A CONVOLUTIONAL NEURAL NETWORK FOR THE MNIST DATASET USING TENSORFLOW

Project report

FABIO LOCHE - STEFANO TEDESCHI

Corso di Reti Neurali
Anno Accademico 2016/2017

Corso di Laurea Magistrale in Informatica
Scuola di Scienze della Natura
Università degli Studi di Torino

ABSTRACT

Deep Neural Networks (DNNs) are becoming a central topic in machine learning research. They can be used for a large amount of complex tasks, including automatic speech recognition, image recognition, natural language processing, recommendation systems, but also toxicology, bioinformatics, and neuroscience. In these contexts, many different frameworks for deep learning have been released; among them Google's TensorFlow provides a complete and well documented toolbox for programming DNNs.

The aim of this report is to analyze a widely used kind of deep architecture, the Convolutional Neural Network, as well as briefly introduce the TensorFlow framework. In the second part a simple model for the MNIST dataset of handwritten digits built with TensorFlow is presented and some conclusions about the architecture of the network are drawn.

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [**knuth:1974**]

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

DNN	Deep Neural Network
CNN	Convolutional Neural Network
MLP	Multylayer Perceptron
ReLU	Rectified Linear Units
NLP	Natural Language Processing
TF	TensorFlow
CPU	Central Processing Unit
GPU	Graphical Processing Unit
CUDA	Compute Unified Device Architecture
GUI	Graphical User Interface

1 INTRODUCTION

Machine learning is becoming more and more important in a huge number of domains. Softwares able to increase their performances by means of knowledge about the world are now used in smartphones as well as social networks, search engines and intelligent cars, too. At the same time, tasks are becoming increasingly hard and powerful models have to be developed.

Deep learning is the branch of machine learning based on a set of algorithms that attempt to model high level abstractions in data. Research in this area attempts to make better representations and create models to learn these representations from large-scale unlabeled data. Some of the representations are inspired by advances in neuroscience and are loosely based on interpretation of information processing and communication patterns in a nervous system, such as neural coding which attempts to define a relationship between various stimuli and associated neuronal responses in the brain.

In this context neural networks, and in particular deep neural networks, play a fundamental role. Various deep learning architectures such as convolutional deep neural networks, deep belief networks and recurrent neural networks have been applied to fields like computer vision, automatic speech recognition, natural language processing, audio recognition and bioinformatics where they have been shown to produce state-of-the-art results on various tasks. Actually, in many of these fields the amount of data to be processed is too large to use traditional Multi Layer Feed Forward Neural Networks with success.

At the same time, in the last few years, a lot of different frameworks for deep learning have been released. For instance, in 2014 Berkeley Vision and Learning Center released Caffe [**jia2014caffe**]. Theano [**2016arXiv160502688full**] is a numerical computation library for Python developed by a machine learning group at the Université de Montréal. Microsoft, too, developed CNTK, its own deep learning framework.

Among all these tools Google's TensorFlow deserves a special mention. It is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows to deploy the computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. TensorFlow was originally developed by researchers and engineers working on the Google Brain Team, within Google's Machine Intelligence research organization, for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well. Actually, nowadays dozens of commercial products by Google use TensorFlow: speech recognition, Gmail, Google Photos, Google Search, and so on.

The aim of our project was to learn some basic concepts about convolutional neural network as well as familiarize with the TensorFlow framework. In particular, we tried to deploy a convolutional model built with TensorFlow following the official tutorial. Since we were interested in the field of image recognition, our model was focused on the MNIST dataset of handwritten digits. The network is designed for recognizing handwritten digits and, if properly tuned, reaches pretty satisfying levels of accuracy. A simple Graphical User Interface (GUI) was developed, too, in order to test the trained network. We then tried to tune the network in order to reach a good trade-off between accuracy and training time, acting upon both the architecture and the network hyperparameters.

This report is organized as follows:

CHAPTER 2 introduces the main features and building blocks of convolutional neural networks, with special regard to the parameters that have to be taken into account when designing one of them;

CHAPTER 3 is focused on the TensorFlow frameworks; the main datastructures and available operations are presented and a very simple model is shown;

CHAPTER 4 describes the model built for the MNIST dataset and the experiments made on that architecture;

CHAPTER 5 shows the main results obtained during the experiments, with respect to the accuracies reached with the different tested solutions;

CHAPTER 6 finally draws some conclusions about the experiments, and about convolutional neural networks in general, and proposes a possible direction for future work.

2 CONVOLUTIONAL NEURAL NETWORKS

Convolutional Neural Networks (CNNs) are a kind of deep artificial feed-forward neural network in which the organization of the connections between the neurons is inspired by the animal visual cortex. Actually, individual cortical neurons respond to stimuli in a restricted region of space known as **receptive field**. The receptive fields of different neurons partially overlap such that they tile the whole visual field. Similarly, in CNNs each neuron is connected only with a small subset of inputs from the previous layer.

These networks are extremely useful when dealing with data with a grid-like topology, such as time-series data or images.

The name "convolutional neural network" indicates that the network employs a specialized kind of linear operation called **convolution**. In short, as stated in [**Goodfellow-et-al-2016**], *convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers*.

This chapter starts with a list of the main features of every convolutional neural network. Then, the building blocks of the CNN architecture are analyzed with particular attention to the hyperparameters tuning. In the last part, some applications of this model are proposed.

2.1 MAIN FEATURES

While traditional Multylayer Perceptron (MLP) models were successfully used in the past for image recognition, they now suffer from the curse of dimensionality due to the full connectivity between nodes and, thus, they do not scale well to higher resolution images.

For instance, in the CIFAR-10 dataset, images are of size 32x32x3 (32 wide, 32 high, 3 color channels). A single fully connected neuron in the first hidden layer of a regular MLP would have $32 * 32 * 3 = 3,072$ weights. A 200x200 image, however, would lead to neurons that have $200 * 200 * 3 = 120,000$ weights. Moreover, such network architecture does not take into account the spatial structure of data, treating input pixels which are far apart or close together exactly in the same way. The full connectivity of neurons is wasteful in the framework of image recognition, and the huge number of parameters quickly leads to overfitting.

As said before, convolutional neural networks are biologically inspired variants of multilayer perceptrons, designed to emulate the behavior of the visual cortex. These models mitigate the challenges posed by the MLP architecture by exploiting the strong spatially lo-

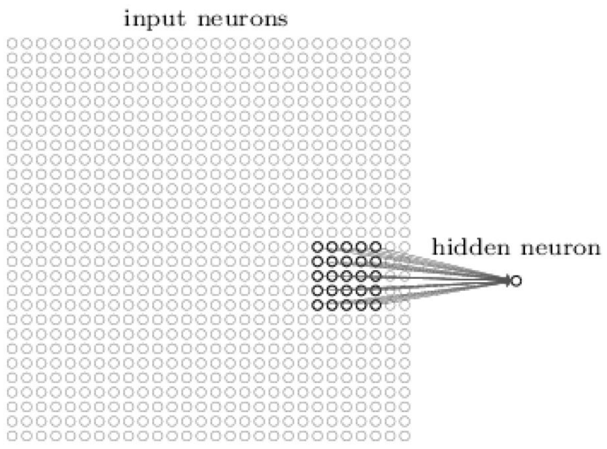


Figure 1: Implementation of a 5x5 receptive field in a CNN

cal correlation present in natural images. In particular, CNNs have the following distinguishing features:

3D VOLUMES OF NEURONS The layers of a CNN have neurons arranged in 3 dimensions: width, height and depth. Neurons inside a layer are only connected to a small region of the layer before it, called a receptive field. Figure 1, for instance, shows a 5x5 receptive field from the input neurons to the first hidden layer, with a 28x28 input. Distinct types of layers, both locally and completely connected, are stacked to form the CNN architecture.

LOCAL CONNECTIVITY Following the concept of receptive field, CNNs exploit spatially local correlation by enforcing a local connectivity pattern between neurons of adjacent layers. The architecture, thus, ensures that every **filter** (i. e. weight patch) learnt produces the strongest response to a spatially local input pattern. Stacking many of such layers leads to non-linear filters that become increasingly "global" (i. e. responsive to a larger region of input space). This allows the network to first create good representations of small parts of the input, then assemble representations of larger areas from them.

SHARED WEIGHTS In CNNs, each filter is replicated across the entire visual field. These replicated units share the same parameterization (weight vector and bias) and produce a feature map. This means that all the neurons in a given convolutional layer detect exactly the same features. Replicating units in this way allows for features to be detected regardless of their position in the visual field, thus constituting the property of translation invariance.

Together, these properties allow convolutional neural networks to achieve better generalization performances in vision problems. More-

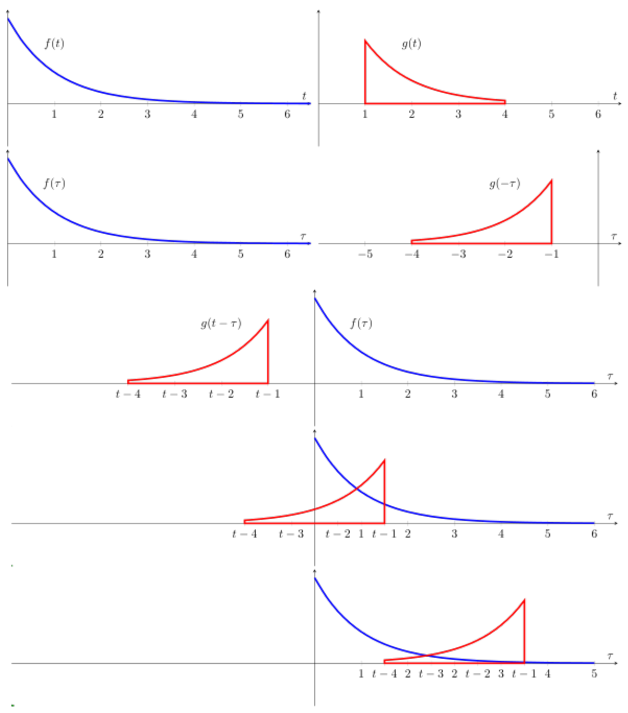


Figure 2: Visual explanation of the convolution operation

over, the weight sharing helps by dramatically reducing the number of free parameters being learnt, thus lowering the memory requirements for running the network. Decreasing the memory footprint allows the training of larger and more powerful networks.

