UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER'S THESIS

Towards fast hybrid deep kernel learning methods

Author:
Miquel LARA

Supervisor:
Lluís A. BELANCHE

A thesis submitted in fulfillment of the requirements
for the degree of Master's Degree in Informatics Engineering

in the

Facultat d'Informàtica de Barcelona

April 15, 2019

"People worry that computers will get too smart and take over the world, but the real problem is that they're too stupid and they've already taken over the world."

Pedro Domingos

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Abstract

Facultat d'Informàtica de Barcelona

Master's Degree in Informatics Engineering

Towards fast hybrid deep kernel learning methods

by Miquel LARA

This work studies the hybridization of neural networks and approximated kernel methods. Different methods of approximating infinite-dimensional kernels are explored here for use within deep neural networks, as are various optimization methods used for training them. The objective is to obtain an optimal performance for the resulting network while reducing training time as much as possible. While other methods cannot yet be discarded as valid approaches from testing done in this thesis, in general the best results were obtained by using random Fourier features with the adaptive optimizer RMSprop.

Acknowledgements

Heartfelt thanks to all the people that have been by my side throughout this journey, and without whom this work would not be possible. A full list with proper acknowledgements would be far too long, but I must thank my mom, dad and brother for putting me up and supporting me over the years, and Carla for putting up with me more than anybody should. Of course special thanks must also go to my supervisor Lluís Belanche, who has inspired my interest in machine learning and without whom this paper would not be possible.

Contents

Chapter 1

Introduction

1.1 Introduction

Artificial intelligence is at the forefront of computer science research, with applications in almost every field and far-reaching consequences in everyday life. One of the major advances in computers' capacity to solve complicated and nuanced problems is in part due to the increasing popularity of deep learning, which allows us to make inferences which historically have been simple for humans to make, but very complicated for computers.

Although the concept of deep learning is some decades old, it is thanks to the advances in hardware and the prevalence of distributed and cloud computing that it has become viable to apply it on a commercial and industrial scale. This has lead to the recent "Big Bang of AI"[1], which allows computers more autonomy and requires less human fine-tuning when analyzing the massive data sets that are being generated and stored.

Despite the processing capacity available nowadays, it is necessary to improve the underlying algorithms in order to ensure that the hardware is used to its full capabilities. Thanks to the creation of open-source machine learning libraries like Tensorflow, and their ability to use consumer GPUs to speed up parallel calculations, it is possible to train and experiment on deep learning networks of moderate sizes even without specialized hardware.

1.2 Problem to Solve

The interest in deep neural networks has surged recently as a result of the high accuracy which deep learning can achieve compared to other statistical models. Historically, neural networks were outpaced in their predictive abilities by support vector machines and other kernel-based methods, although this trend has recently been reversed. Nevertheless, kernel-based methods are well understood from a mathematical point of view, and being able to combine both approaches is an interesting avenue to explore. It is hoped that the high-dimensional mappings of kernel methods can enhance neural networks' predictive power.

At the same time, the increased computational cost of using kernel mappings should be mitigated. One of the possible improvements is to apply derivative-free training methods to the neural network, or to use adaptive algorithms that can adjust each parameter's learning rate based on previous observations, instead of the usual stochastic gradient descent methods. These methods have already proven to be effective in training standard feedforward neural networks, and it is expected that they can similarly aid in training neural networks when hybridized with kernel methods.

1.3 Aims and Objectives

The main aim of this thesis is to train hybrid neural-kernel networks with different kernel approximation and training methods, and to see whether they can improve training times and model accuracy. In particular, the following comparisons will be made:

- Random Fourier features (RFF) vs. the Nyström method (NM) for kernel approximations
- Stochastic gradient descent (SGD) vs. RMSprop vs. simultaneous perturbation stochastic approximation (SPSA) for layerwise training

The hope is that by synthesizing various approaches to creating and training hybrid network architectures, it is possible to evaluate which approaches give the best

results in terms of performance and training time. Despite these ideas having been studied in isolation, no thorough assessment has been undertaken to see how these ideas can enhance neural networks when implemented in unison. It also remains to be seen what the trade-off between training times and final model performance is and whether it is worth it to pursue better performing models, even if it comes at the expense of higher training times.

For simplicity's sake during the implementation and evaluation of the experiments, only classification problems will be considered. Whether or not the results presented here extend to regression problems as well are left for further research.

1.4 Related Work

The main point of reference for this thesis is the work done by Mehrkanoon et al. [2], which hybridizes a single or two-layer neural network with kernel methods. These functions are typically used in support vector machines to apply linear methods in nonlinear data, but as they are data-dependent representations they do not scale well with large amounts of data. Mehrkanoon et al. show that by replacing the activation function of a typical feedforward neural network by an approximated kernel, they can obtain improved performance with large datasets than an equivalent SVM or feedforward network.

Alternative approximations to the kernel function, in particular the Nyström method versus the random Fourier features, have been explored by Yang et al.[3] for both ridge regression and SVMs, which shows a significant improvement when using the Nyström method compared to RFF, although is not clear how these results will carry over when used in a hybrid neural kernel network. They further show that due to the data-dependent nature of the Nyström method, this improvement is dependent on the large eigengap of the kernel matrix generated by the input data, so this result may not generalize easily if the data used does not display this property.

A further variation on current approaches to hybrid kernel networks is using the derivative-free simultaneous perturbation stochastic approximation versus the typical stochastic gradient descent used for training neural networks. These approaches have been explored in the context of standard neural networks by Wulff et al. [4],

but it is not clear how these techniques will hold up when using kernel methods. On the other hand, the widely used adaptive optimization algorithms like Adam or RMSprop have been studied by Wilson et al. [5] and found to perform worse than stochastic gradient descent for the problem sets which were analyzed, so it is interesting to see whether these results also hold for the hybrid networks considered here.

Other types of layerwise training for deep architectures have been explored by Mora [6], who expanded on the deep hybrid architecture proposed by Mehrkanoon et al. by using layerwise training techniques, as well as applying several regularization methods to prevent the overfitting. Regularization techniques are not widely studied here, but some basic techniques will be used to prevent overfitting the data.

1.5 Thesis Structure

Chapter 2 gives the background knowledge of neural networks and kernel methods, as well as the kernel approximation and training methods which will be compared. The hybrid neural kernel architecture which the experiments will be based on are introduced in Chapter 3, as well as presenting the current state of the art for these types of architectures, kernel approximations and training methods. Chapter 4 presents the methodology which will be used for the experiments, as well as introducing the frameworks used when implementing them, and Chapter 5 gives an overview of the experimental setup and introduces the datasets which the experiments are performed on. The results obtained are presented and discussed in Chapter 6, while Chapter 7 summarizes the conclusions reached and outlines some possible extensions to the research carried out here.

Chapter 2

Neural Networks and Kernel Methods

2.1 Deep Neural Networks

In order to understand what changes have been made to standard neural networks in this hybrid approach, it is first necessary to understand what is meant by a "deep neural network". A neural network is a specific type of statistical model, with the goal of approximating some unknown function f^*, although the approach here is somewhat different to that of a regression or other linear models. This paper will only deal with feedforward networks, which do not have any loops or cycles in the network.

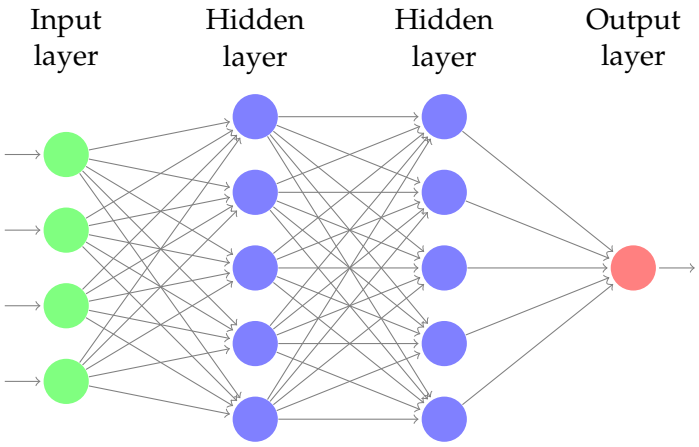
The term "network" refers to the fact that the model is made up of intermediate functions $f^{(i)}$, for $i \leq N$, which are composed together in some way. The most common way of doing this is through a simple chain, eg. $f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$, where each of the functions is referred to as a **layer**. With this representation, $f^{(1)}$ is the first layer, and the outermost function $f^{(N)}$ being referred to as the **output layer**. The output of any layer before the final layer is not used when training the model, and so they are referred to as **hidden** layers.

The number of layers is the **depth** of the model, and while "deep" is somewhat subjective, it typically refers to any network where there is at least one hidden layer between the input layer and the output. The ever-increasing depth of a neural network is what gives birth to the phrase "deep learning". [7]

Finally, the "neural" refers to the fact that neural networks are inspired by the

functioning of the brain, where each input value roughly corresponds to a single neuron, and each successive neuron receives input from the neurons in the previous layers. Although neural circuits in the brain have been used as an inspiration for neural networks, progress in this field is driven mainly by mathematical and engineering principles, and not biological ones. A graphical representation of a neural network can be seen in Figure 2.1.

FIGURE 2.1: Example of a simple ANN[8]



The "classical" approach to an artificial neural network is for each hidden layer $f^{(i)}$ to be formulated as follows:

$$f^{(i)}(\mathbf{x}) = g(\mathbf{W_i x} + \mathbf{b_i})$$

where $\mathbf{W_i}$ is the weight matrix, $\mathbf{b_i}$ is a bias vector, and g is an **activation function**[2]. Each of the rows of $\mathbf{W}$ represent an input, with each column representing a single neuron, so the value at $\mathbf{W}_{ij}$ is the weight for the input i in neuron j.

The purpose of the activation function is to add some nonlinear component to the layer, so that the neural network as a whole is not simply a linear model. This general approach allows an artificial neural network to theoretically approximate a wide array of functions, with only some loose restrictions on the activation functions. Assuming that g is bounded, nonconstant and continuous, then this feed-forward network architecture can approximate any continuous function arbitrarily well, depending on the depth and width of the network. [9]

A standard recommendation for the activation function is the rectified linear unit (ReLU) function, given by $g(x) = \max(0, x)$[7], but in this paper, we will extend on

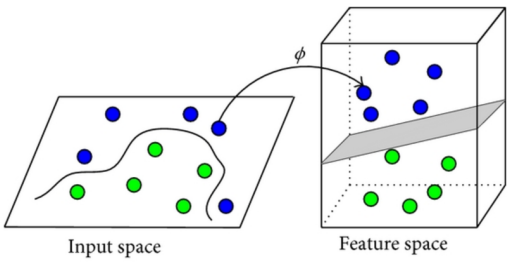
the approach taken by Mehrkanoon et al.[2], which essentially replaces the activation function with a kernel function, as described in Section 3.1.

2.2 Kernel Methods

Machine learning and statistical methods are well understood and optimized in the case of linearly separable data, but real data is often not so cleanly structured and requires the use of nonlinear methods. Thanks to kernel functions, linear statistical methods can be applied to nonlinear data, while maintaining the performance and robust theoretical background of linear methods.[10] This is done by projecting the data into a different space, which is normally of a higher (potentially infinite) dimension than the original space and where the data can be linearly separated. This is visually represented in Figure 2.2.

Since linear statistical methods often only require the calculation of inner products, if these inner products can be calculated within the target space without having to work explicitly within this space, it is possible to obtain rich representations of the data at a fraction of the computational cost. This is what is referred to as the "kernel trick".

FIGURE 2.2: Visual representation of kernel mapping into a higher dimensional space[11]



Consider a function $\phi : X \to H$, which maps from our original input space X into some new inner product space H, ie. a vector space with a valid inner product $\langle ., . \rangle_H$, then ϕ is said to be the **feature map** for the **feature space** H. The corresponding **kernel** $k : X \times X \to \mathbb{R}$ is given by:

$$k(\mathbf{x}, \mathbf{y}) = \langle \phi(\mathbf{x}), \phi(\mathbf{y}) \rangle_H$$

This kernel is essentially a similarity measure between two observations. [12] This and other similarity measures can in some sense be considered to be the inverse of a distance measure - the value for a similarity measure will increase when two observations in the mapped feature space are a small distance from one another. [13]

Part of the power of kernel functions is that instead of explicitly expanding the feature space with general functions of existing features, it is only necessary to calculate the similarity between previous observations. [12] In this sense, kernels can be considered to be instance-based learning methods - that is, they learn based only on the particular observations which they have already seen, and not based on mathematical generalizations. It is because of this fact, and the efficiency of the **kernel trick**, that they are able to perform computations which would be very difficult or impossible to generalize.

