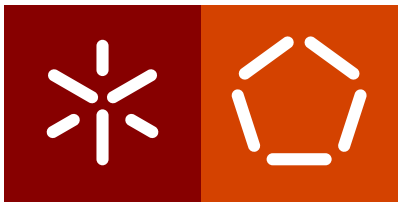
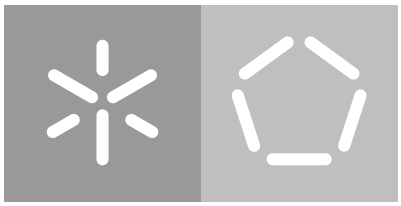


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Bruno Miguel da Silva Barbosa

Image Recognition Using Deep Learning

September 2018



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Bruno Miguel da Silva Barbosa

Image Recognition Using Deep Learning

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
António Ramires
Manuel João Ferreira

September 2018

ACKNOWLEDGEMENTS

I would like to specially thank my family for supporting me during all these years. I also want to thank professor António Ramires as well as Manuel João Ferreia, for the guidance, patience and assistance on this dissertation.

ABSTRACT

Computer vision is a vast knowledge subject responsible for traducing digital images and videos into a higher level of understandable information. Image recognition is one of the several tasks that are inserted in this subject and it can be subdivided in object recognition (also called as object classification), segmentation, identification and detection.

Some of the available alternatives for image recognition are based on Machine Learning (ML) approaches. Deep Learning (DL) is a branch of ML that became very popular in the last years due to its success in previously considered hard tasks. The lack of large amounts of data and efficient computational resources a few years ago, were a barrier for the expansion of DL. However, thanks to the current easy data access and due to development of more powerful computational resources, including CPU and GPU too, the attention turned back on, and it became easier and faster to train a model than can distinguish different types of classes with a very low error rate. One interesting fact about DL is its ability to automatically learn from data and understand the most differentiable features of it.

From the point of view of the industry, many artificial vision inspection lines still do their jobs relying on traditional computer vision methods/algorithms. Yet, with more complex domains, for example like texture patterns, things can get more difficult. This is where DL comes in.

This document begins with an introduction of DL for artificial vision. It starts by addressing the theoretical fundamentals of DL for image recognition and then focuses on the general aspects of Convolutional Neural Networks (CNN). Next, are reviewed the state of the art network configurations that stood out in recently.

A high-level toolkit for image recognition was created to simplify the whole process of building DL models, from the data pre-processing to the trained model testing phase. It allowed to easily prepare a set of experiences that address some of the common practices used on CNNs and highlight the power of DL on image recognition related tasks.

This dissertation was developed under a business environment on a artificial vision company called Neadvance, Machine Vision, SA. The Neadvance, Machine Vision, SA is also interested in researching the new trends related to DL for image recognition in order to know how to apply them on their projects since it opens a new range of challenging opportunities.

RESUMO

A visão por computador é uma área vasta de conhecimmento responsável por traduzir imagens e vídeos digitais para um nível mais alto de informação compreensível. O reconhecimento de imagem é uma das várias tarefas que está inserida nesta área e pode ser sub-dividida em reconhecimento de objectos (também designada por classificação de objectos), segmentação, identificação e detecção.

Algumas das alternativas disponíveis para reconhecimento são baseadas em abordagens de ML. O DL é um ramo de ML e tornou-se muito popular nos últimos anos devido ao seu sucesso em tarefas consideradas difíceis, até ao momento. A falta de grande quantidade de dados e de recursos computacionais eficientes há uns anos atrás, foram uma barreira para a expansão do DL. Contudo, graças à actual facilidade de acesso a dados e devido ao desenvolvimento de recursos computacionais mais potentes, incluindo CPU e GPU também, a atenção à volta do tema voltou a crescer, e tornou-se mais fácil e mais rápido treinar um modelo que consegue distinguir diferentes tipos de classes com uma taxa de erro baixa. Um facto interessante sobre o DL, é a sua capacidade para aprender dos dados e compreender as suas características mais diferenciadoras.

Do ponto de vista da indústria, muitas linhas de inspecção via visão artifical ainda fazem o seu trabalho através de métodos/algoritmos tradicionais de visão por computador. Todavia, com domínios mais complexos, como por exemplo padrões de texturas, as coisas podem tornar-se mais difíceis. É aí onde entra o DL.

Este documento inicia com uma introdução ao DL para visão artificial. Começa por abordar os fundamentos teóricos de DL para reconhecimento de imagem e de seguida focase em aspectos gerais das Convolutional Neural Network (CNN)s. Depois, são revistas as configurações estado da arte das arquitecturas de rede que se destacaram recentemente.

Foi criado um conjunto de ferramentas para simplificar todo o processo de construção de modelos de DL, desde o pré-processamento dos dados até à fase de testes do modelo treinado. Este permitiu ainda preparar facilmente uma série de experiências que abordam algumas das práticas comuns usadas nas CNN e destacar o poder do DL em tarefas relacionadas com reconhecimento de imagem.

Esta dissertação foi desenvolvida sob ambiente empresarial numa empresa de visão artificial chamada Neadvance, Machine Vision, SA. A Neadvance, Machine Vision, SA também está interessada em investigar as novas tendências relacionadas com o DL de forma a saber como aplicá-las nos seus projectos, uma vez que lhe possibilita uma nova gama de desafios.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION

1.1 CONTEXT

Computer vision is such a vast knowledge subject responsible for traducing digital images and videos into a higher level of understandable information. In it, are inserted different kind of tasks just like image recognition in the form of image classification, segmentation, identification and detection. Image classification is a classic computer vision problem and consists into assign a category label to an image. Depending on the approach or in the complexity of the problem, the assignment process can be more or less accurate.

Image classification has a wide range of applications in real world scenarios. For instance, many industries have inspection lines that rely on artificial vision solutions in order to guarantee that products are free of defects or without any worrisome issue. The efficiency of the inspection line is directly influenced by the time it needs to inspect an object and by its error rate. If the inspection takes too much time analyzing each object, less units will be inspected per time unit. Also, if many defective objects end up not being discarded it may require constant human intervention to assert about the status of the object. Both of these situations result in a loss to the manufacturer. So, in this context, having a reliable and balanced system is crucial for industries.

Some of the available alternatives for image recognition are based on machine learning approaches. Deep learning is a branch of machine learning and has been successfully applied in considered very hard tasks so far. In this sense, it stands as a good choice for image recognition essentially due to its robustness, even in complicated domains with a lot of data variance. Deep learning methods learn from data instead of rule based programming. Still, the same CNN architecture can deal with many distinct problems. However, according to what was mentioned before, it is also important to know how to configure a balanced deep neural network with the view to build a system that simultaneously has a high success rate and it is computationally efficient.

So, the process of building a custom CNN requires some theoretical fundamentals about their behavior and about which parameters can be tuned. In this context, this dissertation presents an introduction to deep learning on image recognition by focusing on CNNs. The

state of the art network architectures are a good reference for it since they were meant for large scale challenges. The same way those concepts can be adapted on less complex but demanding tasks that require a high level of accuracy as well.

1.2 MOTIVATION

The hype around deep learning is huge lately and new advances are constantly emerging. The deep learning generalization capabilities makes it very efficient and robust for a large range of tasks and domains. Therefore, this is a subject with a lot of potential that deserves to be explored in order to know how to take the most profit from it.

As an artificial vision company, Neadvance, Machine Vision, SA deals with projects related to image recognition everyday. The introduction of deep learning to their software means an important step and opens a new range of challenging opportunities. Likewise the inspection lines scenario, some of the projects are limited by the available processing time per image and efficiency. Thus, the network configuration stands as a major task to achieve both.

1.3 OBJECTIVES

In its core, deep learning is a complex area and the adaptation to it can be time consuming. So, in the foreground this dissertation pretends to offer a didactic and intuitive introduction to image recognition through deep learning and CNNs. Secondly, this dissertation is meant to expose the fundamental theories needed to understand how CNN behave through a set of experiments. These experiments are also intended to suggest a bunch of common practises when working with deep learning and show the impact of tuning certain parameters, in search for a good configuration for them, basically as a trade-off between efficiency and performance.

Along with this dissertation was developed a high-level abstraction toolkit for image recognition. The toolkit was designed to be the most plug-and-play experience as possible. Although it was not planned to have any graphical user interface, following the hints given to in the command line, the user should be able to build a custom deep neural network, train it on a specific dataset and test the performance of the trained model with just a few commands.

1.4 DISSERTATION STRUCTURE

This document is structured in five chapters.

On the first chapter is introduced the purpose of this master's thesis by contextualizing what it is all about. Then are referred the motivations that lead to it and the goals that meant to be achieved.

The second chapter is divided in three main sections and it presents the state of art of deep learning for image recognition. The first section starts with a brief historical reference about how deep learning emerged and what pushed it to the landing it has nowadays. Then, this section is followed by a theoretical fundamentals section that will be useful for understanding how deep learning works under image recognition, more specifically on CNNs. The last section exposes a set of state of the art network architectures and benchmarks that show the evolution of CNNs performance over the recent years.

On the third chapter is described the common procedures for training and deploying a deep learning model. In it, are also explained some frequently used operations in order to improve the model generalization capabilities as well as a few assertations about tunning specific hyper-parameters. The last section of this chapter is entirely dedicated to the description of the developed high-level deep learning toolkit. There, are explained the reasons that motivated the creation of this toolkit. Besides that, it comes with minimalist examples of how to configure a network architecture, train a dataset on that architecture and test the trained model on the field.

The fourth chapter presents the test environment. It includes the specifications of the machine where the experiences were done, the datasets used on the experiences as well as the chosen deep learning framework. Afterwards it exposes the outcomes for a set of experiments that derived from some of the statements previously presented on chapter 3.

In the fifth and last chapter are presented not only the conclusions that came out of this whole dissertation but also a couple of suggestions for future work. These suggestions consist in an alternative high potential deep learning tasks related to image recognition.

2

DEEP LEARNING BACKGROUND

This chapter starts by making a brief contextualization about deep learning in general through a overview and a historical reference of it. Then will be presented the theoretical fundamentals needed to understand how to properly configure a deep neural network. At last, are exposed the state of the art network architectures over the last years as well as the state of the art benchmarks on well known datasets among image recognition tasks.

Therefore, before diving into deep learning fundamentals, lets make a brief contextualization about what it is, from where did it come from and the reasons that pushed it to the landing it has nowadays.

Machine learning is a field of artificial intelligence and it stands as an alternative to ruled based programs by learning from examples. Deep learning is a branch of machine learning and it is meant to move machine learning closer to the conception of Artificial Intelligence Buduma and Locascio (2015); Tawfique (2016); Team. A Deep Neural Network (DNN) can be seen as an extension of a traditional Artificial Neural Network (ANN). DNNs have application on many distinct domains, as Barker (2016); Wang and Raj (2017); Goodfellow et al. (2016) mention:

- **Internet & Cloud** - for image classification, speech recognition, in language translation/processing, sentiment analysis and recommendations.
- **Medicine & Biology** - on cancer cells detection, diabetic grading and drug discovery.
- **Media & Entertainment** - with video captioning/search and real time translation.
- **Security & Defense** - for face detection, video surveillance, satellite imagery and fraud detection.
- **Autonomous Machines** - like self driving cars, that require pedestrian detection, lane tracking and traffic sign recognition.

Deep learning has being used to solve very complex tasks Goodfellow et al. (2016); Wang and Raj (2017). As a supervised learning approach, it comes down to train a model with a known set of data in a given network architecture. After that, the model is deployed in order to deal with new or unknown data.

Actually, the model consists in a DNN which is a deep ANN. An ANN is mathematical model composed by many simple connected processors Schmidhuber (2014), called artificial neurons, that are disposed by layers and capable of learning complex functions all together Barker (2016). The term "deep" refers to the number of layers in the network, far surpassing traditional neural networks, with deep networks having dozens of layers Goodfellow et al. (2016). DNNs are highly data dependent, however with sufficient examples, they can infer very complex functions that relate raw input data with the desired outputs Barker (2016).

In summary, deep learning stands out essentially due to three factors: robustness, generalization capacity and scalability. The model is able to automatically learn from data the most important features even in the presence of noise Buduma and Locascio (2015). Also, the model's predicting performance tends to improve when trained with more data Barker (2016).

Figure 1 evidences how deep learning outperforms traditional computer vision algorithms on the annual ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in the recent years. Because of that, many big companies, such as Nvidia, Microsoft and Facebook, as well as several universities and start-ups, are investing ways to apply this knowledge on their businesses and on educational programs Barker (2016); Goodfellow et al. (2016).

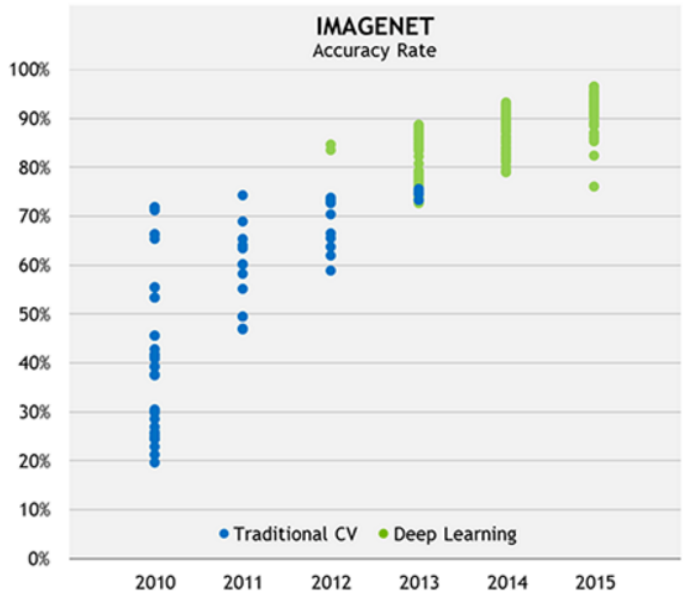


Figure 1: Comparison between traditional computer vision methods and deep learning methods accuracy rate on ImageNet dataset. Image by Huang (2016).

2.1 A BRIEF HISTORICAL REFERENCE

This section is meant to introduce a short historical reference of deep learning through the years. It isn't very extended and it is intended to focus just in the most remarkable achievements.

In 1965, Ivakhnenko and Lapa introduced the first deep-learning-like algorithms using a simple deep network architecture composed of multiple polynomial activation functions. The best features were selected using statistical procedures and forward to the following layer. They used least-squares fitting in which previous layers were fitted independently from later ones Dettmers (2015b); Goodfellow et al. (2016); Schmidhuber (2014).

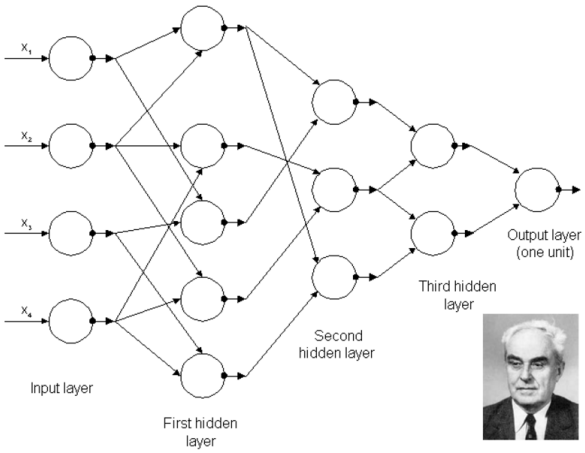


Figure 2: First deep neural network architecture trained by Alexey Grigorevich Ivakhnenko in 1965. Image by Dettmers (2015b).

Fukushima presented the first CNN in 1979, and back then they were already very similar to the current ones, as they were composed by manifold convolutional and pooling layers. The error back-propagation was still an obstacle at that time Schmidhuber (2014). The first practical application of error back-propagation under the context of neural networks, appeared only in 1989 and enabled handwritten digits recognition on the MNIST dataset Goodfellow et al. (2016); Dettmers (2015b).

Despite of this success, there were not much investments in neural networks research. In addition, the financial support for research became even more scarce, particularly, during a period designated as the *AI Winter*, marked by the failed promises of artificial intelligence Kurenkov (2015). Even so, some important advances were made in this period, such as the development of Long Short-Term Memory (LSTM) for Recurrent Neural Network (RNN) in 1997 by Hochreiter and Schmidhuber. However in 1995, due to the arrival of Support Vector Machines (SVM) by Cortes and Vapnik, the new results on deep learning became mostly unnoticed Goodfellow et al. (2016); Dettmers (2015b).

The lack of computational resources and large amounts of data also contributed for the deep learning decay. The computers of that time simply weren't powerful enough to train DNNs in an acceptable time. With the development of new and better hardware, including Graphics Processing Units (GPU), it became possible to accelerate the network training and DNNs started to slowly rise up again, rivalling against SVMs. Actually, the trade-off between SVMs and DNNs stands on time and precision. SVMs are much faster but DNNs can give better classification results on the same data. Besides that, DNNs are able to keep improving with additional training data, while in the SVMs this improvement is more limited Goodfellow et al. (2016); Dettmers (2015b).

CNNs gained a special attention on 2012 Schmidhuber (2014). Alex Krizhevsky together with Ilya Sutskever, and Geoffrey Hinton developed a CNN architecture composed by rectified linear units activation functions and dropout layers for regularization Krizhevsky et al. (2012). Their network performance on the ILSVRC was outstanding Krizhevsky et al. (2012). It was a turning point to the adoption of feature learning, in the form of deep learning, instead of feature engineering which is based on handcrafted feature extraction. Since then, the research and hype around deep learning grew up dramatically and it started to take over traditional computer vision methods Dettmers (2015b,c); Buduma and Locascio (2015).

2.2 THEORETICAL FUNDAMENTALS

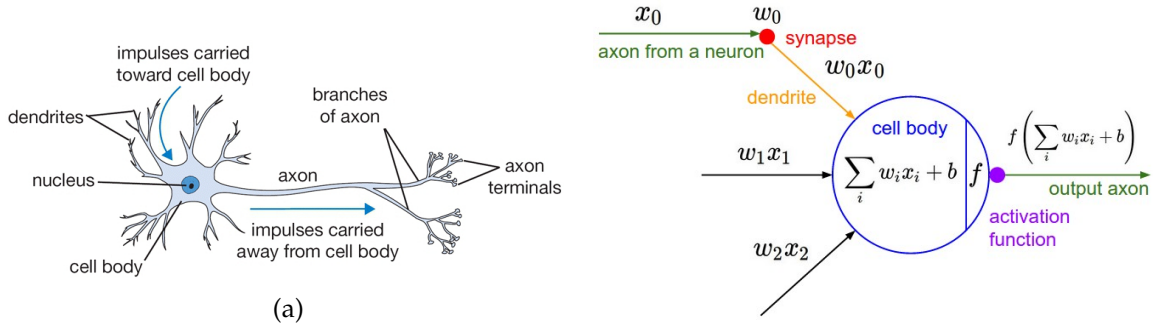
This sections presents the essential theoretical fundamentals to understand the basics of deep learning starting from the notion of artificial neuron, moving on to how they can be assembled in a network, and finally introducing CNNs.

2.2.1 *Artificial Neuron*

The human brain is a massive network of, approximately, 86 billion neurons Karpathy and Johnson (2016); Buduma and Locascio (2015). The neuron represents the basic computational unit of the brain. Neurons inputs come from dendrites, which may be eventually connected to other neurons. The input signals (x_n) are processed all together according to the strength (or weight, w_n) of their connection. Finally, the new generated signal is sent through the output axon to other neurons dendrites Karpathy and Johnson (2016); Buduma and Locascio (2015).

Figure 3 has a visual representation of a biological neuron that inspired the creation of the mathematical model for artificial neurons.

So, an artificial neuron, also known as neuron, is the base element of an ANN. It transforms a set of inputs, $(x_1, x_2, ..., x_n)$, into a single value using a weighted sum, since each input has a respective weight $(w_1, w_2, ..., w_n)$, plus a bias term (b). At last, the neuron output



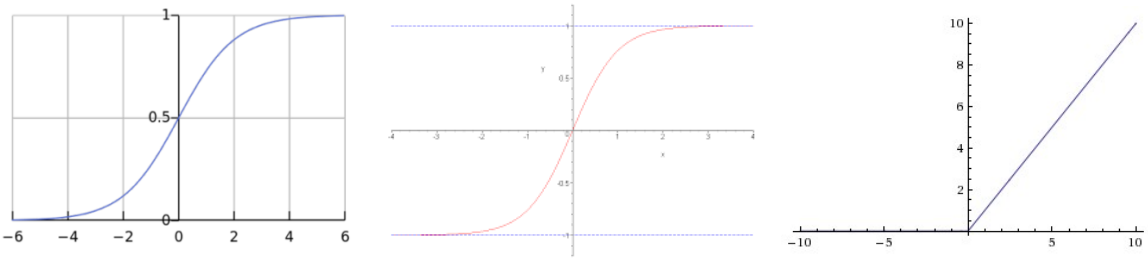
(a) (b)

Figure 3: Visual representations of a biological and an artificial neuron: (a) biological neuron and (b) artificial neuron mathematical model. Both images by Karpathy and Johnson (2016).

value is calculated by applying an activation function to the weighted sum, as shown in Equation 1.

$$f(x) = \sigma(\sum_{i=1}^{n} x_i w_i + b) = \sigma(x_1 w_1 + x_2 w_2 + ... + x_n w_n + b) \tag{1}$$

The most commonly used activation functions are represented on Figure 4. Down below is a short description about the pros and cons of each one of them.



(a) Sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$ (b) Tanh $\sigma(x) = tanh(x)$ (c) ReLU $\sigma(x) = max(0, x)$

Figure 4: Example of the most commonly used activation functions. (a) Sigmoid; (b) Hyperbolic tangent and (c) ReLU (Rectified Linear Unit). All images by Karpathy and Johnson (2016).

- **Sigmoid** outputs a value between 0 and 1. It saturates low input values closer to 0 and high input values near to 1. Still, this activation function has the drawback of not being zero centered, which means that the output will be always positive. So, the gradient of the on the weights will become all positive or negative, depending on its signal. This can produce some unstable dynamics when updating the weights by the gradient Karpathy and Johnson (2016); Buduma and Locascio (2015).

- **Hyperbolic tangent**, also know as *Tanh*, returns a value between -1 and 1. In the same way as the sigmoid activation function, the Tanh also saturates high and low input values. The difference is that for low input values the output will tend to -1 and for high inputs it will tend to 1. Once this function is zero-centered, this factor makes it more preferable than the sigmoid function by the reason enumerated on top Karpathy and Johnson (2016); Buduma and Locascio (2015).

- **ReLU** operates like a threshold at zero Karpathy and Johnson (2016). Negative input values are set to zero while the remaining values are left unchanged. Its computation simplicity accelerates training convergence considerably Krizhevsky et al. (2012). These characteristics made it very popular and preferable in most cases. However, the ReLU definition is fragile and many neurons can irreversibly die during training, in special, with high learning rates Karpathy and Johnson (2016). Its mathematical definition is on Equation 2.

- **Leaky ReLU** is a normal ReLU but with a slightly difference. It is intended to attenuate the dying ReLU problem referenced above. Instead of zero, the branch of negative input values, produces a small negative output by multiplying the input by a tiny constant (for example, $\alpha = 0.01$) Karpathy and Johnson (2016). The mathematical definition of Leaky ReLU is on Equation 3 and as it was already mentioned, differs from Equation 2 on the branch related to negative inputs.

$$\sigma(x) = \begin{cases} \boxed{0} & \text{if } x < 0 \\ x & \text{if } x >= 1 \end{cases} \tag{2}$$

$$\sigma(x) = \begin{cases} \boxed{\alpha x} & \text{if } x < 0 \\ x & \text{if } x >= 1 \end{cases} \tag{3}$$

- **Softmax** is an activation function that converts a set of values into a normalized probability vector that adds up to 1 Karpathy and Johnson (2016); Buduma and Locascio (2015). It is usually placed on the networks output layer in order to return a score or probability that associates a certain input to a specific class. Softmax function is expressed in Equation 4.

$$\sigma(x)_j = \frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_n}} \tag{4}$$

When alone, each artificial neuron is very limited but when several neurons are connected in a network and disposed by layers, they are able to learn and solve complex tasks Barker (2016).

2.2.2 *Artificial Neural Networks*

The term Artificial Neural Network (ANN) is used to characterize a mathematical model composed by a collection of artificial neurons, that are interlinked in a network by layers, in order to learn complex data relations collectively Buduma and Locascio (2015).

Feedforward Neural Network (FNN) and Recurrent Neural Network (RNN) stand as two types of ANNs. FNNs are characterized for being an acyclic graph where the data moves only forward. On the other hand, on RNNs the information can flow in any direction, including to the same layer. So, RNNs are an extended representation of FNNs with the addition of feedback connections. Hafemann (2014); Gravelines (2014); Schmidhuber (2014); Buduma and Locascio (2015); Goodfellow et al. (2016).

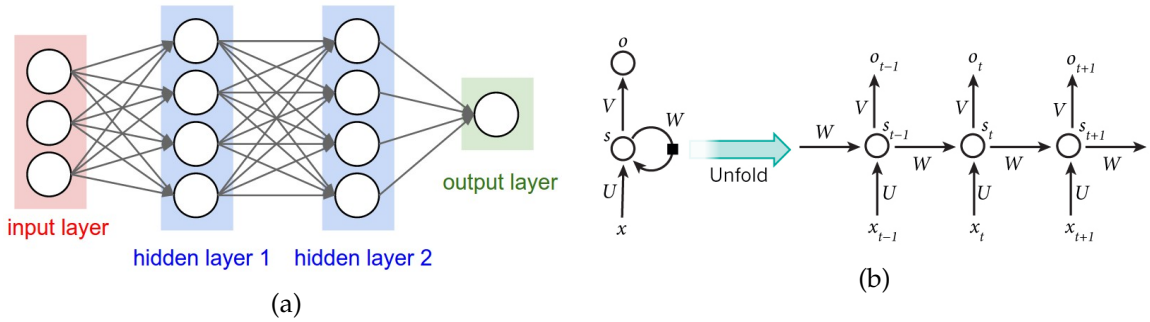


Figure 5: Example of two types of artificial neural networks: (a) feed-forward network with three inputs neurons, two hidden layers (both with four neurons) and an output layer with a single neuron; and (b) recurrent neural net unfolded in a time sequence. Images by Karpathy and Johnson (2016) and Britz (2015), respectively.

Just like Figure 5(a) suggests, traditional FNN models can be disposed into fully-connected layers of neurons and the data moves solely forward. In these layers, only neurons between two layers are fully pairwise connected. So, neurons within a single layer don't have connections between themselves Karpathy and Johnson (2016).

Traditional FNNs are structured in three sections:

- An input layer that receives the data to be learned. The data needs to be in a vectorized representation, for instance to deal with images, they must be converted to a 1-dimensional representation, just like a vector.
- One or several hidden layers composed by artificial neurons with the intent of extracting non-linear features from the input data.
- An output layer that combines the set of non-linear features learned on the previous layers and outputs a score/prediction using a softmax activation function Dettmers (2015a).

