Universidad Autónoma de Madrid
Escuela Politécnica Superior


Double Major in Mathematics and Computer Science

Final Degree Project

# HERRAMIENTAS MODERNAS EN REDES NEURONALES: LA LIBRERÍA TENSORFLOW DE GOOGLE

Author: Antonio Mejías Gil
Tutor: José Ramón Dorronsoro Ibero

May 2017

## Abstract

This project explores deep artificial neural networks and their use with *Google*'s open-source library TensorFlow. We begin by laying the theoretical foundations of these networks, covering their motivation, techniques used and some mathematical aspects of their training. Special attention is paid to various regularisation methods which are applied later on.

After that, we delve into the computational approach, explaining TensorFlow's operation principles and the necessary concepts for its use, namely the computational graph, variables and execution sessions. Through the first example of a deep network, we illustrate the theoretical and TensorFlow-related elements described earlier, applying them to the problem of classifying flowers of the Iris species.

We then pave the way for the problem of image classification: we comment several higher-level TensorFlow wrappers (focusing on *Slim*, a library born within *Google* itself which is used in the last part of the project), describe the basic principles of convolutional networks and introduce the MNIST problem (automatic handwritten digit recognition), outlining its history and current state of the art.

Finally, we create three convolutional networks to tackle MNIST, detailing how such a task is approached with TensorFlow and the workflow followed. All three networks reach over 98% classification accuracy, going as far as 99.52% in the case of the best one. We conclude with an explanation of the obtained results, relating the structures of the different networks with their performance and training cost.

## Keywords

Artificial neural network, convnet, deep learning, backpropagation, dropout, regularisation, TensorFlow, op, session, Slim, MNIST, handwritten digit recognition.

## Resumen

Este proyecto explora las redes neuronales artificiales profundas y su uso con la biblioteca de código abierto TensorFlow de *Google*. Comenzamos exponiendo los fundamentos teóricos de este tipo de redes, su motivación, técnicas usadas y algunos aspectos matemáticos de su entrenamiento. Prestamos especial atención a diferentes métodos de regularización que serán aplicados posteriormente.

A continuación nos centramos en el enfoque computacional, explicando los principios bajo los que opera TensorFlow y los conceptos necesarios para su uso, especialmente el grafo de cálculo, las variables y las sesiones de ejecución. Mediante un primer ejemplo de red profunda, ilustramos los elementos teóricos y de TensorFlow descritos anteriormente, aplicándolos al problema de clasificación de flores de la especie Iris.

Seguidamente preparamos el terreno para abordar el problema de clasificación de imágenes: comentamos diferentes *wrappers* de TensorFlow de más alto nivel (poniendo el foco en *Slim*, librerá nacida dentro del propio *Google* que usamos en la última parte del trabajo), describimos los principios básicos de las redes convolucionales e introducimos el problema MNIST (reconocimiento automatizado de dígitos escritos a mano), recorriendo su historia y el estado del arte actual.

Finalmente, implementamos tres redes convolucionales para abordar MNIST, detallando cómo se enfoca una tarea así con TensorFlow y el flujo de trabajo seguido. Las tres redes alcanzan más de un 98% de precisón en clasificación, llegando al 99.52% en el caso de la mejor. Concluimos con una explicación de los resultados obtenidos, relacionando las estructuras de las diferentes redes con su rendimiento y coste de entrenamiento.

## Palabras clave

Red neuronal artificial, *convnet*, aprendizaje profundo, retropropagación, *dropout*, TensorFlow, *op*, sesión, Slim, MNIST, reconocimiento de dígitos manuscritos.

# Acknowledgements

This project would not have been possible without the help and inspiration provided by many people:
First and foremost, my tutor José Ramón Dorronsoro. His patience, thorough and frequent feedback and guidance when the path was not clear have been essential. I am immensely grateful to him for the generous opportunity of working in the GAA (Machine Learning Group at UAM) while developing my project.
To my partner Sylwia, for enduring my computer science talk while encouraging me to push through and caring to revise my (discouraginly long) English writing. To my parents and brother, for providing me with the best possible environment to pursue my degree. To my friends, for getting my head off university, an underrated but vital asset when progress slows down. You have all been the best of motivations and I owe you my most heartfelt thanks.
I cannot forget my teachers, from school to university, who discovered to me the fascinating world of mathematics and computer science. The list is a long one, including (but not restricted to) José Carlos, Emma, Javi, Dmitry and Alberto Suárez. A special mention goes to my Artificial Intelligence teacher Batu, who first introduced me to artificial neural networks during my Erasmus year in Sweden and is ultimately the reason I chose them as my project topic.
I must also thank the people from CCC (Scientific Computing Centre at UAM), who helped me set up a work environment without which a good part of the computing workload would have been unmanageable.
To everyone above and those who deserve to be there despite my aim for concision, thank you.

# Contents

# Figure Index

# Table Index

# 1 Introduction

A new machine learning technique is deeply transforming the field of artificial intelligence and opening new possibilities which were considered unreachable only years ago: **deep learning**. Since the decade of 1950, artificial neural networks have been used and trained to solve classic machine learning problems: classification, prediction, function approximation, etc. These networks are inspired by the layers of neurons in the human *neocortex* (the newest layer in our brain in terms of evolutionary development), and they work by assembling a number of *artificial neurons* and adjusting the connections between them (training) in order to achieve a tiny and simplified functional model of a brain that performs one such task as accurately as possible. However, large networks are needed in order to solve more complex problems, and for many years this was beyond the capabilities of the available technology and knowledge: deep learning is the science of neural networks with many layers of artificial neurons. In those networks, new challenges arise. For one, training becomes much more computationally expensive and ineffective if done naively. Beginning in the 1990's, the new developments in the field of computer science began to turn deep learning into a real possibility. Firstly, the evolution of hardware led to exponentially more powerful systems and the ability to perform parallel computing. Secondly, research in the field of artificial neural networks gave birth to a number of techniques which addressed the additional issues associated to deep learning. Lastly, the amount of data needed to train large networks in many different areas started to be naturally generated by the increasingly omnipresent computer systems. It was the birth of realistic deep learning.

Today, deep learning is in fashion. Big efforts are being made in universities and companies from all over the world, papers on the subject are continually released and new students and experts are attracted to the field. The explosion in interest and new techniques in the field have allowed it to permeate many technology-related areas of our lives. One of the key assets of deep learning is its flexibility: problems of various types that could not be successfully treated with other machine-learning techniques (support vector machines, random forest, k-nearest neighbours, etc.) are now seeing some light thanks to it. Some of those problems are: image recognition (for instance, pedestrian detection in street cameras), speech recognition, recommendation systems, machine translation (and real-time video translation[1]), fraud detection and natural language processing. One of the advantages of deep networks is their computational lightness once they have been trained, which allows them to be used in all sorts of devices (in particular, they are becoming more and more widely used in smartphones and tablets). Many large companies, such as *Facebook* and *Google*, have acknowledged their use of large-scale deep

[1] See *'Google Translate vs. La Bamba'* on *YouTube*.

learning. In fact, *Google* themselves open-sourced their internal deep-learning library in 2016: TensorFlow, the framework we shall be using in this project.

This document contains a broad overview of deep artificial neural networks divided in three main chapters. In chapter 2, we introduce the theoretical foundations of the field. Starting with the abstraction of a human neuron and how multiple neurons are combined into a feed-forward network architecture, we explain various concepts involved in designing and training them. A mathematical description of **backpropagation** follows, a key algorithm used to calculate partial derivatives during neural network training. The chapter ends by highlighting the problem of overfitting and various techniques used to address it.

Chapter 3 focuses on the computer-engineering side of the topic, explaining first various TensorFlow concepts which need to be understood before using the library: the *graph*, *variables* and *sessions*. Other TensorFlow-related concepts are touched on lightly (such as collections and TensorBoard), and the necessary elements to designed and train neural networks are introduced. All of those pieces are put together in the second part of the chapter, which showcases the TensorFlow implementation of a very simple network for the *Iris* problem (flower classification).

Finally, chapter 4 shows some of the practical aspects of working with deep networks. We first introduce convolutional neural networks (a type of deep network used in image recognition) and related concepts. Then the MNIST problem (handwritten digit recognition) is introduced with some background history and perspective, including its current state. Finally, MNIST is addressed with three different convolutional networks, explaining how they were trained and the results obtained in each case.

We close out the report with some of our conclusions about TensorFlow and interesting topics for future work.

**Note**: The source files created in this project were uploaded to the author's **GitHub** repository, which can be accessed here:

`https://github.com/Antonio95/final-degree-project`

The repository contains two folders:

- `tensorflow_iris` includes the files used in chapter 2.
- `tensorflow_mnist` includes the files used chapter 4. The subfolder `results` contains many output files from the MNIST experiments performed in that chapter, which are important for the discussion. Summary tables are included in the report, although those files can be consulted for further details.

This source code can be downloaded and used if the TensorFlow environment has been set up as explained in annex A.

# 2

# Fundamentals of Feed-Forward Neural Networks

The aim of this chapter is to provide a theoretical background for artificial neural networks. From a very general starting point (an abstraction of the human brain), we describe a mathematical formalisation of these networks and the process of backpropagation, addressing some of the issues found in practice and a few additional techniques.

We begin by describing how feed-forward networks work and their structure, introducing the most common activation functions. The next section explains the various types of machine learning problems, focusing on *supervised learning* and some common cost functions which are used to evaluate the network's performance. We then proceed to the more technical justification of the backpropagation equations and some subtler details that may cause the so-called *learning slowdown*. The last two sections put forward the problem of overfitting and how it is addressed in practice (dataset splitting and regularisation).

## 2.1 Feed-Forward architecture

The structure of virtually any artificial neural network is composed of units (also known as neurons or processing units) which loosely resemble brain cells. In the human brain, a neuron receives a series of electric signals from its dendrites and fires a new signal through its axon if the total input reaches a certain threshold. The axon of a neuron is connected to the dendrites of many others (up to 10,000 [1]) through the so-called synapses, forming a complex web of nodes and links that process information.

The units of an artificial neural network perform a mathematical abstraction of this process: some input values are received from other units, and each of them is multiplied by a weight which in a way represents the strength of the connection between the two neurons. All of these values are added, together with an additional bias, creating an activation value for that unit. The bias is a constant which helps change each unit's threshold. The firing is then represented by a one-dimension-to-one-dimension mathematical function, the activation[1], which takes the activation value and yields an output which is sent forward.

A basic example of activation is the well-known Heaviside step function, which behaves like a switch. However, this mapping is not continuous and therefore not suitable for gradient descent (and related techniques), the most widespread training method nowadays. For that reason, other

[1]There is some ambiguity in the use of the word "activation" in artificial neural networks literature. The term can refer to either the activation function $f$, or to the activation input received by that function. Both meanings are used in this report, and which one applies should be clear from the context.

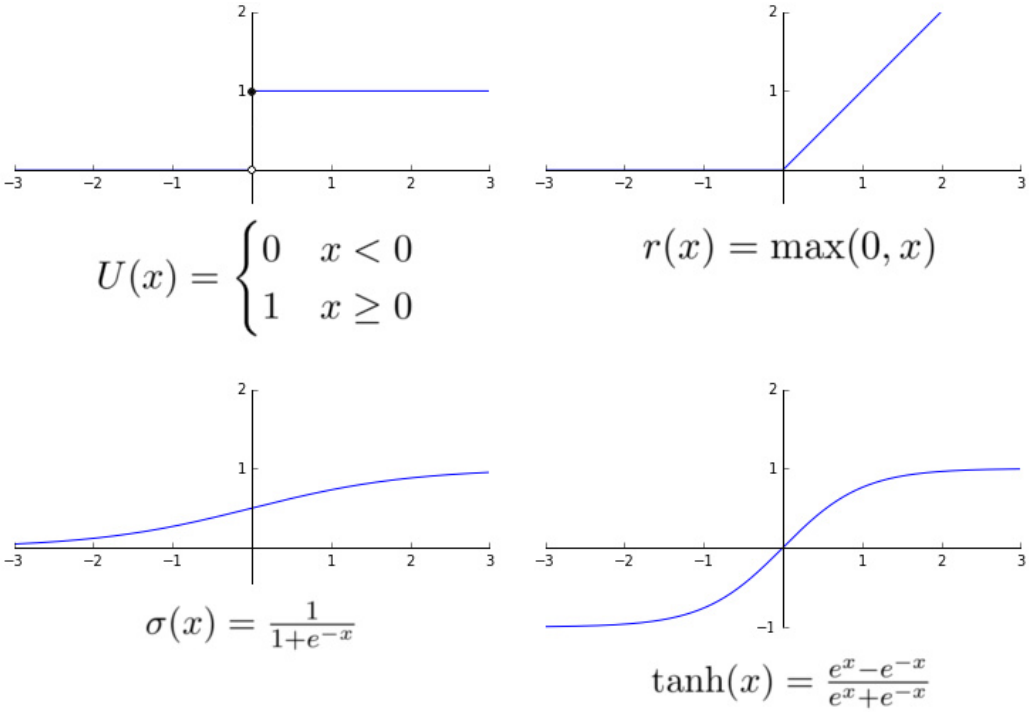



**Figure 2.1:** Common activation functions. From left to right and top to bottom: Heaviside step function, rectifier function, logistic (sigmoid) function and hyperbolic tangent. The function symbols have been selected according to standard, except for the rectifier, for which there is no general convention.

similar but continuous (and often differentiable) activations are used instead, namely the logistic sigmoid function and the closely related hyperbolic tangent. Other mappings are found to be useful for certain problems, such as purely linear activations or the rectifier function. A visual argument that neural networks with only sigmoidal units can approximate arbitrarily well any $\mathbb{R} \to \mathbb{R}$ function can be found in [2], but a rigorous one was given in a 1989 paper by K. Hornik, M. Stinchcombe and H. White [3]. It states that any $\mathbb{R}^d \to \mathbb{R}^d$ Borel-measurable[2] function can be approximated as accurately as desired by a feed-forward neural network with only one hidden layer and sigmoidal units. This is known as the *universal approximation theorem*.

The definitions and plots of the activation functions mentioned above are displayed in figure 2.1. The Heaviside step function is not continuous (and therefore not differentiable) and there are three common definitions of the value at the discontinuity: $U(0) \in \{0, \frac{1}{2}, 1\}$. The rectifier function is continuous but not differentiable and it is, together with the sigmoid functions, the most common activation in deep neural networks. Units having it as their activation function are often referred to as *rectified linear units*, or *ReLUs*, and they help to avoid a problem known as vanishing gradients. The sigmoid functions (named after their $S$ shape) are soft approximations to the step function, with $(0, 1)$ range in the case of the logistic, and $(-1, 1)$ in the case of the hyperbolic tangent. It should be noted that these two functions are closely related, since $2\sigma(2x) - 1 = \tanh(x)$. For that reason, they can be used in similar conditions (with some implications concerning weight initialisation, for instance), although, as a very general statement, tanh seems to lead to slightly faster convergence in practice[3].

[2] A similar notion to continuity, but using the Borel $\sigma$-algebra instead of the usual topology on $\mathbb{R}^d$.

[3] A possible explanation for this is given in [4]: it helps the layer's output to have close to 0 mean (as opposed to the logistic function, where it is always positive), which allows the weights in the next layer to be updated more flexibly when using backpropagation than if they all had the same sign. That is also the reason we always normalise (scale and take to 0 mean) inputs to neural networks.

Feed-forward artificial neural networks are those whose connections contain no feedback loops, as opposed to recurrent artificial neural networks, in which the output of a unit can eventually be fed back to itself. In other words, the associated graph[4] is acyclic. This type of network has the very convenient property that the output for a given set of features (from now on, these items shall be referred to as examples and their features as input) can be expressed as an explicit function of the input to the network, its weights and its biases. This fact allows for a simpler mathematical analysis than that required by recurrent neural networks.

All feed-forward networks can be thought of as being composed of layers of units. The first layer of units, known as the input layer, consists of as many neurons as input features our examples have. These units have no weights, biases or activation, but instead output one feature of the training example each. After the input layer comes any number of hidden layers, where neurons process the information internally. While it is theoretically possible to approximate any function with a single hidden layer, as stated above, this is not always practical. Deep learning, which consists of the use of multiple hidden layers, has arisen as a promising branch of machine learning thanks to the improved computational capabilities of today's machines. Its power resides in the fact that it allows for a certain hierarchy in the layers of units and the level of feature abstraction each layer captures. For instance, a deep neural network trained in face recognition can have a first hidden layer which recognises lines and geometric shapes in the image; a second hidden layer that combines these shapes to find eyes, ears, etc.; and a last hidden layer that takes those elements and determines whether they are placed as they would in a face. This would be impossible in a shallow network. The last layer is the output layer, whose number of units depends on the problem the artificial neural network is intended to solve. The types of problems addressed by neural networks will be discussed in the section 2.3, so for now it will suffice to say that the value returned by the activation functions in the output layer are the network's prediction for the input fed.

It is worth noting that feed-forward neural networks may have any choice of connections between units as long as the acyclic condition is satisfied. A particularly frequent type of feed-forward neural networks are fully-connected networks, in which every unit in each layer is connected to every unit in the next layer. However, this arrangement is not always optimal, since it involves heavier calculations and training than in a network with fewer weights. Additionally, leaving more free parameters than necessary can lead to overfitting[5], which shall be discussed later on. For that reason, many other possibilities are studied within the realm of feed-forward networks, such as connections skipping layers and units connected to few (if any) neurons in the next layer.

Figure 2.2 shows some examples of feed-forward network structures. The convolutional network offers a perfect example of layer hierarchy in deep neural networks: each unit in the first hidden layer is connected only to a certain region of the image (the *local receptive field*). For the sake of clarity, only the connections to one of the units are shown. After any number of filters, there is a downsampling layer and sometimes a fully-connected layer. This pattern can be repeated any number of times before the output layer. Convolutional networks shall be discussed and seen in action in chapter 4.

## 2.2 Types of problems and their cost functions

Artificial neural networks are very general function approximators. They are applied in countless problems, including spam detection, fingerprint recognition, computer vision, automated

[4]We can visualize the structure of an artificial neural network as a directed graph in which nodes represent units, and there is an edge from node $a$ to node $b$ if the output from unit $a$ is directly fed to unit $b$. We shall call this the *associated graph*

[5]The phenomenon occurring when the available training data becomes less abundant in relation to an increasingly large parameter space is known as the *curse of dimensionality*.

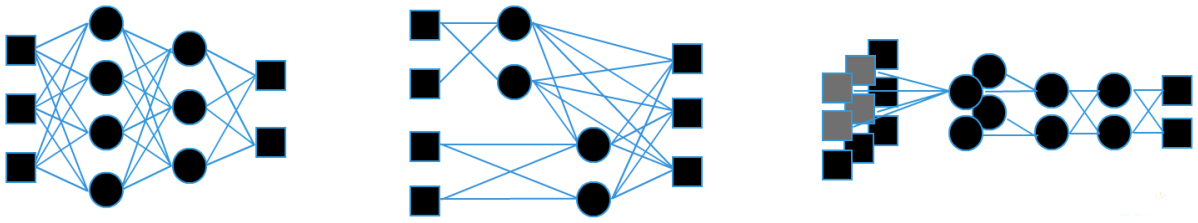


**Figure 2.2:** Various feed-forward structures. Left, a fully-connected deep neural network. Centre, an arbitrary network with connections skipping hidden layers. Right, simplified convolutional neural network.

driving and medical diagnosis. There is thus a need for many different implementations of the mathematical ideas behind them. Nevertheless, their training (and machine learning in general) typically falls into one of the following categories: supervised learning, unsupervised learning and reinforcement learning.

In *supervised learning*, many examples are shown to the network together with a target value for each of them. The goal is for the neural network to correctly predict the target value of new examples generated by the same mechanism. In *unsupervised learning*, examples are shown to the network without any specific target values. It must discover structures and patterns in the data by itself. An article in *Google's* official blog [5] explains how a one-billion-connection neural network (between 100 and 1000 times larger than the standard at the time) learnt the concept of a cat by itself by watching *Youtube* videos without any type of labels or pre-defined notions. The last learning paradigm is *reinforcement learning*, which is inspired on the behaviour reinforcement techniques present in psychology: no specific targets are given but rather each action within a certain context receives some positive or negative feedback, which helps the network evolve towards more beneficial decisions. An example of this are decision-making algorithms in computer games, where it is too computationally expensive to model every environment variable as an input feature. Reinforcement learning has the advantage that the feedback policy can be changed over time.

The level of development of supervised learning is higher than that of unsupervised learning, which is in turn a lot higher than that of reinforcement learning. From now on we shall focus on supervised learning, at which artificial neural networks often excel thanks to the computational power available nowadays and the backpropagation algorithm consolidated in the decade of 1980. The idea behind supervised learning can be roughly described as follows: an input example is fed to the neural network, which forwards the signal through its layers until the output is obtained. The error (also known as loss or cost) is calculated as a function of the output and the target result, and the network's weights and biases are adjusted in order to reduce this error. When this process is repeated for a large number of examples (which is often done through example batches rather than individual or on-line training), the cost as a function of only the weights and biases should have got close to a minimum[6], meaning the network will provide good (i.e. low-cost) approximations of the target value for new examples.

