UNIVERSITAT POLITÈCNICA DE CATALUNYA

NEW HYBRID KERNEL ARCHITECTURES FOR DEEP LEARNING

by

Daniel Mora de Checa

A thesis submitted in fulfillment for the
Master in Artificial Intelligence

Advisor: Luís Belanche Muñoz

in the
Facultat d'Informàtica de Barcelona

April 2018

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Abstract

Facultat d'Informàtica de Barcelona

Master in Artificial Intelligence

by Daniel Mora de Checa

In this work we explore the possibilities of combining neural network architectures and kernel methods by introducing hybrid kernel blocks. These kernel blocks map the hidden features in the network into approximated infinite-width spaces. We present hybrid architectures which can be trained as traditional neural networks and introduce novel methodologies to train this hybrid structures in a layerwise fashion. Additionally, we propose regularization methods which are specific for this kind of architectures and we prove that our approach can be easily extended to support heterogeneous kinds of data.

Contents

Chapter 1

Introduction

1.0.1 Raise of deep learning

Artificial neural networks (ANN), recently renamed as Deep Learning, have become the state-of-the-art in many different complex learning problems using heterogeneous data, such as machine translation, image captioning or face recognition. Though ANN have existed for decades and have gone through different names (e.g. connectionism, cybernetics) it has experimented a rebirth in the last decade due to several facts.

With the appearance of very large datasets we can obtain good performance on deep networks without neither domain expertise (i.e. no need to resort to complex handcrafted representations) nor practical expertise. The availability of more powerful computing resources and general purpose GPUs helped to increase the feasibility of training larger networks, leading to better results.

Though the algorithms training these models have been the same for many years, they have been adapted in order to simplify, scale and make more efficient the training process [1]. Moreover, many software tools and infrastructures have been released that facilitate coding and deploying these networks (e.g. Tensorflow, MXNet).

Deep networks are used not only to solve classification or regression problems but to learn rich representations of the data which can easily be used to solve other problems through transfer learning.

Nevertheless, ANN are usually optimized using cost functions which may have many saddle points and may be hard and slow to train. Moreover, the training of a neural network also involves tuning a handful of hyperparameters.

1.0.2 Kernel methods

Kernel methods have been widely used in machine learning and they have been applied to heterogeneous types of data such as graphs or trees [2]. These methods rely on mapping the input representation into a high dimensional feature space where we can efficiently apply linear models on. Kernel methods solve tractable convex problems which have a single global optimum and can be applied to any kind of data where a dot product can be defined.

On the other hand, they rely on the Gram matrix computation, which is quadratic with respect to the number of training instances and therefore, hardly scalable.

1.0.3 Bringing the best from both worlds

There has been a recent line of research that combines these two approaches in order to get the best from both methodologies. One of these emerging ideas is to create hybrid layers in neural networks using kernel functions. In this work we extend existing work from Mehrkanoon et al. [3], which uses a single layer hybrid approach, into multilayer architectures where several layers are stacked together. These deep hybrid architectures are built using what we name as *kernel blocks*, which is the concatenation of a traditional fully connected layer and an explicit feature mapping using Random Fourier Features [4]. We additionally explore new ways of training these multilayer structures by using different incremental and layerwise training strategies as well as specific regularization techniques for these networks.

1.1 Related work

We find several examples of hybrid methodologies between kernel and deep methods in the literature, which differ in terms of their scalability, complexity and their ease of combining both techniques.

Some of the methods reviewed take advantage of Gaussian Processes. Gaussian Processes are kernel methods within the probabilistic modelling framework that impose Gaussian priors on their parameters [5]. An example can be found in the work from Wilson et al. [6], where they combine deep hidden layouts with a final infinite-width layer which is learn through the marginal likelihood of the Gaussian Process. This model can be interpreted as applying a Gaussian Process to the last layer of a deep network, with the particularity that they are jointly trained. However, the cost of training these models is linearly dependent to the number of instances in the dataset.

Another hybrid example can be found in the work from Pandey and Dukkipati [7], where they use a wide learning (i.e. a single layer architecture of infinite width) through an arc-cosine kernel. They propose exact and approximate procedures to train single-layer wide networks. They provide an approximate strategy to compute the kernel matrix of an arc-cosine kernel, which is constructed with the weight matrix learnt by a Restricted Boltzmann Machine (RBM). The kernel matrix is finally fed into a linear kernel classifier. Approximations for concatenating several kernel arc-cosine kernel layers are also presented, but they show lower performances than the one from the single-layer architecture. They show that this wide network can achieve better results than single layer and deep belief networks. On the other hand, while we see how this method takes advantage of kernel methods it cannot achieve to get stratified representations of the data which make deep learning architectures a good choice for transfer learning.

Several works in the literature have approximated kernel methods within neural networks establishing training procedures that do not deviate much from the traditional deep learning approach (e.g. usage of the same or very similar hyperparameter set, usage of the same cost function or optimization algorithm) while are efficient and easy to implement at the same time. Mehrkanoon et al. [3] propose the basis for implementing hybrid neural networks by stacking an additional layer that approximates a Gaussian kernel using Random Fourier Features onto a fully connected layer. They empirically show that they can match the performance of traditional kernel methods (e.g. LS-SVM) while being able to scale to datasets of any size. In this work we generalize the commented work by stacking multiple kernel blocks building deeper architectures and providing training procedures to train them in a couple of different scenarios (i.e. structured and non-structured data).

Following the recent advances in computer vision, several works have also focused in providing hybrid Convolutional architectures. An example of Convolutional Kernel Networks can be found in [8]. They propose an unsupervised approach, resembling hierarchical kernel descriptors, to obtain shift-invariant representations by learning convolutional filters that approximate the kernel map on the training data. Consequently, they do not use labeled data for learning such filters but instead feed the learned features into a SVM. These convolutional filters are succesively stacked and trained in an iterative fashion: filter k is trained using the feature map from $k-1$. These filters approximate the Gaussian kernel map using a set of operations: a convolution followed by a non-linearity and a downsampling through pooling. Though these representations can be obtained using the same hyperparameter set as traditional neural networks, learning each filter requires a specific cost.

Work in [9] proposes 3 different methods for representing Reproducible Kernel Hilbert Spaces (RKHS) as layers in multilayer architectures. The first one is a nonparametric approach whose parametrization scales with the training data and, though it guarantees to contain the global minimum, it is hardly found due to its high computational complexity. The second approach approximates the kernel function in the layers using learnable vectors pre-trained on the training data. The third approach is a fully approximates one which uses Random Fourier Features to approximate shift invariant kernels. The kernel blocks of this last variant have a similar structure as the ones we are presenting.

The approach we have selected involves minimal changes with respect to traditional deep learning due to its simplicity but takes advantage of the benefits of kernel methods.

1.2 Objectives

We define a set of goals to be unraveled in this work:

- Be able to train multilayer architectures using kernel blocks.
- Compare hybrid architectures with traditional feed forward networks in terms of computing time, accuracy and depth.
- Design layerwise training strategies and compare them with traditional training strategies for hybrid networks.
- Design new regularization approaches for hybrid architectures and compare them with existing regularization techniques.
- Show that our stacked architectures can work in different domains.

1.3 Structure of the document

In Chapter 2 we present how we build our deep hybrid architectures using kernel blocks. Chapter 3 introduces training strategies about how to train these hybrid networks. In Chapter 4 we review available regularization methods to apply to the hybrid architectures and propose new ones. Chapter 5 details the tools and used for the implementation and Chapter 6 shows a detailed explanation of the experiments to compute. Finally, Chapter 7 and Chapter 8 respectively show the results and the conclusions of our work.

Chapter 2

Kernel hybrid layers

2.1 Kernel methods: a review

There are many linear models for regression and classification that can be transformed into dual representations. The dual representation provides predictions given a linear combination of the training data. For example, the cost function of the dual formulation of the traditional linear regression with regularization is:

$$J(W) = \frac{1}{2}\sum_{n=1}^{N}(\boldsymbol{W^T}\boldsymbol{\phi(x_n)} - \boldsymbol{t_n})^2 + \frac{\lambda}{2}\boldsymbol{W^T W} \tag{2.1}$$

Note that $\boldsymbol{W}$ is the weight matrix, λ is the regularization parameter and $\boldsymbol{x_i}$ and $\boldsymbol{t_i}$ respectively refer to the i^{th} example and label in the training data. The cost function can be rewritten in terms of the vector $\boldsymbol{a} = (a_1, ..., a_N)$ and the vector of targets $\boldsymbol{t} = (t_1, ..., t_N)$:

$$J(a) = \frac{1}{2}\boldsymbol{a^T K K a} - \boldsymbol{a^T K t} + \frac{1}{2}\boldsymbol{t^T t} + \frac{\lambda}{2}\boldsymbol{a^T K a} \tag{2.2}$$

Given that $a_i = -\frac{1}{\lambda}(\boldsymbol{W^T}\boldsymbol{\phi}(\boldsymbol{x_n}) - \boldsymbol{t_n})$ and that $\boldsymbol{K}$ is what is defined as the **Gram matrix**. The Gram matrix is a symmetric (n, n) matrix defined as:

$$K_{nm} = k(\boldsymbol{x_n}, \boldsymbol{x_m}) = \langle \boldsymbol{\phi(x_n)^T}, \boldsymbol{\phi(x_m)} \rangle \tag{2.3}$$

The whole derivation can be followed in [10]. The function k is what is referred as kernel function.

2.1.1 The kernel function

The kernel function is a (symmetric) similarity function that is computed as the inner product between two inputs which have been projected into a (usually) high-dimensional space by a feature map ϕ. Given the previous example, an identity map ϕ_I would reduce the problem into the traditional linear regression problem since:

$$k_I(\boldsymbol{x}, \boldsymbol{x'}) = \langle \boldsymbol{\phi_I}(\boldsymbol{x}), \boldsymbol{\phi_I}(\boldsymbol{x'}) \rangle = \boldsymbol{x}^{\boldsymbol{T}} \boldsymbol{x} \tag{2.4}$$

Any function that is an inner product in some space H can be regarded as a kernel function. More formally, this corresponds to kernels which are positive definite. A function that generates a Gram matrix which is positive definite can be considered a kernel [11]. Kernels can be created according to the similarity functions needed in a specific problem as long as they fulfill the requirements we mentioned or can be created given other kernels.

Two important aspects of the success of kernel methods are:

- Being able to map inputs into a high dimensional space where to efficiently compute a separator hyperplane from.
- The fact that we do not need to explicitly define the feature mapping ϕ but just provide a function which is a valid kernel and that is an inner product in some space, even if this feature space is unknown or even infinite [12] (i.e. what is commonly called as the kernel trick).

However, the computational and storage cost to pay for using these methods which rely on a Gram matrix is high, as it is at least quadratic in the number of training examples. This prevents us from applying them on even moderate-size datasets. Research on kernel methods has provided different ways to fix this such as low rank decomposition (i.e. approximate the Gram matrix with one of lower rank) or other methods which directly approximate the kernel function, such as Random Fourier Features or Nyström method [13]. In this work we are using Random Fourier Features.

2.2 Random Fourier Features (RFF)

Rahimi and Recht [4] propose a method to approximate the inner product in a high-dimensional space by a inner-product in a Euclidean (i.e. lower-dimensional) space.

Input data is explicitly mapped into this space by a randomized feature map z in a way that the inner product of the transformed features approximates the kernel function:

$$k(\boldsymbol{x}, \boldsymbol{x'}) = \langle \boldsymbol{\phi}(\boldsymbol{x}), \boldsymbol{\phi}(\boldsymbol{x'}) \rangle \approx \boldsymbol{z}(\boldsymbol{x})\boldsymbol{z}(\boldsymbol{x'}) \tag{2.5}$$

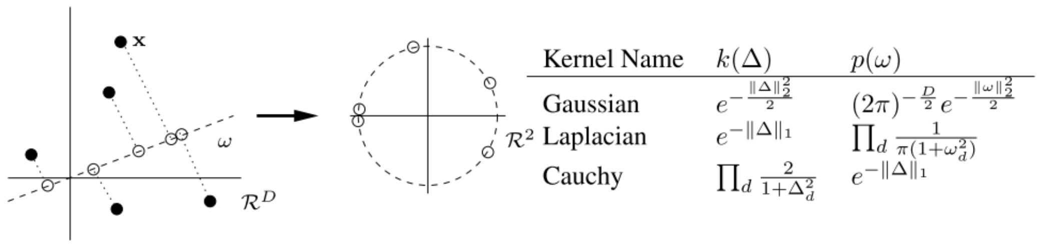
By doing so, we can use an approximated version of a nonlinear kernel machine by using linear efficient methods on the transformed data without resorting to the whole original training dataset. Given the regularized linear regression cost function presented in Section 2.1.1, we can redefine it using mapping the data into a new space as:

$$J(w) = \sum_{n=1}^{N} \frac{1}{2}(\boldsymbol{W^T} \boldsymbol{z}(\boldsymbol{x_n}) - \boldsymbol{t_n})^2 + \frac{\lambda}{2}\boldsymbol{W^T}\boldsymbol{W} \tag{2.6}$$

Equivalently, given a learnt hyperplane w we can evaluate point x as:

$$f(x) = \boldsymbol{W^T}\boldsymbol{z}(\boldsymbol{x}) \tag{2.7}$$

This results into much faster training and evaluation methods than traditional kernel learning.



Kernel Name	$k(\Delta)$	$p(\omega)$
Gaussian	$e^{-\frac{\|\Delta\|_2^2}{2}}$	$(2\pi)^{-\frac{D}{2}} e^{-\frac{\|\omega\|_2^2}{2}}$
Laplacian	$e^{-\|\Delta\|_1}$	$\prod_d \frac{1}{\pi(1+\omega_d^2)}$
Cauchy	$\prod_d \frac{2}{1+\Delta_d^2}$	$e^{-\|\Delta\|_1}$

FIGURE 2.1: Visual explanation of how Random Fourier Features are computed. Source: [4]. Note that the w does not refer to a hyperplane but the sample drawn from the Fourier Transform of the kernel.

The Random Fourier Features are built by Random Fourier bases of the form:

$$z_i(x) = cos(\boldsymbol{\xi_i^T} \boldsymbol{x} + b_i), \;\; \boldsymbol{\xi_i} \in \mathbb{R}^d, \;\; \boldsymbol{x} \in \mathbb{R}^d, \;\; b_i \in \mathbb{R} \tag{2.8}$$

Given a positive definite shift-invariant kernel k and its Fourier transform p, a Random Fourier Feature maps the input data into a random direction $\boldsymbol{\xi_i}$ drawn from p. Then, this line is wrapped into a unit circle in $\mathbb{R}^2$. The b_i component, which is sampled uniformly from interval $[0, 2\pi]$, rotates the circle by a random amount. This process is visually depicted in Figure 2.1.