The RNNs are meant for processing sequential data Goodfellow et al. (2016); Britz (2015); Buduma and Locascio (2015). The term recurrent refers to their ability to execute the same job for every element of a sequence Britz (2015). Figure 5(b) shows that, for each time stamp (t), the hidden state (s_t) is calculated by adding the current input (Ux_t) with the previous hidden state (Ws_{t-1}), where f is an activation function. Equation 5 translates the statement presented previously.

$$s_t = f(Ux_t + Ws_{t-1}) \tag{5}$$

For $t = 0$, s_{-1} is normally initialized with zero. The o_t is the output result at step t. For example, in a sentence prediction, the operation $o_t = softmax(Vs_t)$ would infer which is the most probable next word Britz (2015).

In the context supervised learning, on both types of ANN the error between the network predictions and the ground truth is iteratively back-propagated through the network and it is used to update the network weights so that predictions can get more accurate Dettmers (2015a).

2.2.3 *Convolutional Neural Networks*

As previously presented, ANNs are composed by sets of artificial neurons organized into layers such as input, hidden and output layers. CNNs follow the same rules (as they derive from ANN) but they differ in one simple thing: they assume that the input are images. Hence, the architecture of a CNN disposes the neurons in a different and more efficient way in order to create an improved model Karpathy and Johnson (2016). Neurons are disposed in three dimensions: width, height and depth, just like a volume Karpathy and Johnson (2016); Buduma and Locascio (2015). The idea behind CNNs is to use small ANNs which are convoluted along the image to extract relevant local information. This idea differs from a fully connected architecture which demands much more parameters.

For example, a single fully-connected hidden layer would require $120,000$ parameters (or weights) in order to handle a 200x200 RGB image (200x200x3 $= 120,000$). Such amount of parameters is wasteful and it would lead the model to easily overfit on the training data Karpathy and Johnson (2016); Buduma and Locascio (2015). Besides, the fully-connect approach relates all image pixels together. In reality, features are usually locally related. By using a smaller convolutional network, it is possible to identify local features and by adding successive convolutions layers relate the local information from different networks to achieve a global classifier. Figure 6 suggests a better understanding of how the neurons are set on a CNN in comparison with the traditional ANN.

The concept of receptive fields is the key to ensure that neurons inside a layer are only connected to a small region of the previous layers, avoiding the wastefulness of fully-connected

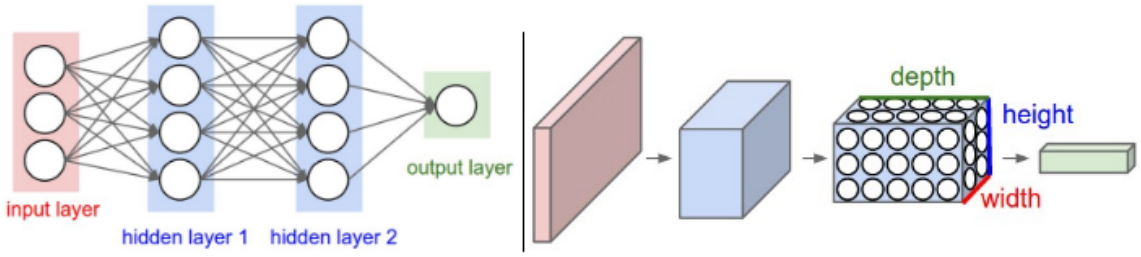


Figure 6: Representation of a traditional artificial neural network and a convolutional neural network. Image by Karpathy and Johnson (2016).

neurons. This allows to explore a local connectivity pattern among the neurons of adjacent layers and progressively assemble small representations on larger areas. The weight sharing is done by replicating filters upon the whole visual field. Each filter results in a feature map that is stacked in the output volume. When combined, these properties make CNNs capable of generalization on vision problems and help reducing the number of parameters needed to be learned as well as the amount of memory required Karpathy and Johnson (2016). Typically, a CNN is structured in two main sections, as Figure 7 illustrates. Firstly, comes the feature extraction and then the classification process.

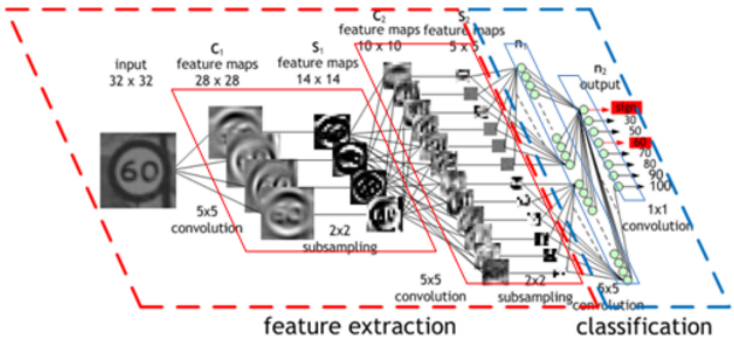


Figure 7: Typical structure of a convolutional neural network. Image by Maurice Peemen.

The main goals of these two sections are:

1. **Feature extraction** - Feature extraction has an important role on CNNs because it is responsible for learning the patterns that belong to each class. They learn automatically how to extract the most relevant features and, as the network grows deeper, the more complex features are extracted. For example, take a look on Figure 8 where it is shown two levels of features Dettmers (2015a).

   This first section of the CNN is computationally expensive essentially due to convolutional layers. Activation and pooling layers are also frequently used on it, however,

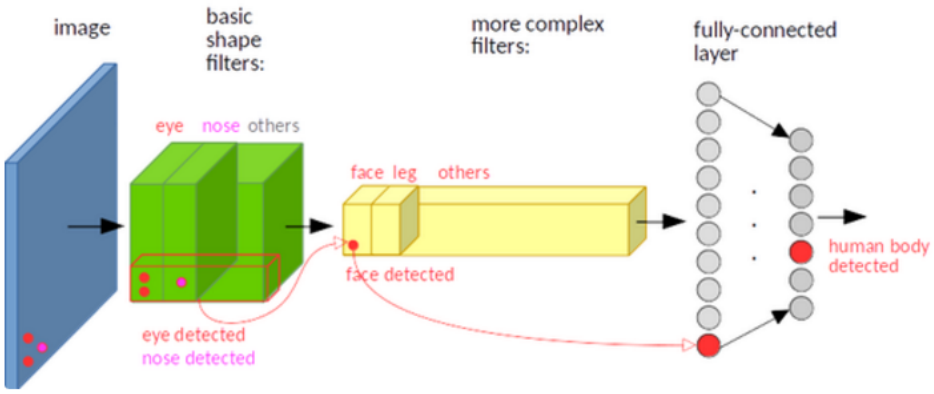


Figure 8: Hierarchical features extracted from a convolutional neural network. Image from a Quora question about how CNNs work.

their computation is not as heavy as the convolutional layers. Below, the typical layers used in the feature extraction phase will be explained in more detail.

2. **Classification** - This section receives the high-level features extracted on the previous step and looks for those which strongly correlate to a particular class. It can be composed by one, or more, fully connected layers as well as some dropout layers in between them. In the end of these layers, a softmax function is used so that the resulting vector is transformed into a probability vector where each element is the probability associated with a specific class Karn (2016).

As mention before CNNs can have several different types of layers. Next a description is provided about the most common layers.

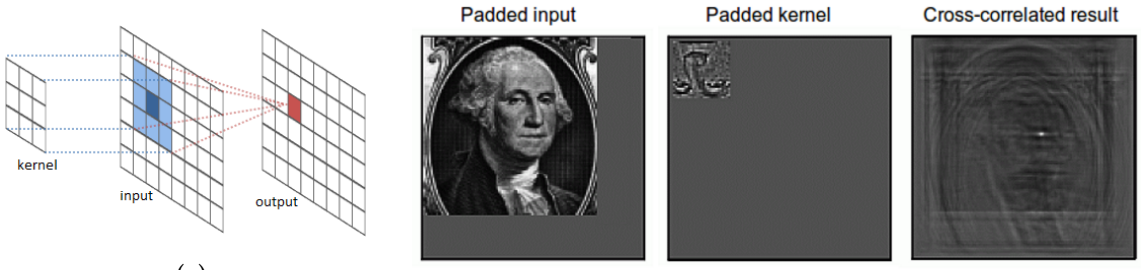
Convolutional layers

Convolutional layers represent the base element of CNNs. They are responsible for computing several convolutions, where each one has a specific filter (or kernel) that is meant to be learned. A convolution is a mathematical operation that aggregates two types of information:

1. An input feature map, that can be a volume or a raw image.
2. A filter or kernel.

When both are combined, a new feature map its created that highlights where the kernel features were found. The most correct interpretation for a convolution is a cross-correlation once the convolutional filter works as a feature detector, i.e., if an input image produces a big output feature map after being filtered by a certain kernel, it means that the information

of that kernel is present on the image Dettmers (2015a); Buduma and Locascio (2015). Look at the example on Figure 9.



(a) (b)

Figure 9: Example of applying a convolution on an image: (a) How it is calculated and (b) Output feature map. Images by Colah's blogpost and Dettmers (2015a), respectively.

Nevertheless, there are a few more properties regarding the convolutional layers that need to be taken into account. As previously said, the **local connectivity** is essential for dealing with high-dimensional inputs, such as images Karpathy and Johnson (2016). It doesn't make sense to fully connect all neurons to all pixels in an image, because nearby pixels have a higher probability of being correlated than those that are far apart. So, each neuron is only connected to a small receptive field that has the same size of the applied filter, just like Figure 10 shows.

(a) (b)

Figure 10: Concept of the receptive field used on convolutional neural networks: (a) Receptive field for a convolutional layer with 5 5x5 filters and (b) Demonstration of how nearby pixels tend to be more correlated. Images by Karpathy and Johnson (2016) and Nervana's video lesson about CNNs, respectively.

The convolution operation is controlled by three parameters: depth, stride and zero-padding Karpathy and Johnson (2016). The output volume spatial arrangement will be based on them.

- **Depth** - The depth of an output volume is equal to the number of convolutional filters. Each slice contains the feature map for one particular kernel.
- **Stride** - Stride handles how receptive fields overlap each other, hence, it determines the spatial dimensions of the output volume. For example, a big stride means less overlap between receptive fields, resulting on a smaller output volume and vice-versa.
- **Zero-padding** - Zero-padding is sometimes conveniently used to preserve input volume spatial size. By adding zeros to the volume borders, kernels can be applied even if they "get out" of the volume dimensions. On Figure 11 the "valid" padding corresponds to the zero-padding taking zero value. In turn, the "same" padding means one zero-padding border around all the volume since the convolutional layer has 3x3 kernels. "Valid" and "same" padding are two Tensorflow Abadi et al. (2015) conventions.

Figure 11 summarizes graphically the notions presented for a 28x28x1 input volume on a 8 3x3 filters convolutional layer.

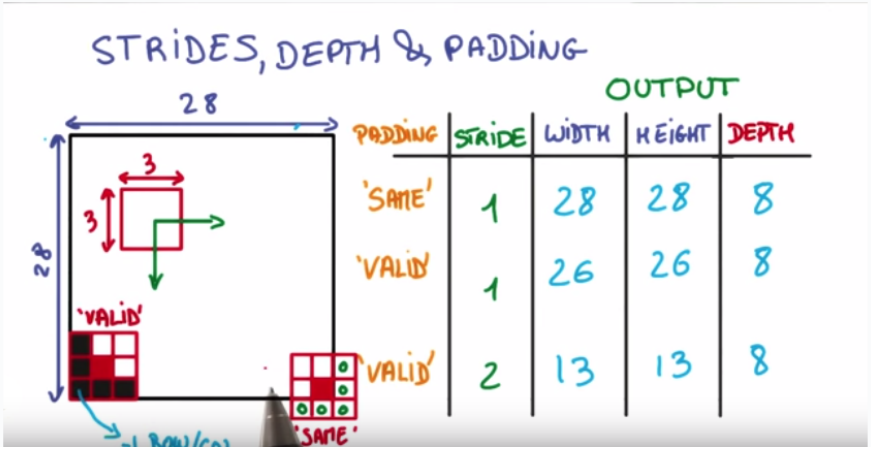


Figure 11: Output volumes spatial arrangement variation according to depth, stride and zero-padding. Image by Vanhoucke.

For a squared input volume sized I, a filter size F, a stride S and a zero-padding P, the size of the output volume O is given by the following equation (note that, in case of a fractional result, it is only considered the unit part):

$$O = \frac{(I - F + 2P)}{S} + 1 \tag{6}$$

Weight sharing enables translation invariance once it's assumed that it can be useful to compute one feature on two distinct spatial positions. Moreover, it helps controlling the number of free parameters by sharing the neuron's weight and biases required memory in every depth slice Karpathy and Johnson (2016).

Activation layers

Activation layers are meant to add non-linearity properties to the data. They keep the volume unchanged and perform an element-wise fixed mathematical operation Karpathy and Johnson (2016). On Figure 4 are graphically represented the most frequently used activation functions on CNNs, likewise on ANNs. From all functions presented on Figure 4, ReLU layer is the simplest and hence makes the training faster without compromising the network's generalization Krizhevsky et al. (2012).

Pooling layers

Pooling layers are usually inserted in-between convolutional layers and they perform down-sampling along the spatial dimensions (width and height only; depth remains untouched). They shrink a certain area to a single value. So, a larger aggregation area condenses more information. This procedure brings benefits in many aspects because:

1. It improves the computational performance by reducing the number of parameters through down-sampling.
2. Down-sampling also contributes for less memory required, which is helpful to fit inputs on GPUs (limited RAM) in order to accelerate the training operation.
3. Still, the parameter reduction helps preventing overfitting.
4. It provides a form of local translation invariance. Even if the inputs shifts a little bit, the output will remain constant.

Nevertheless, a large pooling area may result on data loss. Figure 12 shows an example of a max pooling operation. On a average pooling operation it would, instead, calculate the average value covered by the filter Karpathy and Johnson (2016); Dettmers (2015a); Buduma and Locascio (2015).

The two most common pooling layer parameters are the 2x2 filter with stride 2 and the 3x3 filter with stride 2 Buduma and Locascio (2015).

Fully-connected layers

As the name suggests, the neurons of a fully connected layer are connected to all previous layer's outputs. Just like it was discussed on the classification item, on CNNs they are placed right after the feature extraction. CNNs output layer is also a fully-connected layer that has as many neurons as the number of classes. The main difference, is that it uses a softmax activation function to ensure a normalized probabilistic output, in which, the predicted class is the one with highest probability. On Figure 5(a) is presented a feed-forward neural network composed by two fully-connected hidden layers.

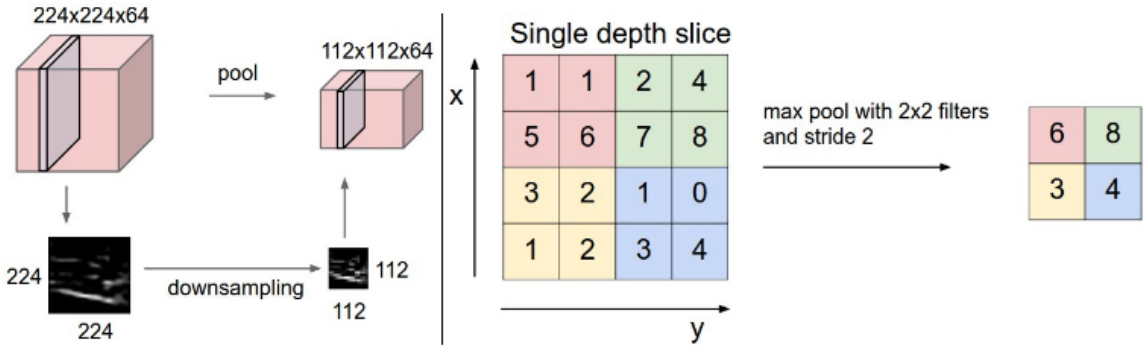


Figure 12: Example of a max pooling operation using a 2x2 kernel with stride 2. Image by Karpathy and Johnson (2016).

Dropout layers

Dropout layers are also used to prevent overfitting. This regularization method is only applied on training and randomly discards some nodes according to a probability. So, it helps decreasing the time spent on training too. It is meant to avoid that well trained neurons, with high activation values, influence neurons nearby. In other words, it avoids units from co-adapting too much and be influenced by highly activated neurons Srivastava et al. (2014); Wager et al. (2014). Check Figure 13 for a better explanation.

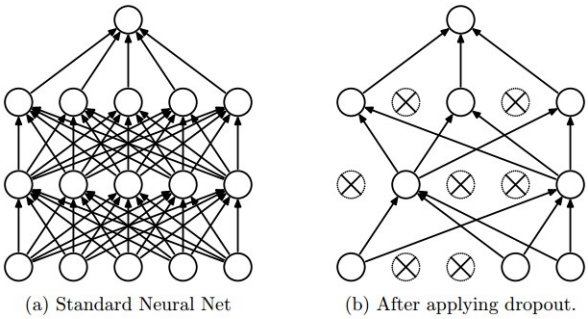


Figure 13: Illustration of how dropouts works. Image by Karpathy and Johnson (2016).

In fact, besides preventing overfitting, dropout also represents an efficient way of combining many different neural networks since each node discarded can represent a distinct network topology Srivastava et al. (2014).

Normalization layers

Although normalization layers have fallen out due to its minimal contribution for training, they were originally intended to accelerate the training and make the trained model more robust to variations Karpathy and Johnson (2016); Buduma and Locascio (2015).

1. The **batch normalization** fixes the means and variances of layers input. It reduces the dependency on the parameters initialization (less careful initialization) and allows the use of higher learning rates Buduma and Locascio (2015). In some cases, it also reduces the need of dropout layers too. Besides that, the training can taker shorter while maintaining the same accuracy level Ioffe and Szegedy (2015).

2. **Local response normalization** layers (LRN) are useful to deal with unbounded activations, like it happens on ReLU activation functions. Once detected a high-frequency feature, it uniformly damps large responses in a local neighborhood. This type of regularizer encourages "competition" for big activities among nearby groups of neurons Krizhevsky (2014).

3. **L1 and L2 normalization** add an extra term to the cost function that penalizes large weights forcing the network to prefer smaller weights Nielsen.

Inception modules

Inception modules were inspired by the indecision of which convolution would be the best: 1x1, 3x3 or a 5x5? Vanhoucke. The idea of inception modules is to concatenate multiple convolution and pooling filters on the same input. With it, the model is then able to learn multi-level features simultaneously through smaller (1x1), medium (3x3) and larger (5x5) filters Szegedy et al. (2014).

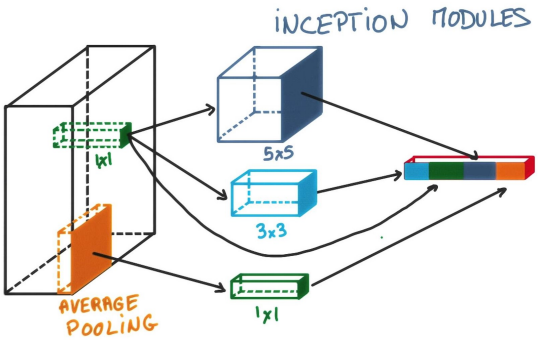


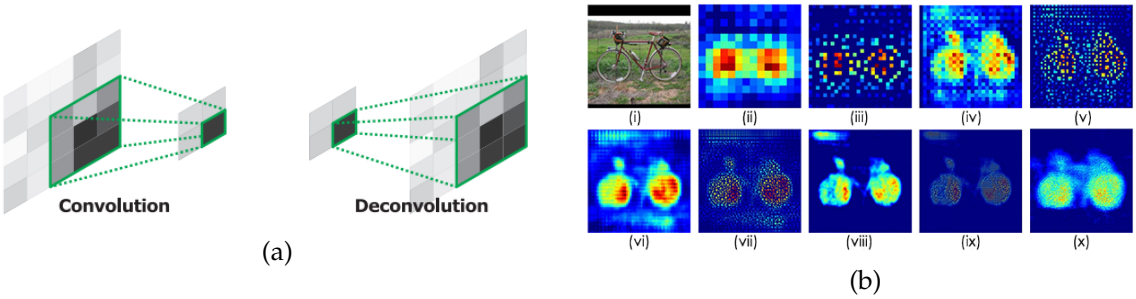
Figure 14: Example of an inception module. Image by Vanhoucke.

The output of an inception module is a big feature map which is then fed into the next layer which can be, eventually, another inception module Dettmers (2015a).

Deconvolutional layers

The deconvolutional layers, also known as the transposed convolutional layers, do the inverse operation of the convolution layers Gao et al. (2017); Odena et al. (2016). Instead, they learn filters in order to reconstruct the shape of an object. Therefore, they associate a

single input to multiple outputs Noh et al. (2015). Stacked deconvolutional layers produce a hierarchical structure of features too. Lower deconvolutional layers tend to capture the overall shape of the object while higher deconvolutional layers tend to capture class-specific (finer) details Noh et al. (2015). Due to its properties, these layers are commonly used for semantic segmentation on encoder-decoder architectures in order to perform feature map up-sampling Gao et al. (2017).



(a) (b)

Figure 15: Visual representation deconvolutional layers: (a) Comparison with convolutional layer and (b) Output feature maps from lower level to higher level layers (from (ii) to (x)) . Images by Noh et al. (2015).

Figure 15(a) suggests that deconvolutional layers can produce larger feature maps from smaller feature maps. However, this process adds some checkerboard artifacts to the output feature map as well Gao et al. (2017); Zeiler et al. (2010); Odena et al. (2016). In turn, Figure 15(b) indicates that the deconvolutional layer output feature maps get finer and more evident along with increase of the number of deconvolutional layers.

Residual Blocks

Residual blocks are the base element of Residual Neural Networks He et al. (2015b). As Figure 16 points out, the output of a residual block is an element-wise addition among its input (X) and a transformation of the input ($F(X)$). This requires that X and the output of $F(X)$ must have exactly the same size.

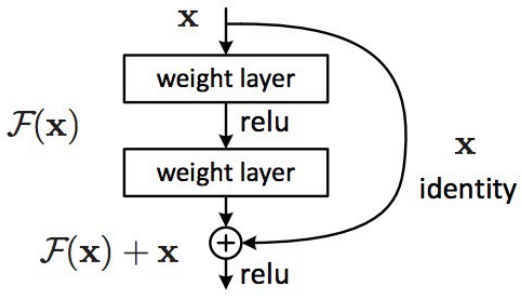


Figure 16: Exemplary of a residual block. Image by He et al. (2015b).

The transformation $F(X)$ represents a sequence of convolutional layers (two or more) interspersed by activation functions He et al. (2015b). This operation bypasses the input and adds the features learned on the current layer to it so the next layer/block can learn from both.

2.3 State Of The Art Network Architectures And Benchmarks

Convolutional Neural Networks represent the state of the art for image recognition Simonyan and Zisserman (2014); Russakovsky et al. (2015); He et al. (2015b); Zhang et al. (2017). Although its complexity in terms of computational load and memory usage, CNNs are robust enough to deal with data variance such as illumination, occlusion, rotation and scale Achatz (2016). Even so, they are also capable of understanding the essential features that distinguish one class from the others.

To train a CNN based image classifier are needed a dataset and a network architecture. So, in the next sections will be presented the state of the art network architectures as well as benchmarks on well known datasets among image recognition tasks.

2.3.1 *State of the Art Network Architectures*

The CNN architectures presented underneath established relevant achievements over the state of the art for image recognition in the last years. They delineate the evolution of CNNs and suggest good practises for building tough network architectures Dettmers (2015b); Culurciello (2017); Kurenkov (2015).

AlexNet

The year of 2012 was a turning point for the adoption of feature learning approaches instead of feature engineering. Besides, this year was marked by the affirmation of deep learning against traditional computer vision methods. Dettmers (2015b); Krizhevsky et al. (2012) Alex Krizhevsky, along with Ilya Sutskever and Geoffrey Hinton developed a network architecture composed by rectified linear units (ReLU) for higher performance (since they compute faster than *tanh*) and dropout regularization to improve its generalization capacity Dettmers (2015b); Culurciello (2017).

Its architecture includes five convolutional layers (some of them followed by overlapping max-pooling layers) and three fully-connected layers interleaved by dropout layers at its end Krizhevsky et al. (2012). Additionally, local response normalization (LRN) layers were used because of the unbounded ReLUs activation functions. The large size of the network in terms of parameters could lead to overfitting Krizhevsky et al. (2012). In order to avoid overfitting at all costs, both dropout layers and data augmentation techniques were used.

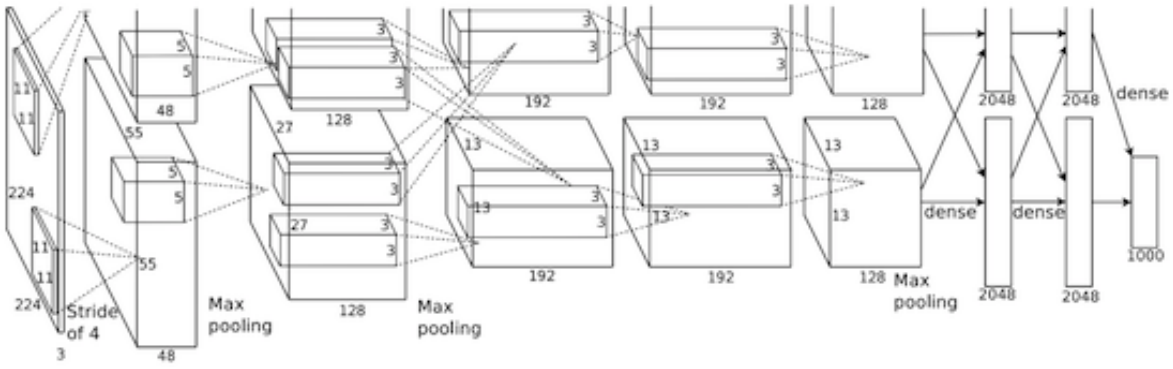


Figure 17: Alexnet network architecture. Image from Krizhevsky et al. (2012).

Dropout layers prevent overfitting by forcing the network to learn even in the absence of some neurons while data augmentation techniques artificially generate extra data in order to get an enlarged dataset Culurciello (2017); Krizhevsky et al. (2012). These procedures contibute for a more generic trained model. Table 1 has a detailed description of the Alexnet architecture.

Table 1: Detailed description of Alexnet architecture.

Layer	**Filters**	**Filter size**	**Stride**	**Activation**
Convolutional	96	11x11	4x4	ReLU
Max Pooling	1	3x3	2x2	-
Local Response Normalization				
Convolutional	256	5x5	1x1	ReLU
Max Pooling	1	3x3	2x2	-
Local Response Normalization				
Convolutional	384	3x3	1x1	ReLU
Convolutional	384	3x3	1x1	ReLU
Convolutional	256	3x3	1x1	ReLU
Max Pooling	1	3x3	2x2	-
Local Response Normalization				
Fully-Connected	4096			TanH
Dropout	0.5			
Fully-Connected	4096			TanH
Dropout	0.5			
Fully-Connected	nclasses			Softmax

As Table 1 points out, the filter size gets smaller as the the network gets deeper. The first convolutional layer is the only on with stride 4 because it is the distance between the receptive field centers of neighboring neurons in a kernel map Krizhevsky et al. (2012). The LRN layers appear on each block of convolutional layers right after max pooling layers .