There are at least two noteworthy concepts in the previous paragraph. Firstly, it seems that the knowledge acquired by an artificial neural network resides in its weights and biases. That is the very difference between an untrained and a trained network (with the same original structure and applied to the same problem). These weights and biases are tuned thanks to input examples and target values only, so no understanding of the problem itself is needed. However it would be unwise to regard neural networks as a completely automatic tool, since some key elements must be chosen before the training starts. These include the network structure, activation

[6]Local minima are a troublesome aspect of neural network training, since a network has no knowledge of the overall aspect of the cost function and could 'get stuck' in a local minimum instead of a global one. Weight and bias initialisation, learning rate and other elements are tweaked in order to mitigate this issue

functions for each layer and data curation. Human knowledge of the problem helps choose those elements, and automatic training will only perform optimally if they are selected correctly. An example of this is provided by convolutional neural networks, widely used in image processing, where the input layer is often arranged as a two-dimensional array and filters are applied to the image to capture features of contiguous pixels as shown in figure 2.2. This would be impossible without our understanding of the problem logic. Secondly, the cost function lies at the very heart of supervised training. It is what the network will minimise during training and what the backpropagation algorithm uses when deciding how to update the weights and biases after examples have been shown to the network.

We are now ready to go over the classical types of problems in supervised learning, and then delve into the most common cost functions. We shall consider three sorts of problems: classification, regression and prediction.

Classification deals with determining which category a certain example belongs to, among a given number of possibilities. For instance, we could be interested in classifying pictures of mammals as 'monkey', 'cow', 'whale' or 'other'. The target values are $n$-vectors (where $n$ is the number of possible categories) which are filled with zeroes except for a one in the position corresponding to the true category. This is known as *one-hot encoding*, and usually involves the use of logistic activations (whose target set is $(0, 1)$) in the output layer. The last layer can be a *softmax* layer instead, which converts the output to a vector of membership probabilities to each categories. Specifically, softmax takes in a vector $a \in \mathbb{R}^n$ and yields a vector $o \in (0, 1)^n$. If we denote the $i$-th component of a vector $v$ as $v_i$, then:

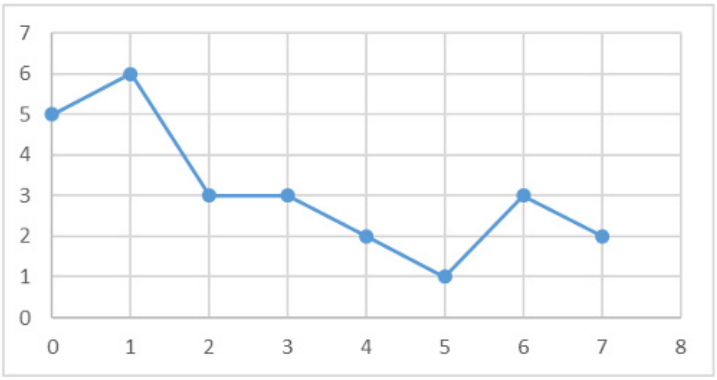
$$s(a)_i = \frac{e^{a_i}}{\sum_j e^{a_j}}.$$

Since those components are numbers between 0 and 1 whose total sum is 1, they can be interpreted as probabilities. It should be noted that, given the non-linear character of the exponential function, if we scale its input up, it will yield more extreme values (closer to 0 and 1), and if we scale it down, the output values will be closer to each other. Despite these subtleties and what they involve (regarding, for instance, weight and bias initialisation), softmax normalisation is widely used in classification problems.

Regression problems involve finding a numerical function of the input data. In this case, the target value of each example is the image of its features under that function (which often includes a small random error). For instance, we could try and calculate a person's life expectancy based on their weight, age, gender and average number of hours spent on sport per week. The target set can have more than one dimension, so it is possible to have more than one output unit. It may be tempting now to view classification as a particular of regression, since we could think we are trying to approximate the mapping that assigns to each example the category it belongs to. However, there is a key difference between them: the target set in a regression problem is nearly always continuous (or a union of continuous sets at least), whereas it is a discrete set of labels in the case of classification (which, in addition, have no numerical relation between them). This allows us to use one-hot encoding in the latter case only, and give the output a probabilistic interpretation.

Prediction problems try and forecast the value of a certain time series at a future time based on past values. This has countless applications in, for instance, financial advisory. Prediction problems can be transformed into regression ones[7] by considering a time window of $n$ time units in the past, and converting the history data into a set of examples where the input is an $n$-vector

[7]Since regression problems are a common task for traditional (meaning, in this case, feed-forward) neural networks, we can solve decision problems with those networks as well. A different approach to time-dependent problems is recurrent neural networks, which contain feedback loops and can, in a way, emulate short-term memory (and therefore do not need any time-window transformation). Recurrent neural networks outperform feed-forward ones in this task, but have a more complicated mathematical gear behind them.




| Input | Output |
|---|---|
| (5, 6, 3, 3) | 2 |
| (6, 3, 3, 2) | 1 |
| (3, 3, 2, 1) | 3 |
| (3, 2, 1, 3) | 2 |

**Figure 2.3:** Converting a time time series to an input-output format. The chosen window size is 4 time units.

of consecutive values and the target value is the value of the series in the next time unit (see figure 2.3). This is not a straightforward process and it has many subtleties. For one, if the window size is chosen to be too small there will be equal input values yielding possibly different outputs. If it is chosen to be too large, there will not be enough data to train a function with so many input variables (the curse of dimensionality again).

Each of these problems requires different types of cost functions, yet there are some restrictions that apply to all of them. Mainly, the range of the cost function must be one-dimensional (target set) and real-valued so that it has a gradient. Let us introduce now some of the notation we will be using throughout this text:

- $L$ is the number of layers in the network. They are numbered starting at 1, the input layer.
- $n_l$ is the number of units in layer $l$.
- $w^l_{jk}$ is the weight of the connection from unit $k$ in layer $l-1$ to unit $j$ in layer $l$. Units also start at 1. The reason for the apparent inversion in the subindices should become clear in a few lines.
- $b^l_k$ is the bias applied to unit $k$ in layer $l$.
- $a^l_k$ is the activation (sum of all inputs multiplied by their corresponding weights and the bias) received by unit $k$ in layer $l$.
- $o^l_k$ is the output of unit $k$ in layer $l$.
- $g^l(x)$ is the activation function of all units in layer $l$.
- $y^x_k$ is the $k$-th component of the target value for input $x$.

We omit the subindices to denote vectors (bias, activation, output and target value) and the weight matrix. For instance, with this notation

$$o^l = g^l(a^l) = g^l(w^l o^{l-1} + b^l) \quad if\ 1 < l \leq L\,, \tag{2.1}$$

where the activation function $g^l$ applied to a vector must be understood as a component-wise operation. Equation (2.1) shows the reason for the swapped indices in weights: it allows us to write the product of weights and outputs as a matrix applied to a column vector, which is familiar and convenient.

Before discussing particular cost functions, a comment should be made regarding their notation. The cost is primarily a function of the network's output $o^L$ and the target value $y^x$ for a particular input, so we may write $C(x) = C(o^L(x), y^x)$. This is known as the *local cost*, since it refers to a single example. However, $o^L$ depends on the network parameters (weights

and biases), so we may want to explicitly write $C(x) = C(o^L(x; w^1, ..., w^L; b^1, ..., b^L), y^x)$. This means that, even when using the local cost expression, it makes sense to write $\frac{\partial C}{\partial w_{ij}^l}$. Although the first expression is more compact, we will need to use the second form when, for instance, discussing L2 regularisation (section 2.5).

Let us start by considering a loss function for regression problems. Since we showed how prediction problems can be transformed into regression ones, this could be applied to both cases. The quintessential loss function used in regression is the *Squared Error* (SE). The formula for an individual example is:

$$SE(x) = \frac{1}{2} \sum_{k=1}^{n_L} (o_k^L - y_k^x)^2 \, . \tag{2.2}$$

As can be noticed, this is the square of the euclidean distance between the output and target vectors, with an adjusting factor of $\frac{1}{2}$ which slightly simplifies calculations (it cancels out when we take the derivative). The closer the output and target vectors are, the smaller the error becomes. It is a very natural solution which has been thoroughly tested throughout the history of artificial neural networks and statistical regression in general. Nevertheless, it has some issues related to learning slowdown when some of the output neurons are close to saturation (when their output values are in a flat part of the image, such as 0 or 1 in the case of the logistic sigmoid function). We shall address the reason for this in the section 2.3. Squared Error can be averaged over all examples in a mini-batch (or the whole dataset), becoming *Mean Squared Error*.

Classification has the peculiarity of one-hot encoding and its most typical cost function, *Cross-Entropy* (CE), is based on probabilistic methods, specifically maximum likelihood estimation. It may be tempting to use classification error (the ratio of examples wrongly classified) as our loss function, but it hides a lot of necessary information (for instance, it does not reflect whether the predicted class has been chosen with a very high or a very low degree of confidence). The cross-entropy cost function, which performs much better, has the following expression for a single training example:

$$CE(x) = -\sum_{k=1}^{n_L} y_k^x \; ln(o_k^L) \, . \tag{2.3}$$

All (2.3) means is the negative natural logarithm of $o$'s component whose target value is 1 (the correct class). It should be noted that when there are only two output classes, we may use a single output between 0 (class A) and 1 (class B) rather than two one-hot encoded vectors $(0, 1)$ and $(1, 0)$. In that case, we use the equivalent cost function (notice that the target $y^x$ and the output $o^L$ have one single element so we omit the subindices):

$$CE_{single}(x) = -[y^x \; ln(o^L) + (1 - y^x) \; ln(1 - o^L)] \, . \tag{2.4}$$

The main reason we cannot do this with more than two classes is that by assigning them a numerical value $1, 2, ..., n$, we would establish a non-existent mathematical relationship between them. The cross-entropy function is drawn from statistics, where it is used to compare different distributions. When applied to neural networks, it avoids the learning slowdown caused by saturated sigmoid neurons mentioned earlier, as we will show in section 2.3.

## 2.3 The backpropagation algorithm

While there exist techniques that do not require derivatives in order to find suitable weights and biases for an artificial neural network, such as genetic algorithms, the most widespread and powerful ones do make use of this mathematical tool. Gradient-based optimisation algorithms (stochastic gradient descent (SGD), momentum, adam, adagrad, adadelta, etc.) aim to minimise

the cost function $C$ by tuning the weights and biases in the layers of units following its negative gradient.

As mentioned earlier, feed-forward neural networks have the convenient property that the output (and therefore the cost of an example) can be expressed as an explicit function of the input, the weights and the biases in terms of multiplication, addition, and composition (of activation functions). It would be thus possible to directly obtain the desired partial derivatives by applying symbolic differentiation to the entire expression. However, this is not desirable, since said expression can become unmanageable for medium-sized networks, and many operations will be performed more times than necessary due to the repetitions in the structure of a typical feed-forward neural network. The backpropagation algorithm provides us with a much simpler way to compute those derivatives making use of matrix multiplication (which is remarkably convenient, given the efficient algebraic modules available today, particularly in Python, TensorFlow's language).

In order to show how the algorithm works, we shall now expand on the notation introduced earlier:

- $w^l = (w^l)_{j,k}$ is the matrix containing the weights between layer $l-1$ and layer $l$.
- $w^l_{:k} = (w^l)_k$ is the $k$-th column of the weight matrix to layer $l$. It is the vector of weights coming out of unit $k$ in layer $l-1$.
- $w^l_{j:} = (w^l)_j$ is the $j$-th row of the weight matrix to layer $l$. It is the vector of input weights to unit $j$ in layer $l$.
- $\odot$ is the Hadamard product, or component-wise multiplication of vectors or matrices.

We will now derive the fundamental backpropagation equations. A proof of them (exposed in a different order and with some differences from the one here) can be found in [2]. We start by defining the *generalised error* of unit $j$ in layer $l$ as

$$\delta^l_j = \frac{\partial C}{\partial a^l_j} \,. \tag{2.5}$$

The reason for this name is that, in the particular case of the squared error, that partial derivative coincides with the actual error (difference between output and target). Backpropagation is based on four equations relating those deltas with other quantities in the network. The goal is to express in simple terms the partial derivatives of the cost function with respect to weights ($\frac{\partial C}{\partial w^l_{jk}}$) and biases ($\frac{\partial C}{\partial b^l_k}$), since we will use those values to update our network towards more optimal configurations. Due to this fact, we need to study the cost for a fixed example as a function of the network's weights and biases, rather than a function of the input for fixed weights and biases.

The first backpropagation equation describes the error of neurons in the output layer. Since $o$ is the vector of outputs, which directly depend on $a^L$, we can use the chain rule to write:

$$\begin{aligned} \frac{\partial C}{\partial a^L_j} &= \sum_{i=1}^{n_L} \frac{\partial C}{\partial o^L_i} \frac{\partial o^L_i}{\partial a^L_j} = \\ = \frac{\partial C}{\partial o^L_j} \frac{\partial o^L_j}{\partial a^L_j} &= \frac{\partial C}{\partial o^L_j} (g^L)'(a^L_j) \,. \end{aligned}$$

In the first line we express the derivative of a function of many variables as the sum of the derivatives with respect to each of them, and then all the terms but one get cancelled because the only output that depends on our activation $a^L_j$ is $o^L_j$, and thus the derivative of the rest of them is 0. Then we express the second factor as what it actually is: the derivative of the

activation function. This yields the following formula for the **generalised error of an output unit**:

$$\delta_j^L = \frac{\partial C}{\partial o_j^L}\,(g^L)'(a_j^L)\,. \tag{BP I}$$

The second equation expresses the error of a neuron through the error of the neurons in the next layer. Provided $l < L$,

$$\delta_j^l = \frac{\partial C}{\partial a_j^l} = \sum_{k=1}^{n_{l+1}} \frac{\partial C}{\partial a_k^{l+1}}\,\frac{\partial a_k^{l+1}}{\partial a_j^l}\,. \tag{2.6}$$

We know we can express the activation of neuron $k$ in layer $l+1$ as:

$$a_k^{l+1} = w_{k:}^{l+1} \cdot g^l(a^l) + b_k^l\,. \tag{2.7}$$

If we take derivatives, we get:

$$\frac{\partial a_k^{l+1}}{\partial a_j^l} = w_{kj}^{l+1}\,(g^l)'(a_j^l)\,,$$

since the dot product depends linearly on each of the components of the vectors it involves. We can apply this back in equation (2.6), and also replace $\frac{\partial C}{\partial a_k^{l+1}}$ by $\delta_k^{l+1}$ (by definition), obtaining:

$$\delta_j^l = \frac{\partial C}{\partial a_j^l} = \sum_{k=1}^{n_{l+1}} \delta_k^{l+1}\,w_{kj}^{l+1}\,(g^l)'(a_j^l) = (g^l)'(a_j^l)\sum_{k=1}^{n_{l+1}} \delta_k^{l+1}\,w_{kj}^{l+1}\,.$$

If we express the sum as a product of a matrix row and the $\delta$ column vector, we get a simple matrix operation which becomes our second equation, the **generalised error of a hidden unit**

$$\delta^l = (g^l)'(a^l) \odot ((w^{l+1})^t \cdot \delta^{l+1})\,. \tag{BP II}$$

The third equation relates the error of a unit to the partial derivative of the cost with respect to any given bias:

$$\frac{\partial C}{\partial b_j^l} = \sum_{k=1}^{n_{l+1}} \frac{\partial C}{\partial a_k^l}\,\frac{\partial a_k^l}{\partial b_j^l} = \frac{\partial C}{\partial a_j^l}\,\frac{\partial a_j^l}{\partial b_j^l} = \delta_j^l\,\frac{\partial a_j^l}{\partial b_j^l}\,.$$

All the terms in the sum but one disappear because there is no relation between the bias of a unit and the activation received by the others. It follows from equation (2.7) that

$$\frac{\partial a_j^l}{\partial b_j^l} = 1.$$

Applying this to the previous equation gives us the **partial derivative of the cost with respect to a given bias**:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l\,. \tag{BP III}$$

Lastly, we establish the relation between the error and the partial derivative of the cost with respect to any given weight. Similarly to the previous equations,

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial a_j^l}\,\frac{\partial a_j^l}{\partial w_{jk}^l} = \delta_j^l\,\frac{\partial a_j^l}{\partial w_{jk}^l}\,.$$

Using equation (2.7) again, we can derive:

$$\frac{\partial a_j^l}{\partial w_{jk}^l} = o_k^{l-1}\,.$$

This yields our final equation, the **partial derivative of the cost with respect to a given weight**:

$$\frac{\partial C}{\partial w^l_{jk}} = \delta^l_j \; o^{l-1}_k \; . \tag{BP IV}$$

With those four ingredients, the backpropagation algorithm is simple:

1. **Forward propagation: Feed an input forward from the input layer to the output layer.** This allows us to obtain $a$ and $o$ for every unit in the hidden and output layers
2. **Backpropagation: transmit the partial derivatives backwards from the output layer to the first hidden layer**:
   (a) Calculate the generalised error in the output units with (BP I).
   (b) Use the gradients in each layer to calculate the generalised errors in the previous one with (BP II).
   (c) Compute the the partial derivatives of the cost function with respect to each weight and bias using (BP III), (BP IV) and the generalised error of each unit.

As can be inferred from such a simple recipe, backpropagation allows for easy calculation of gradients. However, there are many subtleties to it. We shall now describe a problem that may take place when training an artificial neural network using backpropagation: learning slowdown. Suppose we are teaching a network to classify data in $K$ categories. This means we have $K$ units in the output layer, and let us assume they have logistic activations (which would be a very reasonable set-up in this situation). Finally, suppose we have chosen squared-error loss. In order to train the $j$-th output unit, gradient descent (and related training methods) will compute $\frac{\partial C}{\partial w^L_{jk}}$, where $w^L_{jk}$ is any weight to that unit. We begin by calculating the error $\delta$ in that unit using (BP I):

$$\delta^L_j = \frac{\partial SE}{\partial o^L_j} \; (g^L)'(a^L_j) = \frac{\partial SE}{\partial o^L_j} \cdot \sigma'(a^L_j)$$

The first factor is the derivative of $SE$ with respect to the unit's output, which is

$$\frac{\partial SE}{\partial o^L_j} = \frac{\partial}{\partial o^L_j}\left(\frac{1}{2}\sum_{k=1}^{K}(o^L_k - y^x_k)^2\right) = o^L_j - y^j_k \; ,$$

since the derivative of every term except for one is 0. Finally, using (BP IV) and substituting the values calculated in the two previous equations, we get:

$$\frac{\partial SE}{\partial w^L_{jk}} = \delta^L_j \; o^{l-1}_k = \frac{\partial SE}{\partial o^L_j} \; \sigma'(a^L_j) \; o^{l-1}_k = (o^L_j - y^x_j) \; \sigma'(a^L_j) \; o^{l-1}_k \; . \tag{2.8}$$

As we can see, there is a $\sigma'(a^L_j)$ factor, which also shows up when taking the derivative of the loss function with respect to a bias in the output layer. This means (especially since $o^L_j$ is bounded above by 1) that the partial derivative will be very small if $\sigma'(a^L_j)$ is so, and this is common when working with sigmoid functions, which tend asymptotically to 1 when the input becomes large. Since gradient descent updates weights and biases according to the partial derivatives of the loss function, (2.8) means that, when a neuron saturates[8], it essentially stops learning.

[8]The saturation regions of sigmoid functions (logistic, *tanh* and similar ones) are the values of $x$ whose image is almost 1 (or whichever value the function tends asymptotically to for large values of $x$). At these points, the derivative becomes very close to zero. Note this can also happen for very negative values of $x$, since there is an asymptotic tendency in that region too (although it is not called saturation in that case).

Let us now motivate the seemingly arbitrary cross-entropy loss function as a solution to this issue: we wish to find a cost function $C$ which does not contain the $\sigma'$ factor in (2.8). That is, ideally:

$$\frac{\partial C}{\partial w_{jk}^L} = (o_j^L - y_j^x)\ o_k^{l-1}\ . \tag{2.9}$$

This is a sensible request, as it shows that the more different the output and target values are, the larger the partial derivative is (in absolute value), and therefore the quicker the learning pace will be. It is reminiscent of human learning: a typical learning curve shows fast learning at first (when the person is new to that skill), which becomes gradually slower as the person gets closer to the their 'maximum competence'. From (2.9) and (2.8) we deduce we need to cancel the $\sigma'$ term in the error, which is to say

$$o_j^L - y_j^x = \delta_j^L = \frac{\partial C}{\partial o_j^L}\ \sigma'(a_j^L)\ ,$$

which yields

$$\frac{\partial C}{\partial o_j^L} = \frac{o_j^L - y_j^x}{\sigma'(a_j^L)} = \frac{\sigma(a_j^L) - y_j^x}{\sigma'(a_j^L)}\ . \tag{2.10}$$

Let us now use a convenient property of the logistic function and its derivative:

$$\sigma'(a) = \frac{\partial}{\partial a}\left(\frac{1}{1-e^{-a}}\right) = \frac{-e^{-a}}{(1-e^{-a})^2} = \sigma(a)\ (1-\sigma(a))\ .$$

Substituting that in the denominator of (2.10) leaves us with

$$\frac{\partial C}{\partial o_j^L} = \frac{\sigma(a_j^L) - y_j^x}{\sigma(a_j^L)\ (1-\sigma(a_j^L))} = \frac{o_j^L - y_j^x}{o_j^L\ (1-o_j^L)}\ .$$

Integrating this expression with respect to $o_j^L$ results in

$$C = -[y_j^L\ ln(o_j^L) + (1-y_j^L)\ ln(1-o_j^L))] + A\ ,$$

where $A$ is an arbitrary integration constant. If we set it to 0, we obtain (an indexed version of) equation (2.4), the single-output cross-entropy cost function. The above argument, which can be extended to the cross-entropy cost for one-hot-encoded outputs, shows why using this cost function avoids the learning slowdown when logistic units saturate.