A D-dimensional Random Fourier Feature vector can be formed by concatenating D Random Fourier bases whose respective $\boldsymbol{\xi}$ and b have been independent and identically distributed (iid) sampled. Dividing them by a normalizing constant, we obtain:

$$\boldsymbol{z(x)} = \sqrt{\frac{2}{D}}[cos(\boldsymbol{\xi_1^T x} + b_1) \cdots \ cos(\boldsymbol{\xi_D^T x} + b_D)]^T = \sqrt{\frac{2}{D}}[z_i(x) \cdots \ z_D(x)]^T \tag{2.9}$$

There are some important properties from the RFF that we can extract from [4]:

- Given Hoeffding's inequality, it can be concluded that:

  $$Pr[||\boldsymbol{z(x)^T z(x')} - k(\boldsymbol{x}, \boldsymbol{x'})|| \geq \varepsilon] \leq 2exp(-D\varepsilon^2/4)$$

  Therefore, the quality of the estimation is directly proportional to the number of Random Fourier Features being used.

- The inner product of the approximated points $\boldsymbol{\phi(x)}$ and $\boldsymbol{\phi(x')}$ is an unbiased estimator of $k(\boldsymbol{x}, \boldsymbol{x'})$.

We also know that the generalization error bound is given by $\mathcal{O}(\frac{1}{\sqrt{N}} + \frac{1}{\sqrt{D}})$ [13], meaning that larger datasets asymptotically reduce the error of the approximation.

2.3 Building hybrid kernel blocks

The learning units of our hybrid architectures are constructed by replacing the non-linear functions which happen in traditional neural networks by a feature map. More precisely, given a non-linearity g, input $\boldsymbol{X}$, weights $\boldsymbol{W}$ and bias $\boldsymbol{b}$, we translate the common fully connected layer output $\boldsymbol{H}$ as:

$$\boldsymbol{H} = g(\boldsymbol{XW} + \boldsymbol{b}) \tag{2.10}$$

into:

$$\boldsymbol{H} = \varphi(\boldsymbol{XW}) \tag{2.11}$$

Where φ is the function that maps data into the dimension the kernel k computes the inner product from. Given a positive definite shift-invariant kernel k and its Fourier transform p, we can easily build kernel layers. To sample generic Random Fourier Features we need to generate two main components:

- Matrix $\boldsymbol{\xi} \in \mathbb{R}^{(D,n_x)}$ where n_x are the dimensions of the input. As in [3], we sample from a Normal distribution $\mathcal{N}(0|\sigma^2 \boldsymbol{I_D})$, where $\boldsymbol{I_D}$ is the identity matrix of size (D, D).
- Vector $\boldsymbol{b} \in \mathbb{R}^D$ sampled from the uniform distribution $U \sim (0, 2\pi)$.

In this work we restrict our kernels to be Random Bases Functions (RBF). The RBF kernel (also known as Gaussian kernel), is generally defined as:

$$K_{RBF}(\boldsymbol{x}, \boldsymbol{x'}) = exp(-\gamma||\boldsymbol{x} - \boldsymbol{x'}||^2 \tag{2.12}$$

Given that σ^2 is the variance of the Fourier Transform of k, it is defined as:

$$\sigma^2 = 2n_x\gamma \tag{2.13}$$

We label as *kernel blocks* the concatenation of a traditional fully connected layer followed by a random feature map. The fully connected layer, however, has no activation and just outputs the dot product between the learned weights and the input. Instead of going under a non-linearity, the output of the fully connected layer is implicitly mapped into another space which hopefully is more suited for the task in hands. The sampled components $\boldsymbol{\xi}, \boldsymbol{b}$ which we use to compute the feature mapping are set once at the beggining and remain fixed during the training process.

As we want our implementation to be efficient we need to vectorize the mapping. Given an input $\boldsymbol{X} \in \mathbb{R}^{(n,n_x)}$, we compute $\boldsymbol{H} \in \mathbb{R}^{(n,h)}$ as the output of the hidden layer with h hidden units as:

$$\underline{\mathrm{H}} = \boldsymbol{XW} \tag{2.14}$$

Note that $\boldsymbol{W}$ has size (n_x, h). Then, if we apply the D-dimensional explicit mapping on the hidden layer output H, we got $\boldsymbol{Z}$, which is a matrix $\boldsymbol{Z} \in \mathbb{R}^{(n,D)}$:

$$\boldsymbol{Z} = \boldsymbol{C} * COS(\boldsymbol{H} \bigotimes \boldsymbol{\xi^T} + \boldsymbol{B}) \tag{2.15}$$

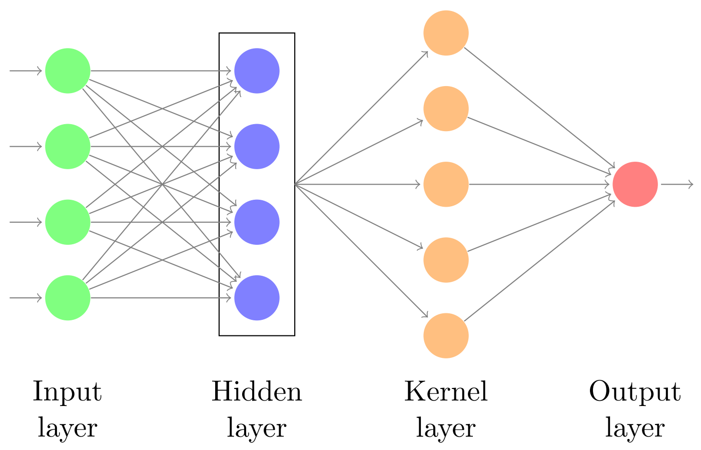


FIGURE 2.2: Example of a hybrid architecture with a single hidden layer and a kernel map

In Equation (2.15) the operator $\bigotimes$ is the matrix dot product and the operator COS performs an element-wise cosinus operation on a matrix. Matrix $\boldsymbol{B} \in \mathbb{R}^{(n,D)}$ is built so each column i contains n copies of the sampled b_i. Finally, matrix C contains copies of the constant $\sqrt{\frac{2}{D}}$ and $*$ represents the element-wise multiplication. In the code implementation we have used broadcasting[1] instead of replication for better efficiency.

Section 2.3 shows a neural network using a single kernel block. The input layer feeds the input $\boldsymbol{X}$ ino the first layer, which outputs:

$$\boldsymbol{H_1} = \boldsymbol{X W_1} + \boldsymbol{b_1}$$

The result is mapped into the approximated RBF kernel, which produces:

$$\boldsymbol{Z_1} = z(\boldsymbol{H_1})$$

Then the output of the kernel layer is fed into the output layer, which is another fully connected layer which uses a softmax activation to get the class probability distribution.

2.4 Stacked hybrid kernel architectures

Given that we want to combine the benefits of kernel methods and deep learning, we need to be able to train architectures of larger depth using the blocks defined in the previous section. We can create deep networks by simply concatenating the basic hybrid

[1]https://www.tensorflow.org/performance/xla/broadcasting

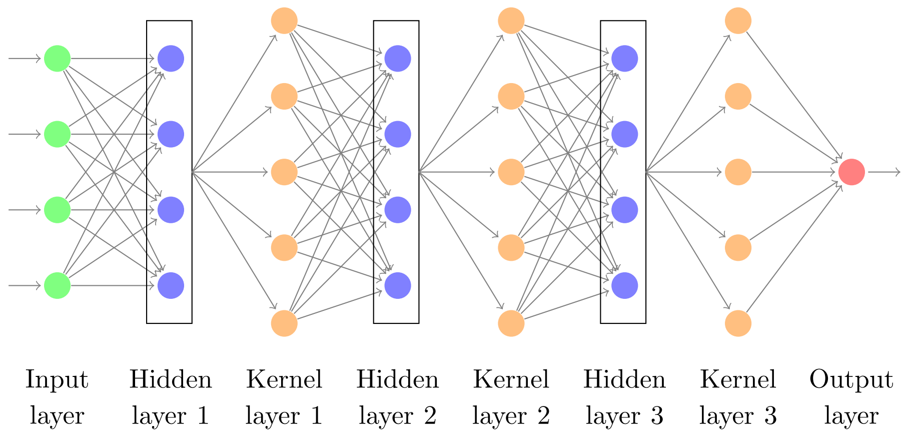


FIGURE 2.3: Example of a hybrid architecture with three concatenated kernel blocks

kernel blocks stacked one after another. This schema resembles the usual kernel machine structure, where first the data is mapped into a high dimensional space, which is fed into a linear classifier or regressor. In our case, the feature mapping is composed by a set of iterative feature maps and the linear model at the end is represented by the output layer of the network, which is the only layer which does not become hybrid.

Given a kernel block i producing $\boldsymbol{Z_i}$, it receives the RFF mapping $\boldsymbol{Z_{i-1}}$ generated by the previous block $i-1$ (i.e. considering input data as Z_0). The linear classification task is performed after concatenating s blocks amd finally $\boldsymbol{Z_s}$ is fed into the output layer. Section 2.4 provides a visual schema of an architecture with 3 hybrid kernel blocks.

2.5 Hybrid Convolutional Neural Networks (HCNN)

The stacking procedure presented in the previous section is general enough to be easily extended to other architecture types and support heterogeneous kinds of data. It can be easily adapted to support image data, similarly to what we have reviewed in [9].

Given a network with N_c convolutional layers followed by N_{fc} fully connected layers, we define hybrid convolutional kernel architectures by placing kernel layers after each of the convolutions. We define *convolutional kernel blocks* and use them together with the *kernel blocks* defined in the previous section. As a result, we obtain architectures that contain, successively, N_c convolutional kernel blocks, N_{fc} kernel blocks and an output layer. We assume we have an image input $\boldsymbol{I} \in \mathbb{R}^{(n,h,w,c)}$ where h, w and c respectively are the height, width and channels of the n images of the batch. Then, we convolve the volume using m squared filters of size (f, f, c) and a stride of s, resulting in a volume $\boldsymbol{I'} \in \mathbb{R}^{(n,t_h,t_w,m)}$, where t_h and t_w are defined as:

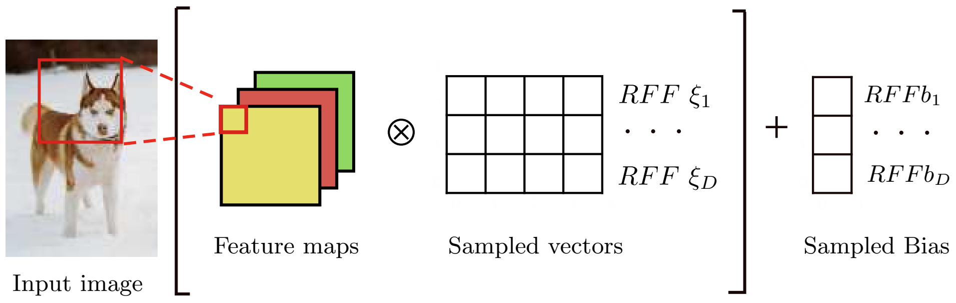


FIGURE 2.4: Visual explanation of how to apply the approximation of the kernel map using Random Fourier Features on image feature representations. The red dashed lines represent the convolution operation on an image patch, which generates feature maps. The tensor product (represented as $\bigotimes$) between these features and the sampled Random Fourier Features is performed. The result is added to the sampled Random Fourier *bias*. Though not depicted in the image, an element-wise cosinus operation followed by an element-wise multiplication by a constant is performed as we have already seen in the structured data case.
Image: *Siberian Husky* by Ritmó is licensed under CC BY 2.0

$$t_h = \frac{h - f + 1}{s} + 1 \tag{2.16}$$

$$t_w = \frac{w - f + 1}{s} + 1 \tag{2.17}$$

We assume we always use non-padded convolutions. Then we can proceed generating the D kernel features using the sampled matrices $\boldsymbol{\xi} \in \mathbb{R}^{(D,c)}, \boldsymbol{b} \in \mathbb{R}^D$ by using the tensor dot product along the second dimension defined in Equation (2.15). Note that we do not use pooling layers after our convolutions due to the fact that empirical studies have shown that strided convolutions can perform as good as pooling layers [14].

We name Convolutional networks using hybrid convolutional blocks as Hybrid Convolutional Neural Networks (HCNN).

2.6 Hybrid architectures vs traditional neural architectures

In these final section of the chapter we want to get insights of the differences between traditional neural networks and the hybrid presented architectures. We can easily observe that there is an increase on the set of hyperparameters to tune in the hybrid case: we need to choose the number of Random Fourier Features to sample and their parametrization (i.e. the scale of the normal distribution, in the Gaussian case).

Hybrid layers are also more computational expensive both in the forward and the backward pass. During the forward propagation, since they need to compute the explicit mapping operation, an additional dot product and an element-wise cosinus and multiplication (as seen in Equation (2.15)) are performed.

Considering we have a two-layered layer neural network N_r built concatenating a feedforward layer and an output layer and a two-layered layer hybrid neural network N_k formed by a kernel block and an output layer, we review some equations that are shared in both architectures for a particular classification problem. Given that we name $\boldsymbol{Z^{[l]}} \in \mathbb{R}^{(n,h)}$ to the result of the dot product between the weights at layer l and the input of the layer and $\boldsymbol{A^{[l]}} \in \mathbb{R}^{(n,h)}$ is the output of the layer l (e.g. after a non-linearity), we have:

$$\mathcal{L}(\boldsymbol{A^{[2]}}, \boldsymbol{Y}) = \frac{1}{n}\sum_{i=1}^{n} \log(A_i^{[2]})Y_i - \log(1 - A_i^{[2]})(1 - Y_i)$$

$$\boldsymbol{A^{[2]}} = \sigma(\boldsymbol{Z^{[2]}})$$

$$\boldsymbol{Z^{[2]}} = \boldsymbol{A^{[1]}}\boldsymbol{W^{[2]}} + \boldsymbol{b^{[2]}}$$

$$\boldsymbol{Z^{[1]}} = \boldsymbol{X}\boldsymbol{W^{[1]}} + \boldsymbol{b^{[1]}}$$

Where $\boldsymbol{Y}$ is the label information, σ is the softmax function and $\boldsymbol{W^{[l]}}, \boldsymbol{b^{[l]}}$ are the respective weights and biases for layer l. The difference between N_k and N_r resides in how they compute the *activation* of the hidden layers. A traditional neural network would do (i.e. let's take previous layer one as an example):

$$\boldsymbol{A_r^1} = \sigma_r(\boldsymbol{Z^{[1]}})$$

Function σ_r is typically a ReLU. Our hybrid layers would instead:

$$\boldsymbol{A_k^1} = \varphi(\boldsymbol{Z^{[1]}}) = \sqrt{\frac{2}{D}}[cos(\boldsymbol{Z^{[1]}}\boldsymbol{\xi_1^T} + b_1), ..., cos(\boldsymbol{Z^{[1]}}\boldsymbol{\xi_D^T} + b_D)]^T$$

Therefore, the activation output of any kernel layer is bounded in the interval $[-\sqrt{\frac{2}{D}}, \sqrt{\frac{2}{D}}]$. We will generally see how N_k performs more operations than N_r in their hidden layer activations. Modern neural networks rely on simple non-linearities (e.g. ReLU or ReLU variations such as Leaky ReLu) while the presented kernel blocks use an additional dot product.

