DEPARTMENT OF AUTOMATIC CONTROL
UNIVERSITY OF BELGRADE FACULTY OF MECHANICAL ENGINEERING

Convolutional Neural Networks

Mentor: PhD Radiša Ž. Jovanović

Student: Lara Laban 1097/17

Master's thesis, September 2019

1. MACHINE LEARNING

1.1. WHAT IS MACHINE LEARNING AND TYPES OF MACHINE LEARNING

Machine learning is a form of science and a modeling technique that involves data. Moreover, it actually determines the model based on the input data. Here when we utilize the word data, we essentially mean documents, audio files, images, or any useful type of information. Machine learning is nothing more than a part of the branch of artificial intelligence, thus the topic of this paper is deep learning, which is actually a type of machine learning. Nevertheless, in order to explain the concept of deep learning we must first hark back to the basics of machine learning and its problematics, seeing that it verily laid out all of the fundamental concepts.

We can take the spam mail from our e-mail as an example and explain why machine learning is more convenient in a more understandable way. Let's assume we want to clear our spam mail from our e-mail by using a simple code. First we analyze the problem, then we come to the conclusion that some of the words are repeated often in a part of the name of the e-mail, or perhaps that the sender of the e-mail appears more than once. In that case, we should create such an algorithm that can recognize all of the repetitive patterns, and then mark the e-mails that have them. Notwithstanding, this problem is not trivial and there are a lot of rules that we would have to introduce. Unlike ordinary code, machine learning code is able to learn words automatically, ie. repetitive expressions, which may appear later on and are not included in the original algorithm, are good predictors for some novel words. Machine learning code is able to learn certain expressions on its own and to eliminate them, as well.

Different types of problems, where this sort of code is useful, are complex problems for which there are yet no specific algorithms. Furthermore, by diving into an example that was recently employed by NASA, where they managed to photograph a black hole after years of research, we can explain such a phenomenon. Now, let's stop and ask ourselves how is this actually possible? As we have come to know them, black holes are made up of a large amount of matter that had been accumulated in a very small area; mostly they are formed by the remnants of the red giant that disappears in a supernovae explosion. As a consequence, black holes have such strong gravitational fields that not even light can escape them. In addition, it is also quite clear that in order to see details in objects that are very far away from Earth, a couple of parsecs or even further, we need to collect as much light as possible in a very high resolution. Since such a large telescope is impossible to build at the moment, they came up with the idea to assemble a network of multiple telescopes — Event Horizon Telescopes (EHT). EHTs have an aim that is essentially to capture a black hole image, all the while enhancing a technique that allows recording of remote objects, known as very long baseline interferometry, or VLBI. This technique is used to track spacecrafts and to record distant space quasi-stellar radio sources, such as quasars. However, what has not been mentioned is that none of this would have been possible if deep learning had not been applied to collect images and information provided by telescopes. In later chapters it will be explained exactly how this deep learning technique is used for convolutional neural networks to recognize images, train the network, and remember similarities. For now, we just need to understand that with the aid of a data set, which in this case was composed of various

images of stars, galaxies, as well as images taken directly by EHT telescopes, they managed to compile a code that not only recognized similar images, but succeeded in composing the first image of a black hole. Howbeit, we will not be dealing with such convoluted problems here, alas we will first try to elucidate all of the techniques and ideas behind image recognition within deep learning.

We may distinguish the ensuing types of machine learning:

1. Depending on whether or not they were trained by a supervised network (supervised, unsupervised, semi-supervised and reinforcement learning)
2. Depending on whether they can learn gradually during training (on-line learning and batch and mini-batch learning)
3. Do they work on the principle of comparing new data with already known data, or do they find patterns during network training and build a model based on that (instance-based learning and model-based learning)

These criteria are not mutually excluding, and nonetheless can be combined while writing code.

Supervised learning — the data with which we supply our algorithm includes the desired solutions that are termed labels. A typical task for supervised learning is classification. The previously mentioned spam filter used on e-mail is a good example of supervised learning, for the reason that it is trained with diverse examples of e-mails and learns how to classify latest spam emails based on this data. Another typical task is to predict the targeted numerical value, such as the price of cars, given the set of functions (mileage, age, brand,...) that is made up of these predictors. This task is called regression. Logistic regression is often used for the classification problem, which is a precursor to the problem of convolutional neural networks, however we will take a step back at the moment and come about this topic later once again.

Unsupervised learning is a learning technique that utilizes data which does not have the desired solutions, and here our system tries to train without a teacher. An illustrious example of this is the social network YouTube. A part of the algorithm that trains and provides the output data on how many people, what gender, age, and nationality visit the YouTube channel is definitely done in this manner. All the while without assistance and instruction, the goal of unsupervised learning is to provide us with this data and if in addition a hierarchical clustering algorithm is used we can divide each group into smaller groups, and this abets, for example, when we want to attract a certain group of people. These algorithms try to preserve as many structures as they can to separate the clusters in the input space from the overlap in the visualization, so that we can understand how the data is organized and identify unexpected patterns.

A subset of this learning is learning the rules of association, in which the objective is to dig up large amounts of data that reveal interesting relationships between attributes. For example, when we use the social network Facebook all the time, the algorithm collects things we like and based on that throws out various advertisements. Also, a subgroup of this teaching is feature extraction. Here we have a sole purpose to reduce dimensionality and simplify data without losing a lot of information. For example, the mileage that a car

had traveled is related to age, and by combining these two inputs we have one characteristic that roughly describes the wear and tear of that car. A further discussion about this type of learning will be revised in more detail in Chapter 8.

Semi-supervised learning — some algorithms may deal with partially labeled training data, in that case there is frequently a lot of unmarked data and a slight amount of labeled data. This type of learning is clearly recognizable on the Internet; for example when based on one photo of a person Google photos can easily recognize the said person each and every time.

Reinforcement learning — has an agent who can observe the environment, choose and perform actions and in return receive rewards (either positive or negative). The agent has to figure out for himself what the best tactics are over time. An example of such learning is the AlphaGo program, which had to use deep learning to beat a person, since GO is a notorious Chinese game that devises over 10^{172} moves to choose from. This type of learning was able to learn based on a million games played beforehand and hence for the first time beat a man in 2016.

Batch learning — here the discussion revolves around a system that is not able to learn gradually, hence it must be trained using all available data. This process takes a lot of time as well as a lot of computer resources, as we first train the system, and then the neural network is launched and it applies what it has learned so far. If we have brand-new data, we will have to stop the network and train it again and only then replace the old system with a new one. In conclusion this requires a lot of effort, a lot of space on the CPU unit, and limited resources. This type of learning is also called offline learning.

Online learning — system i.e. the neural network is trained gradually with successive feeding of data, individually or in small groups (mini-batches). This is precisely the tactic used in convolutional neural networks. Each learning step is fast and cheap, so that the system can learn new data during training.

1.1.1. PROBLEMS AND CHALLENGES OF MACHINE LEARNING

In this chapter, the drawbacks of machine learning will be provided from the theoretical point of view, and in Chapter 4 we will explain this in more minutiae with apt examples. Incipiently, it should be emphasized that it is crucial for machine learning to obtain independent training data that adequately reflects the characteristics of the model. The process which is utilized to make model performance consistent regardless of training data or input data is known as the generalization process. The attainment of machine learning lies precisely in how successfully the generalization process is carried out. It can be said that we distinguish two focal problems, which are overfitting and underfitting of the neural network, subsequently.

Underfitting — simply put it concerns the neural network that either does not have an adequate number of input data with which it can maneuver i.e., train the neural network, or a network that has an insufficient number of layers.