## 2.4 Training, validating, testing

A large amount of today's artificial neural network techniques owe their existence to the arguably most fundamental problem in artificial neural network training: overfitting. The following quote was attributed to Von Neumann by Nobel prize physicist Enrico Fermi when discussing the amount of free parameters in a certain problem: 'With four parameters I can fit an elephant, and with five I can make him wiggle his trunk'[9]. What Von Neumann referred to is the fact that statistical models with too many free parameters can be adjusted to any given data without actually capturing the pattern behind them. For instance, curve fitting yields questionable results when the degree of the polynomial increases, which can be seen in figure 2.4.

It is often the case for a neural network to admirably classify all the examples it is shown after some training, and then struggling when facing real-world (previously unseen) examples: it has overfit the training data. The underlying problem is that a large number of free parameters

[9]Freeman Dyson, *A meeting with Enrico Fermi* in Nature 427 (22 January 2004) p. 297.

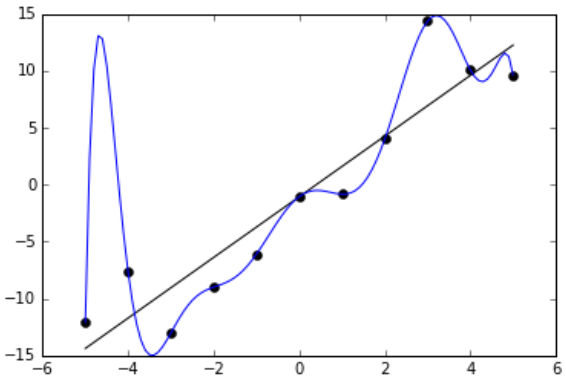



**Figure 2.4:** Linear (black) and ten-degree (blue) polynomials fitting a set of points. The polynomial with more free parameters (eleven coefficients) provides a perfect fit to those points but is an arguably worse representation of the model behind the data, since it captures the noise present in those specific examples. The linear fit will give better approximations on previously unseen points.

(weights and biases) involve high flexibility, and high flexibility allows the network to record the particularities of our training set rather than the overall patterns they follow. Overfitting is somewhat mitigated when the training set large enough that it broadly reflects every peculiarity in the model, and therefore the real power in today's artificial neural network relies on large amounts of information as much as it does on large computational capabilities. This highlights the tension between training set size and number of free parameters, which is depicted in figure 2.5. As we can see, when there are many free parameters and few examples, the network overfits the data and will fail to generalise when shown new examples. The distinction between few and many parameters is not so dramatic when training data is large enough, although too few free parameters may be too inflexible to fit the real model as accurately.

It should seem clear now that in order to assess our network's accuracy after training, we must evaluate it on previously unseen examples. For this reason, before training begins, some of the available examples are put aside and never shown to the network until it is fully trained. They are then used to evaluate the network's generalisation capabilities, and are therefore known as the *test* set. However, we also want to evaluate how our network is doing after some training iterations in order, for instance, to decide when to stop training. We therefore need another set, separate from the training set which we will use to evaluate our network's generalisation capabilities *during training*. This set, known as the *validation* set, is what will allows us to fine-tune other features (the so-called *hyperparameters*) than the weights and biases themselves, including number of hidden units and layers.

The argument sketched above leads to the *holdout* technique, present in virtually all neural network applications nowadays: dividing all the available data in training, validation and test sets. The size of each of them is a much-argued-about topic, but it is generally accepted that the training set must be significantly larger than the other two. Some conventions are 60-20-20 and 80-10-10 (respective set percentages).

However, a case can be made that by putting aside a validation set and using it as a measurement of our network's performance, we are fitting our training to that particular validation set over many iterations. In order to avoid this, we can use *cross validation*. In a frequent form, $k$-fold cross validation, we initially put aside the test set only. We divide the remaining examples in $k$ subsets, and train the network using $k-1$ of those subsets as the training set and the remaining subset as the validation set. We then repeat this process $k$ times, using a different validation set each time.

$k$-fold cross validation improves on the holdout technique by ensuring our validation set is a

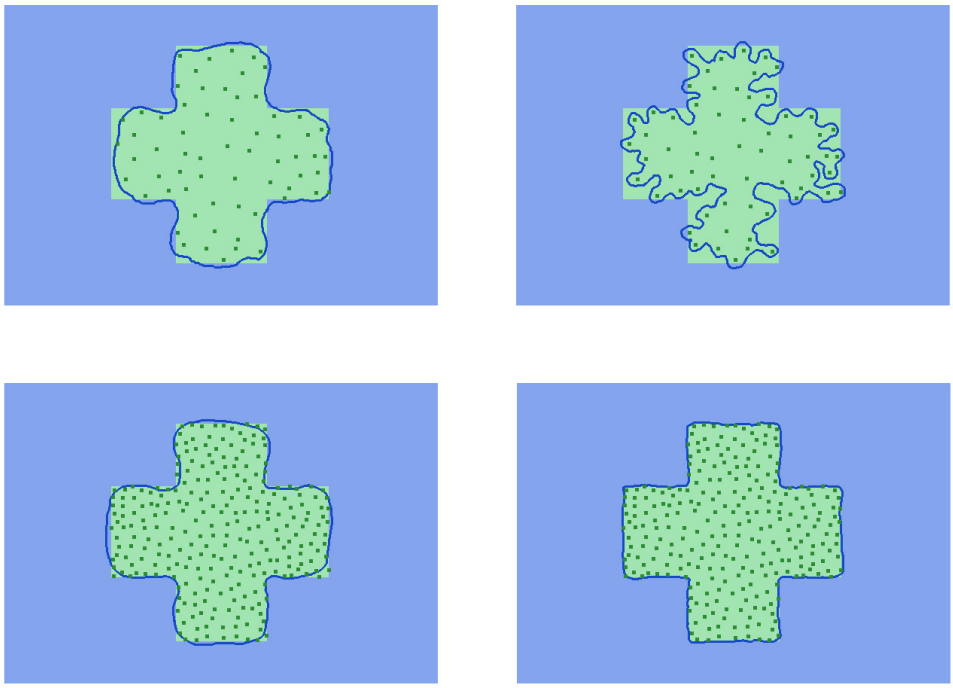


**Figure 2.5:** Overfitting. Background: real model. Green dots: training examples. Blue line: network's decision boundary after training. The four situations are as follows: Top row: few examples, bottom row: abundant examples. Left column: few free parameters, right column: many free parameters.

better representation of the whole of the data we have. A positive side effect of cross validation is that we make a better use of the data we have: if, for instance, we use traditional holdout with a 60-20-20 split, we are only using 60% of the available information to actually train the network, whereas with cross-validation we would use 80%. As has been shown above, the amount of training data is a key to training good artificial neural network with generalisation capabilities. This is even more true today, with powerful computers training networks with millions of parameters.

## 2.5 Regularisation

Thus far only some basic feed-forward neural networks techniques have been described. Although we shall not go deep into state-of-the-art methods, several tools have arisen in the last years which greatly improve the performance of these networks, particularly regularisation. This term comprises a variety of techniques that attempt to avoid overfitting by reducing the network's complexity (we can think about this as the principle of *Occam's razor* applied to artificial neural networks). Below is a brief discussion of early stopping, L2 regularisation and dropout regularisation.

### 2.5.1 Early stopping

While early stopping is not technically a regularisation technique, it is included in this section since it pursuits the same goal: avoiding overfitting. It was in fact the first attempt to do so apart from the dataset split described earlier.

Traditional training either proceeds until a certain threshold in the validation error is achieved

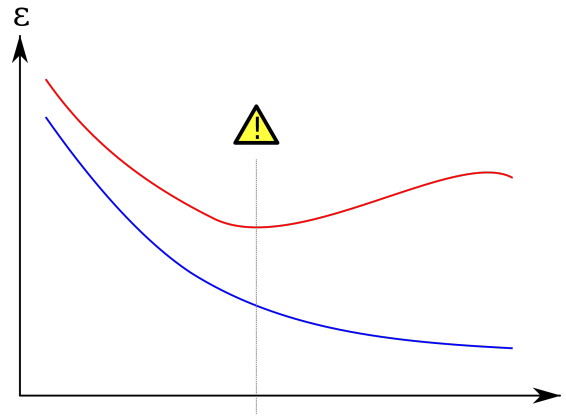



**Figure 2.6:** Validation (red) and training (blue) error over time. The best generalisation error is archived when the validation error reaches its valley, before the network begins to overfit the training set.

or runs for a predetermined number of epochs[10]. The former does not seem optimal since said threshold may never be reached (local minima, problem complexity, etc.) or could be easily improved. When using the latter, we could stop while the validation error is still decreasing, or even worse, we could see our training error decrease while the validation error (which is an estimation of the generalisation error[11]) begins to increase again. A well-known artificial intelligence diagram that reflects this phenomenon is figure 2.6 (image credit to Wikipedia[12]), a real example of which can be seen in section 3.2.

As we can see, there is a point after which validation error starts growing while the training error keeps on decreasing: we are overfitting the training data, which is the very thing we are trying to avoid. This makes it clear why we may want to apply early stopping: if we believe we are close to a minimum in the validation (and thus generalisation) error, we finish training before overfitting begins. A fine article on early stopping [6] suggests three classes of stopping criteria, of which only the third one shall be mentioned here since it is simple to understand and implement. It is defined as:

*$GP_s$: Stop training when the validation error does not decrease in $s$ consecutive steps. Keep the configuration with the lowest validation error up to that point.*

The idea behind it is as simple as it sounds: even with a large number of examples in the training set, overfitting will occur in the long run, and we want training to stop before that happens. This technique has become a lot less useful with the arrival of the more advanced techniques discussed in the next subsections. However, its interest has not disappeared but shifted from reducing overfitting to another very reasonable concern: preventing our computers from doing unnecessary calculations. Why keep our processors performing training steps (even if this does not cause overfitting, thanks to regularisation techniques) if that is not going to improve our network's performance? Stopping when the validation error ceases to decrease can avoid wasted computing time, which is especially important in the shared, costly processing centres often used to train deep neural networks nowadays.

[10] In machine learning terminology, an epoch is a pass over the whole training set. The number of training steps this translates into depends on the chosen mini-batch size.

[11] The generalisation error is the hypothetical error a network would make when shown an unlimited number of previously unseen examples. The closest quantity we can compute is the test error, followed by the validation error. The training error is, as we know, a bad estimate of the generalisation error.

[12] `https://upload.wikimedia.org/wikipedia/commons/thumb/1/1f/Overfitting_svg.svg/2000px-Overfitting_svg.svg.png`

### 2.5.2 L2 regularisation

*L2 regularisation* (also known as *weight decay*) aims to constrain how much the weights in the network grow. It has been empirically found that networks with smaller weights have less of a tendency to overfit data. Although it is probably an oversimplification, it could be argued that in a network with larger weights, each example has a greater impact on the final result, whereas a network with small weights may only be responsive to patterns present in larger amounts of data, precisely as we want. This technique aims to limit weights by adding a regularisation term to the cost function:

$$C_r(o^L(x; w^1, ..., w^L; b^1, ..., b^L), y^x) = C(o^L, y^x) + \frac{\lambda}{2} \sum_{l=1}^{L} \|w^l\|^2 \tag{2.11}$$

where $C$ is the original cost function and $\|w^l\|^2$ denotes the sum of the squares of all elements in $w^l$. This means configurations with higher weights (in absolute value) have a higher cost. The regularisation parameter $\lambda$ determines how much large weights are handicapped: the higher $\lambda$, the more these high-weight configurations lag behind. The new gradient can be easily computed, since:

$$\frac{\partial C_r}{\partial w^l_{kj}} = \frac{\partial C}{\partial w_{kj}} + \lambda w_{jk} \tag{2.12}$$

$$\frac{\partial C_r}{\partial b^l_j} = \frac{\partial C}{\partial b^l_j}. \tag{2.13}$$

Equation (2.12) presumably makes it clear why the method is called weight decay: the partial derivative of the new cost function with respect to a given weight increases (or decreases if the weight is negative) so when the gradient is subtracted from the weights, that term tends to reduce the weight's absolute value. This means configurations with larger weights are in disadvantage with respect to small weights. Of course, large weights can be achieved, but only as long as they bring a significant improvement in the accuracy. While it is possible to include the biases in L2 regularisation, this is not often done. The reason is, it does not seem to reduce overfitting noticeably, and large biases help sigmoid neurons saturate, which is sometimes desirable.

As an additional note, the 2 in L2 refers to the power weights are raised to when adding them together in the cost function. Other L regularisations exist, of which L1 is the most frequent one, but we shall not discuss them here.

### 2.5.3 Dropout regularisation

*Dropout* is an original and relatively recent idea (suggested in a paper from 2012 [7]) that aims to reduce overfitting by making each individual unit learn meaningful features rather than need other units in conjunction in order to detect patterns in the data (which is known as co-adaptation). An initial attempt to do this would be to average all the possible neural configurations (weights and biases) with an averaging weight equal to their success rate. As can be expected, however, this process is too computationally expensive. Dropout is a viable alternative to this which has obtained better results in practice than many other techniques, with only a moderate increase in computational cost.

A detailed description of this technique can be found in [8]. Essentially, when training a network using dropout, a proportion $p$ of the total number of neurons is 'dropped' for each mini-batch, as shown in figure 2.7. What this means is they do not forward any signal during the feed-forward pass (their output is multiplied by 0). After the backpropagation update (which is done normally, with the real unit outputs), a proportion of the units is randomly dropped again and the process is repeated. However, the final network contains all units, so after training (and

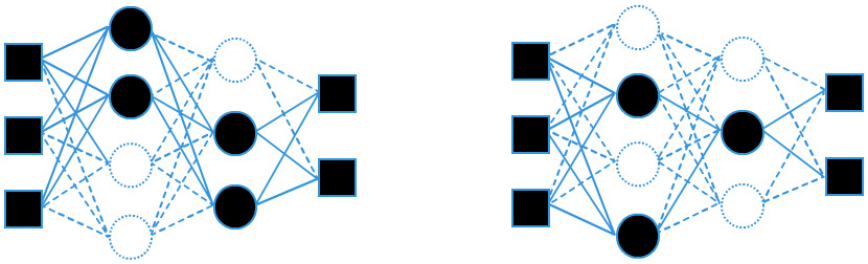


**Figure 2.7:** Two configurations of the same artificial neural network when trained with on different mini-batches using dropout regularisation. The units with dotted borders do not forward any signal and are therefore functionally deleted, which forces units to learn individually meaningful features rather than rely on other units (co-adaptation).

whenever the validation loss needs to be calculated) the weights are multiplied by $p$ to make up for the fact that they were trained when only some of the units were present (so that the expected value remains the same).

Dropout regularisation can be used in conjunction with most training methods (SGD, momentum, etc.) and other regularisation techniques (according to [8], it is especially powerful together with max-norm regularisation[13]), and it is among the most effective neural network tools available nowadays.

[13] In max-norm regularisation, the length of the vector of incoming weights to any particular unit is clipped above by an arbitrary constant $c$. If it surpasses that value at any point during training, the vector is rescaled to have a norm equal to $c$.

# 3

# A First Approach to TensorFlow

This chapter introduces TensorFlow and its python API, from the key concepts and its internal workings to a practical example of use. We begin by explaining how it is different from other numerical computation libraries (such as *numpy*) and the fundamental division between graph definition and execution. Next we dive into each of these concepts, describing ***tensors***, ***ops*** and how they are combined to form the operation graph and how sessions perform the defined operations, receiving and returning different data. The discussion of TensorFlow ends with a brief explanation of ***variables*** and how they are treated during neural network training. The second part of the chapter illustrates all of these concepts in practice through the discussion of a small program for the classification of flowers of the Iris species.

## 3.1 TensorFlow basics

TensorFlow is a software library for large-scale numerical computation (focusing on machine learning) developed by Google and open-sourced under the Apache 2.0 license in November 2015, when its whitepaper [9] was published. It is Google's continuation to their original DistBelief library within Google Brain (a deep learning research project), bringing about better performance, support for a wider variety of platforms (including smartphones and parallel systems) and a cleaner programming interface based on the lessons learned thanks to DistBelief.

This section explains some of its key concepts. More advanced ones, like distributed execution and the placement algorithm, are not discussed here but can be found in the whitepaper. Due to its complexity and the fact that it can be used in a distributed or single-device fashion, the installation of TensorFlow can be tricky. Annex A explains a possible way to go.

### 3.1.1 The TensorFlow graph

The way TensorFlow works is noticeably different from other numerical computation libraries such as *numpy* in that the definition of operations and their execution are separate pieces of code. The computation definition takes the form of a directed graph whose nodes are operations (called ***ops***) and whose edges are the ***tensors*** passed between them[1]. ***Tensors*** can be thought of as multi-dimensional arrays, and an ***op*** takes in any number of ***tensors***, performs a certain operation and outputs any number of ***tensors*** (including 0 in both cases). ***Ops*** can perform a wide range of actions including linear algebra, I/O functions and control flow operations.

[1]There is an additional type of edges called *control dependencies*, which are used to enforce that a certain ***op*** is executed before another.

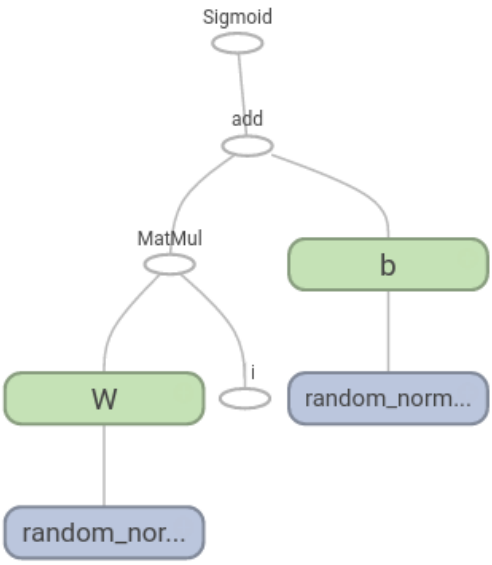



**Figure 3.1:** Simple graph with the operations that compute the output of a sigmoid unit. The variables `W` (weight vector) and `b` (bias) have their corresponding `random_normal` initialisers. `W` is multiplied by `i` and added together with `b`, which is then passed through a sigmoid activation.

Each ***op*** is implemented internally in the form of many kernels, each of which performs that operation in a given platform. This allows for the same graph definition to run optimally on multiple devices by only slightly modifying the code.

Let us now see how to define a TensorFlow graph. It should be noted that two APIs are currently available: a python one and a *C++* one[2]. We will show python examples only, since it is the language the rest of this project is written in. Also, the line importing TensorFlow will be assumed to be `import tensorflow as tf` in the examples, since this saves some typing and it is the standard practice. Let us consider our first piece of TensorFlow code, which computes the output of a layer of units in an artificial neural network:

```
import tensorflow as tf

W = tf.Variable(tf.random_normal((3, 4)), name='W')
b = tf.Variable(tf.random_normal((3, 1)), name='b')
i = tf.placeholder(tf.float32, (4, 1), name='i')
o = tf.sigmoid(tf.matmul(W, i) + b)
```

The resulting graph can be seen in figure 3.1 Many elements may still be unclear there, but let us focus on the last line. As we can see, the object returned by many TensorFlow methods (in all of the above cases, a ***tensor***) can be passed as an argument to other constructors when the output of the former element is an input to the latter. But even more interestingly, we can use operators such as `+` with ***ops***. The reason is TensorFlow overloads them so that if any operand is a `tf.Tensor`, a corresponding ***op*** is created and added to the graph. There is a small difference between using python operators and the full methods (e.g. `add(tf.matmul(W, a), b)`): in the second case we can specify additional parameters, such as the ***op*** name. The full list of operators that are overloaded by the API can be found in annex B

TensorFlow allows the definition of multiple graphs, but most programs only use the default one, which is available as `tf.get_default_graph()` after importing the module. When an ***op*** is created, it is automatically added to the default graph (which is initially empty). If additional graphs are desired, the programmer must use context managers, which override the default graph within their scope so that all the operations added go into that context's graph. For instance:

[2]In fact, new APIs for *Go* and *Java* were released during the time of writing. They are is still experimental.

```
g = tf.Graph()
with g.as_default():
    v = tf.Variable(tf.zeros([10]))
    assert v.graph is g
```

The default graph can also be reset (its operations removed) with `tf.reset_default_graph()`.

### 3.1.2 TensorFlow sessions

Sessions are the second half of the TensorFlow process: operation execution. Regular `tf.Session` objects must be constructed after all the operations have been added to the graph, and only then can their operations be executed. There exists an additional class, `tf.InteractiveSession`, which allows to interleave graph-defining code and execution, intended for interactive environments such as python consoles.

The session constructor takes an optional argument with the graph to be launched (if not specified, the default graph is used). One session can only execute operations from one graph, but the same graph can be used in multiple sessions. Another optional argument is an execution engine to connect to (this is only relevant in situations of explicit multiprogramming). If not specified, an internal TensorFlow engine is used called the default session. The method `Session.close()` frees several resources related to it, and can be omitted if we use a context manager (which works similarly to that of graphs) except if we are using a non-default session.

Leaving aside the intricacies of session definition, the way they are used is fairly straightforward. The two key concepts are feeds and fetches, which are passed to the `Session.run()` method whose prototype is:

```
tf.Session.run(fetches, feed_dict=None, options=None, run_metadata=None)
```

.