Analysing the backward pass in the backpropagation step, we can also get some insights. The update term for $\boldsymbol{W}^{[\mathbf{1}]}$ at time t is:

$$\boldsymbol{W}_{\boldsymbol{t}}^{[\mathbf{1}]} \leftarrow \boldsymbol{W}_{\boldsymbol{t}}^{[\mathbf{1}]} - \alpha \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{[\mathbf{1}]}}$$

Where $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{[\mathbf{1}]}}$ is:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{[\mathbf{1}]}} = \frac{1}{n} \boldsymbol{X}^{\boldsymbol{T}} \frac{\partial \mathcal{L}}{\partial \boldsymbol{Z}^{[\mathbf{1}]}}$$

The updated value depends on the inputs and the derivative of the loss with respect to the pre-activation of the first hidden layer. In N_r, this partial derivative is defined as [15]:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{Z}^{[\mathbf{1}]}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{A}^{[\mathbf{1}]}} \frac{\partial \boldsymbol{A}^{[\mathbf{1}]}}{\partial \boldsymbol{Z}^{[\mathbf{1}]}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{Z}^{[\mathbf{2}]}} \boldsymbol{W}^{[\mathbf{2}]\boldsymbol{T}} * \sigma_r'(\boldsymbol{Z}^{[\mathbf{1}]}) = \frac{\partial \mathcal{L}}{\partial \boldsymbol{Z}^{[\mathbf{2}]}} \boldsymbol{W}^{[\mathbf{2}]\boldsymbol{T}}$$

Where $*$ is the element-wise product and we assume that σ_r is a ReLU non-linearity (i.e. derivative is 1 whenever the function is bigger than zero). In the case of N_k, we would have:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{Z}^{[\mathbf{1}]}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{A}^{[\mathbf{1}]}} \frac{\partial \boldsymbol{A}^{[\mathbf{1}]}}{\partial \boldsymbol{Z}^{[\mathbf{1}]}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{Z}^{[\mathbf{2}]}} \boldsymbol{W}^{[\mathbf{2}]\boldsymbol{T}} * -\sqrt{\frac{2}{D}} SIN(\boldsymbol{Z}^{[\mathbf{1}]} \boldsymbol{\xi}^{[\mathbf{1}]} + \boldsymbol{b}^{[\mathbf{1}]}) \boldsymbol{\xi}^{[\mathbf{1}]}$$

We can appreciate that the backward pass is also more computationally expensive, specially compared to architectures using ReLU activations.

Despite being more computationally expensive, we expect this architectures to provide rich representations by approximating infinite-width kernel layers.

2.6.1 Infinite kernel maps

As seen in Section 2.1.1, any function that represents an inner product in some space is a valid positive definite kernel. The kernel layer estimates the original kernel map $K(\boldsymbol{x}, \boldsymbol{x'})$ by using a set of Random Fourier Features. Though the amount of these features are limited, the kernel that they approximate has infinite dimensions. We are going to prove this point in this section.

The kernel trick is a very popular technique that allows us to compute similarities between two points mapped into a high dimensional space without needing to compute the map explicitely. This can be visualized with a simple example (extracted from [12]).

Assuming we have a pair of two-dimensional points x an x' and a function:

$$k(\boldsymbol{x}, \boldsymbol{x'}) = (1 + \boldsymbol{x^T} \boldsymbol{x'})^2$$

We can ensure that it is a valid kernel if it is an inner product in some space. If we expand this, we obtain:

$$k(\boldsymbol{x}, \boldsymbol{x'}) = (1 + \boldsymbol{x^T} \boldsymbol{x'})^2 = 1 + x_1^2 x_1^{2'} + x_2^2 x_2^{2'} + 2x_1x_1' + 2x_2x_2' + 2x_1x_1'x_2x_2'$$

It is easy to see that this is a valid kernel where inputs have been mapped into a higher dimensional space with a function φ:

$$\varphi(\boldsymbol{x}) = (1, x_1^2, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2)$$

$$\varphi(\boldsymbol{x'}) = (1, {x_1'}^2, \sqrt{2}x_1', \sqrt{2}x_2', \sqrt{2}x_1'x_2')$$

Indeed, the presented function corresponds to the polynomial kernel of degree 2. Similarly, we can try to replicate this experiment with the Gaussian kernel K_{RBF}:

$$k_{RBF}(\boldsymbol{x}, \boldsymbol{x'}) = exp(-\gamma||\boldsymbol{x} - \boldsymbol{x'}||^2)$$

For simplification, we will assume $\gamma = 1$ and that x and x' are one-dimensional. If we expand the previous equation, we obtain:

$$k_{RBF}(x, x') = exp(-x^2)exp(-{x'}^2)exp(2xx')$$

Replacing the term $exp(2xx')$ by its Taylor series expansion, we get:

$$\begin{aligned} k_{RBF}(x, x') &= exp(-x^2)exp(-x'^2)\textstyle\sum_{n=0}^{\inf} \frac{2^n x^n x'^n}{n!} \\ &= exp(-x^2)exp(-x'^2) + exp(-x^2)exp(-x'^2)2xx' \\ &\quad + exp(-x^2)exp(-x'^2)2x^2x'^2 + exp(-x^2)exp(-x'^2)\tfrac{4}{3}x^3x'^3... \end{aligned} \tag{2.18}$$

This can be reorganized as a dot product in an infinite space generated by the map φ_{RBF}:

$$\varphi_{RBF}(x) = (exp(-x^2), exp(-x^2)\sqrt{2}x, exp(-x^2)\sqrt{2}x^2, exp(-x^2)\sqrt{\frac{4}{3}}x^3, ...)$$

$$\varphi_{RBF}(x') = (exp(-x'^2), exp(-x'^2)\sqrt{2}x', exp(-x'^2)\sqrt{2}x'^2, exp(-x'^2)\sqrt{\frac{4}{3}}x'^3, ...)$$

As we can see, the explicit map of each point has infinite dimensions. Therefore, we can say that the maps in our kernel blocks approximate a kernel function that is an inner product in an infinite space.

Chapter 3

Training deep hybrid architectures

3.1 When to stop?

The functions that neural network tend to optimize are quite complex, usually hard to optimize and depend on many heuristics (i.e. hyperparameters). The empirical evolution of the training and validation performance is generally not monotonic and tend to oscillate through time. Therefore, it is hard to know the are where the model lies at a particular step or wether it is going to improve or not in the next epochs. Even though both training and validation errors are intended to asymptotically decrease during training, whenever the model starts to *memorize* the training data the validation error suddenly raises. This due to the degradation of the generalization power of the model due to *overfitting*. Overfitting can be dealt with several regularization techniques (e.g. dropout, weight decay) which will be reviewed in .

One of the strategies to deal with overfitting is *early stopping*. It consists of stopping the training process whenever the validation error starts to increase. The training and validation performance are monitored and the model with the smallest generalization error is kept at each epoch. When the validation error is representative enough (i.e. it is drawn from the same distribution as the test data) it becomes an unbiased estimator of the generalization error. Therefore, having good results on the validation error should be a strong guarantee of a general model.

Additionally, early stopping is a very suited technique for training neural models without much human intervention, since it allows to stop the process whenever the training improvements is too small (i.e. model convergence) or when it degradates too much.

Another advantage of early stopping is that we do not need to provide the number of epochs or steps to train but just set an upper bound.

The main disadvantage of *early stopping*, as presented in [16], is that it is not an orthogonal technique. This means that it does not tackle a single independent relevant factor of the training process but more than one, creating an obvious obvious trade-off:

- Allowing bigger degradation of the validation error allows for longer training and the chance to find better solutions.
- Restricting the worsening of the validation error we can reduce the training time.

The biggest challenge when using early stopping is deciding when to stop the training process. When we use mini-batch training, it is normal to see both errors oscillate since the gradients computed at a mini-batch level are an estimation of the error on the whole batch. Consequently, validation error alternatiely increases and decreases and it becomes hard to decide whether we are losing generalization power or observing random error oscillations due to the approximate nature of the process.

Work by Prechelt [17] shows several systematic and reliable ways to decide when training should stop. From those, we have outlined a stopping strategy for our work, which we detail in next section.

3.1.1 Stopping criteria: successive strips and training progress

From all the techniques presented in [17] we select the one that provides better average quality solutions, called UP_s *criteria*. This criteria stops once the estimation of the generalization error has increased in s successive strips. More formally, it is defined it as:

UP_S: stop after epoch t iff UP_{s-1} stops after epoch $t-k$ and $E_{va}(t) > E_{va}(t-k)$

UP_1: stop after first end-of-strip epoch t with $E_{va}(t) > E_{va}(t-k)$

Where $E_{va}(t)$ is the validation error after epoch t. We use $s = 3$ successive strips error increases before stopping. This criteria, however, does not guarantee termination (e.g. training error reaching 0.0 and getting constant validation errors). Therefore, we need to add an additional stopping criteria based on the *progress* measure, which is defined as:

$$P_k(t) := 1000 \frac{\sum_{t'=t-k+1}^{t} E_{tr}(t')}{k \times min_{t'=t-k+1}^{t} E_{tr}(t')} - 1$$

Where $E_{tr}(t)$ is the training error after epoch t and k is the length of the strip. If the progress drops below $p_t = 0.1$ we stop training. In order to restrict the total number of iterations we also set an upper bound for the total number of epochs to T_{it}.

In this work we define strips of length 5. At the end of each epoch t which is multiple of $k = 5$ we evaluate whether to stop according to UP_t. Additionally, we stop if progress $P_k(t)$ is below $p_t = 0.1$ or whenever the epoch counter t has reached T_{it}. The model state at the epoch which results in the better performance is chosen as the best candidate for a particular hyperparameter setting. This procedure is formalized in algorithm 1.

Algorithm 1 Early stopping criteria

```
1: $k \leftarrow$ strip length
2: $p_t \leftarrow$ progress threshold
3: $T_t \leftarrow$ maximum epoch
4: $s \leftarrow$ maximum successive error increases
5: $t \leftarrow 1$
6: $fails \leftarrow 0$
7: $e_{best} \leftarrow inf$
8: $t_{best} \leftarrow 0$
9: while $t \leq T_t$ do
10:    train_epoch()
11:    if $t \ mod \ k == 0$ then
12:       if $E_{va}(t) < e_{best}$ then
13:          $e_{best} \leftarrow E_{va}(t)$
14:          $t_{best} \leftarrow t$
15:          save_model()
16:       end if
17:       if $E_{va}(t) > E_{va}(t-k)$ then
18:          $fails \leftarrow fails + 1$
19:          if $fails == s$ then
20:             stop
21:          end if
22:       else
23:          $fails \leftarrow 0$
24:       end if
25:    end if
26:    if $P_k(t) \leq p_t$ then
27:       stop
28:    end if
29: end while
```

3.2 Layerwise training procedures

Following the intuition that kernel layers can be tuned easily in an incremental or iterative fashion and as we have already observed in some of the works in the literature (e.g. [8]), we define a set of layerwise training procedures for our hybrid architectures. Layerwise training can boost the training performance by optimizing the cost function only on a subset of the network parameters at a time in a similar way Coordinate Descent does. These methods are also reminiscent of extreme machine learning techniques, in which iterative training strategies have also been proposed [18].

We propose three varieties of layerwise training which we present in the following subsections. Note that we will talk about layer throughout the descriptions of the methods since they are general training methods for any kind of neural architecture. However, we will refer to kernel blocks instead of layers in the rest of this work.

3.2.1 Incremental Layerwise Training (ILT)

Out of the 3 strategies we are proposing this is the only one that do not require a fixed number of layers beforehand but an upper bound.

This training procedure iteratively stacks new layers as long as they make the generalization error decrease. We start by training a single layer and the output layer. Afterwards, a second layer (which is randomly initialized) is placed between the first layer and the output layer. Both the new layer and the output layers are initialized randomly. Then, only the parameters related to the newly added layer and the ones belonging to the output layer are updated, while the rest of the parameters of the networks are kept fixed.

This is repeated until the maximum number of layers is reached or as soon as a new layer makes the validation error increase with respect to the best error achieved training the previous layer. Given that after adding a new layer i we get a decrease in the validation performance, the best model for the ILT procedure corresponds to the best performing model from the layer $i-1$. Each of the layers trained using this procedure follow the early stopping criteria presented in Section 3.2.3. After stopping, if the best model found for the new layer outperforms improves the previous found models, we keep adding layers. Otherwise, we select the best model that uses $i - 1$ layers. The ILT process can also stop whenever the maximum number of epochs is reached.

This procedure is formalized in Algorithm 2.

Algorithm 2 Incremental Layerwise Training procedure

```
1: $e_{best} \leftarrow$ best error up to layer i
2: $t_{best} \leftarrow$ best epoch found
3: $l_{best} \leftarrow$ best layer found up
4: $maxl \leftarrow$ maximum number of layers
5: $i \leftarrow 0$
6: while $i \leq maxl$ do
7:     $E_{va}, t_{best} \leftarrow early\_stop\_training(i)$
8:     if $E_{va} > e_{best}$ then
9:         stop
10:    else
11:        $e_{best} \leftarrow E_{va}$
12:        $l_{best} \leftarrow i$
13:        save_model()
14:    end if
15:    $i \leftarrow i + 1$
16: end while
```

3.2.2 Cycling Layerwise Training (CLT)

This nest procedure is a derivation of the ILT presented in the previous section. In this case we need to know the number of layers we want to train beforehand and all layers are present in the network from the beginning. Again, we train each layer separately using an early stopping fashion for switching between layers.

At start time, all layers are initialized randomly and one of the layers is chosen to be the one that we start to train. Only the parameters involving that layer and the output layer are updated. Once early stopping decides to stop training a layer, we switch to the next one. The output layer is always updated. Given a neural network with l layers, we name as a cycle the process of training l layers (which may be repeated) in a row.

The criteria for stopping CLT is another early stopping strategy on top of one that controls the training evolution of each layer. This additional regularizer monitors the errors at the end of each cycle: if the progress of the training error is too low or the validation error has increased with respect to the previous cycle, CLT stops and retrieves the best model found.

We define three different policies in order to define how to switch between layers:

- **Cyclic policy**: A cycle starts training the first layer and ends when the training of layer l is completed.
- **Inverse cyclic policy**: Intuitively, the cycles using this policy start with layer l and end at layer 1.

- **Random policy**: Layers are trained in no specific order and one layer can be trained more than once in the same cycle. We have provided an initial seed to this policy in order to make it deterministic when needed.

The pseudo code for CLT is detailed in Algorithm 3.

Algorithm 3 Cycling Layerwise Training procedure

```
1: $p_{layert} \leftarrow$ progress threshold
2: $T_t \leftarrow$ maximum epoch
3: $policy \leftarrow$ layer switching policy
4: $t \leftarrow 0$
5: $e_{prev} \leftarrow$ error at the end of the previous cycle
6: $e_{prev} \leftarrow$ inf
7: while $t \leq T_t$ do
8:     $l \leftarrow$ get_current_layer(policy)
9:     train_epoch(l)
10:    if $cycle\_ended()$ then
11:        if $E_{va}(t) > e_{prev}$ then
12:            stop
13:        else
14:            if $P_k(t) \leq p_{layert}$ then
15:                stop
16:            else
17:                $e_{prev} \leftarrow E_{va}(t)$
18:            end if
19:        end if
20:    end if
21:    $t \leftarrow t + 1$
22: end while
```

3.2.3 Alternate Layerwise Training (ALT)

This last training procedure we are presenting is a slight variation of the CLT where the switching between layers is performed after a fixed number of epochs. The idea behind this training procedure is to reduce the time we switch between layers during training. One more time, only the layer under the focus and the output layer are trained at each iteration. In this case, we use a single global early stoppig regularizer which works the same way as the base strategy defined in .