Overfitting — let's say that we do not know a random person and in addition to that based on two or three things we observed at that instant, we copy those (two or three) characteristics to people who look similar at first glance. Seams inaccurate, right. Our brains often do this, and computers have the same generalization problem. If we give our neural network too much data, as well as too much information, as a consequence we actually do not give it enough space to make a decision on its own, an unbiased one; hence the network entirely follows the patterns we have outlined for it.

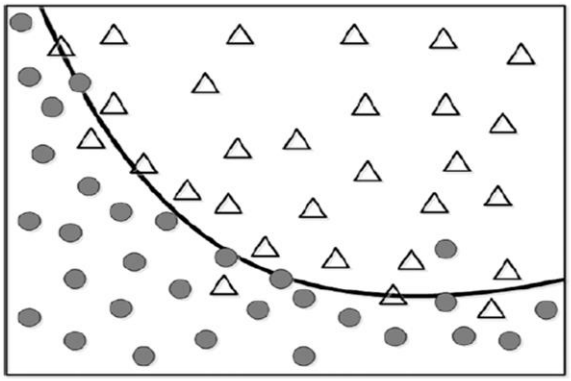


Figure 1.1.1.- 1 Classification problem

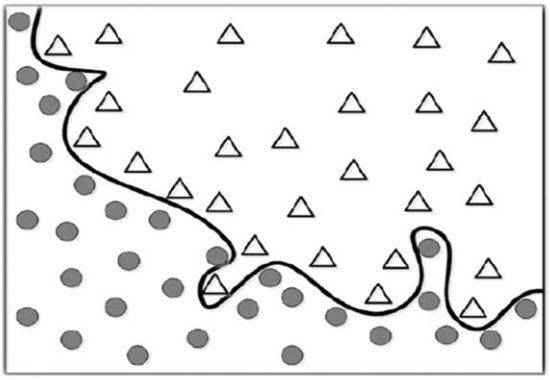


Figure 1.1.1.- 2 Classification with a complex curve

This essentially denotes that our neural network will perform well for the data on which we performed the training, however, if new data is brought to the input, an error will occur, i.e. poor classification. Considering the classification problem depicted in Figure 1.1.1.- 1 we must divide the position data into two groups. The points and triangles in the figure are the input data that we need to train, and the objective is to determine the curve that sets the boundary between these two sets of training data. The proposed curve appears to be a good enough boundary, although there are some overlaps. Let's say we set a complex curve now and thus achieve a perfect grouping of data, as we can observe in Figure 1.1.1.- 2. Unfortunately, we may derive to the conclusion that this is still not sufficiently accurate.

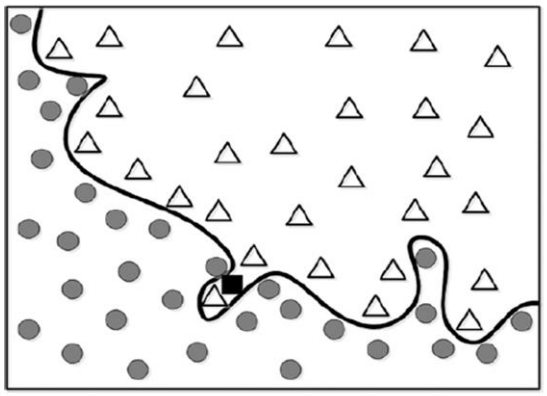


Figure 1.1.1.- 3 The new input data ■

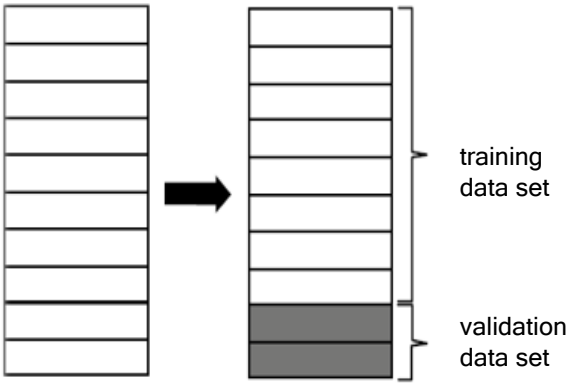


Figure 1.1.1.- 4 Division into training data set and validation data set

Let us undertake the implementation of this model in the real world, the brand-new input is indicated on Figure 1.1.1.- 3 using the symbol ■. This error-free model identifies new data as a member of class Δ. Nonetheless, we know that it could be observed from both

sides of the curve and that grouping of the novel data with the class • might be much more reasonable. A model with this kind of accuracy has led to the data set itself containing a lot of noise. So, if machine learning takes into account all of the data, even the noise, in the end an incorrect model (curve in this case) is produced, which cannot be used later on for a new set of input data.

One of the ways to solve this problem, which we will employ later in our code, is testing, i.e. a makeshift data set for testing / validation. Testing consists of the following steps and is illustrated in Figure 1.1.1.- 4. Initially, we divide the training data into two groups, one for training and the other for testing. According to the "rule of thumb", the ratio of the data set for training and testing of the neural network is $8:2$. We will then train the model with a training data set, and evaluate the model performance using the validation data set. If we are pleased with the results, the training of the model is completed, else if we are not, we modify the model and restart the whole procedure.

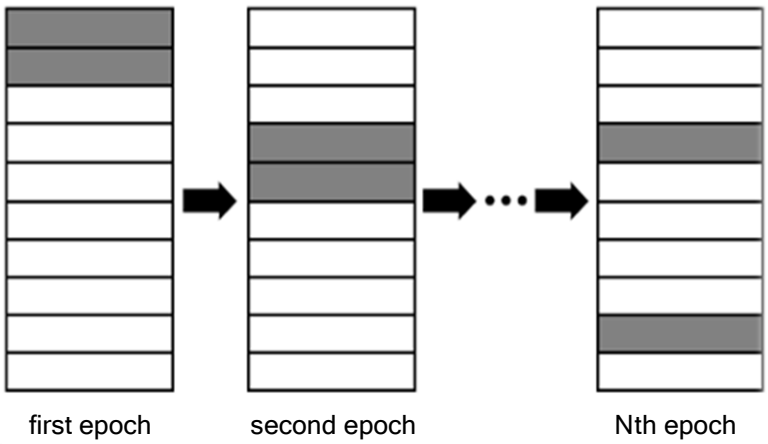


Figure 1.1.1.- 5 Cross validation

Cross validation is a slight variation of the validation process. It is demonstrated in Figure 1.1.1.- 5 and implies that after the first epoch our set of input data is again divided into a training and validation data set, and that they then adjust like this after each subsequent epoch.

1.1.2. MATLAB OR PYTHON

In this paper, we will go through examples and codes that are written entirely in the python programming language. However, before we continue, we will first explain why and how we came to the solution that this programming language is optimal for constructing our neural network. The factual truth is that just ten years earlier there would have been absolutely no difference whether we used matlab or python to form our code. Moreover, it is considered to be of an immense importance to first understand and mathematically write on a piece of paper, for example the back-propagation algorithm / BP algorithm, and then implement it into matlab, which is a downright mathematical program, so that we can further understand its implementation within python. This is perhaps the easiest way, a path that allows us to understand the fundamental concept of neural networks. However, the main difference between python and matlab is that python contains libraries that have the ability to automatically implement some parameters that can succor us and significantly accelerate our work (even though matlab poses a few of its own libraries that

can increase the code, it is notwithstanding that python has incomparably more possibilities). It is these libraries that python has that are of momentous importance when, for example, we work with a set of data that contains a million images, or an incredibly large number of audio or video recordings. Here are some of the most important libraries that aid in our paper:

$TensorFlow$ — is a powerful open source library, i.e. numerical computing software, especially good for fine-tuning of large-scale machine learning. Its basic principle is simple — first the number and method of computations to be performed are defined in python, and then $TensorFlow$ takes that graph and executes it efficiently using optimized C++ code. Most importantly, it is possible to break a graph into several parts and run them in parallel transversely via multiple CPUs or GPUs. $TensorFlow$ can train a neural network with a million parameters, on a training data set consisting of a billion copies with a million functions, each individually. $TensorFlow$ was developed by the Google Brain team and it is used for the purposes of all the most famous Google search engines Google Cloud Speech, Google Photos, as well as Google Search. When $TensorFlow$ became available in November 2015, there were already many popular libraries. In spite of this, its performance as well as compatibility with other libraries is what set it apart. $TensorFlow$ works on Windows, Linux, macOS, and android devices. It is compatible with the $Scikit - Learn$ library, which can be used to train different types of neural networks in just a few lines of code, and is compatible with the $Keras$ library into the bargain.