Fetches are the elements in the graph we want to compute during that particular execution. It can be a single graph element (primarily a `tf.Tensor`, a `tf.Operation` or a string with the name of one of those), or a list of them (other more general iterables are admitted). When this method is called, TensorFlow determines which ***ops*** need to be executed in order to run the fetched ones (for instance, the product of two ***tensors*** requires us to compute whichever operations yield those ***tensors***). This is called the transitive closure of the requested nodes in the graph, and it can be thought of as if we were "pulling" the nodes whose result we want, so that all nodes which feed them are pulled as well, and so on.

Feeds (`feed_dict`) are an optional argument which allows to override the output of certain nodes. It is a dictionary whose keys are tensors in the graph and whose values are the output for each ***tensor*** to produce (for instance, a scalar value, a string, a *numpy ndarray*, etc.). A specially common type of overridden ***tensors*** are placeholders. They are constructed as `tf.placeholder(dtype, shape=None, name=None)`, and they represent the inputs to our graph for that particular execution. They cannot be evaluated, and their value must necessarily be specified in the `feed_dict`. An advantage of using placeholders for input rather than any other nodes (besides cleaner code, since they are specifically meant for that task) is that TensorFlow will check the shape and type of the tensor (if specified) against the value's.

If the execution of only one ***op*** `o` or ***tensor*** `t` is desired, an more concise alternative to `Session.run` is:

```
o.run(feed_dict=None, session=None)  or  t.eval(feed_dict=None, session=None).
```

If `session` is `None`, the default session is used, so `t.eval(a_feed_dict)` is equivalent to calling `tf.get_default_session().run(t, feed_dict=a_feed_dict)`. This is very convenient, and a good reason to use context managers with the method `tf.Session.as_default()`.


**Figure 3.2:** Nodes added by TensorFlow when we create a ***variable*** as `tf.Variable(tf.random_normal((2, 2)), name='w')`. We can see the node with the initial value `random_normal`, the ***variable*** `W` itself and the `assign` ***op*** that will pass the value from one to the other upon initialisation.

It should be noted that the python object returned when we call a method to add an ***op*** to the graph can be a `tf.Operation` itself, or the `tf.Tensor` that operation outputs (which determines whether we use `tf.eval` or `tf.run` respectively), and the distinction can be subtle. Examples of constructors returning ***tensors*** are most calculations, such as `tf.add` and `tf.exp`. Constructors which return the ***op*** itself include `tf.train.Optimizer.minimize` and `tf.global_variables_initializer`, which leads us to the next topic.

### 3.1.3 TensorFlow variables and training

A question that could have arisen naturally from the previous lines is: what happens with tensor values when we execute some nodes, switch back to python and then run some nodes again? The underlying computation engine is only terminated when the python process that imported TensorFlow finishes, but it is unclear what happens during context switching within the same program. The answer is those values are lost, which could make it unsuitable to work with this library since we will most likely need to interleave python instructions and TensorFlow operations during our process. However, TensorFlow offers a program-level persistent type of nodes: ***variables***.

The constructor of a `tf.Variable` receives many default arguments (in fact, it has no non-default ones), the most relevant of which are arguably `initial_value`, `dtype` and `trainable`. `initial_value` receives any tensor-convertible argument[3] and sets it as the desired value for that ***variable*** when we initialise it. `dtype` specifies the type of data the ***variable*** will store (and tells the constructor to convert the input to that type, raising an error if not possible). `trainable` is a more interesting boolean parameter defaulting to `True` which tells TensorFlow whether the ***variable*** is to be trained by optimizers (for instance, a weight matrix in an artificial neural network). If so, the ***variable*** is added to the graph's `GraphKeys.TRAINABLE_VARIABLES` collection, which determines with respect to which ***variables*** the `tf.Optimizer` will calculate gradients.

Once all ***variables*** in a graph have been defined, we need to add the `tf.global_variables_initializer()` operation. This assigns to each ***variable*** the argument which was passed to its constructor, and it should be the first operation we run once execution begins. It is possible to assign a particular value to a ***variable*** during execution time as well, by adding `tf.assign` nodes to the graph. In fact, when we call the `tf.Variable` constructor, TensorFlow adds three nodes: the ***variable*** itself, the `assign` ***op*** and the initial value ***tensor***, as we can see in figure 3.2. We can also initialise a ***variable*** `u` with the value from another one, `v`. In this case, the `initial_value` we need to pass to `u`'s constructor is a special tensor: `v.initialized_value`. The reason for this is that `tf.global_variables_initializer()` returns an ***op*** which, when executed, initialises all ***variables*** in parallel, so `v` might not be ready to feed its value to `u` during initialisation. We can also define custom initialisation (some ***variables*** only) using

[3] All TensorFlow methods that receive a ***tensor*** can receive equivalent objects instead, including *numpy* arrays, python lists and scalars. These methods internally call `tf.convert_to_tensor` on those arguments, which yields a `tf.Tensor` object and also allows to specify parameters such as name and preferred `dtype`.

the `GraphKeys` class and `tf.variables_initializer(var_list)`, although this is not a very common TensorFlow practice.

The TensorFlow API also provides a simple way to save and load the values in `tf.Variables` from/to disk. This is in fact the *go-to* way of saving and loading artificial neural networks when using TensorFlow. In order to do so, we create a `tf.train.Saver()` object, say `saver`. This adds `save` and `restore` nodes to the graph which are stored as attributes of the object returned by the constructor. We then call `saver.save()`, which receives the session and the desired path, in order to write to disk; or `saver.restore()`, which receives the same arguments (the path is a source instead of a destination) and assigns the values from disk to the ***variables*** (restored ***variables*** do not need to be initialised). The way `restore` knows which value to assign to each ***variable*** is through the ***variable***'s `name` attribute, which must have been set for all ***variables*** involved, both when saving and restoring. If only certain ***variables*** need to be saved, we can use the optional argument `var_list` in the constructor `tf.train.Saver()`, a dictionary whose keys are names (strings) and whose values are the ***variables*** to be stored under each name.

As we have seen, there is a close relation between ***variables*** and training. TensorFlow comes with an large sub-module devoted to machine learning: `tf.train`. One of its main components are optimisers, which compute and apply gradients for us. The `tf.train.Optimizer` class cannot be constructed directly, but is implemented by various subclasses (eight at the time of writing) which correspond to common optimisers, such as `tf.train.GradientDescentOptimizer`, `tf.train.AdadeltaOptimizer` and `tf.train.RMSPropOptimizer`. Although it has not been described in Chapter 2 for lack of space, those are various ways of updating weights and biases after each gradient descent step: some of them include an adaptive learning rate (larger at first and smaller as we get close to a minimum), others use a momentum so that our point in the parameter space moves taking into account previous updates (a sort of acceleration), etc. A detailed description including an animation comparing them can be found in [10]. The code to create a basic stochastic gradient descent optimiser in TensorFlow is:
`opt = tf.train.GradientDescentOptimizer(learning_rate=<x>)`.

After creating an optimiser, we will likely want to use it to minimise the cost function, and we are given two alternatives to do so:

- **Using the built-in optimisation method automatically**. This possibility consists of calling `opt.minimize(loss)`. The only mandatory argument is the loss function: an ***op*** in our graph that outputs the cost for a given input and target. An optional argument is `var_list`, which indicates the list of ***variables*** to compute the gradient with respect to (and therefore to update). As stated in the python API documentation, the default value is `GraphKeys.TRAINABLE_VARIABLES`, which was expected from what we described earlier in this section.

- **Fine-tuning how the gradient is used**. This option splits the training step into three stages: calculating the gradient, processing it as desired, and applying it to the ***variables*** involved. Gradient calculation is achieved with the `tf.train.Optimizer.compute_gradients(loss)` method, which returns a list of (*pd*, *var*) pairs, where *pd* is a ***tensor*** containing the partial derivative of the loss with respect to a ***variable***, and *var* is that `tf.Variable`. Processing gradients is done by the programmer in this case, using those ***variables*** and any other nodes in the graph. When the final gradient to be applied is ready, we use `tf.train.Optimizer.apply_gradients(grads_and_vars)`, which returns an ***op*** that can be run in order to update the ***variables***. The argument it takes is a list of (*pd*, *var*) pairs with the value to be applied to each ***variable***. We can also pass an optional argument, `global_step`, which is a ***variable*** to which TensorFlow will add 1 whenever parameters are updated (it is simply an alternative to keeping the count manually in python).

It is worth noting that `opt.minimize(loss)` returns `opt.apply_gradients(opt.compute_gradients(loss))`. In other words, it is simply a wrapper that skips the "fine-tune gradients"

stage. The *compute gradients* stage is the same for all optimisers, and it is the *apply gradients* phase that characterises each `tf.train.Optimizer`. It might also seem odd that we specify the loss function when we declare the train step, and not in the constructor of the `Optimizer` subclass. However, this is in fact a more flexible design: we may want to train several independent elements simultaneously with the same training method.

Those are the essential concepts to be aware of when training with TensorFlow. Finer aspects of training methods include *gates* (which allow us to manage the degree of parallelism when computing gradients) and *slots* (additional ***variables*** required by some training methods, such as the momentum), but they shall not be discussed here.

## 3.2 A TensorFlow example: Iris flower classification

After the basic concepts of TensorFlow have been covered, we are in good shape to create and train our first neural network with it. The chosen problem is a classic example of very basic machine learning: the Iris dataset, published by Ronald Fisher in a paper from 1936. The dataset contains 150 examples of flowers belonging to three species: *Iris setosa*, *Iris versicolor* and *Iris virginica*. Each example consists of four measurements for a certain flower (length and width of sepals and petals) together with the flower's species. The task is simple: use machine learning to classify them correctly. In our case, train a neural network to do so using TensorFlow.
**Note**: the files used in this section can be found in:
`https://github.com/Antonio95/final-degree-project/tree/master/tensorflow_iris`

Before delving into the actual program, it should be noted that it uses a small python module created by us in order to handle datasets (loading, normalising, randomising, splitting, etc.). A brief description of the module's interface and functionality can be found in annex C.

We can now move onto Iris flower classification. Regarding the chosen network, the number of layers and the amount units in each layer is controlled by a list. The chosen cost function is cross-entropy with sigmoid activations (and therefore one-hot-encoded targets), and L2 regularisation is applied. Both early stopping and training for a fixed amount of steps have been tested. A clarification should be made about the fully-matrix-based approach of modern neural networks software. Most of the equations in chapter 2 describe what is known as on-line training: the network is shown one example at a time. However, stochastic gradient descent (and related methods) trains the network with a mini-batch of many examples at once[4]. An advantage of mini-batch training is that we can convert the feed-forward and backpropagation operations into matrix calculations. Essentially, the output column vectors become matrices, where each column corresponds to a single example. For instance, equation (2.1) becomes:

$$O^l = g^l(w^l \cdot O^{l-1} + B^l), \tag{3.1}$$

where the output vectors $o^l$ and $o^{l-1}$ have become matrices, the activation function $g^l$ is still applied element-wise, and the bias vector $b^l$ has become a matrix $B^l$ (all of its columns are the same, $b^l$, since the bias is the same for all examples). This matrix-based approach makes mini-batch training a lot more efficient than on-line training, since matrix operations are extremely fast in today's computational frameworks. It is the standard in neural network implementations for large problems nowadays, and it is the approach used in this project as well.

After that clarification, we are ready to see the code itself. Note that the line numbers displayed here are for reference within this report and do not correspond to those in the source

[4]The idea behind it is that the average of the gradient over many examples constitutes a good approximation (from a probabilistic point of view, it is the expected value) of the average gradient over the whole dataset, and it is much faster to compute. Mini-batch training usually has a larger learning rate (step size in our parameter space) to make up for the fact that it takes fewer (but more accurate) steps than on-line training.

code. Additionally, many comments, blank lines and pieces of code unrelated to this discussion have been omitted. Leaving aside the header and `import` statements (including `import settings as st`, which loads the settings file with the learning rate, mini-batch size, etc.), the program starts with some housekeeping: loading our current path and detecting the existing execution logs to assign the next integer to the current one. The first few lines after that prepare the Iris dataset using the module described earlier:

```
1  ds = Dataset(base_dir + '/aux_files/iris_dataset.csv',
       'classification', 'iris_dataset')
2  ds.to_one_hot_encoding()
3  ds.normalise()
4  ds.split(0.6, 0.2)
```

The first line loads the dataset from disk, specifying it corresponds to a classification problem. This sets the module up internally so that line 2 is able to transform the target values, which are read as strings, into one-hot-encoded vectors (our desired representation). Normalisation is applied in line 3. Although this topic was not discussed in chapter 2, normalising the input (scaling it to a different range than the original) has many very desirable properties in machine learning, and is therefore standard practice. In this case, standard normalisation is applied (the default option). This method takes the values of each input feature and scales them linearly into a standard distribution ($\mu = 0, \sigma = 1$). Finally, line 4 splits the dataset into training (60% or 90 examples), validation (20% or 30 examples) and test (20% or 30 examples) sets, shuffling them randomly (the validation percentage is inferred from the other two). The next few lines contain some set-up for the network structure:

```
5  h_l = [4, 3]
6  if not h_l:
7      raise ValueError()
8  activation_f = tf.nn.sigmoid
9  n_classes = ds.get_n_classes()
10 n_features = ds.get_n_features()
11 layers = [n_features] + h_l + [n_classes]
```

The list in line 5 describes the hidden layers: starting with the layer after the input, each number describes a new hidden layer with that many units. The program is flexible in that this list can be modified to any valid hidden layer set-up. Line 8 simply centralises the definition of the activation function. We have used the same activation for all units in the network: the logistic sigmoid, which is a TensorFlow built-in. Note that this line only defines the function as a variable: it does not add any nodes to the graph since we are not instantiating any nodes (which would be done with `tf.nn.sigmoid()`). Regarding the use of sigmoid units vs. ReLUs, the network is shallow enough that vanishing gradients[5] are not an issue. Lines 9 through 11 complete the list of layers, adding an input and an output one (the number of units in each is retrieved from the dataset, although it is well-known beforehand). The next lines create the core network structure:

```
12 input_ = tf.placeholder(tf.float32, shape=[n_features, None],
       name='l1/a')
13 op_weights, op_biases, op_outputs = [None], [None], [input_]
14 for i, n in list(enumerate(layers))[1:]:
15     with tf.name_scope('l' + str(i + 1) + '/'):
16         op_normal = tf.random_normal([n, layers[i - 1]],
               stddev=np.sqrt(n), seed=st.SEED)
17         op_weights.append(tf.Variable(op_normal, name='w'))
18         op_bias_normal = tf.truncated_normal([n, 1], seed=st.SEED)
19         op_biases.append(tf.Variable(op_bias_normal, name='b'))
20         activation = tf.matmul(op_weights[-1], op_outputs[-1]) +
               op_biases[-1]
```

[5] An interesting phenomenon that arose with the spread of deep neural networks are vanishing gradients: when applying backpropagation, the partial derivatives become smaller and smaller as we get closer to the input layer, so much so that learning becomes very slow in the first hidden layers. With the growth in depth of deep nets, this became a real problem. The solution came in the form of ReLUs, which have a (piecewise) constant derivative

```
21          op_outputs.append(activation_f(activation, name='o'))
```

Line 12 creates the placeholder for the input to the network, so it is the first line to actually add a node to the graph. The shape of this layer, `[n_features, None]`, indicates we want `n_features` rows (as corresponds to each input vector), and an unspecified number of columns, which gives us flexibility regarding mini-batch size. It is also implicit in this line that we are using TensorFlow's default graph described in subsection 3.1.1, since no `with g.as_default()` scope has been created. Line 13 defines three lists which will contain the weight, bias and output matrices. A padding of one element (`None`) has been added to the first two lists, since the first layer has an output, but no weights or biases.

The loop defined in line 14 iterates over those lists of layers, creating the nodes that compute each layer's output. First, a random normal distribution is created in order to initialise weights[6]. The dimension of the weight matrix created in lines 16 and 17 is obtained from the number of units in the layer and in the previous one. Then we add the bias vector in our graph in a similar way.

Line 20 contains the key operation in our network: calculating the activation for the units in the next layer. We do so by multiplying the weight and previous output matrices, and adding the bias. It should be noted that it is possible to add a matrix and a vector together. TensorFlow's variable casting policy (inherited from *numpy*) translates this to adding that same vector to all rows or columns in the matrix if possible. Another convenient fact is that the weights and biases of the current layer have already been appended to the list, but not so the output (which is in fact being calculated), so we simply take the last element in each list and the indices in equation (3.1) work automatically. Finally, we apply the activation function in line 21 (instantiating the node this time), and the layer is ready. The network's output is simply that of the last layer.

As a side note, line 15 defines something which has not been explained yet: a node scope. It could be omitted, but including it has to benefits: first, it adds the prefix `l<n + 1>/` to all nodes created under it. For instance, by giving the weight matrix the name `w` (line 17), we are actually creating a matrix named `l1/w` (or whichever index corresponds). Secondly, it groups the nodes defined under it from a logical point of view. This tells TensorBoard (a powerful visualization tool for TensorFlow graphs) to display them as a single higher-level node (which we can peek inside to see the inner nodes), so we can view the structure of our network from a layer perspective. Once the core network structure is ready, we define the nodes involved in training:

```
22 y_ = tf.placeholder(tf.float32, shape=[n_classes, None], name='y_')
23 output = op_outputs[-1]
24 with tf.name_scope('loss/'):
25     loss = -tf.reduce_mean(tf.reduce_sum(y_ * tf.log(output) + (1 -
           y_) * tf.log(1 - output), reduction_indices=0))
26     if st.L2_PARAM > 0:
27         for w in op_weights[1:]:
28             loss += st.L2_PARAM / 2 * tf.reduce_sum(tf.square(w))
29
30 with tf.name_scope('accuracy/'):
31     accuracy = tf.reduce_mean(tf.cast(tf.equal(tf.argmax(y_, 0),
           tf.argmax(output, 0)), tf.float32))
32 global_step = tf.Variable(0, name='global_step', trainable=False)
33 train_op =
       tf.train.GradientDescentOptimizer(st.LEARNING_RATE).minimize(loss,
       global_step= global_step)
34 init = tf.global_variables_initializer()
```

Line 22 creates the placeholder for the target value (again, the shape indicates we may have any number of columns). Then that target and the output are combined together in the loss

[6]Normal distributions are a common choice to initialise weights and biases in a network. In the case of weights, a non-standard normal is sometimes used, with $\sigma = 1/\sqrt{n}$ instead of 1, where $n$ is the number of units in the layer. This reduces the variance of the input to units in the next layer, which avoids high values that could saturate them from the start.

function, which is a mean of the cross-entropy cost (equation 2.3) over all the examples. The loop in line 27 adds the L2 regularisation term (sum of squares of all weights) if the regularisation parameter $\lambda$ is greater than 0. We then add an ***op*** to calculate accuracy (line 31), checking whether the index of the highest value in the output column (the class predicted by our network) corresponds to the index of the target class. This boolean value is cast into a float and averaged over all examples. When the loss function has been included in the graph, we add a ***variable*** to keep track of the training step (line 32), and two essential ***ops***. The first one, in line 33, is the optimiser: basic SGD with a constant learning rate retrieved from the settings file. As explained in subsection 3.1.3, the call to `minimize` adds the necessary nodes to calculate gradients using backpropagation and apply them to the trainable ***variables***.The other basic ***op*** is the ***variable*** initialisation. It should be note that both of these methods return a `tf.Operation` that we simply execute with `tf.Session.run`, and not ***tensors*** we need to `tf.Tensor.eval`.

