UNIVERSITÀ
DEGLI STUDI
FIRENZE

MASTER DEGREE COURSE IN COMPUTER ENGINEERING

# Deep Convolutional Neural Networks for Egyptian Hieroglyphs Classification


*Author:*
Marco LOSCHIAVO

*Supervisors:*
Prof. Fabrizio ARGENTI
Prof. Alessandro PIVA
Prof. Andrea BARUCCI
Prof. Massimiliano FRANCI


April 7, 2021

Abstract of thesis entitled

# Deep Convolutional Neural Networks for Egyptian Hieroglyphs Classification

Submitted by

**Marco LOSCHIAVO**

for the degree of Master Degree

at The University of Florence

in April, 2021

Convolutional neural networks (CNN or ConvNet) are a network architecture for deep learning suitable for image processing and have found enormous success in recent years. Therefore, we are interested in the performance of CNN to classify the ancient Egyptian hieroglyphs. To do this we developed our own CNN architecture and compared it with three other famous architectures.

Two different datasets were used, one that contains 4310 grayscale photos of dimentions 50x75 taken from the book The Pyramid of Unas by Alexandre Piankoff, and one containing 1310 hieroglyphs in the form of sculptures and paintings by Prof. Massimiliano Franci. The images contained in these dataset should be merged into a single dataset and preprocessed. To train the network the dataset have been divided into train, validation and test sets. Image augmentation was applied to the train set in order to improve the effectiveness and generalization of the network.

The results show an excellent ability of our network in recognizing the type of hieroglyph. The model achieves an accuracy of over 97%, even better than other networks. Furthermore, this architecture guarantees a shorter image prediction time and under some hypothesis also a shorter training time.

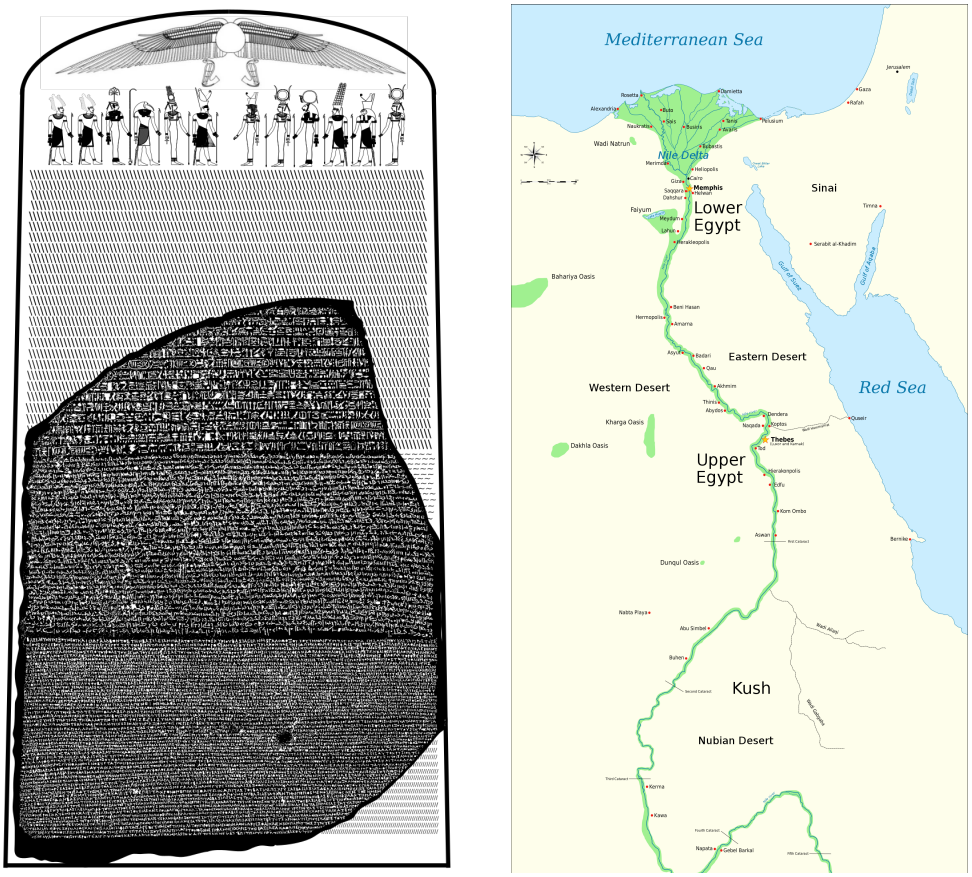
# Contents

# 1

# Introduction

The ancient Egyptian hieroglyphic script was one of the writing systems used by ancient Egyptians to represent their language. It have always been a mysterious writing system as their meaning was completely lost in the 4th century CE because during the Ptolemaic (332-30 BCE) and the Roman Period (30 BCE-395 CE) in Egypt, Greek and Roman culture became increasingly influential and the Coptic became the first alphabetic script used in the Egyptian language.

For many years hieroglyphs were not understood at all.
In 1799 members of Napoleon Bonaparte's Egyptian Campaign discovered The Rosetta Stone in Egypt, a decree issued in 197 BC by Ptolemy V. It was an ancient Egyptian stele, divided into three text registers, with the lower right corner and most of the upper register broken. The stone was engraved with three inscriptions: hieroglyphs in the upper register, Greek at the bottom and demotic (derived from the ancient Egyptian hieroglyphic script, but used for writing on documents). It was hoped that the Egyptian text could be deciphered through its Greek translation, especially in combination with the presence of the Coptic language, the last phase of the Egyptian language.




**(a)** **(b)**

**Figure 1.1:** (a) One possible reconstruction of the original stele of the Rosetta stone. (b) Map of ancient Egypt

This discovery allowed researchers to investigate the hieroglyphs, but it wasn't until 1822 when Thomas Young and Jean-Francois Champollion discovered that these hieroglyphs don't resemble a word, but each hieroglyph resembles a sound and multiple hieroglyphs form a word. Their work has had a revolutionary impact on the deciphering of ancient Egyptian hieroglyphs. Over time some researchers corrected and refined their understanding of Egyptian and, starting in 1850, it was possible to fully translate the ancient Egyptian texts. The ability to understand hieroglyphs has uncovered much of the history, customs and culture of Egypts ancient past.

It is these ancient hieroglyphs that tell the tales of the otherwise long forgotten Pharaohs. Their achievements and victories on the battlefields

have all been stored within these hieroglyphs.

The aim of this thesis is to implement a classifier for ancient Egyptian hieroglyphic using a Convolutional Neural Network.

There are three main parts of this thesis. First, in chapter 2, we will briefly go through theory. First Artificial Neural Networks is explained, leading to Convolutional Neural Networks and Separable Convolution, which is the foundation of this project. The second part, chapter 3 describes the datasets used and the preprocessing techniques used on it, while in chapter 4 we will show the part relating to the classification problem. The last part contains the result, chapter 5, and comments and analysis on these in chapter 6.

# 2

# Machine Learning, Neural network and CNN

Artificial neural networks and especially convolutional neural networks are the main topic of this thesis. So in this chapter we start with a short introduction to the basics of machine learning and continue with an introduction of artificial neural networks and convolutional neural networks.

## 2.1 Machine Learning

Machine learning is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. Machine learning focuses on the development of computer programs that can access data and use it to learn for themselves.

A practical view of a machine learning system is depicted in Figure 2.1. The process is split into two phases. In the first phase, the machine learning algorithm is used to learn from the training data, and the second phase is the prediction. The training data could be labeled images of Egyptian hieroglyphic with the task to predict their meaning. The machine learning algorithm learns a model on these data and this model

can then used to predict unseen images of the same task. This prediction is happen in the second phase and apply only the learned model. In our example, the learned model get images of a hieroglyphic and must predict the meaning.

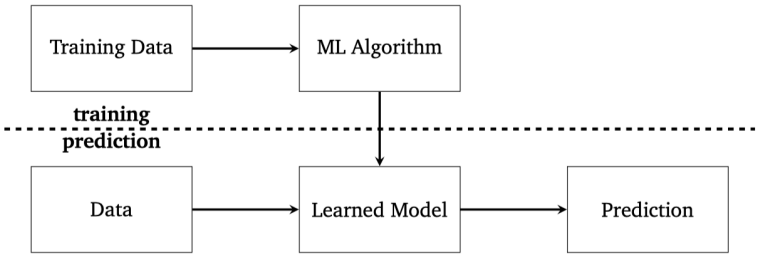



**Figure 2.1:** Workflow of a machine learning problem. The data will be used to train a model with a machine learning algorithm. Then, in the prediction phase, the learned model can be used to generate a prediction.

### 2.1.1 Learning Problems

In machine learning it can be distinguished between different learning problems. The main learning problems are:

**Supervised Learning** : In supervised learning, the machine learning algorithm gets a labeled training set. The training set consists of several examples, and every example is a pair of input data and a labeled output data. The goal is to find a general function or rule that maps the input to the desired output label. Furthermore, the mapping should be general so that unseen data is also correct mapped.

The supervised learning could be further splitted in classification and in regressions learning problems. *Classification* means, that the input of an example is divided into two or more classes. And the goal is to find a mapping of the input to this classes, even if the the input is an unknown example. For example could be used the spam-classification of emails. The classes are “spam” or “not spam”. And the machine learning algorithm should be learn from examples with the given label a function or rule, which can classify an email to these two labels.

Where as *regression* means a learning of a continuous value. Or rather, it should be learned a function, which represents the label value in dependency of the input. This could be for example the price of a house, and the input is the size of the house and the size of the property.

**Unsupervised Learning**: In unsupervised Learning, the machine learning gets an unlabeld training set. The training set has only examples, but no label. The machine learning algorithm should find a structure in the data. This could be done with clustering methods. This means, that examples should be grouped, which have similar properties.

**Reinforcement Learning**: In reinforcement learning the machine learning algorithm interact with an environment and must reach a certain goal. This could be to learn to playing a game, or to driving a vehicle in a simulation. The algorithm gets only information how good or bad he has interact with the environment. For example in learning to play a game, could this information the winning or losing of a game.

The classification of the supervised learning can be further distinguished into

- **Multi-Class**: Multi-class describes only, that it exists more then two classes for one label. For example, the label weather could be sunny, rainy or cloudy, which are three classes for the label weather.
- **Multi-Label**: Multi-label describes the fact, that one label could have multiple classes for one example. For example, a document has the classes “politics”, “sports” and “science”. But sometimes, a document can be classified to two of the classes like politics and sports.

The learning problem of this thesis is associated to the supervised learning and more precisely to a multi-class problem. Our dataset consists of serveral images, and to every images exists labels. The dataset will be described in Chapter 3.

### 2.1.2 Underfitting and Overfitting

A machine learning algorithm must perform well on unseen data. The ability to perform well on unseen data is called generalization. The generalization error is measured on a test set. If a model perform not well on unseen data, then there are two reasons. Either the model has not enough capacity and underfit the underlying function, or it has too much capacity and overfit the underlying function. If a model is underfitting, then will be the training error and also the test error high. If the model is overfitting, then is the training error low and the test error high. The goal is to find a model, which has a low generalization error. The typical curves for training and generalization error are depicted in 2.2.

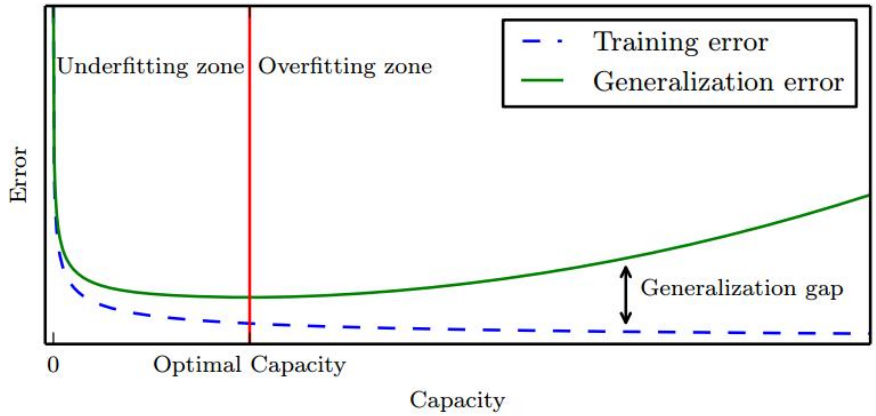



**Figure 2.2:** Shows typical curves for training and generalization error in dependency of the capacity of the model. Optimal capacity is reached at the minimal generalization error. Left of the optimal capacity is the model underfitting. On the right side of the optimal capacity is the model overfitting. The generalization error has typical a U-shaped curve.

In Figure 2.3 are three diagrams depicted, which shows the same noisy sampling of a sinus function. The samples are used learn a model, which describes the underlying function. The left diagram learned a model with a low capacity. So the training error is high and a test set would be also produce a high test error. In the center is depicted a model with a higher capacity. This shows a good approximation of the underlying function. Nevertheless the model has a training error, because we sampled the sinus with noise. But this model will have on a test set the

best generalization error compared to the other two models. The third diagram shows a model with a high capacity. The training error will be low, but the learned model is not a good approximation of the sinus function, which would result on a test set with a high error.

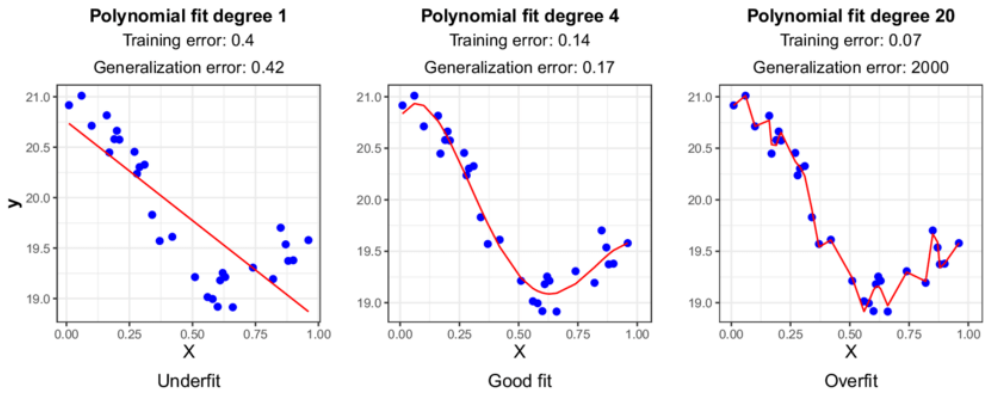



**Figure 2.3:** Theses three diagrams show three different models, which tries to fit the sampled points. The models are described by a polynom of degree 1,4,15. The left model underfits the underlying function. The model in the center is a good approximation of the underlying function. The right model overfits the underlying function. It has the smallest error to fit the sampled points, but on unseen samples, it will be provide a bad error.

### 2.1.3 Hyperparameter and Model Selection

Hyperparamters are parameters which belongs to the machine learning algorithm. With these parameters, the behavior can be changed of the algorithm, to be precise with some of the parameters can be regulate the capacity of the model. These parameters will be not learned during the training, but will be set by the user at start of the learning process. In case of neural networks, this could be the learning rate, or the architecture of the model (number of layers or width of layers). In other words, hyperparameters are parameters, which are not learned during the learning process.

Normally, we test several models with different hyperparameters and chose the model with the lowest error. If we do this test only on the training set, then is the lowest error reached with the model, which has the highest capacity. This is not useful, how we saw in the section about

overfitting. We want the model with the lowest generalization error. So the training set will be further split in a validation set. So that we have three data sets, a training, validation and a test set. The validation set is only used to measure the generalization error, and is not used for the training. The model with the smallest generalization error will then be used as the best model. But the predicted performance on the validation set has a bias, because we chose the model, which maximize the validation set. Therefore, the performance should be not measured on the validation set. That is why we have a third dataset, the test set. On this test set could we now predict the performance of the model, and this is a good approximation, how good the performance will be on unseen data.

If the dataset is small, then is normally used a cross validation, instead of a hard split in training and validation set. In this thesis, we have a small datasets and we decided to do both.