Simply put, the $Keras$ library deals with the implementation of the algorithm with backpropagation, i.e. the BP algorithm which we will explain later in another chapter. $Keras$ makes it much easier to implement the code on the training data set.

$OpenCV$ is written in the C / C++ programming language, but the connection to python is provided when launching the installation. $OpenCV$ is the standard when it comes to image processing, and in this paper it is used to load images from the disk, further display them on the screen and perform basic image processing operations.

$Scikit - Learn$ is an open source python library utilized for machine and deep learning. It is used for cross validation and visualization — this library complements the $Keras$ library very meticulously and especially assists when it comes to separating data sets for training / testing / validation, as well as in validating our deep learning models.

1.2. FUNDAMENTALS OF IMAGES AND NUMPY

Each image consists of a series of pixels, and they withal represent its basic elements. A pixel is considered to be the "color" or "intensity" of light that appears in a particular place in an image (there is no finer granularity than a pixel). Moreover, if we survey the image as a grid, each square contains one pixel. Let us take as an example an image with the following resolution of 1000×950, this means that a 1000 pixels is the width of our image, and 950 pixels is the height of the image. Given that the image can be viewed as a multidimensional matrix, we conclude that it has 1000 columns and 950 rows. Subsequently we can arrive to the conclusion that the image is composed of a total of $1000 \times 950 = 950000$ pixels.

Most pixels are represented in two ways: either by a gray scale or by a color scale. For black and white images, each pixel is a scalar value between 0 and 255, with zero corresponding to a "black" value and 255 to a "white" value. Accordingly, values between 0 and 255 are different shades of gray. Having said that, color pixels are more often than not represented in the RGB (red, green, blue) color space.

Figure 1.2.- 1 Depiction of an RGB image (with 3 channels)

We can presume that an RGB image is consisted of three independent matrices of width W and height H, one for each of the RGB components, as illustrated in Figure 1.2.- 1. Additionally, we combine those three matrices in order to obtain a multidimensional array, where $W \times H \times D$ is our matrix. D is the depth or number of channels (for the RGB color space, $D = 3$). Images are also programmatically defined as a 3D $NumPy$ multidimensional array with width, height and depth.

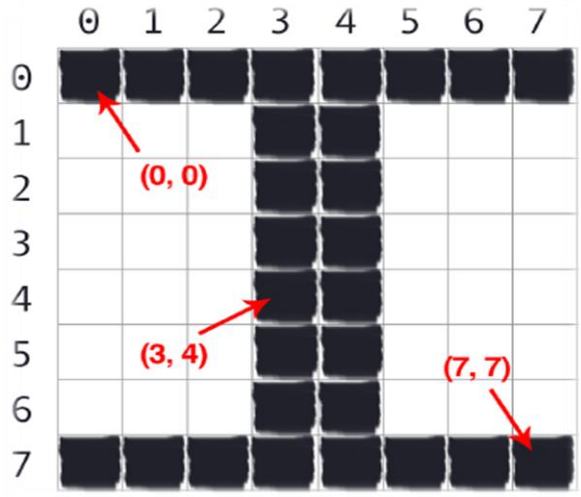


Figure 1.2.- 2 Example of indexing in python on a black and white image

Consider Figure 1.2.- 2, if we make an assumption that the starting point is $(0,0)$ we see that it corresponds to the upper left corner of the image. Howbeit, as we move down, we see that the values of x and y increase. It is clear that this is an 8×8 grid with 64 pixels.

All the while it is important to note this, because unlike other programming languages python is indexed with zero, which means that each countdown starts with 0. The $OpenCV$ library, as well as $Scikit-learn$ represent RGB images with multidimensional $NumPy$ matrices that are shaped heigh, width, depth, give emphasis to this, because it is important that the matrix in this case will not be viewed by default in the code as rows, then columns, but vice versa, precisely because of $NumPy$.

1.2.1. EXPERIMENTAL DATA SET

In the following bulletin points, experimental data sets, which we will use when training neural networks within this master thesis, will be introduced. Some of these data sets are very well known in the world of deep learning, as well as machine learning and represent the basic data sets on which numerous training methods are tested, while other input data sets are the ones we shall label and employ to demonstrate the methods we have learned.

1. MNIST ("NIST" represents the National Institute of Standards and Technology, while "M" means "modified" because the data has been pre-processed to reduce the computational burden). This set of input data is one of the simplest data sets for the necessitates of machine, as well as deep learning. The aim of this data set is to accurately classify handwritten digits from 0 to 9. MNIST itself consists of 60000 images for training and 10000 images for testing. Each vector, i.e. the image is 784 pixels in size, i.e. a 28×28 black and white image. We can clearly see this in Figure 1.2.1.-1 where all of the digits are set on a black background, while they themselves are white, or light gray.

Figure 1.2.1.- 1 MNIST data set

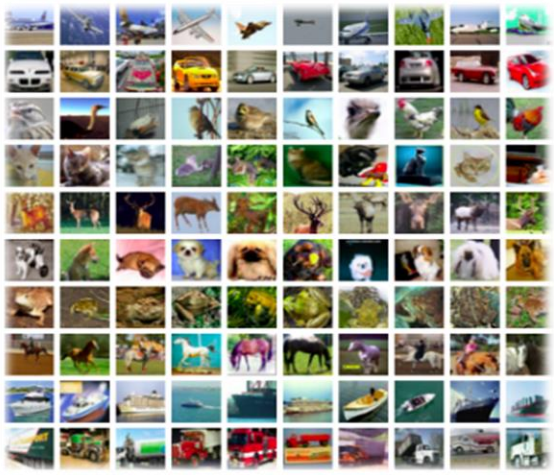


Figure 1.2.1.- 2 CIFAR-10 data set

2. Animals are an input data set on animals, and they are divided into three classes and consist of 3000 images. The task is to accurately classify images that contain either a panda, a dog or a cat. This set of images can easily be trained with a good accuracy, and very quickly so, either on the CPU or on the GPU (we will

explain the advantages and disadvantages of this later in the paper). Henceforth we will use this data set to display ways of easy image recognition, i.e., training of the model, as well as model storage on the disk.

3. CIFAR-10 consists of 60000 images $32 \times 32 \times 3$ (RGB) resulting in a vector with dimensions 3072. As the name suggests, CIFAR-10 consists of 10 classes (deer, planes, cars, birds, cats, dogs, frogs, trucks, ships and horses, this can clearly be seen in the frame of Figure 1.2.1-2). The challenge that arises within this input is that it comes from a dramatic variance in the way objects appear. For example, we can take one random green pixel and we cannot be sure if it is a frog, or a part of the forest where a deer is located, or perhaps it is a green truck.

4. The SMILES data set consists of faces that are either smiling or not. In total, there are 13165 images in gray tones in the data set, and each image is 64×64 in size. The images in this data set are cropped so that only the lips on the face are visible, which allows us to devise a learning algorithm that focuses only on the task of smile recognition.