The policies for switching between layers are the 3 same policies for the previous technique: *cyclic policy*, *inverse cyclic policy* and *random policy*. A special trait of this algorithm is the fact that the layer we train at epoch t is deterministic given the switching policy.

3.3 Hyperparameter search

Multilayer feed-forward networks have been regarded as universal approximators [19] under very general conditions (e.g. given that sufficient hidden units are available). This means that a wide range of functions can be represented using a neural networks. However, not all the functions that a network can represent are actually learnable in practise due to the complexity of the optimization. We depend on many choices such as the regularization of the model, its capacity or the choice of the cost function. Since many hyperparameters need to be tuned, several strategies have been devised in order to obtain the model with the lowest generalization error under existing external constraints (e.g. hardware, time). We introduce here a small survey on common hyperparameter tuning techniques:

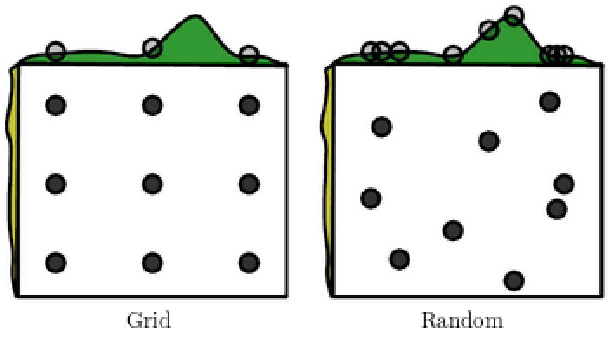


FIGURE 3.1: Simplification of how grid search and random search compare to each other. The picture shows 9 trials for 2 hyperparameters, where each dot represents a single independent trial. The green plots on each axis show the variation on the performance (e.g. accuracy) of the model. Grid search (left) explores a discretized subset of the whole search space, which grows exponentially with the number of hyperparameters. It is common that many of the hyperparameter dimensions do not contribute to improve the performance of the model and therefore, we spend computing resources exploring regions of the search space which are not relevant (i.e. vertical axis in the image). It has been empirically proven that random exploration of the search space can asymptotically provide the same results as the grid search. Image source: [1]

- Manual hyperparameter tuning. This strategy requires a high expertise of how the hyperparameters influence the behavior of the network. The modeller needs to be familiar diagnosing these types of processes and requires monitoring the evolution of both the training and the generalization error and tune the parameters accordingly whenever we observe underfitting or overfitting. For instance, if a model is underfitting the data, the modeller should increase the network capacity in order to facilitate the learning tasks of higher complexity. This can be done, by increasing the number of hidden layers. It is a good practise to achieve low training error first by using networks of high capacity and then apply regularization techniques in order to gain generalization power while keeping the training error low. Manual tuning is effective but requires a lot of manual intervention which is sometimes not feasible when handling projects where lots of different experiments need to be performed.

- Automatic hyperparameter tuning. They can be divided into two strategies:
  - **Hyperparameter optimization**. These approaches map the hyperparameter search problem into another optimization problem itself. There are methods that are gradient-based (i.e. compute gradient of the validation set with respect to the hyperparameters) though in many cases this gradient is not possible or feasible to compute. We can find an alternative to the gradient-based hyperparameter optimization in Bayesian optimization, though its effectiveness highly depends on the problem and it is not suitable as a general purpose procedure.
  - **Search**. Based on searching the best configuration among all the possible settings. There are two main methods:
    - **Grid search**. It is the most common method traditionally used in machine learning, suited for optimizations that have few hyperparameters. Explores the whole search space as the Cartesian product of all the individual hyperparameter search spaces and picks the best configuration. The main problem with this strategy is that its computational cost grows exponentially with the number of parameters and requires $\mathcal{O}(n^m)$ trials, where n is the number minimum number of values each of the m hyperparameters can take.
    - **Random search**. Work in [20] shows how, for each algorithm and dataset, only a few of the optimization dimensions (i.e. hyperparameter space) are actually effective. This means that many of the trials performed in a grid search explore subspaces which are not relevant to the specific problem. This is visually depicted in Figure 3.1. It also states that given a fixed domain, random search can perform as good or better than grid search using less computational time. For random search, instead of defining a discretized search space for each hyperparameter, we sample from a continuous or discrete domain given a distribution according to the nature of the hyperparameter. We have adopted Random Search for our experiments. More details can be found in Section 6.3.

3.4 Optimizing hybrid architectures

The cost function is one of the most important choices for a learning problem. This function needs to be minimized by the network and should be representative of the task that we pretend to optimize. We have focused on classification problems, using following cost function in our trials:

$$\mathcal{J}(\boldsymbol{\theta}|\boldsymbol{X},\boldsymbol{Y}) = \sum_{i=1}^{N} \frac{1}{N}\mathcal{L}(f(X_i), Y_i) + \lambda\Omega(\boldsymbol{\theta}) \tag{3.1}$$

Variable θ represents the set of parameters of the network computing the function f and $\boldsymbol{X}$ and $\boldsymbol{Y}$ are respectively the data and labels. The function jointly minimizes the cross entropy loss of the data (i.e. first term) and the regularization term. The cross entropy loss is defined as:

$$\mathcal{L}(f(X_i), Y_i) = -\sum_{j=1}^{N_c} \log_2(f(X_i)^j) * Y_i^j \tag{3.2}$$

Note that $f(X_i) \in \mathbb{R}^c$ are the network output probability for each of the N_c possible labels given the input instance $X_i \in \mathbb{R}^{n_x}$ while Y_i is the one-hot vector of the correct label. The value $f(X_i)$ is the activation of the output layer of the function, which can be a sigmoid activation for binary problems or a softmax function for multiclass problems.

The second term in Equation (3.1) is the regularizer, which limits the value of the matrices $\boldsymbol{W_i} \in \subseteq$ and helps prevent overfitting. We use L2 the penalty term:

$$\Omega(\boldsymbol{\theta})_{L2} = \frac{1}{2}||\boldsymbol{W}||_2^2 = \frac{1}{2}\sum_i \boldsymbol{W_i}^2 \tag{3.3}$$

The biases are ignored for regularization. The sampled values in the kernel layers are also ignored since they are not trained.

We train these networks using gradient-based algorithms based on stochastic gradient descent, where the gradient of the data is approximated by the gradients in small batches of the training data.

3.5 Dealing with the shift invariance: batch normalization

One of the main challenges in the neural network research has been to achieve stable and effective learning in deep architectures. Internal covariate shift is an important driver that makes training difficult. This phenomenon is the variation of the distribution of the layer inputs during training due to the modification of the parameters of the previous layers. It usually forces using small learning rate for stability. Ioffe and Szegedy [21] introduced batch normalization, which allows us to train neural network in a more stable way using bigger learning rates and more general initialization.

The experiment design of our work is mainly restricted by our limited computing resource. This limitation forces us to deal with moderately deep architectures. During the experiments we observed that in spite of dealing with networks of moderate depth can anyway observe how the covariate shift effect is present in relatively small layouts.

Batch normalization reduces the dependency between the success of the training process and the scale of the gradient or the initialization of the network parameters. It does so by fixing the mean and the variance of the input of each hidden layer using an estimate of these statistics from each independent mini-batch. In order to adapt this normalization step for any kind of activation function (i.e. so identity function can also be represented) a couple of parameters are added and are used to scale and shift the normalized version of the inputs.

Given a batch of examples $\boldsymbol{X} = \{\boldsymbol{x_1}, \boldsymbol{x_2}, ..., \boldsymbol{x_n}\}$ where $\boldsymbol{x_i} = \{x_i^1, x_i^2, ..., x_i^m\}$, the mini-batch approximation of the Batch normalization function $BN_{\gamma,\beta} : \mathbb{R}^{(n,m)} \to \mathbb{R}^{(n,m)}$ is defined as:

$$BN_{\gamma,\boldsymbol{\beta}}(\boldsymbol{x_i}) = \boldsymbol{\gamma}\hat{\boldsymbol{x_i}} + \boldsymbol{\beta}$$

Where $\hat{\boldsymbol{x_i}}$ is the normalized version of the input $\boldsymbol{x_i}$:

$$\hat{\boldsymbol{x_i}} = \frac{\boldsymbol{x_i} - \boldsymbol{\mu_\beta}}{\sqrt{\boldsymbol{\sigma_\beta^2} + \epsilon}}$$

Note that the value ϵ is introduce to avoid divisions by zero. Values $\boldsymbol{\sigma_\beta^2}$ and $\boldsymbol{mu_\beta}$ respectively are the variance and the mean of the mini-batch $\boldsymbol{X}$:

$$\boldsymbol{\mu_\beta} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x_i}$$

$$\boldsymbol{\sigma^2_\beta} = \frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{x_i} - \boldsymbol{\mu_\beta})^2$$

Parameters $\boldsymbol{\gamma}, \boldsymbol{\beta}$ are to be learnt through gradient descent since the function is differentiable. These parameters are used during inference time and kept fixed together with the moving average of the mean and the variance of the training population.

The original paper places the batch norm step between the dot product of the hidden layer and its activation, though other options have been studied. The best location of the batch norm layer can depend depend on factors such as the specific problem to solve or the architecture being used [22]. Given that we apply Batch Normalization to a hidden layer $\boldsymbol{H_i}$ with weights $\boldsymbol{W_i}$, biases $\boldsymbol{b_i}$ and activation function g, we would have, after applying the normalization to our batch $\boldsymbol{X}$:

$$\boldsymbol{H_i} = g(BN_{\gamma,\boldsymbol{\beta}}(\boldsymbol{X}\boldsymbol{W_i} + \boldsymbol{b}))$$

The effect of batch norm is maximized when the intensity of other regularization techniques is lowered (e.g. remove dropout, reduce L2 loss term).

Chapter 4

Regularization

Regularization is the technique aimed at reducing overfitting. Overfitting occurs when the model memorizes the details of the data instead of the patterns, resulting in non-generalizable solutions. Usually, overfitting occurs when the complexity of the model chosen is higher than the one the underlying problem requires and it is common in cases where data is scarse.

In this section we give a summary of available regualization techniques for neural networks and present two specific regularization techniques for hybrid neural networks.

4.1 Regularization in deep learning architectures

4.1.1 Norm penalty

Deep learning cost functions usually have the following form:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{X}, \boldsymbol{Y}) = J(\boldsymbol{\theta}, \boldsymbol{X}, \boldsymbol{y})$$

Where J is the objective function. One of the most common regularization techniques is to impose an extra term in the cost function that penalizes the complexity of the model, such as:

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{X}, \boldsymbol{Y}) = J(\boldsymbol{\theta}, \boldsymbol{X}, \boldsymbol{Y}) + \alpha\gamma(\boldsymbol{\theta})$$

The parameter α measures the contribution of the regularization to the cost function, and it is usually tuned as another hyperparameter. Function γ is a function of the model parameters and forces the optimization process to prefer those solutions which are simpler from a complexity perspective. This is usually translated in controling the magnitude of the weights of the network. Biases are typically not regularized because they require less data to be fit properly [1].

Different functions can be applied as regularizer functions which can produce slightly different results. The most common options are L2 and L1 regularization. L2 regularization (also known as weight decay or ridge regression) imposes the following penalty:

$$\gamma(\boldsymbol{\theta})_{L2} = \frac{1}{2}||\boldsymbol{W}||_2^2 = \frac{1}{2}\sum_i \boldsymbol{W_i^2}$$

The effect of L2 regularization is to mantain non-relevant dimensions near zero while preserving the value of those dimensions that are crucial for the problem.

On the other hand, L1 regularization uses the following penalty:

$$\gamma(\boldsymbol{\theta})_{L1} = \frac{1}{2}||\boldsymbol{W}||_1 = \sum_i |\boldsymbol{w_i}|$$

This regularization term penalizes the absolute value of the weights instead of its squared value. Empirically, it has proven to have a similar effect to feature selection since it sets to zero those dimensions which are not relevant.

4.1.2 Noise insertion

Noise insertion is a technique designed to achieve more robust networks. It consists of injecting random noise to components of the network such as the inputs, the hidden layers or even the targets. A typical form of noise injection is inserting random perturbations in the model weights that empowers the stability of the model to be learnt. As stated, it is also common to see noise injection in the targets to prevent the model from learning from mislabelled examples. Additionally, label smoothing can be applied to classification labels replacing the strict ones and zeros from the groundtruth data by data between zero and one, reflecting some uncertainty in the labelled data and helping the model converge easier.

4.1.3 Dropout

Dropout is a regularization technique that has become a strong standard in neural networks. A natural way to prevent overfitting is to average over several predictions from separated neural networks. However, tuning, training and predicting in multiple large neural networks can be very prohibitively expensive. Srivastava et al. [23] presented an efficient way to average the predictions of many networks by randomly dropping connections (Figure 4.1) in a single network at each training step with a probability of p.

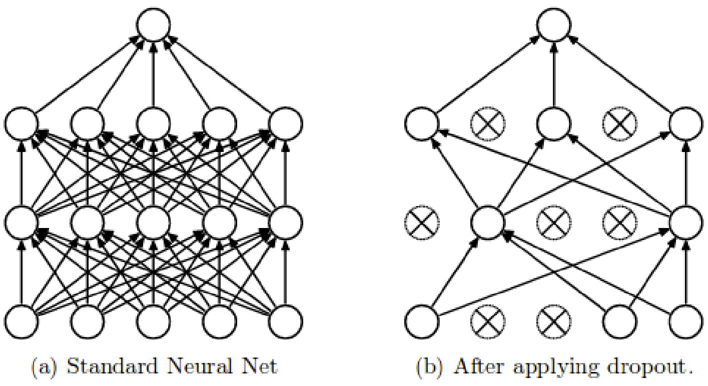


FIGURE 4.1: Visual explanation of dropout. It is equivalent to train a exponentially large amount of different networks which share the same weights. On the left, we see a traditional neural network architecture. On the right, we observe that some units have been randomly dropped and therefore, are not contributing in either the forward or the backward pass.