## 2.2 Artificial Neural Networks

An *Artifical Neural Networks* (ANN) – or short only Neural Networks (NN) – is a construction inspired by the human brain, as the word neural indicates. There are similarities between a nerve cell and a node belonging to an ANN. A nerve cell have dendrites handling the cell input and if the stimuli is large enough the signal is passed on to new cells through axons. ANNs are constructed of nodes with weighted input, an activation function and output. In figure 2.4 we can see a comparison between a neuron and its counterpart in Artificial Neural Networks.

Due to ANNs capacity and black box behaviour it is easy to think of it as complicated. Large complex structures may be built with amazing results, but the smallest elements are rather simple. In this section we shall start with the simplest possible network and successively build larger and more complex structures, ending with CNNs, which have been used in this project.

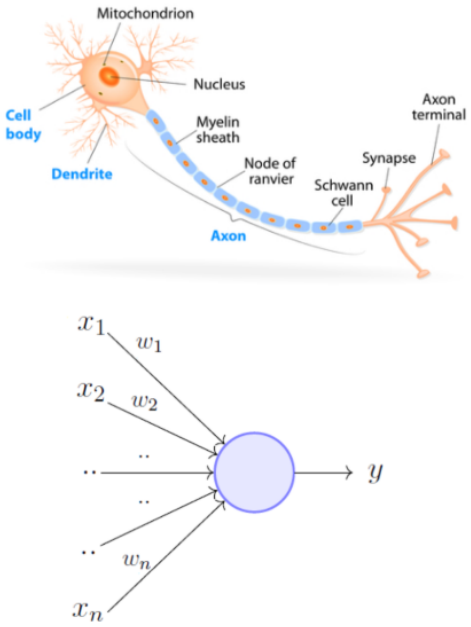



**Figure 2.4:** Comparison between neuron and its counterpart in Artificial Neural Networks.

### 2.2.1 Perceptron

The perceptron is the smallest possible ANN, it consists of only one neuron. The perceptron (see figure 2.5) has multiple inputs $(x_1, x_2, ..., x_n)$ , and only one output $y$ . Furthermore, the perceptron has a bias input, which is called $x_0$ , and has the typical value of $-1$. For every input is derived a linear combination with the weights $(w_0, w_1, ..., w_n)$:

$$z = \sum_{i=0}^{n} x_i \cdot w_i$$

After the linear combination is computed, the value $z$ is inserted into the activation function $\sigma(z)$ . This activation function activates the output, if the threshold is fulfilled. There are different activation functions with different properties, but for the simplicity, we using the heavyside-function:

$$\sigma(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

So, the final form is:

$$y = \sigma(\sum_{i=0}^{n} x_i \cdot w_i)$$

With this activation function, the perceptron has the ability to linear split the hyperspace. This means, that the perceptron can only learn problems, which are linear separable. For many tasks, this is not suffcient. But with multilayer perceptrons, it is possible to solve nonlinear problems, which is as next presented.

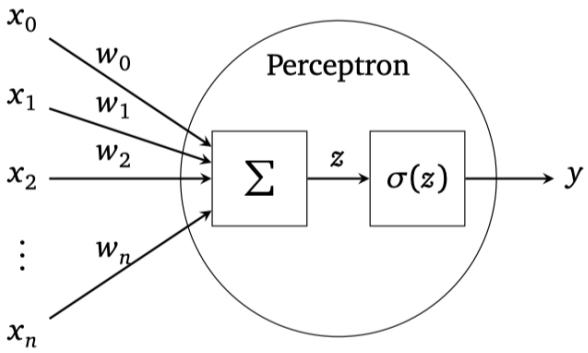



**Figure 2.5:** One perceptron with n inputs and one output. The circle describes the perceptron, in which is applied the linear combination and the activation function.

### 2.2.2 Multilayer Perceptron

The multilayer perceptron or known under the name feedforward neural networks, are a multilayered networks of perceptrons. This means, that several perceptrons are linked together. In Figure 2.6 is an example feedforward neural network depicted. A feedforward neural network consists of multiple layers:

- *Input layer*: the first layer of the network. It consists of a set of $n$ input nodes, without processing capacity, associated with the $n$ inputs of the network: $x_i \in \mathbb{R}, \quad i = 1, \ldots, n$;
- *Hidden Layers*: $L - 1$, whit $L \geq 2$ different layers each consisting of several perceptrons, which are called units
- *Output Layer* the last layer of the network. It consist of $K \geq 1$ neurons whose outputs constitute the outputs of the network $y_i \in \mathbb{R}, \quad i = 1, \ldots, K$;

The depth of a network $L - 1$ describes the number of layers, and the width of a layer describes the number of units. The depicted example network in Figure 2.6 has the depth of three. The width of the hidden layers are four and the width of the input and output layers are two. The bias units are depicted with the $+1$, and will be not count to the width of the layer.

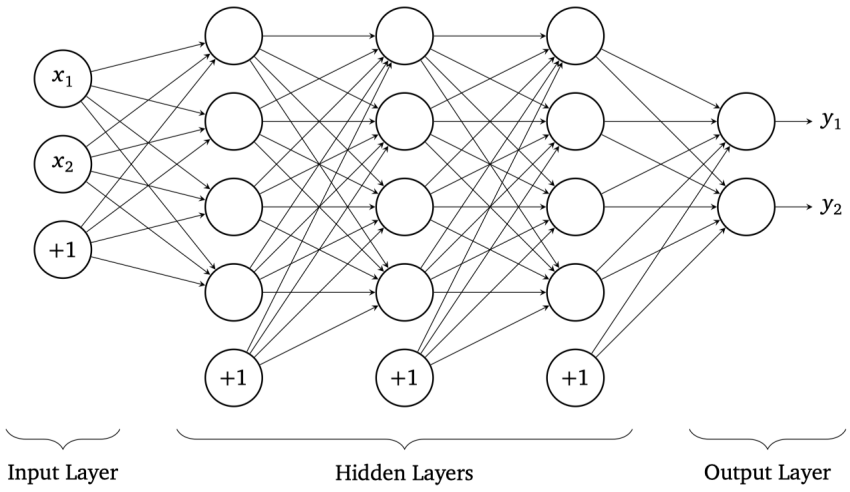



**Figure 2.6:** One perceptron with n inputs and one output. The circle describes the perceptron, in which is applied the linear combination and the activation function.

Between every layer, the units are fully connected (FC). This means that every unit of layer $l + 1$ has the output of all units from layer $l$ as input. The number of links and consequently the number of weights, will fast increase, if the width of the layers are increased. If layer $l$ has $a$ units and layer $l - 1$ has $b$ units, then the number of weights $|W_l|$ between this

layer are $|W_l| = a \cdot (b + 1)$. The $+1$ comes from the bias input. So it is not uncommon that modern NNs have millions of parameter, which must be learned.

Through this concatenation of several layers, the feedforward neural networks gets the capability to solve non-linear problems, if the activation function is not linear. Would be the activation function a linear function, then has the network only the capability to solve linear problems, because the composition of linear function produce again a linear function. So the activation function should be a non-linear function. In Section 2.2.4 we show some useful activation functions.

Furthermore, a feedforward neural network has a very powerful property. With only one hidden layer with sufficient units and the right activation function, like the sigmoid function, the network can approximate arbitrarily closely every continuous functions on a closed and bounded subset of $\mathbb{R}^n$. This means that a network with a single layer has the ability to represent any function. But it is not guaranteed, that the training algorithm will find this function. It is possible, that the training algorithm is not able to find the right value for the parameters. Or the training algorithm choose the wrong function due to overfitting. Furthermore, the single layer may be infeasible large and it exists no heuristic how large the layer should be.

Nevertheless, deeper networks are empirical better and have a lower generalization error as shallow networks with one hidden layer. For example in the recognition of objects in images, the deeper networks learns in the lower layers edges and corners, and compose this information to more complex structures. In other words, in lower layers will be learned simple representations, which then are composed to more complex representations, up to an approximation of the searched function.

For the designing of NNs, it is practical to use a layer as a uniquely building block. So the layers of this section, will be called fully connected (FC), because all units of the layer are connected with the input of the layer. Later we will present other types of layers, like convolution, pooling or dropout layer. Commonly, a layer has only one specific task.

Furthermore, the layers could be seen as a function $f^{(i)}(\mathbf{x}) = \mathbf{y}$, which manipulates the input $\mathbf{x}$ to output $\mathbf{y}$. So a feedforward neural network can be described as a composition of functions with

$$f(\mathbf{x}) = f^{(n)}(\dots(f^{(2)}(f^{(1)}(\mathbf{x})))$$

where $f(\mathbf{x})$ describes the complete network.

### 2.2.3 The training

So far, we have not described how the learning work for ANNs. In this section, we consider only the supervised training. This means, that we have a training set of input vectors $\{\mathbf{x}_n\}$, where $n = 1, ..., N$ , with a corresponding set of output vectors $\{\mathbf{y}_n\}$. The goal is for the training the function $f(\mathbf{x}; \mathbf{w}) = \mathbf{y}$ of the ANN so to adjust, that the function approximates the training data. $\mathbf{w}$ describes the parameter, which are learnable during the training. In our case, this are the weights on the links between the units. Therefore, it will be used a loss function. A simple loss function $J(\mathbf{w})$ is the mean square error (MSE), which compute the squared difference between the $f(\mathbf{x}; \mathbf{w})$ and $\mathbf{y}$. Furthermore, the loss function computes the average loss over the complete training set

$$J(\mathbf{w}) = \frac{1}{2N} \sum_{n=1}^{N} ||f(\mathbf{x}_n; \mathbf{w}) - \mathbf{y}_n||^2 \tag{2.1}$$

The loss function $J(\mathbf{w})$ is used to minimize the error between the training data and the ANN with respect to $\mathbf{w}$. For the minimization of the loss function, we will use a gradient descent method, which needs the gradient of the loss function with respect to $\mathbf{w}$. To calculate the gradient, we using the backpropagation algorithm, which is later in this section introduced. The gradient descent method is an iterative optimization algorithm, which updates the weights with the help of the gradient. In the simplest version, it has the following form:

$$\mathbf{w} \leftarrow \mathbf{w} - \epsilon \cdot \nabla J(\mathbf{w})$$

where $\epsilon$ describes the learning rate. There are some modifications of the gradient descent, which will be later introduced. The gradient descent methods find not necessarily the global minimum. The gradient descent can find a local minimum or in the worst case it can stuck in a saddle point.

As next, we show the algorithm for the backpropagation. After this we introduce a loss function for the classification problem, and then we describe the modification of gradient descent for a faster convergence.

**Backpropagation**

The backpropagation is an algorithm to propagate back the error from the loss function through the network to compute the gradient $\frac{\partial J(\mathbf{w})}{\partial w_{j,k}^{(l)}}$. Backpropagation is not the learning algorithm of the ANN, but rather an efficient method to compute the gradient in respect to the weights. The learning algorithm is the gradient descent, which needs the gradient of the loss function, to update the weights. To derive for every weight the gradient, the backpropagation algorithm has the following recursively equation:

$$\frac{\partial J(\mathbf{w})}{\partial w_{j,k}^{(l)}} = \delta_j^{(l)} \cdot y_k^{(l-1)}$$

with

$$\delta_j^{(l)} = \begin{cases} \frac{\partial J(\mathbf{w})}{\partial y_j^{(l)}} \cdot \sigma_l'(z_j^{(l)}) & \text{if } l \text{ is the output layer} \\ (\sum_{i=1}^{q} \delta_j^{(l+1)} \cdot w_{i,j}^{(l+1)}) \cdot \sigma_l'(z_j^{(l)}) & \text{if } l \text{ is a hidden layer} \end{cases}$$

Where $y_j^{(l)}$ describes the output of unit $j$ in layer $l$ and $q$ is the size of units in layer $l+1$. The function $\sigma_l$ is the activation function of layer $l$ and $z^{(l)} = \sum_{i=1}^{K} w_{j,i}^{(l)} \cdot y_i^{(l-1)}$ is the activation value from unit $j$ in layer $l$. In Figure 2.7 is for a better understanding of the values for one unit $j$ in layer $l$ the values depicted.

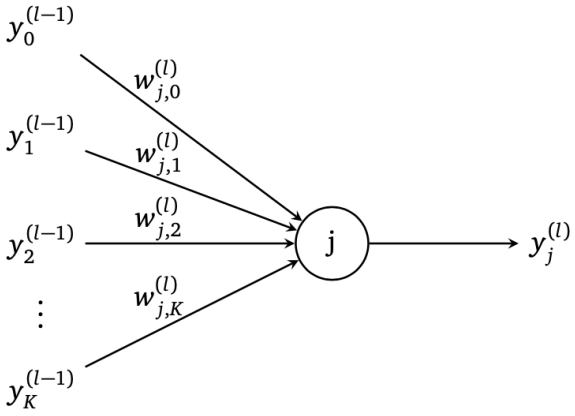



**Figure 2.7:** Shows one unit of the layer l.

Before the backpropagation process can be started, the forward propagation must be computed, to generate the activation values $z^{(l)}$ and the output of $y^{(l)}$. These values are needed for the backpropagation. After all gradients are computed, the gradient descent step can be applied, with

$$w_{j,k}^{(l)} \leftarrow w_{j,k}^{(l)} - \epsilon \cdot \frac{\partial J(\mathbf{w})}{\partial w_{j,k}^{(l)}}$$

And then the next iteration can be started. This is repeated, until an abort criterion is reached. This could be a maximal number of iterations, or that the loss is smaller than a predefined value.

### Loss Function

In loss function equation 2.1, we introduced the MSE loss function. This is one of many loss functions, which can be used. A loss function maps values of one or more variables to a single value of $\mathbb{R}$, and this value represents the loss. The loss describes the discrepancy between the function $f(\mathbf{x}; \mathbf{w})$ and the target value $\mathbf{y}$ . The learning of the neural network is the reducing of the loss, which are described by the loss function. Therefore, the loss function J($\mathbf{w}$) is minimized with respect to $\mathbf{w}$ , which are the parameters of the neural network.

It exists different loss functions and the chose of the loss function depends on the learning problem. In regression problems, the MSE loss

function is a good choice. But for the problem of classification it is not so practicable. Therefore, it exists a common loss function for multi-class problems. The name is cross-entropy loss or also known as log loss. It calculates a loss between two distributions, with

$$L(\mathbf{p}, \mathbf{q}) = -\sum_{i=0}^{K} p_i \cdot \ln q_i \tag{2.2}$$

where $\mathbf{p}$ is the target distribution , $\mathbf{q}$ is distribution which is computed by the neural network , $p_i$ and $q_i$ are the groundtruth and the CNN score for each class $i$ in $K$.

To understand why the output of the network is now a distribution, we do an excursion to the output encoding for classification. For classification, the output is so encoded, that every class has its own output, which is represented in a vector of $\mathbf{q}$. The target label is also encoded as a vector $\mathbf{p}$ with the property, that only one entry is 1 and all other are 0. This is called a one-hot vector. The classes are enumerates, so that every class represents an entry in the vector. This means, that the class $i$ is represented in the vector $\mathbf{p}$ at index $i$ . The vectors $\mathbf{p}$ and $\mathbf{q}$ are then a distribution, if the sum of all the entries are 1, and all entries are $\geq 0$. For the one-hot vector $\mathbf{p}$, it is easy to see, that this property is fulfilled. But the computed vector of the neural network has not necessary this property. Therefore, it is used the softmax activation function at the output layer to get a distribution for the output. It changes the output so, that the required properties are fulfilled. We introduce the softmax activation function later. The learning goal is now to minimize the loss between this two distributions. The nice side effect is, that the distribution $\mathbf{q}$ gives us the percentage how sure the neural network is, that the prediction is this classes.

After we introduced the output encoding, we would like add a note to the Equation 2.2. This is not the final loss function $J(\mathbf{w})$, because this describes only the loss of one example in the training set. How in Equation 2.1 for the MSE, the average is computed of the cross entropy loss over all training examples.

### Gradient Descent

The gradient descent algorithm is an iterative optimization algorithm to find a local minimum of a function, in our case it will be $J(\mathbf{w})$. For a better overview, we show the equation seen previously in 2.2.3 again:

$$\mathbf{w} \leftarrow \mathbf{w} - \epsilon \cdot \nabla J(\mathbf{w})$$

The choice of the learning rate $\epsilon$ is important, because if we use a to small step size, the algorithm needs very long and it could stuck in a local minima. If $\epsilon$ is too large, it could happen that you bypass the domain area with the best solution. Hence it is one of the main hyperparameters of a neural network, and should be chosen carefully. The starting point is also important, because another starting point results in a different path, which could find a better (local) minimum.