5. Flowers-17 is a set of data that is divided into 17 classes, i.e. categories and contains about 80 images per class (some of the classes are shown in Figure 1.2.1.-3). Based on the image of the flower, it is necessary to accurately classify which group it actually belongs to. Here we will show why it is problematic to train our neural network insufficiently (underfitting). This is precisely because of the small number of pictures per class (smiling faces). As it had been explained in the previous chapter, it is tremendously important that there is a sufficient amount of input data in order to be able to train the neural network well. In deep learning and convolutional neural networks, it is advised that you have $1000 - 5000$ images per class, minimum.

Figure 1.2.1.- 3 Flowers-17 data set

1.2.2. A CLASSIFICATION FORM ACCORDING TO WHICH DEEP LEARNING IS DONE

It can be said that the process of training neural networks when using deep learning consists of the following steps.

- First, we collect the input data set, all of the images, as well as, classes associated with each image. These classes should come from the final set of categories. Moreover, it is also desirable that a set of images for each category, i.e., class be approximately uniform. In addition, there are methods by which this imbalance can be corrected and we will cover this in detail in Chapters 7 and 8.

- Now that we have the initial data set, we need to divide it into two parts, the training set and the validation set. The training set uses a classifier, that in turn teaches what each category looks like by making predictions on the input data and then correcting when the predictions are wrong. So therefore, when the training is completed, the classifier moves on to a validation data set. Hence, it is of great importance that we separate these two data sets and that they are mutually independent.

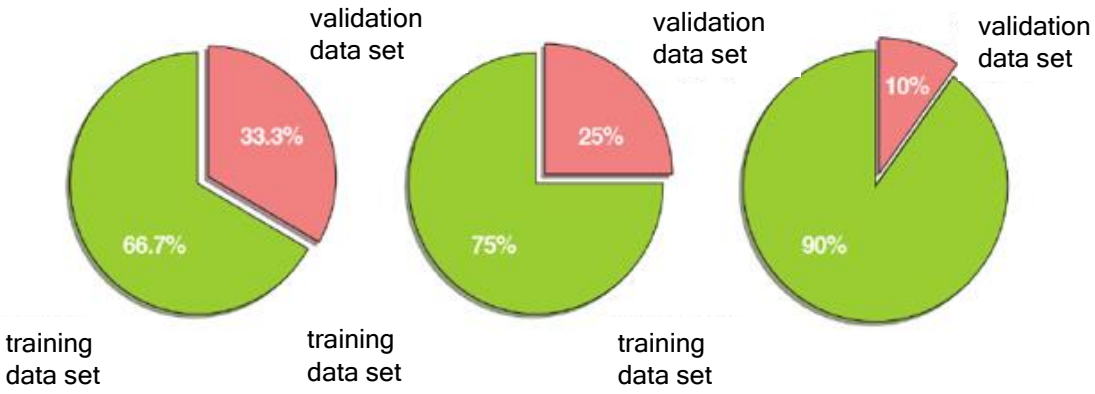


Figure 1.2.2.- 1 Standard division of the training and validation data set

The validation data set must be kept entirely separate from the training process and used only for network assessment. Otherwise, testing the network will not make sense later on. The usual division of input data, shown in Figure 1.2.2.-1, is into a training data set and a data set for validation to a certain extent, i.e. specific proportion. Neural networks have a large number of parameters (e.g. learning rate, dropout, regularization, etc.) that need to be adjusted in order for it to have an optimal performance.

- The next step is training, i.e. training our network where and when to apply the gradient method. The goal of the neural network is to be able to recognize each category of labeled data. When a model makes a mistake, it learns from its mistake and improves by adjusting the parameters and training the neural network again.

- Finally, we need to evaluate our trained neural network. We insert each of the images from the validation data set into the network, and demand it to predict the

class to which each image belongs to. We then compare these model predictions with the accurately labeled values from our validation data set. Labels i.e. the labeled values represent the exact class to which the image actually belongs to. From there, we can calculate the number of predictions that the classifier received and determine the accuracy of the neural network, the feedback of information on the accuracy, as well as the adherence to a particular class. Classifiers are essentially utilized to quantify the performance of the neural network as a whole.

1.3. DEEP LEARNING PARAMETERS

Parameterization is a process of defining the necessary parameters for a given model. In the machine learning task, parameterization includes defining problems in the form of data, accuracy functions, loss functions, weights and bias.

- ✓ Улазни Input data are essential for neural network training. They include data such as pixel intensity and the class that joins them as a label. The input data is displayed in the form of a multidimensional matrix. Each row in the matrix represents one pattern, while each column of the matrix corresponds to a different property. For example, let us view at a data set of 100 images in an RGB color space, each image measuring in dimensions of 32×32 pixels.

  The matrix for this input data set would be obtained as follows:

$$\boldsymbol{X} \subseteq \mathbb{R}^{100\times(32\times32\times3)} \tag{1.1}$$

  where X_i defines $i-th$ image in $\mathbb{R}$.

- ✓ The accuracy function accepts the data as input and maps the data, i.e. connects them to their class label. Hence, given a set of input images, the accuracy function takes these data points and applies some function (accuracy function), and then returns the predicted class labels.

- ✓ The loss function quantifies how much the predicted class labels agree with the exact values. The higher the level of agreement between these two labels (the exact class and the class we predict), the lower the losses (and the higher the accuracy of the classification, at least on the data set used for training). Our goal when training machine learning models is to minimize the loss function, and along these lines also increase the accuracy of classification.

- ✓ The weight matrix, which is denoted by W and the bias b are termed the weights or classifier parameters and they will be the ones that are actually optimized. Thus, based on the output data of the accuracy function and the loss function, we will adjust and change the values of the weights, as well as the bias, in order to increase the accuracy of the classification.

These four components are part of the process of creating a linear classifier, transforming the input data into accurate predictions.

1.3.1. LINEAR CLASSIFICATION

Let us make an assumption that the input data set is denoted as x_i and that each image is associated with a notation of class d_i, and is represented as a set of dimensionalities D. We assume that je $i = 1, 2, \dots, n$ and that $\boldsymbol{d} = 1, 2, \dots, k$, this means that we have n points of input data, separated into k unique categories. Now let us take for example the data set we mentioned earlier, Animals. Within this data set we have a total of 3000 images, all of them are 32×32 pixels in size and are represented using the RGB spectrum. We may display each image as $D = 32 \times 32 \times 3 = 3072$ with different values. There are a total of $K = 3$ classes: one for a dog, one for a cat and one for a panda.

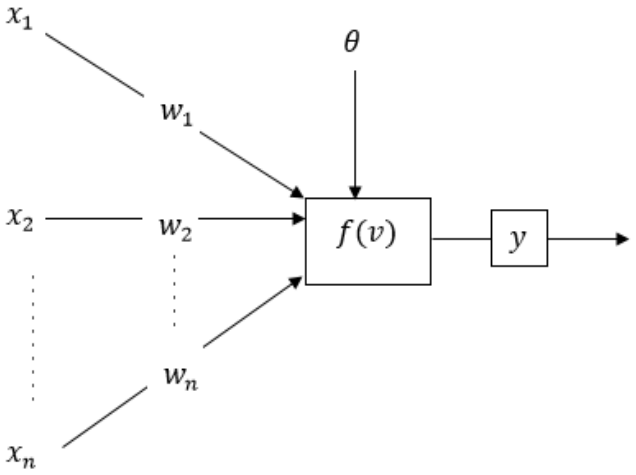


Figure 1.3.1.- 1 General model of a neuron

Given all these variables, we need to define a function that maps images to the labeled classes. Further, observing the Figure 1.3.1.- 1, which depicts a general model of a neuron, primarily we define the $n-$ dimensional input vector:

$$\boldsymbol{x} = [x_1 \quad x_2 \quad x_3 \dots x_n]^T \tag{1.2}$$

Let the output from the neural network be the vector $\boldsymbol{y}$, the magnitude of which always coincides with the number of classes k of the input data in the case of convolutional neural networks. Then we define the vector of weight coefficients, i.e. the weight vector:

$$\boldsymbol{w} = [w_1 \quad w_2 \quad w_3 \dots w_n]^T \tag{1.3}$$

The θ parameter represents the bias. If we observe one node first, that is a neuron, we can easily mathematically explain the nonlinear mapping of the input vector to the output vector (for a moment we will observe y : the output of the neural network as a scalar). The input mapping here represents a linear weight mapping from the n dimensional input space $\boldsymbol{x}$ to the one-dimensional space v i.e. to the activation value of the neuron.