The following piece of code sets up the necessary nodes and O.S. resources for TensorBoard to function properly (which shall not be discussed here) and that concludes our graph definition, after which comes the execution:

```
35 sess = tf.Session()
36 with sess.as_default():
37     inputs_v, targets_v = ds.get_validation_data(transpose=True)
38     inputs_te, targets_te = ds.get_test_data(transpose=True)
39     init.run()
40     step = 0
41     n_unimproved_steps = 0
42     best_loss_v = float('inf')
43     t_start = time.clock()
```

We begin by starting a session in line 35. Since no graph is passed, TensorFlow uses the default graph, which is the one we had defined our network in. We then enter a session context manager (line 36), an optional but convenient line which spares us from having to call `tf.Session.close()` and allows us to run `tf.Tensor.eval` and `tf.Operation.run` instead of `tf.Session.run([<tensors_and_ops>])` when all we want is the result of one tensor or the execution of one operation. We then load the validation and test data from our Dataset module (not the training data, since it will be read in mini-batches during the training loop). It needs to be transposed due to the fact that we need targets as columns (mathematical convenience) and the module returns them as rows (logical convenience[7]). After that, we initialise the ***variables*** by running the corresponding operation (as explained in subsection 3.1.3, this executes multiple `tf.assign` nodes). Lines 41 and 42 set up the early stopping method: stop when the validation loss does not decrease for $N$ consecutive steps. The number of steps without improvement is kept in the first ***variable***, and the second one stores the best loss in order to determine when there is an improvement. Lastly, we start the timer so training can begin.

```
44     while n_unimproved_steps < st.STEPS:
45     # while global_step.eval() < 5000:
46         inputs_t, targets_t = ds.get_minibatch(st.MINI_BATCH_SIZE,
47             autoreset=True, transpose=True)
48         t, loss_t, acc_t, summary = sess.run(
49             [train_op, loss, accuracy, summary_op],
50             feed_dict={input_: inputs_t, y_: targets_t})
51         step = global_step.eval()
52         t_summary_writer.add_summary(summary, step)
53         loss_v, acc_v, summary = sess.run([loss, accuracy, summary_op],
54             feed_dict={input_: inputs_v, y_: targets_v})
55         v_summary_writer.add_summary(summary, step)
56         if loss_v < best_loss_v:
57             best_loss_v = loss_v
58             n_unimproved_steps = 0
```

[7]Our data matrix is a *numpy* array, and the length of these objects (which we can retrieve with python's built-in `len`) is defined as their number of rows (or, in general, first-index elements). It is most natural that the length of a dataset is the number of examples in it, and this is why the Dataset modules returns examples as rows rather than columns

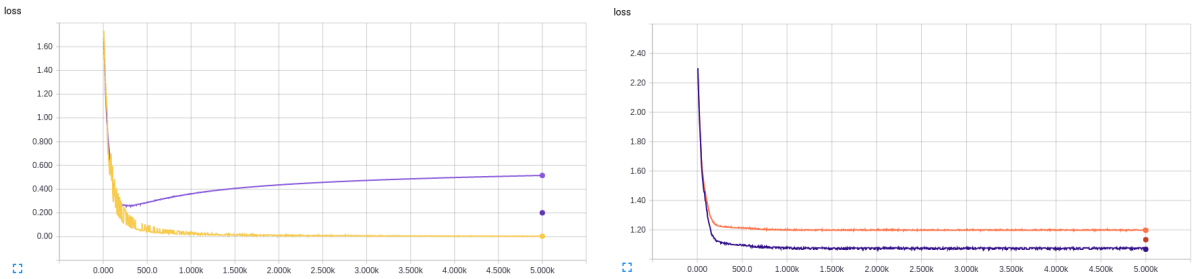



**Figure 3.3:** Loss over time. Left: no regularisation (yellow: training; light purple: validation; dark purple dot: test). Right: L2 regularisation with $\lambda = 0.01$ (purple: training; light red: validation; dark red dot: test). As we can see, overfitting starts to appear around step no. 250 in the figure on the left, so the validation loss begins to increase while the training loss keeps on decreasing. This is solved by regularisation, as we can see on the right.

```
59            else:
60                n_unimproved_steps += 1
61
62        t_end = time.clock()
63        loss_te, acc_te, summary = sess.run([loss, accuracy, summary_op],
64            feed_dict={input_: inputs_te, y_: targets_te})
65        te_summary_writer.add_summary(summary, step)
66
67    sess.close()
```

The stopping criterion is reflected in line 44. As an alternative for testing purposes, we can comment that line out and uncomment the next one to run a fixed number of training steps. During each training iteration, we start by loading a mini-batch of 30 training examples in line 46. We then run the training step itself (lines 48-50), passing a feed with the training inputs and targets. This applies gradient descent and updates the weights and biases, so that line 53 can evaluate the accuracy of the new network configuration on the validation data. Lines 56-60 update the information for the early stopping method. When the loop is exited, lines 62-65 gather some data for the user (e.g. accuracy on the test set). The rest of the program, which is not shown here, prints the results and launches TensorBoard if the user wishes to see more in-depth statistics (for instance, lines 52 and 55 add those statistics to the TensorBoard logs). We finish by closing the session in line 67, which is necessary even though we were inside a `sess.as_default()` context manager.

Annex D describes how to execute the script and browse TensorBoard. The annex also shows the graph corresponding to our network. Some executions are already stored in the log folder `tensorboard_logs`, which is automatically loaded by our program. This folder also contains the `key.txt` file explaining what each of the executions corresponds to (parameters, stopping criteria, etc).

It is now time to discuss some results from our Iris program. In the executions presented below, we trained a network with two hidden layers with 4 and 3 units respectively, and the dataset was always split as 60-20-20% (90, 30 and 30 examples respectively). The training method was SGD with a mini-batch size of 30 examples, and the activation function is the logistic sigmoid for all units (the network is shallow enough that we do not need to worry about vanishing gradients, which would make us lean towards ReLUs).

In our first (and naive) attempt, we train the network for 5000 fixed iterations with no regularisation techniques and a learning rate of 1. As we could expect, the network overfits the training data and the final accuracies are 100% on the training set, 96.67% on the validation set and 93.33% on the test set. If we plot the loss over time, we get figure 3.3 (left), which is very reminiscent of figure 2.6,

When we apply L2 regularisation, overfitting is drastically mitigated. The chosen parameter

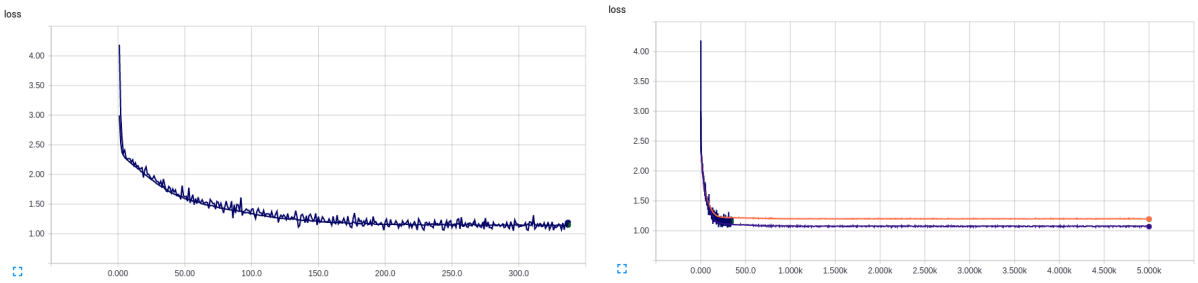



**Figure 3.4:** Loss over time. Left: early stopping after 50 steps of unimproved loss. Right: comparison of this with the previous situation (5000 iterations with L2 regularisation). As can be appreciated, the final loss is very similar in both cases. However, training takes less than 1/10 as long when early stopping is applied.

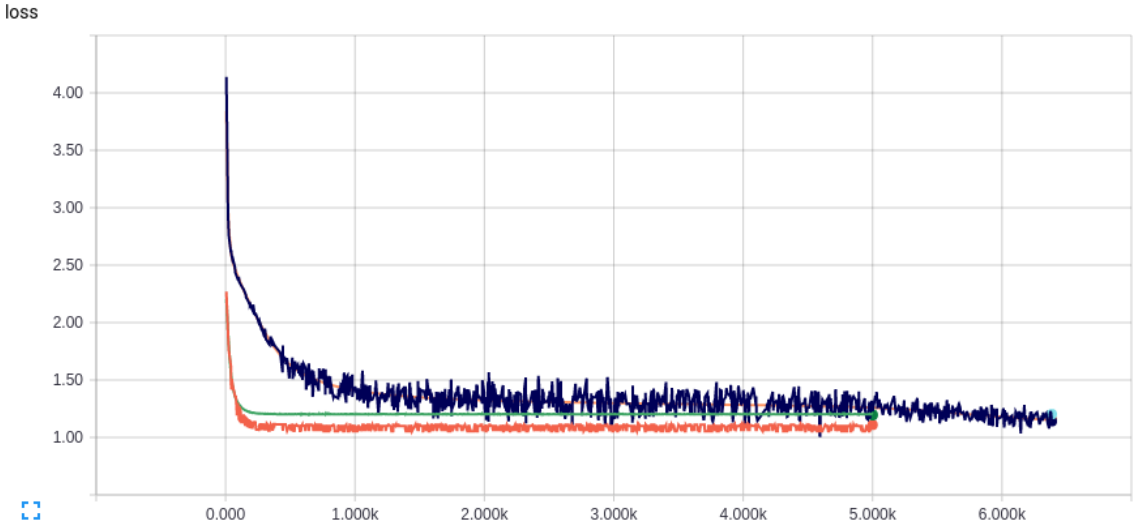



**Figure 3.5:** Loss over time. Top lines (blue: training and red behind it: validation): learning rate reduced to 0.1 with early stopping; Bottom lines (darker red: training and green: validation): learning rate of 1 for 5000 fixed steps. As we can see, the training time with a small learning rate is even longer than 5000 steps, and no significant improvement in the loss is achieved.

after several attempts is 0.01. The result can be seen in figure 3.3 (right): even though the training loss is still lower than the validation one, they stabilise after some time and keep a nearly constant difference. It should also be noted that the fact that the values of the loss are higher in this new execution does not mean it is actually performing worse: we have added a positive term with the sum of squares of weights. It has, in fact, the same accuracy on the test set (93%).

In our next attempt, we introduce early stopping as presented in subsection 2.5.1 (regularisation is still applied). The chosen criterion is: stop training when the loss does not decrease for 50 consecutive steps. The value of 50 has been tuned by trial and error: too large and it will not be actual "early" stopping. Too small and it will be too sensitive to flatter areas of our function, triggering before actually reaching a minimum. With this set-up, we obtain the results in figure 3.4. As mentioned earlier, the use of early stopping nowadays is to avoid training for much longer than necessary rather than to prevent overfitting.

Our last two tests relate to the effect of increasing or decreasing the learning rate $\alpha$ (early stopping and regularisation are still applied in both cases). We start by dividing it by 10, from 1 to 0.1. The result can be see in in figure 3.5: taking too small steps leads to a longer training

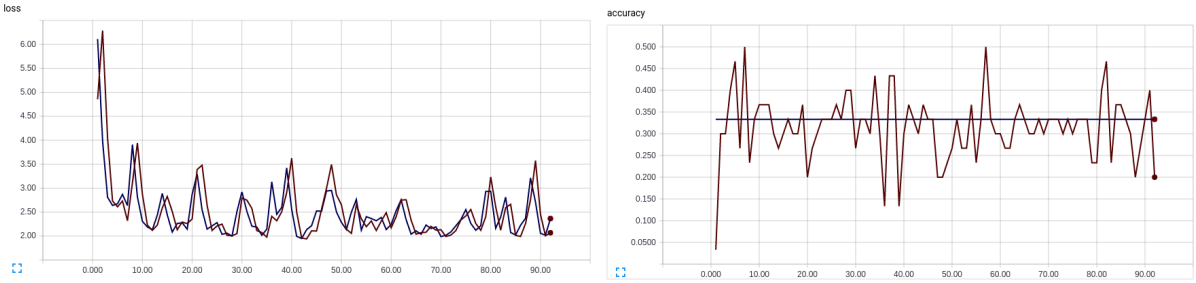



**Figure 3.6:** Loss (left) and accuracy (right) over time when the learning rate is increased to 10. Red: validation set, blue: training set. As we can see, the plots are a lot less smooth than with a small learning rate, and the values oscillate instead of converging.

time without an improvement in performance.

The final accuracy in this case was 96.67% for all three datasets. It may be tempting to think this "regularity" in the accuracies is a consequence of the reduction in the learning rate, but this is not the case. Executions under the same conditions (with different initialisations and dataset splitting) gave, for instance, a result of 90%-100%-90%. We may also wonder what happens when we multiply the original learning rate by 10, which can be thought of as 'rushed learning'. The results are in figure 3.6: the loss and accuracy vary sharply and do not reach any minimal value (in particular, the accuracy does not exceed 50%). What is most likely happening is that the gradient points towards a minimum, but the step taken is too large and our parameters move past the minimum. When we take more steps, we move around that minimum but never actually getting close enough. In other words, it becomes impossible to converge.

These last two examples highlight the awkwardness of fixed learning rates. Many gradient-based training algorithms implement an adaptive learning rate (often on a per-variable basis), such as Adam (published in a paper from 2015[11]). TensorFlow includes Adam and more as built-in optimisers. The adjustment of network elements such as the number of layers, number of units in each layer, learning rate and regularisation parameter is known as *hyperparameter tuning*, and it shall be discussed in chapter 4.

# 4

# Using TensorFlow: Handwritten Digit Recognition

In this last chapter we shall apply TensorFlow to a problem within one of today's real-world uses of machine learning: image processing. We start with an overview of a few high-level wrappers that make working with TensorFlow much easier. We focus on **Slim**, a lightweight library that will facilitate defining network architectures in the final section. After that comes an introduction to the **MNIST** problem of classifying handwritten digits, as well as some background about convolutional neural networks, the basic tool in neural image recognition. This discussion will also include an explanation of *hyperparameter tuning*, the process of finding the right network features for a given problem (and the computational environment in which this was done). We close the chapter by tackling the **MNIST** problem with TensorFlow, featuring the use of three different networks to solve it and commenting the results.

## 4.1 High-level TensorFlow wrappers

The core of TensorFlow described in chapter 3 can become tedious when designing and training large neural networks. Furthermore, it does not feature any utilities for handling convnets (which shall be described in subsection 4.2.2), meaning a lot of low-level work would need to be done in order to create and apply the filters, pooling, and dealing with the three-dimensional peculiarities of these networks. Luckily, several higher-level APIs exist that make this task (and many others) a lot more convenient, of which we shall discuss three. Two of them are external (*Keras* and *TFLearn*) and the other one is included in TensorFlow itself (*Slim*). Here is a quick overview of each of them:

- **Keras**[1] is in fact a machine learning library for python released in 2016. It was originally a wrapper for *Theano*, a powerful symbolic and numerical computation library. However, the possibility to use TensorFlow as the backend instead of *Theano* was introduced later on, and it is therefore reasonable, although not entirely correct, to accept Keras as a high-level API for TensorFlow. Keras focuses on deep learning capabilities, offering built-in layers, optimisers, tracking and logging mechanisms and image and text manipulation capabilities. Despite its recent creation, its use is widespread and well-established nowadays.
- **TFLearn**[2] is an independent high-level API for TensorFlow. It is an open-source project released under the MIT License and, like Keras, it is very recent (the first release took

[1] `https://keras.io/`.
[2] `http://tflearn.org/`.

place in August 2016). TFLearn contains a large amount of resources, including statistical tools, optimisers, metrics, and multiple types of artificial neural networks (for instance convolutional, recurrent and generative), besides hiding some of the 'dirty' visualisation set-up required by TensorBoard. Although not as popular as Keras yet, TFLearn has a chance of becoming the standard wrapper for TensorFlow.

- **Slim**[3] is part of the TensorFlow project (`tf.contrib.slim`). It is a lightweight wrapper for core TensorFlow created by Google employees Sergio Guadarrama and Nathan Silberman, and its release was given a special attention. It contains the most typical components present in high-level APIs (layers, losses, metrics, etc.) as well as more specific ones, such as argument scopes and queue runners. Argument scopes make it much more convenient to code network architectures by defining sections of code in which certain arguments (such as layer parameters and variable initialisers) take a predefined value instead of having to define them in each individual function call. We shall be using *Slim* in the last section of this chapter, specifically its layer types and argument scopes.

  Slim lies inside the **Contrib**[4] submodule, a large collection of utilities within TensorFlow with multiple projects created by various authors. Any programmer can issue a pull request and (if accepted) start adding new features. Due to the collaborative and changing nature of *contrib*, the contributions are not included into standard TensorFlow immediately, but rather kept as a side utility included in the installation. Quoting the documentation:

  *"Any code in this directory is not officially supported, and may change or be removed at any time without notice. The contrib directory contains project directories, each of which has designated owners. It is meant to contain features and contributions that eventually should get merged into core TensorFlow, but whose interfaces may still change, or which require some testing to see whether they can find broader acceptance."*

  *Contrib* contains no fewer than 47 projects at the time of writing, including layers (`contrib.layers`), metrics (`contrib.metrics`) and even more advanced tools such as recurrent neural networks[5] (`contrib.rnn`). Some of the most used packages have in fact made it into the official TensorFlow documentation[6], although not always with the amount of detail that could be expected. In spite of lacking stability, *contrib* is a very useful resource, many of whose tools will converge into standard TensorFlow sooner rather than later.

## 4.2 MNIST preparation

This section contains some necessary explanations to understand the last section (solving the problem itself). First comes a description of MNIST including some history and background on the problem. We then describe convolutional neural networks and how they are used to face the challenge of image recognition. This is a theoretical and rather brief introduction to the topic, only giving an overview on the topic, which will nevertheless suffice to understand the rest of the project. The section ends with a quick motivation of hyperparameter tuning and its computational implications.

[3] `https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim`.
[4] `https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib`
[5] These networks, unlike feed-forward ones, contain loops in their connections. Although research into RNNs is not as mature as its feed-forward counterpart, useful applications in time series predictions have been found.
[6] `https://www.tensorflow.org/api_docs/python/`.

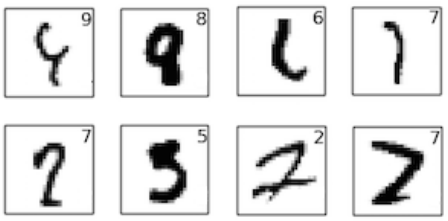


**Figure 4.1:** Some dubious images in the MNIST test set. The number in the top-right corner is the target class. Even these are correctly predicted by some of the top convnets. Image extracted from Nielsen's book [2].
.

### 4.2.1 The MNIST problem

The **MNIST**[7] **dataset** is a large public collection of images of handwritten digits. It was gathered in 1998 by **Yann LeCun**, director of AI Research at Facebook and one of the founding fathers of neural image recognition. The dataset was created as follows: 30 000 images were taken from the larger NIST *Special Database 1* (SD-1), and another 30 000 from the *Special Database 3* (SD-3). These are large databases of handwritten digits by university students (SD-1) and employees at the United States Census Bureau (SD-3). The images collected this way make up the training set. The same was done in order to create a 60 000-image test set, of which only 10 000 became the standard test set eventually (5 000 from each of the databases). Digits by around 250 different authors were included in the training set, and special attention was put into choosing different authors for the training and test sets in order to create true independence between them. Lastly, the pictures were processed with anti-aliasing since the original database images contained only white and black pixels, producing (softer) greyscale images. Additionally, they were centred using their centre of mass[8] and placed in a $28 \times 28$ canvas, yielding the final MNIST dataset. Many researches use a technique called dataset expansion, which aims to increase the size of the training set by applying various transformations. Most typically when dealing with image datasets, this involves rotations and affine transformations (displacements and elastic distortions). While this is a perfectly valid (and useful) preprocessing stage for image recognition, the resulting images are not part of MNIST.

MNIST is a classic dataset in machine learning research, and arguably the most common for testing character recognition despite the fact that it contains digits only. It is one of the largest too, surpassed only by *Chars74k*, which (not surprisingly) contains 74 000 images of handwritten characters. The problem of classifying MNIST digits has been long studied and attempted using various ML techniques, from Support Vector Machines (SVMs) and K-Nearest Neighbours (KNN) to different types of artificial neural networks. The first experiments were carried out by LeCun himself in 1998. Using a linear classifier (essentially a feed-forward network with no hidden layers), he obtained a 12% error rate (misclassified images in the test set). When using convolutional networks, the error dropped to 1.7%, and to 0.8% later that year. A great amount of research and diverse approaches have taken place since then, with outstanding achievements. Rodrigo Beneson keeps a repository of state-of-the-art results for this and other popular ML problems[9]. Only for reference, we shall introduce the record holder for MNIST. The best error rate to date is 0.21%, achieved by a group of researchers from the New York University and presented at ICML[10] 2013. This means only 21 of the 10 000 test images were

[7] Mixed National Institute of Standards and Technology (NIST)

[8] This essentially means calculating the average position of all coloured pixels and translating the image so it lies in the centre.

[9] `http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html`.

[10] International Conference on Machine Learning

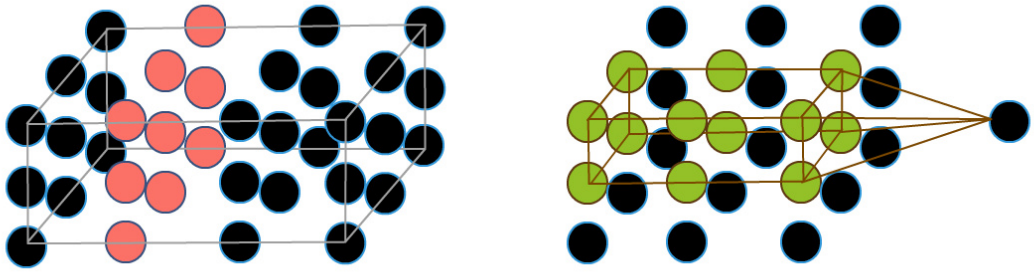


**Figure 4.2:** Two unit volumes in a convnet. The first one has 36 units arranged in 4 $3 \times 3$ feature maps, one of which is highlighted in red, and its dimensions are $4 \times 3 \times 3$ (depth $\times$ height $\times$ width). The second one has dimensions $3 \times 3 \times 3$, and the neurons highlighted in green form the $3 \times 2 \times 2$ local receptive field (LRF) of a neuron in the next layer.

sorted incorrectly, which is very surprising when we see some of the examples in figure 4.1: the system would outperform most humans. The network in question uses convolutional layers but adds a new technique for the final fully-connected stage: *DropConnect*. This is a generalisation of dropout in which individual weights are dropped (set to 0) instead of full outputs. It should be noted that dropping the output of a unit is equivalent to dropping all the weights from that unit to the next layer simultaneously, and therefore *DropConnect* is more general and powerful than traditional dropout.

### 4.2.2 Convolutional Neural Networks

**Convolutional neural networks** (also **CNNs** or **ConvNets**) are a type of feed-forward neural networks specifically designed for image processing. They use the spatial properties of pixels and apply a series of filters and other layers that reduce the complexity of the input and specialise in recognising particular image features. ConvNets are a very good example of how the layers of deep networks acquire a hierarchy of levels of abstraction. Dating back to the 1980s, they have become today's standard for machine-learning-based image recognition. This section acts as a crash course on this type of layer.