The success of AlexNet on ILSVRC-2010 and ILSVRC-2012 encouraged many researchers to take CNN-based approaches in the following years. So, in 2013 two more CNNs were proposed: ZFNet Zeiler and Fergus (2013), that derived from Alexnet and the Network in Network (NiN) Lin et al. (2014).

ZFNet

The first two initials of ZFNet came from the authors surname (Zeiler and Fergus) Zeiler and Fergus (2013). This network is more than a fine tuning of AlexNet. Even though it has a similar architecture, it introduced some new concepts that helped improving the performance of the CNNs. In the first layer, Alexnet has 11x11 filters with stride 4. On ZFnet, 7x7 filters with stride 2 are used instead. This modification was made because smaller filters on the first layer tend to retain more original pixel information. On the other hand, 11x11 filters with stride 4 may skip relevant information. Still, the respective paper Zeiler and Fergus (2013) explains how it is possible to visualize the network weights through deconvolution 2.2.3. It allows the visualization of the features learn by the network, which can be used to tune the network more carefully. Moreover, this visualization helped understand that as the model grows deeper, more complex features are learned. Figure 18 describes the ZFnet architecture and as it can be seen, it is very similar to Alexnet architecture. The main differences are on the first convolutional layer (which has a lower filter size and stride) and in the addition of a deconvolutional network to visualize the network weights.

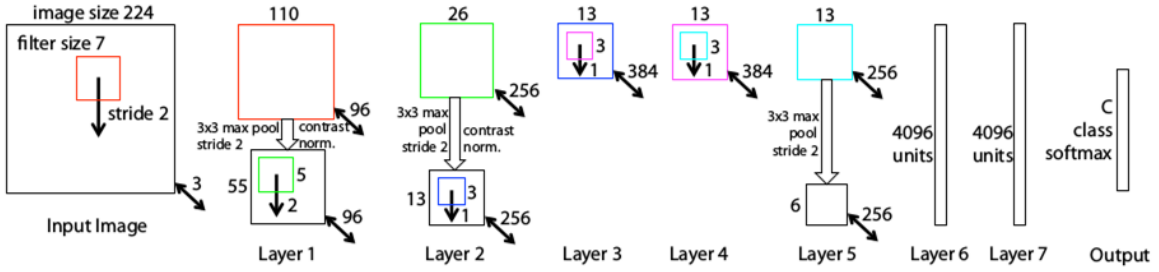


Figure 18: ZFnet network architecture. Image by Zeiler and Fergus (2013).

In fact, the model visualization supports the idea that the learned features assume a hierarchy (from low-level to high-level features) and that the network performance tends to improve as long as it gets deeper. Hereupon, besides winning ILSVRC-2013, ZFNet contributed not only on performance improvements on CNNs but also with a visualization method that gives great intuition and insight about which features CNNs learn.

Network in Network

The Network in Network architecture was built with the intent of better combine features before feeding them to another layer. It has the particularity of using spatial multi-layer perceptron layers in form of 1x1 convolutions, referred by the authors as *mlpconv* Lin et al. (2014). Although it may not make much sense, in the context of CNNs, 1x1 convolutions can actually reduce the number of network parameters and represent an universal function approximator Lin et al. (2014). For example, a 20 1x1 convolutional layer converts a 200x200 input with 50 features (200x200x50) into the size of 200x200x20.

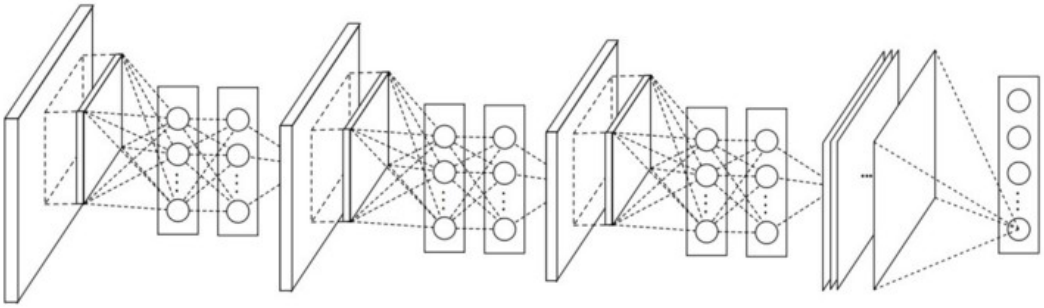


Figure 19: Network in Network network architecture with *mlpconvs* between convolutional layers. Image by Lin et al. (2014).

The network is composed by three *mlpconv* layers with ReLU activaction functions. All *mlpconv* layers are followed by a overlapping max polling layer, except for the last one, which has an global average pooling instead. This layer outputs a vector with size equal to the number of features maps. Each element of the vector contains the average value of one input feature map. It is meant to replace the usual fully connected layers. Here are some of the reason why global average pooling is preferable Lin et al. (2014):

1. It enforces correspondences among features maps and categories. This is more native to the convolution structure. For instance, feature maps can be easily interpreted as categories confidence maps.

2. Avoids overfitting since it doesn't have any parameter to optimize.

3. It is more robust to spatial translations because it sums out the spatial information.

So, there is only a single fully connected layer that has the softmax activation function in order to output the network predictions. Table 2 shows the network performance on the datasets CIFAR-10 Krizhevsky (2009), CIFAR-100 Krizhevsky (2009), Street View House Numbers (SVHN) Netzer et al. (2011), and MNIST LeCun et al.. Its performance is compared with the state of the art performances at the time this architecture came out.

Table 2: Network in Network architecture best performance on CIFAR-10, CIFAR-100, SVHN and MNIST datasets compared with the state of the art at the time Lin et al. (2014).

	CIFAR-10	CIFAR-100	SVHN	MNIST
SoA	9.32%	36.85%	1.94%	0.45%
NiN	8.81%	35.68%	2.35%	0.47%

Further research took advantage of these ideas and applied it into more recent networks architectures, such as the GoogLeNet Szegedy et al. (2014) and the Redisual Neural Networks He et al. (2015b). These ideas were essential for enabling deeper networks that don't compromise inference times.

VGG

The VGG (stands for Visual Geometry Group) architecture appeared on year 2014 and it is characterized by two words: simplicity and depth. It enforces the idea that deeper (16 to 19 layers) architectures with small filters (3x3 is the smallest filter size to capture shifts) and stride 1 lead to a significant improvement on the prior-art configurations Simonyan and Zisserman (2014). Furthermore, deeper networks have a superior hierarchical representation of the learned features. Because of its depth, and in order to balance the network weight, small filters were used to help reducing the number of parameters and consequently the training time. This network also gives a great insight that a sequence of multiple 3x3 convolutions can simulate the same effect of larger filters, like 5x5 or 7x7 filters.

As Figure 20 shows, the VGG network architecture configurations follow a generic design and differ in depth Simonyan and Zisserman (2014). The networks are composed by stacks of 3x3 convolutional layers with stride 1 and zero padding in order to preserve the spatial resolution. Max pooling layers appear in-between blocks of convolutional layers and down sample the features maps through 2x2 filters with stride 2. Figure 20(c) is the only VGG configuration that uses 1x1 convolutional layers. They can be seen as a linear transformation of the input channels Simonyan and Zisserman (2014). In addition, there is also a single architecture that makes use of LRN layers but the authors claim that they didn't bring any improvements on performance. In fact, LRN contribute for a higher usage of memory and an increase on the computation time. At the end, are disposed three fully connected layers with 4096, 4096 and *nclasses* hidden units. All layers use ReLU activation functions except the last fully connected layer which uses softmax. One throwback of these networks are the size of the trained models Canziani et al. (2017). The large number of parameters makes a trained model of round the 500MB for 224x224x3 input images.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 2: **Number of parameters** (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

Figure 20: VGG network architecture configurations. From the left (a) to the right (e) the network depth increases (more layers), all with stride 1. The convolutional layers are denoted as conv <receptive field size>-<number of channels>. For simplicity, ReLU activation functions are not shown. Image by Simonyan and Zisserman (2014).

GoogLeNet

The GoogLeNet network architecture was built with focus on efficiency not only for traditional computers but also thinking on mobile and embedded computing. To achieve that, reducing the number of parameters was a crucial restriction Szegedy et al. (2014). Here, the term deep refers to the new network organization through the introduction of the inception modules and the network depth itself. So, this architecture changed a lot compared to the typical network structure, which is based on stacked blocks with one or more convolutional layers, followed by a activation function and a max pooling operation. Instead,

it stacks several inception modules. These modules aggregate miscellaneous information once they contain distinct operations that are done in parallel Szegedy et al. (2014).

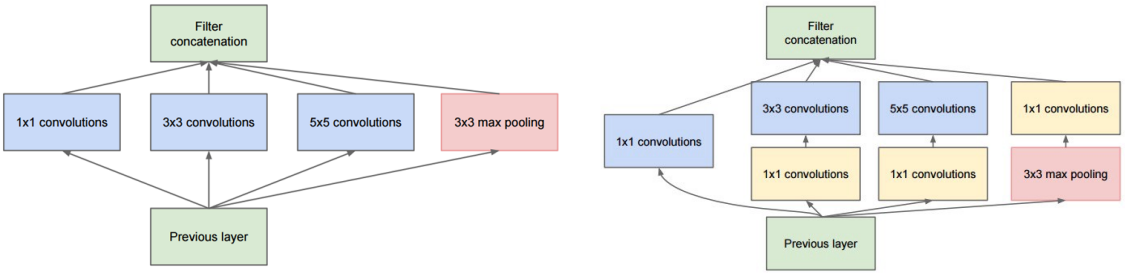


Figure 21: Inception module structure: (a) concept of inception module and (b) example of an inception module and how it reduces the total number of parameters by using 1x1 convolutions. Image by Szegedy et al. (2014).

As shown in Figure 21, inception modules can be composed by several convolutional layers and even max or average pooling operations. Therefore, these layers can collectively extract a diverse levels of information:

1. 1x1 convolutions can capture fine grain details and simultaneously decrease the dimension of model which is essential to get it deeper. Lin et al. (2014); Szegedy et al. (2014).

2. 3x3 filters were shown to be very effective when disposed is mass Simonyan and Zisserman (2014). Its impact is still diminished by prior 1x1 convolution.

3. Higher receptive fields, such as 5x5 filters, extract bigger information and in the same way, are amortized by preceding 1x1 convolution.

4. Max pooling layers are responsible for reducing the volumes spatial size and help prevent overfitting.

5. The ReLU activation layers come after each convolutional layer and add non-linearity to the network.

Although all this complexity (a total of 27 layers), the balance of the network performance and computational efficiency is very good Canziani et al. (2017).

From the original GoogLeNet represented on Figure 22, derived the Inception-vx, $x \in \{2, 3, 4\}$, architectures that were able to reach even lower error rates. In fact, on those newer version, inception modules were re-adapted in order to became more efficient Szegedy et al. (2015). On Appendix A.2.1 is presented the network architecture of Figure 22 yet in a higher resolution, for a better understanding of its components.

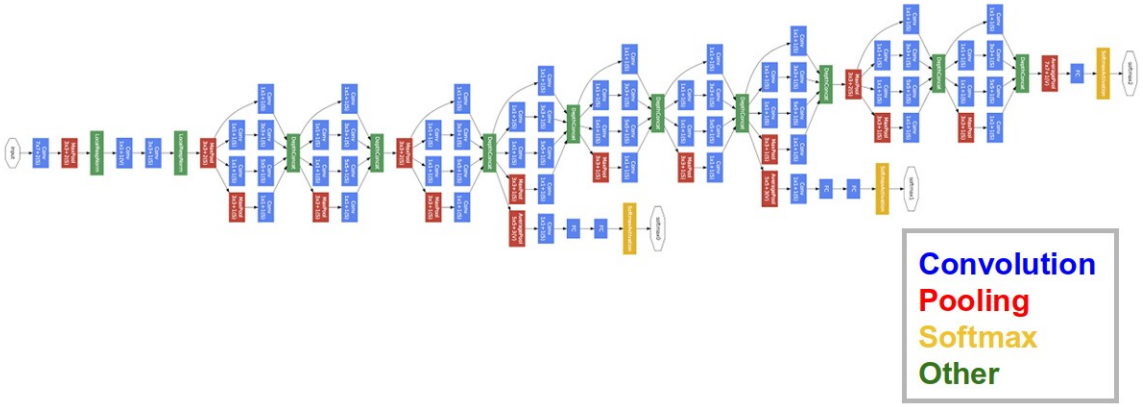


Figure 22: GoogLeNet network architecture. Image by Szegedy et al. (2014).

Once again, more recent architectures used the concepts stated soo far on newer experiments. On 2015, one more network architecture arised and established a new baseline for CNN capabilities. It is the Residual Neural Network from Microsoft Research He et al. (2015b).

Residual Neural Networks

Nearly at the same time as the Inception-v3 network came out, the ResNet (abbreviation for Residual Neural Networks) were announced. The idea behind the Residual Neural Networks is to feed to the next block (or layer) the output of two or more convolutional layers, interleaved by activation functions, and the input of the residual block bypassing it He et al. (2015b). It was also tested with just a single layer bypass instead two or three however it didn't bring considerable improvements He et al. (2015b).

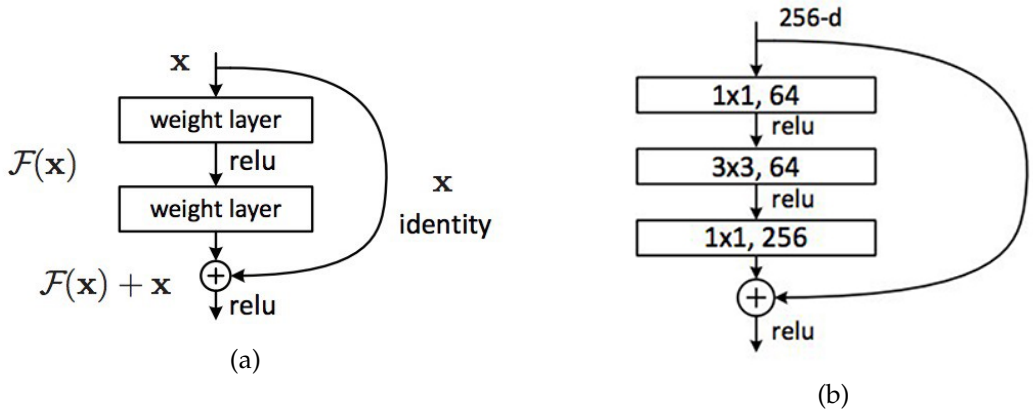


Figure 23: Two examples of a residual block: (a) illustration of a residual block and (b) practical example of a residual block. Images by He et al. (2015b).

Figure 23(a) shows that the output of a residual block is the input plus a transformation of it. It looks similar to computing a *delta* or a slightly change to the original input. In order to compute the element-wise addition their size must be equal He et al. (2015b). On traditional CNNs, x would represent the input while $F(x)$ would represent the output of a convolutional layer. In turn, on Figure 23(b) is presented a practical situation that exemplifies the functionality of a residual block. So, given an 256-dimensional input, the number of features at each layer is reduced to 64, first by a 1x1 convolution, followed by a 3x3 convolution and at last, it restores the original input dimension through a 1x1 in order to add it with the input. Identically to what happens on inception modules, this operation provides a rich combination of features and simultaneously guarantees low computation impact.

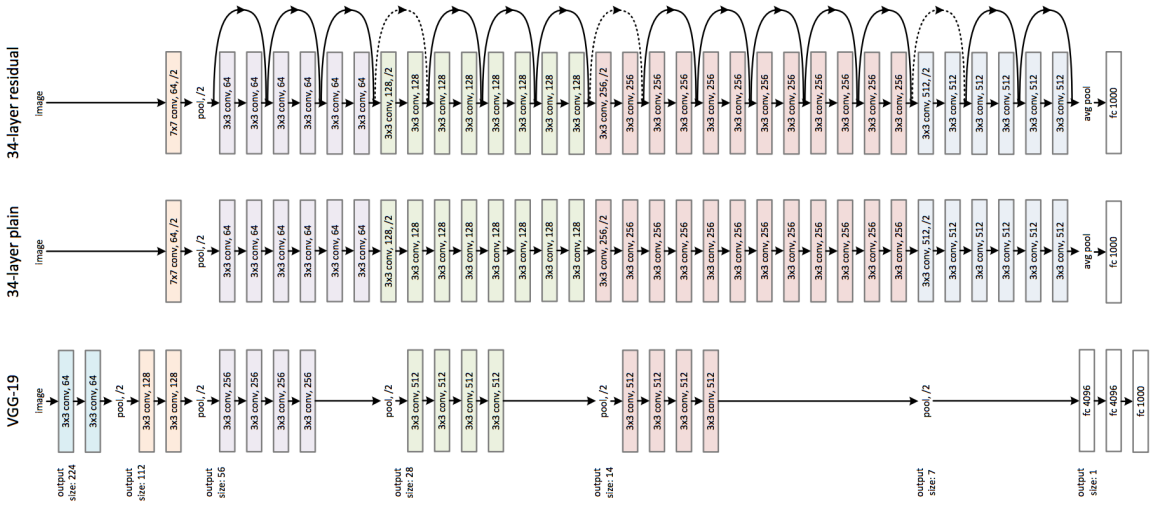


Figure 24: Comparison of VGG-19 architecture (bottom) with two 34-layers networks: one plain (middle) and other residual (top). Images by He et al. (2015b).

Figure 24 compares the VGG-19 architecture with two 34-layer networks along the spatial size. It shows an increasing gap on VGG-19 as the network spatial size gets smaller. Besides that, the initial layers of the 34-layer networks have 7x7 convolutional filters and after entering another block of convolutional layers, the spatial size is already on 56x56. The fully-connected output layer with softmax activation function is preceded by an average pooling layer.

Furthermore, it was also trained a network with 1000 layers, yet the results weren't convincing at all since the trained model was unnecessarily large and even so its performance a slightly worse than a network with 110 layers He et al. (2015b). So, the experiences around this architecture indicate that there is a top-limit on the networks performance when stack-

ing residual blocks. In other words, stacking residual blocks can effectively improve the network capabilities however, just up to a certain point He et al. (2015b).

2.3.2 *Benchmarks*

The benchmarks section is intended to expose the current state on recognizing well know datasets such as the MNIST LeCun et al., CIFAR-10 Krizhevsky (2009) and the subset of the ImageNet Russakovsky et al. (2015). The first two are commonly used on tutorials to teach people how CNNs work. By the other hand, the subset of the ImageNet is a large scale dataset which promotes an annual competition called ImageNet Large Scale Visual Recognition Challenge (ILSVRC) Russakovsky et al. (2015); Krizhevsky et al. (2012). This competition stands as one of the most challenging image recognition tasks. Most of the state of the art networks presented on the previous section were built for that purpose. Their success turn them on a reference to anyone that pretends to make a CNN-based image classifier.

MNIST dataset

MNIST is a balanced database composed by 70,000 gray-scale images of handwritten digits LeCun et al. (from 0 to 9). The recognition of MNIST is very often used as the *"Hello World!"* of convolutional neural networks.

Currently, the recognition performance on MNIST has an error rate of 0.21% Wan et al. (2013). The state of the art approach uses a CNN architecture based on the one presented in Bottou et al. (1998). In addition, it applies a similar method to the dropout regularizer on fully-connected layers. Instead of the common dropout which discards a set neurons activation, this method discards a subset of network connections randomly, turning the network sparsely connected Wan et al. (2013), like Figure 25(c) suggests.

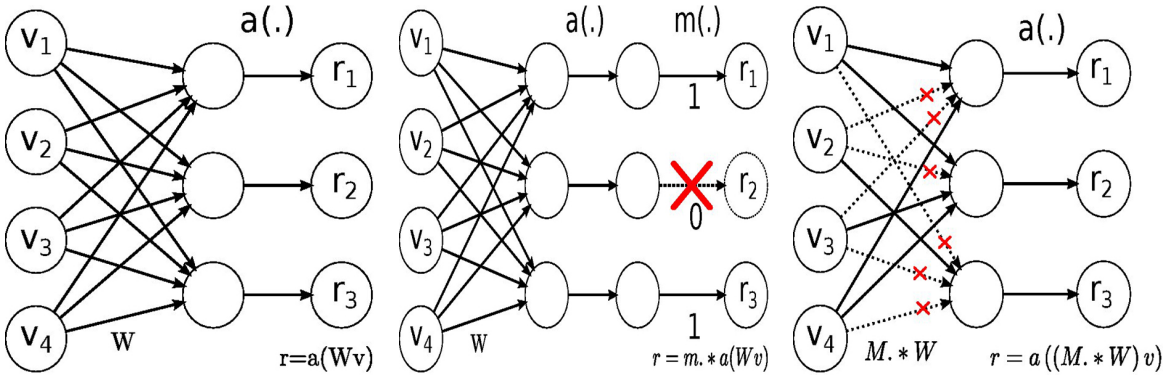
CIFAR-10 dataset

In turn, the CIFAR-10 Krizhevsky (2009) dataset contains 60,000 RGB images equally separated into ten miscellaneous classes. Likewise the MNIST, CIFAR-10 is frequently used on tutorials for introducing how CNNs work.

The best performance on CIFAR-10 achieved an error rate of 2.86% and relies on a residual neural networks architecture, as the paper Gastaldi (2017) refers.

ImageNet Large Scale Visual Recognition Challenge

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is an annual competition that evaluates algorithms for object detection and image classification at large scale on



(a) (b) (c)

Figure 25: Illustration of how drop-connect works: (a) without dropout and drop-connect; (b) with dropout, represented by function $m(.)$ and (c) with drop-connect. Images by Wan et al. (2013).

a subset of the ImageNet dataset. The subset of the ImageNet dataset is composed by $1,261,406$ training images categorized into $1,000$ classes[1], $50,000$ validation images and $150,000$ testing images Russakovsky et al. (2015); Canziani et al. (2017); Krizhevsky et al. (2012). It is one of the biggest challenges of computer vision and machine learning and it represents an important reference, as well as a significant benchmark, for image recognition state of the art.

This competition requires that the submitted models must be evaluated by a specific performance metric: the top-k error. Usually, in the form of top-1 and top-5 error. When classifying an image, the model returns the five most probable class labels c_i for $i = 1, ..., 5$ in a decreasing order of confidence. The top-1 error is calculated by asserting if c_1 is the correct class label. On the other hand, the top-5 error checks if the the correct class label is among c_i. So, considering:

- c_i for $i = 1, ..., 5$ as the five most probable class labels inferred by the model for an image, in a decreasing order of confidence.
- C_k for $k = 1, ..., n$ with n ground truth class labels.
- $d(c_i, C_k) = 0$ if $c_i = C_k$
- $d(c_i, C_k) = 1$ if $c_i \neq C_k$

Then, the model classification performance is computed by the following equation Russakovsky et al. (2015):

$$e = \frac{1}{n}. \sum_{k} \min_{i} d(c_i, C_k) \tag{7}$$

1 http://image-net.org/challenges/LSVRC/2010/browse-synsets

Next, the most relevant achievements on ILSVRC are presented by year. On 2012, AlexNet exceeded the ILSVRC state of the art at that moment, achieving a top-1 and top-5 errors of 37.5% and 17.0% on ILSVRC-2010, respectively. The network was trained with an extended version of the original training set by using data augmentation techniques. A variant of this network was also submitted to the ILSVRC-2012 competition and it won with a top-5 error rates of 15.3%. For instance, the second best entry stood on 26.2% Krizhevsky et al. (2012). By the time it came out, ZFnet set a new record with a top-5 error of 14.8% on ILSVRC-2013 Russakovsky et al. (2015). In 2014, the two networks with best performance on the ILSVRC-2014 classification task were the GoogLeNet and VGG. GoogLeNet won the competition by achieving a top-5 error of 6.67% Szegedy et al. (2014); Russakovsky et al. (2015) while VGG got the second place with a top-5 error of 7.3% Russakovsky et al. (2015); Simonyan and Zisserman (2014). The residual neural network architecture was submitted to the ILSVRC-2015 achieved an outstanding top-5 error of 3.57% which surpassed the human-level performance He et al. (2015b,a).

3

TRAINING METHODOLOGY

In this chapter, the work developed for this dissertation will be detail. First we discuss some common issues regarding CNNs. Based on this discussion we also propose some procedures to improve the training result. This discussion is supported by the experiments in chapter 4. We will start by discussing issues related to how the dataset should be used to train and test a CNN. Considering training we compare existing stopping criteria and propose a novel criterion. It will also be adressed the batch size and learning tuning. Focusing on the architecture we discuss the role of multiple convolutional layers.

Afterwards, we present a toolkit that extends TFLearn Damien et al. (2016) in order to provide an easier entry point for users starting to use CNNs.

Figure 26 outlines the typical proposed procedures for training CNNs and it is supported by the work flow presented on it. This figure is a graphical summary of those procedures for that will be described in the following sections.

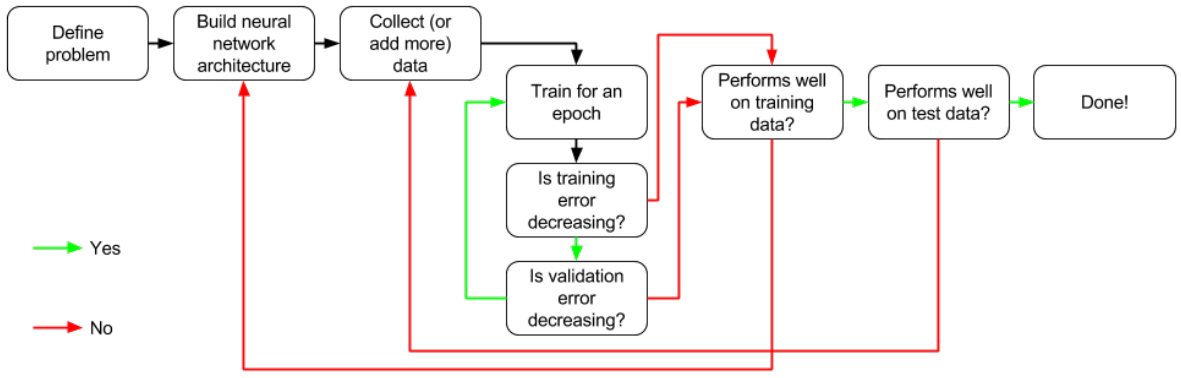


Figure 26: Common deep learning work flow for a training methodology. Image based on Buduma and Locascio (2015).

The state of the art network architectures were already described on Section 2.3. So, in the next sections will be exposed the issues related with the preparation of the data for training, such as pre-processing and data augmentation, and the training procedure itself.

3.1 DATASET PREPARATION AND USAGE

This section details procedures related to the dataset, ranging from the preparation of the dataset, to its possible subsets for training, validation and testing, and finally describing on how to feed the training set to the CNN.

3.1.1 *Pre-processing*

With image datasets we get pixel values ranging from 0 to 255. This interval implies that all inputs are positive. To provide a more balanced set it is common to zero center the pixel values. This can be done simply by shifting all pixel values from the interval $[0, 255]$ to $[-127.5, 127.5]$. A more sophisticated approach computes the average image of the training set and subtracts it from every image in the data set.

These input intervals may cause the weights from the first layers, those more directly affected by the magnitude of the inputs, to have large updates, causing an erratic learning procedure. Therefore it is common to use normalized values in the range $[-0.5, 0.5]$. Again to be more precise we can use normalization by standard deviation which will provide a dataset dependent interval, but in average the range remains similar. This process is depicted in Figure 27.

