is defined as the normalized autocorrelation of the wind speed $s(t)$ at a lag of 24 hours: $R_{ss}(24\,\mathrm{h}) = \langle (s(t) - \bar{s})(s(t+24\mathrm{h}) - \bar{s}) \rangle / \sigma_s^2$, where $\bar{s}$ and σ_s are the mean value and the standard deviation of $s(t)$ respectively, computed over non-overlapping five-day windows. Each symbol represents data averaged over one month for a single location and forecast horizon (colored stars: intermediate horizons; grey triangles short and long horizons).

$\mu = 72\,\mathrm{h}$ for 18 hour ahead predictions. The globally best input-output design is $\mathtt{zm} \to \mathtt{s}$ for all horizons except for the 18 hour horizon, where the global best is $\mathtt{s} \to \mathtt{s}$.

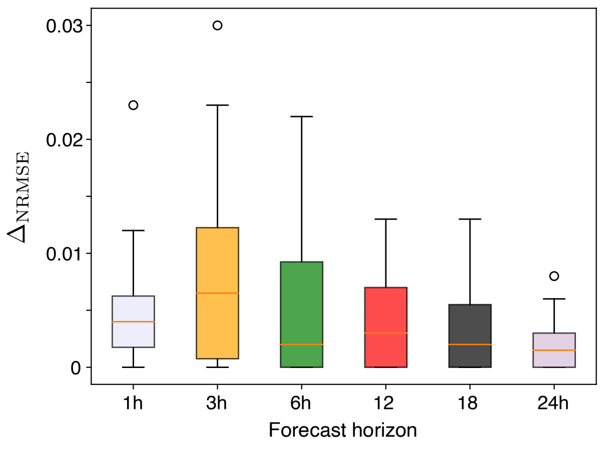


Figure 7.8 Variation of performance between locally and globally best models. Difference in NRMSE ($\Delta_{\mathrm{NRMSE}} = \mathrm{NRMSE(globally)} - \mathrm{NRMSE(locally)}$) between globally best models (with fixed settings across all stations) and locally best models (whose settings are tailored to each station) as a function of the forecast horizon. Box plots give the distribution over all stations which has a median always below 0.01 points (corresponding to a 1 % variation in the RMSE), and at most 0.03 points.

7.4 Non-stationary effects and rolling window approach

The models described up to now, have been trained on all data before a cutoff date (01/01/2018), and tested on wind speed prediction with all available data after the same cutoff date. This approach could incur in significant errors if the series is non-stationary and the statistical distribution of data changes over time. In this case, more recent data can better represent the evolution of the process while old data become obsolete. To take into account the non-stationarity the model needs to be periodically updated (Cheng et al., 2015). Several self-updating models have been used in different frameworks, including stock market forecasting (Hota et al., 2017; Chou et al., 2018), urban traffic control (Mozaffari et al., 2015; Haworth et al., 2014), ocean wave energy prediction (Reikard, 2009) and streamflow estimation (Lima et al., 2016). The sliding window approach is a widely used updating technique that consists in periodically repeating training using a dataset from which obsolete data are removed and newly available data are added. The forgetting approach is a similar technique that consists in weighting the loss associated to the training examples to give more importance to recent data. Both approaches need to be retrained in order to include new information. In Messner et al. (2019) the authors consider the problem of forecasting wind speed from data collected by different wind farms in France and Denmark. They consider a multi-valued linear lasso learning algorithm and update the model with a forgetting approach. They show that updating the model improves performance with respect to the batch (non-updating) model, although the improvement may be somewhat marginal.

Motivated by these results, we test whether performance of our model improves when an updating procedure is incorporated. For the sake of simplicity a single station (A1) is considered, using a forecast horizon of 3 hours and just the locally best model settings (memory of 24 h, `zm` $\rightarrow$ `s` input-output design). The proposed update procedure is based on a simple sliding window approach, of which the following variants are considered (illustrated in Figure 7.9):

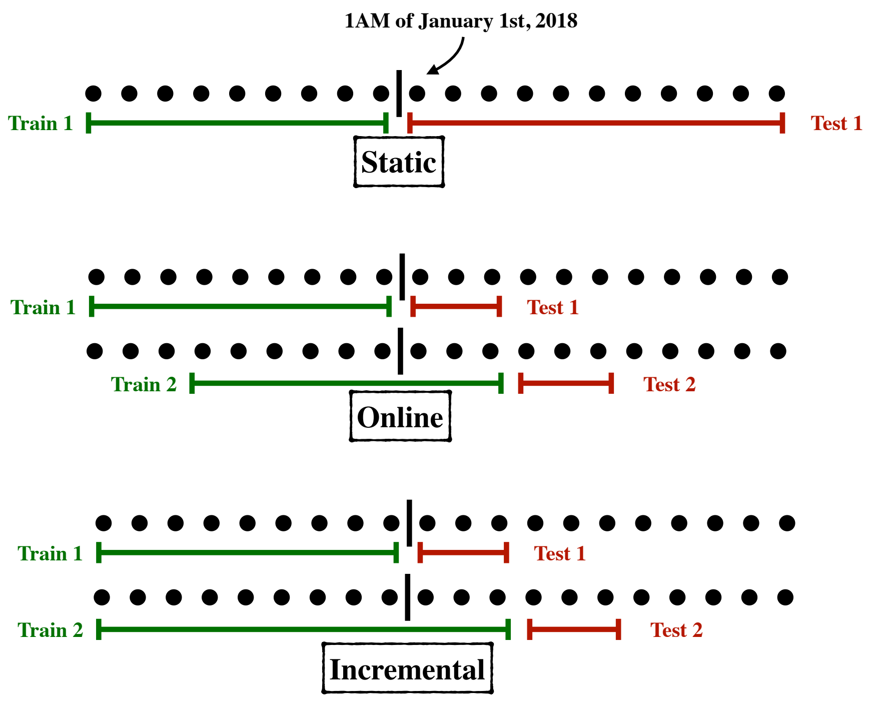


Figure 7.9 A graphical illustration of the updating processes for the static, online and incremental models.

- The *static* (non-updated) model is the one discussed in Section 7.3. It is trained only once, and uses a training set that starts from the beginning of the series until the last hour of the year 2017 (a total of $N = 9805$ data points).

- The *online* (updated) model, where starting from 1 AM of January 1st, 2018, the model is trained on the previous $N = 9805$ data points to predict the data of the next week ($7 \times 24 = 168$ samples). The procedure is repeated by training on the last 9805 points before 1 AM of January 7th and testing on the second week of January, and so on through the end of the series. We found that retraining more frequently than every 7 days provided no advantage (data not shown).

- The *incremental* (updated) model is similar to the *online* model but when re-training, all available data before the time we want to predict is considered. Therefore the training set size increases as the testing dates move forward.

- The *online (3m)* model is defined as the *online* model but using a smaller training set with 2232 samples (3 months) instead of the 9805 used in the online model. This training set spanning one season should demonstrate the potential benefits of forgetting obsolete data.

On the test set of the (A1) station, the NRMSE of the four models summarized in Figure 7.9 is as follows, *static*: 0.39±0.05, *online*: 0.38±0.05, *incremental*: 0.38±0.05 and *online (3m)*: 0.40±0.06[1], suggesting small differences. In fact, differences are indeed negligible, as the mean pairwise difference in NRMSE between any two models is smaller than the standard deviation. Note that discrepancies among the different models do sometimes occur as represented in Figure 7.10 (right panel), although these instances are rare.

This result may be counterintuitive, as updating the model with more recent information is expected to be beneficial. These benefits are modest in our case, as can be seen quantitatively by comparing the NRMSE and qualitatively by comparing the online *vs* static result in Figure 7.10, both in regular conditions (left panel) and irregular conditions (right panel). Additionally, these rolling window approaches come at the cost of increasing the computational load noticeably, since an expensive training step needs to be performed more frequently. On the other hand, the benefits of a large training set can be seen, especially in instances where the wind cycle is disrupted and discrepancies emerge among the different models Figure 7.10 (right panel). In this representative example, the shortest training set (Online 3m) performs poorly with respect to all other models. We propose that the smaller training set does not have enough statistical power to distinguish a variety of less common situations.

[1]Here, errors are the standard deviation of the NRMSE on each individual month

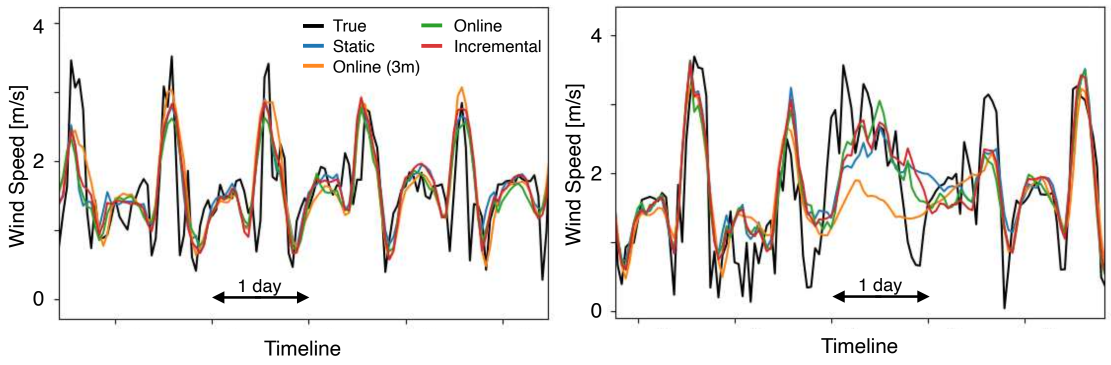


Figure 7.10 Observed wind speed (black) compared to predictions obtained with the 4 different models described in the text (colored lines, see legend).

7.5 Comparison with state-of-the-art models

In this section the performance of the best models of this paper is compared to that of other algorithms which have been proposed in the literature for predicting wind speed. An approximate comparison between two models (even when they are used on different datasets can be made by taking into account each model's improvement over the naïve persistence model. A metric γ_{RMSE} is defined which captures the improvement of a specific model ($\mathcal{F}$) over the persistence (Pers) on a specific dataset in terms of the Root Mean Squared Error RMSE $= \sqrt{\frac{1}{n}\sum_{t=1}^{n}(s_t - \widehat{s}_t)^2}$:

$$\gamma_{\text{RMSE}}(\mathcal{F}) = \frac{\text{RMSE}(\mathcal{F})}{\text{RMSE(Pers)}} \tag{7.9}$$

where RMSE($\hat{f}$) and RMSE(Pers) are the RMSE using predictions $\widehat{s}_t$ obtained from the model $\hat{f}$ and from the persistence respectively. The γ_{RMSE} metric can be used to compare algorithms evaluated on different datasets, as long as the error of the persistence is known. Nonetheless care must be taken to only compare results with the same forecasting horizon and sampling frequency, since the improvement over persistence strongly depends on these factors. In Figure 7.11 the γ_{RMSE} metric is used to compare our results with results from three papers that implement considerably more complex pipelines. Araya et al. (2020) designed a multi-scale deep learning model, based on the LSTM network architecture, with the aim of predicting hourly wind speed recorded at 20m high stations in four different sites in Chile. They report the averaged accuracy of 24h multi-step forecasts. C. Zhang, Wei, et al. (2016) used a Gaussian process stacked onto an autoregressive model, to predict wind speed at one step (hour) ahead for three different sites in China. Trebing et al. (2020) predict hourly wind speed in three danish cities at horizons of 6, 12, 18 and 24 hours. This latter work uses a convolutional neural network to blend both wind speed and other auxiliary data (*i.e.* temperature, pressure), from several cities at once.

	Araya best (m/s)	KRR(μ=24) zm $\to$ s (m/s)	KRR(μ=24) s $\to$ s (m/s)
e01	3.178	**2.662**	2.702
b08	1.673	**1.413**	1.488
d08	3.075	**2.061**	2.081
d05a	2.406	**2.135**	2.180

	Trebing best (m/s)	KRR(μ=48) zm $\to$ s (m/s)	KRR(μ=48) s $\to$ s (m/s)
6h	**1.675**	1.814	1.714
12h	2.144	2.205	**2.092**
18h	2.375	2.317	**2.244**
24h	2.463	2.369	**2.326**

Table 7.1 Comparing performances on available datasets used in recent works. RMSE of our models on the datasets of Araya et al. (2020) and Trebing et al. (2020). KRR with 24 and 48 hours of memory, and trained with different input variables was used. s, z, m indicate wind speed, zonal and meridional components and aux indicates auxiliary data from Trebing et al. (2020).

For each comparison a separate error metric is used for our locations, to be consistent with the compared work. The same *globally best* model is used for all our locations, while for the compared paper we take the best accuracy reported for each location or prediction horizon. The results (see Figure 7.11) show that the proposed model performs noticeably better than the multi-scale model of Araya and others. The results of Zhang and others, are very close to ours, and in this case the γ_{RMSE} metric is higher since persistence becomes harder to improve upon for 1 hour ahead predictions. Trebing and Mehrkanoon – which only provide averages over three different sites – use a model which is better than the proposed kernel ridge regression on short term predictions (6 and 12 hours ahead), and worse on long term predictions (18 and 24 hours ahead). This suggests that the correlations between stations, and the auxiliary data, provide an advantage on the short term, but lose importance over long-term predictions where a model which takes purely the wind into account takes the lead.

A more accurate comparison can also be undertaken by training our models on publicly available datasets used in Araya et al. (2020) and Trebing et al. (2020) adopting the same train/test splits and error metrics as in the original papers. Using the KRR model – which on our datasets clearly outperforms linear models – with the optimal memory as selected using 5-fold cross-validation, the two different input-output combinations which performed best on our data were tested. The results are shown in Table 7.1. We confirm that our model achieves significantly better performance than Araya's original work, both when using the two wind components (zm $\to$ s) and when using solely wind speed (s $\to$ s). Comparing against Trebing's results we also recover the trend suggested by the simple γ_{RMSE} metric: our model performs better on long-term and worse on short-term predictions. These results further indicate that the actual data, for example from multiple stations, using wind direction versus just wind speed, and auxiliary environmental measurements, plays a more important role than the model itself at improving the accuracy of wind speed prediction.

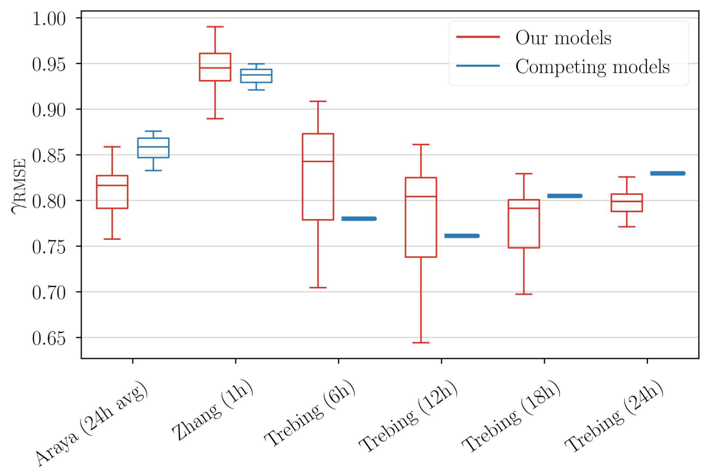


Figure 7.11 Comparing improvement over persistence among different papers. Red: Performance of the *globally best* (see Section 7.3.2) model for each forecast horizon trained on the respective datasets. Blue: best result for each site as reported in the literature. Performance for each model $\hat{f}$ is measured using the $\gamma_{\text{RMSE}}(\hat{f}) = \dfrac{\text{RMSE}(\hat{f})}{\text{RMSE(Pers)}}$ as described in Equation (7.9).

7.6 Conclusions

In this work, we develop a machine learning approach to predict wind at a future time purely from data, *i.e.* with no aid from mechanistic modeling. We conduct a systematic model selection through all our datasets, providing physical principles to understand the patterns that we observe and finally we propose a thorough comparison with state of the art algorithms. First, we compare models where both wind components *vs.* only wind speed are included in the input/output. In this way it is possible to quantify the role of wind direction, a variable which is often neglected in the literature, but has a clear physical meaning for the task of wind forecasting. We find that predicting wind speed from its two components (thus taking direction into account) is favorable in some locations but not in others. This result can be understood by noting that the dynamics of the atmosphere near the ground (where we focus our analysis) is strongly affected by the local orography and the features of the terrain near the point of interest. Thus depending on the location, wind may interact with surface elements in different ways depending on whether it is blowing in one direction or another. Thus it may be beneficial to include direction in the input, depending on the details near the location of interest. Predicting the two components of the wind separately and then reconstructing the speed from its components is never useful, likely because it involves two different models, leading to error buildup.

Second, we analyze the role of *memory*, defined as the number of past observations used as input. We find that a memory of 24 hours is optimal for all intermediate horizons. This observation can be explained by the presence of regular diurnal cycles in wind speed, which are often observed in the atmosphere. To corroborate this intuition we quantify the strength of the diurnal cycle and find that it correlates strongly with the benefits of a 24-hour memory. On the other hand, the diurnal cycle does not have a strong influence on forecasting at short and long horizons (here 1 hour and 24 hours). At a short horizon the strongest predictor is wind persistence and not the diurnal cycle, while at a long horizon the chaotic nature of the atmosphere makes forecasting very hard, and past history provides little information. Such arguments are corroborated by the observation that non-linear models are clearly beneficial at intermediate horizons and only marginally at short and long horizons. Another finding is that using the same model settings (*i.e.*, memory, model type, etc.) across locations provides almost the same accuracy as using model settings tailored to each site (typically the gains are below 1 % in NRMSE). Similarly taking into account non-stationary effects does not provide significant accuracy improvement.

Third, we seek to compare the proposed algorithm to the state of the art. A major hurdle is the lack of benchmarking datasets: most manuscripts test their proposed algorithms on their own dataset – whether public or proprietary. Clearly, to establish a fair comparison, different algorithms must be run on the same data. Hence to compare the models proposed in this paper with competing models we could either (a) run the competing algorithm on our Liguria and Abruzzo datasets, or (b) run the proposed algorithm on the datasets used in competing work. The first option requires to re-implement competing algorithms, which is time consuming and may result in artificial differences due to technical variations in the software. Hence we choose the second option, although it limits the possible comparisons to those works where data is made publicly available.

The main conclusion is that when the proposed models are run on the datasets published in two of the most recent papers, the obtained accuracy is competitive with state of the art algorithms. This is somewhat surprising because recent papers make extensive use of deep architectures and complex pipelines, sometimes enriching the input with additional variables. On the contrary, the proposed accelerated kernel ridge regression models are considerably simpler to implement and additionally benefit from sound theoretical guarantees. This equivalence suggests that carefully optimizing the design of simple models, and paying attention to the input and output variables, can be a fruitful alternative to the development of complex architectures that are typically harder to explain.

Predictions of wind speed close to ground remain challenging both for physics-based models and for purely data-driven strategies. Mechanistic models pose conceptual as well as computational challenges. Atmospheric turbulence couples many spatial scales, hence numerical solutions of the equation of motion require a massive number of grid cells. Moreover, unmodeled mechanisms may affect the solutions. This is especially true close to the ground, where interactions with the orography and local ground features cannot be modeled in detail. Data-driven approaches may recover some of these unmodeled effects statistically, but their performance is limited by the lack of information on the physical processes that take place in the atmosphere at different locations. Some of these information may be recovered by using spatiotemporal wind data as input, an approach that appears promising from current literature (Trebing et al., 2020; Zhu et al., 2018; Xu et al., 2022; Messner et al., 2019). More systematic hybrid approaches based on data assimilation techniques hold the promise to achieve the ideal merge of a mechanistic and a data-driven approach.

Chapter 8

K-Planes: Explicit Radiance Fields in Space, Time, and Appearance

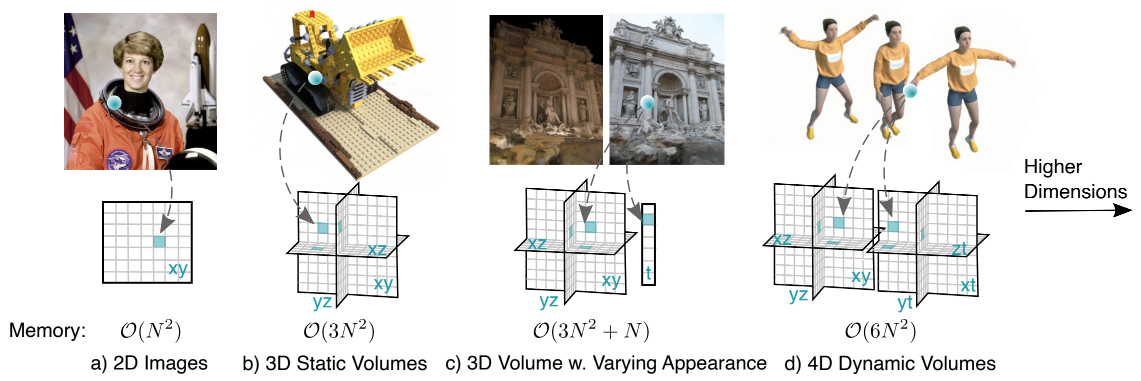


Figure 8.1 Planar factorization of d-dimensional spaces. We propose a simple planar factorization for volumetric rendering that naturally extends to arbitrary-dimensional spaces, and that scales gracefully with dimension in both optimization time and model size. We show the advantages of our approach on 3D static volumes, 3D photo collections with varying appearances, and 4D dynamic videos.