In order to understand convnets and their motivation, we must first analyse their input. Each image is represented as a series of overlapping layers of $l \times l$ pixels[11]. Each of those layers, called a *channel*, contains the normalised intensity value (typically in $[0, 1]$ or $[-1, 1]$) of a colour in each pixel. Images often have either one channel, if they are in black and white (for instance the MNIST dataset, where each image is a $28 \times 28 \times 1$ array) or three channels (RGB), if they are in colour (as in the CIFAR-10 dataset, where the dimensions are $32 \times 32 \times 3$).

The numbers involved here may help us understand the motivation of convnets: if we were to use the fully-connected approach of traditional feed-forward networks on the CIFAR problem, for instance, a single neuron in the first hidden layer would have $32 \cdot 32 \cdot 3 = 3072$ weights. This number escalates very quickly with the amount of units and layers necessary to do any sort of processing, and would soon become unmanageable to train. Furthermore, the spatial properties of the image (proximity, for instance) would be lost, since that neuron would simply add the incoming values from the previous layer regardless of positions. Clearly a different approach is needed.

[11] We shall only consider square images in the present discussion. This is the case for many of the most common image datasets, such as MNIST (`http://yann.lecun.com/exdb/mnist/`) and CIFAR (`https://www.cs.toronto.edu/~kriz/cifar.html`). The concepts explained here can be easily extended to non-square patterns.

**Convolutional layers** are the key piece of CNNs. They take an input volume[12] and apply a series of filters (also known as *kernels* or *convolutions*) to each of its layers, producing a new output volume. They do so by 'looking' at a small portion the input volume at a time, called the *local receptive field* (LRF), which consists of a square region of the input across all of its layers. These concepts can be visualised in figure 4.2. Each filter slides over the whole image, looking for a certain feature in each position. Filters are applied by simply multiplying the values of the pixels in the LRF by the corresponding weights and adding a bias. If there are several filters, each filter is applied individually, and the two-dimensional sections produced by all the filters and are stacked together into a 3D output volume. These vertical sections of the volumes are called *feature maps*, since they produce an output depending on whether the input has a certain feature or not. Note that this output volume can be processed by another convolutional layer under the same principles later on, despite the fact that it is not a natural image.

Each filter in a convolutional layer has a width and a height, which correspond to the dimensions of the local receptive field; and a stride, which is the amount of pixels the filter 'jumps' each time it moves to the next position in the image. Typical values for these hyperparameters are $3 \times 3$ and $5 \times 5$, with stride 1, 2 or sometimes 3. Occasionally, an additional frame of zeros, known as *zero-padding*, is created around the image to solve problems of dimension or in order to preserve the input size. This also helps maintain the information at the edges, which would otherwise be progressively lost since those pixels are never in the centre of an LRF.

In terms of artificial neural networks, the convolutional layer is actually a volume of units. Each vertical section of the volume corresponds to a filter. Within each of those sections, each neuron applies that filter to a different LRF in the input. What this means is that the weights and biases are **shared** by all neurons in each feature map.

As an example, suppose we are designing a neural network for the CIFAR problem, which, as mentioned above, consists of images with $32 \times 32$ pixels and 3 colour channels. The first convolutional layer could have 6 filters with a local receptive field of $5 \times 5$ and a stride of 3. It will therefore contain $6 \cdot 5 \cdot 5 \cdot 3 = \mathbf{450\ weights}$[13] (6 filters, each applying a $5 \times 5$ LRF to each of the 3 channels), which is a much more manageable amount than 3072 per neuron. When each of the filters is slid over the padding-less $32 \times 32$ image, it is easy to check it moves over 10 positions in both height and width, so the output volume has dimensions $10 \times 10 \times 6$. More generally:

- A convolutional layer consisting of $n$ filters with $f \times f$ LRF over $c$ input feature maps contains $w$ weights, where
$$w = n \cdot f^2 \cdot c\ . \tag{4.1}$$
- When that convolution is applied to an $l \times l$ image with a stride of $s$, the output volume has $n \times N \times N$ units, where
$$N = \frac{l - f}{s} + 1\ . \tag{4.2}$$

Notice that $f$, $n$, $c$ and $s$ are all hyperparameters (plus the zero-padding, not included here).

The idea behind filters is that each of them learns to detect a particular feature. For example, a filter which reads the input image may recognise 'green horizontal lines', whereas another one will identify 'isolated red dots'. A filter in a deeper layer, which receives an abstraction of the information (e.g., a series of layers corresponding to the presence of lines and dots) may learn

[12] A volume is a three-dimensional array (also known as a *tensor* of units or, equivalently, outputs). This is an important difference with respect to the feed-forward networks studied in chapter 2, where the layers of units were thought of as one-dimensional columns. We refer to the number of vertical sections as its depth, and the dimensions of each of these sections as its height and width.

[13] Since weights represent such a large fraction of the network parameters (which would include biases too), it is customary to consider only the weights when discussing the network complexity. The total number of parameters in this case would be $6 \cdot (5 \cdot 5 \cdot 3) = 450$

to detect 'two isolated red dots over a green horizontal line', i.e., 'a monster's face'. This type of abstraction hierarchy extends throughout the whole network and allows it to transform the low-level array of pixels into a higher-level concept, such as a certain character, digit or object.

Convolutional layers are often followed by a **pooling layer**. Pooling layers aim to decrease the amount of parameters in the network, thus simplifying training and reducing overfitting. Each pooling layer reduces the size of its input volume by looking at groups of contiguous pixels (essentially, a local receptive field) and producing a single output value for each group. The way that value is computed determines the type of pooling being applied, and there are two widespread choices: *max pooling*, where the pixels in each LRF are replaced by the maximum value among them, and *average pooling*, where they are replaced by the average value in the group instead. The former has become the go-to option, since it has been shown to work better in practice. It is nevertheless questioned, with many researches advocating for fully-convolutional networks using larger stride values to reduce the volume size [12].

Pooling is applied to each vertical section of the input volume independently, and therefore does not change its depth. Most often, this operation is applied to $2 \times 2$ LRFs with a stride of 2 (i.e., no overlapping), meaning the size of the input volume is reduced to 1/4 of the original. This type of layer requires no parameters whatsoever. It is a common practice to use convolutional layers with stride equal to 1 and zero-padding so that they preserve the volume size, leaving all the downsampling to the pooling layers. A more in-depth explanation of these concepts can be found in [13].

Convolutional and pooling layers extract the necessary features in the image and produce an abstracted summary of them. After any number of them have been stacked, the rest of the processing is done using fully-connected layers like the ones described in chapter 2. The last convolutional or pooling layer (which is a 3D volume of unit) is connected to a one-dimensional layer on a neuron-to-neuron basis[14]. When doing so, the spatial structure of the image is lost, so we need to rely on the fact that the previous layers have extracted all the information in it. For instance, it may not be enough to know that 'there is a person' and 'there is a car' in order to detect dangerous situations. We need to know whether the two of them are far apart or not, which resides in the 2D structure of the input, or the convolutional section of our network.

After the flattened 3D-to-1D connections have been made, everything works as in the fully-connected networks discussed in chapter 2. The activation functions and the weights and biases are handled the same way, often ending in an output layer with membership probabilities to different classes. It should be noted that ReLU activations are sometimes considered as part of the convolutional or fully-connected layers, and sometimes considered as layers in their own right. In the latter case, they are represented as a volume of units with the exact same size as the input. However, we shall use the former approach (the ReLU activation, if used, is part of a layer).

With such specific types of layers, designing a convolutional neural network is often a matter of selecting a sequence of them and choosing the necessary hyperparameters for the convolutional and fully-connected portions. There are a number of well-established architectures, each of which performs best on certain problems. We shall introduce some of these architectures along this chapter. These architectures are often represented as a sequence of names of layers, with some flexibility in the notation. For instance,

$$2 \times \texttt{[CONV} \rightarrow \texttt{POOL]} \rightarrow 2 \times \texttt{FC}$$

means there is a convolutional layer followed by a pooling one, then another convolutional and another pooling layer, and finally two fully-connected ones, which connect to the output layer. This is the structure of the *LeNet* network featured later in this document.

As a side note, it can be guessed from the description above that the change from backpropagation in traditional fully-connected feed-forward networks (as discussed in chapter 2) to backpropagation in convnets is not entirely effortless. Although the mathematical principles ap-

[14] This is known as 'flattening' in most programming environments.

plied are the same, shared weights, local receptive fields and three-dimensional volumes require a slightly different treatment. Luckily, TensorFlow and most other machine learning libraries abstract this process for us, so we need not worry about the details. They will not be discussed here, but an interesting description can be found in [14].

### 4.2.3 Tuning and running

When facing the design of a neural network to solve a particular problem, one of the first challenges to arise is finding the right hyperparameters (before training itself can even start): the amount of layers, their size, the learning rate, the L2 regularisation parameter, etc. This process, known as **hyperparameter tuning**, is one of the most fundamental aspects of working with neural networks. In the early days of the field, it was done in an arbitrary way, with a mixture of trial and error, experience and 'black magic'. However, as the science of neural networks developed, this tuning process acquired a much more systematic approach, with even some libraries created specifically for that purpose, such as the recent Optunity[15]. A natural way to perform hyperparameter tuning is the following, known as grid search:

1. Decide which hyperparameters $p_1, \ldots, p_n$ must be tuned. Some others we may want to leave as already decided, such as the network structure.
2. For each parameter $p_i$ select a few values $v_i^1, \ldots, v_i^{m_i}$ among (or close to) which the optimal value could be.
3. Train a network (or several) with each combination $(v_1^{j_1}, \ldots, v_n^{j_n})$ of those values, and keep the hyperparameter setting that yields the best result. We can think of this as exploring a grid of $n$ dimensions with the values of each parameter in one of the orthogonal axes, which explains the name of the method.
4. Once a good combination has been found as above, the process can be started over from step number 2 with a finer set of values around the ones found.

Grid search, which shall be used to tune the architectures in this chapter, is a very systematic technique. However, as could be expected, it is very time-consuming: the amount of values in the grid grows very quickly. In particular, it grows *exponentially* with the number of hyperparameters to tune (it is the product $m_1 \cdot \ldots \cdot m_n$). Training a single convolutional neural network like the ones presented below is a matter of minutes or a few hours. This means training thousands of them in order to find a good set of hyperparameters has limited viability. For that reason, we shall restrict our hyperparameter tuning to L2 regularisation and dropout rate.

Since working with deep networks requires considerable amounts of time and sometimes computer memory (when working with large image datasets, for instance), it is often done in dedicated data centres. The workload involved in tuning and training the networks in this chapter was done at UAM's *Scientific Computing Centre* (CCC), to whom we are deeply grateful. The set-up involved multiple aspects: a delicate TensorFlow installation, system libraries (`glibc`), user permissions and groups, SSH, VPN and automation scripts. Once that was completed, we were able to use CCC's queue system, which is handled via the *Slurm* workload manager. Essentially, the user connects to the CCC servers remotely (via SSH) and submits batch jobs. From that moment, shell interactivity is lost and the console output is redirected to a file. These jobs may take several days to execute with no need of an active connection or supervision, which is perfect for our hyperparameter tuning needs. Although it took some time to get used to and a special workflow (planning, sending jobs into a black box, checking their status, etc.), the CCC environment was essential for this part of the project, which could not have been performed on a personal computer (for instance, launching `tuner.py` on our usual laptop leads to a very quick crash due to lack of memory).

[15] `http://optunity.readthedocs.io/`

## 4.3 Tackling MNIST with TensorFlow

In this section we shall address the MNIST problem using convolutional neural networks and TensorFlow. The classification accuracy of three different convnets will be compared, as well as the training required by each of them. As can be expected, smaller networks[16] can be trained much more quickly than large ones, but they may not have enough complexity to capture the problem logic. There is thus sometimes a delicate trade-off between accuracy and training time.

The three architectures which shall be defined, tuned and compared are:

- Yan LeCunn's original *Le-Net 5*, as described in his foundational paper from 1998 [13].
- A simple convolutional neural network described in one of the TensorFlow tutorials[17] Notice that the only element taken from there is the network architecture. The programs are completely different due to our use of TFSlim and particular needs.
- An improvement on the original *Le-Net 5* proposed by Vishnu Sundaresan, from Stanford University [15].

These networks shall be identified as *lenet*, *tfnet* and *vishnet* respectively. Although their structure varies, some elements have been kept the same for all of them. For instance, the output layer consists of 10 units with ReLU or no activations to which softmax is applied, and the loss function is categorical cross-entropy. The original MNIST training set (60,000 images) is split into a 55,000-image training set and a 5,000 image validation set for hyperparameter tuning (this is provided in `tensorflow.examples.tutorials.mnist`). The size of the minibatches is 50 and the number of training steps is 10,000 during grid search (which means each image is fed to the network about 10 times), and 15,000 for final training.

The programs involved in this subsection are:

- `architectures.py`: a small module containing several functions with a common interface that add the nodes needed by each convnet to the graph.
- `tuner.py`: this program performs grid search on a given architecture, saving a summary of the results and each of the networks trained (called 'models'). It is a shell for the structures defined in `architectures.py`.
- `runner.py`: this program loads a model (saved by `tuner.py`) and allows us to run nodes in. In particular, we can use it to check the accuracy or loss on the test set of a network we trained earlier. In order to present the final results, a more streamlined version of this file was created: `runner_aux.py`, which takes a list of model names, loads them and computes validation and test accuracy on each of them, averaging those values over groups of models.
- `settings.py`: since interactivity is lost when we submit a job to a queue, all the parameters for a particular execution (such as the ID we want to save the model with, the architecture we want to train, the hyperparameter grid, etc.) have to be saved in this file beforehand.

Those files, as well as the result data, can be found in:
`https://github.com/Antonio95/final-degree-project/tree/master/tensorflow_mnist`

### 4.3.1 Lenet

This convnet has 6 hidden layers and is the smallest of the networks we shall consider in terms of number of weights. It contains the following layers:

[16] Always in terms of weights, which relates to the number of intermediate layers, their type and size.
[17] `https://www.tensorflow.org/get_started/mnist/pros`

`l1:` `Input layer, 28x28x1`

`l2:` `Convolutional, 28x28x6`: This layer consists of 6 filters of $5 \times 5$, and no activation function. A padding of width 2 is applied to its input in order to preserve the dimensions. Weights in `l2`: $6 \cdot 5 \cdot 5 \cdot 1 = 150$ (see equation 4.1).

`l3:` `Max pooling, 14x14x6`: Max pooling with stride 2. Weights in `l3`: 0.

`l4:` `Convolutional, 10x10x16`: 16 filters of 5 by 5 and no activation function. In this case no padding is applied, so the vertical section size is reduced from $14 \times 14$ to $10 \times 10$. Weights in `l4`: $16 \cdot 5 \cdot 5 \cdot 6 = 2,400$.

`l5:` `Max pooling, 5x5x16`: Same characteristics as `l3` (in particular, no weights).

`l6:` `Fully connected, 120`: Flat layer (meaning we think of it as a one-dimensional structureless array of units) which is fully connected to `l5` (unit volume) and has sigmoid activations. The number of weights in a fully-connected layer is the product of the units in that layer and the previous one, which in this case is $(5 \times 5 \times 16) \times 120 = 48,000$.

`l7:` `Fully connected, 84`: Fully connected, with no activations and $120 \cdot 84 = 10,080$ weights.

`l8:` `Output layer, 10`: Finally, the output layer has 10 units (as many as classes in the MNIST problem). Weights in the output layer: $84 \cdot 10 = 840$.

The structure can be summarized as: `2` $\times$ `[CONV` $\rightarrow$ `POOL]` $\rightarrow$ `2` $\times$ `FC`
Total number of weights: 61,470.

In order to implement this as TensorFlow code, we have created a dedicated file, `tuner.py`, which creates some elements common to all of our network graphs, such as the input placeholder and the cross-entropy loss; and sets up the training and logging. This allows us to define the various architectures neatly in a separate file, `architectures.py`, which only contains functions which add to the graph the necessary nodes for each convnet. In the case of *lenet*, these few lines are enough to define the structure:

```
1  def tfnet(image, keep_prob, regularisation=0.001):
2      with slim.arg_scope([slim.conv2d, slim.fully_connected],
           weights_regularizer=slim.l2_regularizer(regularisation)):
3          l2 = slim.conv2d(image, 6, [5, 5], activation_fn=None,
               scope='conv_1')
4          l3 = slim.max_pool2d(l2, [2, 2], scope='maxpool_1')
5          l4 = slim.conv2d(l3, 16, [5, 5], padding='VALID',
               activation_fn=None, scope='conv_2')
6          l5 = slim.max_pool2d(l4, [2, 2], scope='maxpool_2')
7          l6 = slim.fully_connected(slim.flatten(l5), 120, scope='fc_1')
8          l7 = slim.fully_connected(l6, 84, activation_fn=None,
               scope='fc_2')
9          output = slim.fully_connected(l7, 10, activation_fn=None,
               scope='fc_3')
10     return output
```

As we can see, the function receives the image (input place holder) and adds architecture to the graph (as a black box for the caller), returning the output which is then passed to the loss function. Line 2 defines an *argument scope* or *argscope* (mentioned when describing TFSlim in section 4.1). What this does is to pass the constructors of the specified layers (the list in the first argument) within the scope, the desired parameter values. By creating this scope, we avoid having to write `weights_regularizer=slim.l2_regularizer(regularisation)` in most of the the following lines. It also prevents us from having to specify that we want Xavier initialisation[18] inside

[18]Xavier Glorot and Yosua Bengio proposed in 2010 [16] an improved method for initialising the weights of neural networks which aims to keep the signal at bounded values throughout many layers by changing the variance

the scope (we cannot see explicitly in the code because `weights_initializer=slim.xavier_initializer` is a default argument of the *argscope* constructor). We may also notice the argument `padding='VALID'` in line 5. TFSlim offers two possible values for the padding of convolutional and pooling layers. `'SAME'` (the default value) tells the TensorFlow to add as much zero padding as necessary to keep the dimensions of the (original) input volume the same after the transformation (if the number of necessary rows/columns is odd, they extra ones are added to the right and on the bottom); whereas `'VALID'` means no padding is applied (if the resulting layer is invalid, meaning the outcome of equation 4.2 is not an integer, enough columns are dropped from right to left to achieve so).

*lenet* has a relatively small structure: the filters are always $5 \times 5$ and the number of feature maps is quite limited (16 at a most, as opposed to, for instance, 256 in *vishnet*), which explains why this network is comparatively fast to train (about 9.5 minutes per individual execution). It should be noted that nearly 80% of the 61k weights are located in the first fully-connected layer. This is not an uncommon situation in convnets, since convolutional volumes often have a large number of units (which does not translate into a large number of weights in them due to the locality of filters). Additionally, no activation function is applied in the convolutional or output layers.

When tuning the hyperparameter[19] $\lambda$, we chose a relatively broad grid first. The values checked were:

$$\lambda \in \{0.001,\ 0.005,\ 0.01,\ 0.05,\ 0.1,\ 0.5,\ 1.0\}$$

The resulting validation accuracies after 10,000 iterations on the training set can be found in table E.1. The validation accuracy is our criterion to decide the best set of hyperparameters, since using the test set would lead to a subtle form of overfitting (as discussed in section 2.4). The best accuracy, 98.50%, is achieved when $\lambda = 0.001$. This presents the issue of having found the best value in one of the bounds of the interval for that parameter, meaning there could be more optimal values under it. For that reason, and since training a single *lenet* is relatively fast (about 9.5 minutes), we did a second exploration with finer values around 0.001:

$$\lambda \in \{0.0001,\ 0.0004,\ 0.0007,\ 0.001,\ 0.004,\ 0.007,\ 0.01\}$$

This yields our final choice: $\lambda = 0.0004$, which makes the validation accuracy 98.38%. It might seem surprising that this is lower than our result from the coarse grid search. However, it is only a proof of the random nature of neural networks compared to other learning mechanisms: accuracies vary even within executions with the same hyperparameters and training conditions. Over a number of executions, we can expect a better average accuracy than 98.38%. A way to mitigate the issue of variance are network ensembles, in which several networks are trained and averages are taken over accuracies and individual predictions.

We finally train a single *lenet* for 15,000 iterations with that choice for $\lambda$. The result is a convolutional neural network with 98,70% classification accuracy on the test set (130 misclasslified images out of 10,000), which is a very respectable rate for such a fast network.

### 4.3.2 Tfnet

This network is much larger than the previous one. As we shall see, it has about 50 times as many weights, which means training is a considerably heavier computation. Its structure is detailed now, including the number of weights in the layers this time (since *lenet* showed how to compute them):

of the normal standard distribution. In the case of TensorFlow Xavier, the chosen standard deviation is $\sqrt{\frac{3}{i+o}}$, where $i$ and $o$ are the number of the layer's input and output units respectively.

[19] In fact, *lenet* is the only convnet of the ones studied here which uses no dropout. In order to keep the uniformity in the interface of the `architectures.py` module, it still accepts a `keep_prob` parameter ($\rho$) which exists as a placeholder in the computation graph. For completion, we have swept several values of $\rho$, as can be seen in annex E, only to verify they do not affect the network's accuracy.

```
l1:  Input layer, 28x28x1.  Weights:  0.

l2:  Convolutional, 28x28x32.  Weights:  800.

l3:  Max pooling, 14x14x32.  Weights:  0.

l4:  Convolutional, 14x14x64.  51,200.

l5:  Max pooling, 7x7x64.  Weights:  0.

l6:  Fully connected, 1024.  Weights:  3,211,264.

l7:  Output layer, 10.  Weights:  10,240.
```

Structure summary: `2` $\times$ `[CONV` $\rightarrow$ `POOL]` $\rightarrow$ `FC`
Total number of weights: 3,273,504.