2.2 THE CONVOLUTION OPERATION

In its most general form, convolution is an operation on two functions of a real-valued argument. It produces a third function, that is typically viewed as a modified version of one of the original functions, giving the integral of the point-wise multiplication of the two functions as a function of the amount that one of the original functions is translated. It is typically denoted with an asterisk and it is defined as:

$$s(t) = (f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\, d\tau$$

The convolution formula can be described as a weighted average of the function $f(\tau)$ at moment t where the weighting is given by $g(-\tau)$ simply shifted by amount t. As t changes, the weighting function emphasizes different parts of the input function. The output is sometimes referred to as **feature** or **activation map**.

Figure 2 shows a visual explanation of the operation. Wherever the two functions intersect, the integral of their product is found. In other words, it computes a sliding, i. e. a weighted-sum of function $f(\tau)$, where the weighting function is $g(-\tau)$.

If we now assume that f and g are defined only on integer t, we can define the discrete convolution as:

$$s(t) = (f * g)(t) = \sum_{-\infty}^{\infty} f(\tau)g(t-\tau)$$

In convolutional networks terminology, the first argument of the convolution is often referred to as input and the second as **kernel** or **filter**. In machine learning applications, the input is usually a multidimensional array (i. e. a tensor) of data and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm.

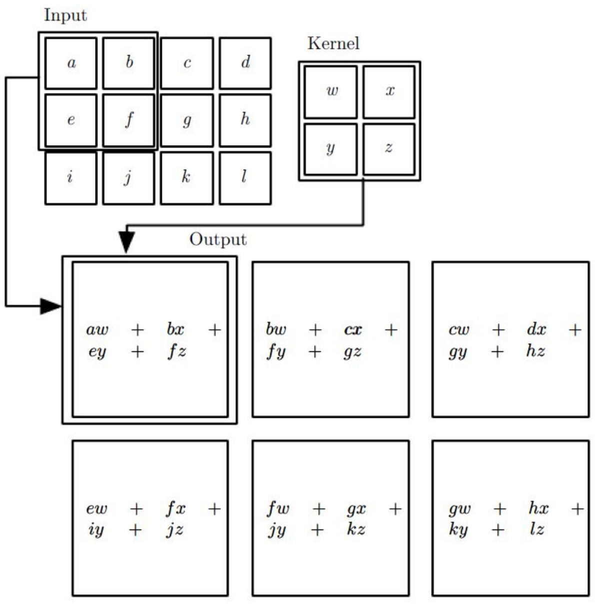


Figure 3: Example of -D convolution without kernel-flipping, as reported in [**Goodfellow-et-al-2016**]

Figure 3 shows an example of convolution (without kernel flipping) applied to a 2D tensor. In this case, the output is restricted to only positions where the kernel lies entirely within the image.

2.3 ARCHITECTURE

A CNN architecture is formed by a stack of distinct layers that transform the input volume into an output volume through a differentiable function. A few distinct types of layers are commonly used and they are described below.

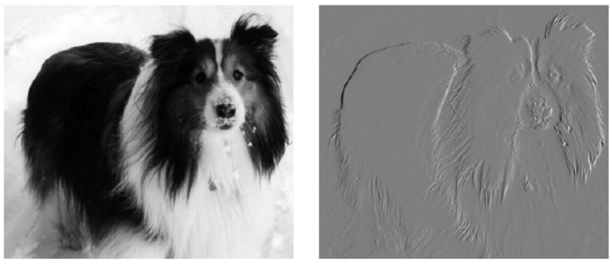


Figure 4: Application of a filter for edge detection (Source: [**Goodfellow-et-al-2016**])

2.3.1 *Convolutional layer*

The **convolutional layer** is the core building block of a CNN. The layer's parameters consist of a set of learnable filters (or kernels), which have a small receptive field, but extend through the full depth of the input volume. During the forward pass, each filter is convolved across the dimensions of the input volume, computing the dot product between the entries of the filter and the input and producing a feature map of that filter.

The amount of units by which the filter shifts is called **stride** and it controls how the filter convolves around the input volume.

As a result of this process, the network learns filters that activate when they detect some specific type of features in a given spatial position in the input.

It is evident that different filters will produce different feature maps for the same input. Figure 4 shows the result of the application of a filter for edge detection to a given image. The image on the right was formed by taking each pixel in the original image and subtracting the value of its neighboring pixel on the left. This shows the strength of all of the vertically oriented edges in the input image, which can be a useful operation for object detection.

Stacking the activation maps for all filters along the depth dimension forms the full output volume of the convolution layer. Every entry in the output volume can thus also be interpreted as an output of a neuron that looks at a small region in the input and shares parameters with neurons in the same activation map.

In fact, in order to recognize an image, we'll need more than one filter. For this reason, the output of a full convolutional layer will be a set of feature maps with a structure similar to the one in figure 5.

Sometimes it is convenient to have an output of the same size of the input. In this case it is necessary to add some additional pixels to the input image; this operation is called **padding**. For instance, a zero padding like the one in figure 6 pads the input volume with zeros all around the border. The size of this zero-padding is an optional

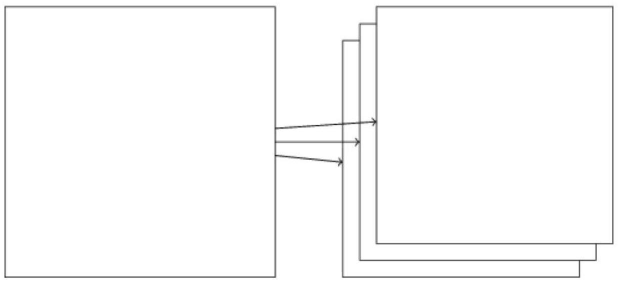


Figure 5: Application of three filters to a given input producing three different feature maps

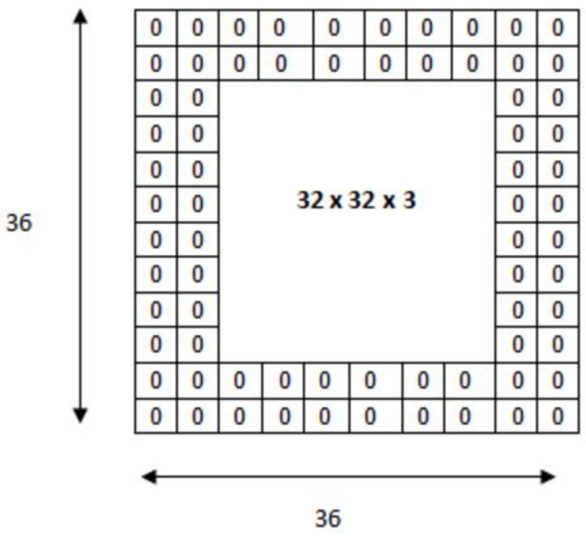


Figure 6: Zero padding of two applied to a 32x32x3 picture

hyperparameter. Zero padding, in particular, provides control of the output volume spatial size.

In general, the formula for calculating the output size for any given convolutional layer is:

$$O = \frac{W - K + 2P}{S} + 1$$

where O is the output height/length, W is the input height/length, K is the filter size, P is the padding, and S is the stride. If this number is not an integer, then the strides are set incorrectly and the neurons cannot be tiled to fit across the input volume in a symmetric way.

Setting the padding to

$$P = \frac{K - 1}{2}$$

when the stride is $S = 1$ ensures that the input volume and output volume will have the same size spatially.

2.3.2 *ReLU layer*

After a convolutional layer, it is convention to apply a non-linear layer (or activation layer) immediately afterward. The purpose of this layer is to introduce non-linearity to a system that basically has just been computing linear operations during the convolution operation (just element wise multiplications and summations). This is a layer of neurons that applies the non-saturating activation function:

$$f(x) = \max(0, x)$$

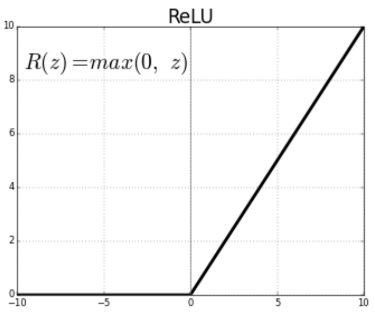


Figure 7: ReLU function

This increases the non-linear properties of the decision function and of the overall network without affecting the receptive fields of the convolution layer. The output of this layer known as **rectified feature map**.

In the past, non-linear functions like $tanh$ and sigmoid were used, but researchers found out that ReLU layers work far better because the network is able to train a lot faster (because of the computational efficiency) without significant reductions in the accuracy [**icml2010_NairH10**]. Using a ReLU function also helps to alleviate the vanishing gradient problem, which is the issue where the lower layers of the network train very slowly because the gradient decreases exponentially through the layers.