In this final chapter, we leave the main theme of this thesis – large scale kernel methods – behind, in order to explore the topics of scene reconstruction and novel-view synthesis. A burgeoning topic in recent years, at the intersection of computer graphics and machine learning, novel-view synthesis is best defined by considering the inputs and outputs to the problem. Given a set of *calibrated* pictures of an object or scene (the inputs), we seek to obtain a model which allows to produce a new rendering of the same scene from a different view-point (the output), which was not present in the training data. In practice this allows to automatically create high-fidelity 3D models of objects and scenes starting from a handful of pictures, with applications in creating assets for cinema and video-games, augmented and virtual reality. The most prominent approach to solving the problem goes under the name of neural radiance fields, or NeRF, and will be presented in Section 8.2. It was first proposed in Mildenhall, Srinivasan, Tancik, et al. (2020) and has since been extended in numerous interesting ways. Our interest for this problem stems from the observation that, while most of computer vision has been steadily moving towards the use of ever larger deep learning models with correspondingly high computational and data cost, novel-view synthesis has remained a problem which does not need the huge representation capability of *e.g.* large scale transformers. In fact, it has been shown (Fridovich-Keil, Yu, et al., 2022; A. Chen et al., 2022) that state of the art reconstruction accuracy can be achieved even with minimal use of non-linearities and neural networks.

In our work we present k-planes: a unified, white-box model for radiance fields, in arbitrary dimensions. This last attribute signifies the possibility of using the proposed model to represent not only static, 3-dimensional scenes, but also dynamic, time-varying 4-dimensional scenes. A planar factorization using $\binom{d}{2}$ planes to represent a d-dimensional scenes, provides a seamless way to go from static ($d = 3$) to dynamic ($d = 4$) scenes. In Section 8.3 we describe in detail our model, along with several dimension-specific priors which can be added to speed up learning and improve performance. We use a linear feature decoder with a learned color basis that yields similar performance as a nonlinear black-box MLP decoder. As described in Section 8.4, k-planes yields competitive (and often state-of-the-art) reconstruction fidelity across a range of synthetic and real, static and dynamic, fixed and varying appearance scenes. Video results and code to reproduce our results are available at sarafridov.github.io/K-Planes.

8.1 Introduction

Recent interest in dynamic radiance fields demands representations of 4D volumes. However, storing a 4D volume directly is prohibitively expensive due to the curse of dimensionality. Several approaches have been proposed to factorize 3D volumes for static radiance fields, but these do not easily extend to higher dimensional volumes.

We propose a factorization of 4D volumes that is simple, interpretable, compact, and yields fast training and rendering. Specifically, we use six planes to represent a 4D volume, where the first three represent space and the last three represent space-time changes, as illustrated in Figure 8.1(d). This decomposition of space and space-time makes our model interpretable, *i.e.* dynamic objects are clearly visible in the space-time planes, whereas static objects only appear in the space planes. This interpretability enables dimension-specific priors in time and space.

More generally, our approach yields a straightforward, prescriptive way to select a factorization of any dimension with 2D planes. For a d-dimensional space, we use $k = \binom{d}{2}$ ("*d-choose*-2") *k-planes*, which represent every pair of dimensions — for example, our model uses $\binom{4}{2} = 6$ *hex-planes* in 4D and reduces to $\binom{3}{2} = 3$ *tri-planes* in 3D. Choosing any other set of planes would entail either using more than k planes and thus occupying unnecessary memory, or using fewer planes and thereby forfeiting the ability to represent some potential interaction between two of the d dimensions. We call our model k-planes; Figure 8.1 illustrates its natural application to both static and dynamic scenes.

Most radiance field models entail some black-box components with their use of MLPs. Instead, we seek a simple model whose functioning can be inspected and understood. We find two design choices to be fundamental in allowing k-planes to be a white-box model while maintaining reconstruction quality competitive with or better than previous black-box models (T. Li et al., 2022; Pumarola et al., 2021): (a) features from our k-planes are *multiplied* together rather than added, as was done in prior work (Chan et al., 2022; A. Chen et al., 2022), and (b) our linear feature decoder uses a learned basis for view-dependent color, enabling greater adaptivity including the ability to model scenes with variable appearance. We show that an MLP decoder can be replaced with this linear feature decoder only when the planes are multiplied, suggesting that the commonly-used MLP decoder is involved in both view-dependent color and determining spatial structure.

Our factorization of 4D volumes into 2D planes leads to a high compression level without relying on MLPs, using 200 MB to represent a 4D volume whose direct representation at the same resolution would require more than 300 GB, a compression rate of three orders of magnitude. Furthermore, despite not using any custom CUDA kernels, k-planes trains orders of magnitude faster than prior implicit models and on par with concurrent hybrid models.

In summary, we present the first white-box, interpretable model capable of representing radiance fields in arbitrary dimensions, including static scenes, dynamic scenes, and scenes with variable appearance. Our k-planes model achieves competitive performance across reconstruction quality, model size, and optimization time across these varied tasks, without any custom CUDA kernels.

8.2 Related Work

K-planes is an interpretable, explicit model applicable to static scenes, scenes with varying appearances, and dynamic scenes, with compact model size and fast optimization time. Our model is the first to yield all of these attributes, as illustrated in Table 8.1. We further highlight that k-planes satisfies this in a simple framework that naturally extends to arbitrary dimensions.

8.2.1 Volumetric rendering

We use the same volume rendering formula as NeRF (Mildenhall, Srinivasan, Tancik, et al., 2020), originally proposed in Max (1995), where the color of a pixel is represented as a sum

	Static	Appearance	Dynamic	Fast	Compact	Explicit
NeRF	✓	✗	✗	✗	✓	✗
NeRF-W	✓	✓	✗	✗	✓	✗
DVGO	✓	✗	✗	✓	✗	✗
Plenoxels	✓	✗	✗	✓	✗	✓
Instant-NGP, TensoRF	✓	✗	✗	✓	✓	✗[1]
DyNeRF, D-NeRF	–	✗	✓	✗	✓	✗
TiNeuVox, Tensor4D	–	✗	✓	✓	✓	✗
MixVoxels, V4D	–	✗	✓	✓	✗	✗
NeRFPlayer	–	✗	✓	✓	✓[2]	✗
K-planes hybrid (Ours)	✓	✓	✓	✓	✓	✗
K-planes explicit (Ours)	✓	✓	✓	✓	✓	✓

[1] TensoRF offers both hybrid and explicit versions, with a small quality gap [2] NerfPlayer offers models at different sizes, the smallest of which with < 100 million parameters but the largest with > 300 million parameters

Table 8.1 Related work overview. We present a simple decomposition that works for a diverse set of scenes and tasks (static, varying appearance, and dynamic). It has a low memory usage (compact) and fast training and inference time (fast). Here "fast" includes any model that can optimize within a few (< 6) hours on a single GPU, and "compact" denotes models that use less than roughly 100 million parameters. "Explicit" denotes white-box models that do not rely on MLPs.

over samples taken along the corresponding ray through the volume:

$$\sum_{i=1}^{N} \exp\left(-\sum_{j=1}^{i-1} \sigma_j \delta_j\right) (1 - \exp(-\sigma_i \delta_i))\mathbf{c}_i \tag{8.1}$$

where the first exp represents ray transmission to sample i, $1 - \exp(-\sigma_i \delta_i)$ is the absorption by sample i, σ_i is the (post-activation) density of sample i, and $\mathbf{c}_i$ is the color of sample i, with distance δ_i to the next sample.

Spatial decomposition. NeRF (Mildenhall, Srinivasan, Tancik, et al., 2020) proposed to model color and density at each 3D position in a fully implicit way, with a large neural network queried many times during optimization. This meant it had high training times, and was essentially a black-box from an interpretability perspective. Several subsequent works have used explicit representations of scene geometry to reduce the optimization time (at the expense of a larger memory footprint). Plenoxels (Fridovich-Keil, Yu, et al., 2022) proposed a fully

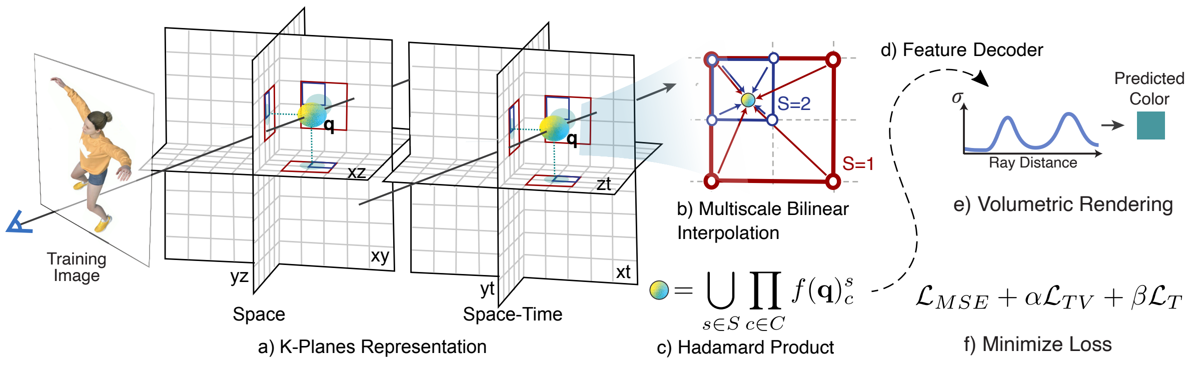

Training Image
yz
xy
xz
q
Space
yt
xt
zt
Space-Time
a) K-Planes Representation
S=2
S=1
b) Multiscale Bilinear Interpolation
$= \bigcup_{s \in S} \prod_{c \in C} f(\mathbf{q})_c^s$
c) Hadamard Product
d) Feature Decoder
σ
Ray Distance
Predicted Color
e) Volumetric Rendering
$\mathcal{L}_{MSE} + \alpha\mathcal{L}_{TV} + \beta\mathcal{L}_T$
f) Minimize Loss

Figure 8.2 Method overview. (a) Our k-planes representation factorizes 4D dynamic volumes into six planes, three for space and three for spatiotemporal variations. To obtain the value of a 4D point $\mathbf{q} = (x, y, z, t)$, we first project the point into each plane, in which we (b) do multiscale bilinear interpolation. (c) The interpolated values are multiplied and then concatenated over the S scales. (d) These features are decoded either with a small MLP or our explicit linear decoder. (e) We follow the standard volumetric rendering formula to predict ray color and density. The model is optimized by (f) minimizing the reconstruction loss with simple regularization in space and time.

explicit model, where color and density were queried on a stored 3D feature grid, using trilinear interpolation. This reduced the optimization time from hours to a few minutes. However, their explicit grid representation of 3D volumes, and that of DVGO (C. Sun et al., 2022), grows exponentially with the number of dimensions, and makes it challenging to scale to high resolution and completely intractable for 4D dynamic volumes.

Hybrid methods (C. Sun et al., 2022; Müller et al., 2022; A. Chen et al., 2022) retain some explicit geometric structure, often compressed by a spatial decomposition, alongside a small MLP feature decoder. Instant-NGP (Müller et al., 2022) proposed a multiresolution voxel grid encoded implicitly via a hash function, allowing fast optimization and rendering with a compact model. TensoRF (A. Chen et al., 2022) achieved similar model compression and speed by replacing the 3D voxel grid with a tensor decomposition into planes and vectors. In a generative setting, EG3D (Chan et al., 2022) proposed a similar spatial decomposition into three planes, whose values are added together to represent a 3D volume.

Our work is inspired by the explicit modeling of Plenoxels as well as these compressed spatial decompositions, particularly the tri-plane model of Chan et al. (2022), the tensor decomposition of A. Chen et al. (2022), and the multiscale grid model of Müller et al. (2022). We also draw inspiration from Extreme MRI (F. Ong et al., 2020), which uses a multiscale low-rank decomposition to represent 4D dynamic volumes in magnetic resonance imaging. These spatial decomposition methods have been shown to offer a favorable balance of memory efficiency and optimization time for static scenes. However, it is not obvious how to extend these factorizations to 4D volumes in a memory-efficient way. K-planes defines a unified framework that enables efficient and interpretable factorizations of 3D and 4D volumes and trivially extends to even higher dimensional volumes.

Dynamic volumes. Applications such as Virtual Reality (VR) and Computed Tomography (CT) often require the ability to reconstruct 4D volumes. Several works have proposed extensions of NeRF to dynamic scenes. The two most common schemes are (1) modeling a deformation field on top of a static *canonical* field (Pumarola et al., 2021; Tretschk et al., 2021; Park, Sinha, Barron, et al., 2021; Du et al., 2021; Yuan et al., 2021; Fang et al., 2022; Z. Li, Niklaus, et al., 2021), or (2) directly learning a radiance field conditioned on time (Xian et al., 2021; Z. Li, Niklaus, et al., 2021; C. Gao et al., 2021; T. Li et al., 2022; Park, Sinha, Hedman, et al., 2021). The former makes decomposing static and dynamic components easy (Yuan et al., 2021; T. Wu et al., 2022), but struggles with changes in scene topology (*e.g.* when a new object appears), while the latter makes disentangling static and dynamic objects hard. A third strategy is to choose a compact 3D spatial representation and essentially repeat it at each time step (*e.g.* NeRFPlayer (Song et al., 2022)), resulting in a model that ignores interactions between space and time and can become impractically large for long videos.

Further, some of these models are fully implicit (Pumarola et al., 2021; T. Li et al., 2022) and thus suffer from extremely long training times (*e.g.* DyNeRF used 8 GPUs for 1 week to train a single scene), as well as being completely black-box. Others use partially explicit decompositions for video (Fang et al., 2022; Guo et al., 2022; F. Wang et al., 2022; Gan et al., 2022; Shao et al., 2022; J.-W. Liu et al., 2022; Lombardi et al., 2019; Song et al., 2022), usually combining some voxel or spatially decomposed feature grid with one or more MLP components for feature decoding and/or representing scene dynamics. Most closely related to k-planes is Tensor4D Shao et al., 2022, which uses 9 planes to decompose 4D volumes. K-planes is less redundant (*e.g.* Tensor4D includes two yt planes), does not rely on multiple MLPs, and offers a simpler factorization that naturally generalizes to static and dynamic scenes. Our method combines a fully explicit representation with a built-in decomposition of static and dynamic components, the ability to handle arbitrary topology and lighting changes over time, fast optimization, and compactness.

Appearance embedding. Reconstructing large environments from photographs taken with varying illumination is another domain in which implicit methods have shown appealing results, but hybrid and explicit approaches have not yet gained a foothold. NeRF-W (Martin-Brualla et al., 2021) was the first to demonstrate photorealistic view synthesis in such environments. They augment a NeRF-based model with a learned global appearance code per frame, enabling it to explain away changes in appearance, such as time of day. With Generative Latent Optimization (GLO) (Bojanowski et al., 2017), these appearance codes can further be used to manipulate the scene appearance by interpolation in the latent appearance space. Block-NeRF (Tancik et al., 2022) employs similar appearance codes.

We show that our k-planes representation can also effectively reconstruct these unbounded environments with varying appearance. We similarly extend our model – either the learned color basis in the fully explicit version, or the MLP decoder in the hybrid version – with a global appearance code to disentangle global appearance from a scene without affecting geometry. To the best of our knowledge, ours is both the first fully explicit and the first hybrid method to successfully reconstruct these challenging scenes.

8.3 K-planes model

We propose a simple and interpretable model for representing scenes in arbitrary dimensions. Our representation yields low memory usage and fast training and rendering. The k-planes factorization, illustrated in Figure 8.2, models a d-dimensional scene using $k = \binom{d}{2}$ planes representing every combination of two dimensions. For example, for static 3D scenes, this results in *tri-planes* with $\binom{3}{2} = 3$ planes representing xy, xz, and yz. For dynamic 4D scenes, this results in *hex-planes*, with $\binom{4}{2} = 6$ planes including the three space-only planes and three space-time planes xt, yt, and zt. Should we wish to represent a 5D space, we could use $\binom{5}{2} = 10$ *deca-planes*. In the following section, we describe the 4D instantiation of our k-planes factorization.

8.3.1 Hex-planes

The Hex-planes factorization uses six planes. We refer to the space-only planes as $\mathbf{P}_{xy}$, $\mathbf{P}_{xz}$, and $\mathbf{P}_{yz}$, and the space-time planes as $\mathbf{P}_{xt}$, $\mathbf{P}_{yt}$, and $\mathbf{P}_{zt}$. Assuming symmetric spatial and temporal resolution N for simplicity of illustration, each of these planes has shape $N \times N \times M$, where M is the size of stored features that capture the density and view-dependent color of the scene.

We obtain the features of a 4D coordinate $\boldsymbol{q} = (i, j, k, \tau)$ by normalizing it between $[0, N)$ and projecting it onto these six planes

$$f(\boldsymbol{q})_c = \psi(\mathbf{P}_c, \pi_c(\boldsymbol{q})), \tag{8.2}$$

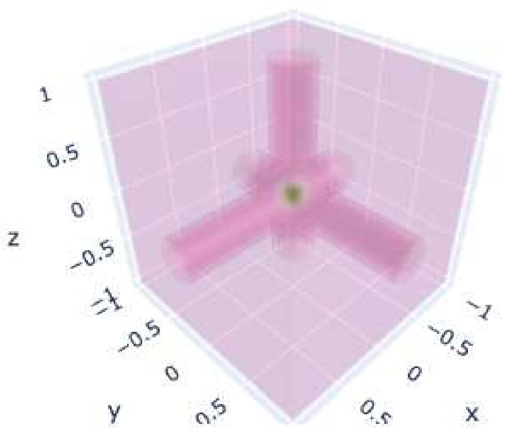
where π_c projects $\boldsymbol{q}$ onto the c'th plane and ψ denotes bilinear interpolation of a point into a regularly spaced 2D grid. We repeat Equation (8.2) for each plane $c \in C$ to obtain feature vectors $f(\boldsymbol{q})_c$. We combine these features over the six planes using the Hadamard product (elementwise multiplication) to produce a final feature vector of length M

$$f(\boldsymbol{q}) = \prod_{c \in C} f(\boldsymbol{q})_c. \tag{8.3}$$

These features will eventually be decoded into color and density using either a linear decoder or an MLP, described in Section 8.3.3.

Why Hadamard product? In 3D, k-planes reduces to the tri-plane factorization, which is similar to that of Chan et al. (2022), except that the elements are multiplied. A natural question is why we multiply rather than add, as has been used in prior work with tri-plane models (Chan et al., 2022; S. Peng, Niemeyer, et al., 2020). Figure 8.3 illustrates that combining the planes by multiplication allows k-planes to produce spatially localized signals, which is not possible with addition.

This selection ability of the Hadamard product produces substantial rendering improvements for linear decoders and modest improvement for MLP decoders, as shown in Table 8.2. This suggests that the MLP decoder is involved in both view-dependent color and determining spatial structure. The Hadamard product relieves the feature decoder of this extra task and makes it possible to reach similar performance using a linear decoder solely responsible for view-dependent color.



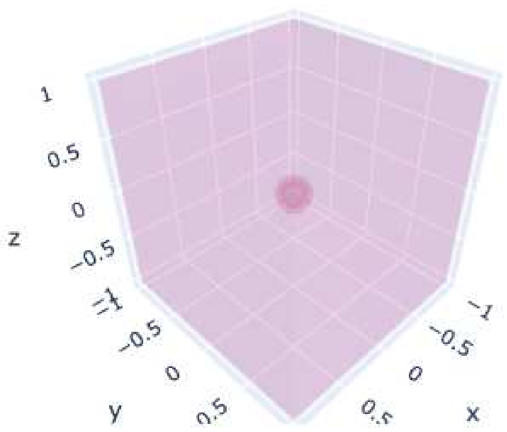


Figure 8.3 Addition versus Hadamard product. Elementwise addition of plane features (left) compared to multiplication (right), illustrated in a 3D tri-plane example. In both cases a single entry in each plane is positive and the rest are zero, selecting a single 3D point by multiplication but producing intersecting lines by addition. This selection ability of multiplication improves the expressiveness of our explicit model.