Consider a given a neural network with hidden layers $l \in \{1, \ldots, L\}$ and let $\boldsymbol{W^l}$ and $\boldsymbol{b^l}$ the weights and the biases at layer l. Let $\boldsymbol{z}^l$ the preactivation value in layer l and $\boldsymbol{h^l}$ be the output at layer l. the output of a layer without dropout is defined as:

$$z_i^{l+1} = \boldsymbol{w_i^{l+1}} \boldsymbol{h^l} + b_i^{l+1}$$

$$h_i^{l+1} = \sigma(z_i^{l+1})$$

Function σ is the chosen activation function. With dropout, this becomes:

$$r_j^l \sim Bernoulli(p)$$

$$\boldsymbol{\hat{h^l}} = \boldsymbol{r_j^l} * \boldsymbol{h^l}$$

$$z_i^{l+1} = \boldsymbol{w_i^{l+1}} \boldsymbol{\hat{h^l}} + b_i^{l+1}$$

$$h_i^{l+1} = \sigma(z_i^{l+1})$$

We can see how dropout adds a new hyperparameter that defines the probability of a single unit to be kept or dropped. A Bernoulli distribution is generated for each parameter at each training step. If the unit is dropped, then the ouput of that unit is and so is its contribution to the gradient. From a theoretical perspective, dropout can be seen as training a set of 2^n networks which share weights and where we train each one a very small amount of time.

Dropout modifies the expected value of the new network activations $\hat{\boldsymbol{h_i}}$:

$$\mathbb{E}(\hat{\boldsymbol{h_i}}) = p\boldsymbol{h_i}$$

During inference, it is not efficient to perform exponentially large different predictions. A fair approximation is to keep all weigths during test time and down-scale them by the parameter p so the expected value of a layer is the same for training and test.

As a result of this process, the activations of the hidden units of a neural networks using dropout become sparse. The main drawback of dropout is the introduction of a trade-off between regularization and training time as networks with dropout are 2 or 3 times more expensive to train than regular ones.

4.1.4 Ensemble methods

When computing resources are available, one of the best techniques to avoid overfitting is using model averaging techniques. In these approaches, several models are trained and they separately *vote* for a label in each test example, acting as an ensemble. The performance of the ensemble is linearly dependent to the number of models contained.

Ensembles can be produced in different ways. A common example is to use *bagging*, where training data is sampled obtaining datasets of the same size of the training set using replacement. Therefore, each model is provided with a data set that contains duplicated data and misses a subset of the total original data. *Bagging* provides models which are inherently different because they use different samples of data during training. However, neural network naturally provide many factors of variation that can also be useful to generate models which generate independet errors, even when they have trained on the same data. Sources of variation can be random initialization, random selection of hyperparameters or random data shuffling.

Ensemble methods can also be created in an incremental way using *boosting*. In boosting, a cascade of models is built where the later models have a bigger capacity.

4.1.5 Data augmentation

The straightforward way to reduce overfitting is to provide more data to the learning algorithm. However, this is not always possible to achieve: training data may be scarce or very expensive to collect. A very effective way to increase the training set size is to synthetise fake data from the existing one. The feasibility of data augmentation is problem dependent and relies on the ability to generate fake data in a particular task.

Data augmentation has been proven specially effective in object detection tasks, where new data can be generated by using common transformations on the training such as image illumination and color changes, random cropping, rotation or flipping. These transformations represent factors of variations we want to be invariant to and that can naturally arise in the data anyway.

4.1.6 Other methods

We have reviewed some of the most common regularization techniques used in deep learning. However, there are many more techniques which can be found in the literature. We have already reviewed in Section 3.2.3 already early stopping for regularization. There are methods such as *max norm regularization* that clamps the weight vector to have an upper bound on their norm. An extensive review of regularization for deep learning can be found in [1].

4.2 Regularization for deep hybrid techniques

The methods described in this chapter are general regularization techniques that can be applied to a wide range of learning algorithms. In this capter we present two regularization techniques for networks using kernel mappings using Random Fourier Features.

4.2.1 Random Fourier Features resampling

This first technique, that we label as Random Fourier Features Resampling (RFFS) gets its inspiration on dropout. It consists of resampling the Random Fourier vectors that are used to compute the kernel map at specific moments of the training with a probability of p_r. Given the matrix of D Random Fourier Features $\xi \in \mathbb{R}^{(D,n_x)}$ sampled from a Gaussian distribution $\mathcal{N}(0|\sigma^2 \boldsymbol{I_D})$, where $\boldsymbol{\xi^i}$ is the $i-th$ feature, we can formally define it as follows:

$$\boldsymbol{\tau^i} \in \mathbb{R}^{n_x},\ \tau^i_j = \upsilon \text{ for } j = 1, \ldots, n_x$$

$$\boldsymbol{\kappa^i} \in \mathbb{R}^{n_x}, \kappa^i_j = 1 - \upsilon \text{ for } j = 1, \ldots, n_x$$

$$\upsilon \sim Bernoulli(p_r)$$

$$\boldsymbol{\xi^{i'}} \sim N(0|\sigma^2 \boldsymbol{I_D})$$

$$\hat{\xi^i} = \boldsymbol{\xi^{i'}} * \boldsymbol{\tau^i} + \boldsymbol{\xi^i} * \boldsymbol{\kappa^i}$$

We represent the element-wise product as $*$. Vector $\boldsymbol{\tau^i}$ contains n_x copies of the sampled Bernoulli variable and $\boldsymbol{\kappa^i}$ contains copies of the complementary of this sample. Probability p_r is the probability that a feature vector gets resampled. If the sample is positive, all elements in the RFF vector are resampled. Otherwise, the vector is not modified. We propose this method with the intuition that resampling a subset of the feature vectors used can help us prevent the model from memorizing the training data and improve generalization. We exclude the sampled vector $\boldsymbol{b}$ from the resampling process. The resampling is performed at the end of each epoch.

As we sample from the original distribution we do not have to perform additional actions because the expected value of the dot product between the inputs and the Random Fourier features is the same.

4.2.2 Gaussian noise injection

As we have seen, noise injection is a common practise in regularization for neural networks. We believe that we can extend this concept by adding noise to the sampled basis vectors:

$$\boldsymbol{\xi'_i} \sim N(0|\sigma^{2'} \boldsymbol{I_D})$$

$$\sigma^{2'} = \sigma^2/\lambda$$

$$\hat{\boldsymbol{\xi^i}} = \boldsymbol{\xi_i'} + \boldsymbol{\xi_i}$$

The variance of the noise σ' has been defined as a ratio of the underlying distribution, which is controlled by a parameter λ. Typical values for λ should be between 5 and 10.

We add small perturbations that we believe can help the model overfit the training data. This technique has been presented but has not been implemented. It is future work to get empirical insights of this method.

Chapter 5

Implementation

The implementation of this presented work has been done using open source tools in Python. In order to ease the reproducibility of the results presented, the implementation has been made public[1] and instructions for execution has been provided in the given repository. Most of the computation performed in our implementation is done by the Tensorflow[24] library by Google, which is presented in the next section.

There are several open tools for deep learning optimization such as Keras, Caffe or Torch. We may choose one or another depending on the needs of our project. Keras and Caffe are very good tool for fast prototyping since they handle higher level operations (i.e. they are built on top of other libraries such as Tensorflow or Theano). While other libraries such as Tensorflow or Theano are more intended for production-level code and make lowe-level computations. Since we expect to tune and modify aspects of the neural networks at a fine level, Tensorflow is a good option. IT has a huge online community, is very documented and we are already familiar with it.

5.1 Tensorflow for Python

Tensorflow is an open source library designed to solve numerical computation and optimization. It is a general purpose tool and provides interfaces for many different operations. Operations can be hosted by either CPU or the GPU, being the later usually orders of magnitude faster.

The way to define a numerical process in Tensorflow is by designing a *computational graph*, which is detailed in the next section. One of the key aspects of Tensorflow, as its competitors, is the ability to perform automatic differentiation on a computational

[1] https://github.com/DaniUPC/deep-kernel

graph. Therefore, when performing a gradient-optimization process such Stochastic Gradient Descent we do not need to manually specify the gradient of the objective function with respect to the parameters but they are automatically inferred from the computational graph using the calculus chain rule.

Tensorflow has a built-in visualization tool called Tensorboard, which is very helpful for debugging and monitoring process, visualizing the inputs and outputs of our operations or have a general picture of the computational graph defined by the framework.

There are some important elements that conform the Tensorflow framework, which are:

- Graph: Computational graph object.
- Operations: nodes in the computational graph. Have either an input or an output and perform a numerical computation task.
- Variables: Data that can be an input or an output of an operation.
- Placeholders: Interface to input data into computational graphs.
- Optimizers: They are objects that can compute gradients of a set of variables with respect to a cost function operation.
- Session: Computational context where we can perform operations in the graph.

5.1.1 Building a computational graph

In a computational graph, nodes represent operations (e.g. sum, subtraction) while edges are the inputs and outputs of these operations. A computational graph is the main component of a Tensorflow program and it just defines the computing structure. Appendix A includes an tutorial on how to build a perceptron from scratch in Tensorflow, so the reader can get familiar with the library syntaxis.

5.1.2 Data ingestion: TFRecords

Data can be fed in multiple ways in Tensorflow, including feeding the graph with on-memory values (i.e. matrices, vectors). However, large datasets are prone not to fit on memory and alternative interfaces are needed to fit our models.

Tensorflow provides a custom binary format called TFRecord which can be easily streamed over into the input pipeline. We will use this format as default for all datasets in our experiments. The conversion of the data into TFFRecord files has to be done *offline*.

We have developed a Python module[2] which serves as a wrapper to both create and read from TFRecord files.

[2]https://github.com/DaniUPC/protodata

Chapter 6

Experiments

In this chapter we introduce the details of the experiments. Section 6.1 and Section 6.2 respectively introduce the datasets and the resources used in our experiments phase. Section 6.4 reproduces the experiments for the original paper [3]. Section 6.5 describe the experiments that compare traditional neural networks and our presented hybrid architectures. We also test here our novel training procedures that we have detailed in Chapter 3. Finally, Section 6.7 introduces the experiments aimed to compare several regularization techniques on hybrid architectures.

6.1 Datasets

We have selected 3 datasets for our experiments (note that Section 6.4 uses additional datasets used in the original work).

The first two datasets containt structured data and have been extracted from the UCI Machine learning repository [25]. Their details are depicted in Table 6.1.

Dataset	Instances	Features	Classes
Motor	58509	49	11
Magic	19020	10	3

Table 6.1: Structured datasets used for our first experiment

The third dataset is called FASHION-MNIST[26] and it is a dataset of iconic clothing images.Some examples are displayed in Figure 6.1. It has the same properties as the popular MNIST: $60k$ grayscale images of size $(28, 28)$ belonging to one out of 10 categories and same training and test split. The main purpose of FASHION-MNIST is to replace

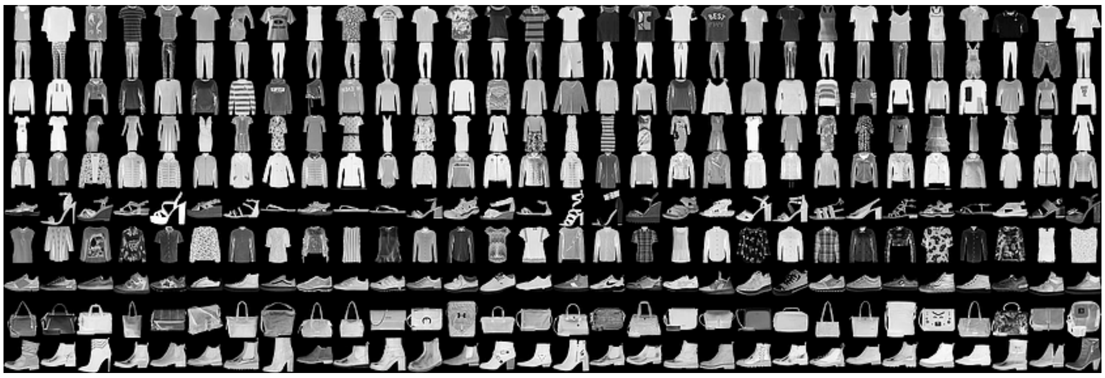


Figure 6.1: Examples from FASHION-MNIST dataset. Classes are (from top to bottom): t-shirt/top, trouser, pullover, dress, coat, sandal, shifrt, sneaker, bag and ankle foot. We can observe how these classes are harder to classify than the classic MNIST.

MNIST as the reference image dataset for benchmarking in computer vision. The authors consider that MNIST has become too easy and many models can already provide an accuracy above 99.50%. They also argue that it has been overly used (and many influencial voices in machine learning have stated so) and that it does not represent the modern computer vision tasks, which are now more complex. FASHION-MNIST then, provides a more challenging problem than the original MNIST and we think it is a good practise to start using it for benchmarking. Details about the dataset are shown in Table 6.2.

FASHION-MNIST is easily overfitted and we usually see how dropout or data augmentation is usually used in the literature for most of image datasets. For simplicity, we have not considered any extra regularization technique apart from L2 penalization and early stopping in our experiments.

Dataset	Instances	Height	Width	Channels	Classes
FASHION-MNIST	60000	28	28	1	10

Table 6.2: Fashion-MNIST details

6.2 Computing resources

For all of our experiments we used an ASUS GTX-1060 GPU with 6GB of memory running on a Linux machine with Intel Core i5-6600 at 3.30 GHz processor and 8 GB of DDR4 RAM.

6.3 Experiment design

Each of the experiments we present here focus on one specific hypothesis but they share many traits.

All experiments use a mini-batch gradient optimization using Adam algorithm [27] with exponentially decaying learning rate lr which is decayed by a factor of $decay_{lr}$ each $epochs_{lr}$ trained. We use the stopping criteria presented in Section 3.2.3 for knowing when to finish training.

While the fully connected layers from the kernel blocks have no activations, we use the ReLU activation for the fully connected of the feedforward traditional neural networks:

$$ReLU(x) = max(0, x)$$

For the classification problems which are multi-class we apply softmax layers at the end of the network and sigmoid activations for the binary classification ones. In both cases the cross entropy function is used as cost function and L2 regularization is added as a penalty term.

Given a layer fully connected layer in any network, biases have been initialized to 0 while weights are randomly initialized using a Gaussian distribution centered at 0 and with standard deviation σ_{layers}:

$$\sigma_{layers} = \frac{1}{\sqrt{\frac{2}{n_l}}}$$

Where n_l is the number of neurons in the layer.

All structured input features are numeric and have been normalized into mean 0 and standard deviation 1, while image data has been normalized to be around 0. No further preprocessing has been performed.

6.3.1 Hyperparameter tuning

Here we present a list of the hyperparameters that we are using throughout all the experiments:

- Initial learning rate lr_0.

- Learning rate decay factor $decay_{lr}$.
- Learning rate decay epochs $epochs_{lr}$.
- Batch size.
- L2 ratio.
- Number of kernel units (only applied when testing hybrid architectures).
- Standard deviation of the RFF sampling (only applied when testing hybrid architectures).
- Number of units for the hidden layers.

Note that we are using the same number of units in all kernel layers and in all hidden layers. This is done to simplify the tuning process as we do not want to get the best performance for the datasets selected but to see how different approaches compare under the the same conditions.