$$v = \boldsymbol{w^T} \cdot \boldsymbol{x} - \theta \tag{1.4}$$

The output of the neurons was obtained, subsequent to the input mapping using the

activation function. This activation function transforms the value obtained by the input mapping into a value that moves further to the next node or the output of the model.

$$\boldsymbol{y} = f(v) \tag{1.5}$$

Conversely, provided that convolutional neural networks consist of input data from multiple neurons, which are coupled into multilayer neural networks with a certain number of hidden and output layers, it is evident that in this paradigm the output from the neural network is the vector $\boldsymbol{y}$. The size of said vector permanently matches the number of classes k of the input data in the case of convolutional neural networks.

In hindsight to the previously started example, let us assume that every x_i is represented as a single column vector with the shape $[D \times 1]$ (in this example, the image the size of $32 \times 32 \times 3$ should be specifically translated into a list of 3072 integers). Our weight matrix $\boldsymbol{w}$ would then have the shape $[k \times D]$ (number of class labels per dimension of the input image). The bias vector θ would be of magnitude $[k \times 1]$. In the Animals example, we conclude that the vector $\boldsymbol{x}$ is represented with $[3072 \times 1]$, the weight matrix with $[3 \times 3072]$, and lastly the bias with $[3 \times 1]$. When the image is translated, i.e. represented with 3072 pixels, this occurs by converting a three-dimensional vector into a one-dimensional list.

1.3.2. LOSS FUNCTION

In order to actually train the neural network on how to perform mapping from input data to class labels using the activation function, we need have a hindsight at the loss functions and the optimization methods.

At the most essential level, the loss function quantifies how “good” or “bad” a given predictor is in classifying the input data into a data set. A visualization of the loss functions during a period of time is plotted for two different models and given in the ensuing example in Figure 1.3.2.- 1.

Hence, the smaller the loss, the better the classifier is in modeling the relationship between the input data and the output class label (although there is a point where the overfitting of the neural network can occur — this transpires when we are giving it too much data, causing the neural network to squander its ability to generalize).

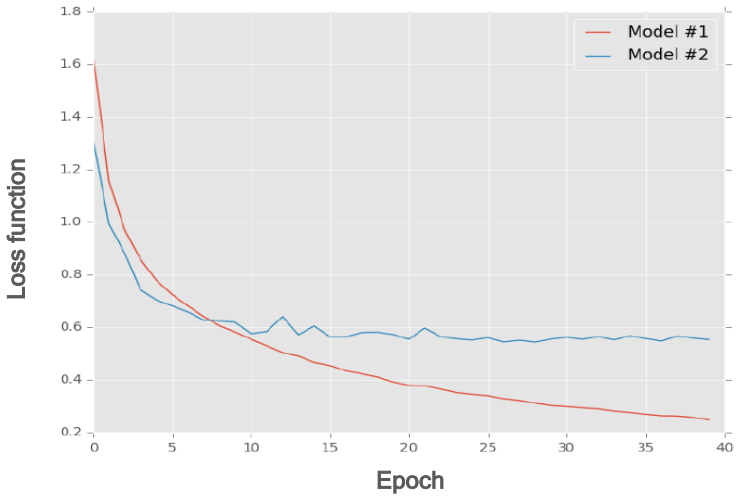


Figure 1.3.2.- 1 Depiction of the loss function for two different models

Idyllically, the loss function should decrease whilst adjusting the model parameters. As can be observed from Figure 1.3.2.- 1 the loss function of model 1 starts by being slightly larger than the loss function of model 2, however it decreases rapidly and remains low when trained on a data set. In contrast, the loss for the model 2 decreases in the beginning, but then stagnates rapidly. In this particular example, model 1 achieves a lower total loss and is probably the more desired model when sorting other images.

1.4. CROSS-ENTROPY AND SOFTMAX CLASSIFIERS

Softmax classifiers provide probabilities for each class label. The Softmax classifier is a binary form of generalization of the logistic regression. The mapping function y is defined so that it acquires an input data set and maps it to the output layer, i.e. class output labels by multiplying $\boldsymbol{x}$ input data and $\boldsymbol{w}$ weight matrix (we assume that the bias is also contained within the input data):

$$s(\boldsymbol{x}, \boldsymbol{W}) = y = f(\boldsymbol{W} \cdot \boldsymbol{x}) \tag{1.6}$$

These scores can be elucidated as unnormalized log probabilities for each class label, which in turn represents the cross-entropy loss:

$$L_i = -\log \frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \tag{1.7}$$

1.4.1. BERNOULLI AND CATEGORICAL DISTRIBUTION

The probability distribution is a description of how likely a random variable or a set of random variables is to employ each of its possible states. The way we describe the probability distribution depends on whether the variables are discrete or continuous.

The anticipated values of some function $f(x)$ with respect to the probability distribution $P(x)$ are the mean value that the function f undertakes when P depends on x. For discrete variables this can be calculated using the following sum:

$$\mathbb{E}_{x\sim P}[f(x)] = \sum_x P(x) f(x) \tag{1.8}$$

while for continuous variables it is calculated using an integral:

$$\mathbb{E}_{x\sim P}[f(x)] = \int p(x) f(x) dx \tag{1.9}$$

When the identity of the distribution is defined, we can denote a random variable in the form of $\mathbb{E}_x[f(x)]$.

The Bernoulli distribution is a distribution over a single random binary parameter. It is regulated by only one parameter $\varphi \in [0, 1]$, that bequeaths the probability that the random variable is equal to 1, and has the following properties:

$$P(x = 1) = \varphi \tag{1.10}$$

$$P(x = 0) = 1 - \varphi \tag{1.11}$$

$$P(x = \mathrm{x}) = \varphi^{\mathrm{x}}(1 - \varphi)^{\ \mathbf{1-x}} \tag{1.12}$$

$$\mathbb{E}_x[x] = \varphi \tag{1.13}$$

$$Var_x(x) = \varphi(1 - \varphi) \tag{1.14}$$

$$H(x) = (\varphi - 1)\log(1 - \varphi) - \varphi \log \varphi \tag{1.15}$$

Categorical distribution is a distribution over one discrete variable with k different states, where k is finite. It is parameterized using the vector $\boldsymbol{p} \in [0,1]^{k-1}$, where p_i provides the probability i of the values, i.e., states. Conclusively, the k state is a state whose probability is depicted with $1 - \mathbf{1}^T\boldsymbol{p}$, where we must note that $\mathbf{1}^T\boldsymbol{p} \leq 1$ is bounded.