8.3.2 Interpretability

The separation of space-only and space-time planes makes the model interpretable and enables us to incorporate dimension-specific priors. For example, if a region of the scene never moves, its temporal component will always be 1 (the multiplicative identity), thereby just using the features from the space planes. This offers compression benefits since a static region can easily

Table 8.2 Ablation study over Hadamard product. Multiplication of plane features yields a large improvement in PSNR ↑ for our explicit model, whereas our hybrid model can use its MLP decoder to compensate for the less expressive addition of planes. This experiment uses the static *Lego* scene (Mildenhall, Srinivasan, Tancik, et al., 2020) with 4 scales: 64, 128, 256, and 512, and 32 features per scale.

Plane Combination	Explicit	Hybrid	# params ↓
Multiplication	36.36	36.97	34M
Addition	30.59	36.73	34M

be identified and compactly represented. Furthermore, the space-time separation improves interpretability, *i.e.* we can track the changes in time by visualizing the elements in the time-space planes that are not 1. This simplicity, separation, and interpretability make adding priors straightforward.

Multiscale planes. To encourage spatial smoothness and coherence, our model contains multiple copies at different spatial resolutions, for example 64, 128, 256, and 512. Models at each scale are treated separately, and the M-dimensional feature vectors from different scales are concatenated together before being passed to the decoder. The red and blue squares in Figure 8.2 a, b) illustrate bilinear interpolation with multiscale planes. Inspired by the multiscale hash mapping of Instant-NGP (Müller et al., 2022), this representation efficiently encodes spatial features at different scales, allowing us to reduce the number of features stored at the highest resolution and thereby further compressing our model. Empirically, we do not find it necessary to represent our time dimension at multiple scales. Table 8.3 presents a small ablation experiment to evaluate the impact of using multiple scales. Including lower-resolution copies of the model brings up to 0.8 PSNR point improvement with the explicit model, while a smaller improvement can be observed using the hybrid model.

Scales	Explicit PSNR ↑	Hybrid PSNR ↑	# params ↓
$64, 128, 256, 512$	36.36	36.97	34M
$128, 256, 512$	**36.44**	**37.23**	34M
$256, 512$	36.40	37.19	32M
512	35.65	36.92	26M
$64, 128, 256$	33.79	35.22	9M

Table 8.3 Ablation study over scales. Including even a single lower scale improves performance, particularly for our explicit model, with a relatively small overhead in train time and model size. Using lower scales only (excluding resolution 512^3) reduces both quality (PSNR) and model size substantially. This experiment uses the static *Lego* scene; each scale uses 32 features.

Total variation in space. Spatial total variation regularization encourages sparse gradients, encoding the prior that edges are sparse in space. We encourage this in 1D over the spatial dimensions of each of our space-time planes and in 2D over our space-only planes:

$$\mathcal{L}_{TV}(\mathbf{P}) = \frac{1}{|C|n^2} \sum_{c,i,j} (\|\mathbf{P}_c^{i,j} - \mathbf{P}_c^{i-1,j}\|_2^2 + \|\mathbf{P}_c^{i,j} - \mathbf{P}_c^{i,j-1}\|_2^2), \tag{8.4}$$

where i, j are indices on the plane's resolution. Total variation is a common regularizer in inverse problems and was used in Plenoxels (Fridovich-Keil, Yu, et al., 2022) and TensoRF (A. Chen et al., 2022).

Smoothness in time. We encourage smooth motion with a 1D Laplacian (second derivative) filter

$$\mathcal{L}_{smooth}(\mathbf{P}) = \frac{1}{|C|n^2} \sum_{c,i,t} \|\mathbf{P}_c^{i,t-1} - 2\mathbf{P}_c^{i,t} + \mathbf{P}_c^{i,t+1}\|_2^2, \tag{8.5}$$

to penalize sharp "acceleration" over time. We only apply this regularizer on the time dimension of our space-time planes. See Table 8.4 for an ablation experiment in which we verify the positive impact of this regularizer.

Time Smoothness Weight	Explicit PSNR ↑	Hybrid PSNR ↑
0.0	29.69	29.86
0.001	30.78	31.24
0.01	**31.17**	31.56
0.1	31.10	**31.50**
1.0	30.67	31.22
10.0	29.72	30.57

Table 8.4 Ablation over temporal smoothness regularization. For both models, a temporal smoothness weight of 0.01 to 0.1 is best, with PSNR degrading with over- or under-regularization. This experiment uses the *Jumping Jacks* scene with 4 scales: 64, 128, 256, and 512, and 32 features per scale.

Sparse transients. We want the static part of the scene to be modeled by the space-only planes. We encourage this separation of space and time by initializing the features in the space-time planes as 1 (the multiplicative identity) and using an ℓ_1 regularizer on these planes during training:

$$\mathcal{L}_{sep}(\mathbf{P}) = \sum_c \|\mathbf{1} - \mathbf{P}_c\|_1, \qquad c \in \{xt, yt, zt\}. \tag{8.6}$$

In this way, the space-time plane features of the k-planes decomposition will remain fixed at 1 if the corresponding spatial content does not change over time.

8.3.3 Feature decoders

We offer two methods to decode the M-dimensional temporally- and spatially-localized feature vector $f(\boldsymbol{q})$ from Equation (8.3) into density, σ, and view-dependent color, $\boldsymbol{c}$.

Learned color basis: a linear decoder and explicit model. Plenoxels (Fridovich-Keil, Yu, et al., 2022), Plenoctrees (Yu et al., 2021), and TensoRF (A. Chen et al., 2022) proposed model versions where spatially-localized features are interpreted as coefficients of the spherical harmonic basis functions and used to describe view-dependent color. Such spherical harmonic decoders can offer both high-fidelity reconstructions and enhanced interpretability compared to MLP decoders. However, spherical harmonic coefficients are difficult to optimize, and their expressiveness is limited by the number of spherical harmonic basis functions used. Often only harmonics up to degree two are used, thereby producing low-frequency (blurry) specular reflections.

Instead, we replace the spherical harmonic basis functions with a learned basis, retaining the interpretability of treating features as coefficients for a linear decoder yet increasing the expressiveness of the basis and allowing it to adapt to each scene, as was proposed in NeX (Wizadwongsa et al., 2021). We represent the basis as a small MLP that predicts a red $b_R(\boldsymbol{d}) \in \mathbb{R}^M$, green $b_G(\boldsymbol{d}) \in \mathbb{R}^M$, and blue $b_B(\boldsymbol{d}) \in \mathbb{R}^M$ *basis* whose input is view direction $\boldsymbol{d}$. The MLP serves as an adaptive drop-in replacement for the spherical harmonic basis functions repeated over the three color channels. We obtain the color values

$$RGB(\boldsymbol{q}, \boldsymbol{d}) = \bigcup_{i \in \{R,G,B\}} \langle f(\boldsymbol{q}), b_i(\boldsymbol{q}, \boldsymbol{d}) \rangle, \tag{8.7}$$

where $\cup$ denotes concatenation (union). Similarly, we optimize a linear decoder for density by predicting a $b_\sigma \in \mathbb{R}^M$ density *basis*, independent of the view direction:

$$\sigma(\boldsymbol{q}) = \langle f(\boldsymbol{q}), b_\sigma(\boldsymbol{q}) \rangle, \tag{8.8}$$

where $\sigma(\boldsymbol{q})$ is the density of a d-dimensional point. The predicted color and density values are finally passed through a nonlinear function to ensure they are in the valid ranges: we apply a sigmoid to the color values to ensure they lie in $[0, 1]$, and an exponential (with truncated gradient) to the density values to ensure they are nonnegative.

MLP decoder: a hybrid model. Our model can also be used with an MLP decoder like that of Instant-NGP (Müller et al., 2022) and DVGO (C. Sun et al., 2022), turning it into a hybrid model. In this version, features are decoded by two small MLPs, one g_σ that maps the spatially-localized features into density σ and additional features f', and another g_{RGB} that maps these additional features as well as the embedded view direction $\gamma(\boldsymbol{d})$ into RGB color

$$\begin{aligned} \sigma(\boldsymbol{q}), \hat{f}(\boldsymbol{q}) &= g_\sigma(f(\boldsymbol{q})) \\ \mathrm{RGB}(\boldsymbol{q}, \boldsymbol{d}) &= g_{\mathrm{RGB}}(f'(\boldsymbol{q}), \gamma(\boldsymbol{d})). \end{aligned} \tag{8.9}$$

As in the linear decoder case, the predicted density and color values are finally normalized via exponential (with truncated gradient) and sigmoid, respectively.

Global appearance. We also show a simple extension of our k-planes model that enables it to represent scenes with consistent, static geometry viewed under varying lighting or appearance conditions. Such scenes appear in the Phototourism dataset of Jin et al. (2021), depicting famous landmarks photographed by tourists at different times of day and in different weather. To model this variable appearance, we augment k-planes with a matrix of shape $M{\times}T$, an M-vector of features for each training image index $1 \ldots T$. Similar to NeRF-W (Martin-Brualla et al., 2021), we optimize this per-image feature vector and pass it as an additional input to either our MLP learned color basis b_R, b_G, b_B, in our explicit version, or to our MLP color decoder g_{RGB}, in our hybrid version, so that in either case it can affect color but not geometry.

8.3.4 Optimization details

Full details of our optimization may be found in our released code. Here we specify three components we modify from prior work to improve flexibility and speed.

Contraction and normalized device coordinates. For forward-facing scenes, we apply normalized device coordinates (NDC) (Mildenhall, Srinivasan, Tancik, et al., 2020) to better allocate our resolution while enabling unbounded depth. We also implement an ℓ_∞ version (rather than ℓ_2) of the scene contraction proposed in Mip-NeRF 360 (Barron, Mildenhall, Verbin, et al., 2022), which we use on the unbounded Phototourism scenes.

Proposal sampling. We use a variant of the proposal sampling strategy from Mip-NeRF 360, with a small instance of k-planes as each density model. Proposal sampling works by first using a fixed sampling strategy along each ray into a density model, and then adaptively choosing samples based on the densities of these initial samples. We use a two-stage proposal sampler, resulting in both fewer samples that must be evaluated in our full model and in sharper details by placing those samples closer to object surfaces. The density models used for proposal sampling are trained with the histogram loss (Barron, Mildenhall, Verbin, et al., 2022).

Importance sampling. For multiview dynamic scenes, we implement a version of the importance sampling based on temporal difference (IST) strategy from DyNeRF (T. Li et al., 2022). During the last portion of optimization, we sample training rays according to the maximum absolute variation in their color within 25 frames, slightly less than one second before

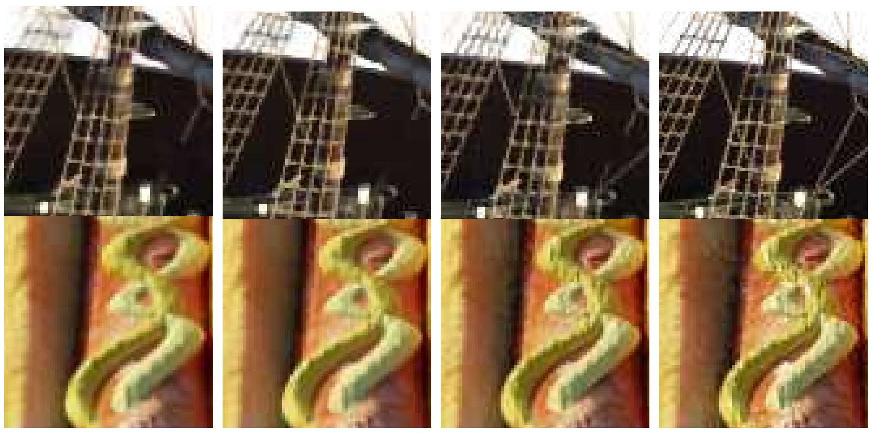


Figure 8.4 Zoomed qualitative results on static NeRF scenes. Visual comparison of k-planes, TensoRF (A. Chen et al., 2022), and the ground truth, on *ship* (top) and *hotdog* (bottom).

or after. This results in higher sampling probabilities in the dynamic region. We apply this strategy after the static scene has converged through standard optimization with uniformly sampled rays. In our experiments, this sampling strategy has only a modest impact on full-frame metrics but improves visual quality in the small dynamic region. Note that importance sampling cannot be used for monocular videos or any datasets with moving cameras.

8.4 Results

We demonstrate the broad applicability of our planar decomposition via experiments in three domains: static scenes (both bounded 360° and unbounded forward-facing), dynamic scenes (forward-facing multi-view and bounded 360° monocular), and Phototourism scenes with variable appearance. For all experiments, we report the metrics PSNR (pixel-level similarity) and SSIM[1] (Z. Wang et al., 2004) (structural similarity), as well as approximate training time and number of parameters (in millions), in Table 8.5. Blank entries in Table 8.5 denote baseline methods for which the corresponding information is not readily available. Full per-scene results may be found in Fridovich-Keil, Meanti, et al. (2023).

8.4.1 Static scenes

We first demonstrate our triplane model on the bounded, 360°, synthetic scenes from NeRF (Mildenhall, Srinivasan, Tancik, et al., 2020). We use a model with four symmetric spatial resolutions $N \in \{64, 128, 256, 512\}$ and feature length $M = 32$ at each scale; please see the full

[1]Note that among prior work, some evaluate using a public implementation of SSIM from MipNeRF (Barron, Mildenhall, Tancik, et al., 2021) whereas others use the implementation in scikit-image, which tends to produce higher values. For fair comparison on each dataset we make a best effort to use the same SSIM implementation as the relevant prior work.

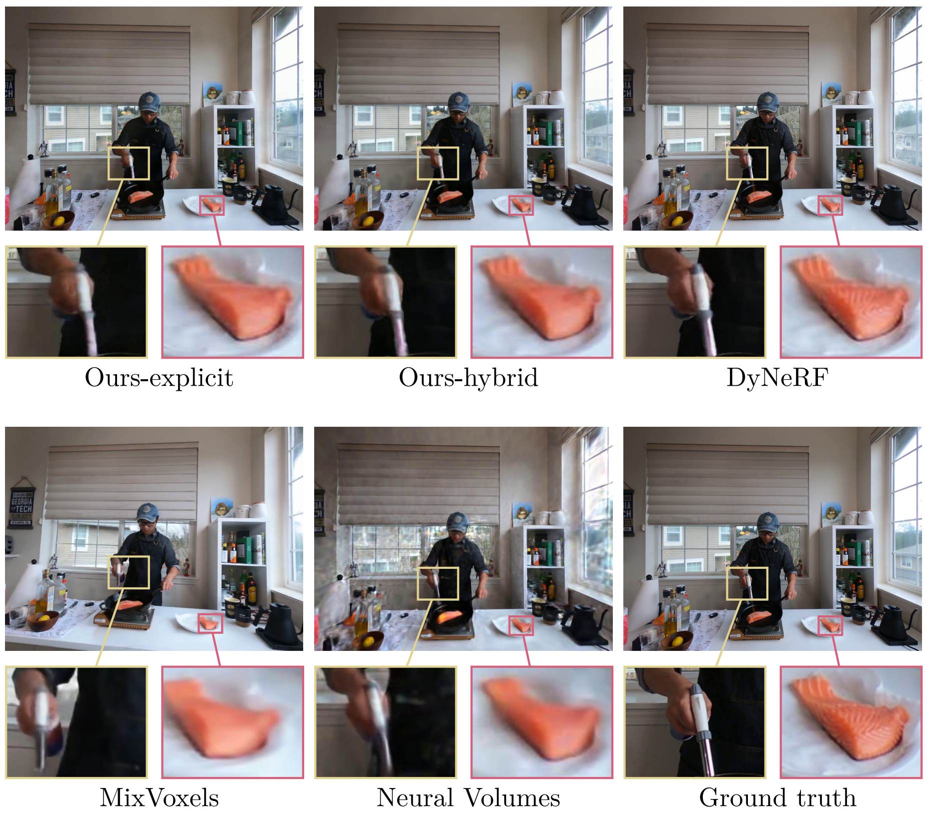


Figure 8.5 Qualitative video results. Our hexplane model rivals the rendering quality of state-of-the-art neural rendering methods. Our renderings were obtained after at most 4 hours of optimization on a single GPU whereas DyNeRF trained for a week on 8 GPUs. MixVoxels frame comes from a slightly different video rendering, and is thus slightly shifted.

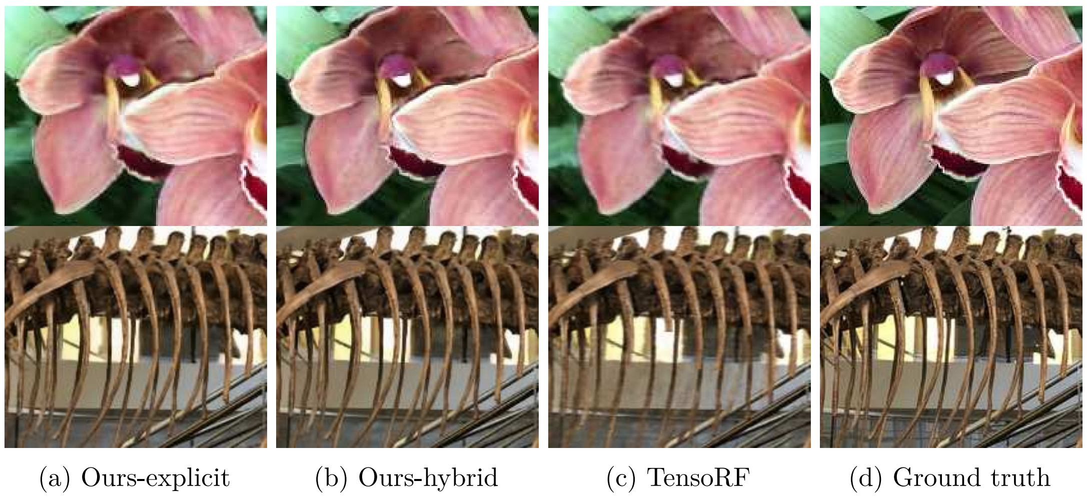


Figure 8.6 Zoomed qualitative results on static LLFF scenes (Mildenhall, Srinivasan, Cayon, et al., 2019). Visual comparison of k-planes, TensoRF (A. Chen et al., 2022), and the ground truth, on *orchids* (top) and *T-rex* (bottom).

Table 8.5 *1st section:* average over 8 synthetic static scenes from Mildenhall, Srinivasan, Tancik, et al. (2020); *2nd section:* average over 8 real forward-facing static scenes from Mildenhall, Srinivasan, Cayon, et al. (2019); *3rd section:* average over 8 synthetic "teleporting camera" dynamic scenes from Pumarola et al. (2021); *4th section:* average over 6 real multiview forward-facing dynamic scenes from T. Li et al. (2022); *5th section:* average over 3 scenes from Jin et al. (2021); MS-SSIM (multiscale structural similarity) is used instead of SSIM. K-planes timings are based on one NVIDIA A30 GPU, see the full paper (Fridovich-Keil, Meanti, et al., 2023) for per-scene results.

	PSNR ↑	SSIM ↑	Train Time ↓	# Params ↓
NeRF (static, synthetic)				
Ours-explicit	33.13	0.964	38 min	34M
Ours-hybrid	33.62	0.967	38 min	34M
Plenoxels	31.71	0.958	11 min	≈500M
TensoRF	33.14	0.963	17 min	18M
I-NGP	33.18	-	5 min	≈16M
LLFF (Mildenhall, Srinivasan, Cayon, et al., 2019) (static, real)				
Ours-explicit	26.78	0.841	33 min	19M
Ours-hybrid	26.92	0.847	33 min	19M
Plenoxels	26.29	0.839	24 min	≈500M
TensoRF	26.73	0.839	25 min	45M
D-NeRF (Pumarola et al., 2021) (dynamic, synthetic)				
Ours-explicit	30.39	0.96	52 min	37M
Ours-hybrid	30.84	0.96	52 min	37M
D-NeRF	29.67	0.95	48 hrs	1-3M
TiNeuVox	32.67	0.97	30 min	≈12M
V4D	33.72	0.98	4.9 hrs	275M
DyNeRF (T. Li et al., 2022) (dynamic, real)				
Ours-explicit	30.88	0.960	3.7 hrs	51M
Ours-hybrid	31.63	0.964	1.8 hrs	27M
DyNeRF	[2]29.58	-	1344 hrs	7M
LLFF	[2]23.24	-	-	-
MixVoxels-L (F. Wang et al., 2022)	30.80	0.960	1.3 hrs	125M
Phototourism (Jin et al., 2021) (variable appearance)				
Ours-explicit	22.25	0.859	35 min	36M
Ours-hybrid	22.92	0.877	35 min	36M
NeRF-W	27.00	0.962	384 hrs	∼2M
NeRF-W (public)	19.70	0.764	164 hrs	∼2M
LearnIt	19.26	-	-	-

[1] TiNeuVox uses half-resolution images for training and evaluation (V4D code is not yet public, so their resolution is unknown). [2] DyNeRF (T. Li et al., 2022) and LLFF (Mildenhall, Srinivasan, Cayon, et al., 2019) only report metrics on the *flame salmon* video; average performance may be higher as this is one of the more challenging videos. [3] Open-source version of Nerf-W (Martin-Brualla et al., 2021) https://github.com/kwea123/nerf_pl/tree/nerfw where we re-implement test-time optimization as for k-planes.

paper (Fridovich-Keil, Meanti, et al., 2023) for ablation studies over these hyperparameters. The explicit version of our model matches the prior state-of-the-art in terms of quality metrics, while the hybrid version achieves slightly higher quality metrics. Figure 8.4 shows zoomed-in visual results on a small sampling of scenes.