The ReLU layer applies the function $f(x) = max(0, x)$ to all of the values in the input volume. In basic terms, this layer just changes all

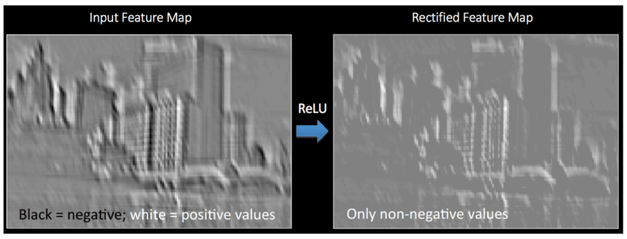


Figure 8: ReLU operation. Source http://mlss.tuebingen.mpg.de/2015/slides/fergus/Fergus_1.pdf

the negative activations to 0. Figure 8 shows the input of a ReLU layer and the resulting output.

2.3.3 *Pooling layer*

Another important concept of CNNs is **pooling**, which is a form of non-linear down-sampling. There are several non-linear functions to implement pooling among which **max pooling** is the most common. It partitions the input image into a set of non-overlapping regions and, for each of them, outputs the maximum value.

The intuition is that once a feature has been found, its exact location isn't as important as its rough location relative to other features. The function of the pooling layer is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to control overfitting, too. It is common to periodically insert a pooling layer in-between successive convolutional/ReLU layers in a CNN architecture.

The pooling operation also provides a form of translation invariance. This is very powerful since we can detect objects in an image no matter where and how they are located.

The pooling layer operates independently on every depth slice of the input and resizes it spatially. This operation is also known as **subsampling** or **downsampling**. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2. Every operation would in this case be taking a max over 4 numbers. Figure 9 shows an example of this operation.

In addition to max pooling, the pooling units can also perform other functions, such as average pooling and even L2-norm pooling. Average pooling was often used historically but has recently fallen out of favor compared to the max pooling operation, which has been found to work better in practice, as reported in [**Scherer:2010**].

Due to the aggressive reduction in the size of the representation (which is helpful only for smaller datasets to control overfitting),

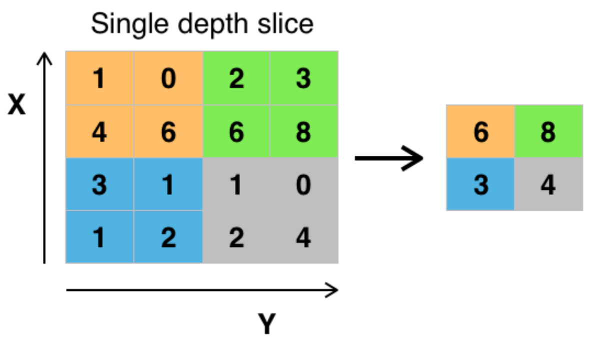


Figure 9: Max pooling with a 2x2 filter and stride $S = 2$

the current trend in the literature is towards using smaller filters [**DBLP:journals/corr/Graham14a**] or completely discarding the pooling layer [**DBLP:journals/corr/SpringenbergDBR14**].

Obviously, the pooling operation is applied separately to each feature map.

2.3.4 *Fully connected layer*

Finally, after several convolutional/ReLU and max pooling layers, the high-level reasoning in the neural network is done via **fully connected layers**. Neurons in a fully connected layer have full connections with all the activations in the previous layer, as in traditional MLPs. Their activations can hence be computed with a matrix multiplication followed by a bias offset.

The output from the convolutional and pooling layers represent high-level features of the input image. The purpose of the fully connected layer is usually to use these features for classifying the input into various classes based on the training dataset. For instance, the image classification task aims to put a given image into a precise category.

Apart from classification, adding a fully-connected layer is also a cheap way of learning combinations of these features. Most of the features from convolutional and pooling layers may be good for the classification task, but combinations of those features might be even better.

2.3.5 *Dropout layer*

Dropout layers have a very specific function in neural networks; they reduce overfitting. The problem of overfitting is very important in machine learning. It happens when after training, the weights of the network are so tuned to the training examples they are given that the network doesn't perform well when given new examples (i. e. it doesn't generalize).

The idea of dropout is simplistic in nature. This layer "drops out" a random set of activations in that layer by setting them to zero in the forward pass. At each training stage, individual nodes are either "dropped out" of the net with probability $1 - p$ or kept with probability p, where p is a user-defined parameter, so that a reduced network is left. The removed nodes are then reinserted into the network with their original weights.

This pruning, in a way, forces the network to be redundant; the network should be able to provide the right classification or output for a specific example even if some of the activations are dropped out.

This makes sure that the network isn't getting too "fitted" to the training data and, thus, helps alleviating the overfitting problem. An important note is that this layer is only used during training.

The method also significantly improves the speed of training. This makes model combination practical, even for deep neural networks. The technique seems to reduce the complex, tightly fitted interactions between nodes, leading them to learn more robust features which better generalize to new data. Dropout has been shown to improve the performance of neural networks on tasks in vision, speech recognition, document classification, and computational biology.

An interesting analysis of the use of dropout in order to manage overfitting can be found in **[Srivastava:2014:DSW:2627435.2670313]**

2.3.6 *Loss layer*

Finally, a loss layer specifies how the network training penalizes the deviation between the predicted and true labels. This is normally the last layer in the network. Various loss functions appropriate for different tasks may be used there.

Softmax loss is used for predicting a class of K different classes (i. e. the digit represented by an image from the MNIST dataset of handwritten digits, see chapter **??** for further details).

The function

$$f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

is called the softmax function: it takes a vector of arbitrary real-valued scores and squashes it to a vector of values between zero and one that sum to one.

Moreover, sigmoid cross-entropy loss is used for predicting K independent probability values in $[0, 1]$. Euclidean loss is used for regressing to real-valued labels in $[-\infty, \infty]$.

These layers are the basic building blocks of any CNN. They can be stacked in order to produce complex architectures. The number of layers depends on the addressed task and on the complexity of the input. A typical convolutional architecture is composed by one or more sets of convolutional, ReLU and pooling layers and ends with a fully connected, a dropout and a loss layer. Again, the loss function to use depends on the task to be solved.

Figure 10 shows an example of architechture for image recognition with two convolutional layers and two pooling layers.

Many different architectures have been proposed. Among them, LeNet **[lecun-98]** was one of the very first convolutional neural networks, for the recognition of handwritten digits, which helped propel the field of deep learning. In 2012, Alex Krizhevsky and others released AlexNet **[krizhevsky2012imagenet]** which was a deeper and

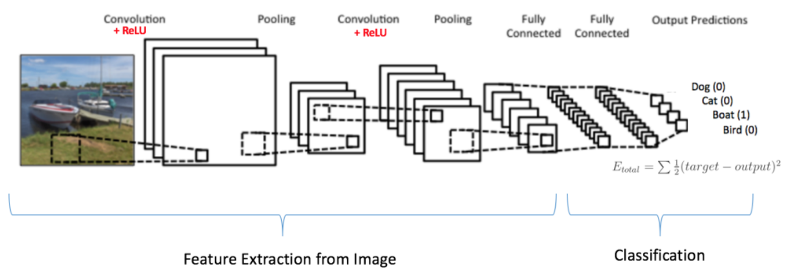


Figure 10: Convolutional neural network for the classification of an image

much wider version of the LeNet and won by a large margin the difficult ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. The ILSVRC 2014 winner was GoogleNet, a convolutional network from [**DBLP:journals/corr/SzegedyLJSRAEVR14**] from Google. Its main contribution was the development of an Inception Module that dramatically reduced the number of parameters in the network. In 2016 GoogleNet (now called Inception) is at its third release.

2.4 HYPERPARAMETERS

CNNs use more hyperparameters than a standard **mlp!** (**mlp!**). While the usual rules for learning rates and regularization constants still apply, the following should be kept in mind when optimising a convolutional network[].

NUMBER OF FILTERS Since feature map size decreases with depth, layers near the input layer will tend to have fewer filters while layers higher up can have more. To equalize computation at each layer, the product of the number of features and the number of pixel positions is typically picked to be roughly constant across the layers. Preserving the information about the input would require keeping the total number of activations (number of feature maps times number of pixel positions) to be non-decreasing from one layer to the next. The number of feature maps directly controls capacity and depends on the number of available examples and on the complexity of the task.

FILTER SHAPE AND SIZE Common filter shapes vary greatly in the literature, and are usually chosen based on the dataset. The challenge is, thus, to find the right level of granularity so as to create abstractions at the proper scale, given a particular dataset.

MAX POOLING SHAPE Typical values are 2x2. Very large input volumes may warrant 4x4 pooling in the first layers. However, choosing larger shapes will dramatically reduce the dimension of the signal, and may result in discarding too much information. Often, non-overlapping pooling windows perform best, as shown in [**Scherer:2010**].

AMOUNT OF EXAMPLES For many applications, only a small amount of training data is available. Convolutional neural networks usually require a large amount of training data in order to avoid overfitting. A common technique is to train the network on a larger data set from a related domain. Once the network parameters have converged an additional training step is performed using the in-domain data to fine-tune the network weights. This allows convolutional networks to be successfully applied to problems with small training sets.

2.5 APPLICATIONS

We conclude this chapter with some examples of application on CNNs in real-world domains. These kind of networks can be used with success to solve many different task, but, as said before, they work well with data with a grid-like topology.

2.5.1 *Image recognition*

Convolutional neural networks are often used in image recognition systems. They have achieved an error rate of 0.23 percent on the MNIST database, which as of February 2012 is the lowest achieved on the database [**6248110**]. When applied to facial recognition, they were able to contribute to a large decrease in error rate [**554195**].