For the gradient descent exists some interesting extensions.

**Stochastic Gradient Descent**: The first and most important extension is the descent of the Stochastic Gradient (SGD) with mini-batch. It iterates the gradient descent with a small subset of examples from the training set, instead of the full training set. Normal gradient descent is very impractical as far as computation time and update rate are concerned. For an update, work out the complete training set, which can contain millions of examples. This means that it has to calculate millions of forward propagations before it can calculate a backpropagation step to update the weights. With SGD with mini-batches, it makes a weight update with a small sample size, but with this huge amount of updates it will find the right direction, which minimizes the loss.

$$\mathbf{w} = \mathbf{w} - \frac{\epsilon}{n} \cdot \sum_{i=0}^{n} \nabla J_i(\mathbf{w})$$

where

$$J(\mathbf{w}) = \frac{1}{2N} \sum_{n=1}^{N} ||f(\mathbf{x}_n; \mathbf{w}) - \mathbf{y}_n||^2 = \frac{1}{2N} \sum_{n=1}^{N} J_i(\mathbf{w})$$

At this point, it is appropriated to introduce two common terms in the training of neural networks:

- **Iteration**: One iteration describes one update with the gradient descent. In other words, it is one learning step.
- **Epoch**: One epoch describes one pass through the complete training set. Normally, one epoch consists of several iterations, because the size of the training set is much bigger than the batch size.

**Momentum Update**: A further extension of the gradient descent is the momentum update. The learning with normal gradient descent is sometimes slow, but can accelerate with the momentum method. The path of the normal gradient descent goes mostly in zigzag lines to the minimum. With the help of the momentum method is tried so solve this problem, by using a decaying moving average of the past gradients. The formula for the gradient descent with momentum update is the following

$$\boldsymbol{\phi} \leftarrow \mu \boldsymbol{\phi} - \nabla J(\mathbf{w})$$

$$\mathbf{w} \leftarrow \mathbf{w} + \boldsymbol{\phi}$$

It introduces a further hyperparameter $\mu \in [0; 1)$, which describes the decaying rate of the momentum. A higher value slows down the decaying rate. Commonly values for $\mu$ are 0.5, 0.9 and 0.99. The momentum helps to stay in the direction of the previously gradient updates.

**ADAM**: Adaptive Moment Estimation tries to overcome the problems of updating by modifying the learning rate in a different way for

each parameter and according to the stage of learning. It is therefore part of the family of adaptive methods. In particular, the algorithm calculates the mean with exponential decay of the gradient and of the square of the gradient; the parameters $\beta_1$ and $\beta_2$ control the decay of these moving averages. The authors recommend the values for these parameters, which are in fact the default values also in every framework that supports Adam.

### 2.2.4 Activation Functions

The activation function $\sigma(z)$ is the main part of a neural network, which gives the network the power to solve non-linear problems. If the units have no activation function or only a linear activation function, then the neural network could only solve linear problems. No matter how deep the neural network is. So the activation function should have a non-linear characteristic. A further property for the activation function is, that it must be continously differentiable. Otherwise, the backpropagation algorithm will not work, because it cannot compute the gradient.

In the beginnings of the research of neural networks, there were used saturated activation functions. This means that the activation was limited to $(0, 1)$ or $(-1, 1)$. As activation functions were used the *sigmoid* or *tanh* function. Both have the problem, that by small or big values of $z$ the value of the gradient goes to zero, and the convergence of the learning decreases. In other words, the training of the neural network needs much more time. Another problem is the vanishing or exploding of the gradient, if the network is depth. The vanishing gradient occurs in earlier layers, because through the backpropagation will often be multiplied with small values. Many multiplications with small values between 0 and 1 tends toward zero. The exploding gradient, can happen, if the weights are big, and the activation is near 0. At this point the derivative of sigmoid and tanh has reached their maximum.

With the time, new activation functions are developed, like the ReLU. In this section, we want to show two activation functions, which are used in this thesis. These are ReLU and Softmax.

**ReLU- Rectifiere Linear Unit**: Actually, the Rectifire Linear Unit (ReLU) is the most used activation function for neural networks. The formula for this function is simple

$$\sigma(z) = \max(0, z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

This means, that the function is 0, if $z$ is negative otherwise the value is $z$ . The ReLU activation is in Figure 2.8 depicted. The interesting fact is, that the function is for positive values linear and not saturated. A further property, which makes ReLU so popular is the fast computation of the derivative. The derivative is either 0 (at negative values) or 1 (at positive values). Through this, the vanishing of the gradient is not more a problem, because a multiplication with 1 doesn't change the error.

Batch normalization with ReLU is heavily used in modern network architectures. We introduce batch normalization in the next section, but ReLU with batch normalization increases the learning speed of the network. This comes from the normalization of the batch normalization, so that the mean is zero. It is known, that the input with zero mean increases the convergence.

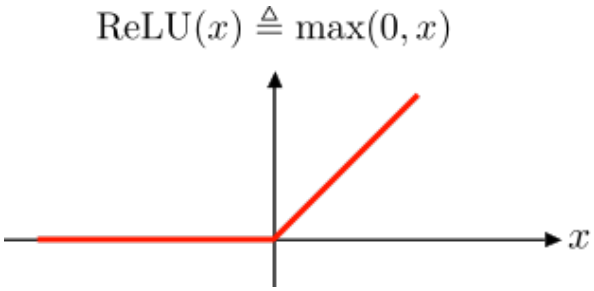



**Figure 2.8:** Diagram for the activation functions of ReLU and ELU

**Softmax**: In the section about the loss function, we have used the softmax activation for the encoding of the output. This is the main application of the softmax activation function, to represent a probability distribution over $K$ classes, and therefore it is only used in

the output layer. The softmax is applied on the complete output layer and the formula is the following

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \qquad \text{for } j = 1, \ldots, K$$

It scales every output of $\sigma(\mathbf{z})_j$ to a range between 0 and 1, and the sum of $\sum_{j=1}^{K} \sigma(\mathbf{z})_j = 1$. This represents a probability distribution. With the exponential function, the largest value in $z$ will be highlighted and the other values will be suppressed. For example, let $z = [8, 5, 3]$ , then is the activation $\sigma(z) = [0.9523, 0.0474, 0.0003]$ . The value 8 get much more of the available space, as the other values.

### 2.2.5 Regularization

Regularization are methods to control the overfitting or rather to improve the generalization error. There are different methods, how to apply regularization. Some of the methods are common approaches in machine learning, and other are only usable for neural networks. We show three regularization methods, which are used in this thesis.

**Early Stopping**: On training of large models, normally the training and validation error decreases over the time, but at one point the validation error starts to increase. At this point the model is starting to overfit, and learn specific properties of the training set. To stop at this point, it is applied the method of Early stopping. It returns the model, which has the lowest validation error. Therefore, the training needs a validation set, to evaluate periodically the validation error. The normal period is after every epoch.
But how could we ensure, that this increasing was not caused by noise? The training is not stopped after the first increasing of the validation error. The network is further trained until a threshold of "number of epochs without improvements" is reached. Through the evaluation of further epochs, we get the trend of the validation error for more training. For example, if 10 times in a row, the

validation error has no improvements compared to the best validation error, then is the training stopped, and the model with the best validation error is returned.

**L2-Regularization**: Let $\mathbf{w} = (w_0, \ldots, w_N)$ and $\boldsymbol{\omega} = (w_1, \ldots, w_N)$ , i.e $\boldsymbol{\omega}$ is the weights vector without the bias weight.
*L2-Regularization* is a regularization methods, which is added to the loss function $J(\mathbf{w})$. It penalizes the weights $\boldsymbol{\omega}$ from $\mathbf{w}$ , but not the bias weights. The regularization term $\Omega(\mathbf{w}) = \lambda \frac{1}{2} ||\boldsymbol{\omega}||^2$ is added to the loss function. This results in a new loss function

$$\tilde{J}(\mathbf{w}) = J(\mathbf{w}) + \Omega(\mathbf{w})$$

which is used to minimize the loss of the neural network. Furthermore, it is added a further hyperparameter $\lambda$ , which represents the strength of the regularization.

So, through the adding of the quadratic weight to the loss function, are weights with a high value penalized, because they increases the loss. In other words, the L2-regularization brings the weights closer to the zero.

**Batch Normalization**: The training of a network changes the weights on every layer. This change has the effect, that the input distribution changes during updates of previously layers. The authors of batch normalization called this effect internal covariate shift. To counteract the change of the distribution during the learning, they introduced the batch normalization. Every mini-batch, which is inserted in the network, will be on every layer normalized on the input of the previously layer. The formula for the normalization is the following

$$\mu \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2$$

$$\tilde{x_i} \leftarrow \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

where $\mu$ and $\sigma$ describe the mean and the standard deviation, $\tilde{x_i}$ is the normalized value of the input. The value $m$ describe the size of the mini-batch. The value $\tilde{\mathbf{x}}$ has then a mean of 0 and a standard deviation of 1. The batch normalization will be applied after the linear combination of a unit.
To not lost the representation power of the units, the value $\tilde{x_i}$ will be scaled and shifted

$$y_i \leftarrow \delta\tilde{x_i} + \beta$$

where $\delta$ and $\beta$ are learned parameter during the training. So the value $y_i$ could have any mean and standard deviation. This is useful in the case of using sigmoid as the activation function. Without the scaling and shifting, the activation function will be act as an almost linear function, because through the normalization, the input is scaled to a range, on which the sigmoid function is almost linear. At test time, $\mu$ and $\sigma$ are replaced with the average, which is collected during the training. So it is possible to predict a single example, without to have a minibatch.
Through the applying of batch normalization, the learning rate can be increased, and this results in a faster training. Furthermore, the accuracy is increasing compared to the same network without batch normalization.
The batch normalization helps the activation function ReLU to learn faster and have a better performance.

## 2.3 Convolutional Neural Networks

Convolutional neural networks, which we will refer to with the abbreviation CNN, are an evolution of the normal deep artificial networks characterized by a particular architecture that is extremely advantageous for visual (and non-visual) tasks, which has made them very effective and popular over the years. . They were inspired by the biological research of Hubel and Wiesel who, studying the brains of cats, had discovered that their visual cortex contained a complex structure of cells. The latter were sensitive to small local parts of the visual field, called receptive fields. They thus acted as perfect filters for understanding the local correlation of objects in an image. As these systems are the most efficient in nature for understanding images, the researchers attempted to simulate them.

### 2.3.1 Architecture

CNNs are deep neural networks made up of different layers that act as feature extractors and a fully connected network to the end, which acts as a classifier, as shown in Figure 2.9.

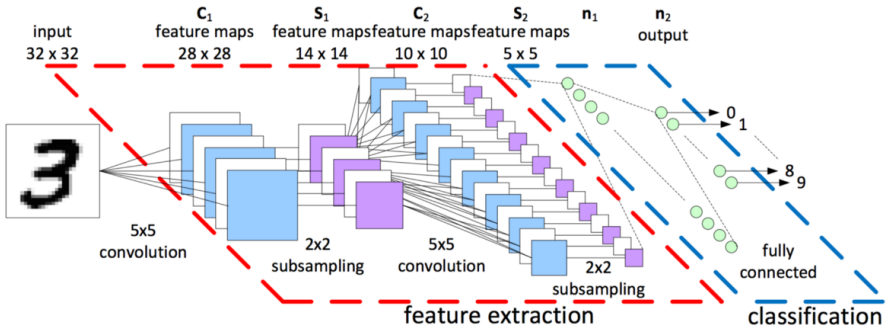



**Figure 2.9:** Architecture of a CNN that classifies numbers: the division between the layers that act as feature extractor and the final classifier is highlighted

These layers where image features are extracted are called convolution layers and are generally followed by a non-linear function and a pooling step. There may then be image processing layers, such as the contrast normalization layer, see Figure 2.10.

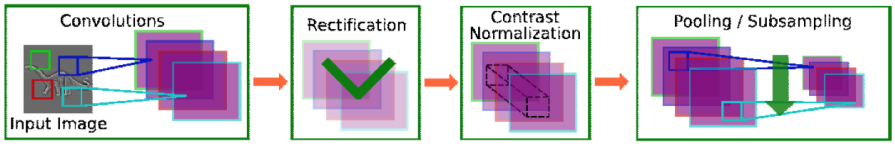



**Figure 2.10:** The different layers typical of a CNN

Convolution and pooling have the purpose of extracting the characteristics, while the non-linear unit serves to strengthen the strongest characteristics and weaken the less important ones, that is, those that have stimulated the neurons less. From Figure 2.9, we can also note that, for each input image, different groups of images correspond in the various layers, which are called feature maps. The feature maps are the result of the convolution operation carried out through a bank of filters, also called kernels, which are nothing more than matrices with useful values to search for certain characteristics in the images. Finally, once the convolutional layers are finished, the feature maps are "unrolled" into vectors and given to a "classical" neural network which performs the final classification.

### Convolutional Layer

A digital grayscale image can be considered as a two dimensional matrix real or discrete values. Each value of the matrix is called a pixel and its indices are also called coordinates: each pixel $\boldsymbol{X}(m,n)$ represents the intensity in the position indicated by the indices. A “filter” or “kernel” is defined as a matrix $\boldsymbol{W}$, usually (3x3), with weights to be applied to the image through a transformation in order to produce a filtered output. The transformation is carried out through the convolution operation between the input image and the filter. The convolution, discrete in the case of digital images, can be defined as:

$$\boldsymbol{F}(m,n) = \boldsymbol{X}(m,n) \circledast \boldsymbol{W}(m,n) = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} \boldsymbol{X}(i,j) \cdot \boldsymbol{K}(m-i, n-j)$$

where $\boldsymbol{F}$ is the feature maps. Each pixel of $\boldsymbol{F}$ is thus the result of

a weighted sum by $\boldsymbol{K}$ of the sub-region which has its center in the pixel indicated by the coordinates $m, n$. An example of a convolution is shown in Figure 2.11.