We also present results of our triplane model on the unbounded, forward-facing, real scenes from LLFF (Mildenhall, Srinivasan, Cayon, et al., 2019). Our results on this dataset are similar to the synthetic static scenes; both versions of our model match or exceed the prior state-of-the-art, with the hybrid version achieving slightly higher metrics than the fully explicit version. Figure 8.6 shows zoomed-in visual results on a small sampling of scenes.

8.4.2 Dynamic scenes

We evaluate our hexplane model on two dynamic scene datasets: a set of synthetic, bounded, 360°, monocular videos from D-NeRF (Pumarola et al., 2021) and a set of real, unbounded, forward-facing, multiview videos from DyNeRF (T. Li et al., 2022).

The D-NeRF dataset contains eight videos of varying duration, from 50 frames up to 200 frames per video. Each time step has a single training image from its own camera viewpoint; the camera "teleports" between adjacent timestamps (H. Gao et al., 2022). Standardized test views are from novel camera positions at a range of timestamps throughout the video. Both our explicit and hybrid models outperform D-NeRF in both quality metrics and training time (by a large margin), though they do not surpass very recent hybrid methods TiNeuVox (Fang et al., 2022) and V4D (Gan et al., 2022), as shown in Figure 8.7.

The DyNeRF dataset contains six 10-second videos recorded at 30 fps simultaneously on approximately 15-20 video cameras from a range of forward-facing view directions; the exact number of cameras varies per scene because a few cameras produced miscalibrated videos. A central camera is reserved for evaluation, and training uses frames from the remaining cameras. Both our methods again produce similar quality metrics to prior state-of-the-art, including a very recent hybrid method MixVoxels (F. Wang et al., 2022), with our hybrid method achieving higher quality metrics. Please see Figure 8.5 for a visual comparison.

Decomposing time and space

One neat consequence of our planar decomposition of time and space is that it naturally disentangles dynamic and static portions of the scene. The static-only part of the scene can be obtained by setting the three time planes to one (the multiplicative identity). Subtracting the static-only rendered image from the full rendering (*i.e.* with the time plane parameters not set to 1), we can reveal the dynamic part of the scene. Figure 8.8 shows this decomposition of time and space. This natural volumetric disentanglement of a scene into static and dynamic regions may enable many applications across augmented and virtual reality (Benaim et al., 2022).

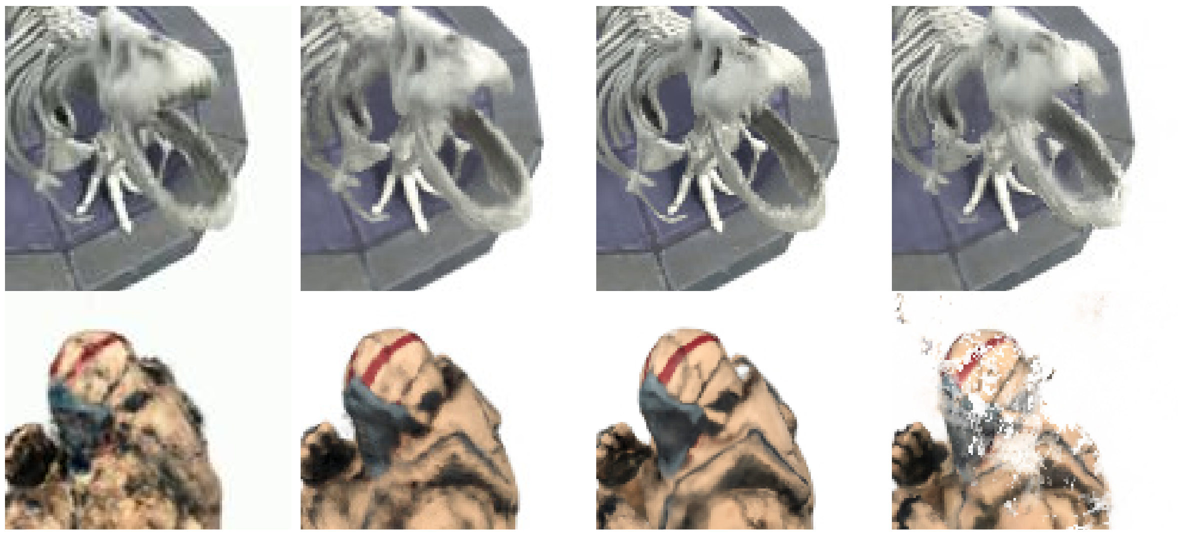


Figure 8.7 Zoomed qualitative results on monocular dynamic scenes from D-NeRF Pumarola et al., 2021. Visual comparison of k-planes, D-NeRF (Pumarola et al., 2021), TiNeuVox (Fang et al., 2022) and V4D (Gan et al., 2022), on *t-rex* (top) and *hook* (bottom).

We can also visualize the time planes to better understand where motion occurs in a video. Figure 8.9 shows the averaged features learned by the xt plane in our model for the *flame salmon* and *cut beef* DyNeRF videos, in which we can identify the motions of the hands in both space and time. The xt plane learns to be sparse, with most entries equal to the multiplicative identity, due to a combination of our sparse transients prior and the true sparsity of motion in the video. For example, in the upper portion of Figure 8.8 one of the cook's arms contains most of the motion, while in the *lower* figure both arms are moving. Having access to such an explicit representation of time allows us to add time-specific priors.

8.4.3 Variable appearance

Our variable appearance experiments use the Phototourism dataset (Jin et al., 2021), which includes photos of well-known landmarks taken by tourists with arbitrary view directions, lighting conditions, and transient occluders, mostly other tourists. Our experimental conditions parallel those of NeRF-W (Martin-Brualla et al., 2021): we train on more than a thousand tourist photographs and test on a standard set that is free of transient occluders. Like NeRF-W, we evaluate on test images by optimizing our per-image appearance feature on the left half of the image and computing metrics on the right half. Results are visualized in Figure 8.10.

Also similar to NeRF-W (Martin-Brualla et al., 2021; Bojanowski et al., 2017), we can interpolate in the appearance code space. Since only the color decoder (and not the density decoder) takes the appearance code as input, our approach is guaranteed not to change the geometry, regardless of whether we use our explicit or our hybrid model. Figure 8.11 shows

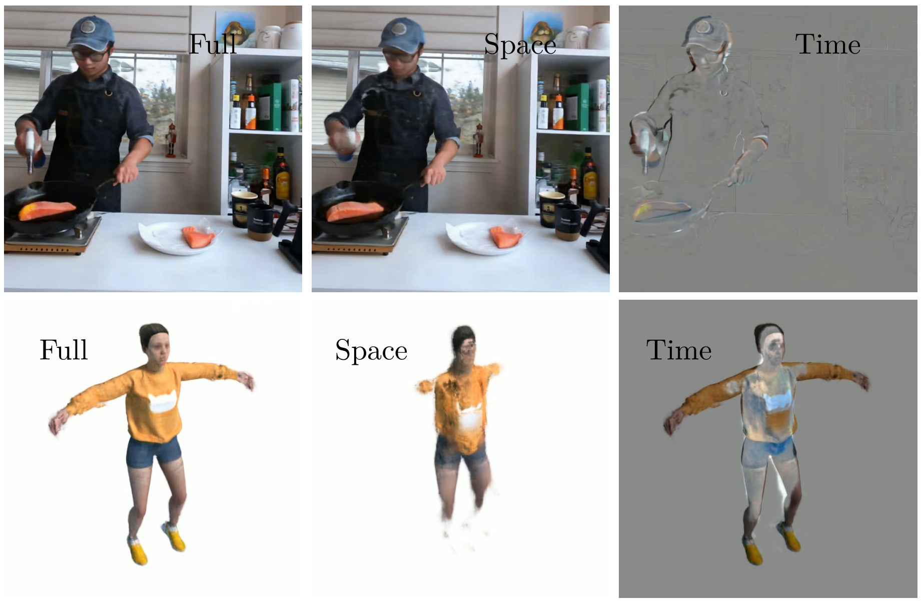


Figure 8.8 Decomposition of space and time. *K*-planes (left) naturally decomposes a 3D video into static and dynamic components. We render the static part (middle) by setting the time planes to the identity, and the remainder (right) is the dynamic part. Top shows the *flame salmon* multiview video (T. Li et al., 2022) and bottom shows the *jumping jacks* monocular video (Pumarola et al., 2021).

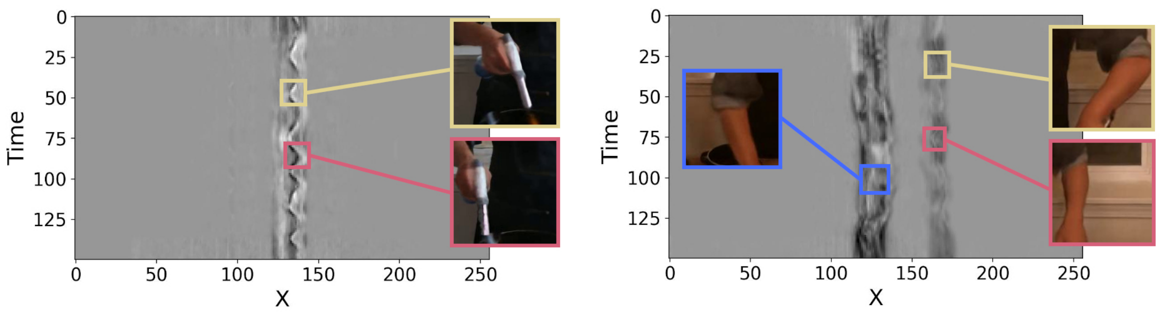


Figure 8.9 Visualization of a time plane. The xt plane highlights the dynamic regions in the scene. The wiggly patterns across time correspond to the motion of the person's hands and cooking tools, in the *flame salmon* scene (top) where only one hand moves and the *cut beef* scene (bottom) where both hands move.

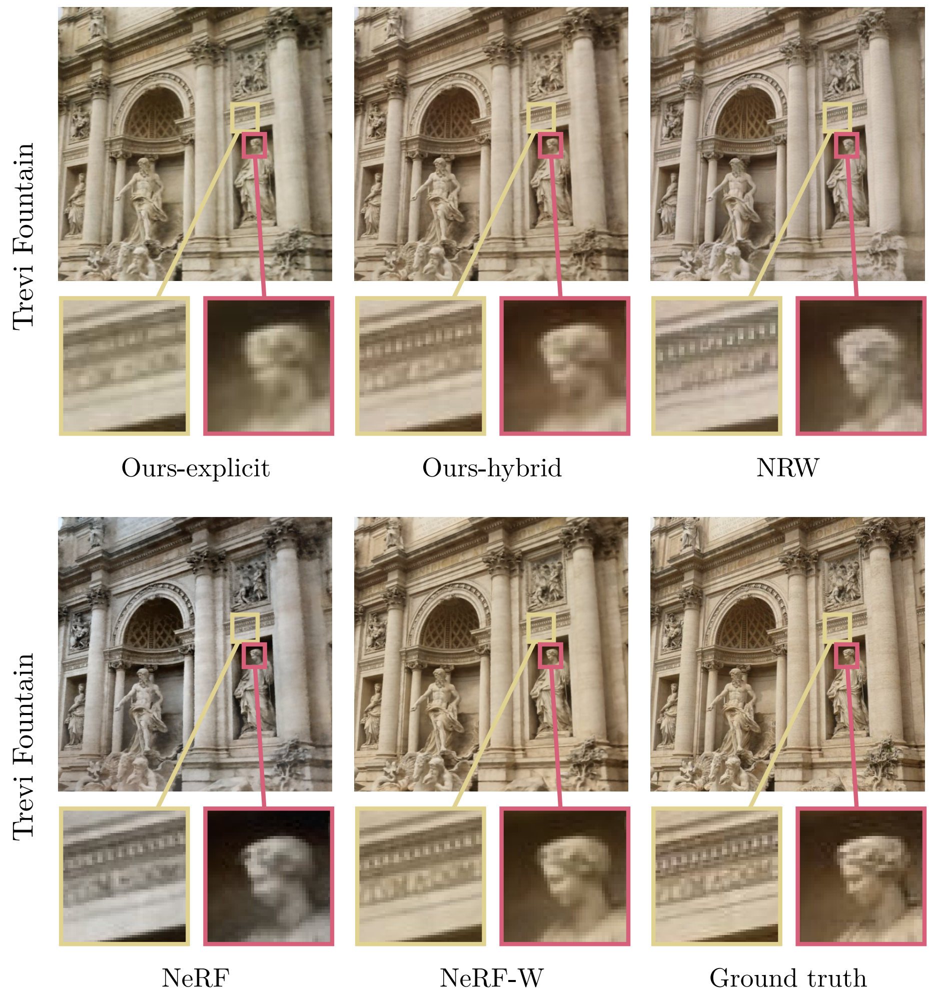


Figure 8.10 Qualitative results from Phototourism dataset. We compare our model with strong baselines including NRW (Meshry et al., 2019). Our method captures the geometry and appearance of the scene, but produces slightly lower resolution results than NeRF-W. Note that our model optimizes in just 35 minutes on a single GPU compared to NeRF-W, which takes 2 days on 8 GPUs.

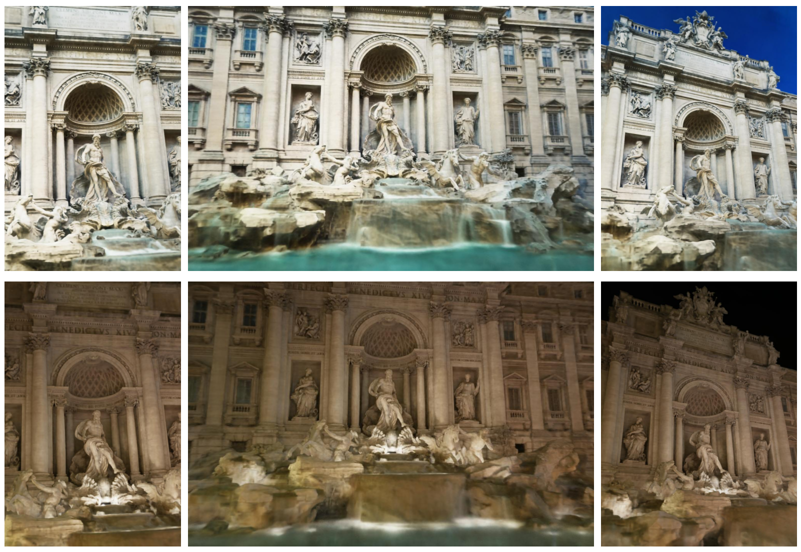


Figure 8.11 Appearance interpolation. Like NeRF-W (Martin-Brualla et al., 2021), we can interpolate our appearance code to alter the visual appearance of landmarks. We show three test views from the *Trevi fountain* with appearance codes corresponding to day and night.

that our planar decomposition with a 32-dimensional appearance code is sufficient to accurately capture global appearance changes in the scene.

8.5 Conclusions

We introduced a simple yet versatile method to decompose a d-dimensional space into $\binom{d}{2}$ planes, which can be optimized directly from indirect measurements and scales gracefully in model size and optimization time with increasing dimension, without any custom CUDA kernels. We demonstrated that the proposed k-planes decomposition applies naturally to reconstruction of static 3D scenes as well as dynamic 4D videos, and with the addition of a global appearance code can also extend to the more challenging task of unconstrained scene reconstruction. K-planes is the first explicit, simple model to demonstrate competitive performance across such varied tasks.

Chapter 9

Conclusions

From the first chapter of this thesis, we set out to demonstrate how kernel methods and other "shallow learning" algorithms could be adapted to better handle modern problems. There are two main obstacles, which must be overcome: slow performance with larger datasets (an old problem with kernel methods which is tackled in depth in Chapter 3) and the prevalence of kernel functions which are too simple to provide precise data fits, having very few kernel-specific parameters. In Chapters 3 and 4, we proposed some possible solutions to these hurdles, and validated them experimentally. This showed that it is possible to translate abstract algorithms into practical implementations, and the computational efficiency of the algorithms can translate into high performance on real hardware, although the two require a different approaches and optimizations. The result of our optimizations is a kernel solver with a great accuracy/efficiency tradeoff. Furthermore, by allowing an increasing number of hyperparameters to be trained with gradient descent, the models themselves can become much more compact, without any drop in accuracy, but with increased efficiency when the model needs to be evaluated (*i.e.* at inference time). Having a method for automatically setting model hyperparameters allows the complexity of kernel functions to be increased without fear. The code used for all experiments in these first two chapters is available as part of the open-source Falkon library with the aim of both encouraging further research on large-scale kernel methods, and facilitating the use of fast kernel methods by practitioners. We believe that future research in the same direction as our work could focus on the following: (a) the use of loss functions other than the squared or logistic – for example the hinge loss enjoys a preferred status in the kernel methods community and imposes useful biases. (b) Development of alternative optimization algorithms, in order to adapt to those losses which are harder to optimize for, or based on stochastic optimization which only looks at a few training samples at every iteration. (c) Extending the use of kernels with richer parameterizations, for which some initial steps have been made in Chapter 4, where we only scratch the surface of what can be accomplished with structured kernels applied to appropriate data-types.

In Chapter 5 we developed a very general framework in which, under the hard-margin condition, it is possible to show that many different kinds of models, using commonly adopted surrogate losses, have exponentially decreasing classification error in the multiclass setting. While we have considered two (the hard-margin and low-noise) conditions on the class separation due to their theoretical tractability, an interesting extension would be to consider other, more realistic, noise models: for example allowing some non-vanishing fraction of the samples to be mislabeled.

In Part II of this thesis we have investigated how kernel methods could be applied to a diverse set of tasks stemming from real-world problems. Firstly in Chapter 6 we focused on the ability of a learning system to rapidly adapt to the changing environment by *retraining* when new information arises. The setting is that of robot learning, where the robot is tasked with recognizing objects within its field of view. Of course, in a real-world setting it would be impractical to have a pre-trained model which can recognize every possible object, due to its size, and to the possibility of encountering an object absent from the training set; hence the need for a model which can be retrained with new data whenever it encounters a new object. In this chapter we showed that retraining the deep neural network which is used for interpreting visual signals is very computationally demanding, while instead it is possible to solve the same problem by only retraining a kernel machine, placed top of the features extracted from the pre-trained vision model. Such retraining is indeed much faster and in practice we demonstrate how it can be used by the robot to learn how to recognize new objects.

In Chapter 7 we turned to the task of forecasting wind speed in the near future (up to 24 hours ahead). The adopted approach was purely data-driven, by training kernel machines on physical variables measured in the hours preceding the forecast. To analyze the impact of different variables on the problem, a huge number of models were trained: we quantified the role of different inputs such as wind direction, and that of the amount of *memory* on which each prediction is based on. These variables all have strong physical interpretations which are described in the chapter. Furthermore, we compared kernel-based models against other algorithms proposed in the literature for the same task, and discovered that even deep learning architectures do not provide an advantage from the point of view of neither forecasting accuracy, nor computational efficiency. Of course, a purely data-driven approach neglects the fundamental physical structure of the problem: future work should focus on the development of so called *hybrid* models, which integrate mechanistic weather simulations with a data-driven approach.