These hyperparameters are tuned using Random Search. Search spaces for each dataset have been defined in an early manual tuning stage and then kept fixed for the dataset for all experiments. The nature of each parameter has determined its sampling distribution. For instance, we can sample the learning rate decay factor $decay_{lr}$ in a continuous domain as:

$$lr_{decay} \sim U(0.1, 1.0) \tag{6.1}$$

For other parameters, such as the initial learning rate lr_0, it is more suited to sample from a uniform distribution on a log-scale:

$$log\_lr_0 \sim U(-4, -1) \tag{6.2}$$

$$lr_0 \sim 10^{log\_lr_0} \tag{6.3}$$

6.4 Reproducing baseline experiments

The architectures we have presented in this work are an extension of the work in [3]. Therefore, for a single layer we expect to get similar results under the same conditions.

It is useful for us to replicate the experiments for two main reasons: it can provide us with a baseline for other experiments and it can also prove the soundness of our implementation.

Dataset	Instances	Features	Classes	[3] test error	Our test error	Train time (s)
Australian	690	11	2	0.15 ± 0.01	0.11 ± 0.02	5.95 ± 0
Sonar[1]	208	60	2	0.25 ± 0.04	0.17 ± 0.06	6.71 ± 0
Titanic	2201	3	2	0.22 ± 0.01	0.22 ± 0.02	8.00 ± 0
Monk2	432	6	2	0.00 ± 0.00	0.02 ± 0.01	1.28 ± 0
Balance	625	4	3	0.04 ± 0.01	0.04 ± 0.01	3.15 ± 0
Magic	19020	10	3	0.14 ± 0.04	0.14 ± 0.00	256.90 ± 1
Covertype	581012	54	7	0.15 ± 0.03	0.065 ± 0.00	7826.54 ± 530
SUSY	$5M$	18	2	0.20 ± 0.02	0.1974 ± 0.00	3985 ± 76

TABLE 6.3: Comparison between original paper [3] results and our reproduced results as the average of 10 simulations.

We have had several challenges to replicate the results, mainly due to omitted information in the original paper (e.g. gradient optimization algorithm is not provided. No list or domain of hyperparameters provided). We have also had to restrict some of the experiments due to our hardware limitations.

As in the original work, we randomly split the datasets into training and tests sets using 80% of the data for training. We have also used a 10-fold cross validation procedure for the tuning of most of the models. For Covertype, Magic and SUSY datasets we are not using cross validation due to the their size and we have chosen to use a random subset of the training data as validation data for each trial.

In the case of the Sonar dataset we only have 208 instances, leaving only around 16 instances for validation using 10 folds, which is hardly representative of the test data for the generalization error estimation. In order to fix that, we have used 5-fold cross-validation on that specific dataset.

Table 6.3 shows the results we got reproducing the original experiments. These numbers are the outcome of the best model found on 50 random trials on the defined sampling spaces of the hyperparameters (only 5 trials for the Covertype and SUSY datasets due to their size). The parametrization found has been then independently trained on the training set and evaluated on the test set 10 times. We can see how results are, on average, as good as the ones from the paper. It is noticeable that we have achieved lower errors than the original ones in some cases, probably due to a more intensive hyperparameter search from our side.

[0]Used only 5 folds due to its limited size so we have more instances in each fold and validation set is more representative

6.5 Experiment 1: Training procedures

This first experiment is an intent to evaluate which is the most suited strategy to train our hybrid multilayer neural networks among the considered methods. We compare the performance of each training methodology against traditional multilayer neural network training using 1 up to 4 layers.

Given a dataset, we use the same hyperparameter distributions for all the experiments. We additionally introduce two new hyperparameters to the set that has been defined in Section 6.4:

- We add a layer switching policy hyperparameter that uniformly samples from the 3 available policies defined in Section 3.2.2 for the CLT and ALT training procedure.
- We add a new parameter that controls the number of epochs that we must train each layer in the ALT training method:

$$epochs\_per\_layer \sim U(epochs_{min}, epochs_{max}) \tag{6.4}$$

  Where $epochs_{min}$ and $epochs_{max}$ are problem dependent.

The goal of this experiment is to evaluate each iterative training procedure independently for layers from 1 to 4 and comparing against 2 baselines: the traditional neural network training using a common feed forward network and a hybrid network.

For each dataset we have performed 25 random trials for tuning, from which we have picked the best configuration.

6.6 Experiment 2: Hybrid Convolutional Neural Networks

One of the key features of the success of deep learning is its outstanding results in non-structured data where handcrafted features and expertise domain were traditionally required in order to achieve good performance. To be able to acknowledge how our hybrid strategies perform with heterogeneous kind of data and how our proposed training procedures behave with image data, we use FASHION-MNIST data.

Among the best training procedures spotted in the previous experiment we select the most promising ones for this experiment. Then, we compare HCNN performances on images against regular Convolutional Neural Networks. By default, we place two fully connected layers or kernel blocks at the end of the stacked convolutional blocks.

Convolutional network architectures generally involve more computations than normal feedforward neural networks (though they contain less parameters). Therefore, we reduce the number of random trials of these experiment to 5 in order to take more profit of our computing resources. Because of the nature of these architectures, we need to additionally add some more hyperparameters that need to be tuned:

- Image filter side size for the squared convolutional filters.
- Number of filters per convolutional layer.
- Number of Random Fourier Features per convolutional layer. We have decided to use a separate value for the RFF used in convolutional kernel blocksand the ones in regular kernel blocks (only needed in hybrid settings).
- Stride of the convolution.

Note that he size of the output at the last convolutional block depends on the size of the convolutions and the stride used. Therefore, the search space for some parameters has been adapted depending on the particular experiment in order to avoid having huge volumes in the middle of the network but also avoid negative sizes. Batch norm has not been considered in this experiment.

6.7 Experiment 3: Regularization

The last experiment tests our presented regularization strategies in Section 4.2. The dataset chosen has been FASHION-MNIST and we will use the best hybrid setting from the previous experiment as the baseline. We will compare the novel RFFS regularization technique with traditional dropout for neural networks. We perform resampling for RFFS at the end of each epoch.

Chapter 7

Results

7.1 Experiment 1: training procedures results

This experiment is performed in two datasets: Motor and Magic. Due to the fact that Magic dataset has been easy to solve for all the settings due to its simplicity, it does not give us important intuitions in this experiments and we have moved its results into Appendix B. The results for the second experiment are more significative and are depicted in Table 7.1. Motor dataset is more complex than the previous one (i.e. it has 11 different classes and more instances) and we see how increased complexity leads, on average, into lower errors. Hybrid architectures generally achieve better results.

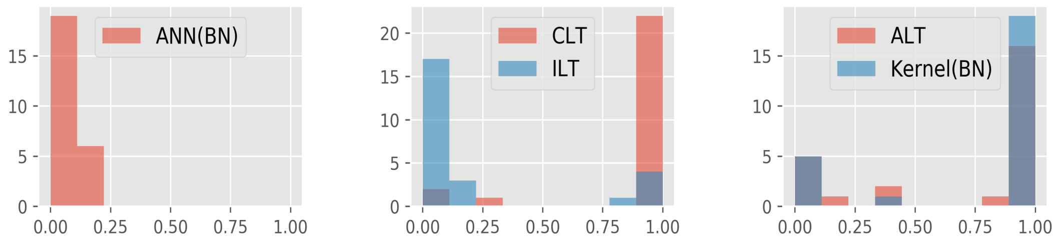


FIGURE 7.1: Histogram of the validation error results for the Motor dataset using 4 layers for each of the training procedures tested.

It is clear that deeper architectures easily suffer from the shift invariant problem described in Section 3.5 as we observe that the error increases as the depth of the network does when batch norm is not used. It may also be possible that these deeper architectures require a very small learning rate in order to be trained properly, as batch norm is also known for having a regularization side-effect which enables to use bigger learning rates with stability. Layerwise procedures, on the other hand, seem to overcome this shift invariant effect and seem to be more robust in deep layouts.

We observe that Cyclic Layerwise Training (CLT) method, because of its nature, is the slowest from the hybrid training procedures tested while ILT and ALT get highly competitive results (or even the best) at a lower computational cost.

Hybrid	Layers	Procedure	Batch norm	Test error	Epochs	Time (s)
No	**1**	**Regular**	**No**	**0.1290 ± 0.0055**	**545**	**821 ± 72**
Yes	1	Regular	No	0.1292 ± 0.0032	510	845 ± 171
No	2	Regular	No	0.1447 ± 0.0058	70	91 ± 3
No	2	Regular	Yes	0.1079 ± 0.0035	285	466 ± 10
Yes	2	Regular	No	0.1392 ± 0.0112	95	139 ± 10
Yes	**2**	**Regular**	**Yes**	**0.0909 ± 0.0020**	**230**	**533 ± 12**
Yes	2(2)	ILT	No	0.0914 ± 0.0021	70	108 ± 7
Yes	2	CLT	No	0.0941 ± 0.0028	980	1495 ± 60
Yes	2	ALT	No	0.0946 ± 0.0035	165	264 ± 14
No	3	Regular	No	0.2827 ± 0.0232	10	14 ± 2
No	3	Regular	Yes	0.0926 ± 0.0028	115	169 ± 5
Yes	3	Regular	No	0.3415 ± 0.0355	5	8 ± 1
Yes	3	Regular	Yes	0.0973 ± 0.0036	325	597 ± 5
Yes	3(3)	ILT	No	0.0957 ± 0.0026	725	1021 ± 34
Yes	3	CLT	No	0.0891 ± 0.0024	1770	2915 ± 204
Yes	**3**	**ALT**	**No**	**0.0885 ± 0.0023**	**600**	**1005 ± 23**
No	4	Regular	No	0.5548 ± 0.6884	5	10 ± 1
No	4	Regular	Yes	0.0981 ± 0.0113	465	2651 ± 3
Yes	4	Regular	No	0.1060 ± 0.0129	305	537 ± 8
Yes	**4**	**Regular**	**Yes**	**0.0896 ± 0.0019**	**1170**	**2242 ± 71**
Yes	4(3)	ILT	No	0.0929 ± 0.0030	2185	3364 ± 57
Yes	4	CLT	No	0.0937 ± 0.0019	3396	12331 ± 21
Yes	4	ALT	No	0.0907 ± 0.0026	340	547 ± 15

Table 7.1: Test error, epochs trained and training time for the different training procedures in Motor dataset. For ILT training, the number between brackets are the number of layers that got selected during tuning and the other amount is the maximum number of layers.

It is important to point out that kernel architectures have proven to be more complex to tune and more sensitive to hyperparameter tuning than traditional neural network. More precisely, we have observed a huge sensitivity of the scale parameter of the kernel, resulting in very poor results when it is not properly tuned. This is properly depicted in Figure 7.1. Given a set of iid trials from a fixed search space, traditional architectures show a much better error rate on average, even though they cannot always provide the best result. Nevertheless, we can observe that the incremental training strategy (i.e. ILT) is far more robust than the other presented strategies for hybrid networks. This leads to the need of finding automatic strategies for finding the best parametrization for the scale, which would not only reduce the number of hyperparameters but also improve the average quality of the solutions found during tuning.

Regarding the tuned layer switching policies from methods CLT and ALT, it is remarkable that most of the time the Cyclic Policy is the one selected by the tuning process, even though Random Policy has also been selected in some specific trials.

It also important to consider that time column from Table 7.1 has to be taken with a pinch of salt because of early stopping. There are trials that can be stuck alternatively decreasing and decreasing the validation error for a while before stopping, even if they are computationally not very expensive. On the other side, we can find deeper architectures which are more expensive to converge early and stop. Consequently, it is not a representative measure of the capacity of the model neither its efficiency, as several random trials with the same setting may lead to quite different stop epochs.

Appendix C shows the evolution of the tested training procedures. We observe how, in this case, layerwise procedures are quite inefficient since they take too much time before switching to another layer, what is what makes the error decrease significantly. For CLT and ALT we see that after one or two training cycle has been completed (i.e. training process has iterated once from layer 1 to layer 4) the network starts converging. This suggests that more relaxed criterias for switching layer may improve performance and efficiency. This also leads us to think that using fine-tuning on the layerwise procedures using joint training of all layers may also be interesting for further study.

7.2 Experiment 2: Convolutional Hybrid Networks

The results for the second experiment can be found in Table 7.2. We can observe that kernel methodologies have performed better than the CNN baseline with a higher overfitting but also with stronger generalization power. We have again observed, however, that the quality of the results is much better in the CNN case though we have observed an improvement in the average quality of the performance of hybrid networks when using image data. The results also confirm that the proposed layerwise strategies can be competitive for convolutional strategies.

Layers	Hybrid	Procedure	Train error	Test error	Epochs	Time (s)
2	No	Regular	0.0396 ± 0.0010	0.1043 ± 0.0016	255	3294 ± 128
2	**Yes**	**Regular**	$\mathbf{0.0007 \pm 0.0002}$	$\mathbf{0.0824 \pm 0.0031}$	**95**	$\mathbf{1960 \pm 25}$
2(1)	Yes	ILT	0.0024 ± 0.0003	0.0956 ± 0.0021	130	4064 ± 8
2	Yes	ALT	0.0013 ± 0.0003	0.0969 ± 0.0044	54	820 ± 4

Table 7.2: Results for Fashion Mnist averaged over 5 simulations of the best configuration found out of 5 random search trials. The number of layers in the table refer to the number of convolutional layers (or blocks) that are stacked in the network before the 2 fully connected and the output layer which are present in all the settings. For ILT, the number between the parenthesis is the number of layers that actually got trained as out of the total number of possible layers.

Additional information is shown in Figure 7.2. We see that we that all the settings start converging around epoch 40. We see how hybrid networks keep fitting the training data while the CNN follows a slower pace. It is also noticeable that layerwise strategies stabilize have higher loss values in the early stages of training.

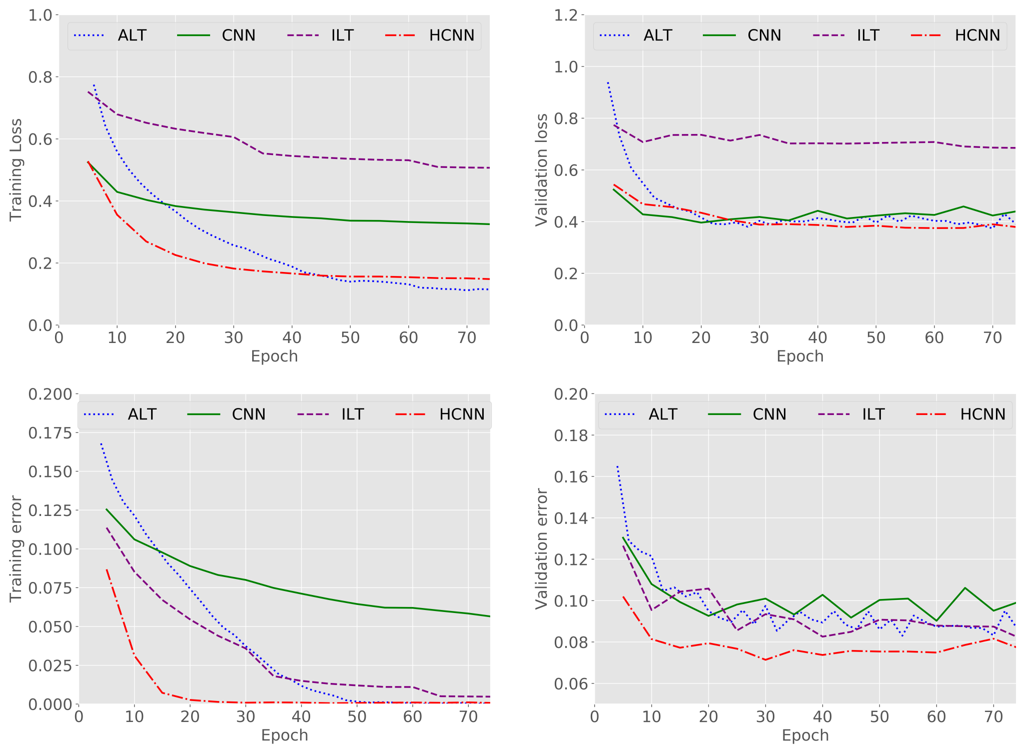


Figure 7.2: Evolution of the error and loss (includes L2 penalty term) for each of the convolutional architectures tested for both training and validation.