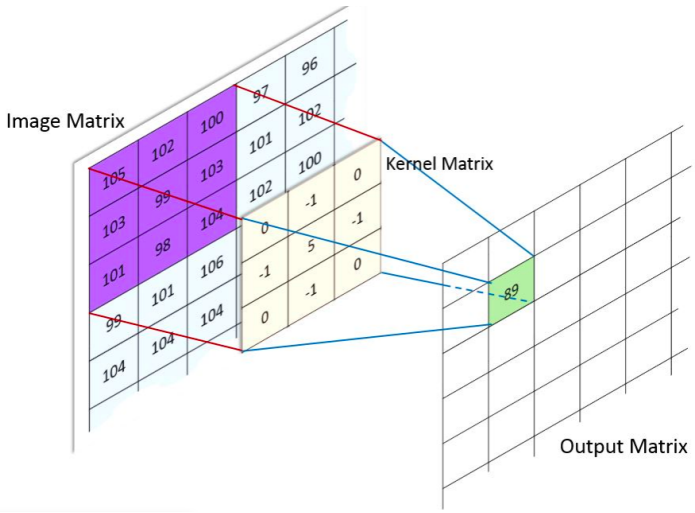



**Figure 2.11:** Example of convolution

In reality, more generally, a convolutional layer does not only take in two-dimensional matrix as in the case of grayscale images, but three-dimensional matrix. Infact, a colored image has normally three channels (red, green, blue), and every channel is described by a two dimensional matrix. The output of a convolution layer is again a three dimensional matrix, with feature maps of two dimensions and this x times for the number of filters for this layer. Every filter produces a feature map. In Figure 2.12 is provided an example. The two figures show the connection between the input and two output units of the feature map. The input is showed with three channels. This could be the color channels R,G,B of an image, or the feature maps of a previously layer. The input is described by a three dimensional matrix. As output is depicted one feature map. For more feature maps is the behavior similar. The activation function and the bias are omitted for this example, but would be also applied on the units. The bias are shared, which means, that one bias weight are used for all units in one feature map. The weights $\boldsymbol{W}_{i,o}$ are a $3 \times 3$ matrix, which describe one kernel, between channel $i$ of input image $\boldsymbol{X}_i$ and the feature map $\boldsymbol{F}_o$. The weights $\boldsymbol{W}_o$ , describe one filter and are shared for one feature map. This means, that the same weights are used for the

computation of all units in one feature map. Then for convolutional layer with input of three-dimensional matrix the output or $o - th$ feature map is calculated with:

$$\boldsymbol{Z_o} = \sigma(\sum_{i=0} \boldsymbol{X}_i \circledast \boldsymbol{W}_{i,o} + b_o \mathbf{1}) \tag{2.3}$$

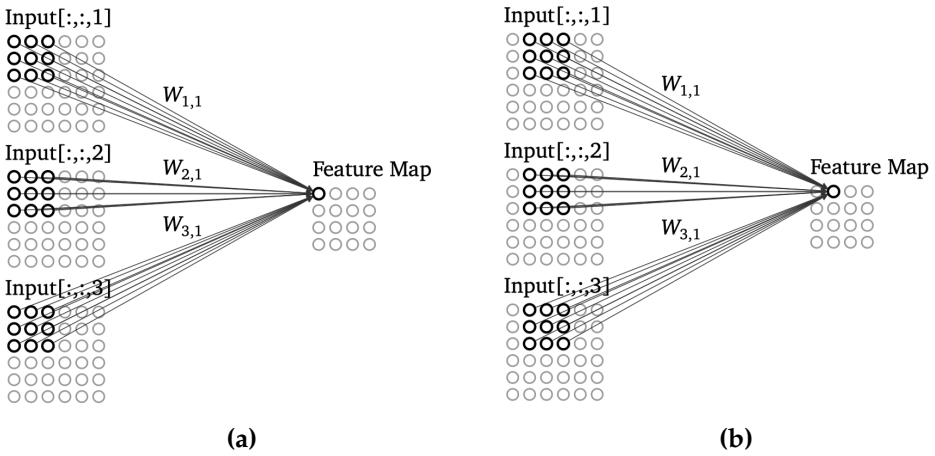



**Figure 2.12:** The two figures show the connection between the input and two output units of the convolution layer. Furthermore, the convolution layer has only one feature map for the output. The activation function and the bias are omitted for a better overview. The local connectivity is depicted in the two figures, with a kernel size of 3×3. (a) and (b) shares the same weights for the computation of the feature map. For another feature map, it would be used new weights. $\boldsymbol{W}_{i,o}$ , describes a 3×3 weight matrix, which is used between the input channel $i$ and the output channel $o$. The output channel is called feature map.

The convolution in a convolution layer are multiple convolutions, which are added up, which we can see in Equation 2.3. Without the addition of the convolutions of the input channels, we would not get a combination of features, which are essential for a CNN.

As already mentioned, these convolutions are performed between the input image/s and an arbitrary number of filters. These filters have values such as to obtain recognition of certain characteristics at the output. The values of the filters are initially chosen randomly, and are then improved at each iteration using the backpropagation algorithm, seen

in 2.2.3. By doing so, the network trains its filters to extract the most important features of the examples of the training set.

There are several hyperparameters to set in the convolution layers:

1. The measure of the filter $F$: also called receptive field. Each filter looks for a specific feature in a local area of the image. Typically they are 3x3, 5x5 or 7x7.
2. The number $K$ of filters: for each layer, this value defines the depth of the output of the convolution operation. In fact, by placing the feature maps one on top of the other, you get a cube in which each "slice" is the result of the operation between the input image and the corresponding filter. The depth of this cube depends precisely on the number of filters.
3. The *stride* $S$: defines how many pixels the convolution filter moves at each step. If the stride is set to 2, the filter will skip 2 pixels at a time, thus producing a smaller output.
4. The *padding* $P$: defines the measure with which you want to add "0" to the input to preserve the output size. In general, when the stride $S = 1$, a value of $P = (F - 1)/2$ guarantees that the output will have the same size as the input.

We can calculate the number of parameters $p$ in a convolutional layer knowing the number of input channels $I$ and the value of the hyperparameters, with

$$p = (F^2 \cdot I + 1) \cdot K$$

When processing images with CNNs, three-dimensional inputs are generally received, characterized by the height $H_1$, the width $W_1$ and the number of color channels $D_1$. Knowing the parameters specified above, you can calculate the size of the output of a convolution layer:

$$H_2 = (H_1 - F + 2 \cdot P)/S + 1$$

$$W_2 = (H_1 - F + 2 \cdot P)/S + 1$$

$$D_2 = K$$

In this regard, observe an example of "volume of neurons" of the first convolution layer in Figure 2.13. Each neuron is spatially connected only to 1 local region of the input but for the whole depth (i.e. the 3 color channels). Note that there are 5 neurons along the depth and they all look at the same input region.

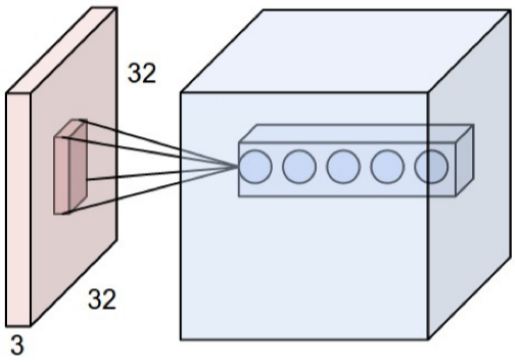



**Figure 2.13:** Each neuron is connected to only 1 local region of the input but at all depth (i.e. color channels). The depth of the output is given by the number K of filters, in this case 5

### ReLU Layer

After the convolution layer the features maps are passed in input to the ReLU 2.8 layer. The ReLU layer is very important for a CNN. This can be mainly due to 2 reasons:

1. the polarity of the features is very often irrelevant to recognize objects
2. the ReLU prevents two characteristics, both important but with opposite polarity, from canceling each other when pooling.

### Pooling Layer

Another property that you want to obtain to improve results on artificial vision is the recognition of features regardless of the position in the image, because the goal is to strengthen the effectiveness against translations and distortions. This can be achieved by decreasing the spatial resolution of the image, which favors a higher computation speed and is

at the same time a countermeasure against overfitting, since the number of parameters decreases.

The pooling layer gets $N$ images of a resolution in input and outputs the same number of images, but with a resolution reduced to some extent, usually by 75%. In fact, the most common form of pooling layer uses $2x2$ filters, which divide the image into non-overlapping 4-pixel areas and choose a single pixel for each area.

The criteria by which to choose the winning pixel are different:

- *Average pooling*: the average value on the pixels of the pool is calculated
- *Median pooling*: the median of the pixel values in the pool is calculated
- *LP-pooling*: the p-norm of the pixel matrix is calculated
- *Max pooling*: the pixel with the highest value is chosen

Of these, the one that has proved most effective is max pooling, Figure 2.14. Crossing the network you will gradually have a higher num-

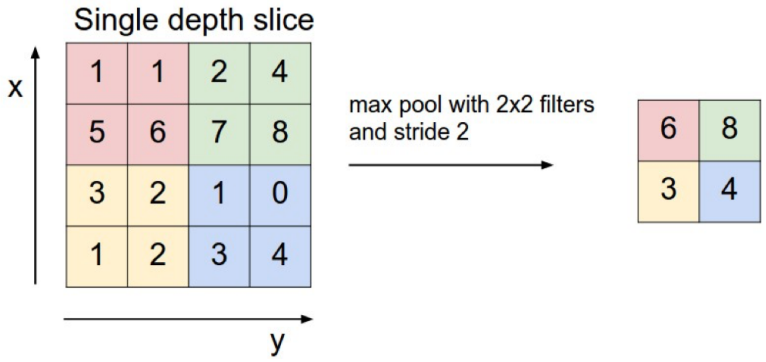



**Figure 2.14:** Max pooling: at the output the image will have 1/4 of the starting pixels.

ber of feature maps and a decrease in input resolution. These factors combined together give a strong degree of invariance to the geometric transformations of the input.

### Fully connected Layer

In the fully connected layer, all inputs from the convolution layers are fed into a normal, fully connected neural network that will act as a classifier.

Figure 2.15 shows the complete architecture of a CNN. Note how the resolution of the image is reduced at each pooling layer (also called subsampling) and how each pixel of the feature maps derives from the receptive field on the set of all the feature maps of the previous level.

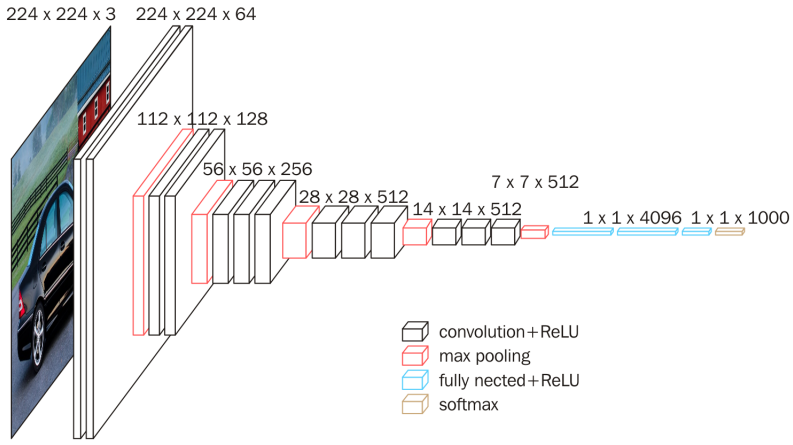



**Figure 2.15:** CNN Architecture

Figure 2.16 instead, you can see a CNN in the classification deed of a car. The network filters are displayed during all the various levels of input processing, and then terminate in a completely connected layer that gives a probability in output. This probability is then translated into a score, from which the winning class is chosen.

## 2.3.2 Depth-wise Separable Convolution

Standard convolution layer of a neural network, as seen previously, involve (without considering the bias) $F^2 \cdot I \cdot K$ parameters. For an input channel of 3 and output of 256 with 5x5 filter this will have 19200 parameters. Having so much parameters increases the chance of overfitting. To avoid such scenarios, people have many a times looked around for different convolutions.

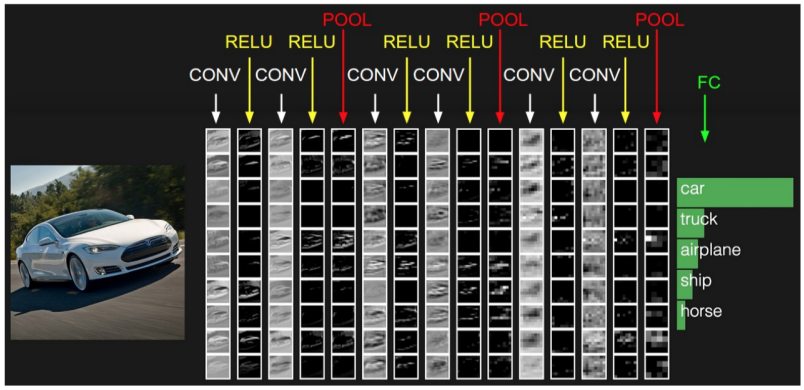



**Figure 2.16:** Typical CNN in a classification task; the winning class is the one with the highest probability, indicated at the end.

Depth-wise Separable Convolution originated from the idea that depth and spatial dimension of a filter can be separated - thus the name separable. This type of convolution separates the process in 2 parts: a depthwise convolution and a pointwise convolution.

1. **Depthwise Convolution**: In this first part we give the input image a convolution without changing the depth. We do so by using $I$ kernels of shape $QxQx1$, where a typical choice of $Q$ is 3 or 5. For example (see Figure 2.17), if we have a input image of 12x12x3, we can use 3 kernels of shape 5x5x1 in order to get a 8x8x3 feature map (convolutions with no padding and a stride of 1).

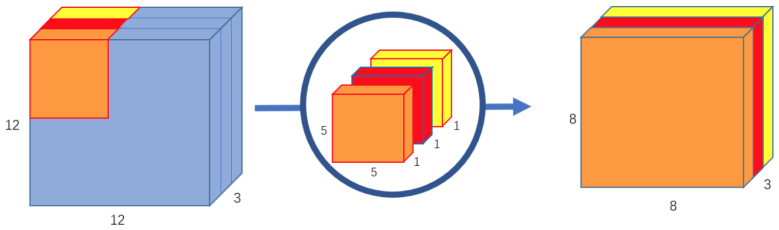



**Figure 2.17:** Depthwise Convolution

2. **Pointwise Convolution**: The pointwise convolution is so named because it uses a 1x1 kernel, or a kernel that iterates through every single point. This kernel has a depth of however many channels the input image has. We can create $S$ $1x1xI$ kernels in order to increase the number of channels of each image. For example (see Figure

2.18), if we have a input image of 8x8x3, we can use 256 kernels of shape 1x1x3 in order to get a 8x8x256 feature map (convolutions with a stride of 1).

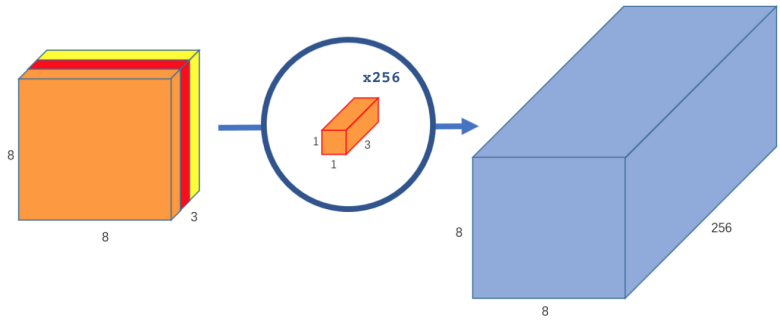



**Figure 2.18:** Pointwise Convolution

In above example the number of parameters using Depth-wise Separable Convolution is: 3x5x5x1 + 1x1x3x256 = 843, much less than the normal convolution, who has 19200.

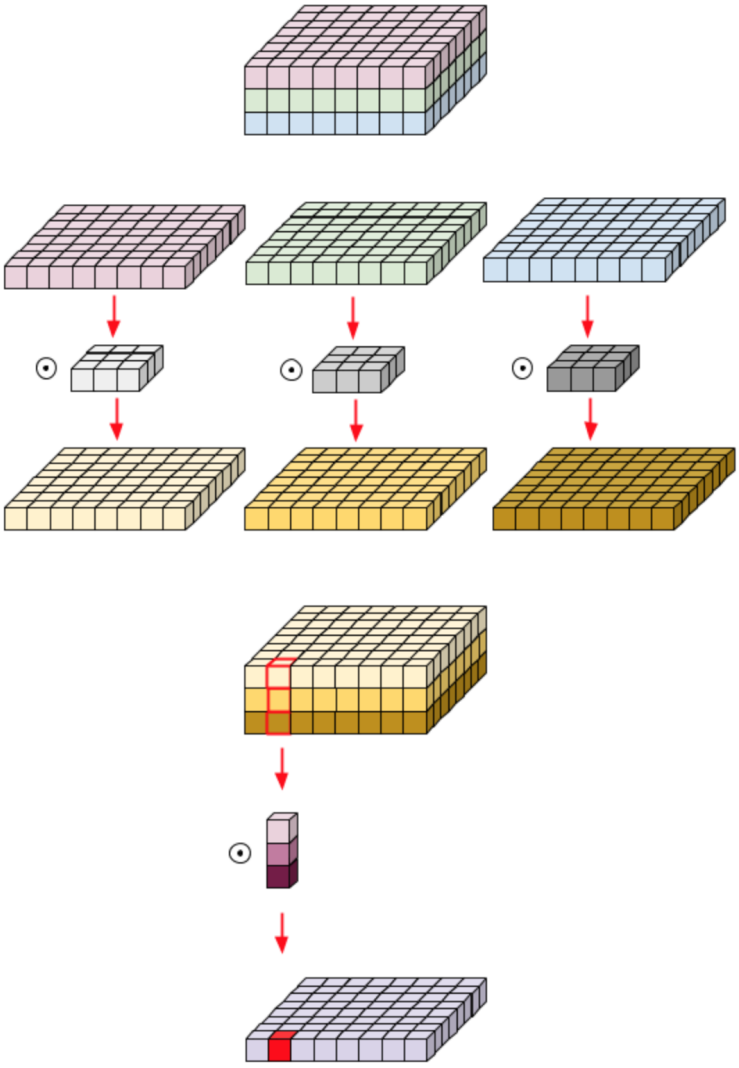


**Figure 2.19:** Depth-wise Separable Convolution

# 3

# The Datasets

In this thesis have been used two different dataset, one containing annotated images of Egyptian hieroglyphs found in the Pyramid of Unas and one containing images selected and annotated by Prof.Massimiliano Franci. Before the images are fed into the Neural Network, the images contained in these dataset should be merged into a single dataset, preprocessed and augmented . The preprocessing steps consist of change the colorspace, denoising and scale the image. This chapter starts with an overview of the initial dataset and after this, will be presented the preprocessing and data augmentation techniques.