1.4.2. SOLVING THE CROSS-ENTROPY FUNCTION

When y is discrete on some finite data set, but not a binary data one, then we can say that Bernoulli's distribution extends to a categorical one. This distribution of data is determined utilizing the vector $N - 1$ of the possibilities that has a sum equal to 1, and where every element is determined with a probability (as it was previously explained that equation $k = y_i$ holds):

$$L_i = -\log \mathrm{P}(Y = y_i | X = x_i) \tag{1.16}$$

The loss function should reduce the probability of negative values of the correct data set class. It is for that particular reason that we need a nonlinearity of the output value that gives such a probability vector, and for that we often use the nonlinearity function dubbed Softmax.

$$\mathrm{P}(Y = k | X = x_i) = \frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \tag{1.17}$$

The standard form for the accuracy function is:

$$s = f(\boldsymbol{x}, \boldsymbol{W}) \tag{1.18}$$

In hindsight, this results in a finite loss function for a single data point:

$$L_i = -\log \frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \tag{1.19}$$

The logarithm here actually has the basis e (natural logarithm). Actual revelation and normalization via the sum of the exponents is a Softmax function. The negative part of the logarithm derives from the cross-entropy function. The calculation of cross-entropy losses for the whole data set is accomplished by calculating the mean value:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i \tag{1.20}$$

1.5. OPTIMIZATION METHOD — GRADIENT DESCENT

The question arises as to how to find the optimal weight vector $\boldsymbol{w}$ and the optimal bias θ that will lead to the finest possible neural network classification. Instead of relying on pure coincidence, we define an optimization algorithm that allows $\boldsymbol{w}$ and θ to be improved. Nonetheless, let us first explain the algorithm used to train neural networks and deep learning models – the gradient descent. The gradient descent method has many variants (which will meticulously be elucidated afterwards), whichever the case, the inkling is the same: iteratively estimate the parameters, calculate the loss, and then move in small steps in the direction that will reduce the loss. The gradient descent method is an iterative optimization algorithm that works over the optimization surface (loss landscape).

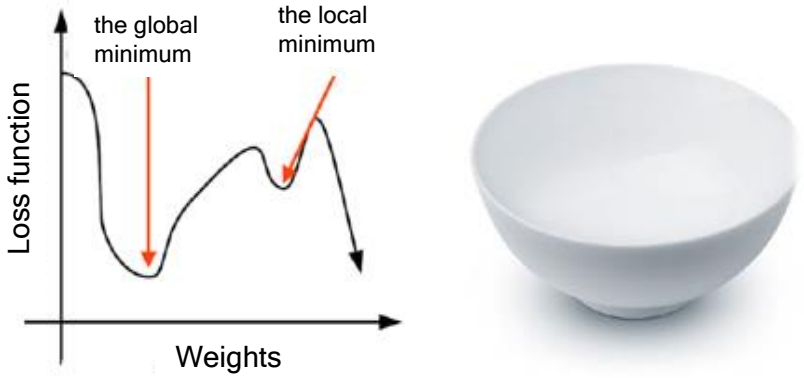


Figure 1.4.- 1 Gradient method display

Considering the Figure 1.4.- 1 we will perceive that the local maximum is the representation of the portion that has a huge loss, and that the local minimum is the portion where the loss is the smallest. However, unlike the depiction in this graph, the maxima and minima are not visible to us while working with neural networks, at least not in this sense. So let's take a bowl for example and look at it as our loss function. So, if we were to start moving from the top of the bowl to the bottom where our hypothetical local minimum is all we would need to do is track the decline of the gradient $\boldsymbol{w}$.

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{1.21}$$

In dimensions > 1, our gradient develops into a vector of partial derivatives. In spite of this, the problem with this equation is that it is an approximation of the gradient and is very slow, so in accordance, it is not applied in practice.

The performance index, which we want to minimize in order to be able to complete the optimization process, is represented by the function $F(x)$, where x denotes a scalar parameter that needs to be adjusted and that belongs to a set of real numbers. For an instant, let us return to the search of optimal values of weights and biases, we can say that our goal is to find a value of x that will minimize the function $F(x)$, described beforehand.

Simply put, we want the value of the function to decrease in each iteration:

$$F(x_{k+1}) < F(x_k) \tag{1.22}$$

The development of a function into the Taylor series around the point x^* is of the form (1.23), while that same development is represented in a matrix form as (1.24), where $\nabla F(x)$ denotes the gradient of the function $F(x)$ and is defined by (1.25).

$$F(x) = F(x^*) + \frac{\partial}{\partial x_1} F(x)|_{x=x^*}(x_1 - {x_1}^*) + \cdots \tag{1.23}$$

$$F(x) = F(x^*) + \nabla F(x)^T|_{x=x^*}(x - x^*) + \cdots \tag{1.24}$$

$$\nabla F(x) = \left[\frac{\partial}{\partial x_1} F(x) \quad \frac{\partial}{\partial x_2} F(x) \ldots \ldots \frac{\partial}{\partial x_n} F(x)\right]^T \tag{1.25}$$

Forthwith, let us consider the Taylor series of the first order of function $F(x)$ around the point x_k:

$$F(x_{k+1}) = F(x_k + \Delta x_k) \approx F(x_k) + g_k^T \Delta x_k \tag{1.26}$$

$$g_k^T \equiv \nabla F(x)|_{x=x_k} \tag{1.27}$$

$$g_k^T \Delta x_k = \alpha_k g_k^T \boldsymbol{p}_k < 0 \tag{1.28}$$

The vector $\boldsymbol{p}_k$ is the exploration direction, while α_k is a positive scalar that symbolizes the learning rate, i.e., the learning parameter and determines the step size. Here we comprehend that the gradient g_k^T is determined in correlation with the value of x_k, and it is evident to us that if it is coveted that $F(x_{k+1})$ be less than $F(x_k)$, than the element containing the gradient must be less than zero. Since the learning parameter (learning rate) is constantly chosen to be a small value which is greater than zero, the previous equation is reduced to a direction of decrease:

$$g_k^T \boldsymbol{p}_k < 0 \tag{1.29}$$

An assumption is introduced that the length of the vector $\boldsymbol{p}_k$ does not change, and that only the direction changes. From the previously posed equations it is quite clear that the decline will occur when the direction vector is equal to the negative gradient, accordingly the vector pointing to the direction of the promptest decline is:

$$\boldsymbol{p}_k = -g_k \tag{1.30}$$

If we take into account this equation and the equation used to adjust the parameters $x_{k+1} = x_k + \alpha_k \boldsymbol{p}_k$, where $k = 0,1,2, \ldots$ we arrive at the equation for the general form of the gradient method:

$$x_{k+1} = x_k - \alpha_k g_k \tag{1.31}$$

1.5.1. CONSTANT LEARNING RATE AND TIME-BASED DECAY

The gradient descent method can be classified based on the manner we determine the learning rate α_k as the method of the constant learning rate and the method of the time-based decay. The constant learning rate method employs a constant value $\alpha_k = \alpha$, for every k, and this is a method with a constant step size. In this particular case, the constant α, i.e. the learning rate is predetermined and fixed, so that the gradient method is reduced to (1.32), and there we have the provided initial condition $x_0 = x(0)$.

$$x_{k+1} = x_k - \alpha g_k \tag{1.32}$$

The time-based decay method tends to minimize the performance index $F(x)$ in relation to α_k with each iteration step. Then the minimization is performed along the line (1.31), α_k is chosen in order to minimize the function $F(x_k - \alpha_k \boldsymbol{p}_k)$. This method requires multiple iterations because the step is determined during each iteration; this is also termed the decay method.
We will be utilizing these two methods constantly in the following chapters, as well as attaining comparisons and drawing conclusions about how they actually affect the training of convolutional neural networks.