The kernel trick relies on the fact that linear statistical analysis often does not require explicit calculations in the feature space H, as long as we know that k is a valid kernel for *some* feature space, and k can be efficiently calculated. Whether k is valid can be verified by looking at the **Gram matrix** of k, with respect to some inputs $\mathbf{x_1}, ..., \mathbf{x_n} \in X$, defined as:

$$\mathbf{G_{ij}} = k(\mathbf{x_i}, \mathbf{x_j})$$

It is possible to show [10] that a function k is a valid kernel if and only if it generates a positive semi-definite Gram matrix, ie. if $\forall \mathbf{x} \in \mathbb{R}^n, \mathbf{x^T G x} \geq 0$. Therefore, it is possible to use the Gram matrix to generate valid kernels.

2.2.1 Approximating Kernel Functions

The Gram matrix relies on the observations that the model has already been seen, so its size will grow quadratically with the number of training examples. Since large training sets are ubiquitous in modern machine learning, it is often necessary to approximate either the Gram matrix itself, or to approximate the kernel function via some method. This paper will consider random Fourier features and the Nyström

method to approximate the kernel function in the hybrid neural network, which are discussed in the next sections.

Random Fourier Features

Random Fourier features are an example of a random feature mapping, a family of algorithms that use stochastic methods to generate a lower-dimensional approximation for a target kernel function k. Since this dimension does not necessarily need to grow proportionally with the number of observations, it is possible to obtain much better training performance even with large datasets, thus enabling the use of kernels for "big data". [14]

Consider $X = \mathbb{R}^d$, that is, our observations are real-valued vectors of dimension d, and we wish to approximate some shift-invariant kernel [1] k with a corresponding feature map ϕ and using D random features. Then a **randomized feature map** for this kernel is a function $z : \mathbb{R}^d \to \mathbb{R}^D$ which satisfies:

$$k(\mathbf{x}, \mathbf{y}) = \langle \phi(\mathbf{x}), \phi(\mathbf{y}) \rangle \approx z(\mathbf{x})^T z(\mathbf{y})$$

There are different types of randomized feature maps[15], but this paper will only consider **random Fourier features (RFF)**. An RFF map can be generated for a kernel k with Algorithm 1.

Algorithm 1: Generating a RFF mapping [15]

Input: k a positive definite, shift-invariant kernel, $D \in \mathbb{N}$ number of RFF

Output: A RFF map $z : \mathbb{R}^d \to \mathbb{R}^D$ such that $k(\mathbf{x}, \mathbf{y}) \approx z(\mathbf{x})^T z(\mathbf{y})$

begin

- Compute the Fourier transform p of k
- Take D samples $\boldsymbol{\omega_1}, ..., \boldsymbol{\omega_D} \in \mathbb{R}^D$ from p
- Take D samples $\mathbf{b_1}, ..., \mathbf{b_D}$ from $U(0, 2\pi)$
- Define $z_i(\mathbf{x}) = \cos(\omega_i{}^T \mathbf{x} + b_i)$
- Define $z(\mathbf{x}) = \sqrt{\frac{2}{D}}(z_1(\mathbf{x}), ..., z_D(\mathbf{x}))$

[1] k is shift-invariant if and only if $k(x, y) = g(x - y)$ for some positive definite function g.

A formal proof of the validity of this algorithm is beyond the scope of this paper, but can be seen in the work of Rahimi et al.[15]. However, it is possible to make the following observations which justify this approach:

- It is valid to interpret p as a probability function from which ω_i are taken because of Bochner's theorem, which guarantees that for a positive definite kernel, its Fourier transform (when properly scaled) is a probability measure.[14]
- Additionally, the fact that $E[z_i(\mathbf{x})^T z_i(\mathbf{y})] = k(\mathbf{x}, \mathbf{y})$ means that $z(\mathbf{x})$ is indeed a valid randomized feature map for k, and due to Hoeffding's inequality this will converge exponentially fast with D. [15]
- The higher the value of D, the closer the RFF approximation will be to the true value of k.

Nyström Method

Unlike random Fourier features, which seeks to approximate the kernel function directly, the Nyström method seeks instead to approximate the Gram matrix itself with a lower-rank matrix. Whilst RFF uses a data-independent probability distribution to obtain samples, the Nyström method picks some of the training examples in order to generate a lower-rank approximation. In this sense, the Nyström method can be said to be data-dependent, while the RFF approach is not.

The Nyström method was originally used for kernel machines by Williams and Seeger [16], where the approximation of the Gram matrix $\mathbf{G}$ was done by randomly selecting, without replacement, a subset of of rows or columns from $\mathbf{G}$, and using these to construct an approximation $\widetilde{\mathbf{G}}$. Further generalization and formalization was carried out by Drineas and Mahoney[17]. They present two algorithms, one of them being a more constrained version of their more general algorithm.

The "Preliminary Approximation" is essentially the original Williams and Seeger algorithm, with the main changes being that the columns are chosen with replacement and using a generalized Moore-Penrose pseudoinverse ($\mathbf{W}^{+}$) instead of a standard inverse matrix ($\mathbf{W}^{-1}$). This is described in Algorithm 2.

Algorithm 2: Approximating a Gram matrix using the Nyström method [17]

Input: $\mathbf{G}$ an $n \times n$ Gram matrix, $c \leq n$

Output: $\widetilde{\mathbf{G}}$ an $n \times n$ approximation of $\mathbf{G}$

begin

Choose c columns of $\mathbf{G}$ according to a discrete uniform distribution with replacement, let I be the set of indices of the sampled columns

Define the $n \times c$ matrix $\mathbf{C}$ which is formed of the chosen columns

Define the $c \times c$ matrix $\mathbf{W}$, formed by $\mathbf{G_{ij}}$, $\forall i, j \in I$

Define $\widetilde{\mathbf{G}} = \mathbf{C}\mathbf{W}^{+}\mathbf{C}^{\mathbf{T}}$

Drineas and Mahoney generalize Algorithm 2 into a "Main Approximation" algorithm. [17]. This will not be taken into consideration here although it may be an interesting avenue for further investigation, since it allows for the following generalizations:

- The probability distribution used to choose the columns are not restricted to the uniform distribution.
- The matrices used are scaled according to the probability distribution used.
- The rank of $\widetilde{\mathbf{G}}$ is parameterized separately from the number of chosen samples from $\mathbf{G}$.

The "Preliminary Approximation" algorithm is compared by Yang et al. with Random Fourier Features for ridge regression and SVM classification tasks, and obtained positive results when compared to RFF [3]. However, as the Nyström method is data-dependent, the generalization error is dependent on the Gram matrix of the kernel. When the Gram matrix has a large eigenspectrum ie. its eigenvalues have large gaps between them, then Nyström is shown to have significantly better generalization error than random Fourier features.

However, the Nyström method may require longer calculation times due to the larger amount of operations required to obtain the approximated Gram matrix. The calculation of the pseudoinverse is especially problematic, as it normally done via the singular-value decomposition of the matrix, which for an $m \times n$ matrix is in the worst case a $O(m^2 n + n^3)$ problem.[18]

2.2.2 Radial Basis Function Kernels

Before a discussion of the hybrid architecture can be carried out, it is first necessary to introduce the kernel function which will be used in this paper - the radial basis function (RBF) kernel, sometimes also referred to as the Gaussian kernel.

The RBF kernel can be defined as: [12][3]

$$k(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{||\mathbf{x} - \mathbf{y}||^2}{2\sigma^2}\right)$$

Or equivalently:

$$k(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{1}{2\sigma^2}\sum_{i=1}^{d}(x_i - y_i)^2\right)$$

where the definition $\gamma = \frac{1}{2\sigma^2}$ is often made in order to simplify the expression for $k(\mathbf{x}, \mathbf{y})$:

$$k(\mathbf{x}, \mathbf{y}) = \exp(-\gamma||\mathbf{x} - \mathbf{y}||^2)$$

When using this kernel, the Fourier transform for this kernel is the normal distribution. In particular, the Fourier transform $p(x) = \mathcal{N}(0, \sigma^2\mathbf{I})$, where $\mathbf{I}$ is the identity matrix. This means that in order to generate RFF for the RBF Kernel, the ω_i will be sampled from a normal distribution.

The RBF kernel is often used in SVM and other machine learning algorithms, partly due to a single hyperparameter which must be optimized. Because of this, it is often easier to find good models when compared to kernels with higher counts of hyperparameters - as an example, the polynomial kernel $k(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^\mathbf{T}\mathbf{y} + \mathbf{c})^d$ requires optimizing 2 hyperparameters. RBF also performs well in experimental results in part due to its flexibility in finding nonlinear relations, since it maps into an infinite-dimensional space.

2.2.3 Dimension of the RBF

The following section aims to both help the reader to see that the RBF indeed a valid kernel, and that the inner product space which it projects into is infinite dimensional. For this explanation, it will be assumed that $\gamma = 1$ and that we are dealing with scalars, without any loss in generality.

Taking the definition for k above, we get that:

$$k(x, y) = \exp(-x^2)\exp(-y^2)\exp(2xy)$$

By replacing the last term with its Taylor series [19]:

$$\begin{aligned} k(x, y) &= \exp(-x^2)\exp(-y^2)\sum_{k=0}^{\infty}\frac{2^k x^k y^k}{k!} \\ &= \exp(-x^2)\exp(-y^2)\sum_{k=0}^{\infty}\left(\sqrt{\frac{2^k}{k!}}x^k\right)\left(\sqrt{\frac{2^k}{k!}}y^k\right) \end{aligned} \tag{2.1}$$

As this expression is separable into x and y, it is clear that this can indeed be written as $k(x, y) = \langle\phi(x), \phi(y)\rangle_H)$, where ϕ will be a mapping into an infinite-dimensional space. Specifically:

$$\phi(x) = \exp(-x^2)(1, \sqrt{2}, \sqrt{2}, \sqrt{4/3}, ..., \sqrt{2^k/k!}, ...)$$

This helps to explain some of the effectiveness of the RBF kernel, as this is essentially a mapping into an infinite-dimensional space. Although an explicit mapping will not be used in practice, the fact that RFF and the Nyström approximate this kernel means that they should be able to find highly non-linear relationships in the data. It should be noted that in general, models using the RBF are shown to be very sensible to over- or under-fitting with different values of γ. In order to prevent this, γ must be optimized via hyperparameter search.

2.3 Training Neural Networks

Before a neural network can be used to make predictions on the input, it must first be trained. Training a neural network is a specific case of an **optimization problem** - that is, minimizing or maximizing the output of some function $f(x)$ based on its input x. In the case of a neural network, we have some loss function defined in terms of the model parameters $\boldsymbol{\theta}$, with a set of n training points $\mathbf{X}$ and labels $\mathbf{Y}$ [2]:

$$J(\boldsymbol{\theta}, \mathbf{X}, \mathbf{Y}) = \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{x_i}, y_i)$$

The loss represents how close the observations of the neural network are to the known labels y_i. This will be different depending on the problem being solved - a standard loss for regression problems is the mean squared error (MSE), while in classification problems such as those treated in this paper it is common to use the negative log-likelihood of the softmax function. This loss is given by:

$$L(\mathbf{x}_i, y_i) = -\log \Pr(y_i|\mathbf{x_i}; \boldsymbol{\theta})$$

Often, an additional regularization term is added to the cost function to penalize the magnitude of the weights in the model parameters $\boldsymbol{\theta}$, as a measure to prevent overfitting. The cost function which would be minimized would then be:

$$J(\boldsymbol{\theta}, \mathbf{X}, \mathbf{Y}) = \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{x_i}, y_i) + \alpha \boldsymbol{\Omega}(\boldsymbol{\theta})$$

This loss can take several forms, but this paper will use the L^2 regularizer, which takes the following form[7]:

$$\boldsymbol{\Omega}(\boldsymbol{\theta}) = ||\mathbf{w}||_2^2$$

Having defined the cost function to optimize, two optimization methods will be compared in this paper. A standard approach in neural networks is to use **stochastic gradient descent** (SGD), with the original aim being to compare it against **simultaneous perturbation stochastic approximation**. Unfortunately, due to replication problems covered in section 4.2.1 with SPSA, it was not possible to optimize any of the networks outlined here using the layerwise SPSA. However, the **RMSprop** algorithm will also be compared against SGD, to evaluate their performance with hybrid networks.

2.3.1 Stochastic Gradient Descent (SGD)

SGD is an adaptation from the gradient descent algorithm which is often used for convex optimization problems. Gradient descent relies on using the derivative of a multivariate function to find the direction of steepest descent of some function with respect to its inputs, and shifting the parameters in that direction by some particular distance. However, gradient descent is too computationally expensive to carry out when applied to machine learning or neural networks.

For example, given the loss function for a neural network defined above, the gradient descent algorithm requires calculating the following:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}, (X), (Y)) = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} L(\mathbf{x_i}, y_i)$$

As this grows with the size n of the dataset being considered, this is not scalable to large datasets.[7] In order to avoid this, we perform gradient descent on a smaller **minibatch** of data of size $m << n$, selected randomly without replacement from the original dataset. Because the minibatch is being selected from the entirety of the data set, this will have the same mean as calculating the entire gradient, but at a fraction of the computational cost. As the squared standard error is proportional to the size of the dataset, a 100-fold increase in computation time with a larger dataset will only reduce the error by a factor of 10.