## 3.1 Images

The first dataset ($D_1$) contains 4310 grayscale photos of dimentions 50x75 taken of walls inside the pyramid of Unas. The photo's are taken from the book *The Pyramid of Unas* by Alexandre Piankoff. Each photo rappresent a hieroglyphs and it is labeled with its class of belonging. In this dataset there are 172 different class.

The second dataset ($D_2$) instead contains 1310 rgb images of variable dimensions with 48 different classes. Also in this case each images rappresent a hieroglyphs and it is labeled with its class of belonging.

The hieroglyphs in the images were labelled according the Gardiner Sign list [2], where each unique hieroglyph is labelled with an alphabetic character followed by a number. The alphabetic character represents the

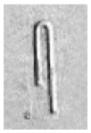
| Gardiner label | I9 | G17 | E34 | M17 | S29 |
|---|---|---|---|---|---|
| Image |  |  |  |  |  |

**Figure 3.1:** Examples of some images labeled according the Garniner Sign list

class of the hieroglyph, for example, class 'A' contains hieroglyphs about a man and his occupations while class 'G' contains hieroglyphs of birds. The entire Gardiner Sign list consist of more than 1000 hieroglyphs in 25 classes.

Starting from these two datasets we then obtained a single dataset $D$. The dataset $D$ that we got from $D_1$ and $D_2$ and which was used to apply the data augmentation is not a simple the union of these datasets. We have decided to take only the images that belong to a class contained in both datasets
So, let

$$D_1 = \{(x_i^{(1)}, y_i^{(1)})\} \quad \text{with} \quad i = 1, \ldots, |D_1|$$

$$D_2 = \{(x_j^{(2)}, y_j^{(2)})\} \quad \text{with} \quad j = 1, \ldots, |D_2|$$

and $Y_1$ , $Y_2$ the set of all the different labels in $D_1$ e $D_2$.
Then,

$$D = \{(x_k, y_k)\} \quad \forall y_k \in Y_1 \cap Y_2$$

$|Y| = |Y_1 \cap Y_2|$ is the number of classes contained in our dataset$D$. In our case there are 40 different class, so $|Y| = 40$. In figure 3.2 we can see a histogram that shows the number of images for each class of belonging

it is possible to see that the dataset is quite unbalanced. This is one of the reasons we use data augmentation.

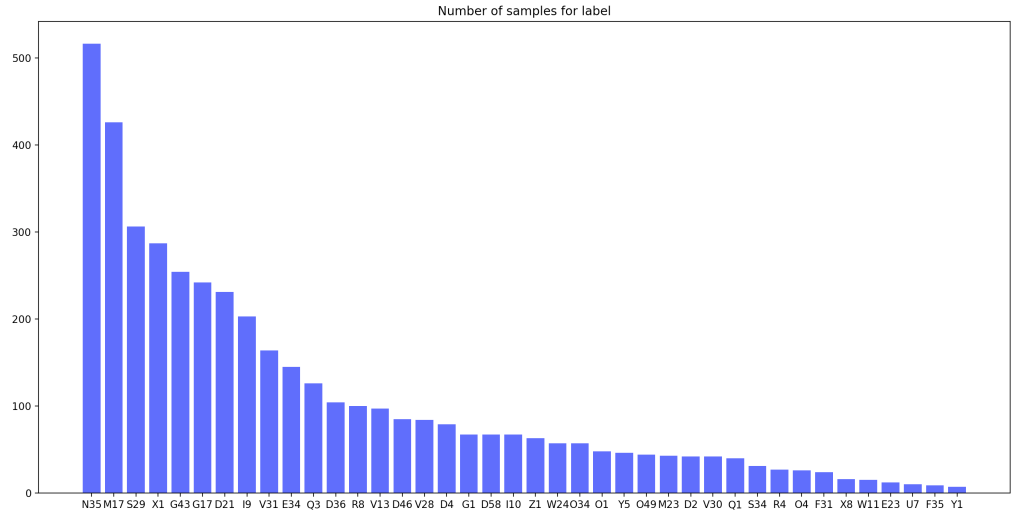



**Figure 3.2:** Number of images for each label

## 3.2 Preprocessing

In this section we will see the preprocessing techniques used to obtain a good starting dataset for the training of our neural network. The goal is to create a dataset containing images of the same size in grayscale. The size chosen for the images is 100x100. This choice is due to the fact that some models of neural networks do not adapt to input dimensions smaller than a certain value.

To achieve this goal the image must pass through the preprocessing pipeline, that consist in three different steps which are depicted in Figure3.3

However not all images will perform all steps. In fact, the images of $D_1$ are already in grayscale and do not require to perform denoize step.

We apply denoizing to images that have a marginal noise that differs from their background. As we can see from figure 3.4, some images of $D_2$ show this border noize .

To remove this noise, see Figure 3.5, a background image was generated following a distribution of a Gaussian field of mean $\mu$ and standard deviation $\sigma$ equal to the pixels intensity mode and standard deviation of the original image without considering the figure and the border.

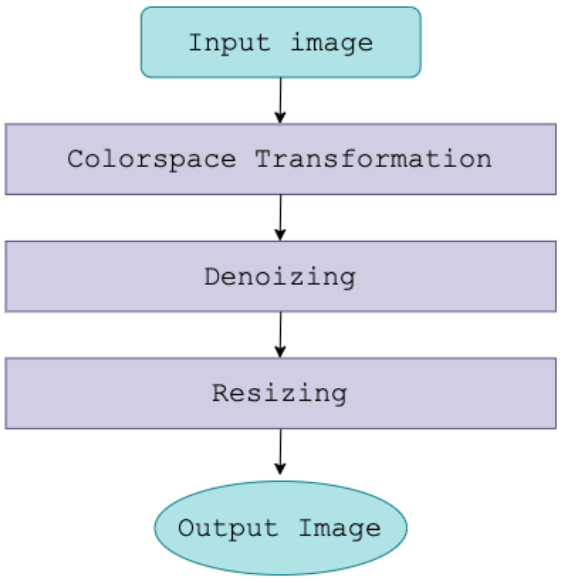



**Figure 3.3:** Flowchart of the Preprocessing Pipeline.

**Figure 3.4:** Some figures with border noise

The size of the background is chosen as the smallest square shape that can hold the original image. In this way the image will keep its aspect radio even after resizing without induce a loss in height/width information from the hieroglyph. After getting the output image, it will be resized to 100x100.

## 3.3 Train Validation and Test Sets

After having carried out the preprocessing of the images just described and having obtained a single dataset, it was necessary to divide the latter into three parts. Validation and test sets was immediately separated from the rest of the dataset, as we don't want it to be augmented. The only

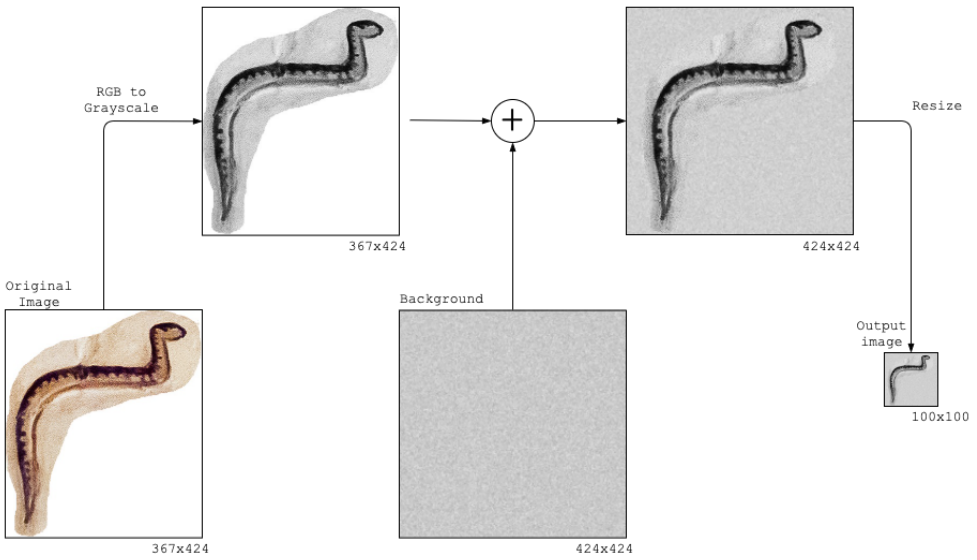



**Figure 3.5:** Example of image filled background and resize

augmentation technique that will be used on the test set is the horizontal flip, in order to verify that the network generalizes well. The size chosen for the test and validation set is 15% for each one of the entire dataset. The remaining part is the train set. It, unlike the latter, was subjected to all the data augmentation techniques that we will see later.

## 3.4 Data Augmentation

Augmentation is widely used in Deep Learning to improve the results. It increases the size of the training-set, and if it is unbalanced it can improve the balance. This results in a reducing of overfitting and helps to increase the performance of the CNN. Transformation like rotation and zooming, helps the CNN to be invariant toward rotation and different scales.

Augmentation transforms the image, so that a new image is produced. This could be reached with simple affine transformation like translation, zooming or rotation.

From this point to the end of the section we mean by dataset the entire dataset excluding the test set. First of all we used the augmentation technique to slightly rebalance the dataset. The dataset was increased

according to the quantity of images present for each label. The number of images that will be increased for each class is:

$$n_{aug}^{(C_i)} = \begin{cases} \frac{\sum_{i=0}^{N} |C_i|}{N} - |C_i| = \mu - |C_i| & \text{if } |C_i| < \mu \\ 0 & \text{otherwise} \end{cases}$$

where $n_{aug}^{(C_i)}$ is the number of images that will be increse for the class $C_i$, $|C_i|$ is number of images of class $C_i$ in $D$, and $|D|$ is the size of dataset $D$.

In Figure 3.6 we can see the augmentation on the train dataset.

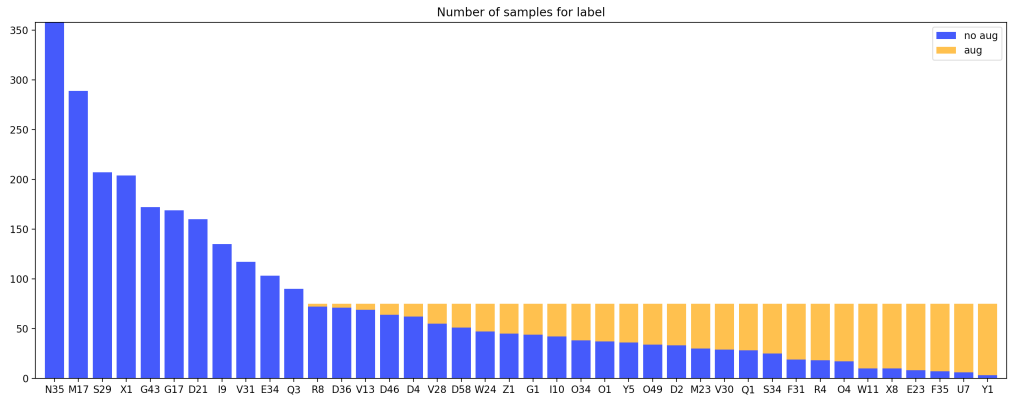



**Figure 3.6:** Data augmentation on the train dataset

We implemented different augmentation transformation, which can be independently used.

The following list describes the used augmentations:

**Translate**: The image is randomly translated to the $x$-axes and $y$-axes in an appropriate range.

**Zoom**: The image will be zoomed in in an appropriate range. It helps the Neural Network to be invariant for different scales of the hieroglyphs.

After applying this, all images in the dataset have been horizonal flipped. Make the network invariant to the horizonal flipped operation

is very important, as the ancient Egyptians allowed the hieroglyphs to be written in 3 directions, either from left to right, right to left and from top to bottom. For this reason it is possible to find the same hieroglyph facing both left and right.

# 4

# Image classification

In this chapter we will present the model architectures for the classification of Egyptian hieroglyphs, which are used in the next chapter for evaluation. At the beginning of the search for a good CNN for the tasks, we tested three existing models used for one of the big classification problems which was the ImageNet challenge. On this challenge, the three CNN have done very well. Analyzing the test results of these models for our problem helped us develop our model architecture. We have defined two requirements for our architecture with respect to these models:

- at least the same performance,
- it's faster.

We want to develop an architecture that has at least the same performance as the models used as tests and that is faster, as we had a limited resource of time and computing power for this thesis.

## 4.1 Model used

The models used as tests are ResNet-50, Inception-v3, and Xception. All three have achieved fabulous results on the ImageNet challenge.

### 4.1.1 ResNet50

ResNet, which was proposed in 2015 by researchers at Microsoft Research introduced a new architecture called Residual Network, in order to solve the problem of the vanishing/exploding gradient. In this network we use a technique called skip connections . The skip connection skips training from a few layers and connects directly to the output. The approach behind this network is instead of layers learn the underlying mapping, we allow network fit the residual mapping. So, instead of learning a direct mapping of $\mathbf{x} \rightarrow y$ with a function $H(\mathbf{x})$ (A few stacked non-linear layers), let us define the residual function using $F(\mathbf{x}) = H(\mathbf{x}) - \mathbf{x}$ , which can be reframed into $H(\mathbf{x}) = F(\mathbf{x}) + \mathbf{x}$, where $F(\mathbf{x})$ and $\mathbf{x}$ represents the stacked non-linear layers and the identity function(input=output) respectively. The author's hypothesis is that it is easy to optimize the residual mapping function $F(\mathbf{x})$ than to optimize the original, unreferenced mapping $H(\mathbf{x})$.

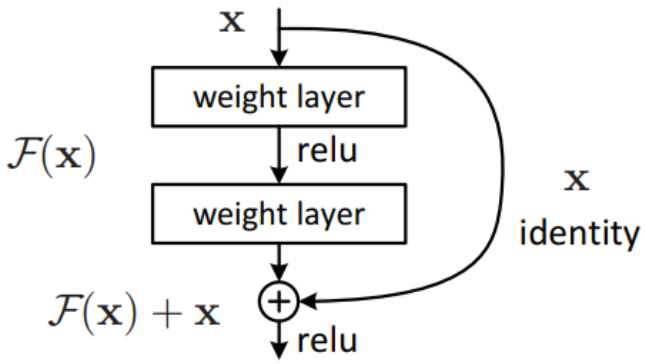



**Figure 4.1:** Residual block

The advantage of adding this type of skip connection is because if any layer hurt the performance of architecture then it will be skipped by regularization. So, this results in training very deep neural network without the problems caused by vanishing/exploding gradient. The authors of the paper experimented on 100-1000 layers on CIFAR-10 dataset.

The architecture of ResNet50, Figure 4.2, containing 50 layers. Stages 1-4 contains blocks of length 3, 4, 6 and 3 respectively, where each block

consists of three convolutional layers. The ResNet-50 has over 23 million trainable parameters.