Surprisingly, this network has one fewer layer than *lenet*, since there is a hidden fully-connected layer rather than two. Despite that fact, we can easily identify the reason for the higher total weight count: the last convolutional layer has many feature maps (64 rather than 16) and the first fully-connected layer contains 1024 units. This connection accounts for 3,211,264 weights, which represents 98.09% of the total. Regarding the rest of the network details, *tfnet* is similar to *lenet*: the filters have size $5 \times 5$ and the output layer has no activation function (the cross-entropy is fed the output directly). However, *tfnet* applies a padding of 2 zeros (on each of the four sides of each feature map) around the layer `l3` so, as we can see, the convolutional layers preserve the size of each feature map ($14 \times 14 \rightarrow 14 \times 14$ rather than *lenet*'s $14 \times 14 \rightarrow 10 \times 10$). Additionally, dropout is applied to the large fully-connected layer `l6`. When tuning the hyper-parameters of *tfnet*, we chose the following values:

$$\lambda \in \{5 \cdot 10^{-6},\ 8 \cdot 10^{-6},\ 1 \cdot 10^{-5},\ 2 \cdot 10^{-5},\ 5 \cdot 10^{-5}\}$$

$$\rho \in \{0.25,\ 0.4,\ 0.5,\ 0.6,\ 0.75\}$$

The choice for the values of $\lambda$ is based on the results of our *lenet* exploration. The following *$\lambda$-rule of thumb* can be applied when training two similar networks (preferably with the same cost function and/or output layers) which work on the same problem: if we remember equation 2.11, performing L2 regularisation simply meant adding to the original cost function a scaled sum of squares of all the weights in the network. If our two networks $A$ and $B$ have $n_A$ and $n_B$ weights respectively, that sum of squares will be about $\frac{n_B}{n_A}$ times larger in the second case, assuming the values of the weights in both nets move in similar ranges (which they do at least at the beginning if we use the same initialisation method). What we are most concerned about is preserving the balance between the original cost function and the L2 term, which is precisely what $\lambda$ is used for. Since the cost functions of $A$ and $B$ have similar magnitude by assumption, this means we want the sums of squared weights to be similar in both networks, and since $B$'s sum has $\frac{n_B}{n_A}$ times as many terms as that of $A$, we want to scale it down by the inverse of that factor. A good starting value for the grid search will therefore be

$$\lambda_B \approx \frac{n_A}{n_B}\lambda_A,$$

if $\lambda_A$ has been shown to work in the first network. In this case, the result is

$$\lambda_{tfnet} \approx \frac{61470}{3211264}\lambda_{lenet} \simeq 0.019 \cdot 0.0004 = 7.6 \cdot 10^{-6},$$

which, with a bit of rounding up ($10^{-5}$), is in the middle of our grid values for $\lambda$.

The results of the grid search can be found in table E.3. In this case, we have two maxima for the validation accuracy at 99.30% corresponding to ($\lambda = 2 \cdot 10^{-5}, \rho = 0.6$) and ($\lambda = 5 \cdot 10^{-5}, \rho =$

0.75). Both these choices would be reasonable, but by looking at nearby grid points we find that the values closer to the latter seem ever so slightly better. For that reason, we keep the pair $(\lambda = 5 \cdot 10^{-5}, \rho = 0.75)$. Due to the size of the network, training a single instance takes about 1 hour and 12 minutes (7.5 times longer than *lenet*), so a finer grid search was not performed around the chosen values.

### 4.3.3 Vishnet

This network, proposed as an evolution of the *lenet*, contains more convolutions than the previous two:

```
l1:  Input layer, 28x28x1.  Weights:  0.

l2:  Convolutional, 28x28x64.  Weights:  1,600.

l3:  Max pooling, 14x14x64.  Weights:  0.

l4:  Convolutional, 14x14x128.  204,800.

l5:  Max pooling, 7x7x128.  Weights:  0.

l6:  Convolutional, 7x7x256.  Weights:  819,200.

l7:  Max pooling, 6x6x256.  Weights:  0.

l8:  Output layer, 10.  Weights:  92,160.
```

Structure summary: `3` $\times$ `[CONV` $\rightarrow$ `POOL]`
Total number of weights: 1,117,760.

There are certainly some odd features in this architecture. For instance, there are no hidden fully-connected layers, and it contains a pooling layer `l7` which takes the volume from $7 \times 7 \times 256$ to $6 \times 6 \times 256$. However, that is as much as could be understood from the paper [15] and, surprisingly, it yields better results than the previous networks. *vishnet* has about one-third as many weights as *tfnet* and they are arguably more spread across layers.

The filters in *vishnet* are again $5 \times 5$ with stride 1, and padding is always applied before convolutional layers in order to preserve sizes, dropout is used on layer `l7` and the activation function in the output layer is ReLU. Despite a lower number of weights than *tfnet*, this network takes considerably longer to train (about 3h per instance), the reason being that there are many more feature maps (256 in the last convolutional rather than 64) and one more convolutional layer. While we often think of weights as the main factor to determine training time, the actual size of the networks in terms of units also has an impact on it. Adding more filters causes the number of operations needed in order to propagate the information forward and backward to increase, and it sometimes does so without significantly increasing the number of weights: the weights are shared for all the neurons performing a filter, but the actual operations (products) have to be performed for every LRF in the input volume. This fact may explain why the *vishet*, with its 256 filters, takes longer to train than *tfnet* even though it has fewer weights.

When tuning this network, the chosen grid values for the hyperparameters were:

$$\lambda \in \{3 \cdot 10^{-7},\ 1 \cdot 10^{-6},\ 3 \cdot 10^{-6},\ 1 \cdot 10^{-5},\ 3 \cdot 10^{-5}\}$$

$$\rho \in \{0.25,\ 0.4,\ 0.5,\ 0.6,\ 0.75\}$$

We can see how the middle value for $\lambda$ is three times that of *tfnet*, which was chosen with the same rule of thumb (the number of weights is now $\sim 1M$ as opposed to the previous $\sim 3M$). While a rule of thumb can be devised for the dropout parameter $\rho$ as well, it is not a simple task: dropout is a subtler and more cryptic technique than L2 regularisation. The explored values for $\rho$ have therefore been kept the same. As can be seen in table E.4, the best accuracy is 99.44%, achieved with $(\lambda = 3 \cdot 10^{-7}, \rho = 0.4)$.

### 4.3.4 Results

After tuning the hyperparameters of the networks, we are ready to train a good number of them and interpret the results of the experiment. At this stage, we trained several instances of each architecture in order to reduce the intrinsic variability of the experiments. Specifically, four networks were trained in each case, and only the top 2 (in terms of validation accuracy) were kept to measure test accuracy. In the case of *lenet*, this does not make a big difference since most instances end up very close to each other after training. However, when we look at the results of *vishnet*, we find those pathological situations more often. For instance, when training it for 20 000 iterations, the validation accuracies [20] were: 99.30%, 99.36%, 89.64% and 88.54%. By dropping the two worst results in each case, we obtain more uniform results and a better assessment of the network's capabilities. The reason for the much higher variability of *vishnet* is twofold: firstly, the larger number of weights increases the dimensions of the parameter space and the amount of randomly initialised values. Secondly, the dropout technique (which is applied in *tfnet* and *vishnet*, but not *lenet*) adds a strongly random component to the training process.

The training of those networks was performed with the hyperparameter values found in the previous sections, and it was repeated for five amounts of training steps: 5 000, 10 000, 20 000, 40 000 and 60 000 for each architecture. This way we can study the balance between amount of training [21] and accuracy, and also find each architecture's optimal number of iterations. The following table summarises the outcome of the experiment:

| | 5 000 | 10 000 | 20 000 | 40 000 | 60 000 |
|---|---|---|---|---|---|
| lenet (0.064) | 97.87%<br>97.82% | 98.34%<br>98.32% | 98.79%<br>98.80% | 98.84%<br>98.85% | 98.88%<br>98.82% |
| tfnet (0.59) | 98.94%<br>98.92% | 99.20%<br>99.27% | 99.31%<br>99.27% | 99.34%<br>99.22% | 99.43%<br>99.36% |
| vishnet (1.75) | 99.12%<br>99.16% | 99.35%<br>99.33% | 99.33%<br>99.45% | 99.52%<br>99.48% | 99.52%<br>99.51% |

**Table 4.1:** Architecture performance comparison. The top index reflects the number of training steps, while the left one indicates the network and the average time per training step in seconds. Inside each cell, the values are the validation and test accuracies (both averaged over the best two instances).

These are some of the conclusions we can draw:

- Regarding the **generalisation capabilities** of each network (the difference between accuracy on the validation set and previously unseen data, i.e. the test set), *lenet* is at a fairly good level, but *tfnet* struggles somewhat with differences of over 0.1% in some cases. *Vishnet*, however, performs very well, with the test accuracy being always over the validation one, or very close below it. The worse result in the case of *tfnet* is is due primarily to the number of network parameters (50 times that of *lenet* and 3 times that of *vishnet*). Even with the use of L2 regularisation and dropout, more weights involve a greater chance of hints of overfitting showing up.

- Regarding overall **accuracy**, *vishnet* is the clear winner, beating *lenet* and *tfnet* on both the validation and the test set in every training setting. Only at 60 000 steps does *tfnet* come close on the validation set (99.43% vs. *vishnet*'s 99.52%), but *vishnet* increases its

[20] The output files of this final subsection can be found in the `results` folder as `<architecture>_<n_steps>.txt`. For instance, the file containing the accuracies mentioned here is `results/vishnet_20000.txt`.

[21] The time per training step is also included in the table. This time measurement appears in the results files. However, the program which performs training was modified during the process to remove a very costly periodic evaluation of the accuracy on the (large) training set. Therefore, some result files contain average times which are not accurate. In the runs for 60 000 steps, the values in the files are correct for all three architectures.

advantage when generalising to the test set (99.52% compared to tfnet's 99.36%), with a 0.15% difference. This may seem negligible, but it is a considerable improvement in the context of machine learning.

- Regarding the tradeoff between **fast training** and accuracy, all three networks could be considered. Each training step of *vishnet* takes about 27 times as long as one of *lenet* and 3 times as long as one of *tfnet*. Additionally *lenet*'s test accuracy nearly peaks at 20 000 iterations, whereas *tfnet* and *vishnet* do at 60 000 [22]. If we were machine learning experts pressed for time, we could consider quickly training a *lenet* for 20 000 steps to get 98.80% test accuracy, instead of *vishnet* for 40 000 (about 50 times as long) for an improvement of 0.7%. When considering only one architecture, we find that we can often reduce the amount of training steps by a factor of 3 and still be close to the architecture's best accuracy (as was shown in the plots of section 3.2, the loss behaves asymptotically with respect to the number of iterations). As a general, reflection, however, training time will often not be a restricted resource when developing a machine learning model that will run on thousands of devices later on, and therefore *vishnet* would still probably be our network of choice among these three.

Our overall winner is Vishnu's architecture, which we can train to 0.48% test error. This is a good result to end at, considering the current record-holder (mentioned in subsection 4.2.1) is at 0.21%. LeCunn's foundational Le-Net 5 from 1998 still holds as a very fast network with a considerable performance too.

[22] Experiments with more training steps (80 000) were performed, and they did not yield better results than those with 60 000, so we can assume that is the ceiling of each network.

# 5 Conclusions and Future Work

## 5.1 Conclusions

Throughout this project, we have considered multiple aspects of deep learning and TensorFlow . From a purely theoretical point of view, deep nets are an ideal machine learning solution provided there is abundant training data and enough computational power. However, many problems arise in practice (local minima, convergence issues, inherent randomness, etc.) and many aspects are far from well-understood. From network initialisation to learning algorithms, there are countless areas at which improvements can still be made. Two key aspects of working with deep neural networks have also been discussed: avoiding overfitting and tuning the hyperparameters of our network. In particular, our experiments have shown how more weights do not necessarily involve better performance or a longer training time.

Regarding TensorFlow, here are a few of the conclusions we can draw from what we learnt in the project:

- TensorFlow is a library of its own, with a special terminology and working mechanisms. Very complex if we want to fully understand it (including aspects not even mentioned here, such as distributed TensorFlow, creating new kernels, etc.), it takes some learning even for simpler tasks. It is worth considering whether it is the best library for the task we want to solve before delving into it. Other more accessible possibilities, such as Keras and Sklearn, must be kept in mind at the start of any project.

- It is growing at a very fast pace. Only during the writing of this document, three different versions existed (0.11, 0.12 and 1.0). With APIs for four different languages being developed and many modules being collaboratively created, the capabilities of TensorFlow are increasing rapidly. Although software evolution is positive, an unstable API is often an issue for early adopters, especially when the updates are not backwards compatible (such as that from 0.12 to 1.0, altough the developers have tried to make the change as seamless as possible).

- At this point, the documentation leaves a bit to be desired. The amount of functions and classes in the API make it complicated to browse through it, the arguments are not always specified, and some texts are simply too cryptic. Although some improvement has been made with the release of TensorFlow 1.0, there is still an arguably long way to go.

- The fact that *Google* is behind TensorFlow is a reassuring attractive towards it: continuous

development, extensive support and widespread acceptance (the name 'Google' draws an additional number of researchers to it).

- The python API makes TensorFlow very convenient to work with, since it is one of today's preferred languages for machine learning research. This is especially true thanks to *numpy*, extensively used and very efficient. Like *numpy*'s, TensorFlow's backend is written in C++, which allows it to keep good performance across many platforms. Although TensorFlow is still not at par with some other machine learning frameworks in terms of speed, its GPU and distributed versions may earn it a place in the top rankings in terms of performance as well.

- Cross-device portability was one of the main goals of TensorFlow when it set off, and it has been widely reached. Thanks to the existence of device-specific kernels, TensorFlow code can be easily moved between systems and adjusted to the features of each device.

- Multiple wrappers and helper libraries for TensorFlow are being developed *outside* TensorFlow too. *Google* has made a large effort to advertise the library and get people involved (including a *Dev Summit*), and an important community is being built around it. Experience tells us that communities are one of the keys to success for any piece of software.

## 5.2 Future work

The amount of open fronts in the field of deep learning is countless, and working with a specific library adds even more room for research. Here are some interesting topics to consider:

Data curation and management is, as we have seen, one of the vital aspects of training neural nets. One of the challenges in this regard is: *How can we better split and randomize the dataset achieve better learning?* This could, for instance, group similar patterns of the same class in the dataset so that they are not shown to the network consecutively, which can lead to a biased learning. An additional question already being explored is how to artificially create more input patterns to multiply the training set size, thus leading to more general learning (dataset expansion).

Another open question, and arguably one of the hottest topics of machine learning in general nowadays, is how to better choose hyperparameters. Hyperparameter tuning has proven as complex as it is vital, and research on it is intense. Grid search is a very basic technique, and many others are used including random search, Bayesian optimisation and particle swarm. As mentioned earlier, the first hyperparameter optimisation libraries are starting to appear as well, so any contribution to this area would be very interesting.

TensorFlow's API lacks many widespread machine learning techniques. In particular, **network ensembles** (in which several networks are trained simultaneously and their predictions are then averaged with certain weights) are commonplace nowadays, since they reduce the variability of the training process and often yield better results than individual networks. Built-in ensemble capabilities would definitely be an interesting project to consider.

Finally, something which will become increasingly necessary if TensorFlow reaches enough popularity is a web library which allows its use. Whether it is thanks to a small *Javascript* or *PHP* API, or integration with web frameworks such as *Django* (python) or *Laravel* (*PHP*), the ability to easily run TensorFlow models on web servers and display their results would be very convenient and a challenging task.

# Glossary

- **Activation function**: One-dimensional real function which yields the output of a neuron given its input. Examples include the rectifier, sigmoid and hyperbolic tangent functions. Cf. section 2.1.
- **Application Programming Interface (API)**: Specification of a set of function prototypes, modules and interfaces that allow the programmer to interact with a library.
- **Artificial Neural Network (ANN)**: Structure composed of layers of artificial neurons which forward an input signal, process it and produced an output. Throughout this document, ANNs are simply referred to as networks, neural networks or artificial networks. Cf. chapter 2.
  - **Deep Neural Network (DNN)**: Artificial Neural Network with more than one hidden layer.
  - **Convolutional Neural Network** (**CNN** or **convnet**): Artificial Neural Network with convolutional layers, i.e. layers which apply filters to groups of contiguous input features. Often used in image processing. Cf. subsection 4.2.2.
- **Backpropagation**: An algorithm which computes the partial derivatives of the cost function in an artificial neural network with respect to its weights and biases, often needed for training. The errors in the last layer are transmitted backwards for this calculation, which accounts for the name of the algorithm. Cf. section 2.3.
- **Hyperparameter**: Any customisable element in an artificial neural network which is not trained (as opposed to the network parameters: weights and biases). Examples include the number of units and layers, activation functions and regularisation parameters.
- **Kernel**: In TensorFlow, the implementation of a particular graph operation (such as matrix multiplication) on a certain device. In convolutional neural networks, a **filter** which applies a matrix operation to the local receptive field of a unit in order to obtain its activation.
- **Loss** (or **cost**) **function**: Function which compares a network's expected and actual outputs and yields a real value (the usual convention is that lower loss values represent more accurate predictions). Examples include squared error and cross-entropy. Cf. section 2.2.
- **Local Receptive Field (LRF)**: The volume of units in a layer which are connected to a particular unit in the next one. Local receptive fields allow multiple neurons to share the same weight by applying the same filter to square regions of the input layer. Cf. subsection 4.2.2.
- **Modified National Institute of Standards and Technology (MNIST) database**: database containing 70 000 images of handwritten digits. It is a commonly used example dataset in machine learning. Cf. subsection 4.2.1.

- **Operation** or **Op**: Any node in the TensorFlow graph. Most ***op*** constructors return one of two different things: the ***op*** itself (when the ***op*** performs an action on the TensorFlow graph, such as `tf.global_variables_initializer`) or a ***tensor*** (when the ***op*** performs a calculation, such as `tf.matmul`). Cf. subsection 3.1.1.

- **Optimiser**: The TensorFlow implementation of a learning algorithm, which updates the weights and biases in an artificial neural network according to the gradient of the loss function in every training step. The vast majority (Adam, Adadelta, RMSprop etc.) are modifications of Stochastic Gradient Descent with improvements such as adaptive learning rates, momentum, etc. Cf. subsection 3.1.3.

- **Overfitting**: Situation in which a neural network (or any other machine learning model) adjusts itself excessively to the training data, losing the ability to correctly predict unseen examples (generalising). Cf. sections 2.4 and 2.5.

- **Rectified Linear Unit (ReLU)**: An artificial neuron whose activation function is $max(0, x)$. This type of units help prevent vanishing gradients in deep neural networks and are therefore a common choice.

- **Regularisation**: An set of techniques which aim to prevent neural networks from overfitting. Examples include L2 regularisation and dropout. Cf. section 2.5.

- **Session**: A TensorFlow execution context which runs the operations defined in one or more graphs. A sessions can be static (in which case all the nodes must have been defined before creating it) or interactive (allows for interleaved node definition and execution). Cf. subsection 3.1.2.

- **Stochastic Gradient Descent (SGD)**: Learning algorithm for neural networks and the inspiration for all gradient-based methods. During each training step, SGD decreases each weight and bias in the network by a multiple of the partial derivative of the cost function with respect to that parameter (averaged over a batch of examples).

- **Tensor**: The output of a calculation node in the TensorFlow graph. It can be thought of as a multi-dimensional array. Cf. section 3.1.

- **TensorBoard**: Visualisation tool included in TensorFlow which allows to view records of different executions, including loss and accuracy, histograms and graphs. Cf. annex D.

- **Unit** or **neuron**: Each of the basic elements of an artificial neural network, which loosely emulates a brain cell. A unit receives a set of input values, multiplies each of them by a weight, adds a bias and applies an activation function, sending that output forward. Cf. chapter 2.

- **Vanishing gradient**: Phenomenon in (very) deep neural networks in which the partial derivative of the cost with respect to the weights of neurons in the first layers becomes very small, making those units virtually impossible to train.

- **Variable**: A type of node in the TensorFlow graph which stores a value between *python*-TensorFlow context switching. Its constructor creates three nodes (assignment ***op***, initial value and the ***variable*** itself). ***Variables*** must be initialised in a session, usually with `tf.initialize_all_variables`. Cf. subsection 3.1.3.

# Bibliography


[1] Luke Mastin. Neurons & synapses, 2010. Available at: http://www.human-memory.net/brain_neurons.html.

[2] Michael A. Nielsen. Neural networks and deep learning, 2015. Also available at: http://neuralnetworksanddeeplearning.com.

[3] Kur Hornik, Maxwell Stinchcombe, and Halber White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2, 1989. pp. 359-366. Also available at: http://www.cs.cmu.edu/~epxing/Class/10715/reading/Kornick_et_al.pdf.

[4] Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. *Neural Networks: Tricks of the Trade*, chapter 1, pages 9–50. Springer Verlag, 2 edition, 2012. Also available at: http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf.

[5] Jeff Dean and Andrew Ng. Using large-scale brain simulations for machine learning and a.i., 2012. Available at: https://googleblog.blogspot.com.es/2012/06/using-large-scale-brain-simulations-for.html.