7.3 Experiment 3: Regularization

In this experiment we compared novel regularization techniques for hybrid networks with common regularization techniques such as dropout and L2 regularization. We have used the best performing hybrid model from previous experiment (i.e. the one without layerwise training) and eliminated the L2 regularization from it. Then, we have added dropout and Random Fourier Features resampling (RFFS) using different parametrization of these methods. We have independently applied dropout to the convolutional layers and to all the network (except from the output layer).

Results are shown in Figure 7.3. We can see that traditional dropout reduces overfitting and that the less conversative a parametrization is (i.e. the ones that drop more units and with higher probability) have worse generalization at the beginning but converges at approximately the same error than the others in later epochs.

On the other hand, we see that our proposed method regularizes too much the model and makes it lose generalization capabilities. As expected, the strategies that add more noise to the training process are the ones that suffer the biggest worsening. As a future task, we propose using RFF resampling on a subset of the layers to see if we get a less *agressive* scheme. Additionally, the alternative regularization strategy proposed in Section 4.2.2 can also be a promising solution.

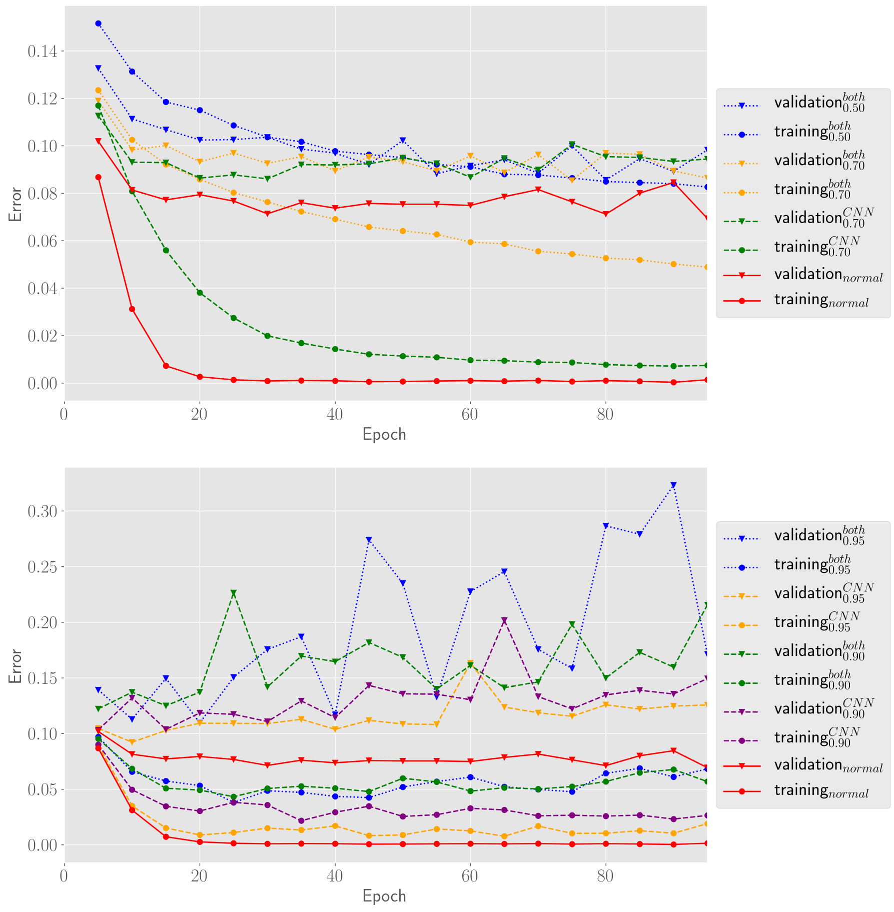


FIGURE 7.3: Evolution of the training and validation error on several parametrization of dropout and the baseline experiment with L2 regularization. On the bottom, we show the same baseline with RFFS regularization using resampling at the end of each epoch. Number displayed below the experiment label are the probability to keep a node for dropout and the probability to keep a RFF feature in RFFS. The text above the label indicate whether we applied the technique on the convolutional part of the network (CNN) or on both the convolutional part and the fully connected part (both).

Chapter 8

Conclusions and future work

8.1 Conclusions

In this work we have built hybrid neural networks that use approximated kernel functions through Random Fourier Features in a multilayer fashion and provided an efficient implementation. We have seen that they have good results against traditional neural networks and that they can be easily trained using the same techniques. The main weakness of these models are the high sensitivity to the kernel parametrization, providing a poor overall quality of the solutions compared to other methods. However, we have empirically find that it can offer better solutions on average than regular neural networks. This could suggest that the methods used in this work are a good choice for problems where even small accuracy improvements are crucial (and worth spending the extra computational resources needed in this case).

We have proposed layerwise methods to train hybrid networks, which show competitive performance with our baselines in all depths tested. These strategies have proven to be robust in deep layouts without needing to resort to batch normalization, such as in the jointly trained procedure. We observed that switching between layers in small intervals is preferred rather than intensively train each layer separately. From the 3 strategies detailed, ALT and ILT have been the most promising, while CLT has been too computationally intensive. We have tested our implementation in structured data and in image data, obtaining satisfying results in both of them. Though we have still observed an important dependency between the model throughput and the kernel parametrization, it has been perceived as slightly lower when dealing with images.

Finally, we have proposed 2 novel regularization techniques: one based on resampling of the Random Fourier Features and another one based on introducing noise to these features. We have compared the performance of the first one against dropout concluding that the regularization effect introduced by the method is to high, decreasing the overall quality of the model. However, we believe that noise injection could help prevent overfitting in this kind of networks.

8.2 Future work

There are several ideas we point to be studied in the future. First, it would be valuable to evaluate the quality of the features learnt in hybrid neural networks. One way we could do this would be to train big datasets (i.e. ImageNet) on these networks and compare the quality of the features learnt with respect to other methods through transfer learning.

Due to our limited hardware equipment, we could not afford to use big networks or datasets if we wanted to get a fair pool of experiments. Future work could be based in proving the stability of these methods in very large neural networks (e.g. ResidualNets).

Though we have restricted our kernels to be Gaussian, we could use any valid kernel function if we sample from its corresponding Fourier Transform. We encourage comparing the performance of different kernels in a variety of problems. Similarly, we would like to see if we can extend these networks to work on other domains such as Natural Language Processing or regression problems.

We see that there is room for improvement also in the introduced layerwise training procedures. Fine-tuning strategies could help improve the results of these methods, as well as forcing smaller switching times between layers.

Probably the most interesting feature to study next would be how to automatically get proper values (or ranges) for the scale parameter of the kernel features, as we have seen that the outcome of the training process is very sensitive to its changes. We could also research on whether we can use different scale values per layer or layer type to get improvements.

Appendix A

Tensorflow as numerical optimization tool: perceptron example

In this appendix we present a simple optimization example in order to understand how to build and execute simple computational graph and have some insight of it using visualization.

We are going to implement a perceptron using Tensorflow. We use a binary classification problem sampling two random blobs using 5 dimensions and 10000 points. The output of the perceptron is determined by:

$$y' = \sum_{i}^{N} w_i x_i + b$$

And the class of the output is computed as follows:

$$y_{pred}(y') = \begin{cases} 1 & \text{if } y' \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

The binary classification problem can be regarded as the minimization of the cross entropy loss $\mathcal{L}$:

$$\mathcal{L}(y, y') = -\frac{1}{N} \sum_{i} y_i \ \log y'_i + (1 - y_i) \log(1 - y'_i))$$

The code to implement this problem using tensorflow for Python can be implemented in a few lines. First, we define the Python imports and the input data:

```
import tensorflow as tf
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt

n, nc, hiddens = 5000, 5, 10
x_train, y_train = make_blobs(
    n_samples=n, n_features=nc, centers=2, random_state=100
)
```

Then we can design the computation graph as:

```
with tf.Graph().as_default() as graph:

    tf.set_random_seed(10)

    # Placeholders for data and groundtruth
    x = tf.placeholder(dtype=tf.float32, shape=[n, nc])
    y = tf.placeholder(dtype=tf.float32, shape=[n])

    # Define perceptron variables
    w = tf.Variable(tf.random_normal([nc, hiddens], stddev=0.01), name='W')
    b = tf.Variable(tf.zeros([hiddens]), name='b')

    # Define perceptron output
    z = tf.reduce_sum(tf.add(tf.matmul(x, w), b), axis=-1)
    output = tf.nn.sigmoid(z)

    # Define function to optimize
    loss_op = tf.reduce_mean(
        tf.nn.sigmoid_cross_entropy_with_logits(labels=y, logits=z)
    )

    # Define optimization process and metrics
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.0001)
    train_op = optimizer.minimize(loss_op)
    predicted = tf.cast(tf.greater(output, 0.5), tf.float32)
    accuracy_op = tf.reduce_mean(tf.cast(tf.equal(y, predicted), tf.float32))
```

At this point we have not performed any computation but just build the nodes and edges in our graph. Additionally we can store this graph for further visualization and also keep track of the metrics of the optimization process (i.e. loss value and accuracy):

```
with tf.Graph().as_default() as graph:

    [...]

    # Gather summaries for visualizations and store graph
    writer = tf.summary.FileWriter('output', graph)
    tf.summary.scalar('accuracy', accuracy_op)
    tf.summary.scalar('loss', loss_op)
    summary_op = tf.summary.merge_all()
```

Finally we can just run iterations feeding the data into a Tensorflow Session:

```
with tf.Graph().as_default() as graph:

    [...]

    with tf.Session() as sess:

        sess.run(tf.global_variables_initializer())

        for i in range(10):

            # Run operations
            _, loss_value, acc_value, predicted_value, summary = sess.run(
                [train_op, loss_op, accuracy_op, predicted, summary_op],
                feed_dict={x: x_train, y: y_train}
            )

            # Store summary for visualization
            writer.add_summary(summary, i)

            print('\%d. Loss: \%f, Accuracy: \%f' \% (i, loss_value, acc_value))
```

Note that prior to run any computation it is essential to initialize all the variables in the graph. Otherwise the program will raise an error. The *train* operation is the one that computes and propagates the gradients through the graph, while the *loss*, *prediction* and the *accuracy* operations are run merely for monitoring purposes. The output of the summary operation is stored so the visualization tool keeps track of the value at the i^{th} iteration.

The output of the program, as shown in the plots in fig. A.1 show how the training operation has updated the values of the weights and the bias properly:

```
0. Loss: 0.643855, Accuracy: 0.500200
1. Loss: 0.623610, Accuracy: 0.504800
2. Loss: 0.604279, Accuracy: 0.524200
3. Loss: 0.585825, Accuracy: 0.572800
4. Loss: 0.568206, Accuracy: 0.664800
5. Loss: 0.551385, Accuracy: 0.780400
6. Loss: 0.535323, Accuracy: 0.880200
7. Loss: 0.519983, Accuracy: 0.942600
8. Loss: 0.505330, Accuracy: 0.978200
9. Loss: 0.491330, Accuracy: 0.993000
```

After we run this example we can monitor the optimization process using Tensorboard:

```
> tensorboard --logdir=output
```

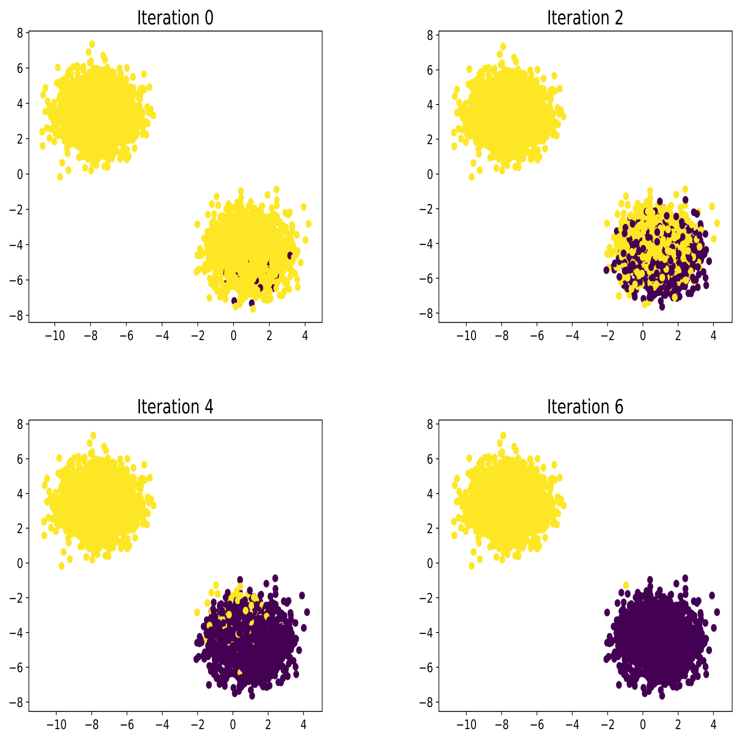


FIGURE A.1: Evolution of the predicted labels for the proposed toy classification problem using a perceptron in Tensorflow.

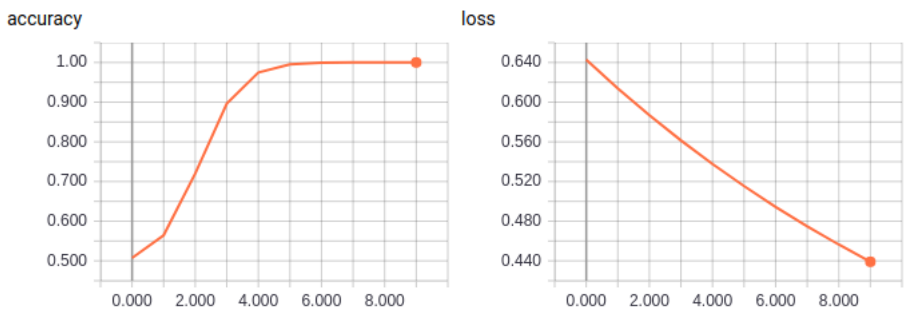


FIGURE A.2: Loss (left) and accuracy(right) evolution through 10 iterations for the perceptron example from Tensorboard visualization tool.