This algorithm gives an approximation of the gradient:

$$\mathbf{g} = \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(\mathbf{x_i}, y_i)$$

The values of $\boldsymbol{\theta}$ are adjusted by moving downhill in this direction, in other words the parameters are adjusted as follows:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \mathbf{g}$$

where ϵ is the so-called **learning rate** of the SGD algorithm. In practice, this value is normally reduced with the number of elapsed epochs, as the random sampling of the SGD algorithm introduces a source of noise which will not disappear even as a

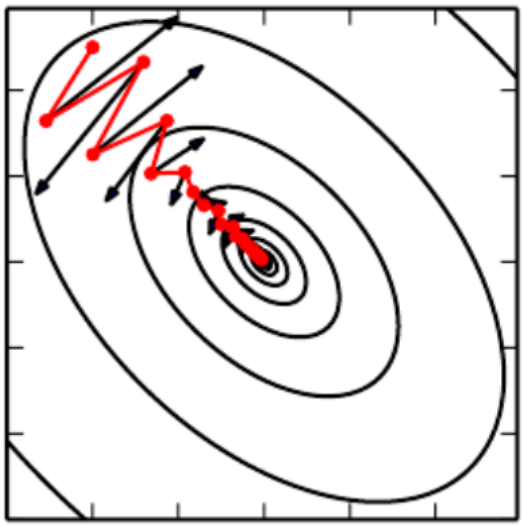
minimum is approached and which will prevent minima from being reached. ϵ is often defined as:

$$\epsilon(t) = \frac{\epsilon}{1 + kt}$$

where the value k is referred to as the **learning rate decay factor** and t is the number of epochs. This will reduce the random oscillations as time decreases, allowing the algorithm to reach minima more efficiently.

Finally, a further improvement to the standard SGD algorithm is the concept of **momentum**. This is analogous to the concept of momentum in physics, where an object with mass will tend to continue moving in the same direction. Similarly, momentum in SGD will accumulate past gradients and continue to move in their direction. Using momentum can help reduce the variance in updating the network weights, since it will tend to move in directions which have in the past been proven to be more advantageous. This can be seen in Figure 2.3.

FIGURE 2.3: Effect of momentum (red) when traversing a solution space [7]



The magnitude of the past gradients can be controlled by a factor $\alpha \in (0, 1)$, which determines how quickly the contributions of the previous gradients decay. Instead of setting $\boldsymbol{\theta}$ based on the gradient, momentum will determine its value based on a "velocity" that will then be used to update the argument values:

$$\mathbf{v} \leftarrow \alpha\mathbf{v} - \epsilon\nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(x_i, y_i) \right)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}$$

2.3.2 RMSprop

Adaptive gradient methods, such as Adam, RMSprop or are similar to stochastic gradient descent, but with the crucial difference that the learning rate is adapted for each parameter instead of being global. They modify the learning rate based on the previous gradients which have been calculated, and use the partial derivatives of the loss with respect to each parameter to determine what its individual learning rate should be.[7] If the signs are the same, then the parameter learning rate will increase, otherwise it will decrease as it will have passed a local minimum.

The RMSprop algorithm is an unpublished adaptive algorithm, first presented by Hinton in an online course. [20] Despite this, it is widely implemented and used due to the robustness of its hyperparameters and its empirical efficacy. This algorithm is presented in Algorithm 3. Note that $\odot$ represents elementwise matrix multiplication.

Algorithm 3: RMSprop Algorithm [7]

Input: loss function L, global learning rate ϵ, decay rate ρ, initial parameters $\boldsymbol{\theta}$, small constant δ (normally 10^{-6}),

while *stopping criteria not met* **do**

- select m random samples $\{\mathbf{x^1}, ..., \mathbf{x^m}\}$, with corresponding targets $\mathbf{y^i}$
- compute the gradient $\mathbf{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(\mathbf{x_i}, \mathbf{y_i})$
- accumulate the squared gradient $\mathbf{r} \leftarrow \rho\mathbf{r} + (1-\rho)\mathbf{g} \odot \mathbf{g}$
- compute parameter update $\triangle\boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\sigma+\mathbf{r}}} \odot \mathbf{g}$
- update parameters $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \triangle\boldsymbol{\theta}$

Although adaptive methods are very popular, there has been some question as to how useful they actually are. In particular, Wilson et Al. [5] showed in their testing that standard SGD showed generalized better to the testing set than RMSprop

and other adaptive methods, even when their training performance was similar. It remains to be seen whether this is the case with the hybrid networks used here, and how well RMSprop performs in both model performance and training time.

2.3.3 Simultaneous Perturbation Stochastic Approximation (SPSA)

SPSA is an optimization algorithm which only relies on observations of the objective function, unlike SGD which requires evaluation of the derivatives of the function. It does this by perturbing all of the vector elements at once in random directions, and uses an estimate of the gradient which relies only on two point measurements. This means that it can make gradient adjustments without knowledge of the underlying model, even if it is non-differentiable.

SPSA relies on the fact that for a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$, its gradient ∇f can be defined by its components:

$$\nabla f_i(\mathbf{x}) = \lim_{\epsilon \to \infty} \frac{f(x_i + \epsilon) - f(x_i - \epsilon)}{2\epsilon}$$

This can be estimated by taking some random perturbation vector δ, and defining:

$$\hat{g}_i(\mathbf{x}) = \frac{f(x_i + \delta_i) - f(x_i - \delta_i)}{2\delta_i}$$

The SPSA algorithm then uses this to compute a pair of perturbations $\mathbf{x}^+$ and $\mathbf{x}^-$ based on the current best solution $\mathbf{x_t}$.[4] Based on these points, the algorithm calculates an approximation of the gradient and descends along this gradient.

Similarly to SGD, there are several scaling factors which reduce the size of the perturbations as the number of epochs increases. The role of these scaling factors can be seen in Algorithm 4.

Algorithm 4: SPSA to find $x^* = \arg\min_x f(x)$ [4]

Input: function f, initial guess $\mathbf{x_0}$, factors $a > 0$, $\alpha \geq 1$ and $c > 0$, $\gamma \in \left[\frac{1}{6}, \frac{1}{2}\right]$

Output: $\mathbf{x}_{t_{max}}$ which approaches x^*

for $t = 1, ..., t_{max}$ **do**

- set $a_t = \frac{a}{t^\alpha}$
- set $c_t = \frac{c}{t^\gamma}$
- sample δ where $\delta_i \sim U(-1, 1)$
- set $\mathbf{x}^+ = x_t + c_t\delta$
- set $\mathbf{x}^- = x_t - c_t\delta$
- take an approximation $\hat{g}$ of the gradient, $\hat{g}_i(x_t) = \frac{f(x_i^+) - f(x_i^-)}{2c_t\delta_i}$
- take $x_{t+1} = x_t - a_t\hat{g}(x_t)$

Although it is possible to apply this to the cost function of a neural network, as has been shown by Song et al. [21], this approach may suffer from the curse of dimensionality when large numbers of parameters are considered. In order to mitigate this, a layerwise approach should be taken when using SPSA, as discussed in Section 3.1.1.

Chapter 3

State of the Art

3.1 Hybrid Neural Kernel Networks

Given the good nonlinear separation qualities which the RBF kernel offers, and the fact that the various kernel approximations can be used to apply it to larger datasets, it becomes possible to formulate architectures which use both the depth of a neural network and the richness of higher-dimensional mappings via kernels. However, as kernel functions require higher computational times than other activation functions, it will be critical to also reduce the training time for these hybrid neural networks so that these networks can be trained efficiently. This will be done by comparing various optimization methods, and seeing how they compare both in performance and training time.

The reference network architecture which will be the basis for most of the work in this thesis is the deep hybrid neural-kernel networks introduced by Mehrkanoon et al. [2]. The fundamental change made by Mehrkanoon et al. is to a standard feed-forward network is to change the activation function in each of the hidden layers of the neural network with a feature map corresponding to a particular kernel function.

Looking at each layer of the neural network, defined as:

$$f(\mathbf{x}) = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{3.1}$$

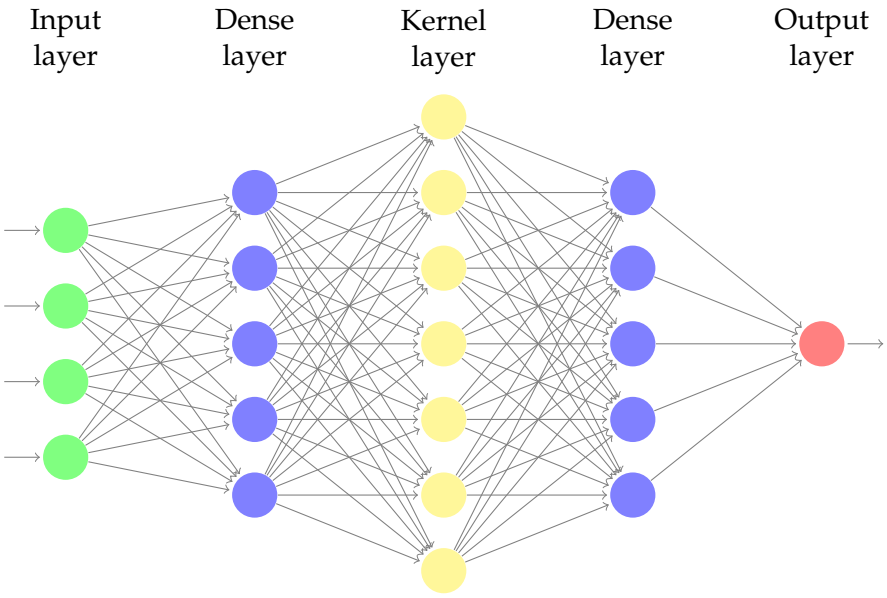
g is be taken to be some feature map ϕ, instead of the usual nonlinear activation functions used in neural networks such as the ReLu, sigmoid or hyperbolic tangent functions. This feature map will correspond to some kernel function k, given by $k(\mathbf{x}, \mathbf{y}) = \langle \phi(\mathbf{x}), \phi(\mathbf{y}) \rangle_H$, as outlined in Section 2.2. In practice, instead of using the

full feature map ϕ, Mehrkanoon et al. use the random Fourier feature mapping $z(\mathbf{x})$ described in 2.2.1.

The output layer is not changed so that the deep hybrid kernel network still outputs a valid value, whether with a softmax function in the case of classification problems or no activation function in the case of a regression.

The architecture defined by Mehrkanoon et al. in the shallow case can be observed in Figure 3.1. They also define a deep architecture, which stacks 2 fully connected layers interleaved with 2 kernel layers, as in 3.2.

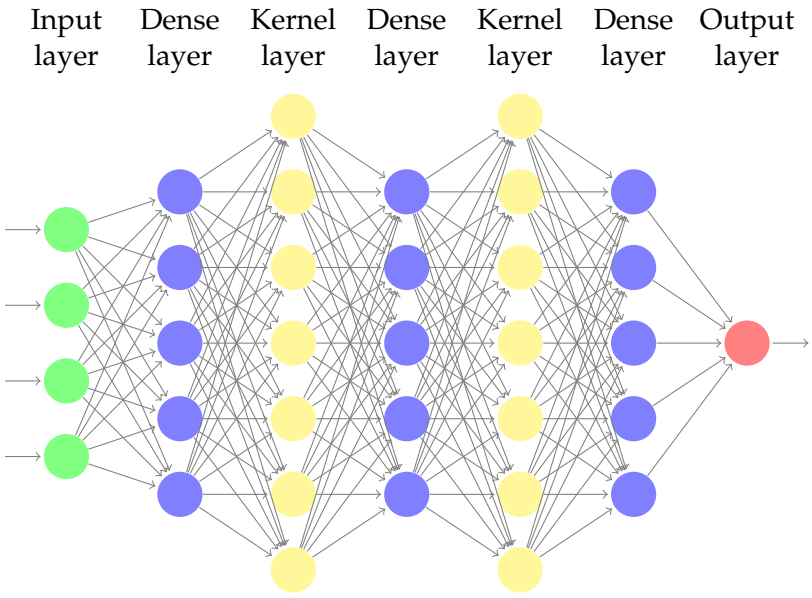
FIGURE 3.1: Example of a shallow hybrid neural kernel network[2]



The deep architecture seen in 3.2 is trained based on the previous weights of the shallow model. More specifically, the weights of the first dense layer are kept fixed and taken from the best result obtained by the shallow model, and the last dense layer is trained separately. In this way, it can be thought of as being semi-layerwise, in the sense that the whole model is not trained via backpropagation.

The use of random Fourier features for machine learning is explored by Rahimi and Recht [15] for use in support vector machines and ridge regressions, where they approximate the full kernel using random Fourier features. They showed that even though the data is not mapped into the same dimensional space as in a full-scale kernel machine, it can still obtain favorable performance for both classification and

FIGURE 3.2: Example of a deep hybrid neural kernel network



regression tasks. Furthermore, Yang et al. compared the Nyström method and random Fourier for ridge regression and support vector machine classification, and concluded that Nyström is, for all the explored data sets, superior to random Fourier features as a non-linear mapping for these algorithms, although this is dependant on the range of the eigenvalues of the Gram matrix generated by the data. The hope is that these results will carry over to neural networks, and will enable us to enhance the hybrid neural-kernel network with a higher performance.