The ImageNet Large Scale Visual Recognition Challenge is a benchmark in object classification and detection, with millions of images and hundreds of object classes. In the ILSVRC 2014, almost every highly ranked team used a CNN as basic framework. The winner GoogLeNet [**DBLP:journals/corr/SzegedyLJSRAEVR14**] (the foundation of Google DeepDream) increased the mean average precision of object detection to 0.439329, and reduced classification error to 0.06656, the best result to date. Its network applied more than 30 layers.

The performance of convolutional neural networks on the ImageNet tests is now close to that of humans. In 2015 a many-layered CNN demonstrated the ability to spot faces from a wide range of angles, including upside down, even when partially occluded with competitive performance [**DBLP:journals/corr/FarfadeSL15**].

2.5.2 *Video analysis*

Compared to image data domains, there is relatively little work on applying CNNs to video classification. Video is more complex than images since it has another (temporal) dimension. However, some extensions of convolutional neural networks into the video domain have been explored. One approach is to treat space and time as equivalent dimensions of the input and perform convolutions in both time and space [**6165309**].

2.5.3 *Natural language processing*

Convolutional neural networks have also seen use in the field of natural language processing. CNN models have subsequently been shown to be effective for various NLP problems and have achieved excellent results in semantic parsing, search query retrieval, sentence modeling, classification, prediction, and other traditional NLP tasks.

2.5.4 *Drug discovery*

Convolutional neural networks have been used in drug discovery. Predicting the interaction between molecules and biological proteins can be used to identify potential treatments that are more likely to be effective and safe.

In 2015, Atomwise introduced AtomNet, the first deep learning neural network for structure-based rational drug design. [**DBLP:journals/corr/WallachDH15**] Subsequently, AtomNet was used to predict novel candidate biomolecules for several disease targets, most notably treatments for the Ebola virus and multiple sclerosis.

2.5.5 *Playing Go*

Convolutional neural networks have been used in computer Go. In December 2014, Christopher Clark and Amos Storkey published a paper ([**DBLP:journals/corr/ClarkS14**]) showing a convolutional network trained by supervised learning from a database of human professional games could outperform GNU Go and win some games against Monte Carlo tree search Fuego 1.1 in a fraction of the time it took Fuego to play.

Shortly after it was announced that a large 12-layer convolutional neural network had correctly predicted the professional move in 55% of positions, equalling the accuracy of a 6 dan human player.

Finally, a couple of CNNs for choosing moves and evaluating positions were used by AlphaGo, Google Deepmind's program that was the first to beat a professional human player.

3 TENSORFLOW BASICS

TensorFlow (TF) is an open source software library for machine learning in various kinds of perceptual and language understanding tasks. It is currently used for both research and production by different teams in dozens of commercial Google products [**DBLP:journals/corr/AbadiABBCCCDDDG1** such as speech recognition, Gmail, Google Photos, and Google Search, many of which had previously used its predecessor DistBelief [**40565**].

TensorFlow was originally developed by the Google Brain team for Google's research and production purposes and later released under the Apache 2.0 open source license on November 9, 2015. Many teams nowadays at Google have migrated from DistBelief to TensorFlow for research and production uses.

This library of algorithms originated from Google's need to instruct neural networks, to learn and reason similarly to how humans do, so that new applications can be derived which are able to assume roles and functions previously reserved only for capable humans; the name TensorFlow itself derives from the operations which such neural networks perform on multidimensional data arrays. These multidimensional arrays are referred to as "tensors" but this concept is not identical to the mathematical concept of tensors. Its purpose is to train neural networks to detect and decipher patterns and correlations.

The aim of this chapter is to present the main data structures and operations provided by TF as well as to explain their basic usage. In particular the notions of computation graph, constant, variable and feed will be explained. Then, TensorBoard, a powerful tool for visualizing learning, will be presented. Finally, the last section includes a very simple example model built using TensorFlow.

The material provided in this chapter is taken form [**tensorflow**].

3.1 OVERVIEW

TensorFlow is Google Brain's second generation machine learning system. While the reference implementation runs on single devices, TF can run on multiple CPUs and GPUs (with optional CUDA extensions for general-purpose computing on graphics processing units). It runs on 64-bit Linux or Mac OS X desktop or server systems, as well as on mobile computing platforms, including Android and Apple's iOS.

It provides a Python API, as well as a less documented C++ API. In particular, the TensorFlow Python API supports Python 2.7 and Python 3.3+. It can be easily installed as a common Python package through `pip` or in Anaconda with a `conda` installation.

TensorFlow computations are expressed as stateful dataflow **graphs**. Nodes in the graph are called *ops* (i. e. operations). An op takes zero or more `Tensor` objects, performs some computation, and produces zero or more `Tensors`. In TensorFlow terminology, a Tensor is simply a typed multi-dimensional array. For instance, a batch of images can be represented as a 4D tensor `Tensor` of floating point numbers with dimensions `[batch, height, width, channels]`.

In other words, a TF graph is a description of some computations. In order to actually compute anything, a graph must be launched in a `Session`. A `Session` object places the graph ops onto `Devices`, i. e. CPUs or GPUs and provides methods to execute them. In Python, these methods return tensors produced by ops as numpy `ndarray` objects.

3.2 THE COMPUTATION GRAPH

TensorFlow programs are usually structured into a construction phase, that assembles a graph, and an execution phase that uses a session to execute ops in the graph.

For example, it is common to create a graph to represent a neural network in the construction phase, and then repeatedly execute a set of training ops in the graph in the execution phase.

3.2.1 *Building the graph*

To build a graph it's useful start with ops that do not need any input (source ops), such as `Constant`, and pass their output to other ops that do computation. The ops constructors in the Python library return objects that stand for the output of the constructed ops. You can pass these to other ops constructors to use as inputs.

The TensorFlow Python library has a default graph to which ops constructors add nodes that is sufficient for many applications. However, it is possible to explicitly manage multiple graphs.

The following code builds a graph composed by three nodes. The two `constant()` operations produce two matrices (a 1x2 and a 2x1 matrix), while the `matmul()` operation takes as input the two previously created matrices and outputs the result of the matrix multiplication.

```
import tensorflow as tf

matrix1 = tf.constant([[3., 3.]])
matrix2 = tf.constant([[2.],[2.]])

product = tf.matmul(matrix1, matrix2)
```

In order to actually multiply the matrices, and get the result of the multiplication, the graph must be launched in a `Session`.

3.2.2 *Launching the graph*

Launching follows construction. To launch a graph, a `Session` object. Without arguments the session constructor launches the default graph.

The following code creates a session, launches the previously created graph and prints he result of the matrix multiplication.

```
# Launch the default graph
sess = tf.Session()

result = sess.run(product)
print(result)
# ==> [[ 12.]]

# Close the Session when we're done
sess.close()
```

To run the matmul op we call the session `run()` method, passing `product` which represents the output of the `matmul` op. This indicates to the call that we want to get the output of the `matmul` op back. All inputs needed by the op are run automatically by the session, typically in parallel. The call `run(product)` thus causes the execution of three ops in the graph: the two `constants` and `matmul`. The output of the matrix multiplication is returned in `result`.

Sessions should always be closed to release resources. They can be also surrounded with a `with` block. In this way, the `Session` closes automatically at the end of the block. The code then becomes:

```
with tf.Session() as sess:
    result = sess.run([product])
    print(result)
```

3.3 VARIABLES

`Variable` objects maintain state across executions of the graph. When a model is trained, `Variables` are used to hold and update parameters. They are in-memory buffers containing tensors. They must be explicitly initialized and can be saved to disk during and after training. Saved values can be later restored to exercise or analyze the model.

3.3.1 *Creation*

When a `Variable` a `Tensor` is passed to the `Variable()` constructor as initial value. TF provides a collection of ops that produce tensors that can be used for initialization from constants or random values. All these ops require to specify the shape of the tensors. That shape automatically becomes the shape of the variable. Variables generally have a fixed shape, but TensorFlow provides advanced mechanisms to reshape them.

The following code creates two variables, one for weights and one for biases.

```
weights = tf.Variable(tf.random_normal([784, 200], stddev=0.35),
                      name="weights")
biases = tf.Variable(tf.zeros([200]), name="biases")
```

Calling `tf.Variable()` adds several ops to the graph:

- A `Variable` op the holds the variable value;
- An initializer op that sets the variable to its initial value;
- The ops for the initial value (e. g. the `zeros` op for `biases`).

The output is an instance of the Python class `tf.Variable`.

3.3.2 *Initialization*

Variable initializers must be run explicitly before other ops in the model can be run. The easiest way to do that is to add an op that runs all the variable initializers, and run that op before using the model.

The op `tf.global_variables_initializer()` can be used to add an op to run variable initializers. The op must be run after the model has been fully constructed.

The following code initializes the two variables created before:

```
init_op = tf.global_variables_initializer()

with tf.Session() as sess:
    # Run the init operation.
    sess.run(init_op)
    ...
```

3.3.3 *Saving and restoring*

The easiest way to save and restore a model is to use a `tf.train.Saver` object. The constructor adds `save` and `restore` ops to the graph for all, or a specified list, of the variables in the graph. The saver object provides methods to run these ops, specifying paths for the checkpoint files to write to or read from.

Checkpoints are binary files that, roughly, contain a map from variable names to tensor values.

The same `tf.train.Saver()` object can be used to manage all the variables in the model. Restored variables do not have to be initialized beforehand.