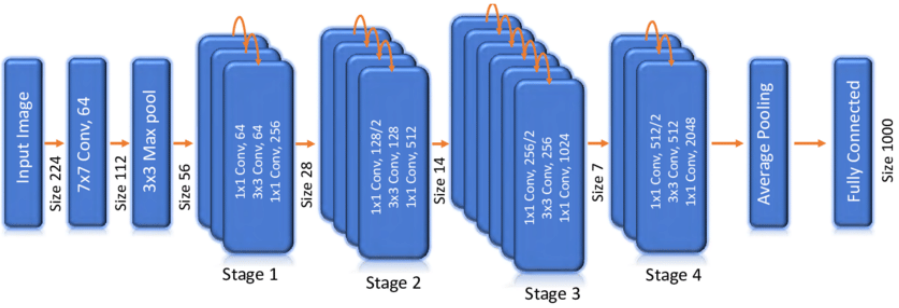



**Figure 4.2:** ResNet-50 architecture

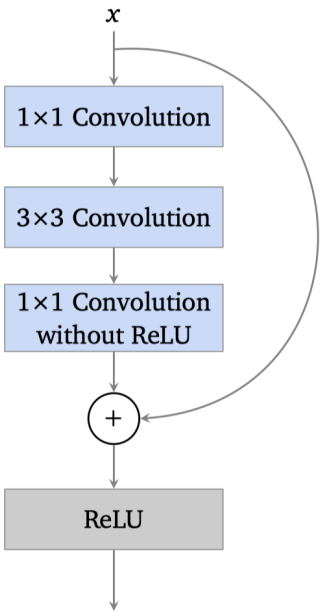



**Figure 4.3:** Example of a residual block in ResNet-50

For a detailed description of the network, we recommend the paper of Kaiming He et al. [3].

### 4.1.2 Inception-V3

The Inception-v3 model is the third version of the GoogleNet model. The network has a relative depth of 42 layers. All convolution layer uses the activation function ReLU, and apply the batch normalization.

The Inception-v3 starts with default convolution and pooling layers, to reduce the dimensional size of a feature map. Then it starts with the main characteristics of the network, the inception modules. The inception modules are parallel running convolution layers with different kernel sizes. One type of inception block is depicted in Figure 6.1, but there are two further types of inception blocks, which follows the same principles, but with other kernel sizes. The idea behind the inception module is, that it can extract similar features on different scales. The $1 \times 1$ kernel sizes has the purpose of the reduction of the number of feature maps, before is applied the expensive $3 \times 3$ convolution layer. For a detailed description of the network, we recommend the paper of Szegedy et al. [4].

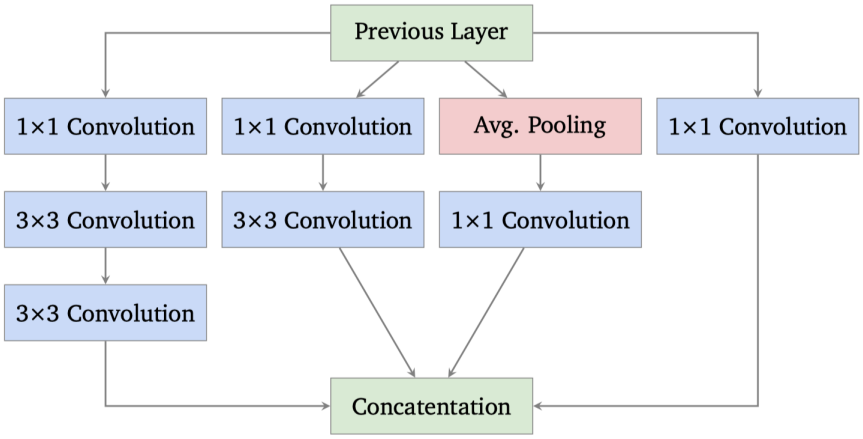



**Figure 4.4:** Example of an inception module in Inception-v3. The two successively 3×3 convolution layers have the same receptive field as a 5×5, but with the benefit, that they are faster to compute. The average pooling is executed with a stride of (1,1), so the pooling has no down sampling character. Furthermore, all layers applying padding, to produce the same spatial dimension of the feature maps, which are concatenated at the end of the inception module.

### 4.1.3 Xception

Xception, stands for Extreme version of Inception. In this new version a canonical Inception Block, Figure 4.5(a), is simplified with a extreme version of Inception Block, Figure 4.5(b).

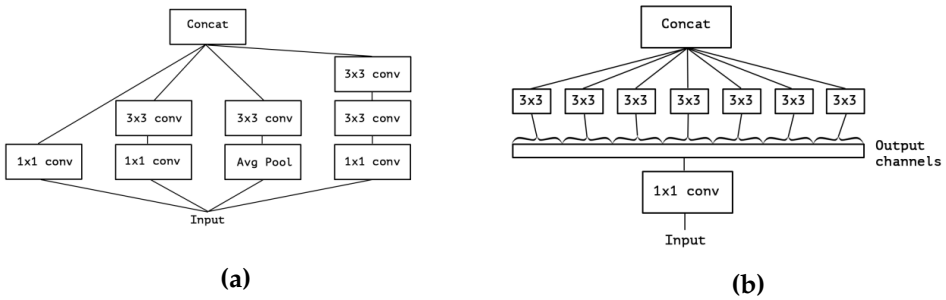



**Figure 4.5:** (a) A canonical Inception module (Inception V3). (b) An "extreme" version of our Inception module, with one spatial convolution per output channel of the 1x1 convolution.

This extreme form of an Inception module is almost identical to a depthwise separable convolution seen in section 2.19. The only difference is the order of the operations: depthwise separable convolutions as usually implemented (e.g. in TensorFlow) perform first channel-wise spatial convolution and then perform 1x1 convolution, whereas Inception performs the 1x1 convolution first. However, the authors argue that this difference is unimportant, in particular because these operations are meant to be used in a stacked setting. So the order of the operation can be changed.

Therefore this architecture relies on two main points :

- Depthwise Separable Convolution
- Skip connection between Convolution blocks as in ResNet

Both of these points have already been covered in section 2.19 and 4.1.1 respectively. In figure 4.6 we can see the Xception architecture. For a detailed description of the network, we recommend the paper of François Chollet [1].

## 4.2 Our Model

We had the goal, to develop a model, which has a similar performance, like the tested models seen previously. Furthermore, our model should be faster than these models.

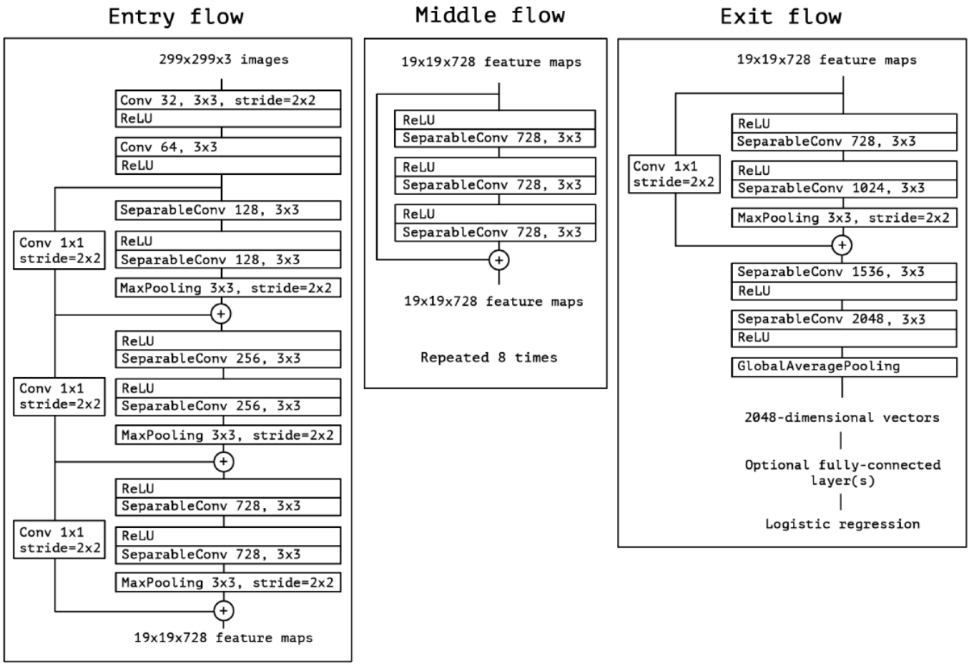



**Figure 4.6:** Xception architecture

Designing a CNN includes tuning several hyper-parameters. Hyperparameters can be divided into two types:

1. Hyperparameter that determines the network structure such as:
    - *Kernel Size* –the size of the filter.
    - *Stride* –the rate at which the kernel pass over the input image.
    - *Padding* –add layers of 0s to make sure the kernel pass over the edge of the image.
    - *Hidden layer* –layers between input and output layers.
    - *Activation functions* –allow the model to learn nonlinear prediction boundaries.
2. Hyperparameter that determines the network trained such as:
    - *Learning rate* –regulates on the update of the weight at the end of each batch.

- *Learning rate decay* -function to decrese the learning rate value every $x$ loop
- *Gradient Descend method* - optimization algorithm for finding a local minimum to training the net(see 2.2.3)
- *Regularizations* - methods to control the overfitting and improve the generalization error (see 2.2.5).
- *A number of epochs* –the iterations of the entire training dataset to the network during training.
- *Batch size* –the number of patterns shown to the network before the weights are updated.

In the next paragraphs we will illustrate the choices of these hyperparameters.

### 4.2.1 Architecture

For an effective way to reduce the computation time, we decided to use a default input size of 100×100 pixels. On this resolution, humans can easily recognize the details of the hieroglyphic, which is a good sign, that the resolution is not to tight for a recognition.

As basis for our model, there are the Separable Convolutional Layer 2.3.2, a feature present in Xception model, which reached the best performance on the task. The network consists of 6 blocks. The first is the input block and have two normal convolution layer with 32 and 64 number of filter respectively, a 3x3 kernel and a 2x2 and 1x1 stride. The last is the output block and have a separable convolution layer with 512 filter followed by fully connected layer and softmax. In the middle there are four block and each of these has two separable convolutional layers. The first two have 128 filters while the last two 256.
All convolution layer are followed by batch normalization. ReLu functions are used after batch normalization or after max pooling. The first normal convolution and max pooling layers are used to reduce the spatial dimension of the feature maps, in that they have 2x2 stride.

| Params in architecture | | |
|---|---|---|
| **Architecture** | **Total** | **Trainable** |
| Our | 480,008 | 475,720 |
| Resnet50 | 23,663,400 | 23,610,280 |
| InceptionV3 | 21,884,168 | 21,849,736 |
| Xception | 20,942,864 | 20,888,336 |

**Table 4.1:** Number of params for CNN architecture

Compared to previous architectures, this have much less trainable parameters (see Table 4.1). It only has 480,008, of which 475,720 trainable.

The architecture of the model is showed in Figure 4.7

### 4.2.2 The training

Our architecture was implemented and trained using Keras. Keras is an open-source software library that provides a Python interface for artificial neural networks. Keras acts as an interface for the TensorFlow library. TensorFlow is an end-to-end open source platform for machine learning. It can on numerous types of CPUs and even on GPUs, thanks to the support of languages such as CUDA or OpenCl. Keras abstracts the concepts of TensorFlow, so that it is easy to model a layered network. Furthermore, it provides the possibility, to train the defined model.

In order to training our model and obtain a good results it was necessary to choose some hyperparameters (see 4.2) and any techniques for learning in the best way. The loss function used to optimize the model during training is the categorical cross entropy. As optimization method is used ADAM(Adaptive moment estimation), a variant of the stochastic gradient descent (SGD), with a batch size of 32. We have used two regularization methods, to improve the generalization. The first is the L2-Loss to regularize the weights in the fully connected layer and the second is an adaption of the learning rate. The adaption of the learning rate decreases the learning rate by a factor of 2, every 15 epochs. The initial learning rate was chosen is 0.001. This value and the learning rate

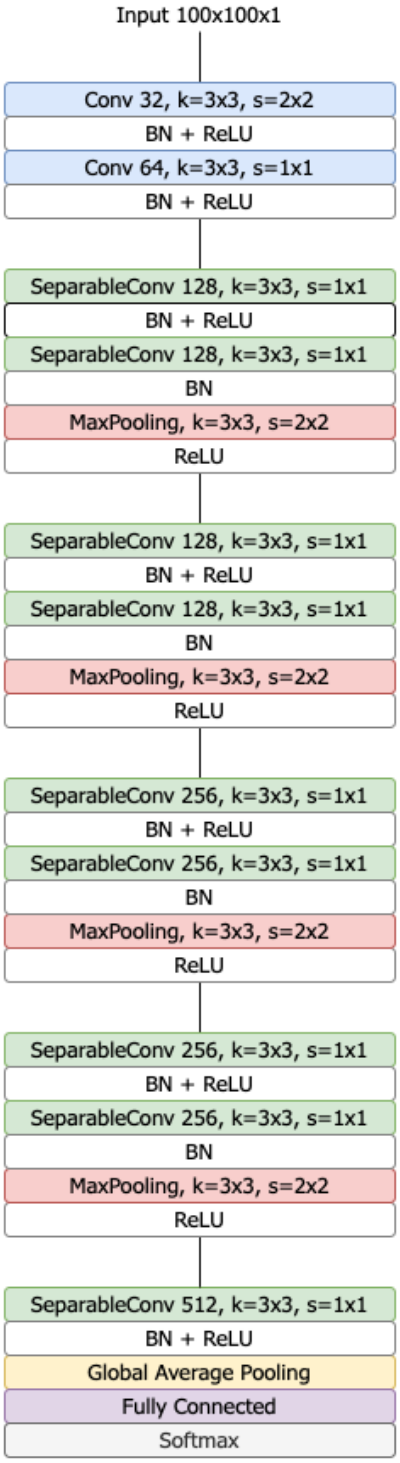



**Figure 4.7:** Our model architecture

decay parameter were chosen empirically, after having performed several test with different values.

### 4.2.3 Testing

The validation and test sets, which we described in 3.3, were used to test the trained model.

For every image and for every task, we get a vector for the prediction. This prediction contains the probability for every class, which describes the probability, that the model detects these classes. Therefore, the class with the maximal probability is used as the prediction for an image.

# 5

# Results

This chapter evaluates the performance of the our Net to predict the hieroglyphic class. It will be also compared to three convolutional neural networks , which was presented in the previously chapter. Computation times related to training and predictions will be evaluated and accuracy and loss trends will be shown during training. Additionally, a modified version of our architecture, which includes skip connections, will be tested. Finally, the results of the cross validation will be shown.

The evaluation is applied on the test-set. This results represent the performance on unseen data, and are suitable for the evaluation. Before we start with the performance evaluation, we introduce the evaluation metrics.

## 5.1 Evaluation Metrics

For the evaluation of the performance of the task will be used several metrics, which quantifies the quality of the predictions. For the evaluation is used as main metric the accuracy and F1-Score. The accuracy is used in the most related works, so it is useful for comparison and in addition, the value is easy to interpret. The F1-score is used to check for discrepancy in a single class. It would show a drop of the macro F1-score,

if one class is in another model really bad classified. To define these metrics it is necessary to introduce the confusion matrix, as they can all be derived from it.

### Confusion Matrix

A common way for evaluating the performance of a classification is too look at the confusion matrix. Let $n$ is a number of class, a *confusion matrix* $X$ is an $nxn$ matrix with the left axis showing the true class (as known in the test set) and the top axis showing the class assigned to an item with that true class. Each element $x_{ij}$ of the matrix is the number of items with true class $i$ that were classified as being in class $j$.

| | | Predicted Number | | | |
|---|---|---|---|---|---|
| | | Class 1 | Class 2 | … | Class ***n*** |
| Actual Number | Class 1 | $x_{11}$ | $x_{12}$ | … | $x_{1n}$ |
| | Class 2 | $x_{21}$ | $x_{22}$ | … | $x_{2n}$ |
| | . . . | . . . | . . . | . . . | . . . |
| | Class ***n*** | $x_{n1}$ | $x_{n2}$ | … | $x_{nn}$ |

**Figure 5.1:** Confusion Matrix

The *recall* for the class $i$ , $R_i$, is defined as the ratio of the number of the correct predicted class $i$ to the number of all samples for this class

$$R_i = \frac{x_{ii}}{\sum_j x_{ij}}$$

The *precision* for the class $i$, $P_i$, is defined as the ratio of the number of the correct predicted class $i$ to the number of predictions for this class

$$P_i = \frac{x_{ii}}{\sum_i x_{ij}}$$

To compute the macro recall ,*Recall*, and macro precision ,*Precision* , it will be calculated the average for $R_i$ and $P_i$ over all classes

$$Recall = \frac{\sum_i R_i}{n}$$

$$Precision = \frac{\sum_i P_i}{n}$$

This values will be need for the calculation of the F1-score.