1.5.2. THE BIAS TRICK

The "bias trick" technique is a method of combining our weight matrix $\boldsymbol{w}$ and the bias θ into one parameter. Since the activation value is defined as:

$$v(\boldsymbol{x}, \boldsymbol{w}, \theta) = \boldsymbol{w^T} \cdot \boldsymbol{x} - \theta \tag{1.33}$$

Every so often it is very problematic to monitor these two separate variables, both in terms of explanation and application so in order to avoid this situation we may combine $\boldsymbol{w}$ and θ.
So as to combine the bias and the weight matrix, we insert an extra dimension (i.e., a column) to the input data in the form of a constant 1, this is actually our bias. We normally append a novel dimension to each individual $\boldsymbol{x}$ as the first dimension, or the last dimension. This allows the activation value to be written in another way:

$$v(\boldsymbol{x}, \boldsymbol{W}) = \boldsymbol{W^T} \cdot \boldsymbol{x} \tag{1.34}$$

We shall demonstrate mathematically what it looks like when we include the bias as the first dimension. The vector of the weight coefficients that we previously depicted as (1.3) and the input vector that we presented as (1.2) will become:

$$\boldsymbol{x} = [x_0 \quad x_1 \quad x_2 \dots x_n]^T \epsilon \mathbb{R}^{n+1} \qquad , x_0 = 1 \tag{1.35}$$

$$\boldsymbol{w} = [w_0 \quad w_1 \quad w_2 \dots w_n]^T \epsilon \mathbb{R}^{n+1} \qquad , w_0 = -\theta \tag{1.36}$$

while the activation value will be described by the above mentioned equation (1.34).

If we return to the example of the preceding input data, i.e. images of Animals the size of 32×32, where each image contains 3072 pixels, now our x_i has dimensions $[3073 \times 1]$, and the weight matrix is now W with $[3 \times 3073]$. In this way we can approach the handling of the bias as a parameter that can be trained within the weight matrix. An example of the bias trick is provided in Figure 1.4.1.- 1.

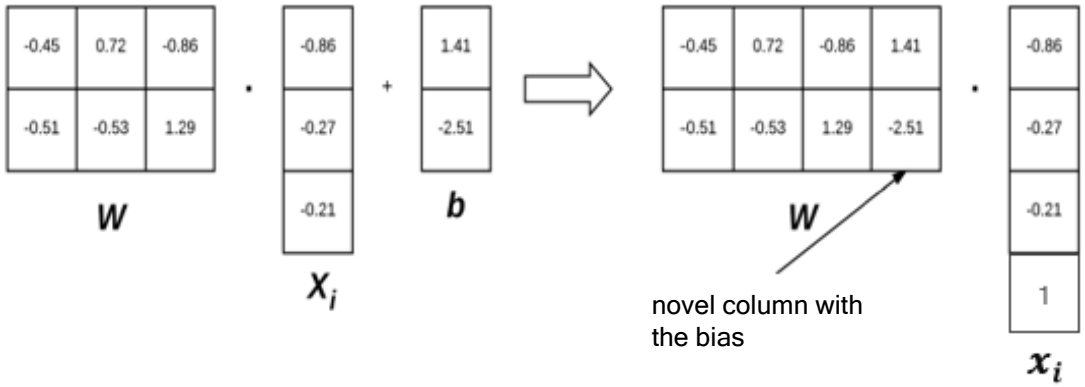


Figure 1.4.1.- 1 Inclusion of the bias in the weight matrix ("the bias trick") – the biases are now trainable parameters in the weight matrix

1.5.3. STOCHASTIC GRADIENT DESCENT

The stochastic gradient descent / SGD method is a simple modification to the standard gradient descent algorithm. It calculates the gradient and adjusts the weight matrix $\boldsymbol{w}$ on small parts of the training data, not on the whole training data set. This modification leads to "noisier" updates, but allows us to take more steps along the ascent (step by step in each series, instead of just one step per epoch), which leads to faster convergence and elimination of negative effects on the loss and the accuracy of classification. SGD is the most important algorithm when it comes to training deep neural networks.

1.5.4. MINI-BATCH STOCHASTIC GRADIENT DESCENT

Analyzing the gradient descent method, based on the previous presentations, it is obvious that this method will be implemented very slowly within the neural network when large data sets are in question. The reason behind this sluggishness is that for each iteration the gradient method requires us to calculate a prediction for each sample from the training data set before we are allowed to adjust the weight matrix.

An alternative type of using the gradient descent method is the usage of mini-batch stochastic gradient method. Instead of calculating a gradient descent on the whole data set, we can take a sample of the data to get a mini-batch. After that, we estimate the gradient on the batch and update the weight matrix w. From an implementation perspective we attempt to randomly arrange the training samples before applying the SGD — as the algorithm is batch sensitive. In a SGD implementation, the size of the mini-batch would be 1, implying that we would randomly take one data point from a training set, calculate the gradient descent and adjust the parameters. However, we often exploit mini-batches that are larger than 1 and the typical batch sizes are $32, 64, 128$, and 256.

If GPU is employed for neural network training, we must primary determine how many samples from the training data set will fit in the GPU, and then use the batch size so that the batch fits on the GPU. For CPU training, we habitually use one of the above mentioned batch sizes to ensure that we take full advantage of the linear libraries to optimize the algebraic calculations within the code. However, it should be noted here that in this paper we only use the CPU, so training one epoch will take us up to a few minutes, while on the GPU it would take a few seconds. For more serious data sets which consist of one million images, per se, it is highly recommended to work solely via the GPU, because otherwise the network training will become a very long process that consumes a lot of time. In turn this process is then even more so difficult when a necessity of carrying out tests and parameter adjustments arises.

1.5.5. MOMENTUM AND NESTEROV'S ACCELERATION

Once, for example, a ball rolls down a hill, the momentum collects more and more, at the same time acting on the ball to increase its acceleration. The momentum applied to the stochastic gradient descent has the same outcome — the objective is to upgrade the standard weight adjustment so as to include the momentum and thus allow our model to achieve a lower loss and a higher accuracy in fewer epochs. The momentum should increase the update strength for dimensions whose gradients move in the same direction, and then decrease the update strength for dimensions whose gradients change directions. The rudimentary idea is to add a term which determines the influence of the last adjustment of weights on the current direction of movement in the weights space. The aforementioned weight update rule, simply involved scaling of the gradient according to the degree of learning:

$$\Delta \boldsymbol{w}(k) = -\eta \nabla E(k) + \alpha \Delta \boldsymbol{w}(k-1) \tag{1.37}$$

Depiction of the inertia term, i.e., the momentum term that permits the increasing of the speed of convergence and achievement of a more efficient learning profile:

$$\Delta \boldsymbol{w}(k) = -\eta \sum_{n=0}^{N} \alpha^n E(k-n) \tag{1.38}$$

The momentum parameter η is commonly set to 0.9. Another method which is used, is to set the parameter γ to 0.5 until learning stabilizes, and thenceforth to increase it to 0.9, and very rarely set it below 0.5. Nesterov's accelerated gradient is nothing but a momentum that "knows when to slow down." When the momentum increases there is a possibility to cross and bypass the local minimum of the error function we are targeting, so then by adjusting the learning parameters we will reduce the momentum and continue to move towards the local minimum.

1.5.6. REGULARIZATION

Regularization facilitates us in controlling the capacity of the model. It allows us to make accurate classifications of the input data set that has not been previously trained. Regularization is nothing but the possibility of generalization. If we do not apply regularization, our classifiers can easily become too complex and overfitting of the neural network can occur and thus we can lose the ability to generalize the validation data set. Anyhow, too much regularization can be a bad thing. We may risk the insufficient use of input data, in which case the model works poorly when used on the training data and is unable to model the relationship between the input data and output class labels.

Let us commence with the cross-entropy loss function, which we explained earlier:

$$L_i = -\log \frac{e^{S_{y_i}}}{\sum_j e^{S_j}} \tag{1.39}$$

The loss on the entire training data set can be penned as:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i \tag{1.40}$$

Assuming that we have obtained a weight matrix w such that every point within the training data set is correctly classified, that is, that our loss is $L = 0$ for all L_i. Then again, such a weight matrix may not perform the classification well, i.e. does not reduce the effects of overfitting. In order to know whether we have adequately selected the matrix, a function is introduced that calculates our weights.