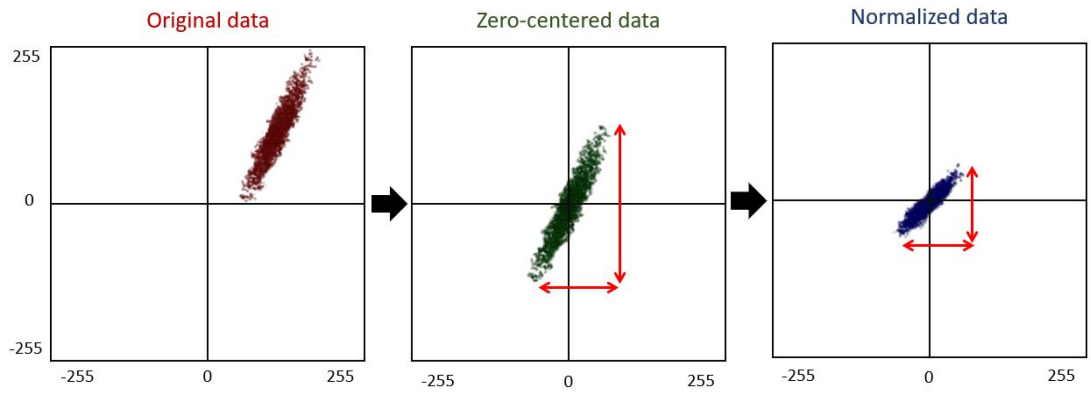


Figure 27: Image pre-processing operations: zero-center and normalization. Image adapted from Karpathy and Johnson (2016).

3.1.2 *Data Augmentation*

As mentioned before this thesis focuses on the use of CNN on classification tasks. Hence, the datasets we consider are composed of a number of labeled samples, distributed among several classes. Ideally all classes would have a similar number of samples. However, some

datasets are unbalanced. To restore the balance data augmentation is required. Whenever possible we can simply gather more data.

Another common approach involves augmenting the dataset with variations of the original images. These variations may include brightness and contrast changes, random blur, image displacement, rotation, scaling, mirroring, flipping, among other possibilities as Figure 28 suggests.

Figure 28: Example of data augmentation operations like flips, noise, rotation random blur.

Data augmentation can also be done to increase the range of samples to teach the CNN. This is done to expose the neural network to a wider range of variations, making it more likely to have a good performance when classifying unseen data. An example where this has been put to good use can be found in Staravoitau (2017).

The German traffic signs training set (Section 4.2.3) is very unbalanced. The number of images per classes varies between 210 to 2250 as Figure 29 evidences.

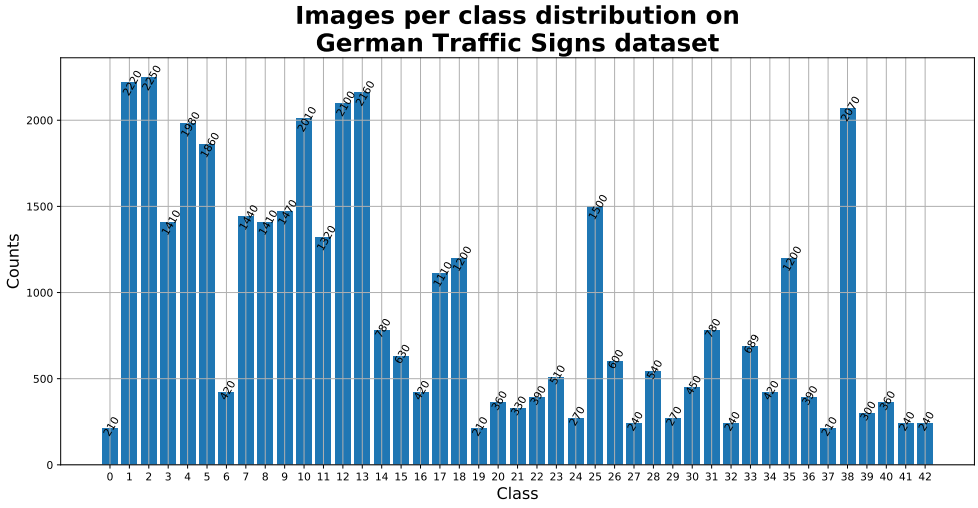


Figure 29: Images per class distribution on German traffic signs training set.

In order to balance the number of images per class, the data augmentation operations presented above can be applied. Note that, not all classes can be artificially augmented by those procedures. For instance, it is not expected to see a flipped stop sign. In other cases, other signs may belong to other class when transformed. The same problem applies to other datasets but in a similar context.

Moreover, the size of the training set can also be reduced by pre-processing. For example, dimensionality reduction is one type of pre-processing methods that is mean to convert the RGB channels into a single gray-scale channel. In the case of the signs, although its colors help on distinguish each category, they can be differentiated by its shape too. That's what this approach relies on.

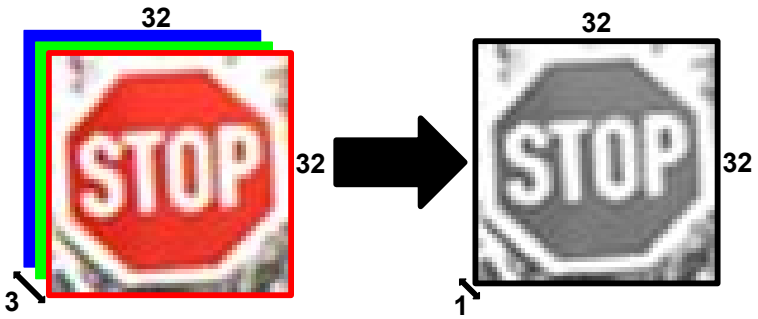


Figure 30: Dimensionality reduction by converting RGB chanels to a single gray-scale channel.

Of course, this can only make sense in the context of colorized images. The example presented above on Figure 30 reduces the input size by a factor of 3.

3.1.3 *Splitting The Dataset*

CNNs are renowed by their capability of performing the classification task with a high degree of accuracy. The accuracy is particularly relevant when the CNN is confronted with previously unseen samples. Therefore, the main goal of training a CNN is not to learn the data used during training, but rather to be able to correctly classify unseen samples.

A successful training session with the entire set would guarantee the required accuracy on the samples of the dataset, but this does not imply getting a similar, or even acceptable level of accuracy on new data.

Considering this we could partition the dataset into two subsets: the training set and the validation set. Yet, the original dataset could also be partitioned in order to have a third subset: the testing set.

Training Set

The training set is used to fit the model. The network weights will be updated from it. The training set must be representative so the model can learn the most important features to distinguish all classes.

The accuracy on the training set tells if the network configuration used is able to learn the data. So, if the model does not perform well on the training data this means that the network architecture must be reviewed and improved.

Validation Set

In turn, the validation set helps guiding the training and it is useful to know how the general learning is progressing. One of the big features of deep learning is the ability to generalize in variate types of images. The validation set is intentionally used to measure the model generalization capabilities and helps understanding if the model it's getting over-fitted to the training data.

Nevertheless, it is also important to take into account how the validation set is considered. Lets consider three options: a dedicated validation set; a percentage from the original shuffled data and a proposal for a custom method for generating a validation proportional to the training set.

In the first option, it is assumed that there is already a dedicated validation set that is loaded like a training set and then it is passed as an argument for the training operation. This method is useful when the validation set is already explicit. So, there is no need to spend time on generating it.

Instead, in the second option is to set a percentage of the original data that will be used only for validation purposes. This method is provided by TFLearn Damien et al. (2016). In this case, the original data is all shuffled first and then it picked the last percentage of it, as specified. This process doesn't give guarantees about the heterogeneity of the validation set because the shuffling operation is random and it may not gather samples of all classes in the last percentage of the data. Figure 31 illustrates how to define a validation set from the original data, using a percentage value.

In addition to the methods presented before, it was implemented a custom data split method that work as Figure 32 suggests. This procedure assures that the validation set is created from the original data and, most importantly, proportional to it. The user is also free to enable or disable this functionality or set the amount of percentage he wants for validation.

In case of unbalanced datasets, the implemented data split method helps in not let the validation set get even more unbalanced. The random factor may not be beneficial in situations where there are some classes over-loaded with more images than the remaining

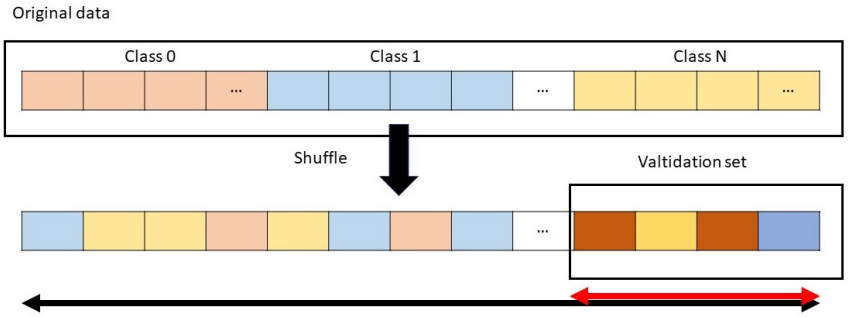


Figure 31: TFLearn method to split data into training and validation set through a percentage.

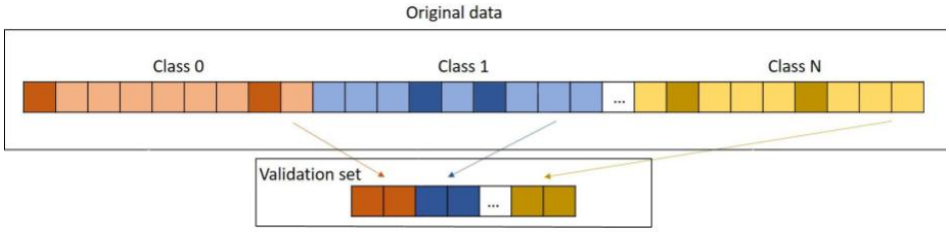


Figure 32: Proposed data split method to create a validation set.

classes. Nevertheless, the original data can be intentionally unbalanced in cases where some classes are more significant than others.

Testing Set

The testing set is meant to test the generalization capabilities of the trained model. Note that, similarly to the validation set, the test set is completely unknown to the network. The objective of training CNNs is to train on a specific dataset and infer in similar instances (not equal at all). Hence, the test set is also important to test how far can the model go on different data compared with the data that it was trained on.

3.1.4 *Feeding The Training Set*

One epoch corresponds to one pass over all training images. Some datasets are too big which mean that they can be feeded to the training as a single piece due to hardware limitations. So, they must be splitted into smaller batches of images.

The batch size is a training hyper-parameter that quantifies the size of the image batch which will be propagated forward through the network in order to calculate the error between the networks predictions and the ground truth targets. Then, according to the error the network weights are updated by back-propagation. So, the error back-propagation is

directly influenced by the number of samples of each batch as well as by its heterogeneity. In addition, this hyper-parameter also controls the frequency which the weights are updated Goodfellow et al. (2016).

If it is set too low, the image batch is more likely to be biased in favor of a particular class or group of classes, therefore, the computed gradient might not be the best estimate of the overall gradient, and, hence, the optimization algorithm may take a longer path. This can in part be avoided by considering a small learning rate. On the other hand, if the learning rate is too small, it may imply longer training time.

Hardware features can also be taken into account in order to determine the batch size. How many samples can we fit in the memory of a GPU? How many samples can be processed in parallel? These are relevant questions to consider if the goal is to optimize the training time per epoch. The size is advised to be base 2 once some hardware devices are specially designed for dealing with those sizes Goodfellow et al. (2016).

In fact, smaller batch sizes have a regularizer effect due to the noise they introduce on the learning process by several weights update Goodfellow et al. (2016).

3.2 TRAINING CNNs

The training method can be categorized in supervised, unsupervised and reinforcement learning. On supervised learning, the model is iteratively exposed to new or unknown information. Given a set of inputs and targets, the models weights are updated and adjusted according to its error, measured as a function relating the predictions to the ground truth. In turn, unsupervised learning algorithms try to understand data relationships without knowing any information about it. Reinforcement learning differs from the previous two because it isn't supervised neither unsupervised. In this type of learning there isn't labeled data. The model learns by receiving rewards which are the outcome of specific predefined actions. So, the model will try to follow the actions that give him the most rewards. Hereupon, that's why it can be considered unsupervised learning.

For this dissertation was only considered supervised learning. The purpose of supervised learning is to use an optimization algorithm capable of finding such weights that minimize the error among network predictions and ground truth values. At the same time as the network weights are updated based on the training set, the model is also evaluated on validation set. The validation set is useful because it guides the training and tells if the model is getting over-fitted to the training data. As soon as the validation set accuracy reaches a pre-established value, the training operation finishes. Later, the model's performance should still be evaluated on a test set, which is completely unknown to the model Tawfique (2016).

In the enumeration below are exposed two possible approaches for training a CNN:

1. **Training from scratch** means start the training from the beginning with a random weight initialization. In this approach, the network weights are solely updated around the input data.

2. **Transfer learning** starts from a previously trained net (for another dataset). Many models trained with the state of the art architectures on the ImageNet subset can be found online Simonyan and Zisserman (2014). In order to use them to classify a custom dataset, the output layer must be adapted to the number of classes and then trained on the new data. These procedure is useful because the feature extraction part is heavily supported by a model that was trained on a large variety of data, including in different levels of information. Some researches use this approach as a start point for considerable complex problems.

3.2.1 *Definition Of Training Stop Criteria*

This section is meant to delineate a training stop criteria. There are some universal applicable procedures to CNNs but since each dataset has it own properties, some experiments need a specific approach. After getting used to the mechanics of the deep learning framework, it started to make sense to establish a stop criteria for the training procedure.

By default, the TensorFlow Abadi et al. (2015) stop criteria is reaching a specific top-1 accuracy (item 2.3.2) on the validation set together with a maximum number of training epochs. Considering a particular sample, this criteria states that the prediction is correct if the output for the correct class is the highest among all predictions. While this is true, it also may be the case where two or more classes have very close output values, which could be interpreted as if the network is not really confident about which class the sample belongs to.

Hence, four stop checks were set for the training procedure that are better explained next:

1. The classification confidence threshold only considers well predicted images with a confidence above a certain value. This restrain will be used on the validation set in order to force the model to separate the data properly.

2. The validation accuracy must be greater or equal to a specific value considering the classification confidence threshold described on the previous item.

3. Learning progress to check if the learning status has evolved. If not, the training ends after some epochs due to no progress.

4. A maximum number of training epochs, just for safeguard.

The classification confidence is an issue that needs a special attention in order to achieve a proper model. The definition of a proper performance metric is a crucial step to get trustworthy results, able to be interpreted correctly.

The proposed accuracy metric for the validation set combines two crucial aspects. When an image is sent to the network it returns a probability array, where each element correspond to a single class and its sum is equal to 1. Instead of just checking if the class with the highest probability is the correct one, it is also verified if that probability (or confidence) exceeds a specific value, for example, 75%. Basically, this new metric is a more restricted version of the top-1 accuracy once it also takes the classification confidence into account. So, using this discretion enforces the confidence on the predicted class because it promotes a higher separating between the correct class and the other classes.

Moreover, it avoids situations where the network can infer correctly the true class but there is another class with a similar confidence. For example, suppose a 4 classes image classification task where the model outputs the probabilities array $[0.1, 0.1, 0.41, 0.39]$. If the image belongs to the class 2 (consider indexation starting on zero) the network would have inferred it well using the maximum probability method. Yet, class 3 confidence is pretty close to class 2 which doesn't give a total trust regarding the network ability to distinguish those two classes. This is the limitation that the developed metric aims to overcome. In this case, the developed metric would penalize this result because although the network made a right prediction it doesn't have enough confidence. In other situation, for example, with a probabilities array of $[0.1, 0.1, 0.75, 0.05]$, there is a clear gap from the well predicted class and the remaining ones, eliminating any doubts about it. Utmost, it would only consider a good prediction when the probabilities array gets like this $[0, 0, 0.75, 0.25]$ and even so, the true predicted class confidence is three times bigger than the network's best second guess. It also works fine as a solid and robust stop criteria for training, specially when it starts to achieve high accuracy values. Nevertheless, the user is free to define or use other custom accuracy metric.

So, similarly to the TensorFlow default training stop criteria, this verification on validation accuracy asserts if it reaches a top value. However, remember the classification confidence threshold previously described. This time, the validation accuracy must hit a certain value whose the classification confidence is 75%, at least.

After a number of epochs without any learning progress (consider it as an increase in the validation or testing accuracy), the training will automatically stop. The intention of this criteria was to detect if the learning process has stagnated. If true, it early stops the training operation and saves the trained model according to the last best checkpoint registered.

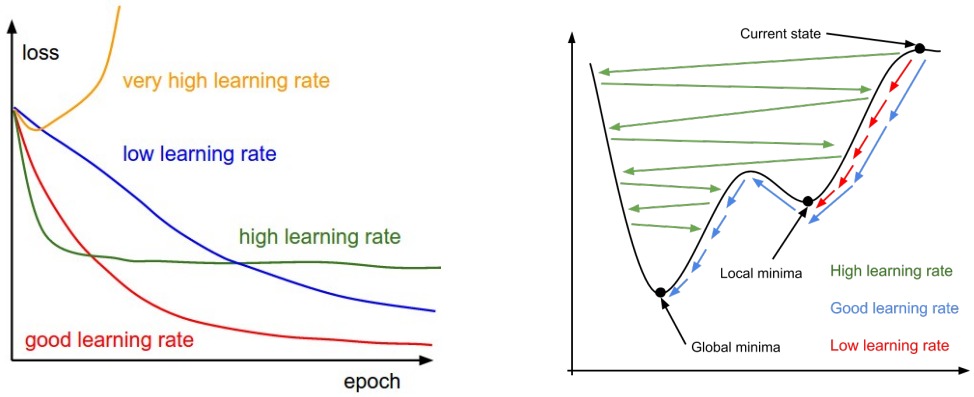
Finally, the maximum number of epochs is the last check. This verification is optional and it is only triggered when the previous two don't. It is used just as a safeguard.

By default these parameters are set to a 75% of classification confidence, a validation accuracy threshold of 97.5%, 250 evaluations for no learning progress early stop and a maximum of 1000 training epochs. Yet, they can be parameterized to whatever the user wants. Take a special look on the experience at Section 4.3 to how the training stop criteria behaves in practice.

3.2.2 *Learning Rate*

The learning rate is a hyper-parameter that influences how much the network weights will be updated. A higher learning rate results in a more abrupt update with large jumps which may hinder on finding the global minima. On the contrary, low learning rates update the weights through small steps. However, with such small steps the training can end up on saddle points where it may be stuck for a very long time.

Thus, its a good practise to use an adaptative learning rate scheduler. According to the chosen optimizer, the learning rate can also be automatically updated during the training progress by a decay factor. Figure 33 illustrates two different scenarios of how the learning can influence the model performance.



(a) (b)

Figure 33: Influence of learning rate on the model performance: (a) Loss values depending on the learning rate and (b) Stochastic gradient descent possible paths with different learning rates. Image by Karpathy and Johnson (2016).

3.3 PROPOSAL FOR A TOOLKIT

During this dissertation it was developed a test toolkit with the intent of ease the training, validation and testing deep learning models for image recognition. Its purpose is to offer a set of tools that can effectively train a network on a certain dataset as simply as possible.

In terms of level of abstraction, this toolkit is entirely written in Python (version 3.5.2) and it is an extension of the TFlearn Damien et al. (2016) (version 0.3.2) code which in turn, stands on top of Tensorflow Abadi et al. (2015) (GPU version 1.4). Figure 34 illustrates the abstraction level of the tools mentioned above.

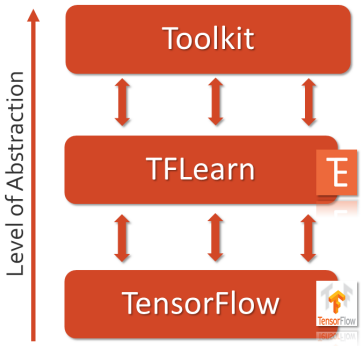


Figure 34: Proposed toolkit level of abstraction.

Comparing with the original Tensorflow syntax which is very extended, the TFLearn syntax offers a much cleaner way of building network architectures. On Listing 3.1 are compared both syntaxes in order to build a convolutional layer.

```
# Tensorflow syntax
with tf.name_scope('conv1'):
    W = tf.Variable(tf.random_normal([5, 5, 1, 32]), dtype=tf.float32,
                    name='Weights')
    b = tf.Variable(tf.random_normal([32]), dtype=tf.float32,
                    name='biases')
    x = tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
    x = tf.add_bias(x, b)
    x = tf.nn.relu(x)

# proposed toolkit syntax (supported by TFLearn syntax)
tflearn.conv_2d(x, nb_filters=32, filter_size=5, strides=1,
                activation='relu', padding='same', name='conv1')
```

Listing 3.1: Comparison between Tensorflow and the TFLearn syntax for building a convolutional layer. TFLearn syntax is cleary more appealing and intuitive. This code snippet was get from Damien et al. (2016).

In practise, Listing 3.2 presents how to train and test a single model on the proposed toolkit. The appendix A.3 has a more detailed explanation about the training behaviour. Nevertheless, the three required arguments for training and testing are the path to the training data, the architecture name and the run identifier. There are other arguments that can be parameterized but they assume default values on these cases.

```
# to train a model
> python training.py --data_dir=<path_to_training_set>
                     --arch=<architecture_name> (check Listing 3.3)
                     --run_id=<training_run_id>

# to test a trained model
> python classify.py --data_dir=<path_to_test_set>
                     --arch=<architecture_name> (check Listing 3.3)
                     --model=<path_to_trained_model>
```

Listing 3.2: Example of how to train and test a model on the developed toolkit (Note: arguments are diposed vertically just for a clearer visualization).

The **data directory** must be organized in a way where classes are separated into sub-directories. For example, class-0 images should be in a folder, class-1 images in other folder, and so on, just like Figure 35 suggests. In fact, it is an efficient way of labeling data too. The same rule applies to the validation and testing sets. In alternative, data can also be loaded through a comma-separated-value (CSV) indexing file in which the first column corresponds to the path to the image and the second column corresponds to its respective class or label.

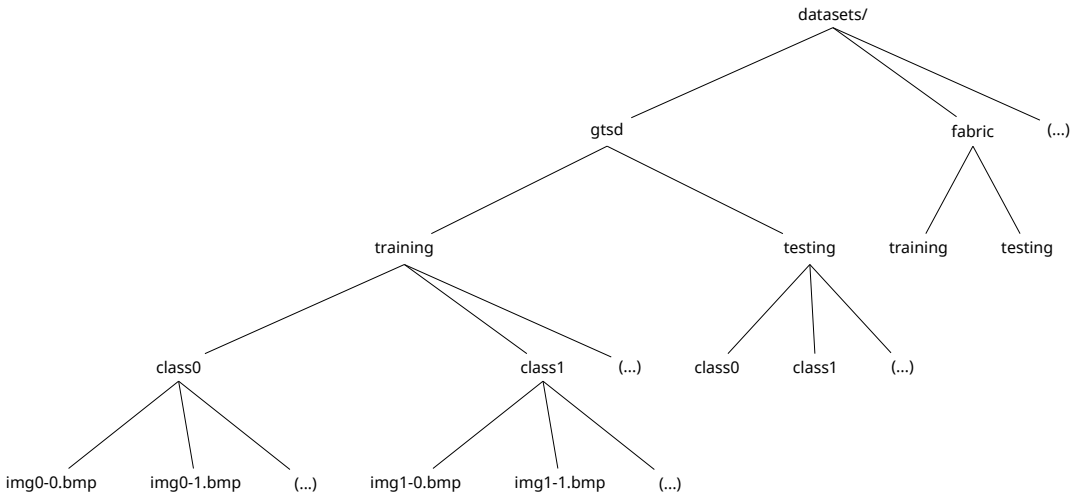


Figure 35: Datasets directory structure.

The **architecture name** is a string that maps the corresponding network architecture definition on the *utils/architecture.py* module (explained in detail on Appendix A.3). In it are expressed the issues related with the configuration of convolutional, max-pooling and fully-connected layers. Listing 3.3 has a code snippet that demonstrates how the architectures

module operates. The function *build network* is the one responsible for matching the architecture name to its definition.

```
import tflearn

# alexnet network architecture
def build_alexnet(nclasses):
    net = tflearn.input_data(shape=[None, 224, 224, 3],
                             data_preprocessing=img_prep
                             data_augmentation=data_aug)

    net = tflearn.conv_2d(net, nb_filters=96, filter_size=11, strides=4,
                          activation='relu', padding='same')
    net = tflearn.max_pool_2d(net, kernel_size=3, strides=2)
    net = tflearn.local_response_normalization(net)

    net = tflearn.conv_2d(net, nb_filters=256, filter_size=5, strides=1,
                          activation='relu', padding='same')
    net = tflearn.max_pool_2d(net, kernel_size=3, strides=2)
    net = tflearn.local_response_normalization(net)

    net = tflearn.conv_2d(net, nb_filters=384, filter_size=3, strides=1,
                          activation='relu', padding='same')
    net = tflearn.conv_2d(net, nb_filters=384, filter_size=3, strides=1,
                          activation='relu', padding='same')
    net = tflearn.conv_2d(net, nb_filters=256, filter_size=3, strides=1,
                          activation='relu', padding='same')
    net = tflearn.max_pool_2d(net, kernel_size=3, strides=2)
    net = tflearn.local_response_normalization(net)

    net = tflearn.fully_connected(net, n_units=4096, activation='relu')
    net = tflearn.dropout(net, keep_prob=0.5)
    net = tflearn.fully_connected(net, n_units=4096, activation='relu')
    net = tflearn.dropout(net, keep_prob=0.5)
    net = tflearn.fully_connected(net, n_units=nclasses,
                                  activation='softmax')

    net = tflearn.regression(net, optimizer='momentum',
                         loss='categorical_crossentropy',
                         learning_rate=0.001)

    return net

# vgg16 network architecture
def build_vgg16():
    (...)
    return net

```

```
# general network architecture builder
def build_network(architecture_name, nclasses):
    if(architecture_name == "alexnet"):
        net = build_alexnet(nclasses)
    elif(architecture_name == "vgg16"):
        net = build_vgg16(nclasses)
    (...)
    else:
        net = None

    return net
```

Listing 3.3: Snippet of how to configure the Alexnet network architecture on the proposed toolkit *architectures.py* module.

The **run identifier** argument must be unique in order to preserve the results obtained on several runs. Besides, it also gives the name to the trained model that will be saved after the training. In case of a repeated run identifier, the previous results as well as the trained model will be overwritten.