The following two pieces of code save and restore two variables:

```
v1 = tf.Variable(..., name="v1")
v2 = tf.Variable(..., name="v2")

init_op = tf.global_variables_initializer()

saver = tf.train.Saver()

with tf.Session() as sess:
    sess.run(init_op)
    ...
    save_path = saver.save(sess, "/tmp/model.ckpt")
    print("Model saved in file: %s" % save_path)
```

```
v1 = tf.Variable(..., name="v1")
v2 = tf.Variable(..., name="v2")

saver = tf.train.Saver()

with tf.Session() as sess:
    saver.restore(sess, "/tmp/model.ckpt")
    print("Model restored.")
    ...
```

If no argument is passed to `tf.train.Saver()`, the saver handles all variables in the graph. Each one of them is saved under the name that was passed when the variable was created.

It is sometimes useful to explicitly specify names for variables in the checkpoint files. For example, there might be a trained model with a variable named "weights" whose value have to be to restored in a new variable named "params".

It is also sometimes useful to only save or restore a subset of the variables used by a model. For example, a neural net with 5 layers could have been trained. In order to train a new model with 6 layers, the parameters from the 5 layers of the previously trained model can be restored into the first 5 layers of the new model.

The names and variables to save can be easily specified by passing to the `tf.train.Saver()` constructor a Python dictionary: keys are the names to use, values are the variables to manage.

3.4 FEEDS

The examples above introduced tensors into the computation graph by storing them in constants and variables. TF also provides a feed mechanism for patching a tensor directly into any operation in the graph.

A feed temporarily replaces the output of an operation with a tensor value. Deed data are supplied as an argument to a `run()` call. The feed is only used for the run call to which it is passed.

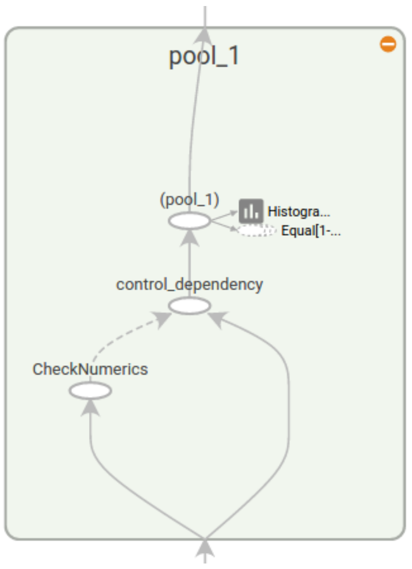


Figure 11: Graph visualization using TensorBoard

The most common use case involves designating specific operations to be "feed" operations by using `tf.placeholder()` to create them:

```
input1 = tf.placeholder(tf.float32)
input2 = tf.placeholder(tf.float32)
output = tf.mul(input1, input2)

with tf.Session() as sess:
    sess.run([output], feed_dict={input1:[7.], input2:[2.]})
```

A `placeholder()` operation generates an error if a feed for it isn't supplied.

3.5 TENSORBOARD

The computations performed with TensorFlow - like training a massive deep neural network - can be complex and confusing. To make them easier to understand, debug, and optimize TensorFlow programs, the developers included a suite of visualization tools called TensorBoard. It can be used to visualize a TensorFlow graph, plot quantitative metrics about the execution, and show additional data like images that pass through the graph.

TensorBoard operates by reading TensorFlow events files, which contain summary data that can be generated when running a TensorFlow graph.

Figure 11 shows an example of graph visualization using this tool. Every node can be expanded or minimized in order to analyze the graph at different granularities.

Further information about the usage of TensorBoard are are available in the dedicated page on the TensorFlow website.

3.6 A VERY SIMPLE MODEL

We conclude this chapter showing a very simple linear model built with TF. The program makes up some data in two dimensions, and then fits a line to them.

```
import tensorflow as tf
import numpy as np

# Create 100 phony x, y data points in NumPy, y = x * 0.1 + 0.3
x_data = np.random.rand(100).astype(np.float32)
y_data = x_data * 0.1 + 0.3

# Try to find values for W and b so that y_data = W * x_data + b
# (We know that W should be 0.1 and b 0.3, but TensorFlow will
# figure that out for us.)
W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = W * x_data + b

# Minimize the mean squared errors.
loss = tf.reduce_mean(tf.square(y - y_data))
optimizer = tf.train.GradientDescentOptimizer(0.5)
train = optimizer.minimize(loss)

# Before starting, initialize the variables.
init = tf.global_variables_initializer()

# Launch the graph.
sess = tf.Session()
sess.run(init)

# Fit the line.
for step in range(201):
    sess.run(train)
    if step % 20 == 0:
    print(step, sess.run(W), sess.run(b))

# Learns best fit is W: [0.1], b: [0.3]
```

The first part of the code builds the data flow graph. TensorFlow does not actually run any computation until the session is created and the run function is called, in the second part of the code.

4 A BASIC TENSORFLOW MODEL

In this chapter we will discover a basic TensorFlow model developed in order to classify images from the MNIST dataset. Our scope is to illustrate the structure of a CNN and how that basic model can be readjust as needed, gathering data about training accuracy during the epochs and test accuracy on known and unknown images from the dataset.

After an introduction to the MNIST dataset, we will study in deep the structure of the Convolutional Neural Network used in our experiments, trying to understand which parameters are relevant to the improvement of results, and how the structure can influence most of all on the accuracy of the network during training.

We identify two levels where we can intervene to modify the neural network. A architectural one, where we add, move and remove convolutional or pooling levels, and a parameter level, where we modify number of features, patch dimension, etc. The first one is more important, the second one is a more precise intervention.

4.1 A FOCUS ON MNIST DATASET

Before beginning, let us focus on MNIST database, which is widely used in machine learning for training and testing. The acronym MNIST stands for Mixed National Institute of Standards and Technology because this database contains a large number of handwritten digits, it is a re-mixing of a previous dataset (NIST dataset) and collects 60000 images for training and 10000 for testing. The images were taken half from the American Census Bureau employees and half from American high school students and includes labels telling which digit it is.

Figure 12 shows us four examples of handwritten digit images from MNIST dataset, in order we have 5, 0, 4, 1.

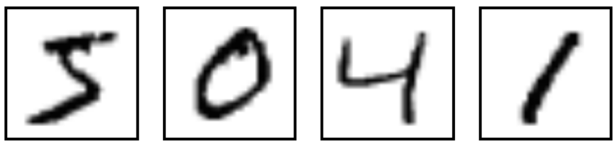


Figure 12: Digit images from MNIST dataset

4.1.1 *Usage in the machine learning*

As mentioned earlier, MNIST dataset is widely used for training and testing neural networks, because it is a reference dataset in machine learning. During the years, lots of solution in neural networks research trained and tested on this dataset. Sometimes exciting results led to stale that neural networks achieved "near-human performance". But first attempts obtained mediocre results: a linear classifier had a error rate from 12.0% to 7.6%, which is really not satisfying for such a simple task.

Other solutions led to better results. Non-linear classifier drastically reduced error rate to 3.3%, but more appreciable results were obtained with 2-layers $(784 - 800 - 10)$ neural networks, error rate: 1.6%, 0.7% with elastic distortion. Boosted stumps achieved an error rate of 0.87%, support vector machines 0.56%, K-nearest neighbours 0.52% and 6-layer deep neural network 0.35%.

The best result have been achieved with CNNs: different kinds of CNN reached the lowest error rate, from 0.31% to 0.21%. Although the problem is quite simple, it clear that this kind of neural network is more powerful than others.

4.1.2 *Technical informations about images*

The digits have been size-normalized and centered in a fixed-size image of 28x28 pixels. The dataset includes labels for each image telling us which digit it is.

Originally NIST dataset contained black and white, bilevel images. As a result of anti-aliasing technique used in normalization algorithms, images now contain grey levels. Digits were centered by computing the center of mass of the pixels and properly translating them.

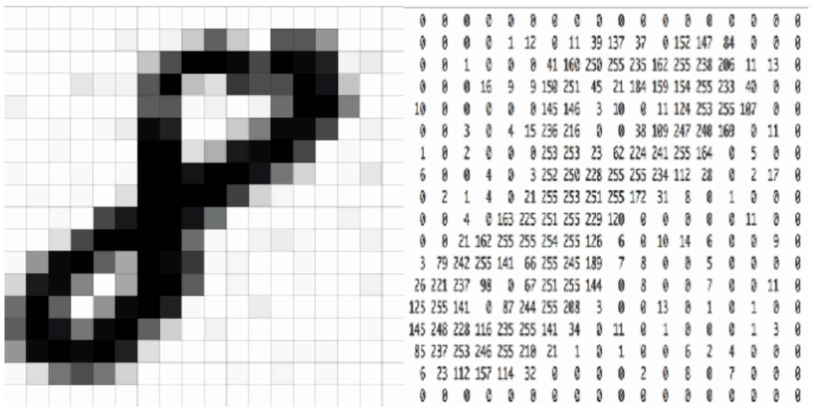
Images are not in a standard image format, the data is stored in a very simple format designed for storing vectors and multidimensional matices. Clearly labels values are 0 to 9, while pixel values are 0 to 255, 0 is white and means background, 255 is black and means foreground; pixels are organized row-wise. In TensorFlow we normalize 0 to 255 values in 0 to 1 range.

Figure 13 shows uf an example of images storing format: each image can be translated in numeric matrix, pixel by pixel, encoding pixel colours (white, black and grey levels) in 0 to 255 range.

More detailed informations about data format are not useful to read and compute the files in the dataset.