**Macro F1-Score**

The macro F1-score is the harmonic mean of the macro recall and the macro precision.

$$F1score = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$$

It combines the two measurements to one score, and the score reaches its best value at 1 and the worst at 0. The combination of macro recall and macro precision makes the $F1score$ sensitive to outliers in single classes. This means, if a single or multiple classes are bad classified, then would this cause in a drop of the $F1score$, if the error rate stays the same. But normally, if the error rate increases, the $F1score$ decreases and the if the error rate decreases, the $F1score$ increases.

**Accuracy**

The accuracy is given by the sum of the elements placed on the diagonal of the confusion matrix divided by the number of samples $n$

$$accuracy = \frac{1}{n} \cdot \sum_i x_{ii}$$

The *accuracy* metric cannot represent misclassified classes, especially if the test dataset has unbalanced classes. For example, suppose we have a class $c_1$ with 90 samples and another $c_2$ with 10. If the accuracy shows that 90% of the data is rightly classified, at best this could mean that both

$c_1$ and $c_2$ have 90% of true prediction. But in the worst case, this could mean that all samples are labeled as $c_1$ and $c_2$ has 0% accuracy.

In our case, the test-set classes are not well distributed, as we can see from figure 5.2.

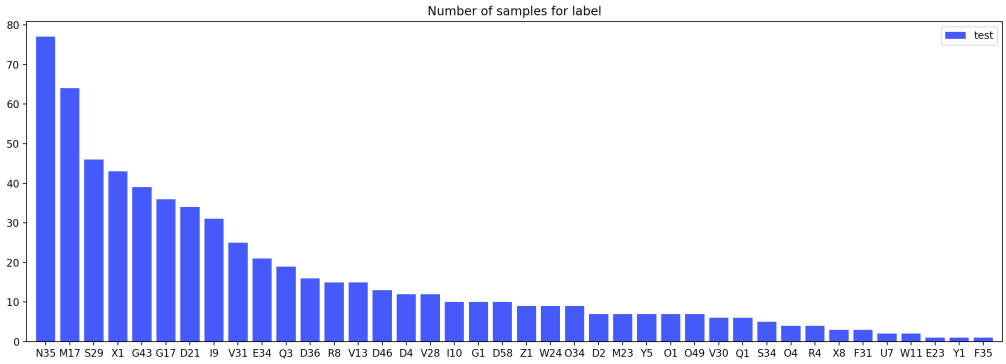



**Figure 5.2:** Test-set distibution

## 5.2 Evaluation Metrics

In this section, we report the results in order of precision, recall and F1-score previously mentioned for our architecture. In addition, the behaviors of the architectures in training and testing will be shown and analyzed.

Table 5.1 report the evaluation of the models.

| **CNN evaluation** | | | | |
|---|---|---|---|---|
| | **Metrics** | | | |
| **Architecture** | Accuracy | Precision | Recall | F1-Score |
| Our | **0.974** | **0.973** | **0.95** | **0.958** |
| Resnet50 | 0.945 | 0.919 | 0.905 | 0.903 |
| Xception | 0.964 | 0.955 | 0.944 | 0.946 |
| InceptionV3 | 0.956 | 0.933 | 0.930 | 0.925 |

**Table 5.1:** Evaluation of the architectures based on the defined metrics.

We can see how our model achieves better performance in reference to all evaluation metrics to other architectures. In Figures 5.3, 5.4, 5.5, 5.6, we can see the trend in the training and testing progress over the epochs of the various models.

In Figures 5.7 is shown accuracy trends on the test-set with the various architectures.

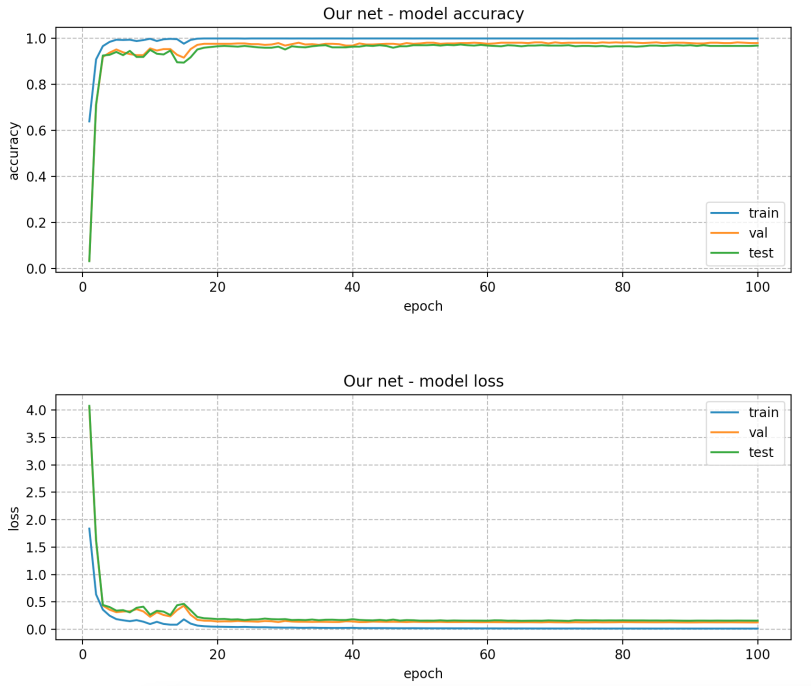



**Figure 5.3:** Accuracy and loss trend in training, validation and testing using Our architecture

## 5.3 Computation Time

In the previous chapter, we mentioned that the our net should be faster to train than the other three architecture. To compare the training time of our network with the InceptionV3, ResNet50 and Xception, the training time for an epoch on different input resolutions was measured. For training, the model takes 8,358 images for each epoch, while 647 images for validation. Times are measured on an Nvidia Tesla T4 and are shown in Figure 5.8. The our net is among all the tested CNN it is the fastest with 100x100 and 200x200 image resolution. As the resolution

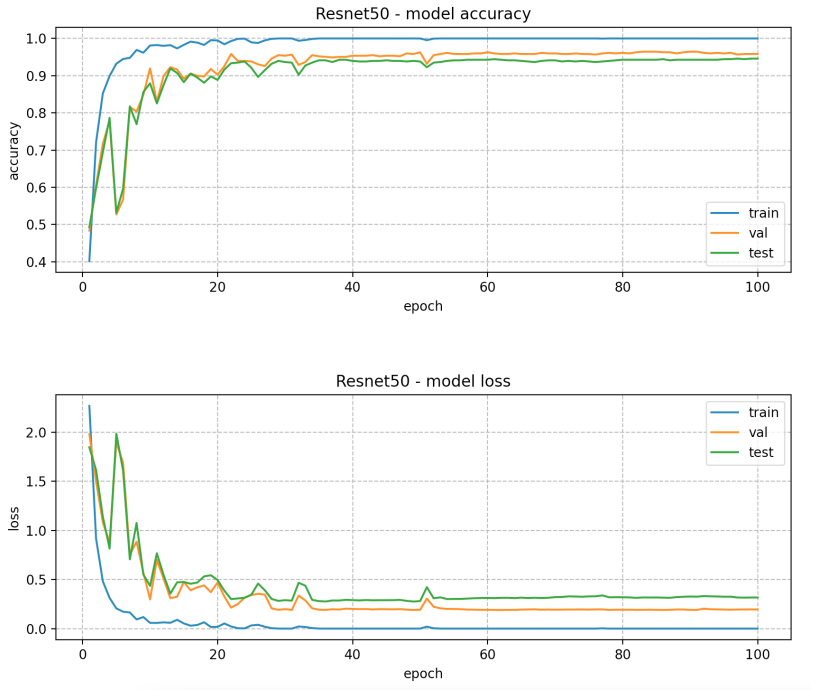



**Figure 5.4:** Accuracy and loss trend in training, validation and testing using Resnet50 architecture

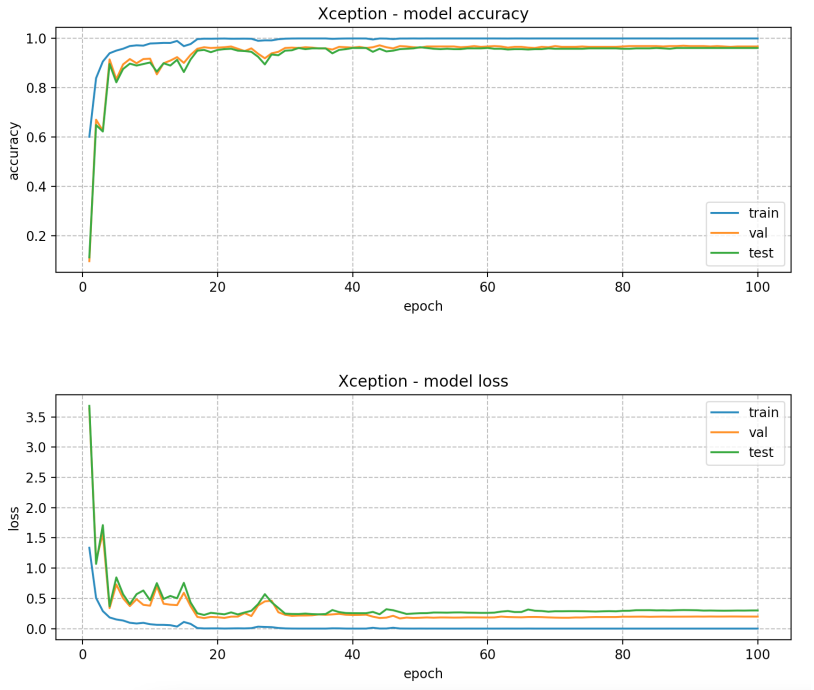



**Figure 5.5:** Accuracy and loss trend in training, validation and testing using Xception architecture

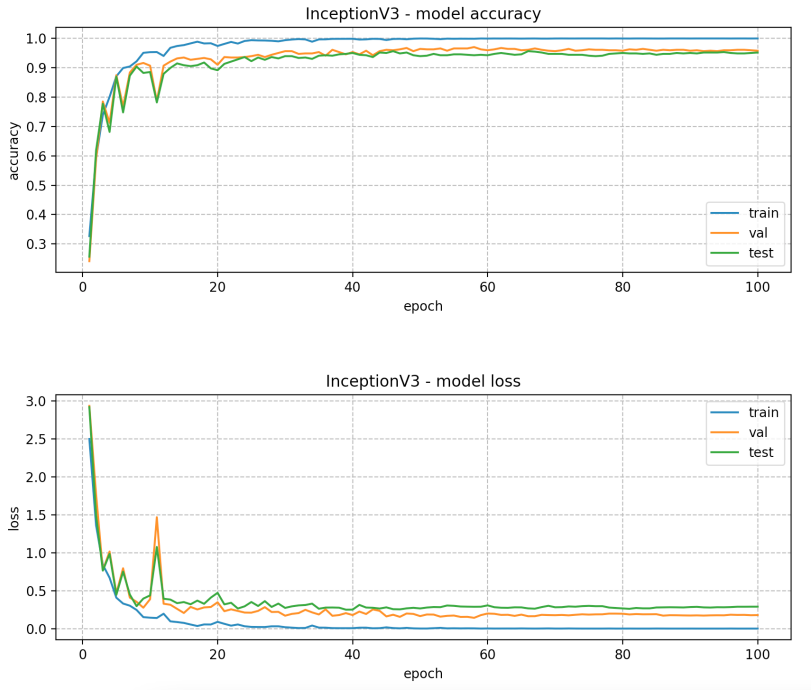



**Figure 5.6:** Accuracy and loss trend in training, validation and testing using InceptionV3 architecture

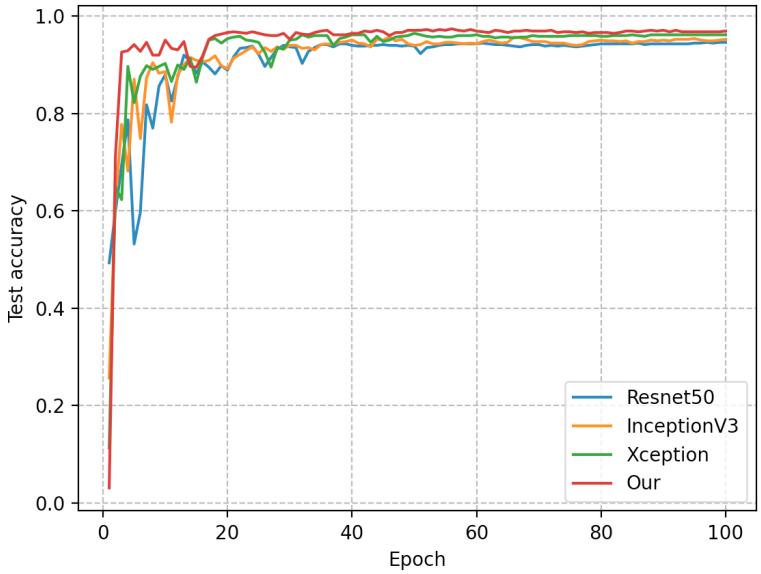



**Figure 5.7:** Accuracy progress on the test-set with the various architectures

increases the computation time for the our net increases more than InceptionV3. This is due to the high number of features maps generated by our network. However, we must point out that this network have been designed to work with larger images. Infact, the default input image size for InceptionV3 is 299x299 and does not support images smaller than 71. However for all the different resolutions our network is much faster than Xception which is the one of the three that offers the best performance on the task. The training time increases exponentially. This has a lot of influence on the CNN, because the feature maps are all increased by a factor of four.

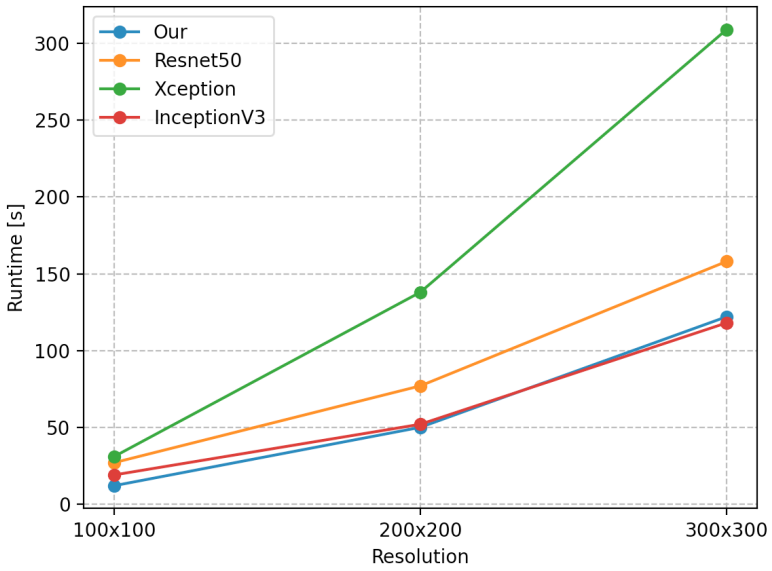



**Figure 5.8:** Training computation time

In addition to the computation time related to the training, the time related to the prediction of the images was also calculated with batch-size 1 and 32 on the same GPU. We collected 100 measuring points and averaged them. The results are depicted in the Table 5.2. The our net is among all the tested CNN it is the fastest to image prediction for any resolution.

| | | Prediction runtime [ms] | | |
|---|---|---|---|---|
| | | Resolution | | |
| | Architecture | 100x100 | 200x200 | 300x300 |
| Batch-size 1 | Our | **4** | **5** | **5** |
| | Resnet50 | 8 | 10 | 13 |
| | Xception | 7 | 10 | 11 |
| | InceptionV3 | 12 | 13 | 14 |
| Batch-size 32 | Our | **10** | **30** | **63** |
| | Resnet50 | 29 | 86 | 181 |
| | Xception | 27 | 110 | 275 |
| | InceptionV3 | 20 | 58 | 116 |

**Table 5.2:** Prediction runtime for batch-size 1 and 32 on Nvidia Tesla P100-PCIE-16 GPU.

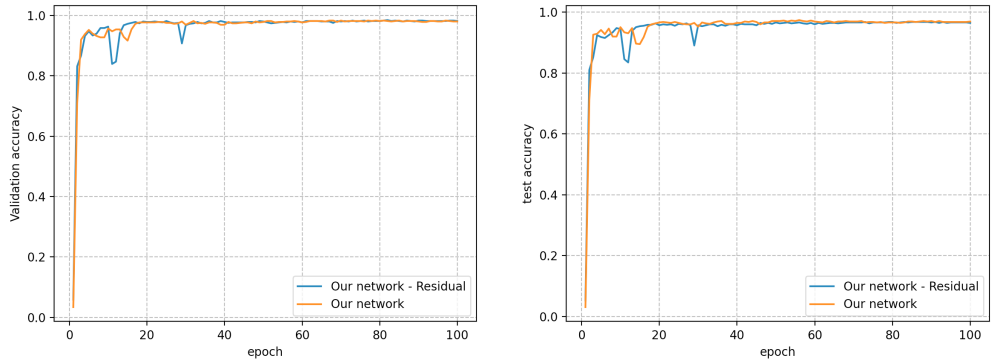
## 5.4 Skip Connection

Our model unlike Resnet50, InceptionV3 and Xception does not use residual connections. In order to evaluate whether the residual connections could have any benefits, we compared a modified version of our model that includes them on the same dataset. The results are shown in figure 5.9.