Finally, in Chapter 8, shallow learning models are used for novel-view synthesis. We proposed an efficient factorization of 3D space, which can be trivially extended to 4D environments (where the fourth dimension represents time), and more. The proposed model explicitly stores learned feature vectors such that, given a query point in space-time, one can obtain a high-dimensional representation of such point by simple lookups, multiplications and concatenations – *i.e.* without querying a deep network. A small non-linear model is then used to interpret such feature vectors as RGB color and material density, which are then coupled with a rendering formula to generate images from arbitrary view-points. We demonstrated that the proposed k-planes decomposition applies to both 3D scenes and dynamic 4D videos, and also extends to scenes for which variable-appearance training images are given. Furthermore, by running experiments on several different datasets we showed that our model reaches a good trade-off between accuracy and training speed; however, it remains slower to train than other models proposed in the literature which make use of custom CUDA kernels. Hence one could focus on further improving the efficiency of k-planes.

References

Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* URL: https://www.tensorflow.org/.

Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, G. A., S. Hammarling, A. McKenney, and D. Sorensen (1999). *LAPACK Users' Guide.* Third. Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9780898719604.

Anzt, H., S. Tomov, P. Luszczek, W. Sawyer, and J. Dongarra (2015). "Acceleration of GPU-based Krylov solvers via data transfer reduction". In: *International Journal of High Performance Computing Applications* 29.3, pp. 366–383. DOI: 10.1177/1094342015580139.

Araya, I. A., C. Valle, and H. Allende (2020). "A Multi-Scale Model based on the Long Short-Term Memory for day ahead hourly wind speed forecasting". In: *Pattern Recognition Letters* 136, pp. 333–340. DOI: 10.1016/j.patrec.2019.10.011.

Arlot, S. (2007). "Resampling and Model selection". PhD Thesis. University Paris-Sud (Orsay).

Arlot, S. and F. Bach (2009). "Data-driven calibration of linear estimators with minimal penalties". In: *NeurIPS 22.*

Aronszajn, N. (1950). "Theory of Reproducing Kernels". In: *Transactions of the American Mathematical Society* 68.3, pp. 337–404. DOI: 10.1090/S0002-9947-1950-0051437-7.

Audibert, J.-Y. and A. B. Tsybakov (2007). "Fast learning rates for plug-in classifiers". In: *The Annals of Statistics* 35.2, pp. 608–633. DOI: 10.1214/009053606000001217.

Avron, H., K. L. Clarkson, and D. P. Woodruff (2017). "Faster Kernel Ridge Regression Using Sketching and Preconditioning". In: *SIAM Journal on Matrix Analysis and Applications* 38.4, pp. 1116–1138. DOI: 10.1137/16M1105396.

Bach, F. (2013). "Sharp analysis of low-rank kernel matrix approximations". In: *COLT 26.*

Bai, M. and R. Urtasun (2017). "Deep watershed transform for instance segmentation". In: *CVPR 2017*, pp. 5221–5229. DOI: 10.1109/CVPR.2017.305.

Barla, A., F. Odone, and A. Verri (2003). "Histogram intersection kernel for image classification". In: *Proceedings 2003 International Conference on Image Processing.* Vol. 3. DOI: 10.1109/ICIP.2003.1247294.

Barron, J. T., B. Mildenhall, M. Tancik, P. Hedman, R. Martin-Brualla, and P. P. Srinivasan (2021). "Mip-NeRF: A Multiscale Representation for Anti-Aliasing Neural Radiance Fields". In: *ICCV 2021*, pp. 5835–5844. DOI: 10.1109/ICCV48922.2021.00580.

Barron, J. T., B. Mildenhall, D. Verbin, P. P. Srinivasan, and P. Hedman (2022). "Mip-NeRF 360: Unbounded Anti-Aliased Neural Radiance Fields". In: *CVPR 2022*, pp. 5460–5469. DOI: 10.1109/CVPR52688.2022.00539.

Bartlett, P. L., M. I. Jordan, and J. D. McAuliffe (2006). "Convexity, Classification, and Risk Bounds". In: *Journal of the American Statistical Association* 101.473, pp. 138–156. DOI: 10.1198/016214505000000907.

Bartlett, P. L., S. Boucheron, and G. Lugosi (2002). “Model Selection and Error Estimation”. In: *Machine Learning* 48. DOI: 10.2139/ssrn.248567.

Ben-Israel, A. and T. N. E. Greville (2003). *Generalized Inverses: Theory and Applications*. Second. CMS Books in Mathematics. Springer.

Benaim, S., F. Warburg, P. E. Christensen, and S. Belongie (2022). *Volumetric Disentanglement for 3D Scene Manipulation*. arXiv: 2206.02776 `[cs.CV]`.

Bergstra, J. and Y. Bengio (2012). “Random Search for Hyper-Parameter Optimization”. In: *J. Mach. Learn. Res.* 13, pp. 281–305.

Bivona, S., G. Bonanno, R. Burlon, D. Gurrera, and C. Leone (2011). “Stochastic models for wind speed forecasting”. In: *Energy Conversion and Management* 52.2, pp. 1157–1165. DOI: 10.1016/j.enconman.2010.09.010.

Bochner, S. (1959). *Lectures on Fourier Integrals*. Princeton University Press.

Bojanowski, P., A. Joulin, D. Lopez-Paz, and A. Szlam (2017). *Optimizing the Latent Space of Generative Networks*. arXiv: 1707.05776.

Bolya, D., C. Zhou, F. Xiao, and Y. J. Lee (2019). “YOLACT: Real-time instance segmentation”. In: *CVPR 2019*, pp. 9157–9166. DOI: 10.1109/ICCV.2019.00925.

Borgwardt, K. and H. Kriegel (2005). “Shortest-path kernels on graphs”. In: *Fifth IEEE International Conference on Data Mining (ICDM) 2005*. DOI: 10.1109/ICDM.2005.132.

Boyd, S. and L. Vandenberghe (2004). *Convex Optimization*. Cambridge University Press. DOI: 10.1145/2020408.2020410.

BP p.l.c. (2022). *bp Statistical Review of World Energy*. URL: https://www.bp.com/en/global/corporate/energy-economics/statistical-review-of-world-energy.html (visited on 02/10/2023).

Brochu, E., V. M. Cora, and N. d. Freitas (2010). *A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning*. arXiv: 1012.2599.

Burt, D. R., C. E. Rasmussen, and M. v. d. Wilk (2020). “Convergence of Sparse Variational Inference in Gaussian Processes Regression”. In: *JMLR* 21, pp. 1–63.

Cabannes, V. A., F. Bach, and A. Rudi (2021). “Fast Rates for Structured Prediction”. In: *COLT 2021*. Vol. 134, pp. 823–865.

Calandriello, D., L. Carratino, A. Lazaric, M. Valko, and L. Rosasco (2019). “Gaussian process optimization with adaptive sketching: Scalable and no regret”. In: *Conference on Learning Theory*, pp. 533–557.

Calandriello, D., L. Carratino, A. Lazaric, M. Valko, and L. Rosasco (2020). “Near-Linear Time Gaussian Process Optimization with Adaptive Batching and Resparsification”. In: *ICML 37*.

Calandriello, D. and L. Rosasco (2018). “Statistical and Computational Trade-Offs in Kernel K-Means”. In: *NeurIPS 31*.

Calli, B., A. Singh, A. Walsman, S. Srinivasa, P. Abbeel, and A. M. Dollar (2015). “The YCB object and model set: Towards common benchmarks for manipulation research”. In: *ICAR 2015*, pp. 510–517. DOI: 10.1109/ICAR.2015.7251504.

Camelo, H. d. N., P. S. Lucio, J. B. V. Leal Junior, P. C. M. d. Carvalho, and D. v. G. d. Santos (2018). “Innovative hybrid models for forecasting time series applied in wind generation based on the combination of time series models with artificial neural networks”. In: *Energy* 151, pp. 347–357. DOI: 10.1016/j.energy.2018.03.077.

Camoriano, R., T. Angles, A. Rudi, and L. Rosasco (2016). “NYTRO: When Subsampling Meets Early Stopping”. In: *AISTATS 19*, pp. 1403–1411.

Camoriano, R., G. Pasquale, C. Ciliberto, L. Natale, L. Rosasco, and G. Metta (2017). “Incremental robot learning of new objects with fixed update time”. In: *ICRA 2017*, pp. 3207–3214. DOI: 10.1109/ICRA.2017.7989364.

Camps-Valls, G., L. Gomez-Chova, J. Muñoz-Marí, J. Vila-Francés, and J. Calpe-Maravilla (2006). “Composite kernels for hyperspectral image classification”. In: *IEEE geoscience and remote sensing letters* 3.1, pp. 93–97. DOI: 10.1109/LGRS.2005.857031.

Cao, Y. and Y. Golubev (2006). “On oracle inequalities related to smoothing splines”. In: *Mathematical Methods of Statistic* 15.4.

Caponnetto, A. and E. De Vito (2007). “Optimal Rates for the Regularized Least-Squares Algorithm”. In: *Foundations of Computational Mathematics* 7, pp. 331–368. DOI: 10.1007/s10208-006-0196-8.

Carpinone, A., M. Giorgio, R. Langella, and A. Testa (2015). “Markov chain modeling for very-short-term wind power forecasting”. In: *Electric Power Systems Research* 122, pp. 152–158. DOI: 10.1016/j.epsr.2014.12.025.

Carratino, L., S. Vigogna, D. Calandriello, and L. Rosasco (2021). “ParK: Sound and Efficient Kernel Ridge Regression by Feature Space Partitions”. In: *NeurIPS 34*. Vol. 34, pp. 6430–6441.

Catanzaro, B., N. Sundaram, and K. Keutzer (2008). “Fast Support Vector Machine Training and Classification on Graphics Processors”. In: *ICML 25*. DOI: 10.1145/1390156.1390170.

Cawley, G. C. and N. L. C. Talbot (2004). “Fast exact leave-one-out cross-validation of sparse least-squares support vector machines”. In: *Neural Networks* 17.10, pp. 1467–1475. DOI: 10.1016/j.neunet.2004.07.002.

Ceola, F., E. Maiettini, G. Pasquale, G. Meanti, L. Rosasco, and L. Natale (2022). “Learn Fast, Segment Well: Fast Object Segmentation Learning on the iCub Robot”. In: *IEEE Transactions on Robotics* 38.5, pp. 3154–3172. DOI: 10.1109/TRO.2022.3164331.

Ceola, F., E. Maiettini, G. Pasquale, L. Rosasco, and L. Natale (2021). “Fast Object Segmentation Learning with Kernel-based Methods for Robotics”. In: *ICRA 2021*, pp. 13581–13588. DOI: 10.1109/ICRA48506.2021.9561758.

Ceola, F., E. Maiettini, G. Pasquale, L. Rosasco, and L. Natale (2020). *Fast region proposal learning for object detection for robotics*. arXiv: 2011.12790.

Chan, E. R., C. Z. Lin, M. A. Chan, K. Nagano, B. Pan, S. D. Mello, O. Gallo, L. J. Guibas, J. Tremblay, S. Khamis, T. Karras, and G. Wetzstein (2022). “Efficient Geometry-aware 3D Generative Adversarial Networks”. In: *CVPR 2022*, pp. 16102–16112. DOI: 10.1109/CVPR52688.2022.01565.

Charlier, B., J. Feydy, J. A. Glaunès, F.-D. Collin, and G. Durif (2021). “Kernel Operations on the GPU, with Autodiff, without Memory Overflows”. In: *Journal of Machine Learning Research* 22.74, pp. 1–6.

Charlier, B., J. Feydy, J. A. Glaunès, and G. Durif (2020). *KeOps*. Version 1.4. URL: https://github.com/getkeops/keops.

Chen, A., Z. Xu, A. Geiger, J. Yu, and H. Su (2022). “TensoRF: Tensorial Radiance Fields”. In: *ECCV 2022*. DOI: 10.1007/978-3-031-19824-3_20.

Chen, H., K. Sun, Z. Tian, C. Shen, Y. Huang, and Y. Yan (2020). “BlendMask: Top-down meets bottom-up for instance segmentation”. In: *CVPR 2020*, pp. 8573–8581. DOI: 10.1109/CVPR42600.2020.00860.

Chen, J., H. Avron, and V. Sindhwani (2017). “Hierarchically Compositional Kernels for Scalable Nonparametric Learning”. In: *JMLR* 18.1, pp. 2214–2255.

Chen, D.-R. and T. Sun (2006). “Consistency of Multiclass Empirical Risk Minimization Methods Based on Convex Loss”. In: *Journal of Machine Learning Research* 7.86, pp. 2435–2447.

Chen, X., R. Girshick, K. He, and P. Dollár (2019). “TensorMask: A foundation for dense object segmentation”. In: *CVPR 2019*, pp. 2061–2069. DOI: 10.1109/ICCV.2019.00215.

Cheng, C., A. Sa-Ngasoongsong, O. Beyca, T. Le, H. Yang, Z. Kong, and S. T. Bukkapatnam (2015). “Time series forecasting for nonlinear and non-stationary processes: a review and comparative study". In: *IIE Transactions* 47.10, pp. 1053–1071. DOI: 10.1080/0740817X.2014.999180.

Chitsazan, M. A., M. S. Fadali, A. K. Nelson, and A. M. Trzynadlowski (2017). "Wind speed forecasting using an echo state network with nonlinear output functions". In: *2017 American Control Conference (ACC)*, pp. 5306–5311. DOI: 10.23919/ACC.2017.7963779.

Chou, J.-S. and T.-K. Nguyen (2018). "Forward forecast of stock price using sliding-window metaheuristic-optimized machine-learning regression". In: *IEEE Transactions on Industrial Informatics* 14.7, pp. 3132–3142. DOI: 10.1109/TII.2018.2794389.

Cucker, F. and S. Smale (2002). "On the mathematical foundations of learning". In: *Bulletin of the American mathematical society* 39.1, pp. 1–49. DOI: 10.1090/S0273-0979-01-00923-5.

Cutajar, K., E. V. Bonilla, P. Michiardi, and M. Filippone (2017). "Random Feature Expansions for Deep Gaussian Processes". In: *ICML 34*.

Cutajar, K., M. Osborne, J. Cunningham, and M. Filippone (2016). "Preconditioning Kernel Matrices". In: *ICML 33*.

Cuturi, M. and K. Fukumizu (2006). "Kernels on Structured Objects Through Nested Histograms". In: *NeurIPS 2006*. Vol. 19.

Dai, B., B. Xie, N. He, Y. Liang, A. Raj, M.-F. Balcan, and L. Song (2014). "Scalable Kernel Methods via Doubly Stochastic Gradients". In: *NeurIPS 27*.

Dai, J., K. He, Y. Li, S. Ren, and J. Sun (2016). "Instance-sensitive fully convolutional networks". In: *ECCV 2016*, pp. 534–549. DOI: 10.1007/978-3-319-46466-4_32.

Dai, J., Y. Li, K. He, and J. Sun (2016). "R-FCN: Object Detection via Region-based Fully Convolutional Networks". In: *NeurIPS 2016*, pp. 379–387.

Damianou, A. and N. Lawrence (2013). "Deep Gaussian processes". In: *AISTATS 16*.

Danielczuk, M., M. Matl, S. Gupta, A. Li, A. Lee, J. Mahler, and K. Goldberg (2019). "Segmenting unknown 3D objects from real depth images using Mask R-CNN trained on synthetic data". In: *ICRA 2019*, pp. 7283–7290. DOI: 10.1109/ICRA.2019.8793744.

Della Vecchia, A., J. Mourtada, E. De Vito, and L. Rosasco (2021). "Regularized ERM on random subspaces". In: *AISTATS 24*. Vol. 130, pp. 4006–4014.

Du, Y., Y. Zhang, H.-X. Yu, J. B. Tenenbaum, and J. Wu (2021). "Neural Radiance Flow for 4D View Synthesis and Video Processing". In: *ICCV 2021*, pp. 14304–14314. DOI: 10.1109/ICCV48922.2021.01406.

Efron, B. (2004). "The estimation of prediction error: covariance penalties and cross-validation". In: *Journal of the American Statistical Association* 99.467, pp. 619–632. DOI: 10.1198/016214504000000692.

Eitel, A., N. Hauff, and W. Burgard (2019). "Self-supervised transfer learning for instance segmentation through physical interaction". In: *IROS 2019*, pp. 4020–4026. DOI: 10.1109/IROS40897.2019.8967915.

El Alaoui, A. and M. W. Mahoney (2015). "Fast randomized kernel methods with statistical guarantees". In: *NeurIPS 28*.

Elsken, T., J. H. Metzen, and F. Hutter (2019). "Neural architecture search: A survey". In: *JMLR* 20.1, pp. 1997–2017.

Everingham, M., L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman (2010). "The Pascal Visual Object Classes (VOC) Challenge". In: *International Journal of Computer Vision* 88.2, pp. 303–338. DOI: 10.1007/s11263-009-0275-4.

Fang, J., T. Yi, X. Wang, L. Xie, X. Zhang, W. Liu, M. Nies $\rightarrow$ sner, and Q. Tian (2022). "Fast Dynamic Radiance Fields with Time-Aware Neural Voxels". In: *SIGGRAPH Asia 2022*. DOI: 10.1145/3550469.3555383.

Feldman, V., V. Guruswami, P. Raghavendra, and Y. Wu (2012). "Agnostic learning of monomials by halfspaces is hard". In: *SIAM Journal on Computing* 41.6, pp. 1558–1590. DOI: 10.1109/FOCS.2009.26.

Franceschi, L., M. Donini, P. Frasconi, and M. Pontil (2017). "Forward and Reverse Gradient-Based Hyperparameter Optimization". In: *ICML 34*.

Fridovich-Keil, S., G. Meanti, F. R. Warburg, B. Recht, and A. Kanazawa (2023). *K-Planes for Radiance Fields in Space, Time, and Appearance*. arXiv: 2301.10241 [cs.CV].

Fridovich-Keil, S., A. Yu, M. Tancik, Q. Chen, B. Recht, and A. Kanazawa (2022). "Plenoxels: Radiance Fields without Neural Networks". In: *CVPR 2022*. IEEE, pp. 5491–5500. DOI: 10.1109/CVPR52688.2022.00542.

Fu, W., K. Wang, J. Tan, and K. Zhang (2020). "A composite framework coupling multiple feature selection, compound prediction models and novel hybrid swarm optimizer-based synchronization optimization strategy for multi-step ahead short-term wind speed forecasting". In: *Energy Conversion and Management* 205, p. 112461. DOI: 10.1016/j.enconman.2019.112461.

Gan, W., H. Xu, Y. Huang, S. Chen, and N. Yokoya (2022). *V4D: Voxel for 4D Novel View Synthesis*. arXiv: 2205.14332 [cs.CV].

Gao, C., A. Saraf, J. Kopf, and J.-B. Huang (2021). "Dynamic View Synthesis From Dynamic Monocular Video". In: *ICCV 2021*, pp. 5712–5721. DOI: 10.1109/ICCV48922.2021.00566.

Gao, H., R. Li, S. Tulsiani, B. Russell, and A. Kanazawa (2022). "Monocular Dynamic View Synthesis: A Reality Check". In: *NeurIPS 2022*.

Gao, N., Y. Shan, Y. Wang, X. Zhao, Y. Yu, M. Yang, and K. Huang (2019). "SSAP: Single-shot instance segmentation with affinity pyramid". In: *CVPR 2019*, pp. 642–651. DOI: 10.1109/ICCV.2019.00073.

Gardner, J. R., G. Pleiss, D. Bindel, K. Q. Weinberger, and A. G. Wilson (2018). "GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration". In: *NeurIPS 31*.

Gardner, J. R., G. Pleiss, R. Wu, K. Q. Weinberger, and A. G. Wilson (2018). "Product Kernel Interpolation for Scalable Gaussian Processes". In: *AISTATS 21*, pp. 1407–1416.

Garreau, D., W. Jitkrittum, and M. Kanagawa (2017). *Large sample analysis of the median heuristic*. arXiv: 1707.07269.

Girshick, R., J. Donahue, T. Darrell, and J. Malik (2014). "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *CVPR 2014*. DOI: 10.1109/CVPR.2014.81.

Gittens, A. and M. W. Mahoney (2016). "Revisiting the Nyström Method for Improved Large-Scale Machine Learning". In: *Journal of Machine Learning Research* 17, pp. 3977–4041.

Golub, G. H., M. Heath, and G. Wahba (1979). "Generalized Cross-Validation as a Method for Choosing a Good Ridge Parameter". In: *Technometrics* 21.2, pp. 215–223. DOI: 10.1080/00401706.1979.10489751.

Gonen, A., F. Orabona, and S. Shalev-Shwartz (2016). "Solving Ridge Regression using Sketched Preconditioned SVRG". In: *ICML 33*, pp. 1397–1405.