4.2 THE HYPERPARAMETERS LEVEL

Editing hyperparameters of the network is very delicate, it is a precision intervention which hopefully allows uf to improve training and



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	12	0	11	39	137	37	0	152	147	84	0	0	0
0	0	1	0	0	0	41	160	250	255	235	162	255	238	206	11	13	0
0	0	0	16	9	9	150	251	45	21	184	159	154	255	233	40	0	0
10	0	0	0	0	0	145	146	3	10	0	11	124	253	255	107	0	0
0	0	3	0	4	15	236	216	0	0	38	109	247	240	169	0	11	0
1	0	2	0	0	0	253	253	23	62	224	241	255	164	0	5	0	0
6	0	0	4	0	3	252	250	228	255	255	234	112	28	0	2	17	0
0	2	1	4	0	21	255	253	251	255	172	31	8	0	1	0	0	0
0	0	4	0	163	225	251	255	229	120	0	0	0	0	0	11	0	0
0	0	21	162	255	255	254	255	126	6	0	10	14	6	0	0	9	0
3	79	242	255	141	66	255	245	189	7	8	0	0	5	0	0	0	0
26	221	237	98	0	67	251	255	144	0	8	0	0	7	0	0	11	0
125	255	141	0	87	244	255	208	3	0	0	13	0	1	0	1	0	0
145	248	228	116	235	255	141	34	0	11	0	1	0	0	0	1	3	0
85	237	253	246	255	210	21	1	0	1	0	0	6	2	4	0	0	0
6	23	112	157	114	32	0	0	0	0	2	0	8	0	7	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 13: Example of translation from image to numeric matrix

performance, or reduce in the worst case. We will consider the following parameters: number of features to identify in convolutional layers, patch size, stride size, number of input and output channels and padding of images.

NUMBER OF FEATURES That is one of the most important parameters to set, because here is specified how many features the network can learn. There is not a formula which allows us to find the perfect number of features based on the complexity of a task, but after several attempts with different values for this parameter we find a good solution for MNIST images. It can be concluded that a simple task, e.g. classify 28x28 greyscale images of digits, requires a small amount of features: 8 in first convolutional layer, 16 in the second one. When we will analyze the result with different values of this parameter, we will find that extraordinary performance can be obtained with 2 and 4 features. Most difficult task with bigger and higher quality images, probably will require tens or hundreds features. An opposite problem is a too high value, the result is that some features can be equal or very similar, so it could be not really useful and performing.

PATCH SIZE Patch size value allows us to adjust the size of the area where the network find features. Contrary to expectations, this parameter will not change from a convolutional and pooling layer to the next, because while it is a fixed value, the image dimension is reduced, so the patch cover a greater area and allows the recognition of more complex features in more space. In our example with 28x28 images a good patch size value is 5x5. Obviously a bigger image require appropriate sized patch.

STRIDE SIZE According to the vanilla version proposed in the example, our stride value is 1. This value is combined with the padding in convolution ad pooling in order to obtain an input of the same size of the input.

INPUT AND OUTPUT CHANNELS These parameters can vary from a convolutional layer to another. In the first layer the input channel is linked to the colours of the input images, so in our example we have the value 1; output channel value depends on the number of features recognized in the layer. In later layers the input channel must correspond to the output channel of previous layer, and output channel to the features recognized in the layer.

PADDING DIMENSION As described previously with the formula, we determined the zero padding dimension. It is very important because with the stride size value permit to keep stable dimensions of input and output layer after layer.

4.3 MAIN STRUCTURE: THE ARCHITETURAL LEVEL

We introduced in Chapter 2 the different types of layers useful to build a CNN], now we can analyze how they are combined in our TensorFlow model. This architecture was not modified, so it is exactly the one suggested in the TensorFlow example.

After the MNIST images are loaded, the TensorFlow session started and the variables initialized, we start building the multilayer CNN.

4.3.1 *Weights initialization*

First of all we need to define a matrix of weights, its dimensions depend on images dimensions, so if images are 28x28 pixel and we have 10 categories, weights matrix have to be $[28 * 28, 10]$, so $[784, 10]$ dimensioned. At first weights are random values from a truncated normal distribution, the following code lines shows their initialization.

```
def weight_variable(shape):
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial)
```

4.3.2 *First convolutional layer*

The following code lines introduce the first block: a convlutional layer followed by the ReLU layer and the pooling layer. In the first two lines there is the initialization of weight and bias; weights are random values from a truncated normal distribution, bias is a constant tensor. Then on the third line there are convolution and ReLU operation, followed on the fourth line by the max pooling.

That kind of structure in the block we described is not the only solution, the different layers can be combined in different ways, e.g. we can add in this block a second convolutional layer instead of one before ReLU and max pooling.

```
W_conv1 = weight_variable([5, 5, 1, features1])
b_conv1 = bias_variable([features1])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)
```

4.3.3 *Second convolutional layer*

Similarly to the previous subsection we can add a second block which has exactly the same structure of the first one: a convolutional layer with ReLU and pooling operations. The following code is equal to the previous, it does the same operations.

```
W_conv2 = weight_variable([5, 5, features1, features2])
b_conv2 = bias_variable([features2])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
```

The only difference between the two blocks we newly added is in the hyperparameters, in particular the number of features the network recognises analyzing the images in input.

Any changes are possible: not only a single convolutional level con be added, but a whole block like the two we described right now. A more structured CNN can be built with more complex task.

4.3.4 *Densely connected layer*

The densely connected layer is the core of classifying process in our CNN, it is implemented as matrix multiplication added to a bias offset. With that layer it is also possible to learn combinations of features.

```
W_fc1 = weight_variable([7 * 7 * features2, 1024])
b_fc1 = bias_variable([1024])
h_pool2_flat = tf.reshape(h_pool2, [-1, 7 * 7 * features2])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

4.3.5 *Dropout leyer*

In order to reduce overfitting we introduce a dropout layer, which is very important to perform better wih new examples. The implementation of that operations is really simple: a random set of activations is setted to zero, so the pruned nodes are reinitialized and reinserted into the network. Dropout layer is used only during training, not during testing.

```
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

4.3.6 *Loss (readout) layer*

The loss layer is the last one but also the core of regression model in our CNN, here we find the implementation of regression model with the loss function: a multiplication of vectorized input images by the weight matrix, added to bias.

Our loss function is the cross entropy between the target and the softmax activation function applied to the model's prediction.

```
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])
y_conv = tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
```

4.3.7 *Training, testing and evaluating the model*

TRAINING We build our CNN layer by layer, convolution by convolution and finally we defined the loss function. Now it is time to train that model and test its capabilities.

Train with TensorFlow allows to use automatic differentiation to find the gradient of the loss with respect to each of the variables. In the wide range of built-in optimization algorithms offered by TensorFlow we choose the Adam optimizer. The optimizer adds new operations, which include ones to compute gradients, compute parameter update steps, and applied update steps to the parameters.

```
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y_conv)
    , reduction_indices=[1]))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_,
     1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32)
    )
```

Training the model can be therefore be accomplished repeatedly running the train_step for a number of epochs.

TESTING AND EVALUATING After a long training session, we want to evaluate performances of our network. It is possible with the following code line, where a percentage value of the accuracy is calculated.

```
test_accuracy = accuracy.eval(feed_dict={x: mnist.test.images, y_
    : mnist.test.labels, keep_prob: 1.0})
```

Similarly we can evaluate the learning trend during training epochs. It is one of the most important measurement we made during lot of tests with different parametrized CNN.

4.4 MODIFICATIONS OF THE ARCHITECTURE

Previously we talk about architectural interventions, classifying them as more structural modifications. We reinvented the wheel during the first tests, where we tried to change something in the layers structure adding convolutional levels. Disastrous results led us to the conclusions that in case of quite simple task, like digits classification, ad excessive amount of layers ends in unsatisfactory results.

Just like the number of features should be assessed on the basis of task complexity, also the structure must be proportionate to it much more than hyperparameters.

Certainly task like facial recognition or high-defined images classification requires more complex structures, with more convolution levels and much more features.

5 ANALYSIS OF COLLECTED DATA

In order to reach satisfying conclusions, we will analyze carefully the collected data every time the structure of the network has been changed, trying to understand how it performs, what kind of adjustment we can do to improve the results, and comparing the different structured networks performances on the same classifying task with MNIST images.

5.1 GATHERING DATA

To begin with, we will describe the gathering data process focusing on what, when and why we recorded data during network executions. It is important to understand the core of our analysis and the conclusions we will reach at the end of this project report.

5.1.1 *What, when, why*

Now will follow an accurate description of the data we gathered, focusing also on when they were recorded and explaining reasons which moved to. Choosing what and when to record data is very important, because that data project us into the TensorFlow model and permit to understand behaviour, performance and results.

Gathered data can be summarized in accuracy during training phase and test accuracy after training, both strongly connected to the number of features we set to the convolutional network. Now we explain them in detail:

TRAINING ACCURACY An important measurement is the accuracy during training phase because it allows us to focus on learning trend and discover the behaviour of networks with different number of features. As we will see later with the help of some graphics, we can deduce lots of useful information. These values are between 0 and 1 and were recorded every 10 epochs, guaranteeing dense and rich sampling for a good analysis.

FINAL TEST ACCURACY An important value as the previous is the measurement of accuracy in testing phase, after training. With these values we can analyze performances of networks with different number of features. Focusing on these results we will conclude possible guidelines useful to set a number of features commensurate to our necessities.

EXECUTION TIMES Measurements of execution times were not made with precision or recorded, but we took notes about faster or slower nets to understand how structural changes or different number of features can affect training duration.

These informations, in relation to analysis on performances in training and testing phases, will help us to draw conclusions about diverse structures and parameterized CNN.