However, when it comes to plan a experience on a larger scale things can get time consuming. DNNs are characterized for having a lot of hyper parameters to tune. Tuning those parameters manually, one by one, becomes impractical. Hereupon, this toolkit is also meant to expedite the process of combining different sets of parameters in order to find which one gives the best outcome. To achieve it, this toolkit includes a schedule script in where it can be explicit the training parameters such as the the training data, the network architecture, the batch size and many others. The schedule script is exposed on Listing 3.4.

```
import os
import sys

# ready control flag
try:
    ready = sys.argv[1].lower() in ["go", "ready"]
except:
    ready = False

operations = ["python training.py"]

train_dirs = ["/path/to/dataset1/training/",
              "/path/to/dataset2/training/",
              "/path/to/dataset3/training/"]

test_dirs = ["/path/to/dataset1/testing/",
             "/path/to/dataset2/testing/",
             None]
```

```
19
20 architectures = ["alexnet", "vgg16", "resnet"]
21
22 batches = [128, 256, 512]
23
24 n_runs = 3
25
26 width  = 224
27 height = 224
28
29 # for all commands
30 for op in operations:
31     # for each train/test data pair
32     for traind,testd in zip(train_dirs, test_dirs):
33         # get data ID from path
34         data_id = traind.split(os.sep)
35         data_id.reverse()
36         data_id = data_id[2]
37         # for each architecture
38         for arch in architectures:
39             # for each batch size
40             for bs in batches:
41                 # repeat for N times
42                 for run in range(0, n_runs):
43                     # build run ID automatically
44                     run_id = "%s_%s_bs%d_r%d" % (data_id, arch, bs, run)
45                     # build command to execute
46                     cmd  = "%s --data_dir=%s --arch=%s " % (op, traind, arch)
47                     cmd += "--run_id=%s --bsize=%d "     % (run_id, bs)
48                     cmd += "--width=%d --height=%d "     % (width, height)
49                     cmd += "--test_dir=%s"               % (height, testd)
50
51                     print(cmd)
52                     if(ready):
53                         try:
54                             # try to execute command
55                             os.system(cmd)
56                         except Exception as e:
57                             # otherwise, goes to next iteration
58                             pass
```

Listing 3.4: Code snippet of experience planning using the schedule script.

When called in the commands prompt without any extra argument, the schedule script will just print the commands ready to be executed. If the script is executed with an additional argument like *go* or *ready* it starts executing the commands configured on it.

As evidenced on Figure 36, from the schedule script of Listing 3.4 will be performed 81 training operations by combining 3 datasets on 3 network architectures through 3 distinct batch sizes and repeating each experience for 3 times. The (...) symbol represents the procedures replication for the other cases.

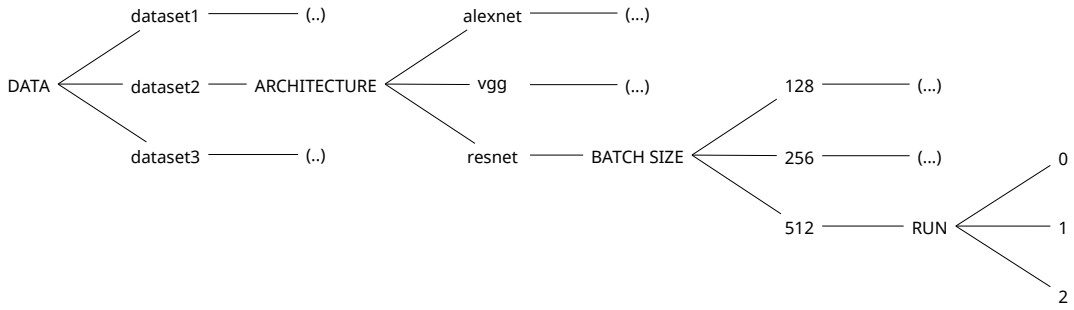


Figure 36: Combination of commands that will be executed from the schedule script described on Listing 3.4.

This automation saves a lot of time because it only requires one single configuration in order to obtain several outcomes. Meanwhile, the toolkit produces one report file (see Listing 3.5) at each training that shows how the learning evolves.

```
################ TRAINING REPORT #################
# Images path    | <path_to_dataset>
# Validation     | 0.30
# Images shape   | (<height>,<width>,<channels>)
# Architecure    | <architecture_name>
# Bacth Size     | <batch_size>
# Snap/Epoch     | 5
# Max epochs     | 1000
# Eval criteria  | 0.75
################################################
train,validation,test,time
0.00,0.00,0.00,0.00
79.87,78.23,72.86,66.10
92.00,90.61,85.54,130.89
100.00,99.13,98.21,199.34
          (...)
```

Listing 3.5: Snippet of a training report file.

Once plotted, these reports allow a visual interpretation of the results and help understanding the impact of tuning a specific parameter.

On Appendix A.3 is exposed the structure of the whole toolkit and a more detailed description about all its components.

4

EXPERIMENTS

This chapter describes the most relevant experiments performed during this dissertation work. These experiments act as support for Chapter 3 which was more theoretically oriented.

We'll start by presenting the experimental environment, Section 4.1. Then, in Section 4.2, the datasets used for the experiments are described. Afterwards, experiments related with the training stopping criteria are detailed in Section 4.3. Learning consistency is addressed in Section 4.4. Next, comes the experiments on the images color-space on Section 4.5. On Section 4.6 are shown which filters does the CNN automatically learns. Tuning the batch size is explored in Section 4.7. Then it is studied the learning rate tune in Section 4.8. Finally, some general performance considerations are presented in Section 4.9.

4.1 EXPERIMENTAL ENVIRONMENT DESCRIPTION

All the experiments were executed on a computer with specifications as presented in Table 3. The information was gathered from Intel Support and HP Costumer Support.

Table 3: Testing machine specifications.

Processor	Intel Core™ i7-6700HQ
Processor's base frequency	2.60 GHz
Max turbo frequency	3.50 GHz
# Cores	4
# Threads	8
Cache memory	6 MB SmartCache
RAM memory	2x 8 GB SDRAM DDR4 2133 MHz
Storage	256 GB PCIe ® NVMe ™ M.2 SSD
GPU	Nvidia ® GTX 960m 4GB

4.1.1 *Deep Learning Framework*

In the beginning, a very important decision stood in the choice of a deep learning framework. An intuitive framework is crucial for a quick adaptation and implementation of different network architectures. From this reference[1], there are several alternatives and TensorFlow was the chosen one. TensorFlow was originally developed by Google's machine learning engineers Abadi et al. (2015); Buduma and Locascio (2015). Although it isn't the framework with the best performance in terms of execution time and memory consumption, TensorFlow is very flexible and it has a growing community of users as well as contributors Bahrampour et al. (2016); Shi et al. (2016); Kuster (2016); Buduma and Locascio (2015).

In order to accelerate the creation and adaptation of architectures, a TensorFlow high-level API called TFLearn Damien et al. (2016) was used. It offers a higher level abstraction, the code gets a lot simpler, clearer and much more understandable. It also allowed the training and building of an image classifier in just a few code lines. The available user-friendly examples bring already some of the state of the art network architectures that were referenced before. In addition, both TensorFlow and TFLearn syntax can be written all together, enabling to go into a deeper-level when needed. Hence, besides the abstraction level, the main reason for choosing this framework is the possibility of taking advantages of all TensorFlow features and maintain a readable code Damien et al. (2016); Silver (2016). However, the lack of pre-trained models stands as one drawback of this API.

4.2 DATASETS

In order to have a wide range of domains several image datasets were used. Every test has a purpose and the choice of a good input dataset is one of the key points to achieve the pretended conclusions. This data diversity also brings different levels of features, which means that specific network architectures will be needed in order to learn them. Table 4 contains a summary of the dataset properties detailed underneath.

4.2.1 *Subset of Kylberg Texture Dataset*

Kylberg texture dataset is composed by 28 texture classes and is available in two different subsets, one with and other without rotated patches Kylberg (2011). All images are composed by low-level features. The subset without rotated patches has less classes as well as less samples and is described on the following points:

- 6 classes (canvas, cushion, linseed, sand, seat and stone);

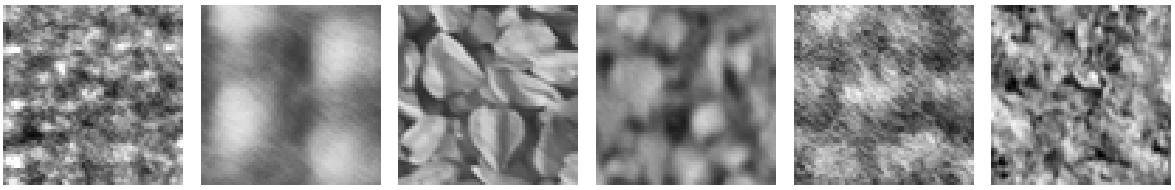
1 http://deeplearning.net/software_links/

Table 4: Summary of the datasets properties.

Dataset	Classes	Samples per Class	Total Samples	Images size
Subset of Kylberg texture	6	640	3,840	64x64x1
Fabric texture	7	363	2,541	64x64x3
MNIST	10	7000	70,000	28x28x1
Digits	11	705 to 873	8,634	64x64x1
German Traffic Signs	43	210 to 2,250	51,839	32x32x3
Subset of Parking Lot	2	3,038 and 3,244	6,957	64x64x3

- 40 images per class (30 for training + validation and 10 for testing);
- 576x576 gray-scale images (single channel);
- Source: http://www.cb.uu.se/~gustaf/texture/.

On Appendix A.1.1 is presented one original sample of each texture class. However, those images weren't used for training on its original size because they are too big and it would take unnecessary resources and time to process them. In addition, they would lose texture detail when resized to a smaller dimension. So, a set of 64x64 crops were extracted, from each training image, using a sliding window, as shown in Figure 37.



(a) (b) (c) (d) (e) (f)

Figure 37: Example of cropped textures used for training networks: (a) Canvas; (b) Cushion; (c) Linseeds; (d) Sand; (e) Seat and (f) Stone.

This approach has benefits because these crops contain the main features necessary to distinguish one texture from another. In this dataset, these crops are big enough to catch, at least, 3 texture elements from any texture image. Textures are defined by a certain pattern that is repeated along the whole image and 3 texture elements encompass the minimum requirements to represent those patterns. Moreover, smaller images make the training and the classification procedures lighter and faster, since the number of parameters will be smaller too. The training set contains 3,600 images and the test set has 240 images.

4.2.2 *Fabric Texture Dataset*

This image dataset was captured and provided by Neadvance, Machine Vision, SA. In total, it contains 2,541 RGB images equally distributed by seven classes (363 images per class). The training set has 2,310 images and the test set has 231 images. In Figure 38 one sample of each class is illustrated.

(a) (b) (c) (d) (e) (f) (g)

Figure 38: Samples of the fabric texture dataset.

Here are some particularities about this fabric dataset that deserve to be mentioned:

1. Although Figure 38(b) and Figure 38(g) look like the same class, there is a little detail that differs them. The color of a wispy diagonal line (left bottom to top right) is brown and grey, respectively;
2. Figure 38(c) and Figure 38(d) represent two classes with suchlike colors yet with two different levels of grain pattern, a finer and a thicker one, respectively;
3. Figure 38(e) and Figure 38(f) have exactly the same pattern but the first has grey dots while the second has blue dots;
4. The remaining class pairs can be easily differentiated either by color or pattern.

Technically, this dataset can also be considered as texture based and like the previous one, its images are composed by low-level features.

4.2.3 *German Traffic Signs Dataset*

The German traffic sign recognition dataset is known for being used as a image recognition benchmark Stallkamp et al. (2011). The training operation requires that all images have the same dimensions. Since this dataset contains images with different sizes, they were all resized to 32x32. Comparatively to the previous texture datasets, this one has high-level features and it is expected to perform better in a network configuration with more convolutional layers. Relatively to the dataset specifications, it contains 51,839 RGB images, with 39,209 images for training and 12,630 images for testing, distributed by 43 classes. Each class represents a specific sign.

(a) (b) (c) (d) (e) (f)

Figure 39: Samples of the German taffic signs dataset.

From Figure 39 it is possible to verify that the illumination is not constant along all images. For instance, Figure 39(c) is much brighter than Figure 39(d). Moreover, some images have partial shadows (Figure 39(a)) and others are saturated (Figure 39(f)). Furthermore, signs can assume miscellaneous shapes (circles, squares, triangles, rectangles or even diamond) and a range of four colors (red, blue, white or yellow).

4.2.4 *MNIST Dataset*

The MNIST dataset LeCun et al. is a handwritten digits database commonly used as the *"Hello World!"* of the CNN. It contains a training set of 60,000 images and a test set of 10,000 images, all in grayscale with size 28x28 LeCun et al.; Bottou et al. (1998). The state of art test error rate, in this dataset, stands on an impressive 0.21% Wan et al. (2013).

Figure 40: MNIST samples. Image by TensorFlow's MNIST tutorial.

4.2.5 *Digits Dataset*

The digits dataset was also provided by Neadvance, Machine Vision, SA and was extracted from images of mechanical energy counters through an automatic segmentation method. It has eleven classes that include ten digits, ranging from 0 to 9, and an eleventh class that was intentionally designed to store all non-digits images, such as noise or other artifacts that result from the segmentation method (Figure 41(g)). Original digit sizes had considerable variation, so they were all resized to 64x64 and converted to gray-scale since there is no useful color information present on these images.

The training and test sets have 7,835 and 799 images respectively. As for the MNIST dataset, the main challenge on this dataset will be the recognition of the same digit but in

(a) (b) (c) (d) (e) (f) (g)

Figure 41: Samples of the digits dataset. (g) Represents one of the non-digit characters.

different fonts. This dataset was meant to train a model for a optical character recognition tasks (OCR).

4.2.6 *Parking Lot Dataset*

The parking lot dataset Almeida (2015) has 695,899 RGB images of parking spaces labeled in just two classes: empty/free or busy/occupied with an approximated rate of 44% and 56%, respectively. These images were captured during daytime in different ways in order to achieve a representative and sturdy dataset Almeida (2015). In Figure 42 is a representation of the images that constitute this dataset.

(a) (b) (c) (d) (e) (f) (g) (h)

Figure 42: Samples of the parking spaces dataset.

The following enumeration has a few facts about the diversity of the images. On the Appendix A.1.2 Figure 66 shows the three top views of each entire parking lot from where these smaller images were extracted. In summary, these images were captured in following conditions:

1. From three distinct angles, each one corresponding to a specific parking lot;
2. Under three types of weather (cloudy, rainy and sunny) and, consequently, different illumination conditions and also with shadows;
3. Partially occluded by trees or afforestation.

For training CNNs was extracted a subset of 6,957 parking spaces images that respects the original dataset proportion of busy/free images. The training and testing sets contain 6,262 and 695 images.

In this context, the dataset was intended for a binary classification problem. However, it can be used for other tasks, such as image segmentation and object detection, because it comes with XMLs files that map each parking space location on the parking lot images.

4.3 DEFINITION OF TRAINING STOP CRITERIA

This experience was meant to find out the robustness of the defined training stop criteria. The motivations behind the training stop criteria were presented on the Section 3.2.1. Two models were trained: One model was trained on the Kylberg texture dataset 4.2.1 with the network topology described on Table 5 and another model was trained on the Digits dataset 4.2.5 using the architecture presented in Table 6.

Table 5: Architecture used on Kylberg texture dataset to test the stop criteria robustness.

Layer	**Filters**	**Filter size**	**Stride**	**Activation**
Convolutional	8	3x3	1x1	ReLU
Max Pooling	1	2x2	2x2	-
Batch Normalization				
Convolutional	16	3x3	1x1	ReLU
Max Pooling	1	2x2	2x2	-
Batch Normalization				
Convolutional	24	3x3	1x1	ReLU
Max Pooling	1	2x2	2x2	-
Batch Normalization				
Fully Connected	128			ReLU
Dropout	0.5			
Fully Connected	6			Softmax

Regarding the datasets, both contain gray-scale images, but the first one is composed by textures while the second one is composed by digits. As opposed to a texture, a digit can be represented by a combination of features. Because of that, a more complex network (with more convolutional filters) was selected to train on Digits dataset.

Besides, both architectures have characteristics in common. They are composed by three blocks with a convolutional layer followed by a 2x2 max pooling and a batch normalization layer. On the Kylberg dataset were used 8, 16 and 24 3x3 filters while on the Digits dataset were used 16, 32 and 64 3x3 filters for the first, second and third convolutional layers, respectively. ReLU was used in all activations. Zero-padding was used on convolutional layers in order to preserve the dimensions after the convolutions. It avoids loosing information at the volumes borders. At the end, the architecture used on the Kylberg texture dataset has two fully connected layers with 128 and *nclasses* units. In turn, the architecture used on the Digits has three fully-connected layers with 256, 128 and *nclasses* units. Between

Table 6: Architecture used on Digits dataset to test the stop criteria robustness.

Layer	Filters	Filter size	Stride	Activation
Convolutional	16	3x3	1x1	ReLU
Max Pooling	1	2x2	2x2	-
Batch Normalization				
Convolutional	32	3x3	1x1	ReLU
Max Pooling	1	2x2	2x2	-
Batch Normalization				
Convolutional	64	3x3	1x1	ReLU
Max Pooling	1	2x2	2x2	-
Batch Normalization				
Fully Connected	256			ReLU
Dropout	0.5			
Fully Connected	128			ReLU
Dropout	0.5			
Fully Connected	11			Softmax

fully-connected layers were added dropout layers with a keep probability of 50%, to force the model to handle absence of information.

In all these experiments, for each dataset the initial conditions are the same.

Figure 43 illustrates the training evolution on the Kylberg texture dataset considering the TensorFlow default stop criteria and the proposed stop criteria.

As expected, on Figure 43 the training took much more epochs when using the proposed stop criteria. In both cases, the training finished when a validation accuracy of 97.5% was achieved. However, with the TensorFlow default stop criteria the accuracy on validation set is calculated by only asserting if there is a match among the predicted and the ground truth classes. Beyond that, on the proposed stop criteria the classification confidence must be greater or equal to 75% in order to be considered a well classified image. Although it takes more time to train, the proposed stop criteria, ensures that images are classified more correctly since it forces the model to create a better data separation. In addition, the test accuracy also achieved a higher value when the proposed stop criteria was enabled.

At this point, Figure 44 shows the percentage of images from the Kylberg texture test set that are classified correctly and with a confidence above a certain parameter.

As it can be seen on Figure 44, with the proposed stop criteria the accuracy on the Kylberg texture test set maintains stable independently to the presented classification confidences. Considering the TensorFlow stop criteria the accuracy stood a little lower along the way. In particular, for a classification confidence greater or equal than 60% the accuracy drops slightly which evidences the role of the proposed criteria.

Additionally, and in order to test both models generalization capabilities, an additional test set was created. This new test set contains the same images as the original Kylberg

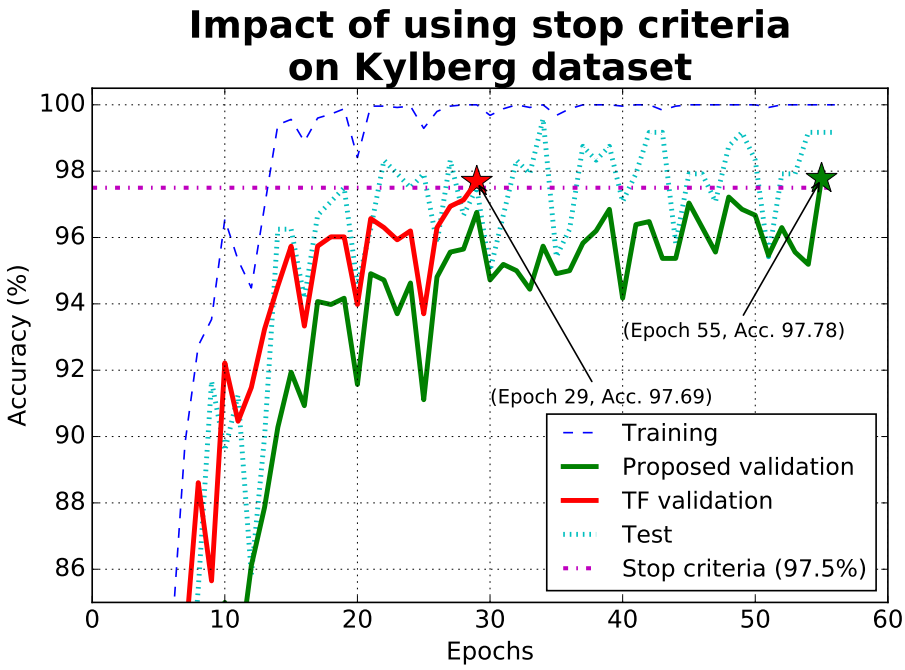


Figure 43: Training evolution with both TensorFlow default stop criteria and proposed stop criteria on Kylberg texture dataset. The star marks represent the moment when the stop criteria were hit.

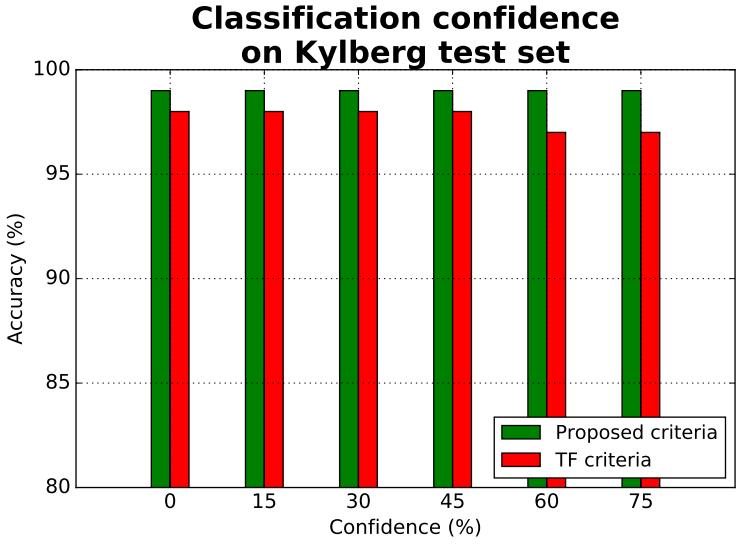
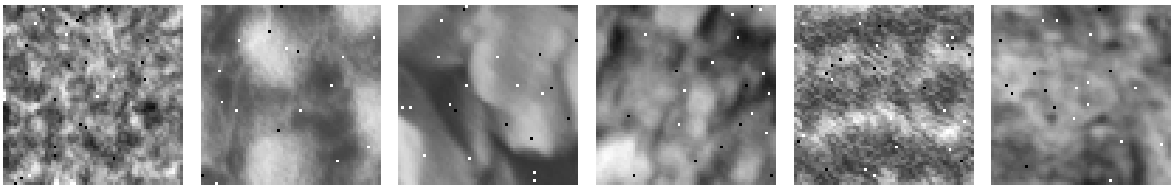


Figure 44: Kylberg texture test set accuracy according to the classification confidence using both stop criteria.

texture test set but with salt and pepper noise. Figure 45 shows one noisy sample of each class.



(a) (b) (c) (d) (e) (f)

Figure 45: Samples of noisy Kylberg textures test set: (a) Canvas; (b) Cushion; (c) Linseeds; (d) Sand; (e) Seat and (f) Stone.

Then, was done the exact same verification for the classification confidence on both criteria but in the noisy test set. If the model generalized well it may behave similarly. Figure 46 shows the performance of the two models in the noisy test set.

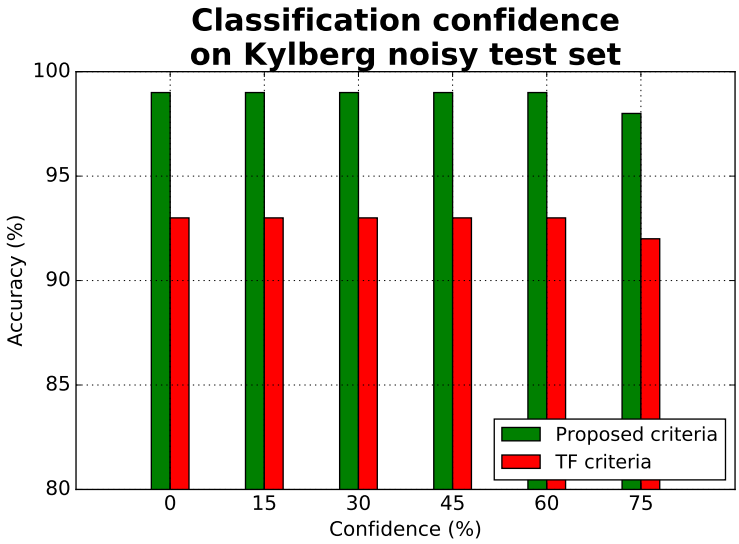


Figure 46: Noisy Kylberg texture test set accuracy according to the classification confidence using both stop criteria.

Once again, the accuracy for the proposed stop criteria kept almost constant around the 99%. For a classification confidence greater or equal than 75%, we get a little decrease. In turn, compared with the previous figure, the accuracy for the TensorFlow stop criteria went even lower by a three percentage points, approximately. The most accentuated drop was registered for classification confidences of at least 75%. So, it means that the model trained on the proposed stop criteria is more generic and it can handle data with more variation.

To reinforce the role of the proposed stop criteria, all the procedures already described above on this experiment were repeated on the Digits dataset. First, it starts by comparing the training evolution on the two models. For the TensorFlow stop criteria the validation accuracy scaled up very quickly and the stop criteria was easily hit. For the proposed

stop criteria it took more epochs to achieve good classification confidences and so the test accuracy was able to reach a higher value. Compared to Figure 43, the training evolution on Figure 47 was much more irregular, indeed. Actually, the Kylberg texture dataset is more regular than the Digits dataset. For instance, variation of the linseed class boils down to transformations as shifts and small rotations. Beside these, the variations of the digit 1 include also different fonts.

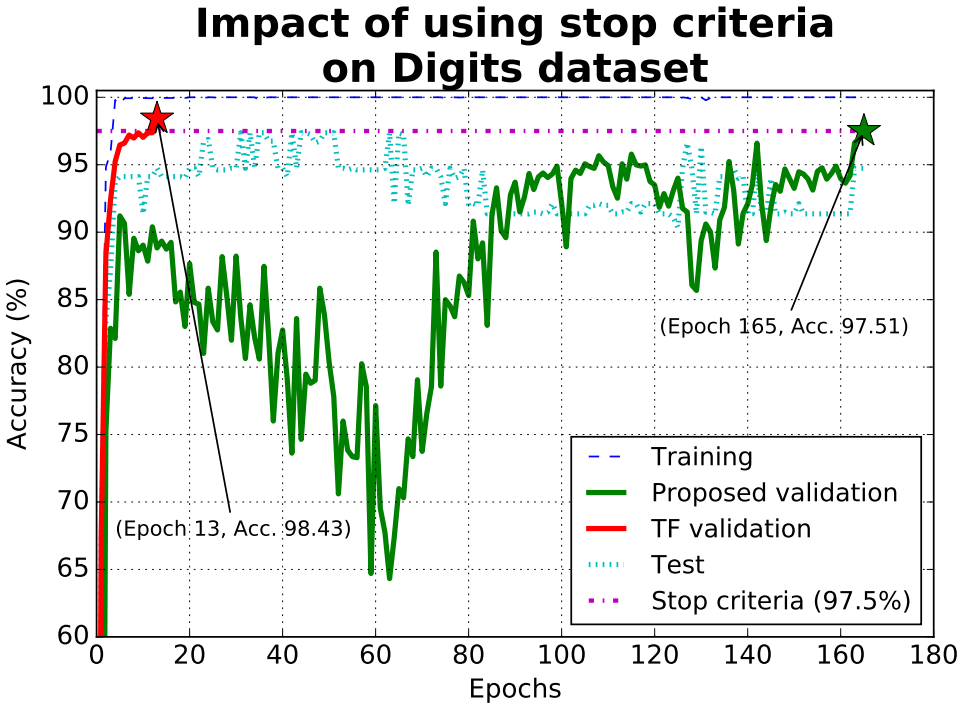


Figure 47: Training evolution with both TensorFlow default stop criteria and proposed stop criteria on Digits dataset. The stars represent the peak validation accuracy on both cases when the stop criteria is hit.