Grazzi, R., L. Franceschi, M. Pontil, and S. Salzo (2020). "On the Iteration Complexity of Hypergradient Computation". In: *ICML 37*.

Guo, X., G. Chen, Y. Dai, X. Ye, J. Sun, X. Tan, and E. Ding (2022). "Neural Deformable Voxel Grid for Fast Optimization of Dynamic View Synthesis". In: *Proceedings of the Asian Conference on Computer Vision (ACCV) 2022*.

Hampali, S., M. Rad, M. Oberweger, and V. Lepetit (2020). "HOnnotate: A method for 3D annotation of hand and object poses". In: *CVPR 2020*, pp. 3196–3206. DOI: 10.1109/CVPR42600.2020.00326.

Hastie, T., R. Tibshirani, and J. Friedman (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Second. Springer Series in Statistics. Springer.

Haworth, J., J. Shawe-Taylor, T. Cheng, and J. Wang (2014). "Local online kernel ridge regression for forecasting of urban travel times". In: *Transportation research part C: emerging technologies* 46, pp. 151–178. DOI: 10.1016/j.trc.2014.05.015.

He, K., G. Gkioxari, P. Dollár, and R. B. Girshick (2017). "Mask R-CNN". In: *ICCV 2017*, pp. 2980–2988. DOI: 10.1109/TPAMI.2018.2844175.

He, K., X. Zhang, S. Ren, and J. Sun (2016). “Deep residual learning for image recognition”. In: *CVPR 2016*, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

Hensman, J., N. Durrande, and A. Solin (2017). “Variational Fourier Features for Gaussian Processes”. In: *Journal of Machine Learning Research* 18.1, pp. 5537–5588.

Hensman, J., N. Fusi, and N. D. Lawrence (2013). “Gaussian Processes for Big Data”. In: *UAI 2013*.

Hensman, J., A. G. Matthews, and Z. Ghahramani (2015). “Scalable variational Gaussian process classification”. In: *AISTATS 18*.

Herbrich, R. (2001). *Learning Kernel Classifiers: Theory and Algorithms*. First. The MIT Press. DOI: 10.7551/mitpress/4170.001.0001.

Hestenes, M. R. and E. Stiefel (1952). “Methods of conjugate gradients for solving linear systems”. In: *Journal of research of the National Bureau of Standards* 49, pp. 409–435. DOI: 10.6028/jres.049.044.

Hota, H., R. Handa, and A. Shrivas (2017). “Time series data prediction using sliding window based RBF neural network”. In: *International Journal of Computational Intelligence Research* 13.5, pp. 1145–1156.

Huang, Z., L. Huang, Y. Gong, C. Huang, and X. Wang (2019). “Mask Scoring R-CNN”. In: *CVPR 2019*, pp. 6409–6418. DOI: 10.1109/CVPR.2019.00657.

Hutchinson, M. F. (1990). “A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines”. In: *Communications in Statistics-Simulation and Computation* 19.2, pp. 433–450. DOI: 10.1080/03610919008812866.

Hutter, F., L. Kotthoff, and J. Vanschoren, eds. (2019). *Automated Machine Learning - Methods, Systems, Challenges*. The Springer Series on Challenges in Machine Learning. Springer.

Izmailov, P., A. Novikov, and D. Kropotov (2018). “Scalable Gaussian Processes with Billions of Inducing Inputs via Tensor Train Decomposition”. In: *AISTATS 21*.

Jin, Y., D. Mishkin, A. Mishchuk, J. Matas, P. Fua, K. M. Yi, and E. Trulls (2021). “Image Matching Across Wide Baselines: From Paper to Practice”. In: *Int. J. Comput. Vis.* 129.2, pp. 517–547. DOI: 10.1007/s11263-020-01385-0.

Kar, P. and H. Karnick (2012). “Random Feature Maps for Dot Product Kernels”. In: *AISTATS 15*, pp. 583–591.

Keerthi, S. S., V. Sindhwani, and O. Chapelle (2007). “An Efficient Method for Gradient-Based Adaptation of Hyperparameters in SVM Models”. In: *NeurIPS 19*.

Kirillov, A., E. Levinkov, B. Andres, B. Savchynskyy, and C. Rother (2017). “InstanceCut: from edges to instances with multicut”. In: *CVPR 2017*, pp. 5008–5017. DOI: 10.1109/CVPR.2017.774.

Koltchinskii, V. and O. Beznosova (2005). “Exponential Convergence Rates in Classification”. In: *COLT 2005*, pp. 295–307. DOI: 10.1007/11503415_20.

Kumar, S., M. Mohri, and A. Talwalkar (2012). “Sampling Methods for the Nyström Method”. In: *JMLR* 13, pp. 981–1006.

Kuo, W., A. Angelova, J. Malik, and T.-Y. Lin (2019). “ShapeMask: Learning to segment novel objects by refining shape priors”. In: *ICCV 2019*, pp. 9207–9216. DOI: 10.1109/ICCV.2019.00930.

Lagomarsino-Oneto, D., G. Meanti, N. Pagliana, A. Verri, A. Mazzino, L. Rosasco, and A. Seminara (2023). “Physics informed machine learning for wind speed prediction”. In: *Energy* 268. DOI: 10.1016/j.energy.2023.126628.

Lawal, A., S. Rehman, L. M. Alhems, and M. M. Alam (2021). “Wind Speed Prediction Using Hybrid 1D CNN and BLSTM Network”. In: *IEEE Access* 9, pp. 156672–156679. DOI: 10.1109/ACCESS.2021.3129883.

Le, Q., T. Sarlós, and A. Smola (2013). “Fastfood: Approximating Kernel Expansions in Loglinear Time”. In: *ICML 30*.

Li, A., M. Danielczuk, and K. Goldberg (2020). “One-Shot Shape-Based Amodal-to-Modal Instance Segmentation”. In: *16th International Conference on Automation Science and Engineering (CASE)*, pp. 1375–1382. DOI: 10.1109/CASE48305.2020.9216733.

Li, F., G. Ren, and J. Lee (2019). “Multi-step wind speed prediction based on turbulence intensity and hybrid deep neural networks”. In: *Energy Conversion and Management* 186, pp. 306–322. DOI: 10.1016/j.enconman.2019.02.045.

Li, F., C. Ionescu, and C. Sminchisescu (2010). “Random Fourier Approximations for Skewed Multiplicative Histogram Kernels”. In: *Pattern Recognition*, pp. 262–271. DOI: 10.1007/978-3-642-15986-2_27.

Li, S., J. Zhou, Z. Jia, D.-Y. Yeung, and M. T. Mason (2020). “Learning Accurate Objectness Instance Segmentation from Photorealistic Rendering for Robotic Manipulation”. In: *2018 International Symposium on Experimental Robotics*, pp. 245–255. DOI: 10.1007/978-3-030-33950-0_22.

Li, T., M. Slavcheva, M. Zollhoefer, S. Green, C. Lassner, C. Kim, T. Schmidt, S. Lovegrove, M. Goesele, R. Newcombe, and Z. Lv (2022). “Neural 3D Video Synthesis from Multi-view Video”. In: *CVPR 2022*, pp. 5511–5521. DOI: 10.1109/CVPR52688.2022.00544.

Li, Y., H. Shi, F. Han, Z. Duan, and H. Liu (2019). “Smart wind speed forecasting approach using various boosting algorithms, big multi-step forecasting strategy”. In: *Renewable Energy* 135, pp. 540–553. DOI: 10.1016/j.renene.2018.12.035.

Li, Z., S. Niklaus, N. Snavely, and O. Wang (2021). “Neural Scene Flow Fields for Space-Time View Synthesis of Dynamic Scenes”. In: *CVPR 2021*. DOI: 10.1109/CVPR46437.2021.00643.

Li, Z., J.-F. Ton, D. Oglic, and D. Sejdinovic (2019). “Towards a Unified Analysis of Random Fourier Features”. In: *ICML 36*.

Lima, A. R., A. J. Cannon, and W. W. Hsieh (2016). “Forecasting daily streamflow using online sequential extreme learning machines”. In: *Journal of hydrology* 537, pp. 431–443. DOI: 10.1016/j.jhydrol.2016.03.017.

Lin, T.-Y., P. Goyal, R. B. Girshick, K. He, and P. Dollár (2017). “Focal Loss for Dense Object Detection”. In: *ICCV 2017*, pp. 2999–3007. DOI: 10.1109/ICCV.2017.324.

Lin, T.-Y., M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick (2014). “Microsoft COCO: Common Objects in Context”. In: *ECCV 2014*. DOI: 10.1007/978-3-319-10602-1_48.

Lindemann, B., T. Müller, H. Vietz, N. Jazdi, and M. Weyrich (2021). “A survey on long short-term memory networks for time series prediction”. In: *Procedia CIRP* 99, pp. 650–655. DOI: 10.1016/j.procir.2021.03.088.

Liu, H., Y.-S. Ong, X. Shen, and J. Cai (2020). “When Gaussian Process Meets Big Data: A Review of Scalable GPs”. In: *IEEE Transactions on Neural Networks and Learning Systems* 31.11, pp. 4405–4423. DOI: 10.1109/TNNLS.2019.2957109.

Liu, H., X. Mi, Y. Li, Z. Duan, and Y. Xu (2019). “Smart wind speed deep learning based multi-step forecasting model using singular spectrum analysis, convolutional Gated Recurrent Unit network and Support Vector Regression”. In: *Renewable Energy* 143, pp. 842–854. DOI: 10.1016/j.renene.2019.05.039.

Liu, J.-W., Y.-P. Cao, W. Mao, W. Zhang, D. Junhao Zhang, J. Keppo, Y. Shan, X. Qie, and M. Zheng Shou (2022). *DeVRF: Fast Deformable Voxel Radiance Fields for Dynamic Scenes.* arXiv: 2205.15723 [cs.CV].

Liu, S., L. Qi, H. Qin, J. Shi, and J. Jia (2018). “Path aggregation network for instance segmentation”. In: *CVPR 2018*, pp. 8759–8768. DOI: 10.1109/CVPR.2018.00913.

Lodhi, H., C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins (2002). “Text Classification Using String Kernels”. In: *J. Mach. Learn. Res.* 2, pp. 419–444. DOI: 10.1162/153244302760200687.

Lombardi, S., T. Simon, J. Saragih, G. Schwartz, A. Lehrmann, and Y. Sheikh (2019). “Neural Volumes: Learning Dynamic Renderable Volumes from Images”. In: *ACM Trans. Graph.* 38.4. DOI: 10.1145/3306346.3323020.

Lorraine, J., P. Vicol, and D. Duvenaud (2020). “Optimizing Millions of Hyperparameters by Implicit Differentiation”. In: *AISTATS 23*.

Ltaief, H., S. Tomov, R. Nath, P. Du, and J. Dongarra (2011). “A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators”. In: *High Performance Computing for Computational Science*. DOI: 10.1007/978-3-642-19328-6_11.

Ma, S. and M. Belkin (2017). “Diving into the shallows: a computational perspective on large-scale shallow learning”. In: *NeurIPS 30*.

Ma, S. and M. Belkin (2019). “Kernel machines that adapt to GPUs for effective large batch training”. In: *MLSys 2019*.

Maclaurin, D., D. Duvenaud, and R. P. Adams (2015). “Gradient-Based Hyperparameter Optimization through Reversible Learning”. In: *ICML 32*.

Maiettini, E., G. Pasquale, L. Rosasco, and L. Natale (2018). “Speeding-up Object Detection Training for Robotics with FALKON”. In: *IROS 2018*. DOI: 10.1109/IROS.2018.8593990.

Maiettini, E., G. Pasquale, L. Rosasco, and L. Natale (2019). “On-line object detection: a robotics challenge”. In: *Autonomous Robots*. DOI: 10.1007/s10514-019-09894-9.

Mallows, C. L. (1973). “Some comments on C_p”. In: *Technometrics* 15.4, pp. 661–675. DOI: 10.2307/1267380.

Maltoni, D. and V. Lomonaco (2019). “Continuous learning in single-incremental-task scenarios”. In: *Neural Networks* 116, pp. 56–73. DOI: 10.1016/j.neunet.2019.03.010.

Mammen, E. and A. B. Tsybakov (1999). “Smooth Discrimination Analysis”. In: *The Annals of Statistics* 27.6, pp. 1808–1829. DOI: 10.1214/009053604000000869.

Marteau-Ferey, U., F. Bach, and A. Rudi (2019). “Globally Convergent Newton Methods for Ill-conditioned Generalized Self-concordant Losses”. In: *NeurIPS 32*.

Marteau-Ferey, U., D. Ostrovskii, F. Bach, and A. Rudi (2019). “Beyond Least-Squares: Fast Rates for Regularized Empirical Risk Minimization through Self-Concordance”. In: *COLT 32*.

Martin-Brualla, R., N. Radwan, M. S. M. Sajjadi, J. T. Barron, A. Dosovitskiy, and D. Duckworth (2021). “NeRF in the Wild: Neural Radiance Fields for Unconstrained Photo Collections”. In: *CVPR 2021*. DOI: 10.1109/CVPR46437.2021.00713.

Massart, P. (2007). *Concentration inequalities and model selection*. Lecture Notes in Mathematics. Springer.

Matthews, A., M. van der Wilk, T. Nickson, K. Fujii, A. Boukouvalas, P. León-Villagrá, Z. Ghahramani, and J. Hensman (2017). “GPflow: A Gaussian process library using TensorFlow”. In: *JMLR* 18.40, pp. 1–6.

Mattos Neto, P. S. G. de, J. F. L. de Oliveira, D. S. de Oliveira Santos Junior, H. V. Siqueira, M. H. Da Nobrega Marinho, and F. Madeiro (2020). “A Hybrid Nonlinear Combination System for Monthly Wind Speed Forecasting”. In: *IEEE Access* 8, pp. 191365–191377. DOI: 10.1109/ACCESS.2020.3032070.

Max, N. (1995). “Optical models for direct volume rendering”. In: *IEEE Transactions on Visualization and Computer Graphics* 1.2, pp. 99–108. DOI: 10.1109/2945.468400.

Meanti, G., L. Carratino, E. De Vito, and L. Rosasco (2022). “Efficient Hyperparameter Tuning for Large Scale Kernel Ridge Regression”. In: *AISTATS 25*. Vol. 151, pp. 6554–6572.

Meanti, G., L. Carratino, L. Rosasco, and A. Rudi (2020). “Kernel methods through the roof: handling billions of points efficiently”. In: *NeurIPS 34*.

Meshry, M., D. B. Goldman, S. Khamis, H. Hoppe, R. Pandey, N. Snavely, and R. Martin-Brualla (2019). “Neural Rerendering in the Wild”. In: *CVPR 2019*, pp. 6878–6887. DOI: 10.1109/CVPR.2019.00704.

Messner, J. W. and P. Pinson (2019). “Online adaptive lasso estimation in vector autoregressive models for high dimensional wind power forecasting”. In: *International Journal of Forecasting* 35.4, pp. 1485–1498. DOI: 10.1016/j.ijforecast.2018.02.001.

Metta, G., P. Fitzpatrick, and L. Natale (2006). “YARP: Yet Another Robot Platform”. In: *International Journal of Advanced Robotics Systems* 3.1, pp. 43–48. DOI: 10.5772/5761.

Metta, G., L. Natale, F. Nori, G. Sandini, D. Vernon, L. Fadiga, C. von Hofsten, K. Rosander, M. Lopes, J. Santos-Victor, A. Bernardino, and L. Montesano (2010). “The iCub humanoid robot: an open-systems platform for research in cognitive development.” In: *Neural networks* 23.8-9, pp. 1125–34. DOI: 10.1016/j.neunet.2010.08.010.

Michieli, U. and P. Zanuttigh (2019). “Incremental Learning Techniques for Semantic Segmentation”. In: *ICCV 2019 Workshops*. DOI: 10.1109/ICCVW.2019.00400.

Mildenhall, B., P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng (2020). “NeRF: Representing scenes as neural radiance fields for view synthesis”. In: *ECCV 2020*. Springer, pp. 405–421. DOI: 10.1145/3503250.

Mildenhall, B., P. P. Srinivasan, R. O. Cayon, N. K. Kalantari, R. Ramamoorthi, R. Ng, and A. Kar (2019). “Local light field fusion: practical view synthesis with prescriptive sampling guidelines”. In: *ACM Trans. Graph.* 38.4. DOI: 10.1145/3306346.3322980.

Minh, H. Q., P. Niyogi, and Y. Yao (2006). “Mercer’s Theorem, Feature Maps, and Smoothing”. In: *COLT 2006*, pp. 154–168. DOI: 10.1007/11776420_14.

Mohandes, M., S. Rehman, H. Nuha, M. Islam, and F. Schulze (2021a). “Accuracy of wind speed predictability with heights using Recurrent Neural networks”. In: *FME Transactions* 49.4, pp. 908–918. DOI: 10.5937/fme2104908M.

Mohandes, M., S. Rehman, H. Nuha, M. Islam, and F. Schulze (2021b). “Wind Speed Predictability Accuracy with Height Using LiDAR Based Measurements and Artificial Neural Networks”. In: *Applied Artificial Intelligence* 35.8, pp. 605–622. DOI: 10.1080/08839514.2021.1922850.

Mozaffari, L., A. Mozaffari, and N. L. Azad (2015). “Vehicle speed prediction via a sliding-window time series analysis and an evolutionary least learning machine: A case study on San Francisco urban roads”. In: *Engineering science and technology, an international journal* 18.2, pp. 150–162. DOI: 10.1016/j.jestch.2014.11.002.

Mroueh, Y., T. Poggio, L. Rosasco, and J.-J. E. Slotine (2012). “Multiclass Learning with Simplex Coding”. In: *NeurIPS 25*.

Müecke, N. (2019). “Reducing training time by efficient localized kernel regression”. In: *AISTATS 22*, pp. 2603–2610.

Müller, T., A. Evans, C. Schied, and A. Keller (2022). “Instant Neural Graphics Primitives with a Multiresolution Hash Encoding”. In: *ACM Trans. Graph.* 41.4. DOI: 10.1145/3528223.3530127.

Nitanda, A. and T. Suzuki (2019). “Stochastic Gradient Descent with Exponential Convergence Rates of Expected Classification Errors”. In: *AISTATS 2019*, pp. 1417–1426.

Nowak, A., F. Bach, and A. Rudi (2019). “Sharp Analysis of Learning with Discrete Losses”. In: *AISTATS 2019*, pp. 1920–1929.

NVIDIA Corporation (2020a). *cuBLAS*. Version 10.2. URL: https://developer.nvidia.com/cublas.

NVIDIA Corporation (2020b). *cuSOLVER*. Version 10.2. URL: https://developer.nvidia.com/cusolver.

NVIDIA Corporation (2020c). *cuSPARSE*. Version 10.2. URL: https://developer.nvidia.com/cusparse.

Ong, F., X. Zhu, J. Y. Cheng, K. M. Johnson, P. E. Z. Larson, S. S. Vasanawala, and M. Lustig (2020). “Extreme MRI: Large-scale volumetric dynamic imaging from continuous non-gated acquisitions”. In: *Magnetic Resonance in Medicine* 84.4, pp. 1763–1780. DOI: https://doi.org/10.1002/mrm.28235.

P. Zhang L. Hu, B. Z. and P. Pan (2020). "Spatial Constrained Memory Network for Semi-supervised Video Object Segmentation". In: *The 2020 DAVIS Challenge on Video Object Segmentation - CVPR Workshops*.

Park, K., U. Sinha, J. T. Barron, S. Bouaziz, D. B. Goldman, S. M. Seitz, and R. Martin-Brualla (2021). "Nerfies: Deformable Neural Radiance Fields". In: *ICCV 2021*. DOI: 10.1109/ICCV48922.2021.00581.

Park, K., U. Sinha, P. Hedman, J. T. Barron, S. Bouaziz, D. B. Goldman, R. Martin-Brualla, and S. M. Seitz (2021). "HyperNeRF: A Higher-Dimensional Representation for Topologically Varying Neural Radiance Fields". In: *ACM Trans. Graph.* 40.6. DOI: 10.1145/3478513.3480487.

Parmezan, A. R. S., V. M. Souza, and G. E. A. P. A. Batista (2019). "Evaluation of statistical and machine learning models for time series prediction: Identifying the state-of-the-art and the best conditions for the use of each model". In: *Information Sciences* 484, pp. 302–337. DOI: 10.1016/j.ins.2019.01.076.

Pasquale, G., T. Mar, C. Ciliberto, L. Rosasco, and L. Natale (2016). "Enabling Depth-Driven Visual Attention on the iCub Humanoid Robot: Instructions for Use and New Perspectives". In: *Frontiers in Robotics and AI* 3, p. 35. DOI: 10.3389/frobt.2016.00035.

Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *NeurIPS 32*.

Pathak, D., Y. Shentu, D. Chen, P. Agrawal, T. Darrell, S. Levine, and J. Malik (2018). "Learning instance segmentation by interaction". In: *CVPR 2018*, pp. 2042–2045. DOI: 10.1109/CVPRW.2018.00276.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay (2011). "Scikit-learn: Machine Learning in Python". In: *JMLR* 12, pp. 2825–2830.

Pedregosa, F. (2016). "Hyperparameter optimization with approximate gradient". In: *ICML 33*.

Peng, H., S. Zhe, X. Zhang, and Y. Qi (2017). "Asynchronous Distributed Variational Gaussian Process for Regression". In: *ICML 34*.

Peng, S., W. Jiang, H. Pi, X. Li, H. Bao, and X. Zhou (2020). "Deep Snake for Real-Time Instance Segmentation". In: *CVPR 2020*, pp. 8533–8542. DOI: 10.1109/CVPR42600.2020.00856.

Peng, S., M. Niemeyer, L. M. Mescheder, M. Pollefeys, and A. Geiger (2020). "Convolutional Occupancy Networks". In: *ECCV 2020*. Vol. 12348. Lecture Notes in Computer Science, pp. 523–540. DOI: 10.1007/978-3-030-58580-8_31.

Perez-Rua, J.-M., X. Zhu, T. M. Hospedales, and T. Xiang (2020). "Incremental few-shot object detection". In: *CVPR 2020*, pp. 13846–13855. DOI: 10.1109/CVPR42600.2020.01386.

Pham, N. and R. Pagh (2013). "Fast and Scalable Polynomial Kernels via Explicit Feature Maps". In: *SIGKDD 19*, pp. 239–247. DOI: 10.1145/2487575.2487591.

Pillaud-Vivien, L., A. Rudi, and F. Bach (2018). "Exponential Convergence of Testing Error for Stochastic Gradient Methods". In: *COLT 2018*, pp. 250–296.

Pinheiro, P. O., T.-Y. Lin, R. Collobert, and P. Dollár (2016). "Learning to Refine Object Segments". In: *ECCV 2016*, pp. 75–91. DOI: 10.1007/978-3-319-46448-0_5.

Pumarola, A., E. Corona, G. Pons-Moll, and F. Moreno-Noguer (2021). "D-NeRF: Neural Radiance Fields for Dynamic Scenes". In: *CVPR 2021*. DOI: 10.1109/CVPR46437.2021.01018.

Quiñonero-Candela, J. and C. E. Rasmussen (2005). "A Unifying View of Sparse Approximate Gaussian Process Regression". In: *Journal of Machine Learning Research* 6, pp. 1939–1959.

Rahimi, A. and B. Recht (2008). "Random Features for Large-Scale Kernel Machines". In: *NeurIPS 20*.

Rahimi, A. and B. Recht (2009). “Weighted Sums of Random Kitchen Sinks: Replacing minimization with randomization in learning”. In: *NeurIPS 21*.

Rajeswaran, A., C. Finn, S. M. Kakade, and S. Levine (2019). “Meta-Learning with Implicit Gradients”. In: *NeurIPS 32*, pp. 113–124.

Raskutti, G., M. J. Wainwright, and B. Yu (2014). “Early stopping and non-parametric regression: an optimal data-dependent stopping rule”. In: *The Journal of Machine Learning Research* 15.1, pp. 335–366. DOI: 10.1109/Allerton.2011.6120320.

Rasmussen, C. E. and C. K. I. Williams (2005). *Gaussian Processes for Machine Learning.* The MIT Press.

Redmon, J. and A. Farhadi (2018). *YOLOv3: An Incremental Improvement.* arXiv: 1804.02767.

Reikard, G. (2009). “Forecasting ocean wave energy: Tests of time-series models”. In: *Ocean Engineering* 36.5, pp. 348–356. DOI: 10.1016/j.oceaneng.2009.01.003.

Ren, S., K. He, R. Girshick, and J. Sun (2015). “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. In: *NeurIPS 2015*. DOI: 10.1109/TPAMI.2016.2577031.

Ritchie, H. and M. Roser (2022). *Energy Production and Consumption.* URL: https://ourworldindata.org/energy-production-consumption.

Robbins, H. and S. Monro (1951). “A Stochastic Approximation Method”. In: *The Annals of Mathematical Statistics* 22.3, pp. 400–407. DOI: 10.1007/978-1-4612-5110-1_9.

Rudi, A., D. Calandriello, L. Carratino, and L. Rosasco (2018). “On Fast Leverage Score Sampling and Optimal Learning”. In: *NeurIPS 31*, pp. 5677–5687.

Rudi, A., R. Camoriano, and L. Rosasco (2015). “Less is More: Nyström Computational Regularization”. In: *NeurIPS 28*.

Rudi, A., L. Carratino, and L. Rosasco (2017). “FALKON: An Optimal Large Scale Kernel Method”. In: *NeurIPS 29*.

Rudi, A. and L. Rosasco (2017). “Generalization Properties of Learning with Random Features”. In: *NeurIPS 31*, pp. 3218–3228.

Ruiz-Aguilar, J. J., I. Turias, J. González-Enrique, D. Urda, and D. Elizondo (2021). “A permutation entropy-based EMD–ANN forecasting ensemble approach for wind speed prediction”. In: *Neural Computing and Applications* 33.7, pp. 2369–2391. DOI: 10.1007/s00521-020-05141-w.

S. Garg, V. G. and S. Kumar (2020). “Unsupervised Video Object Segmentation using Online Mask Selection and Space-time Memory Networks”. In: *The 2020 DAVIS Challenge on Video Object Segmentation - CVPR Workshops.*

Saad, Y. (2003). *Iterative methods for sparse linear systems.* Second. Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9780898718003.

Schölkopf, B., R. Herbrich, and A. J. Smola (2001). “A Generalized Representer Theorem”. In: *COLT 14*. DOI: 10.1007/3-540-44581-1_27.

Schölkopf, B. and A. J. Smola (2001). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond.* Adaptive Computation and Machine Learning series. The MIT Press. DOI: 10.7551/mitpress/4175.001.0001.

Seeger, M. W. (2008). “Cross-validation optimization for large scale structured classification kernel methods”. In: *JMLR* 9.39, pp. 1147–1178.

Seeger, M. W., C. K. I. Williams, and N. D. Lawrence (2003). “Fast Forward Selection to Speed Up Sparse Gaussian Process Regression”. In: *AISTATS 2003*, pp. 254–261.

Shahriari, B., K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas (2016). “Taking the Human Out of the Loop: A Review of Bayesian Optimization”. In: *Proceedings of the IEEE* 104.1, pp. 148–175. DOI: 10.1109/JPROC.2015.2494218.

Shalev-Shwartz, S. and S. Ben-David (2014). *Understanding Machine Learning: From Theory to Algorithms.* Cambridge University Press. DOI: 10.1017/CBO9781107298019.

Shao, R., Z. Zheng, H. Tu, B. Liu, H. Zhang, and Y. Liu (2022). *Tensor4D : Efficient Neural 4D Decomposition for High-fidelity Dynamic Reconstruction and Rendering*. arXiv: 2211.11610 `[cs.CV]`.

Shewchuk, J. R. (1994). *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Tech. rep. Carnegie Mellon University.

Shmelkov, K., C. Schmid, and K. Alahari (2017). "Incremental learning of object detectors without catastrophic forgetting". In: *ICCV 2017*, pp. 3400–3409. DOI: 10.1109/ICCV.2017.368.

Shu, X., C. Liu, T. Li, C. Wang, and C. Chi (2018). "A Self-Supervised Learning Manipulator Grasping Approach Based on Instance Segmentation". In: *IEEE Access* 6, pp. 65055–65064. DOI: 10.1109/ACCESS.2018.2877796.

Si, S., C.-J. Hsieh, and I. S. Dhillon (2017). "Memory Efficient Kernel Approximation". In: *Journal of Machine Learning Research* 18.1, pp. 682–713.

Siam, M., C. Jiang, S. Lu, L. Petrich, M. Gamal, M. Elhoseiny, and M. Jagersand (2019). "Video object segmentation using teacher-student adaptation in a human robot interaction (hri) setting". In: *ICRA 2019*, pp. 50–56. DOI: 10.1109/ICRA.2019.8794254.

Silverman, B. W. (1985). "Some Aspects of the Spline Smoothing Approach to Non-Parametric Regression Curve Fitting". In: *Journal of the Royal Statistical Society. Series B (Methodological)* 47.1, pp. 1–52. DOI: 10.1111/j.2517-6161.1985.tb01327.x.

Smil, V. (2017). *Energy transitions: global and national perspectives*. Second. Praeger.

Smola, A. J. and B. Schölkopf (2000). "Sparse Greedy Matrix Approximation for Machine Learning". In: *ICML 17*.

Snelson, E. and Z. Ghahramani (2005). "Sparse Gaussian processes using pseudo-inputs". In: *Advances in neural information processing systems* 18.

Snoek, J., H. Larochelle, and R. P. Adams (2012). "Practical Bayesian Optimization of Machine Learning Algorithms". In: *Neurips 25*.

Song, L., A. Chen, Z. Li, Z. Chen, L. Chen, J. Yuan, Y. Xu, and A. Geiger (2022). *NeRFPlayer: A Streamable Dynamic Scene Representation with Decomposed Neural Radiance Fields*. DOI: 10.1109/TVCG.2023.3247082. arXiv: 2210.15947.

Steinwart, I. and A. Christmann (2008). *Support Vector Machines*. Information Science and Statistics. Springer.

Steinwart, I., D. Hush, and C. Scovel (2009). "Optimal Rates for Regularized Least Squares Regression". In: *COLT 22*.

Steinwart, I. and P. Thomann (2017). *liquidSVM: A fast and versatile SVM package*. arXiv: 1702.06899.

Sterge, N., B. Sriperumbudur, L. Rosasco, and A. Rudi (2020). "Gain with no Pain: Efficient Kernel-PCA by Nyström Sampling". In: *AISTATS 23*.

Sun, C., M. Sun, and H.-T. Chen (2022). "Direct Voxel Grid Optimization: Super-fast Convergence for Radiance Fields Reconstruction". In: *CVPR 2022*. DOI: 10.1109/CVPR52688.2022.00538.

Sun, Y., A. Gilbert, and A. Tewari (2018). "But How Does It Work in Theory? Linear SVM with Random Features". In: *NeurIPS 31*.

Suykens, J. A. K., T. Van Gestel, J. De Brabanter, B. De Moor, and J. Vandewalle (2002). *Least Squares Support Vector Machines*. World Scientific. DOI: 10.1142/5089.

Tan, M., R. Pang, and Q. V. Le (2020). "EfficientDet: Scalable and efficient object detection". In: *CVPR 2020*, pp. 10781–10790. DOI: 10.1109/CVPR42600.2020.01079.

Tancik, M., V. Casser, X. Yan, S. Pradhan, B. Mildenhall, P. P. Srinivasan, J. T. Barron, and H. Kretzschmar (2022). "Block-NeRF: Scalable Large Scene Neural View Synthesis". In: *CVPR 2022*. DOI: 10.1109/CVPR52688.2022.00807.

Thomann, P., I. Blaschzyk, M. Meister, and I. Steinwart (2017). "Spatial Decompositions for Large Scale SVMs". In: *AISTATS 20*, pp. 1329–1337.

Tian, Z., C. Shen, H. Chen, and T. He (2019). “FCOS: Fully convolutional one-stage object detection”. In: *ICCV 2019*, pp. 9627–9636. DOI: 10.1109/ICCV.2019.00972.

Tibshirani, R. (1996). “Regression Shrinkage and Selection Via the Lasso”. In: *Journal of the Royal Statistical Society: Series B (Methodological)* 58.1, pp. 267–288. DOI: 10.1111/j.2517-6161.1996.tb02080.x.

Titsias, M. (2009). “Variational Learning of Inducing Variables in Sparse Gaussian Processes”. In: *AISTATS 12*.

Trebing, K. and S. Mehrkanoon (2020). *Wind speed prediction using multidimensional convolutional neural networks*. DOI: 10.1109/SSCI47803.2020.9308323. arXiv: 2007.12567 [cs].

Tretschk, E., A. Tewari, V. Golyanik, M. Zollhöfer, C. Lassner, and C. Theobalt (2021). “Non-Rigid Neural Radiance Fields: Reconstruction and Novel View Synthesis of a Dynamic Scene From Monocular Video”. In: *ICCV 2021*. DOI: 10.1109/ICCV48922.2021.01272.

Tsybakov, A. B. (2004). “Optimal aggregation of classifiers in statistical learning”. In: *The Annals of Statistics* 32.1, pp. 135–166. DOI: 10.1214/aos/1079120131.

Tsybakov, A. B. (2003). “Optimal Rates of Aggregation”. In: *Learning Theory and Kernel Machines*. Springer, pp. 303–313. DOI: 10.1007/978-3-540-45167-9_23.

Tu, S., R. Roelofs, S. Venkataraman, and B. Recht (2016). *Large Scale Kernel Learning using Block Coordinate Descent*. arXiv: 1602.05310.

Vapnik, V. N. (1998). *Statistical Learning Theory*. John Wiley & Sons.

Varma, S. and R. Simon (2006). “Bias in error estimation when using cross-validation for model selection”. In: *BMC Bioinformatics* 91. DOI: 10.1186/1471-2105-7-91.

Vedaldi, A. and A. Zisserman (2012). “Efficient Additive Kernels via Explicit Feature Maps”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34.3, pp. 480–492. DOI: 10.1109/TPAMI.2011.153.

Vigogna, S., G. Meanti, E. De Vito, and L. Rosasco (2022). “Multiclass learning with margin: exponential rates with no bias-variance trade-off”. In: *ICML 39*. Vol. 162.

Voigtlaender, P. and B. Leibe (2017). “Online adaptation of convolutional neural networks for video object segmentation”. In: *Proceedings of the British Machine Vision Conference (BMVC)*. DOI: 10.5244/C.31.116.

Wada, K., K. Okada, and M. Inaba (2019). “Joint learning of instance and semantic segmentation for robotic pick-and-place with heavy occlusions in clutter”. In: *ICRA 2019*, pp. 9558–9564. DOI: 10.1109/ICRA.2019.8793783.

Wang, F., S. Tan, X. Li, Z. Tian, and H. Liu (2022). *Mixed Neural Voxels for Fast Multi-view Video Synthesis*. arXiv: 2212.00190 [cs.CV].

Wang, K., G. Pleiss, J. Gardner, S. Tyree, K. Q. Weinberger, and A. G. Wilson (2019). “Exact Gaussian Processes on a Million Data Points”. In: *NeurIPS 32*.

Wang, S., A. Gittens, and M. W. Mahoney (2019). “Scalable Kernel K-Means Clustering with Nyström Approximation: Relative-Error Bounds”. In: *J. Mach. Learn. Res.* 20.1, pp. 431–479.

Wang, Z., A. Bovik, H. Sheikh, and E. Simoncelli (2004). “Image quality assessment: from error visibility to structural similarity”. In: *IEEE Transactions on Image Processing* 13.4, pp. 600–612. DOI: 10.1109/TIP.2003.819861.

Wen, Z., J. Shi, Q. Li, B. He, and J. Chen (2018). “ThunderSVM: A Fast SVM Library on GPUs and CPUs”. In: *Journal of Machine Learning Research* 19, pp. 797–801.

Wilk, M. van der, V. Dutordoir, S. T. John, A. Artemev, V. Adam, and J. Hensman (2020). *A Framework for Interdomain and Multioutput Gaussian Processes*. arXiv: 2003.01115 [stat.ML].

Williams, C. K. I. and M. Seeger (2001). “Using the Nyström Method to Speed Up Kernel Machines”. In: *NeurIPS 13*.

Wilson, A. G. and H. Nickisch (2015). “Kernel interpolation for scalable structured Gaussian processes (KISS-GP)”. In: *ICML 32*.

Wilson, A. G., Z. Hu, R. Salakhutdinov, and E. P. Xing (2016). “Stochastic Variational Deep Kernel Learning”. In: *NeurIPS 30*.

Wilt, N. (2013). *The CUDA handbook: A comprehensive guide to GPU programming*. Pearson Education.

Wizadwongsa, S., P. Phongthawee, J. Yenphraphai, and S. Suwajanakorn (2021). *NeX: Real-time View Synthesis with Neural Basis Expansion*. DOI: 10.1109/CVPR46437.2021.00843. arXiv: 2103.05606 [cs.CV].

Wolpert, D. H. (1996). “The Lack of A Priori Distinctions Between Learning Algorithms”. In: *Neural Computation* 8.7, pp. 1341–1390. DOI: 10.1162/neco.1996.8.7.1341.

Wu, R. (2018). “A Heterogeneous Parallel Cholesky Block Factorization Algorithm”. In: *IEEE Access* 6. DOI: 10.1109/ACCESS.2018.2803794.

Wu, T., F. Zhong, A. Tagliasacchi, F. Cole, and C. Oztireli (2022). *D^2NeRF: Self-Supervised Decoupling of Dynamic and Static Objects from a Monocular Video*. arXiv: 2205.15838 [cs.CV].

Xian, W., J.-B. Huang, J. Kopf, and C. Kim (2021). “Space-time Neural Irradiance Fields for Free-Viewpoint Video”. In: *CVPR 2021*, pp. 9421–9431. DOI: 10.1109/CVPR46437.2021.00930.

Xiang, Y., T. Schmidt, V. Narayanan, and D. Fox (2018). “PoseCNN: A Convolutional Neural Network for 6D Object Pose Estimation in Cluttered Scenes”. In: *Robotics: Science and Systems (RSS)*. DOI: 10.15607/RSS.2018.XIV.019.

Xie, C., Y. Xiang, A. Mousavian, and D. Fox (2020). “The best of both modes: Separately leveraging rgb and depth for unseen object instance segmentation”. In: *CoRL 2020*, pp. 1369–1378.

Xie, C., Y. Xiang, A. Mousavian, and D. Fox (2021). “Unseen object instance segmentation for robotic environments”. In: *IEEE Transactions on Robotics*. DOI: 10.1109/TRO.2021.3060341.

Xu, Y., G. Yang, J. Luo, J. He, and H. Sun (2022). “A multi-location short-term wind speed prediction model based on spatiotemporal joint learning”. In: *Renewable Energy* 183, pp. 148–159. DOI: 10.1016/j.renene.2021.10.075.

Yang, T., Y.-F. Li, M. Mahdavi, R. Jin, and Z.-H. Zhou (2012). “Nyström Method vs Random Fourier Features: A Theoretical and Empirical Comparison”. In: *NeurIPS 24*.

Yao, Y., L. Rosasco, and A. Caponnetto (2007). “On early stopping in gradient descent learning”. In: *Constructive Approximation* 26.2, pp. 289–315. DOI: 10.1007/s00365-006-0663-2.

Yousuf, M. U., I. Al-Bahadly, and E. Avci (2021). “Short-Term Wind Speed Forecasting Based on Hybrid MODWT-ARIMA-Markov Model”. In: *IEEE Access* 9, pp. 79695–79711. DOI: 10.1109/ACCESS.2021.3084536.

Yu, A., R. Li, M. Tancik, H. Li, R. Ng, and A. Kanazawa (2021). “PlenOctrees for Real-time Rendering of Neural Radiance Fields”. In: *ICCV 2021*, pp. 5732–5741. DOI: 10.1109/ICCV48922.2021.00570.

Yuan, W., Z. Lv, T. Schmidt, and S. Lovegrove (2021). “STaR: Self-supervised Tracking and Reconstruction of Rigid Objects in Motion with Neural Rendering”. In: *CVPR 2021*. DOI: 10.1109/CVPR46437.2021.01294.

Zandieh, A., I. Han, H. Avron, N. Shoham, C. Kim, and J. Shin (2021). “Scaling Neural Tangent Kernels via Sketching and Random Features”. In: *NeurIPS 33*. Vol. 34, pp. 1062–1073.

Zhang, C., H. Wei, X. Zhao, T. Liu, and K. Zhang (2016). “A Gaussian process regression based hybrid approach for short-term wind speed prediction”. In: *Energy Conversion and Management* 126, pp. 1084–1092. DOI: 10.1016/j.enconman.2016.08.086.

Zhang, C., S. Bengio, M. Hardt, B. Recht, and O. Vinyals (2021). “Understanding deep learning (still) requires rethinking generalization”. In: *Communications of the ACM* 64.3, pp. 107–115. DOI: 10.1145/3446776.