[6] Lutz Prechelt. *Neural Networks: Tricks of the Trade*, chapter 2, pages 55–69. Springer Verlag, 2 edition, 2012. Also available at: http://page.mi.fu-berlin.de/prechelt/Biblio/stop_tricks1997.pdf.

[7] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv*, 2012. Available at: http://arxiv.org/pdf/1207.0580.pdf.

[8] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15, 2014. pp. 1929-1958. Also available at: http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf.

[9] 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. Available at: http://download.tensorflow.org/paper/whitepaper2015.pdf.

[10] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016. Available at: http://sebastianruder.com/optimizing-gradient-descent/index.html.

[11] Diederik P. Kingma and Jimmy L. Ba. Adam: A method for stochastic optimization. *arXiv*, 2015. Available at: https://arxiv.org/pdf/1412.6980v8.pdf.

[12] Jost T. Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: the all convolutional net. *ICLR*, 2015. Available at: https://arxiv.org/pdf/1412.6806v3.pdf.

[13] Yann Le-Cun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 1998. pp. 2278-2324. Also available at: http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf.

[14] Grzegorz Gwardys. Convolutional neural networks backpropagation: from intuition to derivation, 2016. Available at: https://grzegorzgwardys.wordpress.com/2016/04/22/8/.

[15] Vishnu Sundaresan and Jasper Lin. Recognizing handwritten digits and characters, 2014. Available at: http://cs231n.stanford.edu/reports/2015/pdfs/vishnu_final.pdf.

[16] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, 2010. pp. 249-256. Also available at: http://www.cs.cmu.edu/~bhiksha/courses/deeplearning/Fall.2016/pdfs/1111/AISTATS2010_Glorot.pdf.

# A

# TensorFlow installation

This annex describes how to install the CPU version of *python* TensorFlow on a Linux system. There exists a GPU version which can run on NVIDIA devices with CUDA[1] for parallel execution and improved performance (the same applies to *C++* TensorFlow), and it has a similar installation process. The official documentation[2] offers up to five installation methods: pip, virtualenv, conda, docker or building from source. Conda and virtualenv allow us to keep cleaner separate python environments, so they are our preferred choices and we shall describe installation with conda (it is worth mentioning the pip installation can be troublesome).
This guide **assumes** the user has a Linux system with *python* 3.5. This is included in the current Ubuntu version, 16.x.

1. **Install (Mini)conda**

   Conda is a package and environment manager for *python*. Mini-conda is a *python* distribution which includes Conda and a few essential packages. For a full-blown installation with more than 100 packages for scientific computing, machine learning, statistics, etc., refer to Anaconda. If Conda is already installed in your platform, skip to step 2.

   In the following commands, replace `<installer>` by

   `Miniconda3-latest-Linux-x86_64.sh`

   for 64-bit operating systems, or

   `Miniconda3-latest-Linux-x86.sh`

   for 32-bit ones:

   ```
   $ wget https://repo.continuum.io/miniconda/<installer>
   $ bash <installer>
   ```

2. **Set up a Conda environment**.

   Choose an `<env_name>` for the environment and run:

   ```
   $ conda create -n <env_name> python=3.5
   ```

[1] `http://www.nvidia.com/object/cuda_home_new.html`
[2] `https://www.tensorflow.org/versions/r0.12/get_started/os_setup.html`.

3. **Activate the conda environment**.
   In order to start working in the new Conda environment, run:

```
$ source activate <env_name>
```

   The prompt will change to signify the new source. In particular, we need to do this:

   (a) To install TensorFlow and any other packages (in that environment).

   (b) Whenever we want to run a *python* program which uses TensorFlow.

   (c) Whenever we want to launch TensorBoard[3].

   Additionally, if we want to run a TensorFlow program from an IDE, we need to set the *python* interpreter path to the interpreter in the environment, which may look like `/anaconda/bin/python`.

   When we finish doing a task in the environment, we can exit it by running

```
$ source deactivate
```

   s

4. **Install TensorFlow**

   With the environment active, installing TensorFlow is as simple as running[4]:

```
(env_name)$ conda install -c conda-forge tensorflow
```

   **Note**: The above command will install the latest version of TensorFlow. This project was developed with TensorFlow 0.12, which has now been replaced by 1.0 (and not in a fully backwards-compatible way). In order to install that version, replace `tensorflow` by `tensorflow=0.12`.

   As a side note, the `-c conda-forge` option indicates the channel (which we can think of as a conda repository) to retrieve the package from.

[3] TensorBoard comes with any TensorFlow installation.

[4] With the release of TensorFlow 1.0, the library was added to the official Conda repository. This means the `-c conda-forge` portion of the command is no longer necessary

# B

# Operators overloaded by TensorFlow

The TensorFlow API overloads the some common operators for ***tensors***. This means that if `a` or `b` are ***tensors*** and `*` is a binary operator in the list below, then the code `a * b` evaluates to the corresponding function for `tf.Tensor` (and analogously for unary operators). If only one of them is a ***tensor***, `tf.convert_to_tensor` is called on the other one, raising an exception if not possible. Surprisingly enough, the list cannot be found in the API documentation, but it is easy to find on the internet[1]:

- `__neg__` (unary `-`)
- `__abs__` (`abs()`)
- `__invert__` (unary `~`)
- `__add__` (binary `+`)
- `__sub__` (binary `-`)
- `__mul__` (binary elementwise `*`)
- `__div__` (binary `/` in Python 2)
- `__floordiv__` (binary `//` in Python 3)
- `__truediv__` (binary `/` in Python 3)
- `__mod__` (binary `%`)
- `__pow__` (binary `**`)
- `__and__` (binary `&`)
- `__or__` (binary `|`)
- `__xor__` (binary `^`)
- `__lt__` (binary `<`)
- `__le__` (binary `<=`)
- `__gt__` (binary `>`)
- `__ge__` (binary `>=`)

[1] Credit: this one is taken from *mrry*'s answer in the following *StackOverflow* thread: `http://stackoverflow.com/questions/35094899/tensorflow-operator-overloading`.

# C

# A small dataset module

This annex briefly describes a small module we created in order to manage datasets, which is used in section 3.2. It allows for dataset loading, splitting, normalisation, encoding, and mini-batch handling. TensorFlow's high-level ML API, `tf.contrib.learn`, offers some dataset management capabilities itself within the `tf.contrib.learn.datasets` module and has, in fact, some preloaded popular datasets (*MNIST* and *Iris*). The module described here is therefore more of a programming exercise in python and *numpy* than a necessary utility. However, our ad-hoc interface is very convenient and offers some additional capabilities (e.g. normalisation). Here is the module's internal structure:

```
    class Problems(Enum)
    class Encodings(Enum)
    class DatasetException(Exception)

    class Dataset:

    def __init__(self, path, problem, name='default')

    def to_one_hot_encoding(self)
    def to_numeric_encoding(self)
    def normalise(self, method='normal')
    def split(self, training_prop, test_prop)

    def get_n_features(self)
    def get_encoding(self)
    def get_normalisation(self)
    def get_classes(self)
    def get_class_mapping(self)
    def get_n_classes(self)
    def get_information(self)

    def _subdata(self, inputs, targets, indices, transpose)
    def get_data(self, inputs=True, targets=True, transpose=False)
    def get_training_data(self, inputs=True, targets=True, transpose=False)
    def get_validation_data(self, inputs=True, targets=True, transpose=False)
    def get_test_data(self, inputs=True, targets=True, transpose=False)
    def get_minibatch(self, size, autoreset=False, transpose=False)
    def reset_minibatches(self, shuffle=False)
```

The file has about 500 hundred lines, but this is mostly due to documentation and a sparse programming style. The actual LOC count is around 180. The `Dataset` class abstracts all the data operations. Its constructor loads the information from a CSV file. We need to specify whether it is a regression or a classification problem. In the former case, the process is quite

straight-forward: the last value is taken as the target for that input. If it is a classification problem, however, the module keeps track of all the classes found (which could be strings, such as `'I. versicolor'`) and assigns an index to each of them, creating an internal mapping between class names and indices. The inputs and targets are stored in separate *numpy* arrays (each example corresponds to a row and not a colum, for consistency with the `len` method).

The next few methods offer some control operations on the dataset. The first two, `to_one_-hot_encoding` and `to_numeric_encoding` allow us to seamlessly change between class indices and one-hot target vectors in classification problems. `normalise` offers three ways to normalise the input features: standard (transform each feature linearly into a distribution with mean 0 and standard deviation 1)), linear (transform into [0, 1]) and symmetric (transform into [-1, 1]). The `split` method separates the dataset into training, validation and test data with the given proportions. Interestingly, it is designed to distribute all classes equally in each of the sets in classification problems (or as equally as possible), to prevent unbalanced training. This is known as *stratified sampling*.

After those control operations come some basic getters and setters for abstraction. In particular, we can get the $class\,index \leftrightarrow class\,name$ mapping if the problem is a classification one. The last group of methods allow us to obtain the data in the dataset. They all have two boolean arguments, `inputs` and `targets`, to specify whether we only want either of those, or both the input and target values. An additional parameter, `transpose`, allows us to determine whether we need the examples as rows or columns (which we need to use, since our network considers the input to be a column, as established in chapter 2). What is interesting is that, when we split the dataset, the original *ndarrays* with the inputs and targets remain untouched: the training, validation and test sets are simply lists with the indices of their samples in the original array. This is more efficient and allows for easier re-splitting and mini-batch shuffling. The private `_subdata` method takes in a list of indices and returns the actual corresponding data. All the methods after `_subdata` work with indices only, and then call it to return the actual data.

The last method is `get_mini-batch`. Most of today's neural networks need huge datasets to be trained properly. It would therefore be too expensive to estimate the gradient as an average of the gradients for the whole training set. For that reason, machine learning practitioners often select some of the training examples at random, and estimate the overall gradient as an average of only those. That group of examples is called a mini-batch. `get_mini-batch` returns a number of still unseen examples from the training set (and if there are fewer unseen ones than requested, it returns only the remaining ones). When all of the examples have been consumed, we can call `reset_minibatches` to mark them as unseen and start over, optionally shuffling them. This is unnecessary if we always call `get_minibatch` with `autoreset=False`.

# D

# TensorBoard example: the Iris program

This annex briefly describes how to use the Iris program explained in chapter 2 and browse the Tensorboard results. In order to run the program, activate the corresponding environment if TensorFlow was installed in one (for instance, as described in annex A). Then execute the program as any *python* script:

```
(env_name)$ python network_in_one.py
```

Whenever we run it, a new log folder is created: `tensorboard_logs/exec_<N>`, where $N$ is the next number to the last previous execution (or 1 if there are none). We can change the name of the log folder later on to denote special executions. The runs described in section 3.2 are stored under `tensorboard_logs/` and they all have their own name.

When the program finishes training the network, we are offered to open Tensorboard. If we accept, a local Tensorboard server is launched at `localhost:6006`, and a web browser automatically opens that address. Note that the server is independent from the browser, and it will still be running when we close the window. In order to actually finish using Tensorboard, press `Ctrl + C` in the terminal the program was launched from. If anything goes wrong, Tensorboard can also be shut down directly:

```
$ netstat -tupln | grep 6006
```

allows us to find out Tensorboard's PID. We then execute

```
$ kill <pid>
```

to finish it. We may want to simply run TensorBoard to view the results of previous executions (instead of training the whole network). The command to do so (after activating the environment if necessary) is:

```
(env-name)$ tensorboard --logdir=<path/to/tensorboard_logs>
```

After that we need to open the address `http://localhost:6006` on any web browser.

The TensorBoard interface can be seen in figure D.1. The top menu allows us to switch between logs of scalar values, images, sound, graphs, distributions, histograms (distributions of scalar values) and embeddings. The only values logged in our Iris program are scalars, but the graph is recorded whenever we log other types of values too. Those two are therefore the only options we shall discuss now

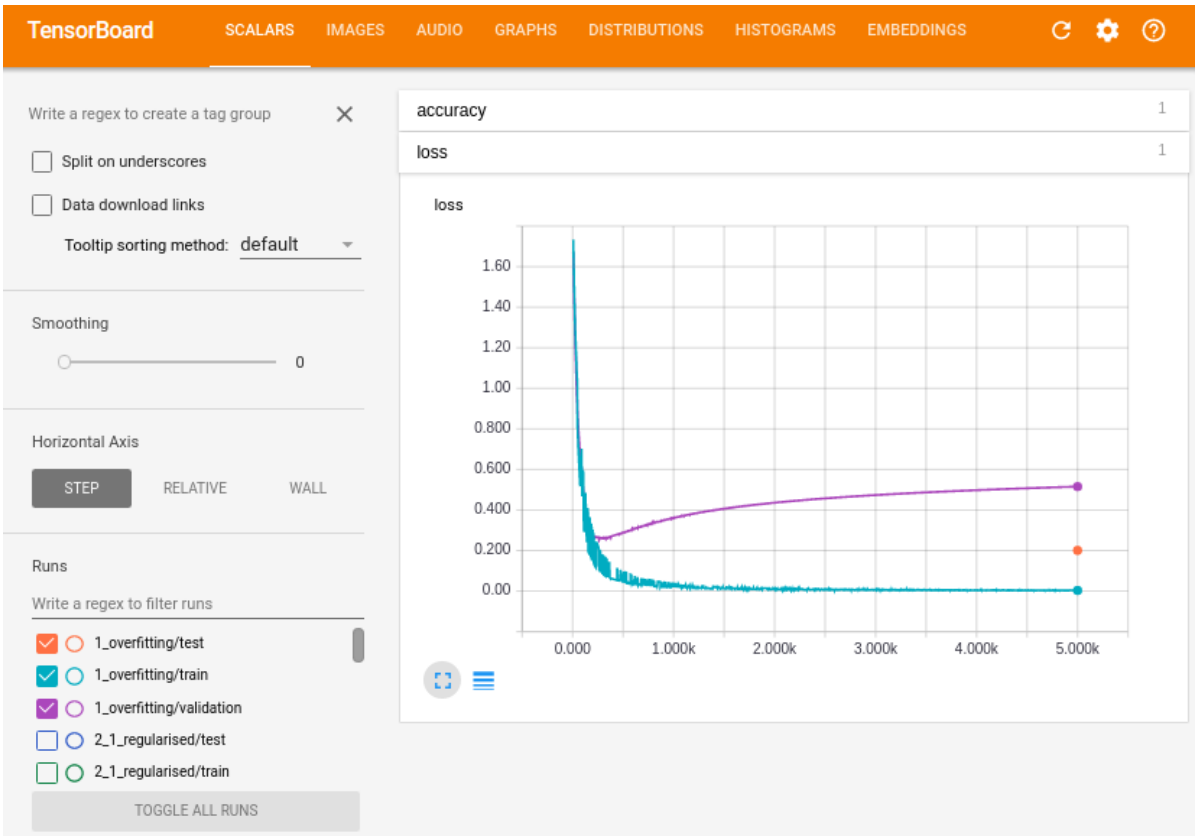



**Figure D.1:** Example of TensorBoard interface.

`Important note`: More often than not, when we open TensorBoard, the main area in the centre is blank, even though our log directory does have data. The way to solve this is to either refresh the page or reload the data by clicking the circular arrow in the top-right corner of the interface. This last solution also solves the issue of missing data when we select any of the runs in the bottom-left menu. Last but not least, depending on the screen size, the "Runs" menu in the bottom-left part may not be visible. Inexperienced users may not even know it exists, which is particularly bad since it is likely the most used menu. In order to fix this, zoom out in the web browser.

The scalars menu shows us our recorded data regarding loss and accuracy. We can select which execution(s) we wish to display with the "Runs" menu mentioned above. Each of our executions generates three subfolders: train, validation and test. We can check and uncheck the runs (one by one, or all of them with "Toggle all runs") in order to compare them. The small blue dashed square in the main area expands the plot, and we can zoom in or out by clicking-and-dragging and double-clicking respectively. Furthermore, placing the cursor anywhere inside the plot displays the values of all the curves at that vertical line.

# E

# Hyperparameter exploration tables

This small annex contains the results of the grid search performed for each network in subsection . The python code used to fo so is contained in `tuner.py`. When launched, it reads (among several others) the following values from the `settings.py` configuration file: `N_STEPS`, the number of training iterations to perform on each network; `NETWORK`, a string with the name of the architecture to use (`'lenet'`, `'tfnet'` or `'vishnet'`); `HYPER_ID`, the name we want to save the results and the model with; and `SWEEP_LAMBDAS` and `SWEEP_KEEP_RATES`, two lists of *float* values for each of the two hyperparameters we want to test (the number of networks trained during the execution is the product of the lengths of these two lists).

When we run the program, two files and one folder are created: `results/<HYPER_ID>.txt` contains all the information about each of the networks trained: the elapsed time, accuracies, etc.; `results/<HYPER_ID>.p` is a python *pickle* file which contains some numeric information (accuracy and hyperparameters of each network) which can be loaded from the code in order to avoid checking information manually later on; and `models/<HYPER_ID>_<n>/)`, one folder per network trained which contains the model with the tuned parameters. This can be run afterwards from the *runner.py* program for specific information.

The tables in this section are drawn from the `.txt` files in the `results/` folder. Except for E.2, the tables are built as follows: the values in the inner cells correspond to the network accuracy % on the MNIST validation set. The top index is the network's $\lambda$ (L2 regularisation) hyperparameter, and the left index is $\rho$ (dropout keep rate). The bottom row shows the average value of each column, whereas the right column shows the average value of each row. The best result in each table is highlighted in green (or blue), as well as the corresponding hyperparameter values.

| | **0.001** | 0.005 | 0.01 | 0.05 | 0.1 | 0.5 | 1 | AVG |
|---|---|---|---|---|---|---|---|---|
| 0.25 | 98.46 | 97.96 | 97.94 | 94.26 | 89.60 | 11.26 | 11.26 | 71.53 |
| **0.4** | **98.50** | 97.92 | 97.66 | 94.38 | 89.36 | 11.26 | 11.26 | 71.48 |
| 0.5 | 97.96 | 98.04 | 97.46 | 94.56 | 89.88 | 11.26 | 11.26 | 71.49 |
| 0.6 | 98.42 | 97.98 | 97.50 | 94.90 | 90.00 | 11.26 | 11.26 | 71.62 |
| 0.75 | 98.04 | 97.96 | 97.66 | 94.56 | 89.92 | 11.26 | 11.26 | 71.52 |
| AVG | 98.28 | 97.97 | 97.64 | 94.53 | 89.75 | 11.26 | 11.26 | |

**Table E.1:** *lenet* coarse tuning results. The corresponding file is `results/broad_lenet.txt`.

| | 0.0001 | **0.0004** | 0.0007 | 0.001 | 0.004 | 0.007 | 0.01 |
|---|---|---|---|---|---|---|---|
| **0.5** | 98.2 | **98.38** | 98.2 | 98.12 | 98.12 | 97.8 | 97.68 |

**Table E.2:** *lenet* fine tuning results. The corresponding file is `results/narrow_lenet.txt`.

| | 0.000005 | 0.000008 | 0.00001 | **0.00002** | **0.00005** | AVG |
|---|---|---|---|---|---|---|
| 0.25 | 99.14 | 99.18 | 99.16 | 99.14 | 99.12 | 99.15 |
| 0.4 | 99.14 | 99.08 | 99.24 | 99.18 | 99.20 | 99.17 |
| 0.5 | 99.02 | 99.20 | 99.04 | 99.24 | 99.20 | 99.14 |
| **0.6** | 99.16 | 99.06 | 98.96 | **99.30** | 99.16 | 99.13 |
| **0.75** | 99.06 | 99.16 | 99.22 | 99.08 | **99.30** | 99.16 |
| AVG | 99.10 | 99.14 | 99.12 | 99.19 | 99.20 | |

**Table E.3:** *tfnet* tuning results. The corresponding files are `results/final_tfnet_<n>.txt`, where `n` $\in \{025, 04, 05, 06, 075\}$ corresponds to each $\rho$. The reason for this is those processes needed to be launched as separate parallel executions, since a single run with all of the networks would have exceeded the maximum execution time in the queue system.

| | **0.0000003** | 0.000001 | 0.000003 | 0.00001 | 0.00003 | AVG |
|---|---|---|---|---|---|---|
| 0.25 | 88.20 | 99.30 | 99.30 | 99.40 | 99.16 | 97.07 |
| **0.4** | **99.44** | 89.44 | 99.18 | 78.46 | 99.26 | 93.16 |
| 0.5 | 79.52 | 67.78 | 77.50 | 99.22 | 50.94 | 74.99 |
| 0.6 | 89.52 | 80.44 | 60.36 | 70.14 | 99.36 | 79.96 |
| 0.75 | 80.96 | 30.00 | 79.54 | 70.62 | 99.06 | 72.04 |
| AVG | 87.53 | 73.39 | 83.18 | 83.57 | 89.56 | |

**Table E.4:** *vishnet* tuning results. The corresponding files are `results/final_vishnet_<n>.txt`, where `n` $\in \{025, 04, 05, 06, 075\}$ follows the same principle as in the *tfnet* table.

`Note`: The fact that the last two columns of values in table E.1 contain the exact same number all across has a simple explanation: when the value of $\lambda$ is too high in relation to the loss function the the best step in each training iteration is the one that reduces the weights (in absolute value) the most, because even an improvement in the network's accuracy (or original loss) will not be enough to compensate the increase in the regularisation term of the cost, which is amplified by such a large factor. As a result, the network always converges to a configuration where all the weights are 0, and the output is the same along all of those 'different' networks. In this case, that means 11.26% is the validation accuracy of a network with null weights (since it is essentially random classification, the accuracy is close to the inverse of the number of classes, 10).