This will create a local server that can be accessed locally using port 6006. Tensorboard can provide the value of the loss and the accuracy that we tracked in the code above, as we can see in fig. A.2 or display the computational graph built (fig. A.3).

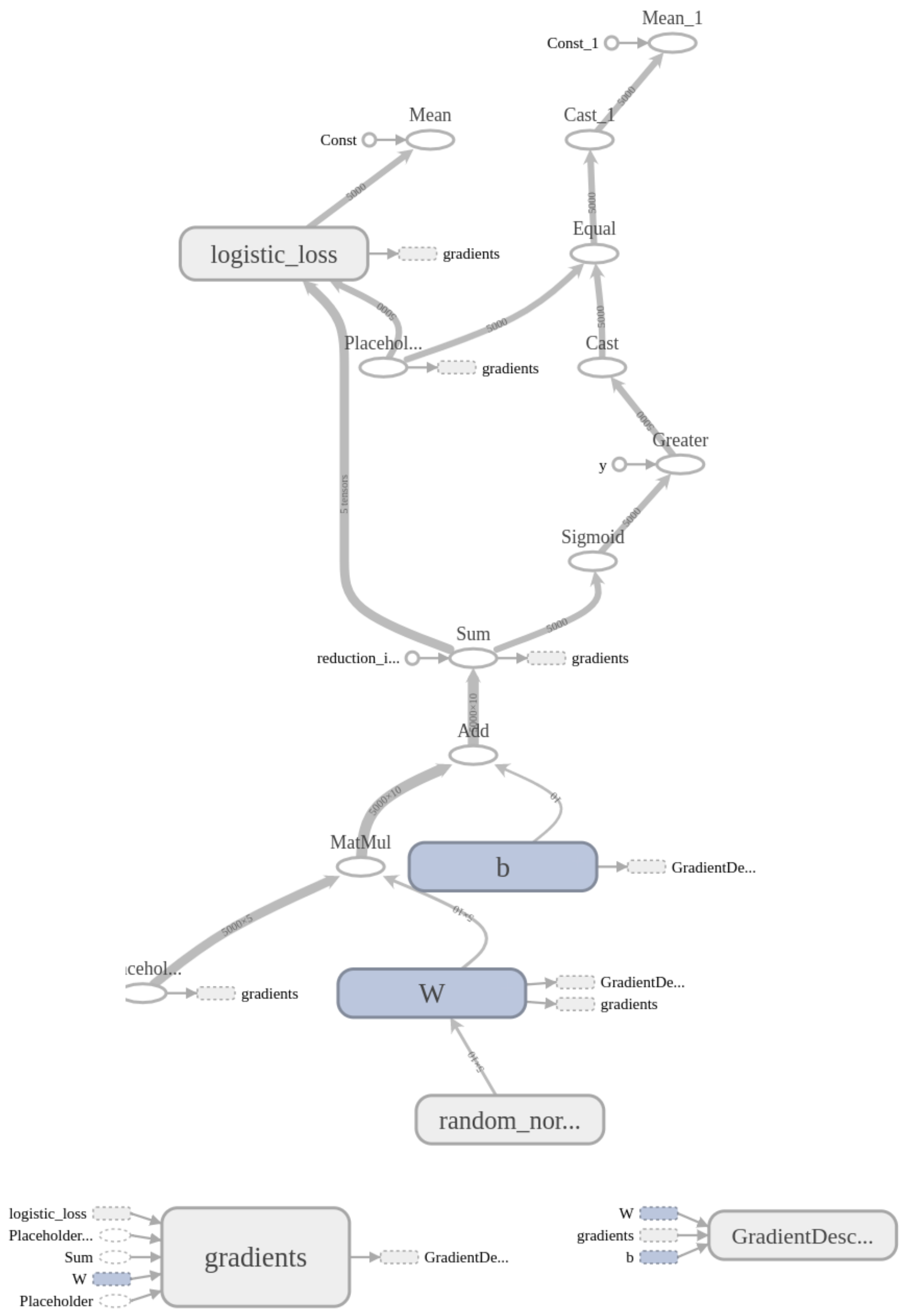


Figure A.3: Visual representation of the computational graph for a binary classification problem in a perceptron.

Appendix B

Magic dataset results

Hybrid	Layers	Procedure	Batch norm	Test error	Epochs	Time (s)
No	**1**	**Regular**	**No**	**0.1359 ± 0.0052**	**35**	**16 ± 1**
Yes	1	Regular	No	0.1393 ± 0.0063	280	115 ± 4
No	2	Regular	No	0.1313 ± 0.0053	155	99 ± 2
No	2	Regular	Yes	0.1373 ± 0.0187	40	18 ± 0
Yes	2	Regular	No	0.1330 ± 0.0060	110	56 ± 5
Yes	2	Regular	Yes	0.1440 ± 0.0017	55	26 ± 0
Yes	2(2)	ILT	No	0.1405 ± 0.0074	475	192 ± 2
Yes	**2**	**CLT**	**No**	**0.1293 ± 0.0038**	**670**	**327 ± 20**
Yes	2	ALT	No	0.1435 ± 0.0062	180	76 ± 0
No	**3**	**Regular**	**No**	**0.1239 ± 0.0047**	**235**	**214 ± 5**
No	3	Regular	Yes	0.1431 ± 0.01194	420	535 ± 0
Yes	3	Regular	No	0.1410 ± 0.0076	90	50 ± 4
Yes	3	Regular	Yes	0.1339 ± 0.0057	35	17 ± 0
Yes	3(1)	ILT	No	0.1348 ± 0.0075	175	68 ± 0
Yes	3	CLT	No	0.1312 ± 0.0059	1045	463 ± 10
Yes	3	ALT	No	0.1417 ± 0.0037	500	204 ± 2
No	**4**	**Regular**	**No**	**0.1233 ± 0.0051**	**105**	**49 ± 2**
No	4	Regular	Yes	0.1276 ± 0.0064	190	134 ± 1
Yes	4	Regular	No	0.1607 ± 0.0109	175	80 ± 3
Yes	4	Regular	Yes	0.1440 ± 0.0072	70	45 ± 5
Yes	4	ILT	No	0.1413 ± 0.0075	70	30 ± 1
Yes	4	CLT	No	0.1336 ± 0.0045	1355	615 ± 29
Yes	4(1)	ALT	No	0.1441 ± 0.0057	25	13 ± 0

TABLE B.1: Test error, epochs trained and training time for the different training procedures using Magic dataset. For ILT training, the number between brackets are the number of layers that got selected during tuning and the other amount is the maximum number of layers. We can observe that hybrid methods cannot beat traditional neural networks for this dataset. While the latter show a decreasing error trend as the number of layers increase we can observe an increasing trend for hybrid architectures instead. However, when the number of layers is higher, the proposed training procedures seem to slightly perform better than the regular training method. Nevertheless, we see that errors tend to oscillate within a certain interval, which can also be due to the random search tuning.

Appendix C

Hybrid network training procedures for Motor dataset

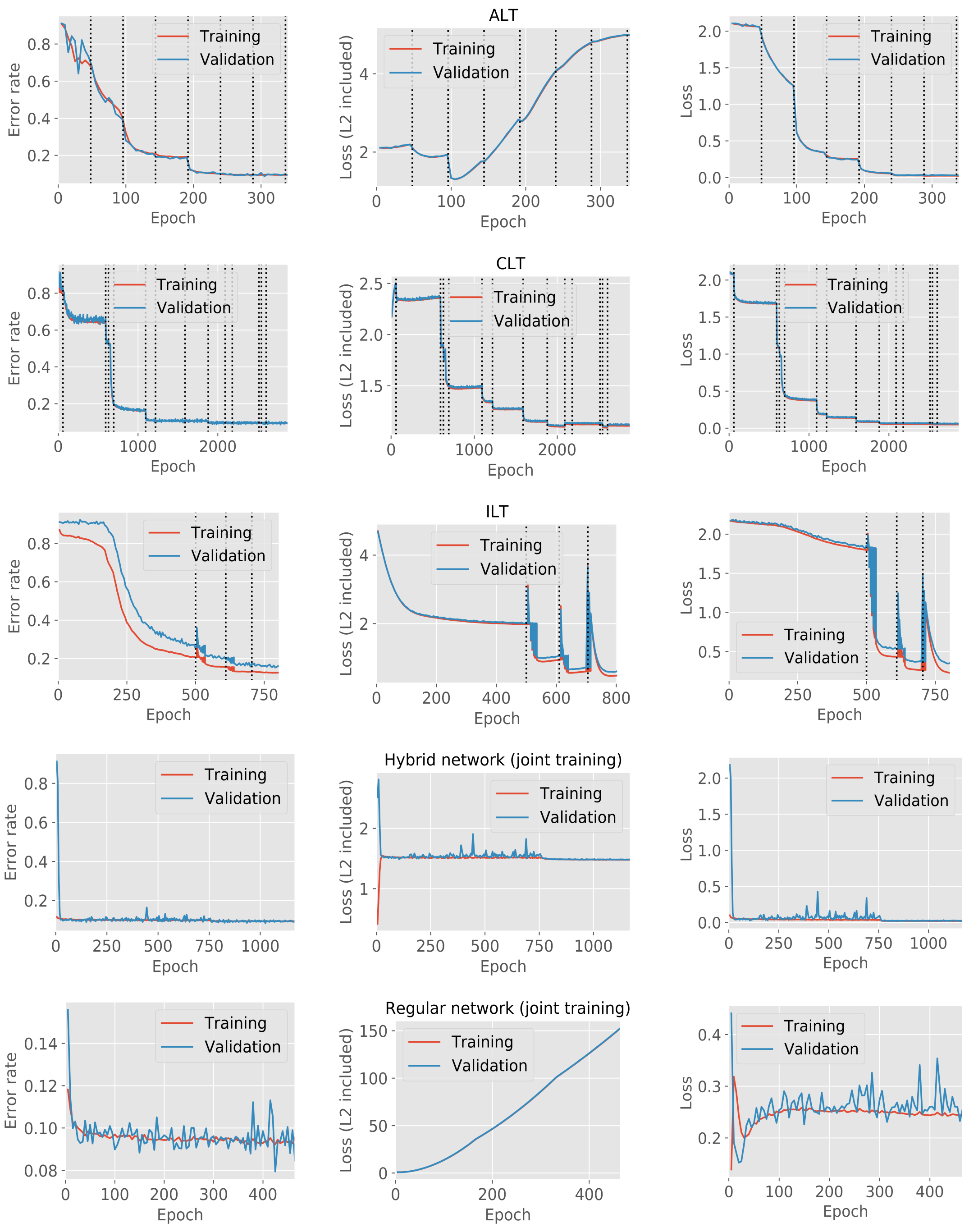


FIGURE C.1: Training and validation error (left), loss (with L2 term) and loss without L2 term for the best configuration found for the Motor dataset for several training strategies. Epochs where the layer to train has been switched are indicated with vertical lines. ALT and CLT used a cyclic switching policy, starting from the first layer.

Bibliography

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016. ISBN 0262035618, 9780262035613.

[2] Ll. Belanche and M. Ruiz. Bridging deep and kernel methods. In *European Symposium on Artificial Neural Networks*, pages 1–10, Apr 2017.

[3] Siamak Mehrkanoon, Andreas Zell, and Johan AK Suykens. Scalable hybrid deep neural kernel networks. In *Proc. of the European Symposium on Artificial Neural Networks*, number accepted, 2017.

[4] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, pages 1177–1184, USA, 2007. Curran Associates Inc. ISBN 978-1-60560-352-0. URL http://dl.acm.org/citation.cfm?id=2981562.2981710.

[5] David Barber. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, New York, NY, USA, 2012. ISBN 0521518148, 9780521518147.

[6] Andrew Gordon Wilson, Zhiting Hu, Ruslan Salakhutdinov, and Eric P. Xing. Deep kernel learning. *CoRR*, abs/1511.02222, 2015. URL http://arxiv.org/abs/1511.02222.

[7] Gaurav Pandey and Ambedkar Dukkipati. To go deep or wide in learning? *CoRR*, abs/1402.5634, 2014. URL http://arxiv.org/abs/1402.5634.

[8] Julien Mairal, Piotr Koniusz, Zaïd Harchaoui, and Cordelia Schmid. Convolutional kernel networks. *CoRR*, abs/1406.3332, 2014. URL http://arxiv.org/abs/1406.3332.

[9] Shuai Zhang, Jianxin Li, Pengtao Xie, Yingchun Zhang, Minglai Shao, Haoyi Zhou, and Mengyi Yan. Stacked Kernel Network. (1), 2017. URL http://arxiv.org/abs/1711.09219.

[10] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387310738.

[11] Thomas Hofmann, Bernhard Schlkopf, and Alexander J. Smola. Kernel methods in machine learning. *Ann. Statist.*, 36(3):1171–1220, 06 2008. doi: 10.1214/009053607000000677. URL https://doi.org/10.1214/009053607000000677.

[12] Yaser Abu-Mostafa. Kernel methods (machine learning course, caltech), 2018. URL https://www.youtube.com/watch?v=XUj5JbQihlU&list=PLCA2C1469EA777F9A&index=15. Accessed 28/02/18.

[13] Tianbao Yang, Yu-feng Li, Mehrdad Mahdavi, Rong Jin, and Zhi-Hua Zhou. Nyström Method vs Random Fourier Features: A Theoretical and Empirical Comparison. In F Pereira, C J C Burges, L Bottou, and K Q Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 476–484. Curran Associates, Inc., 2012.

[14] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806, 2014. URL http://arxiv.org/abs/1412.6806.

[15] Ng Andrew. Bensouda Mourri Younes. Katanforoosh Kian. Backpropagation intuition, 2018. URL https://www.coursera.org/learn/neural-networks-deep-learning/lecture/6dDj7/backpropagation-intuition-optional. Accessed 25/03/18.

[16] Ng Andrew. Bensouda Mourri Younes. Katanforoosh Kian. Orthogonalization, 2018. URL https://www.coursera.org/learn/machine-learning-projects/lecture/FRvQe/orthogonalization. Accessed 21/02/18.

[17] Lutz Prechelt. Early stopping - but when? In *Neural Networks: Tricks of the Trade, volume 1524 of LNCS, chapter 2*, pages 55–69. Springer-Verlag, 1997.

[18] Guang-Bin Huang and Lei Chen. Enhanced random search based incremental extreme learning machine. *Neurocomputing*, 71(16-18):3460–3468, 2008.

[19] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Netw.*, 4(2):251–257, March 1991. ISSN 0893-6080. doi: 10.1016/0893-6080(91)90009-T. URL http://dx.doi.org/10.1016/0893-6080(91)90009-T.

[20] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, February 2012. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=2188385.2188395.

[21] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL http://arxiv.org/abs/1502.03167.

[22] M. Wilber S. Gross. Training and investigating residual nets, 2016. URL http://torch.ch/blog/2016/02/04/resnets.html.

[23] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

[24] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[25] M. Lichman. UCI machine learning repository, 2013. URL http://archive.ics.uci.edu/ml.

[26] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017. URL http://arxiv.org/abs/1708.07747.

[27] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980.