The Nyström method in deep learning has also been explored for convolutional neural networks by Luc et al. [22], with the additional change that the Nyström representation is learned as part of the network instead of being calculated previously. This has the advantage of reducing the number of computations required to find the approximated Gram matrix, as otherwise it requires calculation of a matrix pseudoinverse, which does not scale well with higher-dimensional feature approximations.

3.1.1 Training Neural Network Architectures

Mehrkanoon et al. introduce a semi-layerwise approach using stochastic gradient descent to train neural networks, which serves the purpose of reducing the number

of variables which are being trained at once, compared to training the whole deep network at the same time.

Wulff et al.[4] proposed using SPSA in a layerwise fashion to train neural networks. Their approach uses the SPSA algorithm described in section 2.3.3 to optimize cost function of the neural network, but keeping fixed any weights which are not part of the current layer being trained. The whole algorithm can be seen in Algorithm 5, where it can be observed that each layer of the neural network is perturbed within a single epoch. The order which this is done in does not seem to have an effect on the training time or its efficacy. This algorithm was found to converge more quickly than SGD for the datasets explored, but SGD ultimately obtained better performance with a larger number of epochs.

Algorithm 5: Layerwise SPSA for neural networks [4]

Input: Neural network with layers L, cost function J

for $e = 1, ..., e_{max}$ **do**

  for $l \in L$ **do**

    Keep weight matrices $\mathbf{W_e^{k \neq l}}$ fixed

    $\mathbf{W_{e+1}^l} = SPSA(J, \mathbf{W_e^l})$

Wilson et al. [5] compared RMSprop, Adam and other adaptive training algorithms with stochastic gradient descent, and found that adaptive algorithms in general did not generalize as well as SGD. Although they trained more quickly at the beginning, it was possible to obtain better testing performance using SGD, through a thorough tuning of the hyperparameters.

Chapter 4

Methodology

4.1 Approach and Objectives

The aim of this thesis is to combine and synthesize the various approaches to hybrid neural networks and the training of deep networks which have been discussed in the previous section. To this purpose, the following comparisons will be made to see how they affect the performance and training time of the proposed neural networks:

1. Random Fourier features vs Nyström method
2. SGD vs RMS vs SPSA

There are three main metrics that we will be concerned with and aiming to optimize:

1. Maximizing test accuracy
2. Minimizing test loss
3. Minimizing training time

The progress over the epochs of training of will also be compared in order to observe the speed with which each algorithm converges. The network which will be evaluated has layout described in Figure 3.2, but the kernel layers will use either the Nyström approximation or the random Fourier features used in the authors' original study.

Due to the high complexity of performing the Nyström method with large feature sizes, the kernel layers will not use the Nyström method directly, but will instead use the approach taken by Giffon et al. [22] This takes the required matrix W^+

as a weight to be trained instead of being derived directly from the columns sampled in the Nyström method. This reasons which complicate the calculation of the Nyström method and the reasons for this approach are further discussed in 4.2.

In the case of the SGD and RMSprop algorithm, the same approach will be taken - namely, the shallow network shown in Figure 3.1 will be optimized, the weight of the first layers will be fixed, and the deeper network will be optimized by varying only the values of the later layers. The SPSA approach is to perform layerwise training as outlined in Algorithm 5, although as is discussed in section 4.2.1, it was not possible to replicate the results obtained by the SPSA algorithm even with the original authors' source code, and it has therefore been left out of the final comparison.

4.1.1 Hyperparameters

There are several parameters which are not tuned within the model itself, but which nevertheless have an impact on its performance. Some of these hyperparameters only apply to some variations of the model being considered, while others are universal. In order to optimize these hyperparameters, it is necessary to generate different models, train them, and evaluate which values generate the best models. The details of the methods used to tune these hyperparameters are discussed in Section 5.3.

The list of hyperparameters which need to be tuned is as follows:

Global hyperparameters:

1. L^2 regularization factor
2. Dimension of the densely connected layers
3. Dimension of the approximated kernels
4. γ parameter for the RBF kernel

RMSProp hyperparameters:

1. Learning rate
2. Learning rate decay factor
3. Fuzz factor

SGD hyperparameters:

1. Learning rate
2. Learning rate decay factor
3. Momentum factor

SPSA hyperparameters:

1. Perturbation magnitude factors a and α
2. Learning rate factors c and α

4.2 Implementation in Tensorflow/Keras

The implementation for all the experiments of this thesis has been done in Tensorflow, using Keras as a high-level API when possible and implementing the finer details using the Tensorflow Python API. At the time of writing, Tensorflow 2.0 is still in alpha, but this new version of will streamline and unify much of the code and uses Keras as its central API [23]. Keras allows us to use higher-level abstractions when possible, but still allows finer tuning in its Tensorflow backend when necessary. Keras can also other backends, such as Theano or CNTK, but these are not considered here.

Internally, Tensorflow does not immediately calculate the values of the inputs, but rather builds a computational graph which the data is fed into at runtime. This graph consists of of edges that represent tensors that the data is fed into, and nodes that represent the operations on those tensors. This model allows it to perform automatic differentiation, as well as visualizing the operations which are being performed by using Tensorboard, a built-in dashboard that allows the monitoring and debugging of Tensorflow models. Tensorflow also has the option to work with GPUs to accelerate learning models, which due to the parallel nature of the calculations involved is orders of magnitude faster than using the CPU.

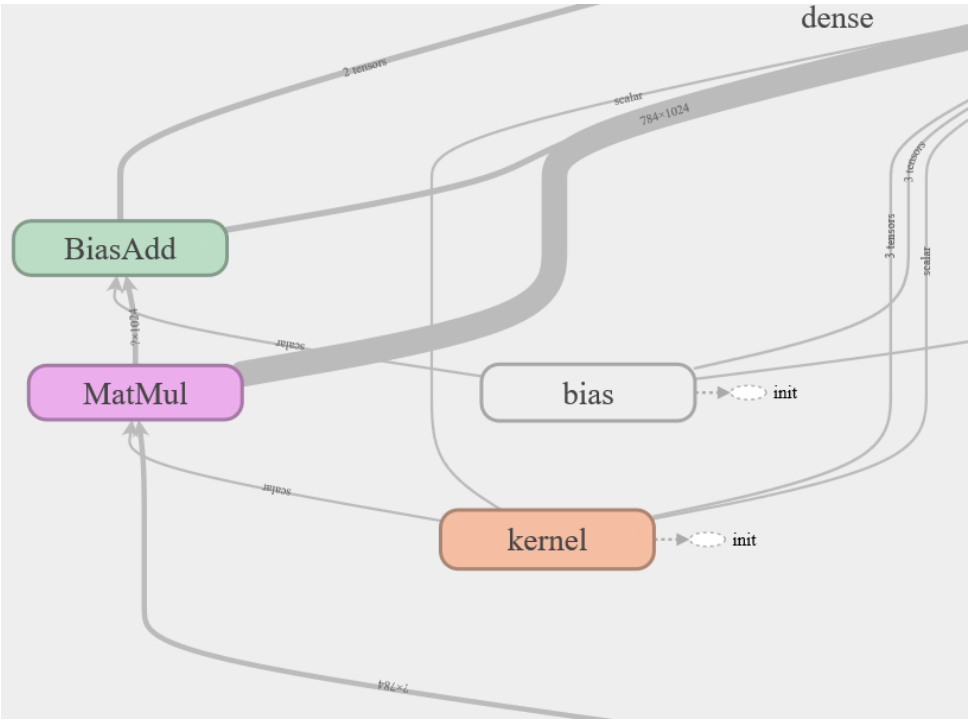
The principal abstraction used with Keras is that of a Model, which consists of several Layers. A layer in Keras is an object which can hold different weights (both

trainable and non-trainable), and which is called as part of a model to process the input tensors.[24] There are several built in layers, such as:

- `Dense` - a fully connected layer, equivalent to $f(\mathbf{W}\mathbf{x} + \mathbf{b})$, with an optional activation function f.
- `Activation` - a layer which can apply an activation function to its input, such as the Rectified Linear Unit (ReLU) or the softmax function.
- `Conv2D` - a convolutional layer for images.

Keras creates the required constructs in its backend in order to build these layers. In the case of Tensorflow, it will create a series of nodes and tensors which flow between them. An example of a fully connected `Dense` layer, with the corresponding weight matrix (referred to here as *kernel*) and bias can be seen in Figure 4.1. Here, the input tensor edges are fed into a MatMul node with the layer kernel, and the bias is then added by the BiasAdd node.

FIGURE 4.1: A `Dense` layer representation in Tensorflow



In order to implement the required operations, it was required to build the following additional Keras layers:

- `RffKernelLayer` - a layer which approximates the RBF kernel by using random Fourier features, with a configurable value for γ and the number of features to be generated. The random tensors are sampled at build time.

- `NystromKernelLayer` - a layer which approximates the RBF kernel via the Nyström method. It requires a Keras callback to periodically resample the training data used for the estimation, but it has not been used in practice due to its reliance on a matrix pseudoinverse.

- `PseudoNystromKernelLayer` - a layer which approximates the RBF kernel via the Nyström method, but which does not generate the W^+ matrix using the pseudoinverse. Instead, this is considered to be another weight which is then optimized as part of the entire model.

The Nyström method, as outlined in Algorithm 2, relies on the calculation of the Moore-Penrose pseudoinverse of the reduced-dimension Gram matrix. This is a computationally expensive operation, which normally is derived from the singular value decomposition of the matrix. This operation is $O(n^3)$ with the size of its input, which severely limits the ability to scale to large output dimensions. Additionally, it is a difficult operation to effectively paralellize, to the point where the current implementation in Tensorflow is slower than the Numpy implementation, which is executed on the CPU. [25] On the system used for testing, it frequently caused slowdowns even when using native Tensorflow functions to compute the inverses for modestly sized feature matrices (<150 features), and values of more than 200 features would frequently cause out of memory errors. For this reason, the `NystromKernelLayer` layer is not used in our testing.

4.2.1 Replicating SPSA Results

During testing, it was not possible to replicate the layerwise SPSA results obtained by Wulff et al. [4]. Despite having the source code for the Tensorflow optimizer available, it was found to have incompatibilities with the most recent version of Tensorflow before the code would run without error, and the example provided with the code was not truly layerwise as it only contained a single layer. Any attempts to

add further layers did not converge to any degree when training, even when these were held fixed and later trained.

The attempts to refactor the code such that it worked for newer versions and with multiple layers were not successful, in large part due to the deeply ingrained functionality within Tensorflow that relies on automatic differentiation and gradient calculations. As the approach taken by SPSA is to calculate the gradient using a random perturbation and by evaluating the cost function, the solution taken by Wulff et al. was to use the Graph Editor from the `tf.contrib` library to make the modifications to the Tensorflow graph manually, and evaluate the cost function in this manner. Unfortunately, `tf.contrib` is explicitly labelled as volatile and experimental by Tensorflow, and there is very little documentation provided for newer versions for the Graph Editor[26] functionality, so it was not possible to test SPSA as a result. The implementation of a more resilient layerwise SPSA algorithm is left as an interesting avenue to explore in future work.

Chapter 5

Experimental Setup

5.1 Datasets

The datasets used in this work are:

1. MNIST, an image set of handwritten digits.
2. Fashion-MNIST, an image set of clothing items.
3. Titanic, a structured set of the attributes and survival outcome of passengers on the titanic.
4. Spambase, a structured set of e-mail attributes and whether they are spam.

Fashion-MNIST and MNIST are somewhat similar, both of them being greyscale image sets with the same structure and sizes. Indeed, the Fashion-MNIST authors intend for it to be a drop-in replacement for the MNIST dataset, which is simple to solve for most deep learning networks nowadays. Although convolutional neural networks are normally used for analyzing image data, it is nevertheless interesting to see how the hybrid networks fare with images.

The dimensions of these datasets are described in table 5.1.

TABLE 5.1: Dimensions of the datasets used

Name	Instances	Feature Dimension	Classes
`MNIST`	70000	28×28	10
`Fashion-MNIST`	70000	28×28	10
`Titanic`	2201	3	2
`Spambase`	4601	57	2

The datasets have been preprocessed so that the image features contain values between 0 and 1, and the structured features have mean 0 and variance 1. Some datasets additionally required the labels to be converted to a numerical value. Image data is flattened as part of the first layer, so that it can be correctly processed by the posterior layers.

For MNIST and Fashion-MNIST the standard test sets are defined at 10,000 instances. For Titanic and Spambase, 10% of each dataset were held for the final testing. Additionally, at the start of each epoch 20% of each dataset were withheld from training, and were used as validation data to evaluate the model performance.