5.1.2 *Storage format*

Data gathered during executions were first saved in arrays and then written to a CSV file. For each different network, a CSV file was created with recorded data of training accuracy during all executions. So, in each column of the file we find the training accuracy of an execution sampled every 10 epochs.

Regarding measurements of accuracy after training, a unique CSV file was created with all test accuracy for each different network and for each execution. Data are organized by rows, on a row we find: the number of features of the network, the values of test accuracy, a final average value of all test accuracy values for the network. These average values will be the measure to evaluate and compare networks with different numbers of features.

5.2 EXECUTIONS AND PARAMETERS CHOICE

This short section will introduce and explain some contextual informations about how and why we set number of executions and parameters values in diverse convolutional networks. These choices are preserved on all executions and are fundamental in our incoming analysis.

5.2.1 *Number of executions*

Gathering data of a single execution and with too large sampling did not give us satisfying and considerable graphics because data were not sufficient and lines too variable. To overcome these problems we executed multiple times training and test of the same TensorFlow model, a good compromise between availability of time, computing power and sufficient amount of data is 30 executions, but previous attempts were limited to 10 executions.

A higher number of executions produce a bigger amount of data, more complete and statistically more significant. Another advantage is that the lines in graphics are more stable and plain, afterwards we will see the graphics and appreciate the results.

Table 1: Selected numbers of features in diverse TensorFlow models

	Number of features	
	1st conv. layer	2nd conv. layer
Model 2_4	2	4
Model 4_8	4	8
Model 8_16	8	16
Model 16_32	16	32

5.2.2 *Number of epochs*

The first version of this TensorFlow model provided 20000 epochs, after first executions we concluded that such a big number was exaggerate, so we reduced training epochs to 15000. That amount is amply sufficient to observe a good convergence and stabilization of training trend, and allows us after lots of executions to save time.

Another advantage of fewer epochs is a representation of data in the graphics that is less compressed, than it turns out to be more clear, informative and readable.

5.2.3 *Number of features*

We executed four different TensorFlow models, the only difference is the number of features we choose for each one. Probably it is the most important and influent parameter for performance during training and test phases, and the following analysis will be based on this parameter.

We choose an increasing number of features and in the convolutional layer we followed the rule of doubling, selected values are summarized in Table 1.

In next section we will also quibble on a too high number of features, trying to understand if it can be evil or not, if the network find features which are equal each other and how can realize that we have chosen satisfying values.

5.3 ANALYSIS OF TRAINING ACCURACY

After a long wait, necessary for introduce useful informations, finally we arrive hopefully to the most interesting section of this project report. Now we are prepared to introduce an accurate analysis of the diverse TensorFlow models described.

We will focus mainly on:

- Training speed
- Training stability

- Training performance
- Training duration
- Influences of parameters on the network

For each of these perspectives we will conduct a thorough analysis with the help of graphics.

5.3.1 *Graphics and notations*

Always considering all graphics:

y-axis On this axis we find the training accuracies in the range from 0.85 to 1. We cut out the part of the graphic below the value 0.85 because it is not really useful and mainly because it allows us to focus on the most interesting area.

x-axis On this axis we find the time values: the epochs. We will consider different time windows according to the needs, but usually we find all the epochs of training phase.

Legend In every graphics of this section will refer to the four TensorFlow model with a notation based on the number of features in the two convolutional layer: number of features in first layer + _ + number of features in second layer.

5.3.2 *Training speed*

The first perspective we explore is the speed of training phase, or rather how quickly the convolutional network learns from the example images. It is interesting to inspect this, because as we shall see in the graphics, in front of a greater number of features we find a more training speed. In other words it means: more features, more rapid learning.

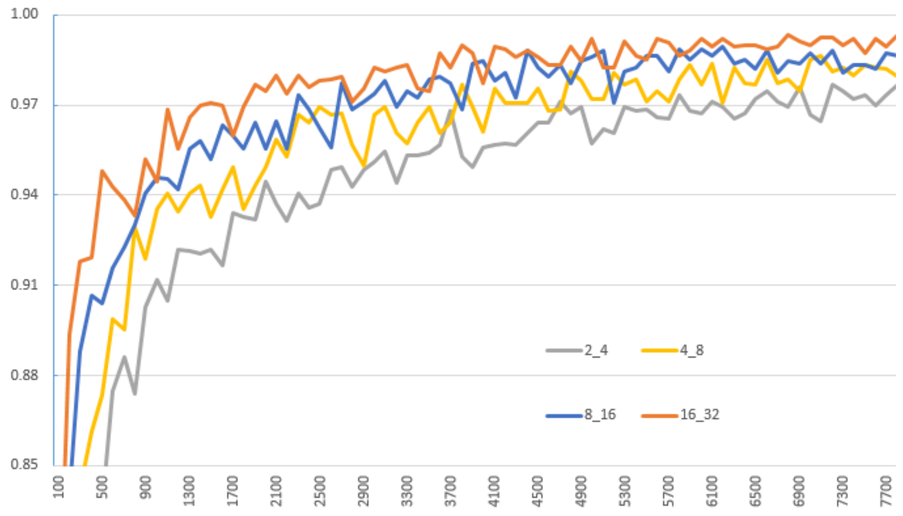
Figure 14 shows the first part of the training phase, about the first half. It allows us to focus on the most interesting part of the graphic for the analysis we are going to do.

Focusing on Figure 14 we can clearly observe that more features imply a more rapid learning. We can notice this in two ways, two points of view.

Considering the same training value, e.g. 0.85, we observe the following facts. that the model with fewer features (Model 2_4) reaches the referring value in more than 500 epochs.

MODEL 2_4 The TensorFlow model with fewer features reaches the referring value of training accuracy in more than 500 epochs.

Figure 14: Comparison of training speeds of the four TensorFlow models with different number of features



model 4_8 Increasing the number of features we obtain the same training accuracy value of 0.85 in about 350 epochs.

model 8_16 The model with 8 and 16 features on first and second convolutional layer reach that training accuracy value in about 200/250 epochs.

model 16_32 The last model with the higher number of features is the fastest, reaching the referring training accuracy value in about 150 epochs.

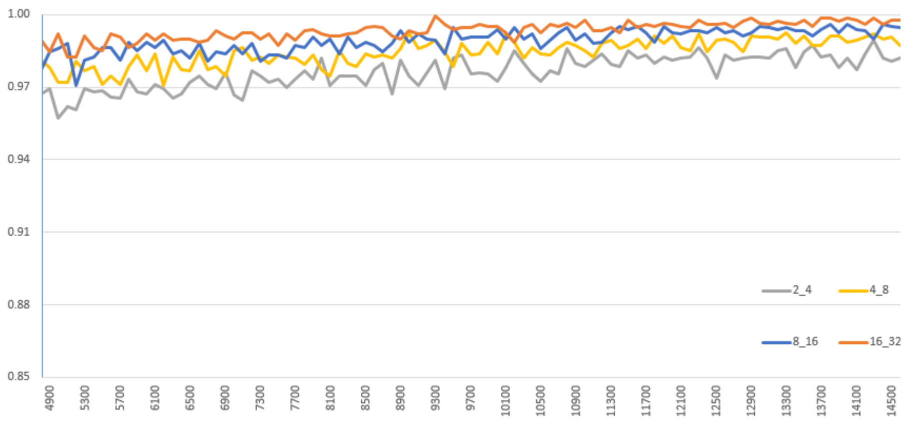
Even without focusing on precise number of epochs, we can notice at a glance the slower and faster model. The same observations we just made, can be done considering a precise epoch during the first part of the training phase where the learning curve mainly grows.

5.3.3 *Training stability*

A second perspective in our analysis is the stability of training, particularly towards the end part of training phase. Figure 15 shows the training curve in its adolescence to its adulthood, about from epoch 5000 to 15000. In this range we can observe a progressive stabilizations of learning, and similarly to training speed, here the number of features we set influence the stability.

Without the need for explain each curve, we can focus on ups and downs: with fewer features we find more pronunced peaks, while increasing the number of features we obtain more stable learning curve.

Figure 15: Comparison of training stability of the four TensorFlow models with different number of features



An important conclusion of these observations is that we can have better and more stable learning with an appropriate and sufficient number of features. In case of too low value training will be instable and require more epochs. But be careful: remember that good and stable training accuracy does not imply directly a good test accuracy, so keep your head even if you do not obtain the most stable learning curve, it could be worse (e.g. your neural network can do a really bad generalization).

5.3.4 *Training performance*

After an interesting analysis of how quick and stable can be the training phase, finally we can focus on the most important aspect: performances. It is clear, now, that is strictly connected to the previous perspectives, and we can naturally imagine the following fact: more feature means better results. After all we have already said, it is a logical deduction, and effectively it is true.

Figure 16 shows a complete graphics of the diverse learning curve of training phases in our four different TensorFlow models. Analyzing this chart we can see immediately that the features number greatly influences performances, the figure speaks for itself.

Together with the two previous perspective, we can also observe that improvements are not proportional to the number of features. Rather we have greater improvement between the models 2_4 and 4_8, instead of between 8_16 and 16_32. In the first case we can observe visible differences in training speed, stability and performance, which are less marked in the second case. We can observe much better these aspects in Figure 17 where the focus is on models 2_4 and 4_8 and Figure 18 where there are only models 8_16 and 16_32.

The observation we just made is really important because it allows us to understand with experiments the extent to which it is convenient, or we consider convenient, increase the number of features that the network should recognize. Until we consider useful a certain improvement of these perspectives, we can set higher values.