Residual connections are clearly not essential in terms of either speed or final grading performance. Figure 5.10 shows the model with the residual connections. This modified version increases both the parameters and the computation times as convolutional layers are added in the residual blocks.

## 5.5 K-Fold Cross-Validation

A further method for evaluating and estimating the performance of the network is the Cross-Validation. Cross-validation is a resampling procedure used to evaluate machine learning models on a limited data sample.




(a) (b)

**Figure 5.9:** Accuracy progress on validation and test sets using Our model with and without residual connections

The procedure has a single parameter called *k* that refers to the number of groups that a given data sample is to be split into. As such, the procedure is often called *k*-fold cross-validation. When a specific value for *k* is chosen, such as k=10 becoming 10-fold cross-validation.

Cross-validation is primarily used in applied machine learning to estimate the skill of a machine learning model on unseen data. That is, to use a limited sample in order to estimate how the model is expected to perform in general when used to make predictions on data not used during the training of the model.

It is a popular method because it generally results in a less biased or less optimistic estimate of the model skill than other methods, such as a simple train/test split. So, cross-validation is used to reduce the variance in the validation error estimates without having to set aside a lot of data for validation.

The general procedure is as follows:

1. Shuffle the dataset randomly.
2. Split the dataset into *k* groups
3. For each unique group:
    - Take the group as a hold out or test data set
    - Take the remaining groups as a training data set

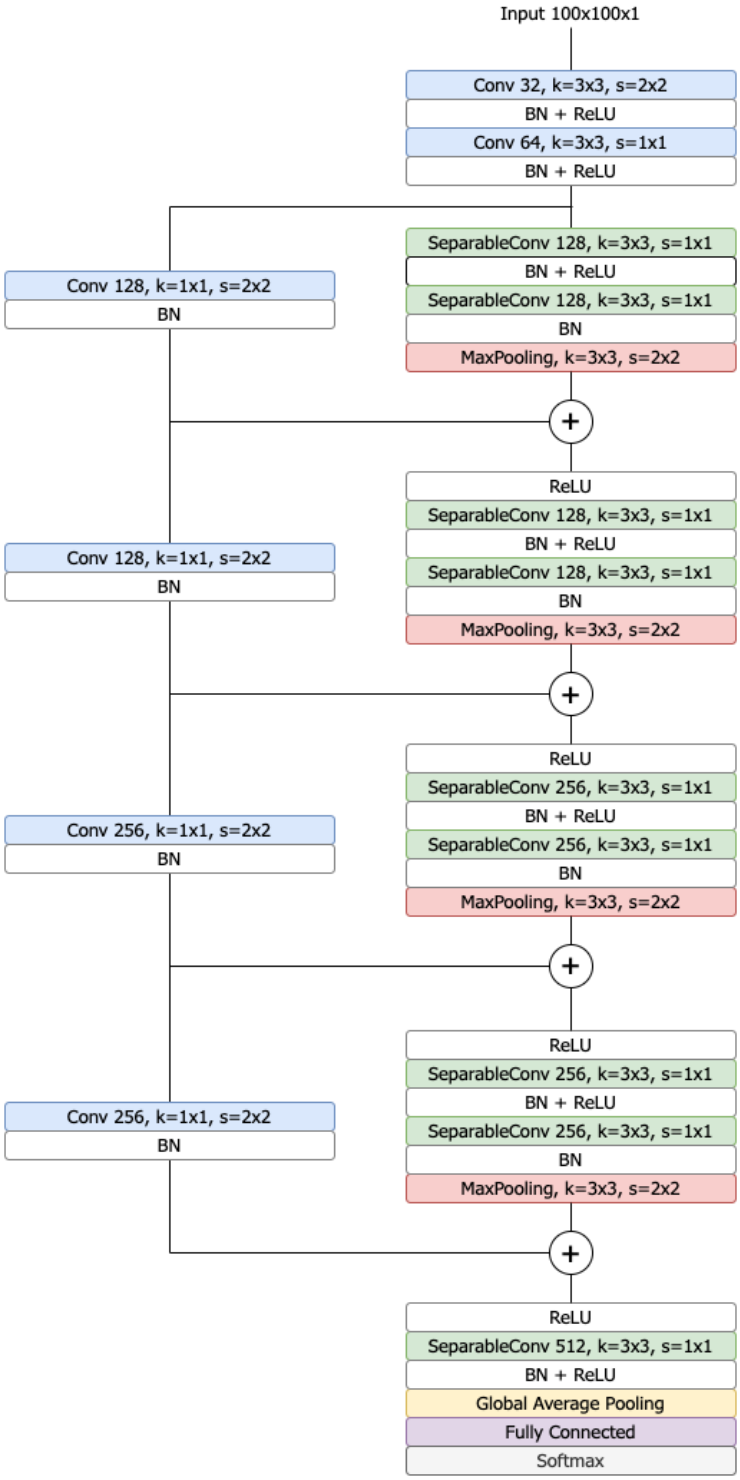



**Figure 5.10:** Our model with residual connections

- Fit a model on the training set and evaluate it on the test set
- Retain the evaluation score and discard the model

4. Summarize the skill of the model using the sample of model evaluation scores

Importantly, each observation in the data sample is assigned to an individual group and stays in that group for the duration of the procedure. This means that each sample is given the opportunity to be used in the hold out set 1 time and used to train the model $k - 1$ times.

Evaluating models on this split would not give them enough examples to learn from, too many to be evaluated on, and likely give poor performance. You can imagine how the situation could be worse with an even more severe random spit. The solution is to not split the data randomly when using k-fold cross-validation or a train-test split. Specifically, we can split a dataset randomly, although in such a way that maintains the same class distribution in each subset. This is called stratification or stratified sampling and the target variable ($y$), the class, is used to control the sampling process.

In this thesis, k-fold cross validation stratified was used, with $k = 5$ (see Figure 5.11). For each iteration the train-set was augmented through previous mentioned techniques, as zoom in, translation and horizontal flip. The validation set, instead it has not been altered.

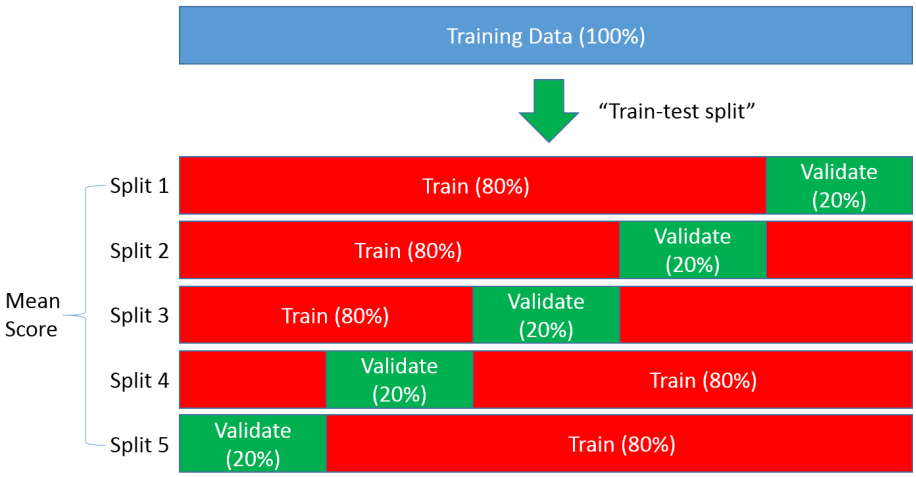



**Figure 5.11:** 5-Fold Cross Validation

The results of the cross validation are shown in Figure 5.12.

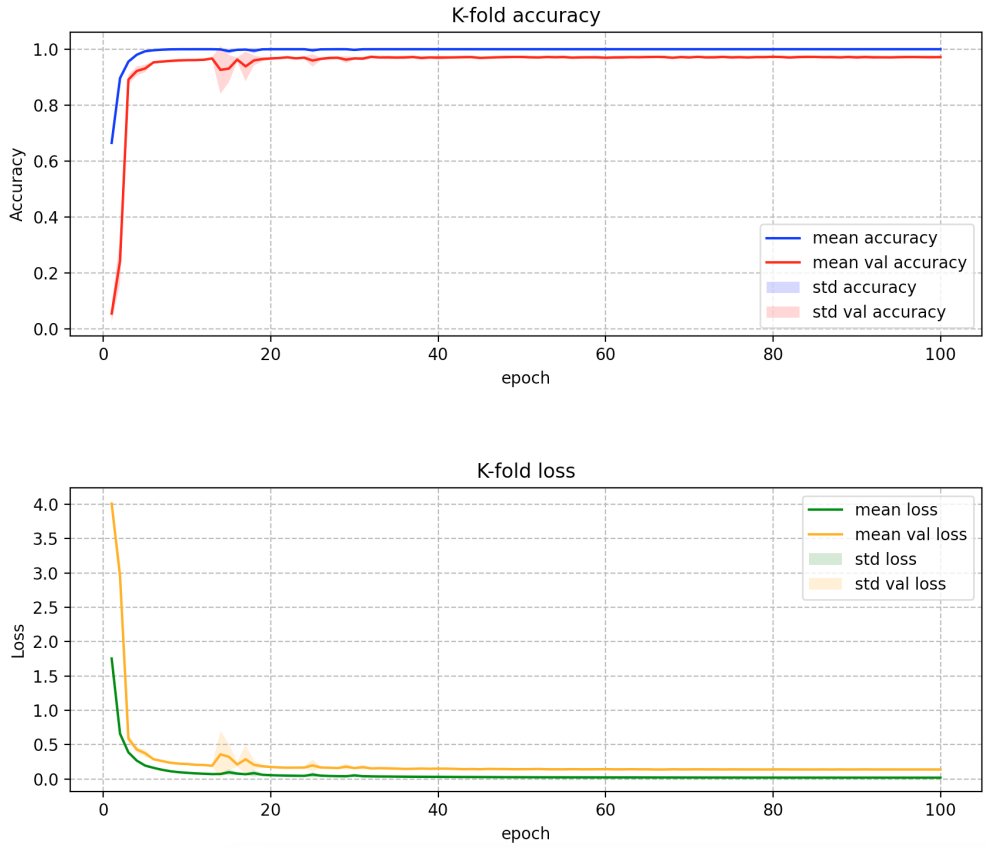
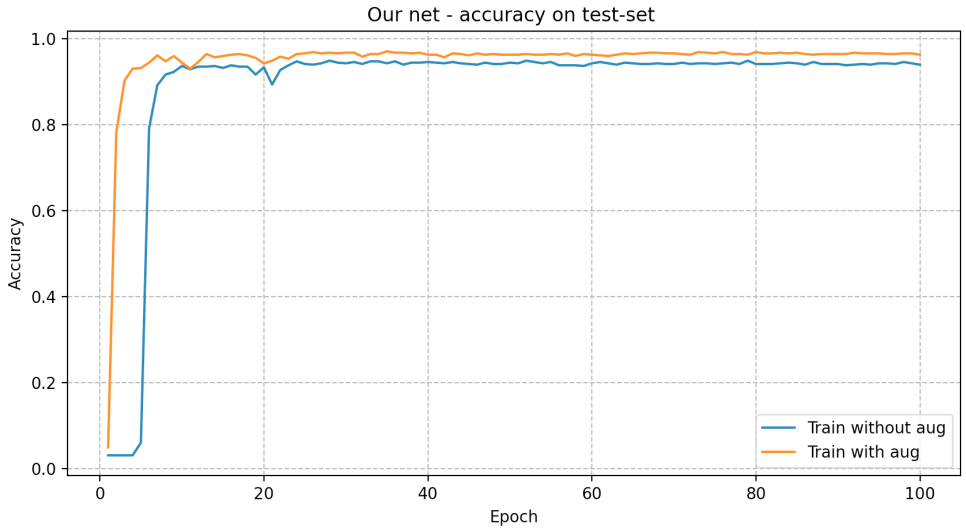


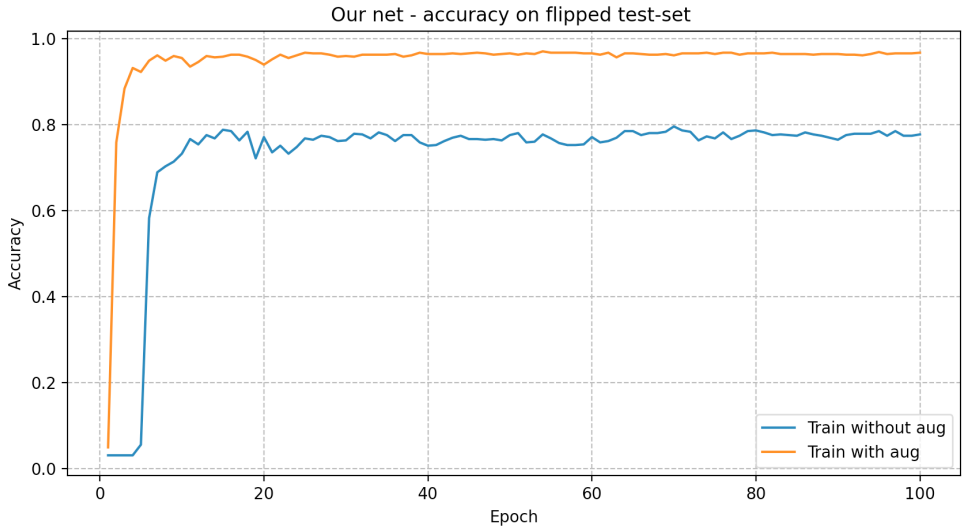

**Figure 5.12:** Cross Validation

## 5.6 Image augmentation

In figure 5.13 it is possible to see the effects of the data augmentation on the train-set. Figure 5.13(a) shows the accuracy trend on the test-set. The graph in orange is inherent to the network trained with the train-set to which data augmentation has been applied, while the one in blue is inherent to the network trained with the normal train-set. As we can see in the first case the accuracy values are better and therefore the effect of the augmentation is remarkable. The effects are even more interesting when we test the model whit the totally horizontally flipped test-test (Figure 5.13(b)). In this case, in fact, with the use of augmentation on the train-set, there is a clear improvement in accuracy. It improves from a maximum of 0.79 to a maximum of 0.97.




**(a)** The orange graph shows the accuracy on the test-set having trained the network with the augmented train-set. The graph in blue shows the accuracy on the test-set having trained the network with the normal test-set.




**(b)** The orange graph shows the accuracy on the flipped test-set having trained the network with the augmented train-set. The graph in blue shows the accuracy on the flipped test-set having trained the network with the normal train-set.

**Figure 5.13:** Effects of augmentation on the test test.

# 6

# Conclusion

# Bibliography

[1] F. Chollet. *Xception: Deep Learning with Depthwise Separable Convolutions*. 2017. arXiv: 1610.02357 [cs.CV].

[2] A. Gardiner. *Egyptian Grammar*. Griffith Institute, 1957. ISBN: 0900416351.

[3] K. He, X. Zhang, S. Ren, and J. Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[4] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. *Rethinking the Inception Architecture for Computer Vision*. 2015. arXiv: 1512.00567 [cs.CV].