5.2 Experimental Design

The experiments carried out were performed for each dataset by combining each variation, and carrying out 3 trials per each dataset. After a period of hyperparameter tuning, detailed in Section 5.3, the hyperparameters were fixed for each combination of dataset, kernel approximation and optimizer method. 3 trials were then carried out with these hyperparameters, and the average time and test errors were averaged out.

Both the hyperparameter and testing phases were carried out with an early stopping criteria callback that would stop the training if the accuracy did not improve after 100 epochs. During the hyperparameter search phase, an additional early stopping criteria was added to stop all training if the performance did not improve above an accuracy of 50% after 50 epochs. This was done due to the high sensitivity of the model to the value of γ, which when not properly tuned would not allow the model to train at all, so as to reduce time spent training ineffective models.

After an initial hyperparameter tuning period, the following hyperparameters have been kept fixed for all experiments, to better evaluate the impact of the changes we have introduced:

- Dimension of the dense layer: 1024
- Dimension of the kernel layer: 2048 for RFF, 256 for Nyström

In the case of the Nyström kernel layers, the components used to estimate the kernel are resampled after every epoch through a custom Keras callback. The final activation function chosen is always the softmax function, and the loss function is always the negative log likelihood, using integer values for the class - in Keras this corresponds to the `sparse_categorical_crossentropy` loss, with the L^2 regularization factor. This regularization has only been applied to the fully connected layer weights.

5.3 Hyperparameter Tuning

Because finding appropriate hyperparameters add another level of abstraction on top of the existing training of models, which only trains its weights, it can be very time-consuming if done manually. The chosen algorithm used for hyperparameter tuning is the Tree-structured Parzen Estimator, or TPE.

TPE is a Bayesian optimization algorithm which, unlike random or grid search, uses previous estimates to build a Bayesian estimate of the cost function. Bayesian optimization methods are effective even for stochastic and non-convex cost functions, and TPE in particular has been shown to be more effective than grid search and other optimization algorithms for multiple domains[27]. Because of its inclusion in the `hyperopt` library, it is simple to implement for these experiments as an alternative to manual tuning.

Hyperparameters were tuned semi-manually in batches of 5 to prevent memory overflow problems. 5 different hyperparameter evaluations were carried out using `hyperas`, a Keras wrapper around `hyperopt`, and if the optimal values were found to be on one end of the specified range, then it was adjusted accordingly to evaluate whether the original range was too restrictive. For example, if the value for the L_2 parameter was in the range $[0, 0.5]$, and the best resulting hyperparameter was found to be 0.49, then a series of 5 more tests would be carried out with eg. the range $[0.4, 1.0]$.

5.4 Experimental Conditions

The experiments were all carried out on a desktop computer with the following specifications, running Ubuntu 18.04.2, Python 3.6.6 and using the Tensorflow-GPU 1.13.1 package.

- CPU: AMD FX-6300 3.8 GHz Six-Core Processor
- 8 GB DDR4 RAM
- Nvidia Geforce GTX 960 with 2GB RAM

Chapter 6

Results

The graphs for the progress of the validation loss and accuracy for each dataset can be seen in the following sections. As the training was performed in a two-stage process, there are four separate graphs for each experiment, two which describe the loss and accuracy of the shallow model, and a separate two for the deep model.

The specific results will be discussed in the following sections, but there are some general observations we can make which are common to all the experiments carried out. Firstly, the Nyström method approach required significantly greater training times for almost all cases, as was expected due to the larger computational complexity when compared to random Fourier features. It is also clear from the graphs that there were some significant oscillations in both loss and training - this corresponds to the resampling of the chosen features for the Nyström kernel layer, which initially causes the loss to plummet, although it generally recovers quickly.

In retrospect, there are some cases where it is likely that the early stopping criteria was too relaxed, as only a marginal improvement would cause the model to train ineffectively. Nevertheless, the overall trends with regards to training time and test accuracy can still be inferred from both the metrics included in the tables and the training graphs included.

Another important caveat with regards to the training time is that this only counts the time calculated after the initial hyperparameter search. In practice, RMSprop required essentially no specific hyperparameter tuning when compared to SGD, and default values were used throughout the tests performed. This is an important consideration, as it greatly simplifies the model generation process due to the reduced set of hyperparameters to consider.

Reducing hyperparameters is especially important when considering that the value for γ in the RBF kernel is extremely influential on model performance. A bad value of γ will cause the model to not converge, and to perform no better than chance. This value was relatively stable for all experiments with the same dataset, but it should be kept in mind that an inaccurate range for γ can cause the hyperparameter search to spend excessive time trying models which will never converge, no matter the other model parameters.

6.1 Fashion-MNIST

Fashion-MNIST is meant to be a more difficult replacement for MNIST, and it can be seen in the lower test accuracy and higher losses when compared to the MNIST data set in Table 6.2. For this dataset, Nyström with SGD did not perform well compared to the other models, as it had a lot of trouble of training the shallow model, which meant that the deep model had similar problems with training.

RFF with RMSprop proved to be the best combination here, training to a higher degree of test accuracy, with a lower training time and epochs needed for the other two approaches. While Nyström with RMSprop still proved to be competitive with regards to the final test accuracy, it took almost twice as long to train when compared to the RFF models, although it did so in fewer epochs than RFF/SGD. This shows that the training time per epoch of the Nyström method is larger than with RFF, likely due to the higher number of weights to train as well as the increased number of operations.

TABLE 6.1: Results for Fashion-MNIST

Kernel	Optimizer	Test Accuracy	Test Loss	Epochs	Training Time (s)
RFF	RMS	0.8858 ± 0.007	0.410 ± 0.15	115	1054
RFF	SGD	0.8694 ± 0.003	0.4238 ± 0.05	240	1755
Nyström	RMS	0.8589 ± 0.032	0.5106 ± 0.12	154	3038
Nyström	SGD	0.5149 ± 0.15	1.514 ± 0.23	258	2669

FIGURE 6.1: Validation accuracy and loss vs epoch for Fashion-MNIST shallow models

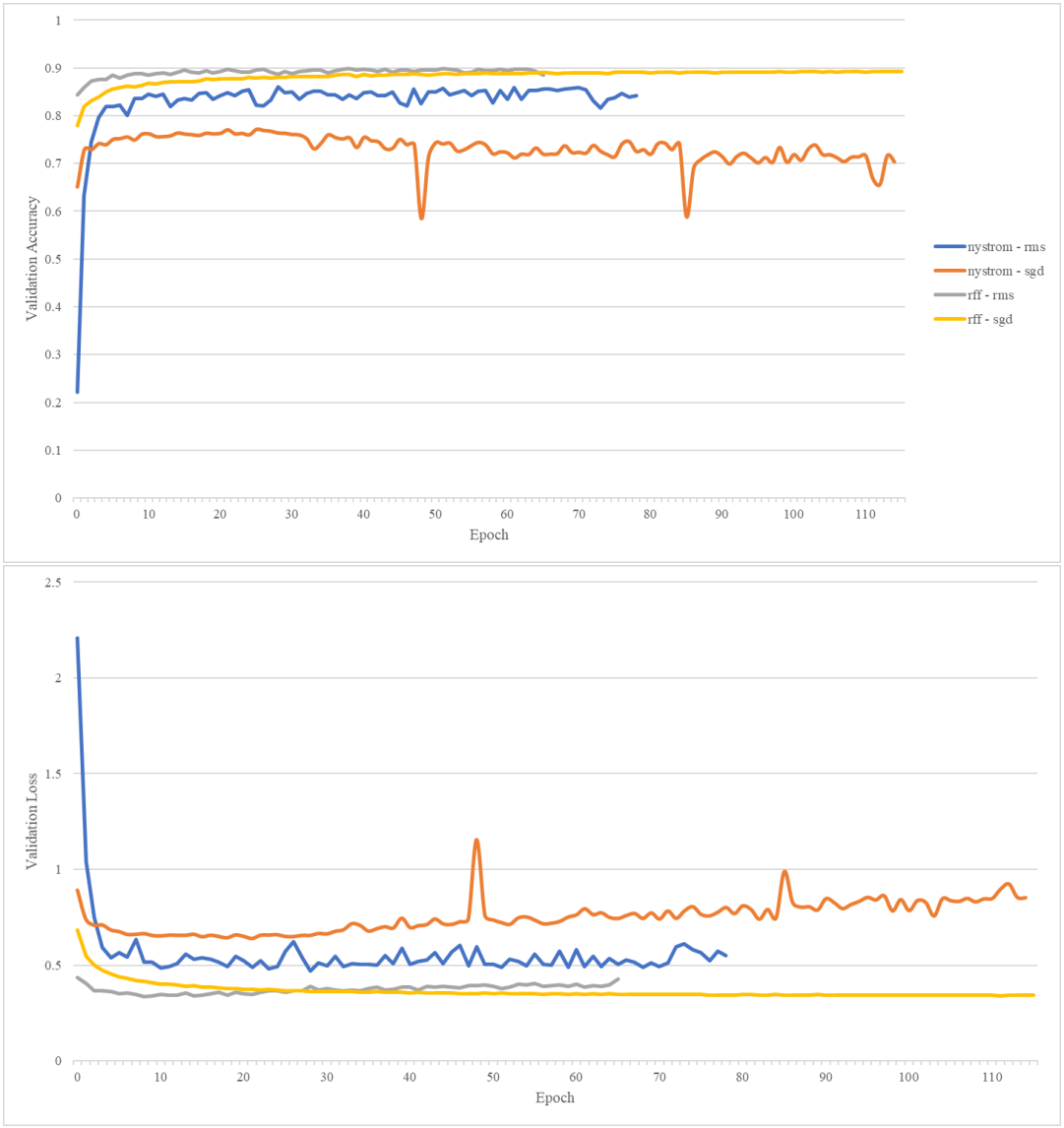
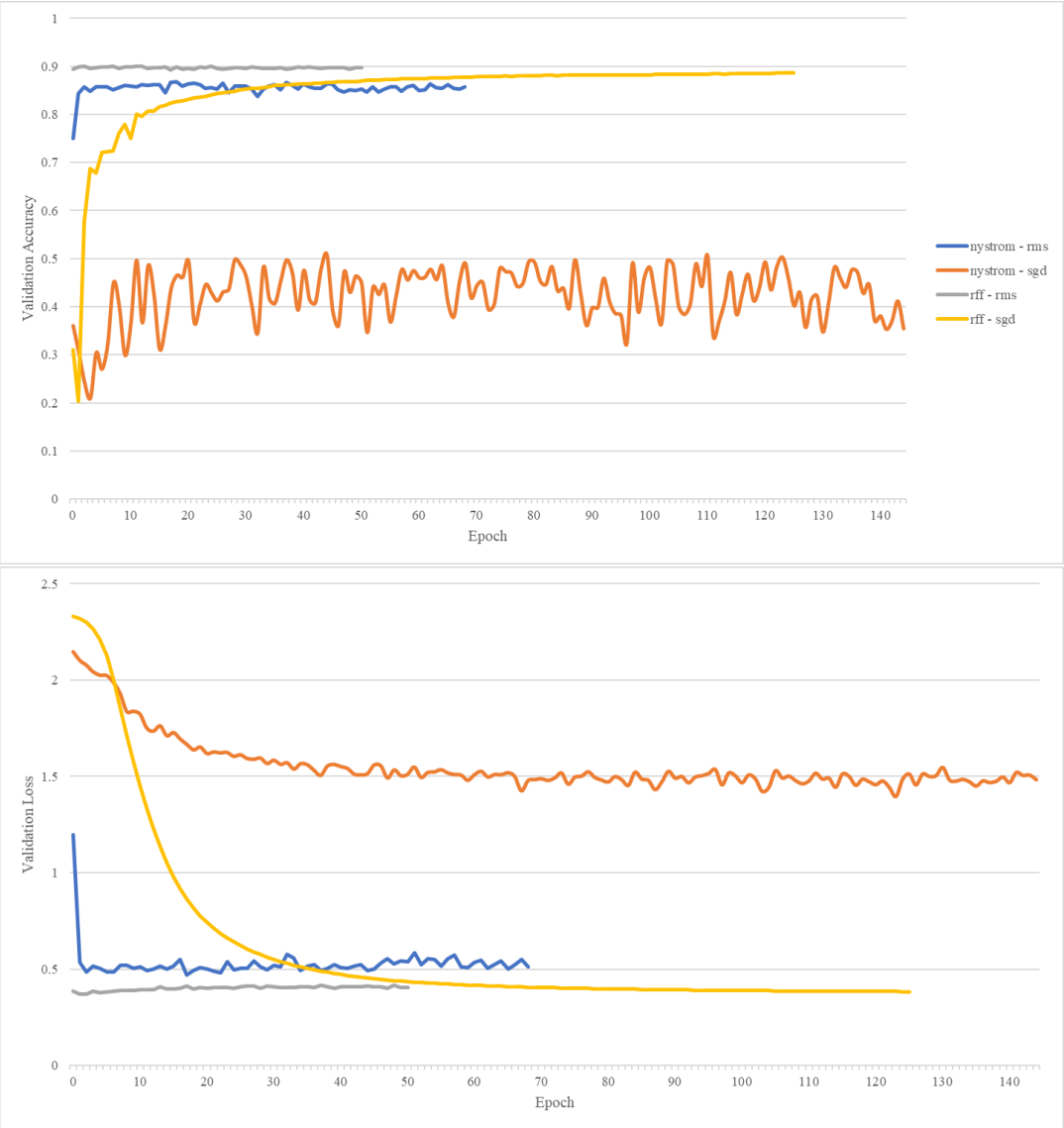


FIGURE 6.2: Validation accuracy and loss vs epoch for Fashion-MNIST deep models



6.2 MNIST

All of the networks achieved better performance in MNIST than Fashion-MNIST, although as seen previously the combination of Nyström method with SGD did not achieve good results, and it was not able to effectively train the shallow model. Although its training time was not the longest, as can be observed in Table 6.1, this is largely due to the early stopping criteria which stopped training once the model was saw not to improve. Here the best result was still given by the combination of RFF and RMSprop - this beat out all other combinations across all metrics, and

it could have potentially a much shorter training time if the early stopping criteria were relaxed.