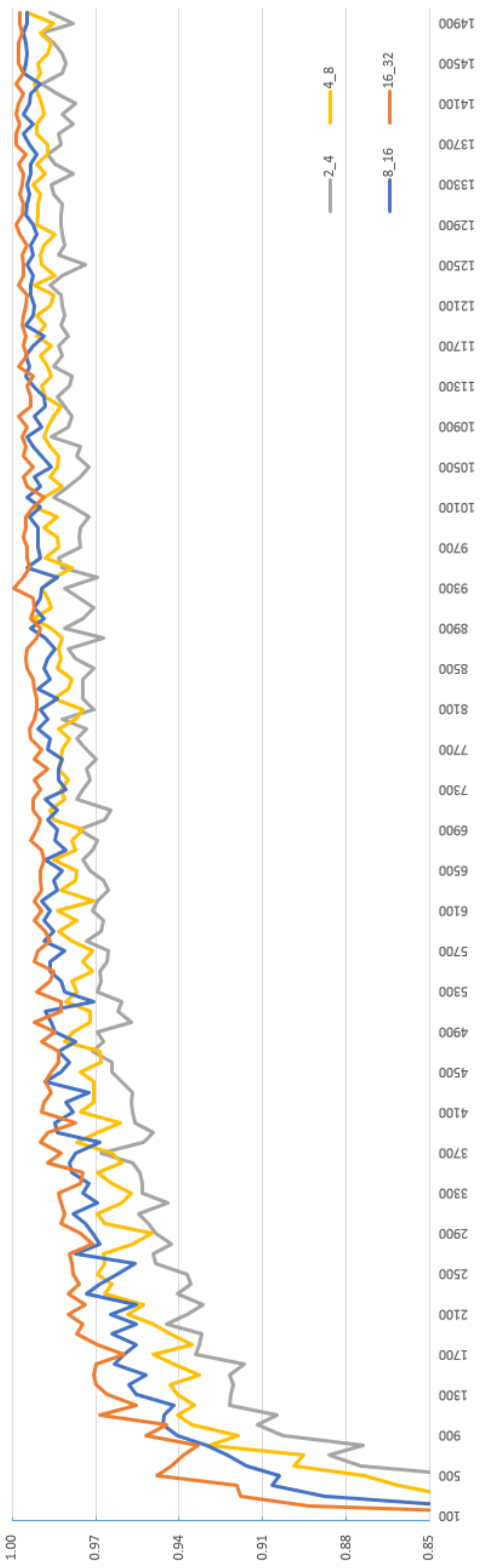


Figure 16: Comparison of training accuracy curves of the four TensorFlow models with different number of features

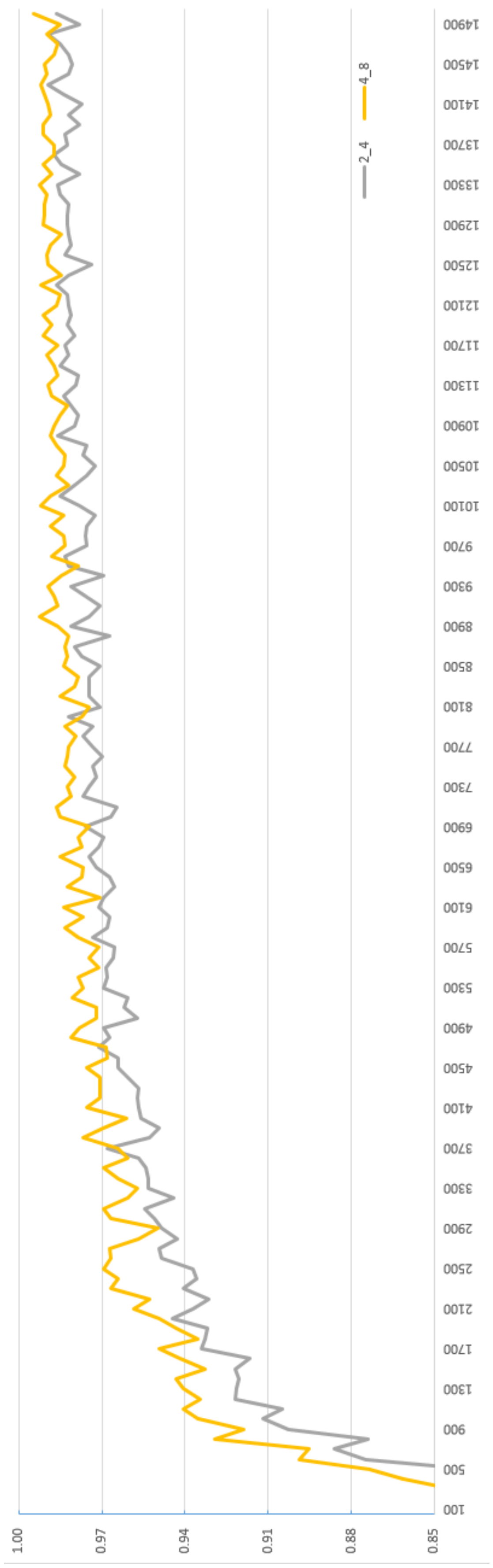


Figure 17: Comparison of training accuracy curves of two TensorFlow models with lower number of features

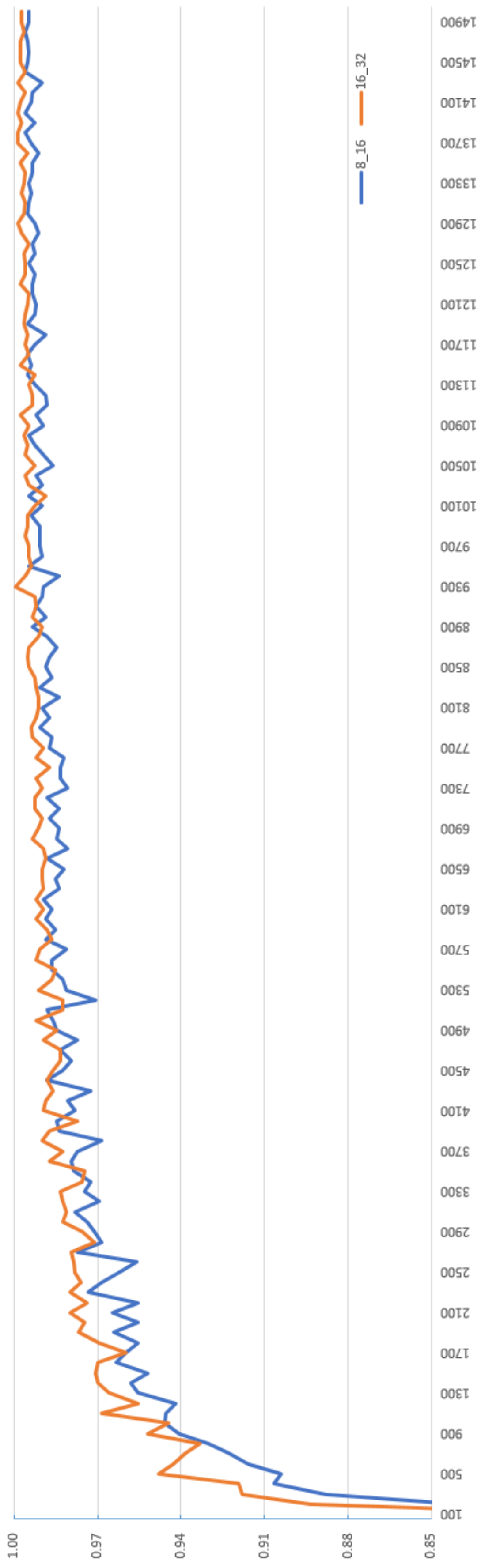


Figure 18: Comparison of training accuracy curves of two TensorFlow models with higher number of features

Table 2: Approximate times measurements of diverse TensorFlow models executions

	Number of features
Model 2_4	7 minutes
Model 4_8	7 minutes
Model 8_16	7 minutes
Model 16_32	7 minutes

5.3.5 *Training duration*

A non-trivial aspect that common people with limited computing power always consider is: how much time does it take? It is a legitimate question, and we will promptly broach the subject.

Despite we do not gathered data about execution times, as we said before, we take some notes about. Consider data in Table 2 non precise, but indicative.

In order to complete time measurements in Table 2, here are some technical informations about relevant hardware characteristics:

- Ultrabook Toshiba Satellite;
- Processor Intel Core i5;
- CPU frequency 1.7 GHz;
- OS Ubuntu 16.10
- No cluster
- No GPU

5.3.6 *Influences of parameters on the network*

What we are about to say has already been said in the previous pages, while we were analyzing all the perspectives. Now we try to summarize the aspect of how diverse numbers of features influenced network performances and times.

It is clear that better trainings can be obtained with higher number of features. But it is also clear that there in not direct proportion between number of features and training speed, stability, accuracy and time execution; so we have to conclude that increasing this key parameter is the way to improve our CNN, but how to increase the values is at your own risk, according to:

- computing power availability;
- time availability;

- desired performances.

We can not leave out an important observation: training a high-valued parameters model can be expensive, very expensive, but we have to consider that this phase is only required the first time, and then we can have a well trained network with very good performances. It might be worth it.

5.4 ANALYSIS OF TEST ACCURACY

6 CONCLUSIONS AND FUTURE WORK

Importance of beeing Ernest

Why CNN are so beautiful

TensorFlow is the most beautiful framework ever in our modest opinion

Our TF model and its results

What a beautiful work you can do next: - changing the padding? What will happen? - in more complex images whit non-uniform background, is 0-padding a good solution? Others solution? How to understand? - trade off between training times and final accuracy according to necessities - transfer leaning! Could be possible use the same trained network re-training the fully connected layer for other tasks? E.G. recognize and classify letters.