Zhang, T. (2004a). “Statistical Analysis of Some Multi-Category Large Margin Classification Methods”. In: *Journal of Machine Learning Research* 5, pp. 1225–1251.

Zhang, T. (2004b). “Statistical Behavior and Consistency of Classification Methods Based on Convex Risk Minimization”. In: *The Annals of Statistics* 32.1, pp. 56–85. DOI: 10.1214/aos/1079120130.

Zhou, X., D. Wang, and P. Krähenbühl (2019). *Objects as points*. arXiv: 1904.07850.

Zhu, Q., J. Chen, L. Zhu, X. Duan, and Y. Liu (2018). “Wind Speed Prediction with Spatio-Temporal Correlation: A Deep Learning Approach”. In: *Energies* 11.4. DOI: 10.3390/en11040705.

Appendix A

Datasets

A.1 Generic datasets for kernel learning

In order to benchmark the algorithms proposed in Chapters 3 and 4 we have used several datasets which we believe represent a broad set of scenarios for kernel learning, in terms of data size, data type, and learning task. They can be roughly divided into three groups: medium sized unstructured datasets (both for regression and binary classification), medium sized image recognition datasets (multiclass classification) and large unstructured datasets (classification and regression). We normally used a standard random split with 80% training, 20% testing data unless predefined splits existed (for example in the MNIST dataset, as noted in Table A.1). Preprocessing mostly consisted in basic data cleaning and data standardization to zero mean and unit standard deviation; we comment in more detail below on specific preprocessing steps applied to the individual datasets. Table A.1 provides a synthetic overview, as well as links from which the data can be retrieved.

The error metrics used are dataset-dependent, and outlined below. For regression problems we use the RMSE, which for predictions with a model $\hat{f}$ is defined as $\sqrt{n^{-1}\sum_{i=1}^{n}(y_i - \hat{f}(x_i))^2}$ and its normalized version the NRMSE:

$$NRMSE : \left| \frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{f}(x_i))^2}}{\frac{1}{n}\sum_{i=1}^{n} y_i} \right|. \tag{A.1}$$

For the MSD dataset we use another metric called the *relative error* which is defined as

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{f}(x_i)}{y_i}\right)^2}. \tag{A.2}$$

For classification problems we use the fraction of misclassified examples (c-error), and the area under the curve (AUC) metric.

HIGGS, SmallHIGGS has dimensions $n = 1.1 \times 10^7, d = 28$ and a binary target. It is thus a very large classification dataset coming from high energy physics. It was preprocessed to 0 mean and unit variance. Results are reported on a 80-20 split with 1 minus the AUC metric in Table 3.3 and with the binary classification error in Table 3.4. It is available for download at https://archive.ics.uci.edu/ml/datasets/HIGGS. We took a small random subsample of 30 000 points to generate the SmallHIGGS dataset, which has predefined training and test sets.

TIMIT has dimensions $n = 1.2 \times 10^6, d = 440$ and a multiclass target with 144 classes. TIMIT comes from audio data, and our dataset uses the 10 ms resampling rate as in Ma et al. (2017) and Ma et al. (2019). It was preprocessed to 0 mean and unit standard deviation. The error metric is classification error on a subset of classes (as used in Ma et al. (2017)), and is calculated over a standardized subset of 57 242 samples. It is available for download at https://catalog.ldc.upenn.edu/LDC93S1.

YELP has dimensions $n = 1.5 \times 10^6, d = 6.52 \times 10^7$ and a continuous target. This dataset consists of text reviews, labeled with their star rating. We used the same data as Tu et al. (2016) (Yelp round 9 dataset), processed by extracting all 3-grams and encoding each review by a count vector which tells us which 3-grams are present. Such encoding produces a large number of sparse features which is reflected in the huge dimensionality of this dataset. Since the data is sparse we did not normalize it. The error metric is RMSE, calculated on random 20% of the samples. The dataset can be provided on request.

TAXI has dimensions $n = 1.1 \times 10^9, d = 9$ with a continuous target. Data are normalized to have zero-mean and unit standard deviation; reported error is RMSE on a 20% random sub-sample. The data can be downloaded by following instructions at https://github.com/toddwschneider/nyc-taxi-data. Consistently with other users of this dataset (H. Peng et al., 2017) we took the data from January 2009 to December 2015, excluding outliers (taxi trips more than 5 hours long) and trips where the pickup or drop off location is outside of NYC.

AIRLINE, AIRLINE-CLS has dimensions $n = 5.930 \times 10^6, d = 8$ and a continuous target. Data are normalized to zero-mean and unit standard deviation, and the error is the MSE over normalized targets calculated on random test-sets of size 33 % of the full data (consistently with the literature (Hensman, Durrande, et al., 2017; Hensman, Fusi, et al., 2013)). The same dataset is also used for binary classification by thresholding the target at 0, which results in the **AIRLINE-CLS** dataset. For this latter variation we used 100 000 random points for testing, reporting classification error in Table 3.3 and 1 minus the AUC in Table 3.4 to facilitate comparisons with the literature. The data can be downloaded from https://www.transtats.bts.gov/Fields.asp?Table_ID=236 and http://stat-computing.org/dataexpo/2009/supplemental-data.html.

MSD has dimensions $n = 5.100\times10^5, d = 90$ with continuous target. Data are normalized to zero-mean and unit standard deviation, and we report the relative error over a standard test-set of size 51 630. The dataset can be downloaded from https://archive.ics.uci.edu/ml/datasets/YearPredictionMSD.

SUSY has dimensions $n = 5\times10^6, d = 18$ with binary target. Data are normalized to zero-mean and unit standard deviation. We report the classification error on 20% of the data. Data is available from the UCI repositories https://archive.ics.uci.edu/ml/datasets/SUSY.

SpaceGA, Abalone, MG, CpuSmall, Energy Small regression datasets between 1385 (MG) and 8192 (CpuSmall) samples, both labels and predictors are normalized to have zero mean and unit standard deviation; the error metric used is NRMSE.

Road3D, Buzz, Protein, HouseElectric, BlogFeedback Regression datasets of medium to large size from the UCI ML repositories. We normalized the labels to have zero mean and unit standard deviation for Road3D, BlogFeedback, Buzz and Protein, and used an additional log transformation on the labels of HouseElectric. Measured error is NRMSE. The predictor matrix is also normalized to zero mean, unit standard deviation.

MNIST, FashionMNIST, SVHN, CIFAR-10 Four standard image recognition datasets. Here the labels are one-hot encoded (all datasets have 10 classes), and the design matrix is normalized using min-max normalization in the 0-1 range. Standard train/test splits are used.

Chiet A time-series dataset for short-term wind prediction with $n = 34\,059$ samples, $d = 144$ dimensions, and a continuous target. The labels and predictors are both normalized to have zero mean, unit standard deviation. The error is measured with the NRMSE. A fixed split in time is used. The dataset is available upon request.

Ictus A binary classification dataset of brain MRI simulations with $n = 29\,545$ samples in $d = 992$ dimensions. Predictors are standardized to have zero mean, unit standard deviation, and a random 80/20 train/test split is used. The dataset is available upon request.

Cod-RNA, SVMGuide1, IJCNN1, CovType Four datasets for binary classification ranging between approximately 3000 points for SvmGuide1 and 5×10^5 points for CovType. The design matrix is normalized to have zero mean, unit standard deviation, while the labels are -1 and $+1$.

Table A.1 Summary of the datasets used.

	n	d	train/test split	Error metrics
SpaceGA	3107	6	70%/30%	NRMSE
Abalone	4177	8	70%/30%	NRMSE
MG	1385	6	70%/30%	NRMSE
CpuSmall	8192	12	70%/30%	NRMSE
Energy	768	8	80%/20%	NRMSE
Road3D	434 874	3	70%/30%	RMSE
Buzz	2 049 280	11	70%/30%	RMSE
Protein	45 730	9	80%/20%	NRMSE
BlogFeedback	60 021	280	52 397/7624	RMSE
MNIST	70 000	784	60 000/10 000	10 class c-error
FashionMNIST	70 000	784	60 000/10 000	10 class c-error
SVHN	99 289	1024	73 257/26 032	10 class c-error
CIFAR-10	60 000	1024	50 000/10 000	10 class c-error
Chiet	34 059	144	26 227/7832	NRMSE
Ictus	29 545	992	80%/20%	binary c-error
Cod-RNA	331 152	8	59 535/271 617	binary c-error
SVMGuide1	7089	4	3089/4000	binary c-error
IJCNN1	141 691	22	49 990/91 701	binary c-error
CovType	581 012	54	70%/30%	binary c-error
SmallHIGGS	30 000	28	10 000/20 000	binary c-error, 1 - AUC
HIGGS	1.100×10^7	20	80%/20%	binary c-error, 1 - AUC
AIRLINE	5.930×10^6	8	67%/33%	MSE
AIRLINE-CLS	5.930×10^6	8	5 829 413/100 000	binary c-error
TAXI	1.100×10^9	9	80%/20%	RMSE
TIMIT	1.200×10^6	440	1 142 758/57 242	multiclass c-error
SUSY	5×10^6	18	80%/20%	binary c-error
MSD	5.100×10^5	90	5 048 370/51 630	relative error
YELP	1.500×10^6	6.520×10^7	80%/20%	RMSE

Appendix B

Out-Of-Core Algorithms

In this section we describe more in detail the out-of-GPU core algorithms for (a) Cholesky decomposition of a positive definite matrix and (b) multiplication of a triangular matrix by its transpose. Both algorithms use a similar technique of dividing the input matrix in smaller tiles such that operations can be performed in-core on the individual tiles. Then the main challenges of such algorithms consist in choosing when to bring which tiles in-core, and how to do so in parallel, handling data-dependencies between different tiles.

We handle parallelism between multiple GPUs using a static work-allocation scheme where the input matrix is divided into block rows or columns (made up of several tiles), and each GPU is assigned one or more such rows (or columns) block-cyclically, to ensure that the workload is approximately balanced. Ensuring a balanced workload is tricky since the input matrices are triangular, and for example a row at the top of a lower-triangular matrix will have many more tiles than a row towards the bottom of said matrix. Smaller tile-sizes (so thinner block rows/columns) make each processor's workload more even, but – in case the input matrix is not big enough – they reduce overall GPU utilization.

Triangular matrix multiplication. We begin by describing OOC triangular matrix multiplication, an operation which is known as LAUUM within the LAPACK library. Given an input upper triangular matrix $U \in \mathbb{R}^{n\times n}$, we want to calculate the upper triangle of $UU^\top$ and store it in the upper part of U (thus making this an in-place operation). We divide U in $N \times N$ tiles of size t (uneven tile sizes are possible, and indeed necessary to support all input sizes, but omitted from the description for clarity), and we index all matrices by their tiles: $U_{2,2}$ is the square tile at the second block-row and second block-column of U. The in-place LAUUM operation can be compactly described as $U_{i,j} = \sum_{k=j}^{N-1} U_{i,k}U_{j,k}^\top$ for $j \geq i$: to update a tile of U we need to multiply two block-rows of the original matrix. However, we can exploit the triangular structure of some of the above matrix multiplications to improve performance: for example, when $i = j$ it is possible to split the update into two parts $U_{i,i} = U_{i,i}U_{i,i}^\top + \sum_{k=i}^{N} U_{i,k}U_{i,k}^\top$ where the first part consists of an in-core LAUUM operation and the second of a symmetric matrix

Algorithm 4 Out-of-core LAUUM operation on an upper-triangular matrix. The algorithm's inputs are matrix U, a synchronization object `barrier`, an array of arrays describing which row indices are assigned to which processor blockAllocs, and the number of tiles per side N. The function described below should be called for every available GPU with a different procId value.

```
function OocLauum($U \in \mathbb{R}^{n\times n}$, barrier, blockAllocs, procId, $N$)
    for $i = 1, \dots, N$ do
        $C \in \mathbb{R}^{t\times t\cdot(N-i)}$ ← ToGPU(procId, $\left[U_{i,i}, \dots, U_{i,N}\right]$)
        barrier.wait()
        for $j \in$ blockAllocs[procId] do
            if $i = j$ then
                $C_1 \leftarrow C_1 C_1^\top$                                  ▷ via LAUUM
                if $i \neq N$ then
                    $C_1 \leftarrow C_1 + C_{1:(N-i+1)} C_{1:(N-i+1)}^\top$     ▷ via SYRK
                end if
            else if $j > i$ then
                $D \in \mathbb{R}^{t\times t\cdot(N-j)}$ ← ToGPU(procId, $\left[U_{j,j}, \dots, U_{j,N}\right]$)
                $C_{(j-i)} \leftarrow C_{(j-i)} D_1^\top$                      ▷ via TRMM
                if $j \neq N$ then
                    $C_{(j-i)} \leftarrow C_{(j-i+1):(N-i+1)} D_{2:(N-j+1)}^\top$   ▷ via GEMM
                end if
            end if
            $U_{i,j}$ ← FromGPU(procId, $C_{(j-i)}$)
        end for
    end forreturn $U$
end function
```

multiplication (BLAS routine SYRK) which can be up to twice as fast as the general matrix multiplication routine. Similarly, for $i < j$, $U_{i,j} = U_{i,j}U_{j,j}^\top + \sum_{k=j+1}^{N} U_{i,k}U_{j,k}^\top$ where the first part can use the TRMM routine from the BLAS library and the second must use the generic GEMM routine. To avoid overwriting parts of U which are still needed for the updates – especially in a multi-GPU setting – the rows of U are to be updated one at a time, from top to bottom. To ensure synchronization between multiple GPUs we use a thread barrier so that all GPUs start updating a given row after having loaded its original, non-updated version in GPU memory. GPU memory requirements for Algorithm 4 are two block-columns (*i.e.* $2Nt^2$ numbers). As discussed above, rows are assigned to GPUs in a 1D block-cyclic way. Such allocations are recorded in the `blockAllocs` variable.

An adaptation of Algorithm 4 is possible when in-place operation is not needed: it is sufficient to remove the synchronization barrier, and change line 18 to write the output to a different matrix.

Cholesky decomposition. We want to decompose positive definite matrix A into lower triangular matrix L such that $L^\top L = A$. But A does not fit entirely in GPU memory, and potentially more than one GPU is available. As before it is convenient to subdivide A into smaller tiles such that the tiles fit in GPU memory.

$$\begin{pmatrix} A_{1,1} & & \\ A_{2,1} & A_{2,2} & \\ \vdots & \ddots & \\ A_{n,1} & \dots & A_{n,n} \end{pmatrix} = \begin{pmatrix} L_{1,1} & & \\ L_{2,1} & L_{2,2} & \\ \vdots & \ddots & \\ L_{n,1} & \dots & L_{n,n} \end{pmatrix} \begin{pmatrix} L_{1,1}^T & L_{2,1}^T & \dots & L_{n,1}^T \\ & L_{2,2}^T & \dots & L_{n,2}^T \\ & & \ddots & \vdots \\ & & & L_{n,n}^T \end{pmatrix}$$

Then the in-place decomposition may proceed column-wise across matrix A, where each column update requires three steps. The first step is to use the in-core `POTRF` function from cuSOLVER NVIDIA Corporation, 2020b on a single tile. Then, a triangular solution step is used to update the remaining rows of the first column (taking the first column as an example $A_{j,1} = L_{j,1}L_{1,1}^\top, 1 < j < N$, so clearly $L_{j,1} = A_{j,1}(L_{1,1}^{-1})^\top$). This can be done by using the TRSM operation from any GPU BLAS implementation. Finally, the *trailing* submatrix must be updated with those terms which can be computed from the current column, so that after this last step such column is not needed anymore. This step consists of running $A_{ij} = A_{ij} - L_{i,1}L_{j,1}^\top$ where if c is the current column $i > c, \quad c < j \leq i$ (refer to Figure 3.3 for a more intuitive picture).

Running this algorithm in parallel requires dealing with several data dependencies in-between tiles, and in general it will not be possible to achieve perfect parallelism due to the inherently serial step of performing the Cholesky decomposition of the first tile in a column. We avoid coarse synchronization mechanisms such as the thread barrier which was used for the LAUUM OOC implementation, since we found they could introduce very high waiting times (barriers

Algorithm 5 Out-of-core, in-place Cholesky decomposition of symmetric positive definite matrix A. The lower triangle of A will be overwritten by L such that $L^\top L = A$. The function OocPotrf should be called for each available GPU with different values of the procId variable to parallelize the decomposition across GPUs. The inputs are the same as for Algorithm 4 but for work-table $T \in \mathbb{Z}^{N\times N}$ whose values are atomically updated by the different GPU processes to ensure synchronization.

```
1:  function OocPotrf($A$, blockAllocs, procId, $T$, $N$)
2:      for $i = 1, \dots, N$ do
3:          if $i \in$ blockAllocs[procId] then
4:              $B \leftarrow$ Load($A$, $T$, $i$, $j$, $i$)
5:              $B \leftarrow$ POTRF($B$)
6:              $A_{i,i} \leftarrow$ Write($B$, $T$, $i$, $i$)
7:          end if
8:          for $j \in$ blockAllocs[procId] do
9:              if $j \leq i$ then
10:                 continue
11:             end if
12:             $B \leftarrow$ Load($A$, $T$, $i$, $i$, $i+1$)
13:             $C \leftarrow$ Load($A$, $T$, $j$, $i$, $i$)
14:             $C \leftarrow C(B^{-1})^\top$                ▷ via TRSM
15:             $A_{j,i} \leftarrow$ Write($C$, $T$, $j$, $i$)
16:         end for
17:         for $j \in$ blockAllocs[procId] do
18:             if $j \leq i + 1$ then
19:                 continue
20:             end if
21:             $C \leftarrow$ Load($A$, $T$, $j$, $i$, $i+1$)
22:             for $y = i, \dots j$ do
23:                 $E \leftarrow$ Load($A$, $T$, $j$, $y$, $i$)
24:                 if $y = j$ then
25:                     $E \leftarrow E - CC^\top$          ▷ via SYRK
26:                 else
27:                     $D \leftarrow$ Load($A$, $T$, $y$, $i$, $i+1$)
28:                     $E \leftarrow E - DC^\top$          ▷ via GEMM
29:                 end if
30:                 $A_{j,y} \leftarrow$ Write($E$, $T$, $j$, $y$)
31:             end for
32:         end for
33:     end for
34: end function

35: function Load($A, T, i, j,$ exp)
36:     while $T_{i,j} <$ exp do
37:         wait
38:     end while
39:     return ToGPU($A_{i,j}$)
40: end function

41: function Write($G, T, i, j$)
42:     $T_{i,j} \leftarrow T_{i,j} + 1$
43:     return FromGPU($G$)
44: end function
```

Table B.1 Benchmark timings using a single GPU. The relative slowdown with respect to Falkon on 2 GPUs is also provided for comparison with Table 3.3.

	1 GPU	2 GPUs	Relative change
TAXI	7215 ± 4 s	3628 ± 2 s	1.99×
HIGGS	715 ± 6 s	443 ± 2 s	1.61×
YELP	1981 ± 6 s	1008 ± 2 s	1.97×
TIMIT	416 ± 4 s	288 ± 3 s	1.44×
AIRLINE	334 ± 2 s	245 ± 5 s	1.36×
MSD	81 ± 0 s	62 ± 1 s	1.31×
AIRLINE-CLS	391 ± 5 s	269 ± 3 s	1.45×
SUSY	29 ± 1 s	22 ± 0 s	1.32×

would be needed after each of the three steps of the algorithm to ensure proper synchronization). Our solution, which somewhat follows Ltaief et al., 2011, uses an integer table T with one entry per tile, which is shared between all GPU threads. The entries of T represent the current state of each tile: basically how many times the tile has been updated. Since we use a static row-cyclic work allocation like for the triangular matrix multiplication, each thread knows the expected state of a tile for each step (*e.g.* to perform the first step on tile $A_{c,c}$ the tile must have been updated exactly c times). So it can wait until such state has been reached, then read the tile into GPU memory, perform the update, write back the tile to main RAM, and increment the corresponding entry in T. Such a scheme is implemented in Algorithm 5 with the help of the LOAD and WRITE sub-routines. Further optimizations are possible by being careful about which tiles are swapped in and out of GPU memory and at what times, overlapping computation with memory transfers when possible. Such optimizations generally require to increase the total memory allocated on the GPU, thus decreasing the maximum possible tile-size.

B.1 Experiments with Falkon on 1 GPU

In Table B.1 we show the performance of the Falkon algorithm on all considered datasets for 1 and 2 GPUs side by side. It is clear that larger datasets scale better with more GPUs since the startup cost (mostly taken up by CUDA initialization) and the lower scaling ratio of the preconditioner are amortized.