TABLE 6.2: Results for MNIST

Kernel	Optimizer	Test Accuracy	Test Loss	Epochs	Training Time (s)
RFF	RMS	0.9707 ± 0.01	0.0895 ± 0.01	205	2624
RFF	SGD	0.8897 ± 0.06	0.9396 ± 0.1	462	5125
Nyström	RMS	0.9508 ± 0.01	0.232 ± 0.05	210	7187
Nyström	SGD	0.6694 ± 0.14	1.029 ± 0.21	308	3276

FIGURE 6.3: Validation accuracy and loss vs epoch for MNIST shallow models

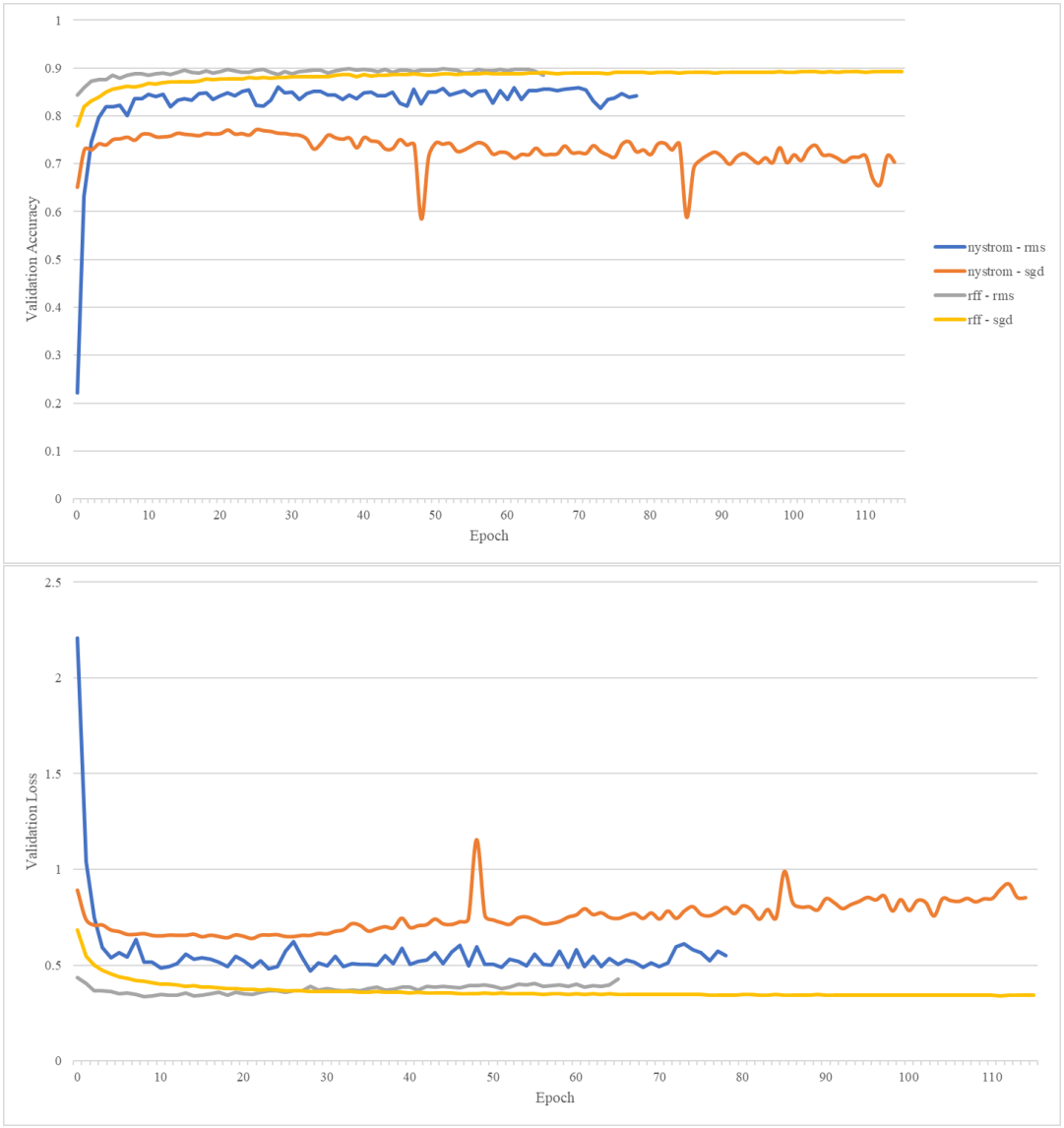
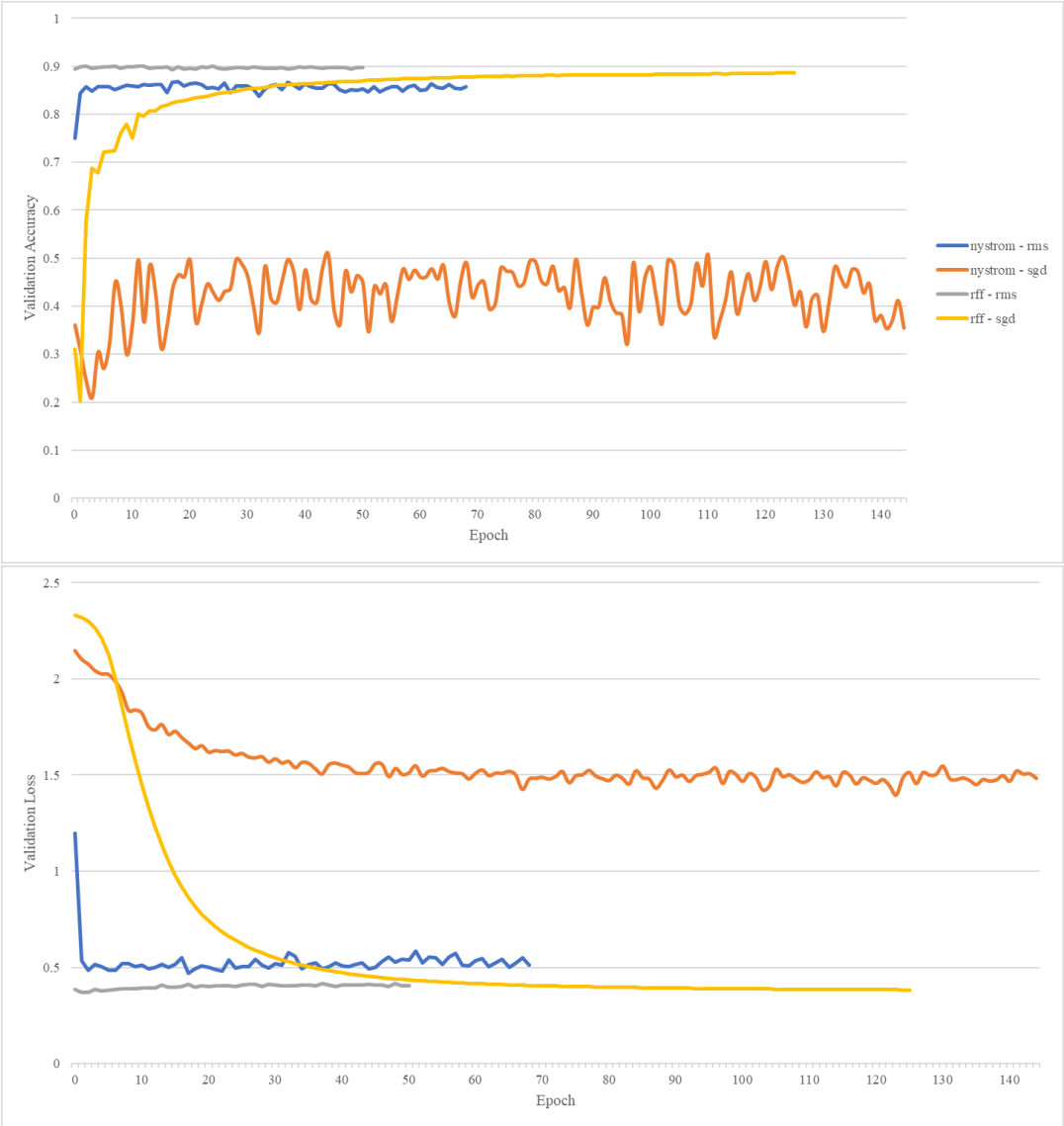


FIGURE 6.4: Validation accuracy and loss vs epoch for MNIST deep models



6.3 Titanic

The results obtained for Titanic show some of the same trends observed in other datasets, such as the increased training time of the Nyström method, but although the best combination remains RFF with RMSProp in terms of performance, here Nyström with SGD proved to be almost as competitive with RFF with RMSProp. RMSProp with Nyström did not prove to be as effective an alternative as seen with the previous datasets.

TABLE 6.3: Results for Titanic

Kernel	Optimizer	Test Accuracy	Test Loss	Epochs	Training Time (s)
RFF	RMS	0.814 ± 0.05	0.463 ± 0.08	239	65
RFF	SGD	0.651 ± 0.17	0.664 ± 0.05	267	59
Nyström	RMS	0.696 ± 0.03	0.673 ± 0.1	205	430
Nyström	SGD	0.810 ± 0.05	0.487 ± 0.05	281	798