Figure 48 compares the classification confidences of the two models trained on the Digits dataset. Here, the distinction of the models is more evident. Again, the major differences were registered from confidences higher than 60%, which exactly points out the restrain of the proposed classification stop criteria.

Figure 49 shows a few samples of the noisy Digits test set. The images black background makes clearer to highlight the noise. This dataset was intended to test the generalization capabilities of both trained models.

As Figure 50 indicates, this time the noisy Digits test set accuracy values started to decay earlier. The first occurrences of classification confidence loss appear already on the 30% and they get bigger along with the increase of the classification confidence threshold.

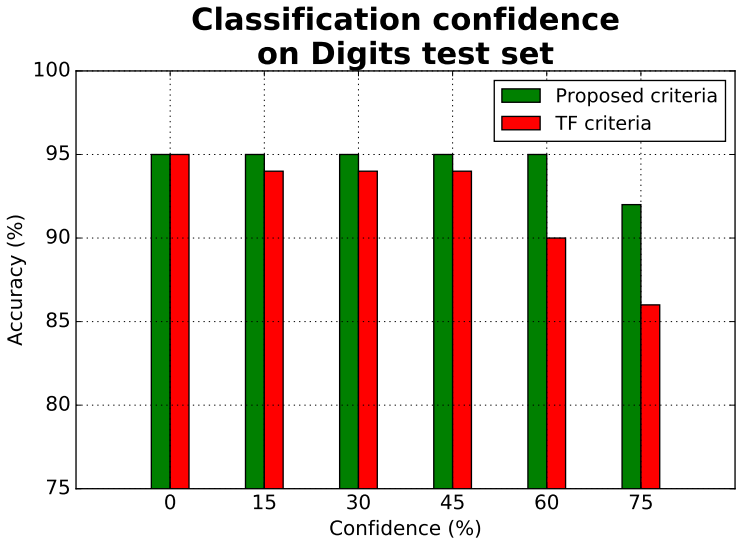


Figure 48: Digits test set accuracy according to the classification confidence using both stop criteria.

(a) (b) (c) (d) (e) (f) (g)

Figure 49: Samples of noisy Digits test set.

Figures 51 (a) and (b) present a comparison of the same procedure that was previously shown on Figure 50 but with higher levels of noise.

For the noise level 2 was registered a similar behaviour with regard to accuracy drop compared with Figure 50. On the 45% of confidence was noticed a considerable difference. From there, the gap between both criteria becomes more evident and accuracy values get gradually lower.

It is notable on both experiences the gap among the two validation accuracy values for each stop criteria. The proposed stop criteria restrain offers a better resolution about the model predicting capabilities. It is useful because it clarifies if the model can predict correctly and most importantly without doubts.

This experience suggests that the proposed training stop criteria can be adequate for the kind of tasks it had been used on. In short, all further experiments will used this criteria.

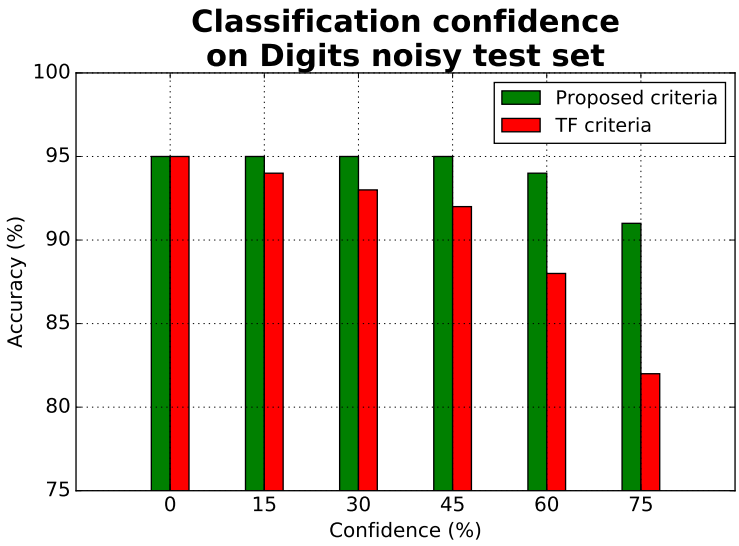
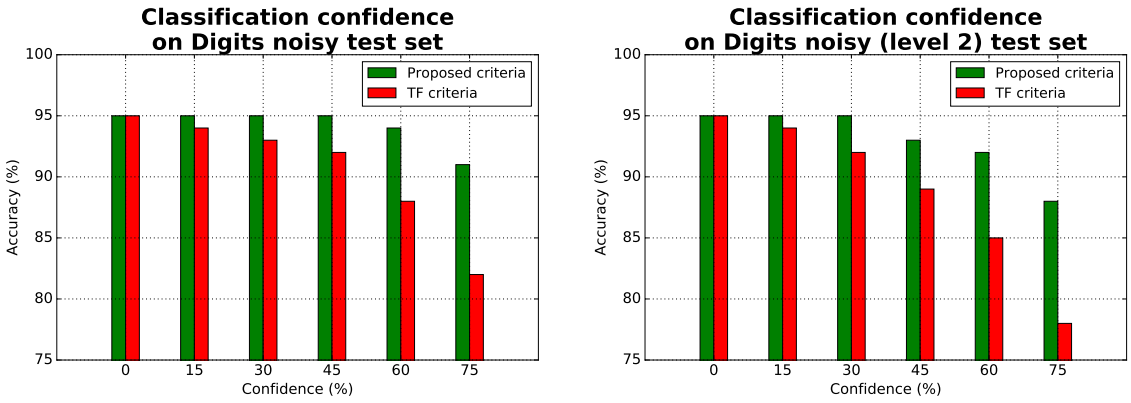


Figure 50: Noisy Digits test set accuracy according to the classification confidence using both stop criteria.



(a) (b)

Figure 51: Noisy Digits test set accuracy according to the classification confidence using both stop criteria: (a) Noise level 1 and (b) Noise level 2.

4.4 LEARNING CONSISTENCY

The training procedure isn't deterministic at all because it contains operations with random processes. For example, by default the network weights are initialized with random values from an uniform distribution. In addition, if the network topology includes any dropout layer, the neurons dropped during training are also randomly selected. This means that, unless the seed points are fixed, it can't be assured that two training operations will have

the same results. Because of that, 100 training operations were performed under the same conditions in order to figure out the dispersion of the results.

Figure 52 shows two box plots that present the number of epochs needed to finish each training: on left with the Fabric dataset and on right with the German traffic signs dataset. Both used the stop criteria exposed in the preceding experiment, on Section 3.2.1. These datasets were trained with the architectures presented on Table 7 and Table 8, respectively.

Table 7: Architecture used on the Fabric dataset for asserting the learning consistency.

Layer	**Filters**	**Filter size**	**Stride**	**Activation**
Convolutional	8	5x5	4x4	ReLU
Max Pooling	1	2x2	2x2	-
Batch Normalization				
Fully Connected	50			ReLU
Dropout	0.5			
Fully Connected	7			Softmax

The architecture used on the Fabric dataset is very tiny. It has only one convolutional layer of 8 5x5 filters with stride 4 followed by a max-pooling layer. 50 units are enough on the fully-connected layer. In general, this dataset is regular and it is composed by images with low-level features. Images were captured in a controlled environment with good illumination conditions and without brusque variations. The differences among images from the same class are essentially shifts.

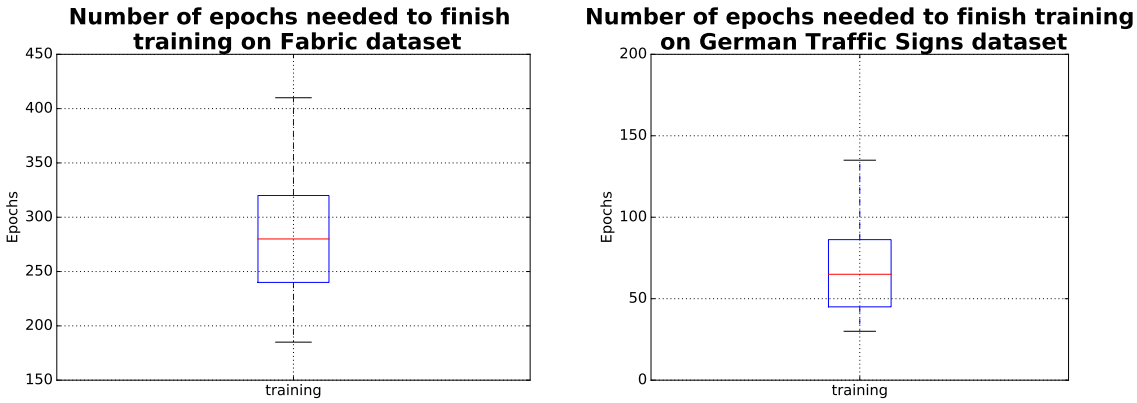
Table 8: Architecture used on German Traffic Signs dataset for asserting the learning consistency.

Layer	**Filters**	**Filter size**	**Stride**	**Activation**
Convolutional	32	3x3	1x1	ReLU
Max Pooling	1	2x2	2x2	-
Batch Normalization				
Convolutional	64	3x3	1x1	ReLU
Convolutional	64	3x3	1x1	ReLU
Max Pooling	1	2x2	2x2	-
Batch Normalization				
Fully Connected	512			ReLU
Dropout	0.5			
Fully Connected	43			Softmax

In turn, the architecture used for the German Traffic signs dataset needs to be more complex. Despite its image have higher-level features, they also appear in a lot of lightning variations. So it requires a more robust feature extraction. Thus, were used three

convolutional layers with 32, 64 and 64 3x3 filters, respectively. This time, the number of fully-connected units is also bigger and it has 512 units.

In both architectures, the batch normalization layers are meant to normalize the inputs before feeding them to the next layers.



(a) (b)

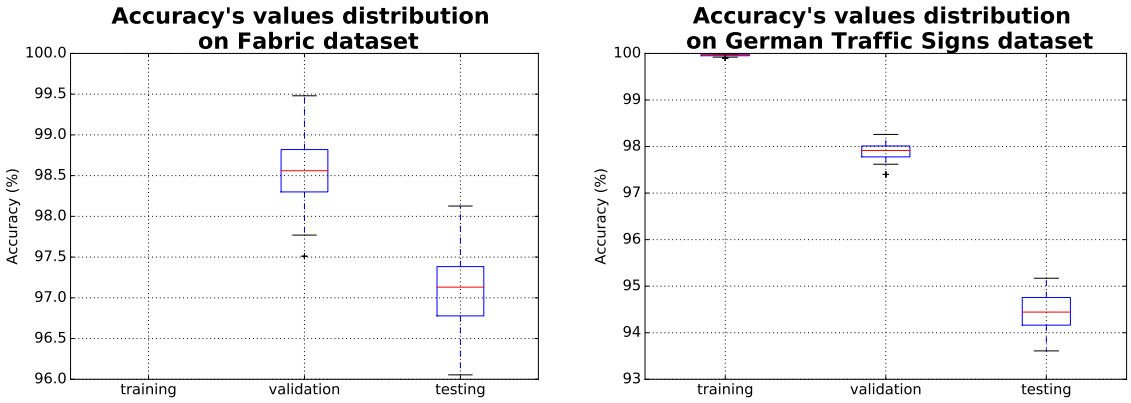
Figure 52: Number of epochs needed to finish training in 100 runs: (a) on the Fabric texture dataset and (b) on the German traffic signs dataset.

In the box plot of Figure 52(a), most of the runs ended up between about 200 and 400 epochs. On Figure 52(b) the values are more condensed in a range from 50 to 140 epochs instead. These results shows that the number of epochs needed to achieve a properly trained model can vary significantly depending on the starting conditions. This is to be expected since the random initialization can get closer or farther away from the optimal weight values. Depending on that, the stochastic gradient descending optimization method may take more or less iterations.

However, the main purpose of experiment was to pay attention to the accuracy values distribution, to check how they stand, together or apart from each other. The two box-plots of Figure 53 present the training, validation and testing accuracy values distribution on the 100 runs.

The Figure 53(a) shows that the training accuracy is so concentrated (nearly not visible) that all its values are very close to 100%. Validation accuracy values are densely inside the 97.75% and 99.5% range with an outlier reaching the 97.5%. However, the top validation accuracy reached near to 99.5%. At last, testing accuracy values stand a little bit lower between 96% and 98%.

On Figure 53(b) it can be seen that once again the training accuracy is very close to 100%. Besides the validation accuracy got slightly lower values compared with the other experience, they are still all concentrated under a range delimited by 97.5% and 98.5%,



(a) (b)

Figure 53: Final accuracy values distribution on the 100 runs: (a) on the fabric dataset and (b) on the German traffic signs dataset.

approximately. Testing accuracy stood one more time below the corresponding validation accuracy with values ranging from around 93.5% to 95.5%.

So, this experience suggests that there is a coherence on the learning method and its ability to converge despite the training operation randomness.

4.5 COLOR SPACES

The designation color space stands for a abstract mathematical model that describes a color by a tuple, typically through one, three or four values components. The RGB and gray-scale color spaces are the mostly common when dealing with image processing Knoche (2017). However, there are many others color spaces that can favor particular computer vision tasks. For example, the HSV (Hue, Saturation, Value) color space is very useful when detecting objects by its color because it makes easier to use restrict color ranges in the Hue channel. Figure 54 illustrates the detection of blue color pixels through the HSV color space.

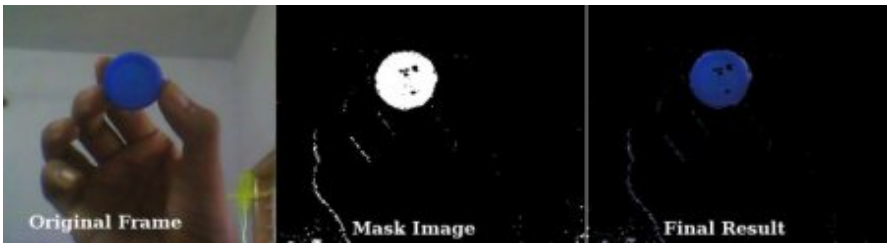
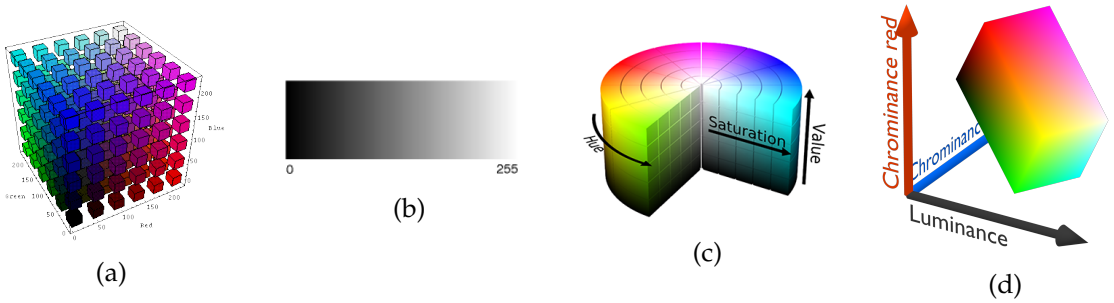


Figure 54: Detection of blue color in an image through the HSV colorspace. Image picked from Python's OpenCV changing color space tutorial.

Most color spaces encode the same information, but with different representations. In fact, they can be converted to another and (back to the original) just by applying a matrix multiplication.



(a) (b) (c) (d)

Figure 55: Color spaces represented geometrically: (a) RGB; (b) gray-scale; (c) HSV; and (d) YCrCb. Images by Purdue University, Collage of Engineering and Knoche (2017)

- The RGB color space describes each color as a combination of three colors red, green, and blue. On Figure 55(a) they are represented by the three Cartesian axis. The intensity of each component determines the final color: the black color result from zero intensity on every channel; and full intensity from each channel results in the white color Knoche (2017).
- As the designation says itself, the Grayscale colorspace encodes the color intensity into a gray shade scale that ranges from 0 to 255 (traduced to color from black to white). In certain cases, this range is normalized to $[0, 1]$. So, colorized features can't be considered in this color space.
- The HSV (Hue, Saturation, Value) color space is a cylindrical representation of the RGB color space.
- The YCrCb is another representation of the RGB colorspace but with the components Luminance (luminous intensity), Chrominance red and Chrominance blue, respectively.

The signs that constitute the German traffic signs dataset 4.2.3 are have a color component that allows to filter some on them immediately. Mandatory and information signs are characterized by the blue color. Similarly, prohibition and warning signs are characterized the red color or by a red border. By these statements, the color seems to be a considerable factor to distinguish the signs. However, signs can also be distinguished by their shape and by the symbols they contain in the middle. So, it makes sense to test single channel colorspaces too. In fact, as it as suggested on Section 3.1.2, converting color image datasets to a single color channel is a way of reducing data dimensionality.

For this experiment was used the German Traffic Signs dataset with the architecture described on Table 8.

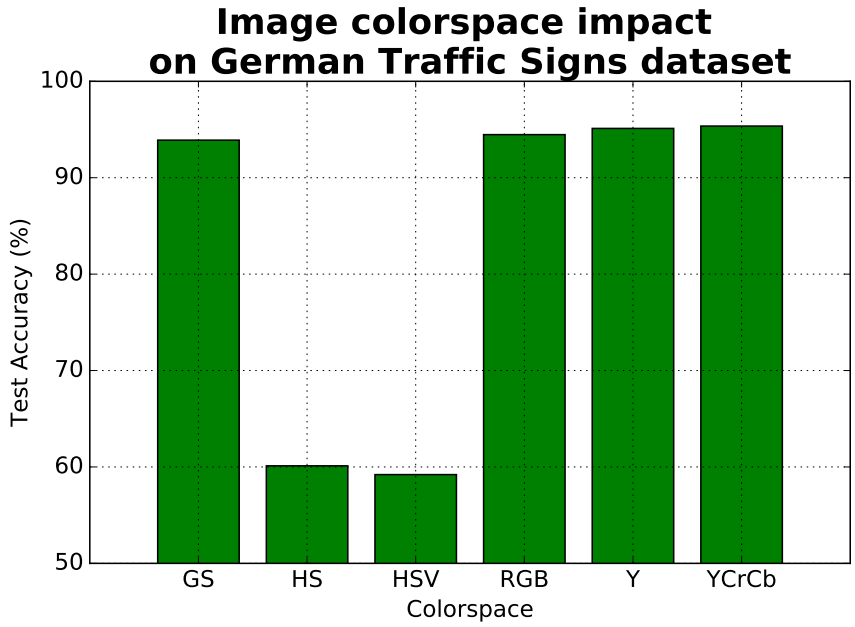


Figure 56: Image colorspace impact on training a CNN.

From Figure 56 it can be extracted that the HS (Hue and Saturation channels only) and HSV were both unable to reach a decent accuracy on the test set. On the contrary, with the remaining color spaces the accuracy on the test set reached similar high values even in the single channel color spaces. Thereby, the alternative of reducing the data dimensionality by squashing the color channels into a single one seems to be reliable on this dataset.

As Figure 57 evidences, on the HSV color space the red color ranges from $[330; 360]$ and $[0; 30]$ due to hue channel circular representation. The pixels represented by the blue and yellow boxes seem to be equal and their color don't differ too much visually. However, on the hue channel their value suffers a big deviation and that could be the cause for the model not being able to converge using the HSV color space. Furthermore, the saturation is very similar on both cases and the value channel is exactly the same. For that reason, was only considered the combination of the hue and saturation channel for this experiment.

4.6 Learned Filters

The deep learning literature emphasizes the network learning as an assembly of hierarchical features Zeiler and Fergus (2013); Simonyan and Zisserman (2014). The learned filters go

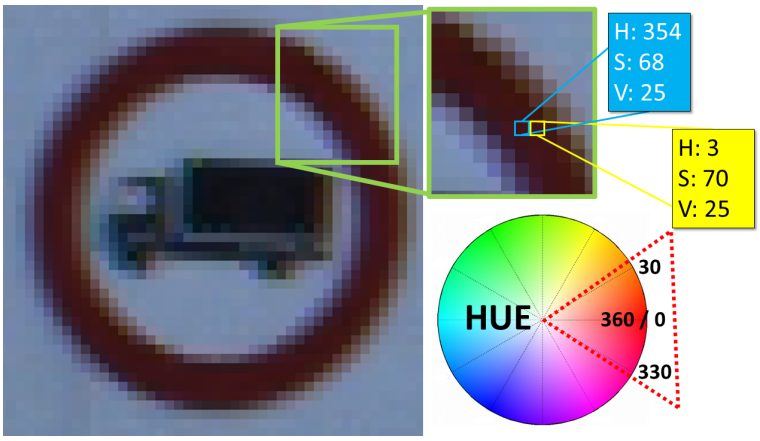


Figure 57: HSV color space values variation on red color range in the hue channel.

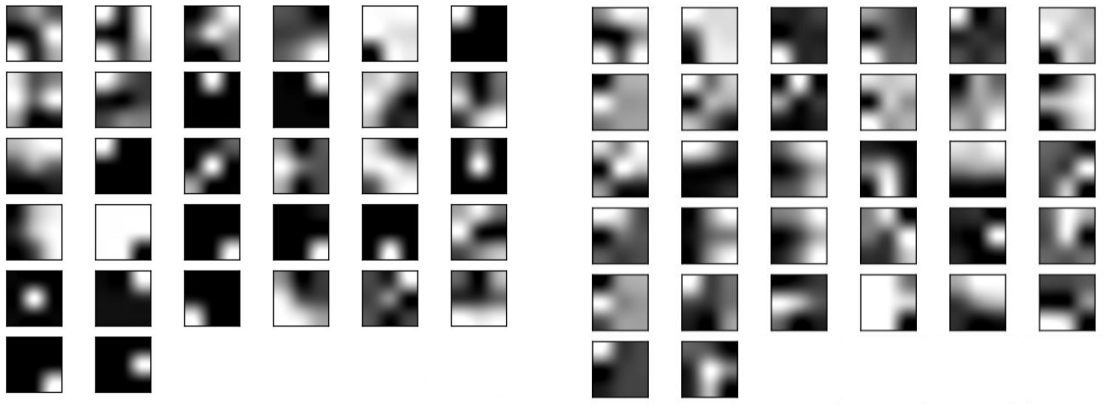
from a lower-level to high-level features as the network goes deeper Zeiler and Fergus (2013).

In this experience was used a single network architecture to train two models on two similar datasets which have features in common since they represent the same information in terms of data. So, for ir were used the Digits and MNIST datasets with architecture described by Table 9.

Table 9: Architecture used on MNIST and Digits datasets visualizae the learned filters.

Layer	**Filters**	**Filter size**	**Stride**	**Activation**
Convolutional	32	3x3	1x1	ReLU
Max Pooling	1	2x2	2x2	-
Batch Normalization				
Convolutional	64	3x3	1x1	ReLU
Max Pooling	1	2x2	2x2	-
Batch Normalization				
Fully Connected	256			ReLU
Dropout	0.5			
Fully Connected	128			ReLU
Dropout	0.5			
Fully Connected	$N_{classes}$			Softmax

The reason to choose these two dataset was because of the similarities they have. Despite each one has a specific font, both contain images with digits that share common features. And since a single architecture can solve many problems, it will be interesting to check how does it will behave on two similiar datasets. On Figure 58 are represented the learned filters on the first convolutional layer. As Table 9 indicates, it contains 32 learnable filters. Note that whiter regions represent higher values and vice verse.



(a) (b)

Figure 58: Learned filters on the first convolutional layer: (a) on MNIST dataset and (b) on Digits dataset.

As expected, at this stage filters can't describe such high level features. Most of them resemble to well known edge detector kernels. Some filters look like Laplacian and others are more likely to Sobel. Compared to the originals, they are slightly shifted or rotated (in diverse directions) but the main structure resemble to the matrices presented on the matrices.

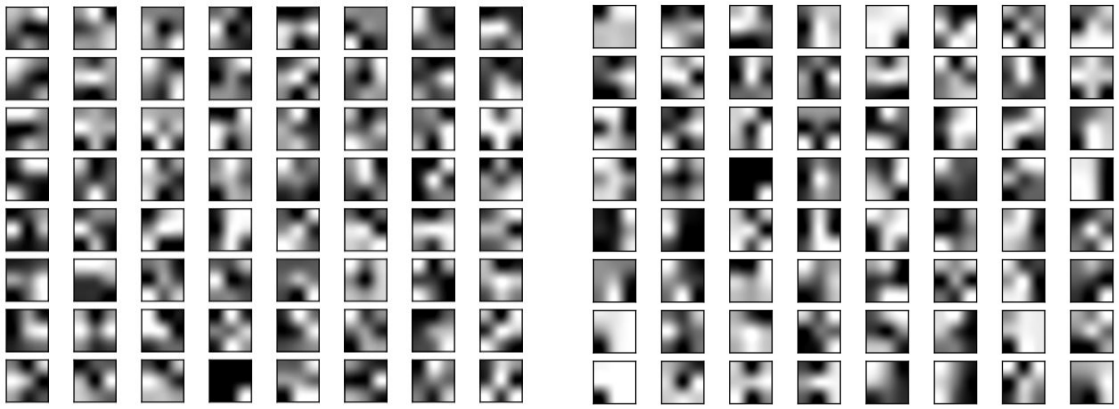
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

On the other hand, Figure 59 shows the 64 filters learned on the second convolutional layer. Note that these filter also derived from combining the filters learned in the previous layers.

The filters learned on the second convolutional layer are a little bit different. Their structure is much more complex and irregular. Still, some features are visible which constitute a partial part of a digit and that combined form an entire digit.

4.7 BATCH SIZE TUNING

During the training, the network is constantly fed with image batches. Each batch makes a forward pass through the network and the error among the network predictions and the ground truth is back-propagated in order to update the weights. So, depending on batch size the model can get better or worse trained and the training will require more or less memory, proportionally.



(a) (b)

Figure 59: Learned filters on the second convolutional layer: (a) on MNIST dataset and (b) on Digits dataset.

In this experience will be tested the impact of tuning the batch size. For it, were used batch sizes with power of 2 as Goodfellow et al. (2016) advises, ranging from 2 to 4096. The motivation to stop was the GPU memory limitation. Once it exhausts, no larger batch sizes are tested. This restriction ensures that the GPU memory will be used in order to process as much as it can.

The MNIST and the subset of PKLot datasets were considered and they have 10 and 2 classes respectively. Assuming a perfect scenario each training batch would be balanced. However, there is no guarantees of that once the data is shuffled before being fed to the network.

The MNIST dataset was trained on an architecture defined on Table 9 and the subset of PKLot was trained on the architecture described on Table 10.

The architecture presented on Table 10 was based on the original VGG16 network architecture Simonyan and Zisserman (2014). In fact, this configuration is a pruned version of it with less parameters and kernels.