FIGURE 6.5: Validation accuracy and loss vs epoch for Titanic shallow models

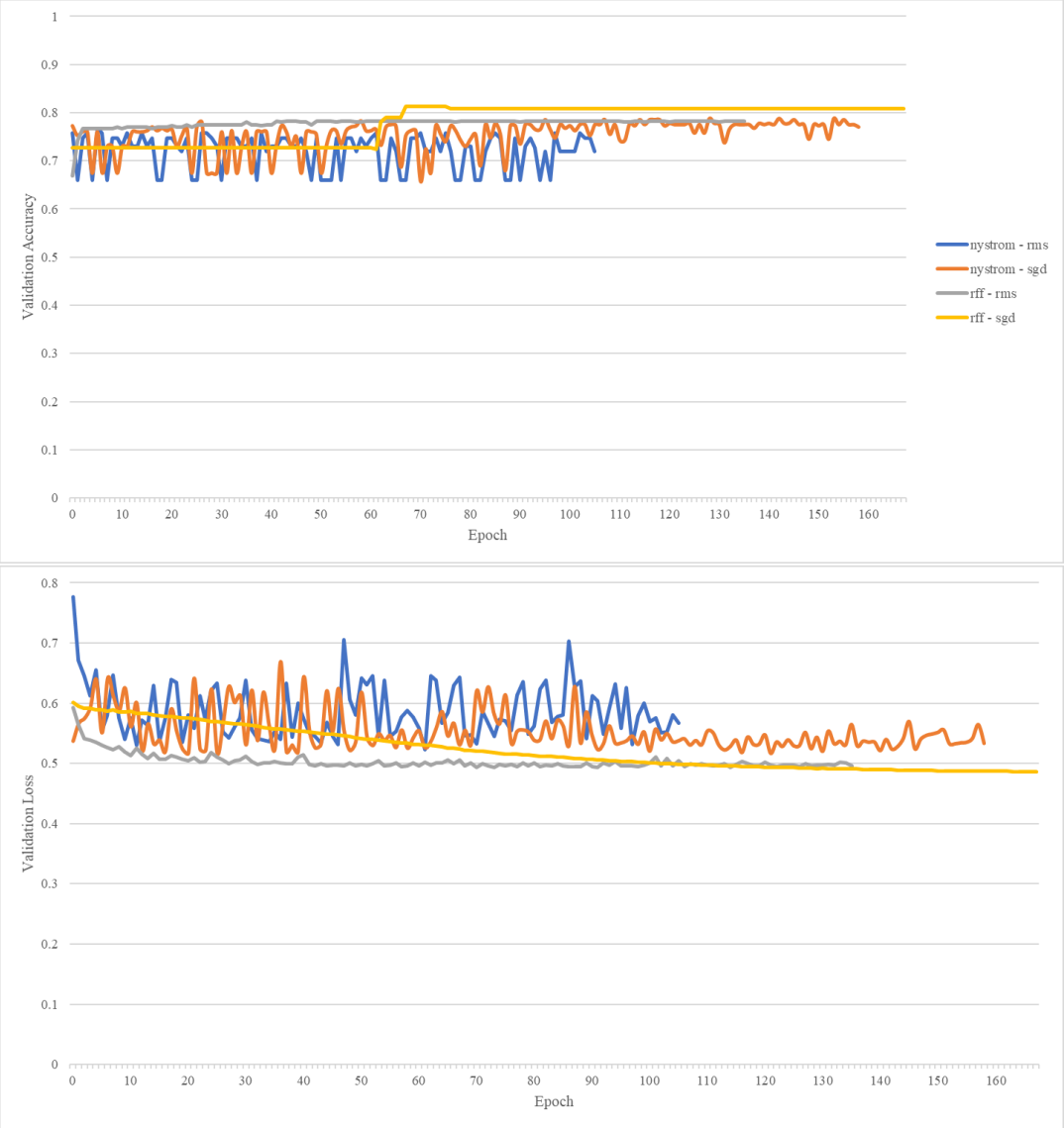
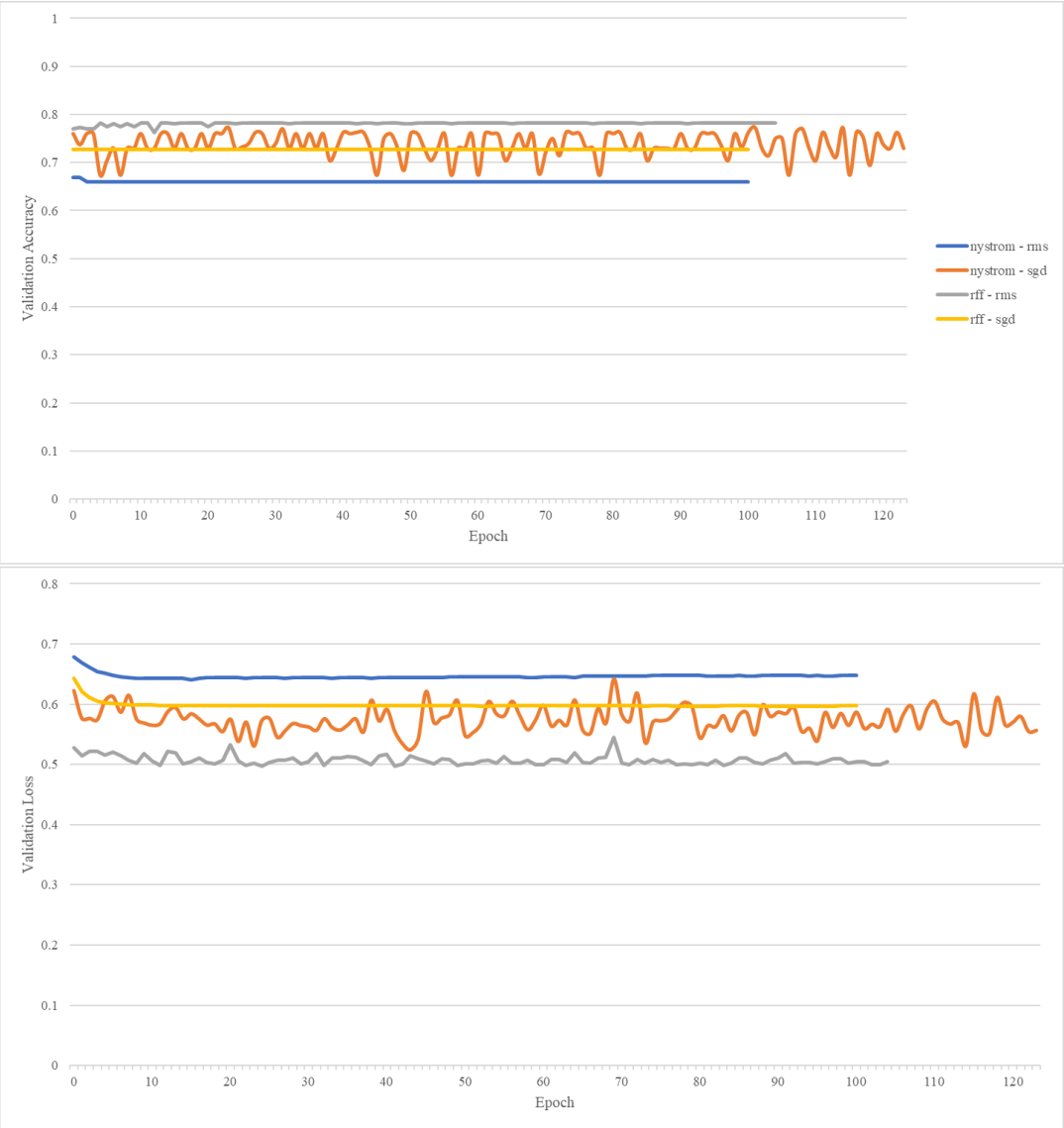


FIGURE 6.6: Validation accuracy and loss vs epoch for Titanic deep models



6.4 Spambase

Spambase proved to have very similar performance results for almost all of the experiments, with the same trends in training time which have already been observed. RFF with RMSprop is still the best performing in test accuracy, although only slightly marginally, while obtaining a similar training time to RFF with SGD. Here Nystrom performed almost as well, but at the expense of a much longer training time.

TABLE 6.4: Results for Spambase

Kernel	Optimizer	Test Accuracy	Test Loss	Epochs	Training Time (s)
RFF	RMS	0.932 ± 0.01	0.184 ± 0.005	204	118
RFF	SGD	0.931 ± 0.01	0.204 ± 0.004	231	113
Nyström	RMS	0.924 ± 0.01	0.220 ± 0.011	222	523
Nyström	SGD	0.924 ± 0.01	0.234 ± 0.010	232	592

FIGURE 6.7: Validation accuracy and loss vs epoch for Spambase shallow models

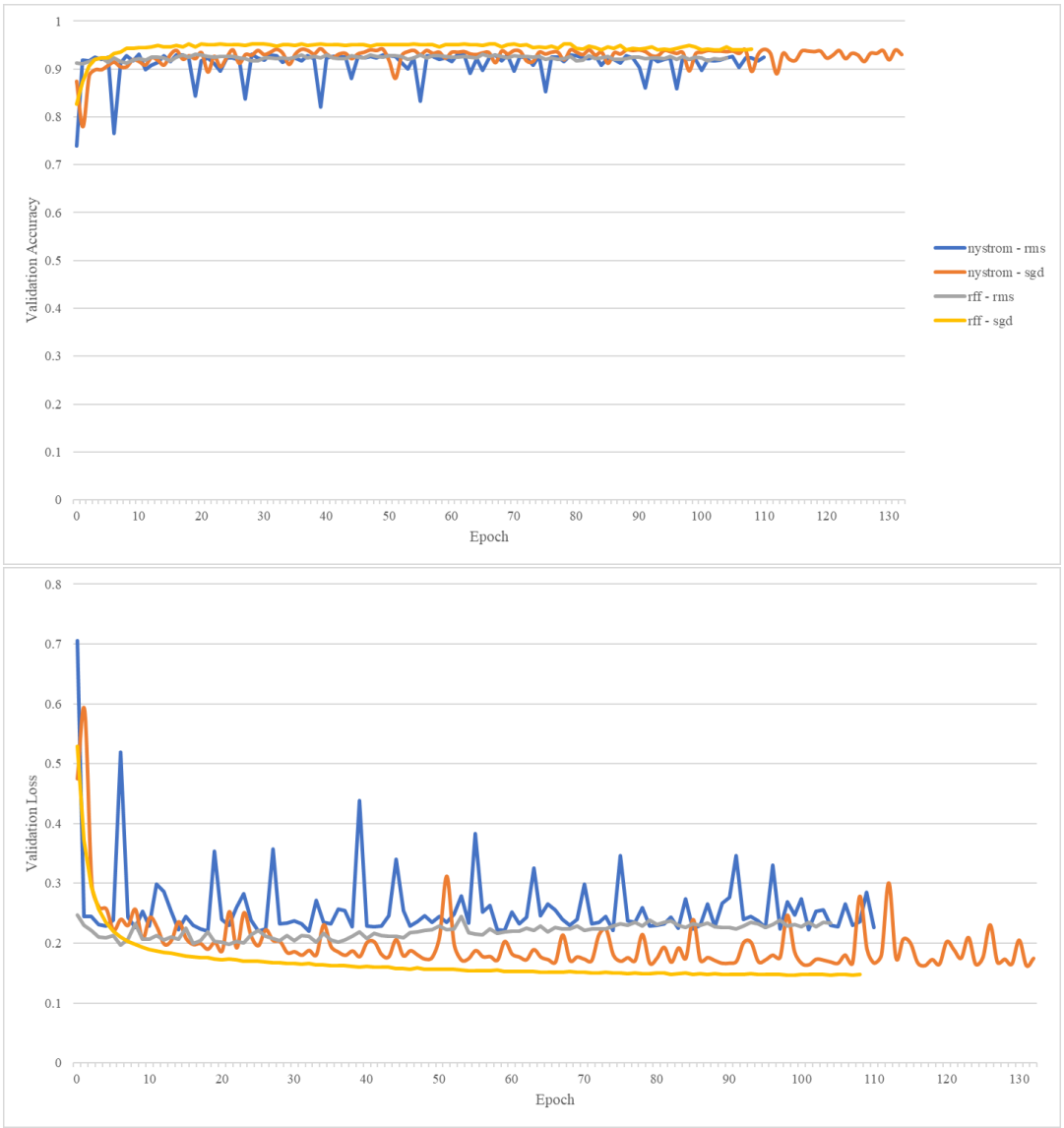
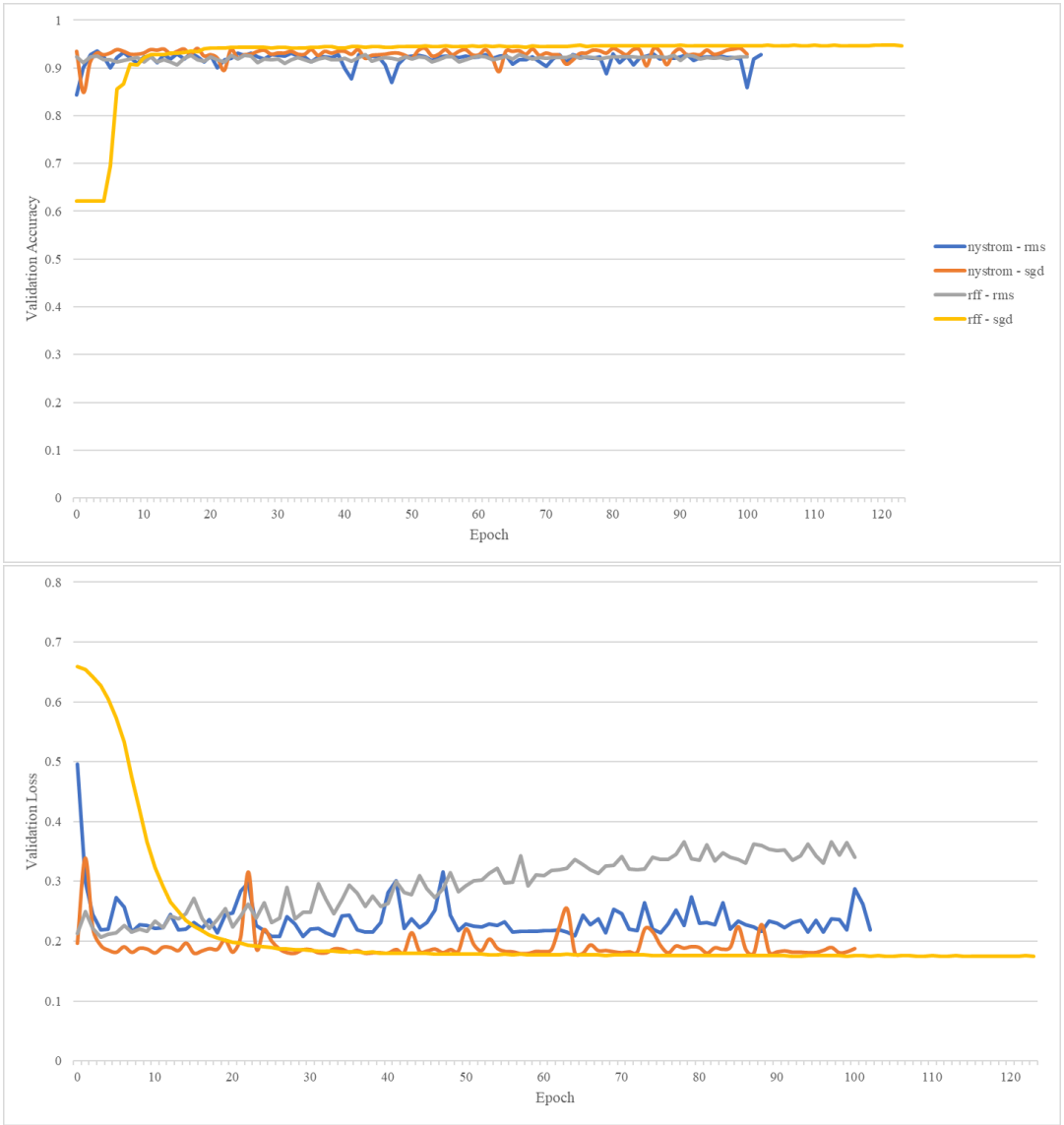


FIGURE 6.8: Validation accuracy and loss vs epoch for Spambase deep models



Chapter 7

Conclusions and Future Work

7.1 Conclusions

The experiments in this thesis were carried out with the aim of evaluating the performance of the Nyström method when compared to random Fourier features, and how the choice of optimization algorithm affected results. By comparing how these different approaches performed both in terms of model accuracy and training times, it is possible to assess how hybrid neural kernel networks can be best implemented and used in order to solve classification problems. The experiments carried out in this thesis can help to guide the development of hybrid neural networks, by identifying which areas have the most potential for future studies, as well as identifying the state of the art amongst these approaches for solving classification problems.

In almost all cases, the best test performance was obtained with the combination of RFF with the adaptive RMSprop method, while the Nyström method with SGD normally performed the worst with the longest training times. SPSA could not be replicated and so it is not possible to evaluate how well it performs with hybrid networks.

Despite the testing performed here not corroborating the findings of Yang et al.[3] when applied to neural networks, this has some caveats. The first is that the method tested here is not a "true" Nyström method, in the sense that it attempts to learn the generalized Gram matrix instead of calculating it explicitly from the input data. However, given that this approximated method was used as a result of the high complexity of the pseudoinverse matrix required for the Nyström method, it still stands that the Nyström method is not a very good alternative to RFF for neural kernel

networks. Because the pseudoinverse calculation must be carried out periodically when training, it is unlikely that the Nyström method will be a feasible solution for deep learning until the SVD algorithm used is more easily parallelized or otherwise accelerated.

Furthermore, it must be noted that the Nyström method is highly data-dependant, and so it may be that these datasets simply do not have the optimal conditions to be effectively used for the Nyström method. The Spambase dataset showed a very similar performance, despite higher training times, which does show that the Nyström method should be tested always in close relation to the input data which is used.

On the other hand, the dismissal by Wilson et al. [5] of adaptive methods can also not be supported with these results. Despite stochastic gradient descent method displaying a better performance than RMSprop for the Titanic dataset with the Nyström method, this is the exception rather than the rule. In all other measures considered (test accuracy, test loss and training time), RMSprop seems to perform better than SGD for almost all cases, which shows that at least in the cases discussed here, RMSProp is a valid algorithm to consider for hybrid neural network optimization. In general, RMSProp converges more quickly at the beginning of training than SGD, and requires fewer epochs and a reduced training time in order to achieve superior results. This, combined with the minor impact which the hyperparameters have on the RMSprop performance, validates its position as one of the most widely used optimizers for neural networks.

The combination of RFF and RMSprop proved to be the most robust across all the tested datasets. In practice, the implementation of RFF and RMSprop required the least tweaking in terms of hyperparameters, converged quickly and obtained the best performance, which suggests that this combination should be singled out for future research and experimentation, although as has been mentioned, the other approaches cannot yet be discarded without further testing.

7.2 Future Work

There are several different paths which could prove interesting for further research. The most obvious would be to attempt to validate the layerwise SPSA approach for

hybrid neural networks which could not be effectively tested here. A new version of the optimizer, which worked effectively with newer versions of Tensorflow, would be an interesting development to consider, as it could be that the Nyström method here considered would work well with such an optimizer.

It is also not clear whether different approaches to the training could achieve better performance for either the Nyström or RFF approaches. This thesis has kept the first layers frozen when training the deep model, but it would be interesting to see whether training the whole network at once using SGD or RMSprop would yield better generalization performance, or whether it would be beneficial to fix the last layers and re-train the first layers after the deep model has been trained.

Due to the Nyström method's high sensitivity to the input data, it may be that other datasets respond better to the Nyström method than what has been tested in this thesis. It may be that these datasets are simply not appropriate candidates for a Nyström approximation, or that a particular normalization or pre-processing may help to yield better results for the Nyström method. Another possibility would be to reduce the resampling frequency, so that it is only carried out every n epochs, instead of doing it every epoch. Other types of data should also be considered, as only structured and image classification problems have been considered here, but it's not clear how effective hybrid neural networks would be when applied to other tasks like regressions or speech recognition.

Despite the widespread usage of the RBF kernel, it would also be interesting to compare how other kernels perform when using hybrid neural networks, as this is currently unexplored. This could also tie in with another possible avenue, which is that of effective hyperparameter searching. A TPE approach was taken due to its theoretical effectiveness over grid or random search, but it has not been tested whether this is the case, or whether other methods could yield better results. This is especially important in the context of hybrid neural networks having hyperparameters which can have a drastic effect on the performance of the network.

Bibliography

[1] D. Takahashi, *Nvidia ceo bets big on deep learning and vr*, Apr. 2016. [Online]. Available: https://venturebeat.com/2016/04/05/nvidia-ceo-bets-big-on-deep-learning-and-vr/.

[2] S. Mehrkanoon and J. A. Suykens, "Deep hybrid neural-kernel networks using random Fourier features", en, *Neurocomputing*, vol. 298, pp. 46–54, Jul. 2018, ISSN: 09252312. DOI: 10.1016/j.neucom.2017.12.065. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0925231218302108 (visited on 01/19/2019).

[3] T. Yang, Y.-f. Li, M. Mahdavi, R. Jin, and Z.-H. Zhou, "Nyström method vs random fourier features: A theoretical and empirical comparison", in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2012, pp. 476–484. [Online]. Available: http://papers.nips.cc/paper/4588-nystrom-method-vs-random-fourier-features-a-theoretical-and-empirical-comparison.pdf.

[4] B. Wulff, J. Schuecker, and C. Bauckhage, "Spsa for layer-wise training of deep networks", *Artificial Neural Networks and Machine Learning – ICANN 2018 Lecture Notes in Computer Science*, pp. 564–573, 2018. DOI: 10.1007/978-3-030-01424-7_55.

[5] A. G. Wilson, Z. Hu, R. Salakhutdinov, and E. P. Xing, "Deep kernel learning", *CoRR*, vol. abs/1511.02222, 2015. arXiv: 1511.02222. [Online]. Available: http://arxiv.org/abs/1511.02222.

[6] D. M. de Checa, "New hybrid kernel architectures for deep learning", Master's thesis, Universitat Politècnica de Catalunya, Apr. 2018.

[7] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*, ser. Adaptive computation and machine learning. Cambridge, Massachusetts: The MIT Press, 2016, ISBN: 9780262035613.

[8] C. McDonald, *Machine learning fundamentals (ii): Neural networks*, Dec. 2017. [Online]. Available: https://towardsdatascience.com/machine-learning-fundamentals-ii-neural-networks-f1e7b2cb3eef.

[9] K. Hornik, "Approximation capabilities of multilayer feedforward networks", *Neural Networks*, vol. 4, no. 2, pp. 251–257, 1991. DOI: 10.1016/0893-6080(91)90009-t.

[10] T. Hofmann, B. Schölkopf, and A. J. Smola, "Kernel methods in machine learning", en, *The Annals of Statistics*, vol. 36, no. 3, pp. 1171–1220, Jun. 2008, ISSN: 0090-5364. DOI: 10.1214/009053607000000677. [Online]. Available: http://projecteuclid.org/euclid.aos/1211819561 (visited on 01/21/2019).

[11] X. Yan, A. Song, and H. Yan, "A graph embedding method based on sparse representation for wireless sensor network localization", *International Journal of Distributed Sensor Networks*, vol. 2014, pp. 1–13, Jul. 2014. DOI: 10.1155/2014/607943.

[12] G. James, D. Witten, T. Hastie, and R. Tibshirani, Eds., *An introduction to statistical learning: With applications in R*, ser. Springer texts in statistics 103. New York: Springer, 2013, OCLC: ocn828488009, ISBN: 9781461471370.

[13] *Build with ai*. [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/affinity-matrix.

[14] J. Yang, V. Sindhwani, H. Avron, and M. Mahoney, "Quasi-monte carlo feature maps for shift-invariant kernels", in *International Conference on Machine Learning*, 2014, pp. 485–493.

[15] A. Rahimi and B. Recht, "Random features for large-scale kernel machines", in *Advances in Neural Information Processing Systems 20*, J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, Eds., Curran Associates, Inc., 2008, pp. 1177–1184. [Online]. Available: http://papers.nips.cc/paper/3182-random-features-for-large-scale-kernel-machines.pdf.

[16] C. K. I. Williams and M. Seeger, "Using the nyström method to speed up kernel machines", in *Advances in Neural Information Processing Systems 13*, T. K. Leen, T. G. Dietterich, and V. Tresp, Eds., MIT Press, 2001, pp. 682–688. [Online]. Available: http://papers.nips.cc/paper/1866-using-the-nystrom-method-to-speed-up-kernel-machines.pdf.

[17] P. Drineas and M. W. Mahoney, "On the nyström method for approximating a gram matrix for improved kernel-based learning", *J. Mach. Learn. Res.*, vol. 6, pp. 2153–2175, Dec. 2005, ISSN: 1532-4435. [Online]. Available: http://dl.acm.org/citation.cfm?id=1046920.1194916.

[18] G. H. Golub and C. F. v. Loan, *Matrix computations*. Johns Hopkins University Press, 2013.

[19] Y. Abu-Mostafa, *Lecture 15 - kernel methods*, May 2012. [Online]. Available: https://www.youtube.com/watch?v=XUj5JbQihlU&list=PLCA2C1469EA777F9A&index=15.

[20] G. Hinton, N. Srivastava, and K. Swersky, *Neural networks for machine learning*. [Online]. Available: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[21] Q. Song, J. Spall, Y. C. Soh, and J. Ni, "Robust neural network tracking controller using simultaneous perturbation stochastic approximation", *IEEE Transactions on Neural Networks*, vol. 19, no. 5, pp. 817–835, 2008. DOI: 10.1109/tnn.2007.912315.

[22] L. Giffon, H. Kadri, S. Ayache, and T. Artières, "Deepström networks", Dec. 2018.

[23] TensorFlow, *What's coming in tensorflow 2.0*, Jan. 2019. [Online]. Available: https://medium.com/tensorflow/whats-coming-in-tensorflow-2-0-d3663832e9b8.

[24] *Writing your own keras layers*. [Online]. Available: https://keras.io/layers/writing-your-own-keras-layers/.

[25] Tensorflow, *Svd in tensorflow is slower than in numpy issue #13222 tensorflow/tensorflow*. [Online]. Available: https://github.com/tensorflow/tensorflow/issues/13222.

[26] *Module: Tf.contrib.graph_editor | tensorflow*. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/contrib/graph_editor.

[27] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization", in *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2011, pp. 2546–2554. [Online]. Available: http://papers.nips.cc/paper/4443-algorithms-for-hyper-parameter-optimization.pdf.

[28] T. Hastie, R. Tibshirani, and J. H. Friedman, *The elements of statistical learning: Data mining, inference, and prediction*, 2nd ed, ser. Springer series in statistics. New York, NY: Springer, 2009, ISBN: 9780387848587.

[29] M. Nielsen, *Neural networks and deep learning*, Oct. 2018. [Online]. Available: http://neuralnetworksanddeeplearning.com/chap4.html.

[30] P. Domingos, *The master algorithm: How the quest for the ultimate learning machine will remake our world*. Basic Books, a member of the Perseus Books Group, 2018.

[31] *Rbf svm parameters*. [Online]. Available: https://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html.

[32] J. C. Spall. [Online]. Available: https://www.jhuapl.edu/SPSA/.

[33] 10100885294624622, *Hyperparameter tuning*, Feb. 2019. [Online]. Available: https://towardsdatascience.com/hyperparameter-tuning-c5619e7e6624.

[34] I. Dewancker, M. McCourt, and S. Clark, *Bayesian optimization primer - sigopt*. [Online]. Available: https://sigopt.com/static/pdf/SigOpt_Bayesian_Optimization_Primer.pdf.

[35] Zalandoresearch, *Zalandoresearch/fashion-mnist*, Oct. 2018. [Online]. Available: https://github.com/zalandoresearch/fashion-mnist.

[36] S. Ruder, "An overview of gradient descent optimization algorithms", *CoRR*, vol. abs/1609.04747, 2016. arXiv: 1609.04747. [Online]. Available: http://arxiv.org/abs/1609.04747.

[37] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, "The marginal value of adaptive gradient methods in machine learning", in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., Curran Associates, Inc., 2017, pp. 4148–4158. [Online]. Available: http://papers.nips.cc/paper/7003-the-marginal-value-of-adaptive-gradient-methods-in-machine-learning.pdf.