According to the previous brief theory explanation about the impact of tuning the batch size and on Section 3.1.4, it is expected that the best outcomes will stand in intermediate batch sizes. The reasons are:

1. A smaller batch size means that the weights will be updated very frequently by a small image set. This set may not be representative for all classes . Yet, it produces a regularizer effect due to the noise it introduces on the learning process. To compensate that, it might require a smaller learning rate to stabilize the large number of updates.

2. In principle, a larger batch size can be more data representative. However, the network weights will be updated with less frequency and once they will be updated based on a bigger image set. Thus, the estimate of the gradient can be more precise but less

Table 10: Architecture used on subset of parking lot dataset for batch size tuning.

Layer	Filters	Filter size	Stride	Activation
Convolutional	32	3x3	1x1	ReLU
Convolutional	32	3x3	1x1	ReLU
Max Pooling	1	2x2	2x2	-
Batch Normalization				
Convolutional	64	3x3	1x1	ReLU
Convolutional	64	3x3	1x1	ReLU
Max Pooling	1	2x2	2x2	-
Batch Normalization				
Convolutional	96	3x3	1x1	ReLU
Convolutional	96	3x3	1x1	ReLU
Convolutional	96	3x3	1x1	ReLU
Max Pooling	1	2x2	2x2	-
Batch Normalization				
Fully Connected	512			ReLU
Dropout	0.5			
Fully Connected	512			ReLU
Dropout	0.5			
Fully Connected	2			Softmax

linear and the model may be not able to converge. In addition, it allows to process more data per time unit which means that it can also accelerate the training.

This experience pretends to clarify if there is any relation between the batch size and the number of classes. The obtained results are presented on Figures 60 and 61.

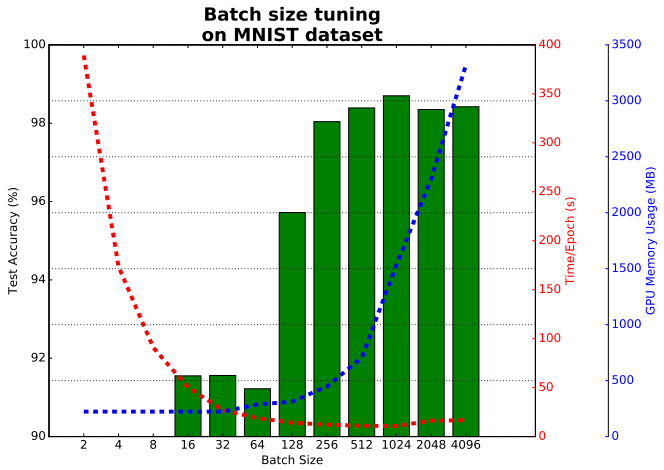


Figure 60: Batch size tunning on MNIST dataset.

Curiously, on Figure 60 for batch sizes smaller than the number of classes, the model wasn't able to learn. That's why there is no accuracy registered for those batch sizes. On the other hand, on Figure 61 the minimum batch size corresponds to the number of classes and the model reached a testing accuracy of 91.37%.

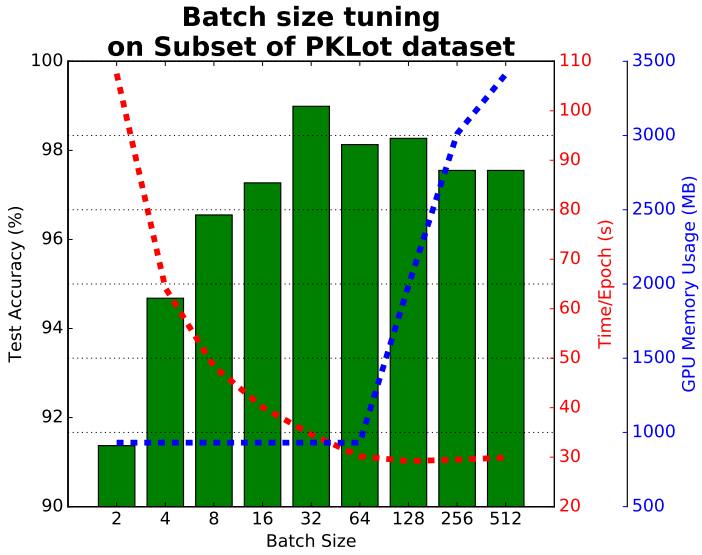


Figure 61: Batch size tuning on the Subset of the Parking Lot dataset.

From Figures 60 and 61 it can be seen that the time per epoch gets evenly smaller along the increase on the batch size. This evidences the GPU paralelization efficiency when dealing with large inputs. As both stagnated from the batch size 256 it suggests that the time per epoch would increase for bigger batch sizes. For instance, the amount of required memory was an issue on this experiment for the subset of the PKLot dataset with a batch size of 1024. For that reason, it is not shown on Figure 61.

Despite, considering the testing accuracy as the most privileged decisive factor of this experience once it describes the network's generalization capacities, the experiences suggests that the best batch size registered for the MNIST dataset trained with the architecture defined by Table 6 is 1024. This batch size nearly one hundred times bigger than the number of classes and the testing accuracy achieved a value of 98.70% respectively.

On the other hand, for the subset of the PKLot dataset which was trained on the architecture defined by Table 10, the best model was achieved with a batch size of 32 which is 16 times bigger than the number of classes. In turn, the testing accuracy hit 98.99%.

This experience wasn't meant to indicate a global batch size that may be applicable to any dataset. Instead, it suggests that the batch size must be, at least, greater or equal to the number of classes. In short, lower batches sizes introduce a lot of noisy during the training process due to the small number of samples from whose the model weights will

be updated. In addition, it compromises the training time by the larger number of batches that will be processed and the overhead that exists when sending data to the GPU. On the other hand, bigger batches sizes contain a more representative set of samples that can infer a more accurate path to the optimum solution. Besides, larger batches can take the most profit of the GPU computation capabilities. Yet, it can turn more difficult for the model to find a "sharp" local minima.

4.8 LEARNING RATE TUNING

In order to support the statements made on Section 3.2.2, the experience of tuning the learning rate was done. The Fabric dataset was used together with the architecture explained on Table 7.

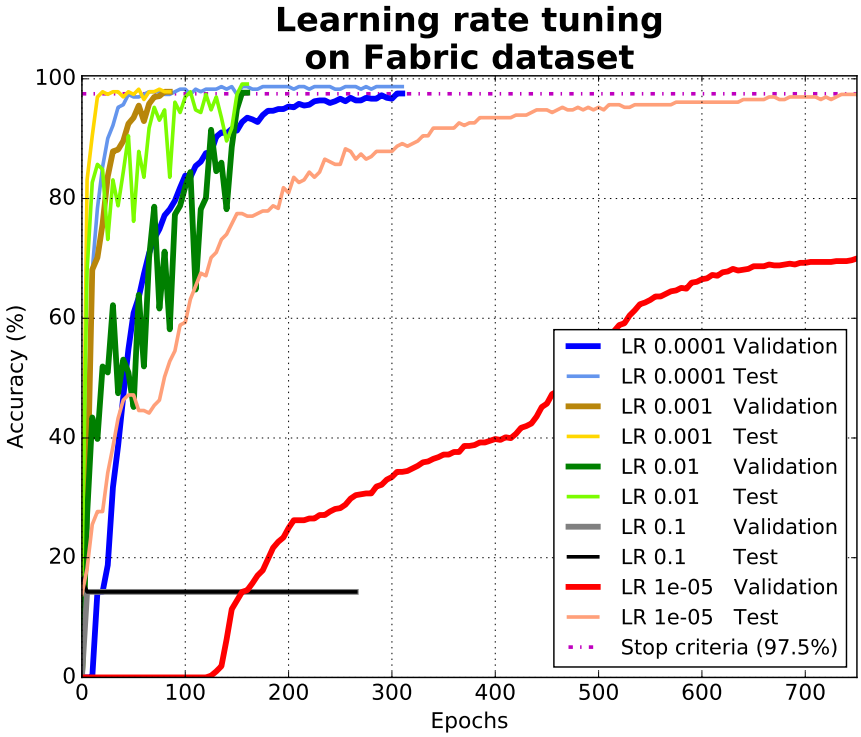


Figure 62: Learning rate tune on Fabric dataset.

Curiously, the model was capable of reaching the stop criteria even in the lowest learning rate but it took more than 3,000 epochs to finish the training operation. That's why it is not presented on Figure 62 in order to not distort the obtained results.

On the contrary, using the highest learning the validation and testing accuracy stagnated on 15%, approximately. The training ended early because there was no increase on both accuracy lines during the period of time defined by training stop criteria tolerance. The

large learning rate might not help finding the descending direction to the global minima. So the validation and test accuracy couldn't get over the 15%.

In general, the accuracy line raised exponentially. The noisiest variation was registered for the learning rate 0.01 which stands on the intermediate value of the tested learning rates.

Taking into consideration the results in Figure 62, and as expected, it is clear that the learning rate is an important parameter to achieve faster training. On the other hand setting a high learning rate can lead to early progress halt. Nevertheless, each dataset, as well as each architecture, have their own properties that influence the training evolution. Considering a different dataset or a different architecture the most adequate learning rate may not be the same as the evidenced here.

4.9 PERFORMANCE

Deep learning is also seen as hierarchical feature extraction method. Its literature states that deeper networks tend to extract more features and thus the trained model is able to generalize better.

Therefore, this experience is intended to tune three network architectures that can effectively recognize three distinct feature level datasets with a high rate of confidence (one architecture for each dataset). The datasets are: the Fabric textures, the Digits and the German Traffic Signs datasets. It would be impossible to test all the possible network combinations due to the large number of hyper-parameters that can be tuned. The filter size could be one of those possibilities as well as the number of fully-connected units or layers.

So, the start point of this experiment stood in considering the higher feature level dataset (the German Traffic Signs) and vary the number of units on the fully-connected layer. For that, was considered a single layer CNN with only 1 3x3 convolutional layer. Then, the number of fully-connected units assumed base 2 values ranging from 8 to 512 as Figure 63 shows.

From Figure 63 it can be seen that with less than 64 units, the model under-fits because it is not able to learn even in the training data. That's why it is not presented the accuracy values when the number of units is lower than 8. From 64 units, the accuracy on the training set gets around the 100% which means that the model is able to learn the training data. However, at this point, the accuracy on the test set stands around the 75% indicating that the model isn't able to generalize still. Hereupon, the focusing now to the accuracy on the test set, in general, the test accuracy climbed until 256 units and then it decreased. Since the test accuracy got a drop for 512 units, it was not tested with more units to avoid over-fitting to the training data.

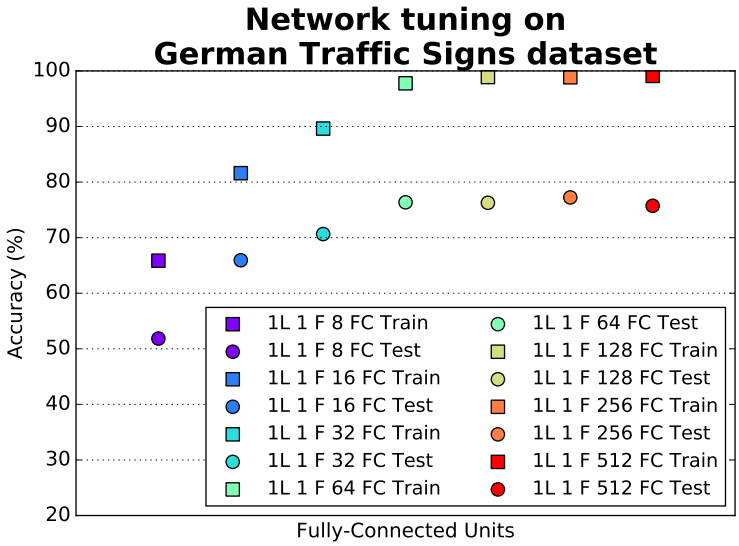


Figure 63: Fully-connected units tune on German Traffic Sign dataset. Squares represent the training accuracy and circles represent the accuracy on the test set.

Table 11 describes the network architectures used to test the model performance taking into account the network depth and the number of filters. The fully-connected units were fixed at 256 based on the results got on Figure 63. In addition, the filter size was also fixed to 3x3 following the principles of the VGG16 network Simonyan and Zisserman (2014) that emphasizes the use of small filters. Since images spatial size is 32x32, are performed two down-sampling operations which reduce its dimension down to 8x8. Basically, these networks are composed by two blocks of convolutional layers. The batch normalization layers helped accelerating the training and normalize the input before feeding it to the next layers. The number of filters ranged from 1x1 to 128x128, assuming base 2 values once again.

To figure out which network configuration would give a reasonable trained models, all the architectures described on Table 11 were trained on the Fabric textures, Digits and German Traffic Signs datasets.

The obtained results are exposed on Figure 64 and they are organized firstly by network depth and then by the number of filters. From left to right, the tested network architectures get deeper. Inside each depth and in the same direction, the number of filters increases and take the values $\{1, 2, 4, 8, 16, 32, 64\}$.

Figure 64 suggests that the models performance tend to improve as the network architecture gets deeper and with more learnable filters. In general, it was registered a considerable improvement each time the network got one more convolutional layer. In general, the best results were obtained for number of filters 32, 64 and 128.

Table 11: Architectures used on 3 different feature level datasets for network tuning, where $N_{filters}$ is in $\{1, 2, 4, 8, 16, 32, 64\}$.

1 **Layer**	2 **Layers**	3 **Layers**	4 **Layers**	5 **Layers**
Conv$N_{filters}$	Conv$N_{filters}$	Conv$N_{filters}$	Conv$N_{filters}$ Conv$N_{filters}$	Conv$N_{filters}$ Conv$N_{filters}$
Max Pooling				
Batch Normalization				
	Conv$N_{filters*2}$	Conv$N_{filters*2}$ Conv$N_{filters*2}$	Conv$N_{filters*2}$ Conv$N_{filters*2}$	Conv$N_{filters*2}$ Conv$N_{filters*2}$ Conv$N_{filters*2}$
Max Pooling				
Batch Normalization				
Fully Connected	256			
Dropout	0.5			
Fully Connected	$N_{classes}$			

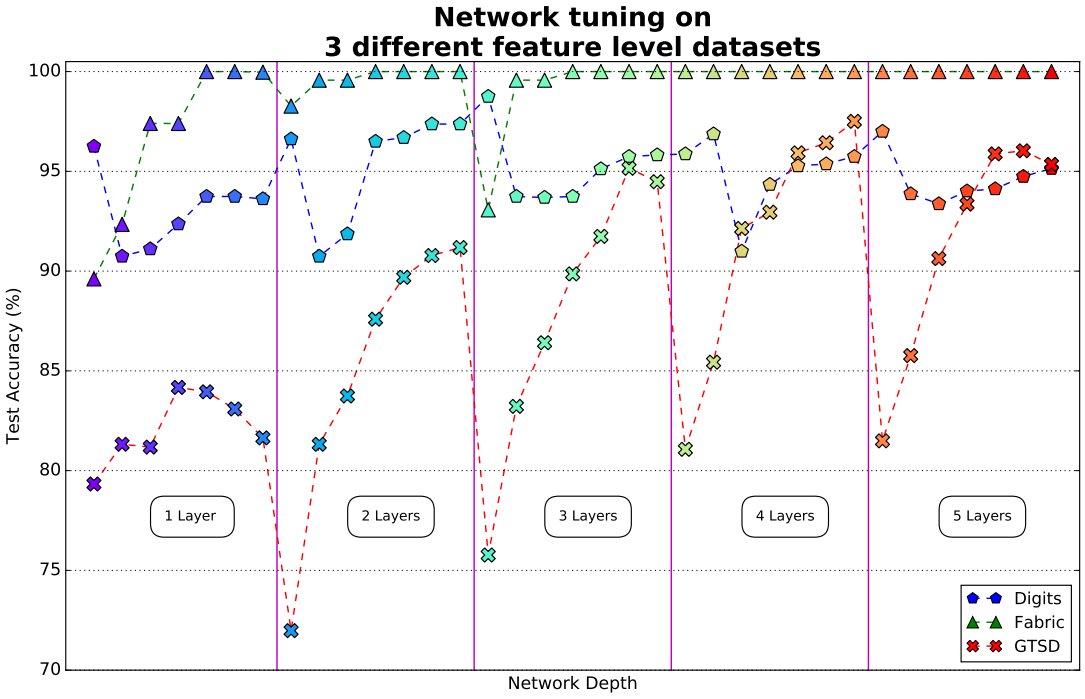


Figure 64: Network tuning on three different feature level datasets: Digits, Fabric textues and German Traffic Signs datasets.

It was shown previously that the Fabric dataset can be learned effectively with a 1 convolutional layer network architecture. The low-level features on the dataset images can be distinguished using 8 filters. Figure 64 indicates a similar behaviour because the test accu-

racy on the Fabric dataset reaches around the 100% immediately on 1 convolutional layer with 16 filters.

The digits dataset images contain medium level features. So, it was expected that for this case it would need a deeper network in order to recognize them. In fact, as Figure 64 evidences the best accuracy was achieved with a 3 convolutional layer depth network. However, strangely in each depth level, the best accuracy was obtained with only 1 or 2 convolutional filters which means that 3 kernels are enough to build up a digit structure effectively.

On the German Traffic Signs dataset, images contain high level features that can result from the combination of lower level features. For every odd number of layers a slightly decrease along with the increase of the number of filters was recorded. For 1 layer depth that decrease emerged already with 16 filters. Yet, for 3 and 5 layers depth the drop occurred on the 64 filters. The best state registered was for 4 convolutional layers depth together with 32, 64 and 128 filters on the respective layer.

Nevertheless, the statements just mention only refer to these datasets, in particular. For instance, for a simpler dataset with lower-level feature images, the peak on test accuracy may occur a little bit sooner. For example, the tiny architecture defined on Table 7 can learn the Fabric dataset with high confidence. By the same reasoning, for a more complex dataset it might be need a deeper network which was the case of the Digits and German Traffic Signs datasets.

5

CONCLUSION

5.1 CONCLUSIONS

Image recognition by deep learning approaches proved to be very successful in diverse contexts. As a machine vision company, Neadvance, Machine Vision, SA is interested on integrating deep learning methods for image recognition on their software because it enables a new range o challenging opportunities.

In conclusion, this dissertation stands as an introduction of deep learning for image recognition. The theoretical fundamentals of CNNs together with the review the state of the art network architectures and benchmarks were essential to understand how CNNs work and how they evolved to the current state.

A step-by-step description of common procedures to train a models serves as a guideline for those who are starting on deep learning. It gives a bunch of hints when it comes to train a model.

The definition of a custom training stop criteria was a crucial step because it was a key point to obtain well trained models once the default stop criteria doesn't seems to be robust enough. Thus, for the validation set the custom training criteria only considers well classified images if their predicted label matches with the ground truth labels and if the classification confidence is greater or equal to a certain value. This forces the model to learn how to separate the data.

Although its randomness, the learning algorithm has consistency enough to give very close results on several runs in the same conditions. Yet, sometimes it can take more or less time to reach the best trained model.

The choice of a suitable color space can determine the model competence on learning the data. In addition, this procedure stood also as a reliable option for dimensionality reduction by squashing color images into a single color channel.

As a hierarchical feature extraction method, the filters learned by CNNs were shown. They get more complex along with the network depth.

The experience of tuning the batch size didn't conclude which is the best batch size for each data set. Instead it hints that the batch size must not be, not too low and not too high

either. Of course, this is an universal statement among researches and it stands as an useful insight about the batch size.

The learning rate tuning supported the supposed theoretical fundamentals since it showed that high learning rates hinders the learning process. Huge jumps on the stochastic gradient descent make it more difficult to reach the combination of weights that minimize the cost function defined as the difference between the network predictions and the ground truth values. Due to that, the model wasn't able to learn the data. By the other hand, the model was capable of learning even in the lowest tested learning rate (it didn't get stuck in a local minima) but it took too much time. This experiment suggested that an adequated learning rate is crucial to let the training evolve properly.

In terms of performance was evidenced the impact of the network depth on the models generalization capacities. Stacking convolutional layers can actually improve the model recognizing capabilities but it goes until a certain point. As soon as the model starts to be too overpowered, its gets each time more over-fitted to the training data and then it is unable to distinguish different but similar data. Beyond that, this experience hinted that along with the increase on the data set images feature level, the network architecture may need to get deeper in order to understand the data.

The proposal for a toolkit was intended to offer a high-level tool that can simplify the entire process of training CNNs for image recognition tasks. So, in it are included methods from the pre-processing and data-augmentation until the network architecture configuration and testing phases. In addition, this toolkit helped on preparing set of experiments that not only suggest ordinary practises when training CNNs but also highlight the generalization capabilities of deep learning for these kind of tasks.

5.2 PROSPECT FOR FUTURE WORK

For future work stand some suggestions about image recognition by deep learning that were not mentioned on this dissertation.

One of the characteristics of deep learning is the huge number of hyper-parameters that can be tuned. Of course, it was not possible to test them all on time. So many others hyper-parameters could be tuned in order to understand their impact on training a model.

The experiences made can also be extended for the remaining datasets. Since each dataset has it own properties, extending the experiences on the remaining datasets would enrich the results consistency.

Currently, the hype around deep learning is outstanding. Besides image recognition, there are other several domains where deep learning methods can be applied on and could not be referred in this dissertation. In the enumeration below are presented some of those

deep learning branches that are have potential to be studied in a future work. Having the knowledge of how CNN work on classifying images makes it easier to learn them.

1. Unlike FNN, RNN allow the data to move to other directions instead of just forward. In addition, they are able to learn temporal dependencies. Dettmers (2016b,a); Goodfellow et al. (2016). Reinforcement learning is associated to RNNs. Typically, it works by setting one or more intelligent agents interacting with an environment and getting feedback from it. Depending on their actions, they are rewarded (or not) and its learning state is updated. Buduma and Locascio (2015). An autonomous agent must learn to perform a task by trial and error, without any guidance from human intervention or labeled data Goodfellow et al. (2016).

2. Dense neural networks are a recent type of neural network architectures based on the concept of residual neural networks. Every layer is connected to each other layer in a feed-forward way, which means that the feature-maps of all preceding layers are used as inputs, and its own feature-maps are used as inputs into all subsequent layers. Still, authors claim that this architecture requires less computation to achieve high performance Huang et al. (2016).

3. Image segmentation is another task related to image recognition. It can be achieved by using fully convolutinal neural networks, for example, in the type of an autoencoder (encoder-decoder) through convolutions and deconvolutions Badrinarayanan et al. (2015), or through other pixel-wise approach Long et al. (2014).

4. Combined with image recognition there are few methods for object localization such as YOLO (You Only Look Once), ROLO (Recurrent You Only Look Once), R-CNN (Regions with Convolutional Neural Networks) and SSD (Single Shot Detection). So, besides classifying, these type of networks output the localization of the objects present on it.

5. Image caption combines CNN with RNN. CNNs are responsible to recognize regions of the image while RNNs attribute a description to it Karpathy and Fei-Fei (2015).

Nevertheless, there are also some points about the developed toolkit that could be improved.

The integration of a graphic user interface would be useful for users that don't appreciate working directly through the commands line. On it, all the settings of selecting a dataset, prepare the data, configure a network architecture, train, test the trained model... and so on, could be done with a few clicks instead of just typing on the keyboard. In fact, it would make the toolkit much more user-friendly.

BIBLIOGRAPHY

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

Simon Achatz. State of the art of object recognition techniques, July 2016. URL https://www.nst.ei.tum.de/fileadmin/w00bqs/www/publications/as/SS2016_AS_ObjectRecogn.pdf.

Oliveira L. S. Silva Jr E. Britto Jr A. Koerich A. Almeida, P. Pklot – a robust dataset for parking lot classification. Technical report, Universidade Federal do Paraná (UFPR), Curitiba, PR, Brazil and Pontifícia Universidade Católica do Paraná (PUCPR), Curitiba, PR, Brazil and Universidade Estadual de Ponta Grossa(UEPG), Ponta Grossa, PR, Brazil, 2015. URL http://www.inf.ufpr.br/lesoliveira/download/pklot-readme.pdf.

Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation, November 2015. URL https://arxiv.org/pdf/1511.00561v1.pdf.

Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. Comparative study of deep learning software frameworks, March 2016. URL https://arxiv.org/pdf/1511.06435v3.pdf. Research and Technology Center, Robert Bosch LLC.

Jon Barker. Deep learning on gpus, March 2016. URL http://on-demand.gputechconf.com/gtc/2015/webinar/deep-learning-course/intro-to-deep-learning.pdf.

L. Bottou, Y. LeCun, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition, November 1998. URL http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf.

Denny Britz. Recurrent neural networks tutorial, part 1 – introduction to rnns, September 2015. URL http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/.

Nikhil Buduma and Nicholas Locascio. Fundamentals of deep learning (first edition), November 2015.

Alfredo Canziani, Eugenio Culurciello, and Adam Paszke. An analysis of deep neural networks models for practical applications, April 2017. URL https://arxiv.org/pdf/1605.07678.pdf.

Eugenio Culurciello. The history of neural networks, April 2017. URL http://dataconomy.com/2017/04/history-neural-networks/.

Aymeric Damien et al. Tflearn. https://github.com/tflearn/tflearn, 2016.

Tim Dettmers. Deep learning in a nutshell: Core concepts, November 2015a. URL https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-core-concepts/.

Tim Dettmers. Deep learning in a nutshell: History and training, December 2015b. URL https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-history-training/.

Tim Dettmers. Understanding convolution in deep learning, March 2015c. URL http://timdettmers.com/2015/03/26/convolution-deep-learning/.

Tim Dettmers. Deep learning in a nutshell: Reinforcement learning, September 2016a. URL https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-reinforcement-learning/.

Tim Dettmers. Deep learning in a nutshell: Sequence learning, March 2016b. URL https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-sequence-learning/.

Hongyang Gao, Hao Yuan, Zhengyang Wang, and Shuiwang Ji. Pixel deconvolutional networks. *arXiv preprint arXiv:1705.06820*, 2017. URL https://arxiv.org/pdf/1705.06820.pdf.

Xavier Gastaldi. Shake-shake regularization, May 2017. URL https://arxiv.org/pdf/1705.07485.pdf.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016. URL http://www.deeplearningbook.org.

Céline Gravelines. Deep learning via stacked sparse autoencoders for automated voxel-wise brain parcellation based on functional connectivity, April 2014. URL http://ir.lib.uwo.ca/cgi/viewcontent.cgi?article=3503&context=etd. Western University.

Luiz Gustavo Hafemann. An analysis of deep neural networks for texture classification, 2014. URL http://www.inf.ufpr.br/lesoliveira/download/LGHafemannMSC.pdf.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015a. URL https://arxiv.org/pdf/1502.01852.pdf.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015b. URL https://arxiv.org/pdf/1512.03385v1.pdf.

Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks, August 2016. URL https://arxiv.org/pdf/1608.06993.pdf.

Jen-Hsun Huang. Accelerating ai with gpus: A new computing model, January 2016. URL https://blogs.nvidia.com/blog/2016/01/12/accelerating-ai-artificial-intelligence-gpus/.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, March 2015. URL https://arxiv.org/pdf/1502.03167.pdf.

Ujjwal Karn. An intuitive explanation of convolutional neural networks, August 2016. URL https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/.

Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions, April 2015. URL https://arxiv.org/pdf/1412.2306v2.pdf.

Andrej Karpathy and Justin Johnson. Convolutional neural networks for visual recognition, 2016. URL http://cs231n.github.io/.

Richard Knoche. Automatic image quality assessment, February 2017. URL http://www.dealingdata.net/2017/02/05/Automatic-Image-Quality-Assessment/.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, Massachusetts Institute of Technology and New York University, April 2009. URL https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

Alex Krizhevsky. High-performance c++/cuda implementation of convolutional neural networks, June 2014. URL https://code.google.com/archive/p/cuda-convnet2/.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks, December 2012. URL https://www.nvidia.cn/content/tesla/pdf/machine-learning/imagenet-classification-with-deep-convolutional-nn.pdf.

Andrey Kurenkov. A 'brief' history of neural nets and deep learning, part 1, December 2015. URL http://www.andreykurenkov.com/writing/a-brief-history-of-neural-nets-and-deep-learning/.

Dan Kuster. The good, bad, & ugly of tensorflow, May 2016. URL https://indico.io/blog/the-good-bad-ugly-of-tensorflow/.

Gustaf Kylberg. The kylberg texture dataset v. 1.0. External report (Blue series) 35, Centre for Image Analysis, Swedish University of Agricultural Sciences and Uppsala University, Uppsala, Sweden, September 2011. URL http://www.cb.uu.se/~gustaf/texture/.

Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The mnist database of handwritten digits. URL http://yann.lecun.com/exdb/mnist/. Courant Institute, NYU and Google Labs, New York and Microsoft Research, Redmond.

Min Lin, Qiang Chen, and Shuicheng Yan. Network in network, March 2014. URL https://arxiv.org/pdf/1312.4400.pdf.

Jonathan Long, Evan Shelhamer, and Trevor Darrel. Fully convolutional networks for semantic segmentation, November 2014. URL https://arxiv.org/pdf/1411.4038.pdf.

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, and Andrew Y. Ng Bo Wu. Reading digits in natural images with unsupervised feature learning, 2011. URL http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf.

Michael Nielsen. Neural networks and deep learning. URL http://neuralnetworksanddeeplearning.com/.

Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning deconvolution network for semantic segmentation, 2015. URL https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/Noh_Learning_Deconvolution_Network_ICCV_2015_paper.pdf.

Augustus Odena, Vincent Dumoulin, and Chris Olah. Deconvolution and checkerboard artifacts. *Distill*, 2016. doi: 10.23915/distill.00003. URL http://distill.pub/2016/deconv-checkerboard.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and

Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y. URL http://www.image-net.org/challenges/LSVRC/.

Jürgen Schmidhuber. Deep learning in neural networks: An overview, October 2014. URL https://arxiv.org/pdf/1404.7828v4.pdf.

Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking state-of-the-art deep learning software tools, September 2016. URL https://arxiv.org/pdf/1608.07249v5.pdf. Department of Computer Science, Hong Kong Baptist University.

David Silver. Tensorflow vs. tf learn vs. keras vs. tf-slim, September 2016. URL https://medium.com/self-driving-cars/tensorflow-vs-tf-learn-vs-keras-vs-tf-slim-b83811966020#.o2urhyw3i.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. URL https://arxiv.org/pdf/1409.1556v1.pdf.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting, June 2014. URL http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf.

Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. The German Traffic Sign Recognition Benchmark: A multi-class classification competition. In *IEEE International Joint Conference on Neural Networks*, pages 1453–1460, 2011. URL http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset.

Alex Staravoitau. Traffic signs classification with a convolutional network, January 2017. URL https://navoshta.com/traffic-signs-classification/.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2014. URL http://arxiv.org/abs/1409.4842.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision, 2015. URL https://arxiv.org/pdf/1512.00567.pdf.

Ziring Tawfique. Tool-mediated texture recognition using convolutional neural network, September 2016. URL https://uu.diva-portal.org/smash/get/diva2:973971/FULLTEXT01.pdf.

Deeplearning4j Development Team. Deeplearning4j: Open-source distributed deep learning for the jvm. URL http://deeplearning4j.org.

Vincent Vanhoucke. Deep learning course by google. URL https://www.udacity.com/course/deep-learning--ud730.

Stefan Wager, Sida Wang, and Percy Liang†. Dropout training as adaptive regularization, June 2014. URL http://papers.nips.cc/paper/4882-dropout-training-as-adaptive-regularization.pdf.

Li Wan, Matthew Zeiler, Sixin Zhang, Yann LeCun, and Rob Fergus. Regularization of neural networks using dropconnect, 2013. URL http://cs.nyu.edu/~wanli/dropc/dropc.pdf.

Haohan Wang and Bhiksha Raj. On the origin of deep learning, March 2017. URL https://arxiv.org/pdf/1702.07800.pdf.

Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks, November 2013. URL https://arxiv.org/pdf/1311.2901v3.pdf.

Matthew D. Zeiler, Dilip Krishnan, Graham W. Taylor, and Rob Fergus. Deconvolutional networks, June 2010. URL http://www.matthewzeiler.com/wp-content/uploads/2017/07/cvpr2010.pdf.

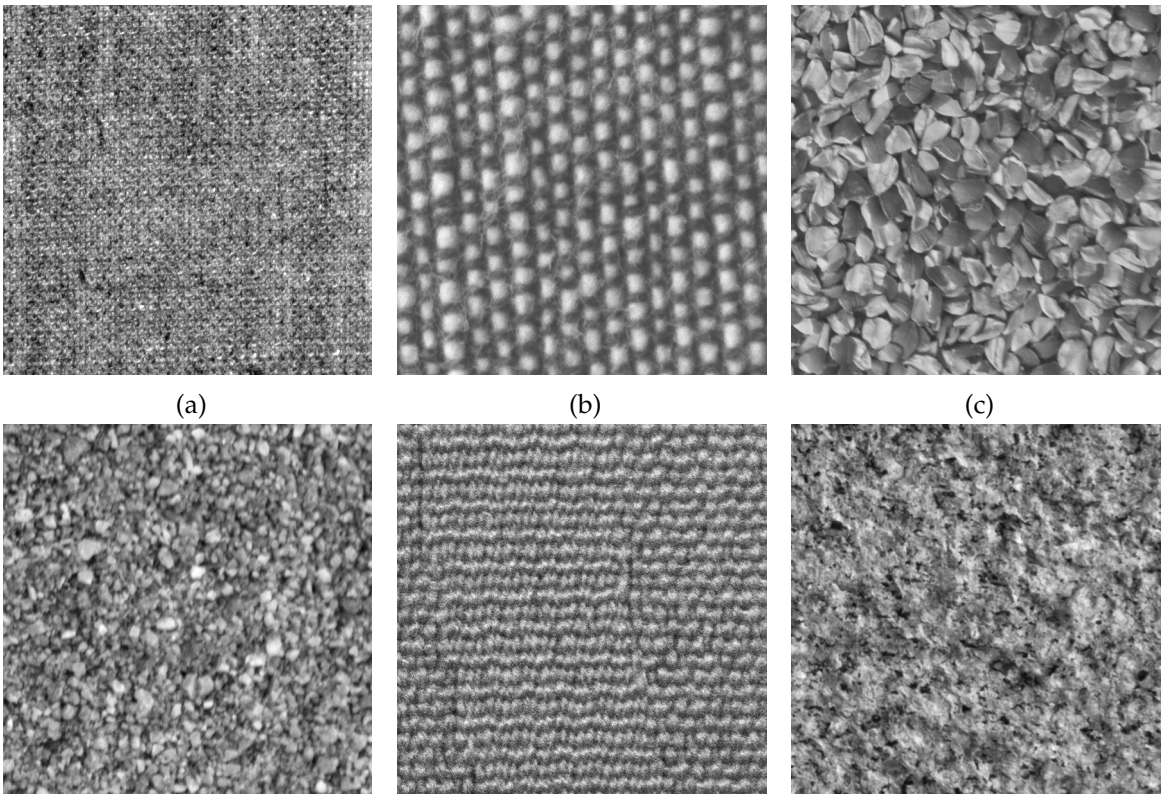
Ke Zhang, Miao Sun, Tony X. Han, Xingfang Yuan, Liru Guo, and Tao Liu. Residual networks of residual networks: Multilevel residual networks, 2017. URL https://arxiv.org/pdf/1608.02908.pdf.

A

SUPPORT MATERIAL

A.1 INPUTS

A.1.1 *Subset of kylberg texture dataset*

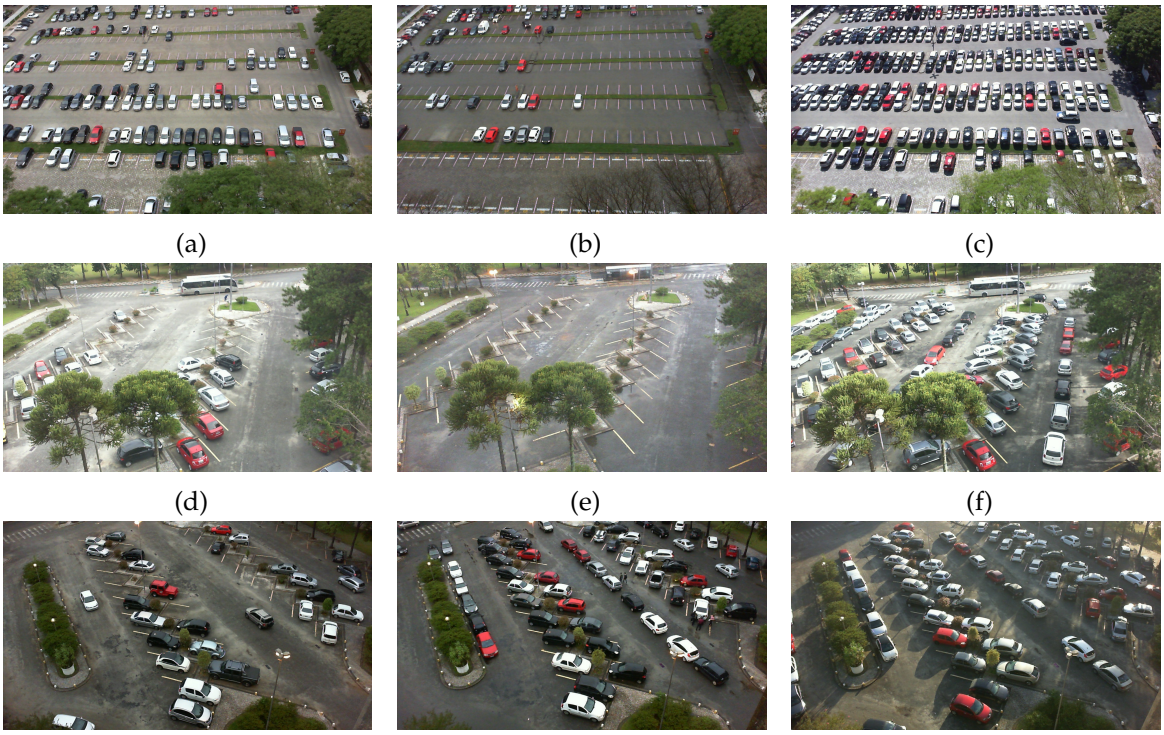


(a) (b) (c)

(d) (e) (f)

Figure 65: Subset of kylberg texture dataset represented with an example of each class: (a) Canvas; (b) Cushion; (c) Linseeds; (d) Sand; (e) Seat and (f) Stone.

A.1.2 *Parking lot*



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

Figure 66: Three parking lot perspectives (each line represents one perspective) on different weather conditions: (a) (d) (g) Cloudy; (b) (e) (h) Rainy and (c) (f) (i) Sunny.

A.2 NETWORK ARCHITECTURES

A.2.1 *GoogLeNet*

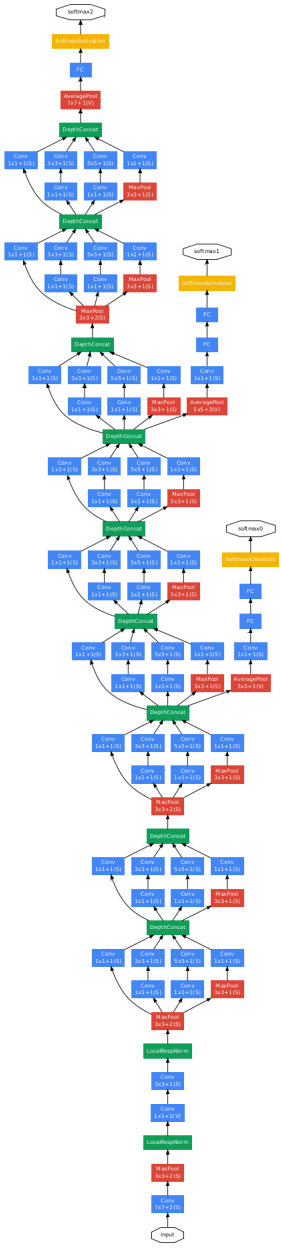


Figure 67: GoogLeNet network architecture in higher resolution. Image by Szegedy et al. (2014).

A.3 PROPOSED TOOLKIT

The proposed toolkit root directory is structured as Figure 68 demonstrates. Down below is a short description about each sub-directory.

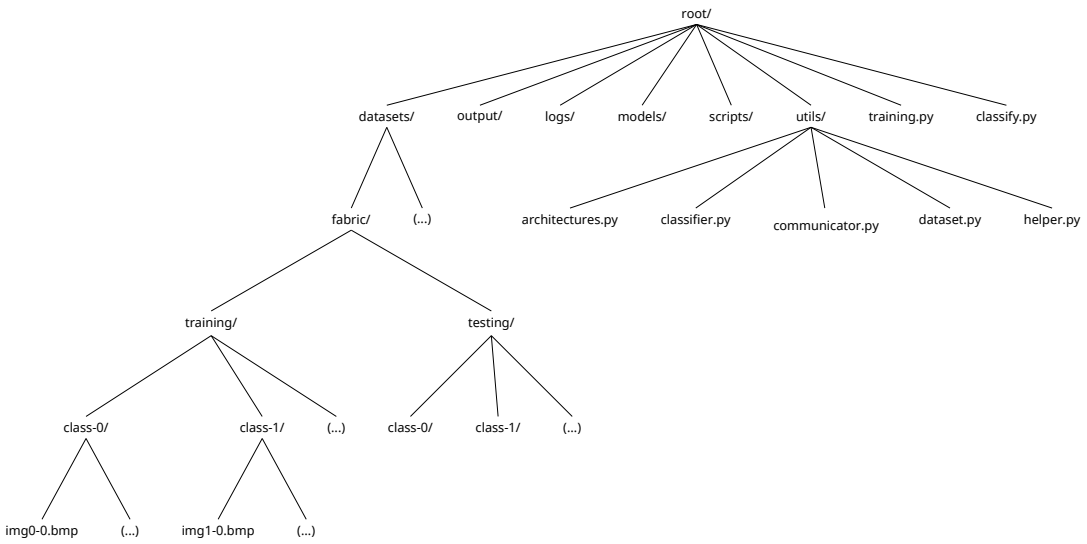


Figure 68: Developed deep learning toolkit directory tree.

- The **datasets** folder is used to store the data (in this case images) to be used for training, validation and testing. This directory aims to simplify the data input, however, if the data is stored on other directory, the toolkit also accepts the most convenient path (either relative or absolute). Inside the training data directory, the program expects that classes are separated in sub-directories. For example, class-0 images should be in a folder, class-1 images in other folder, and so on. It is a practical way of labeling data too. The same rule applies to the validation and testing sets. Another alternative is to load the images set by a index. This file must be in a comma-separated-values (CSV) format where the first column corresponds to the global path to the image and the second column to its respective class or label.

- The **output** folder is meant to store report files that result from training and testing. These files are created with the aim of letting the user know the evolution of the learning, how the training is evolving and figure out the network's inference capabilities. The report files are stored in CSV format and contain the training, validation, test accuracy and the elapsed time. By default, each write is done at every 5 epochs. Yet, the user can set the snapshot frequency to a higher/lower value. By one hand, a lower value, will increase the number of network's evaluations and consequently increase

the training time. It will also give a good resolution how the learning process is evolving. By the other hand, with a higher value, the resolution of the learning evolution is less precise. That's why it is important to tune an intermediate value that does not overload the training operation with evaluation and simultaneously offers a good representation of the learning progress.

```
################ TRAINING REPORT #################
# Images path    | <path_to_dataset>
# Validation     | 0.30
# Images shape   | (<height>,<width>,<channels>)
# Architecure    | <architecture_name>
# Bacth Size     | <batch_size>
# Snap/Epoch     | 5
# Max epochs     | 1000
# Eval criteria  | 0.75
################################################
train,validation,test,time
0.00,0.00,0.00,0.00
79.87,78.23,72.86,66.10
92.00,90.61,85.54,130.89
100.00,99.13,98.21,199.34
          (...)
```

Listing A.1: Snippet of a training report file.

- The **logs** directory holds the original Tensorflow output logs that can be viewed on *Tensorboard*. It allows to visualize the network architecture through a node graph representation. *Tensorboard* can be initialized by the following command:

```
> tensorboard --logdir="path/to/log/file_or_dir"
```

Listing A.2: How to initialize *Tensorboard* to visualize training logs.

This command creates a local server that can be accessed by an internet browser.

- By default, the network's trained models are saved on the **models** directory. This feature is very useful because it allows the user to pick up those trained models and do whatever he wants, whether for continuing the training or immediately using it for image classification tasks.

- The **scripts** folder contains a bunch of auxiliary scripts that can be helpful for the user in certain situations. Sometimes, the original datasets cannot be directly fed to the network due to its properties. So they may need some pre-processing, like cropping or resizing, in order to suit properly on the network.

- The **utils** directory contains a set of modules and a script that are critical for the good execution of the toolkit. The modules inside this folder are imported by the training and testing scripts in order to scope things to a even higher level of abstraction and minimize the number of lines needed to do a certain action. This was meant to maintain a simple and readable code.

  – The module *architectures.py* has a main function that matches an architecture nomenclature with a network topology. Each network topology has its definition, like the configuration of the convolutional, max-pooling and fully-connected layers. The user can create/edit the network topology that he wants. Listing A.3 has a snippet of how to configure the Alexnet Krizhevsky et al. (2012) architecture on the toolkit. The function *build network* is responsible for matching the architecture name to its definition.

```
import tflearn

# alexnet network architecture
def build_alexnet(nclasses):
    net = tflearn.input_data(shape=[None, 224, 224, 3],
                             data_preprocessing=img_prep
                             data_augmentation=data_aug)

    net = tflearn.conv_2d(net, nb_filters=96, filter_size=11, strides=4,
                          activation='relu', padding='same')
    net = tflearn.max_pool_2d(net, kernel_size=3, strides=2)
    net = tflearn.local_response_normalization(net)

    net = tflearn.conv_2d(net, nb_filters=256, filter_size=5, strides=1,
                          activation='relu', padding='same')
    net = tflearn.max_pool_2d(net, kernel_size=3, strides=2)
    net = tflearn.local_response_normalization(net)

    net = tflearn.conv_2d(net, nb_filters=384, filter_size=3, strides=1,
                          activation='relu', padding='same')
    net = tflearn.conv_2d(net, nb_filters=384, filter_size=3, strides=1,
                          activation='relu', padding='same')
    net = tflearn.conv_2d(net, nb_filters=256, filter_size=3, strides=1,
                          activation='relu', padding='same')
    net = tflearn.max_pool_2d(net, kernel_size=3, strides=2)
    net = tflearn.local_response_normalization(net)

    net = tflearn.fully_connected(net, n_units=4096, activation='relu')
    net = tflearn.dropout(net, keep_prob=0.5)
    net = tflearn.fully_connected(net, n_units=4096, activation='relu')
    net = tflearn.dropout(net, keep_prob=0.5)
    net = tflearn.fully_connected(net, n_units=nclasses,
```

```
33                                      activation='softmax')
34
35      net = tflearn.regression(net, optimizer='momentum',
36                               loss='categorical_crossentropy',
37                               learning_rate=0.001)
38
39      return net
40
41  # vgg16 network architecture
42  def build_vgg16():
43      (...)
44      return net
45
46  # general network architecture builder
47  def build_network(architecture_name, nclasses):
48      if(architecture_name == "alexnet"):
49          net = build_alexnet(nclasses)
50      elif(architecture_name == "vgg16"):
51          net = build_vgg16(nclasses)
52      (...)
53      else:
54          net = None
55
56      return net
```

Listing A.3: Snippet of how to configure the Alexnet network architecture on the proposed toolkit *architectures.py* module.

- The module *classifier.py* contains multiple classification methods including one by sliding window. For instance, this method, allows the classification of high resolution images that can't be directly fed to the network. So, firstly they are splitted into smaller images and then feeded to the network. Other option is to load a trained model for classification as a local TCP/IP server that can be accessed remotely in a local area network. The need of this module appeared from many desynchronized versions of multiple scripts that used the same classification algorithm. Thus, from now, when something need to be updated/fixed, it is done just once.
- The *communicator.py* script creates a socket communication with a local server. Loading a moderated size trained models takes a few seconds and it can be a little frustrating to make it several times. The module *classifier.py* has a method to create a local server that loads a trained model once and then the communicator script is used to communicates with the local server by sending the path to the image that will be classified and its respective class label.

  - The *dataset.py* module is responsible for loading the training and testing data and shape it according to the network's definition. This module can also be used to pre-processing and data augmentation operations.
  - The *helper.py* module has a set of auxiliary functions to make the code even cleaner. For example, it contains functions that are responsible for printing to screen or to file and check if certain stop criteria are attained or monitor the hardware resources being used.

- The **training.py** script suffered several updates along the toolkit implementation. Prior versions only allowed the training of the network and saving of the resulting models. The current version mixes both training and testing. To be clearer, during the training, the validation and test set are also evaluated in the current model. Whenever it reaches a new best learning state, the model is saved on disk. By default, the training script considers that a correct classification on the validation set has a confidence of, at least, 75%. Once the accuracy on the validation set reaches a value of 97.5% (considering the classificaton confidence threshold). If there is no progress in the learning process in 250 epochs, the training is automatically stopped and the model is saved on disk. The maximum number of training epochs is set to 1000. Once again, the user can disable this option or create a custom stop criteria with other constraints more interesting to their specific problems.

```
usage: training.py [-h] --data_dir DATA_DIR --arch ARCH --run_id RUN_ID
                   [--bsize BSIZE] [--val_dir VAL_DIR]
                   [--test_dir TEST_DIR] [--height HEIGHT]
                   [--width WIDTH] [--val_set VAL_SET] [--gray GRAY]
                   [--snap SNAP] [--pproc PPROC] [--aug AUG [AUG ...]]
                   [--n_epochs N_EPOCHS] [--eval_crit EVAL_CRIT]

High level deep learning toolkit training script.

list of arguments:
  -h, --help            show this help message and exit
  --data_dir DATA_DIR   <REQUIRED> directory to the training data
  --arch ARCH           <REQUIRED> architecture name
  --run_id RUN_ID       <REQUIRED> run identifier
  --bsize BSIZE         batch size (default=16)
  --val_dir VAL_DIR     directory to the validation data (default=None)
  --test_dir TEST_DIR   directory to the testing data (default=None)
  --height HEIGHT       images height (default=64)
  --width WIDTH         images width (default=64)
  --val_set VAL_SET     percentage of training data to validation
                        (default=0.3)
  --gray GRAY           convert images to grayscale (default=False)
  --snap SNAP           evaluate training/validation/test sets
```

```
                        frequency (default=5)
  --pproc PPROC         enable/disable pre-processing (default=True)
  --aug AUG [AUG ...]   enable data augmentation (default=[])
  --n_epochs N_EPOCHS   maximum number of training epochs (default=1000)
  --eval_crit EVAL_CRIT classification confidence (default=0.75)
```

Listing A.4: Training script help menu.

On Listing A.5 is presented a minimalist snippet of code that explains how the training script is structured.

```
import tflearn
from utils import architecture, dataset, classifier, helper

# load data
X, Y, nc = dataset.load_dataset('path/to/training_data/')
Xv, Yv   = dataset.load_dataset('path/to/validation_data/')
Xt, Yt   = dataset.load_dataset('path/to/testing_data/')

# build network
network = architecture.build_network(architecture_name='architecture_name',
                                     nclasses=nc)
# build model
model = tflearn.DNN(network)

# variables to help controling the training stop criteria
epoch            = 0
best_testing_acc = 0.0
no_progress      = 0
snapshot         = 5

# train model at maximum of 1000 epochs
while epoch < 1000:
    train_acc = classifier.my_evaluate(X, Y)
    # calculate validation set accuracy with classification
    # confidence above 75%
    val_acc   = classifier.my_evaluate(Xv, Yv, confidence=0.75)
    test_acc  = classifier.my_evaluate(Xt, Yt)
    helper.print_accuracy(train_acc, val_acc, test_acc, on_file=True)

    # if reached a new best learning state
    # - resets no learning progress counter
    # - saves current best trained model
    if(test_acc > best_testing_acc):
        no_progress = 0
        model.save('path/to/save/trained_model')
    else:
```

```
    # else, increments the no learning progress counter
        no_progress += snapshot

    # check if validation accuracy has achieved a specific value  OR
    # if the no learning progress counter reached the limit
    if(val_acc > 97.5 or no_progress > 250):
        break

    # train for intervals of 'snapshot' epochs
    model.fit(X, Y, n_epoch=snapshot, batch_size=16, run_id='run_ID',
              validation_set=(Xv, Yv))
    epoch += snapshot

# print training, validation and testing accuracy on screen
helper.print_accuracy(train_acc, val_acc, test_acc, on_screen=True)
```

Listing A.5: Training script minimalist sample.

- The **classify.py** script is meant for testing a trained model on the field. It relies on the classification methods that are defined on the *utils/classifier.py* module described previously. This script is essentially used like a test application template that is edited depending on the requirements of the problem.

```
usage: classify.py [-h] --data_dir DATA_DIR --arch ARCH --model MODEL
                   [--height HEIGHT] [--width WIDTH] [--gray GRAY]
                   [--pproc PPROC]  [--eval_crit EVAL_CRIT]
                   [--aug AUG [AUG ...]] [--video VIDEO]
                   [--save SAVE]

High level deep learning toolkit classifying script.

list of arguments:
  -h, --help            show this help message and exit
  --data_dir DATA_DIR   <REQUIRED> directory to the testing data
  --arch ARCH           <REQUIRED> architecture name
  --model MODEL         <REQUIRED> trained model path
  --height HEIGHT       images height (default=64)
  --width WIDTH         images width (default=64)
  --gray GRAY           convert images to grayscale (default=False)
  --pproc PPROC         enable/disable pre-processing (default=True)
  --eval_crit EVAL_CRIT classification confidence (default=0.75)
  --aug AUG [AUG ...]   enable data augmentation (default=[])
  --video VIDEO         use video capture device/video file (default=0)
  --save SAVE           save output image (default=False)
```

Listing A.6: Classifying script help menu.

Both training and the classifying scripts, as well as many others auxiliary scripts, when called in the command prompt with the '-h' argument, give a minimalist hint about their usage, as Listings A.4 and A.6 suggest.