The regularization function L_2 or the weight decay function:

$$R(\boldsymbol{W}) = \sum_i \sum_j W_{i,j}^2 \tag{1.41}$$

This function, which is the sum of the squares of the weights, L_2 regularization distracts large weights from the matrix W, choosing smaller ones. In addition, by removing greater weights, we can improve the ability of generalization and thus reduce the impact of overfitting of the neural network.

Again, our loss function has the same rudimentary form, only now we complement it with the previously clarified regularization:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda R(\boldsymbol{W}) \tag{1.42}$$

The variable λ is a hyperparameter that controls the amount or strength of regularization we apply. In practice, both the learning rate α and the regularization parameter λ are hyperparameters that we adjust.

Extending the cross-entropy loss function to include L_2 regularization offers the following equation:

$$L = \frac{1}{N}\sum_{i=1}^{N}\left[-\log\frac{e^{S_{y_i}}}{\sum_j e^{S_j}}\right] + \lambda\sum_i\sum_j W_{i,j}^2 \tag{1.43}$$

Forthwith, let's look at our standard weight update rule:

$$\Delta \boldsymbol{w}(k) = -\eta\nabla E(k) + \alpha\Delta \boldsymbol{w}(k-1) \tag{1.44}$$

This method adjusts weights based on the multiple gradient with the learning rate α. Taking into account the regularization, the rule of weight update becomes:

$$\Delta \boldsymbol{w}(k) = -\eta\nabla E(k) + \alpha\Delta \boldsymbol{w}(k-1) + \lambda R(k-1) \tag{1.45}$$

Here we append the negative linear expression to our gradients (i.e., the gradient descent), penalizing large weights, with the ultimate objective of making our model easier to generalize.

2. NEURAL NETWORKS

Neural networks are the building blocks of the deep learning system. In order to understand deep learning, we must first explain the fundamentals of neural networks, the neural network architecture, node types and neural network training algorithms. Many tasks involving intelligence, pattern recognition and object detection are extremely difficult to automate, then again if we look at a human it seems as if the human is performing it easily and naturally. The question that arises, for example, is how a small child manages to learn the difference between a school, a home and a market. In fact, how our brain unconsciously performs complex pattern recognition tasks every single day without even noticing it. The answer lies in the fact that all human beings, as well as animals, contain a biological neuron. Neurons are parts of a neural network and are connected to the nervous systems — this network is made up of a large number of interconnected neurons (nerve cells).

The artificial neural network is a computational system that tries to mimic neural connections in the nervous system of human beings. In the following paper, we will be using the phrase neural networks, and it will be implicit that we are discussing about artificial neural networks. For a system to be considered a neural network, it must contain a marked, directed structure of the graph, where each node in the graph performs some simple calculation. Each node performs a simple calculation, and each connection carries a signal (i.e. the calculation output) from one node to another, and it is marked with a weight that illustrates how much each signal is amplified or reduced. Some connections have a large positive weight and thus amplify the signal, which indicates that the signal is very important during classification. Others have negative weights, reducing the signal strength, thus specifying that the output of that node is less important in the final classification process. Such a system is dubbed an artificial neuron. Let's go back to the biological neuron for an instant, Figure 2-1 displays the structure of an ordinary biological neuron that connects further with other neurons via dendrites and axons.

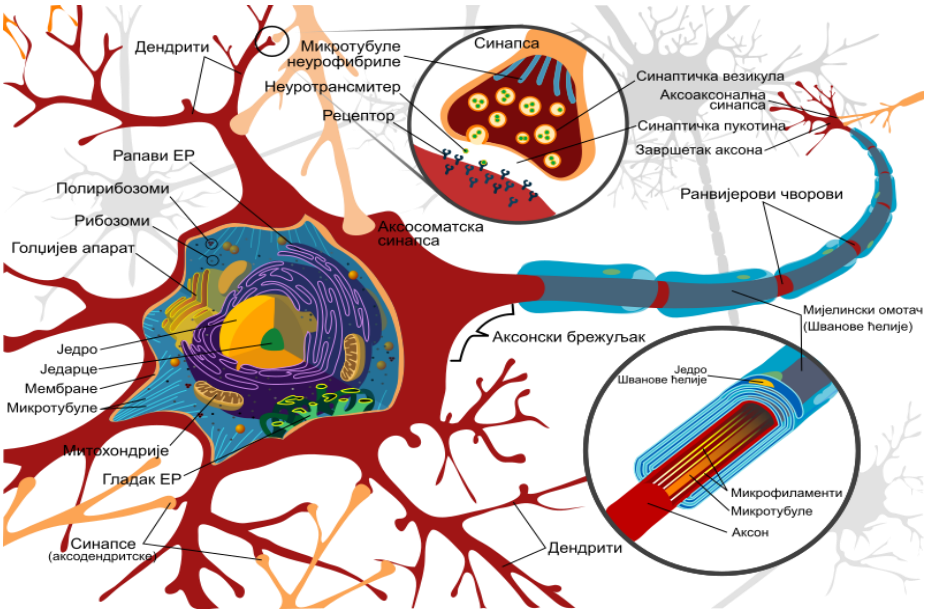


Figure 2.-1 Depiction of a biological neuron

Our brain consists of approximately 10 billion neurons, each connected to about 10.000 other neurons. The cellular body of a neuron is called a soma, and there are dendrites (inputs) and axons (outputs) that connect the soma of this neuron with other neurons. Each neuron receives electrochemical signals from other neurons through its dendrites. If these electrical signals are strong enough they can activate a neuron, and then the activated neuron transmits the signal along its axon, passing it lengthways the dendrites further to other neurons. These newly connected neurons can perform this same operation and continue to transmit signals. Activation of a neuron is nothing more but a binary operation — a neuron is either activated or not activated, depending on whether the set threshold has been crossed when an electrochemical signal has acted upon it. Yet, we emphasize again that the artificial neural networks are only inspired by what we know about the brain. The objective of deep learning is not to imitate how our brain works, but to understand it and utilize it as an inspiration for understanding and developing logic that will allow us to reproduce it one day.

2.1. ARTIFICIAL NEURAL NETWORKS

Figure 2.1.- 1 demonstrates a simple neural network, its values x_1, x_2, x_3 are the inputs to the neural network that usually form one row within the matrix. A constant value of 1 for the bias is added to the matrix with the input data. These input values can be viewed as vectors used to systematically determine the contents of an image.

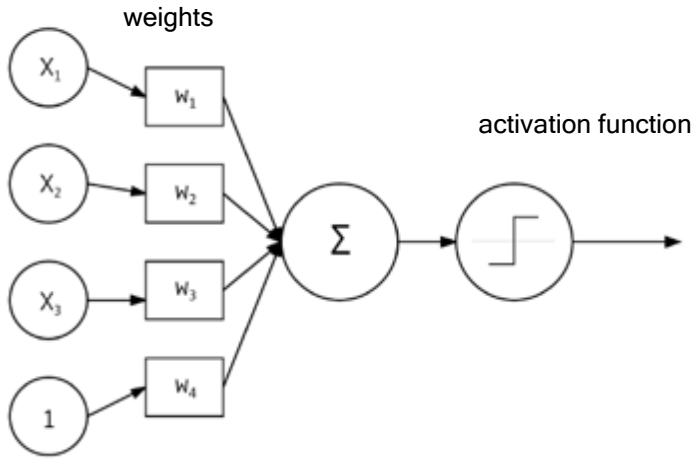


Figure 2.1.- 1 Illustration of a simple neural network

In the background of deep learning, as we will explain later, these inputs are images with a certain pixel intensity. Each input data $\boldsymbol{x}$ is connected to a neuron via a weight vector $\boldsymbol{w}$ consisting of w_1, w_2, w_3, which means that each input $\boldsymbol{x}$ has a corresponding weight $\boldsymbol{w}$. To conclude, the output node on the right side of this figure takes the sum of these multiplied values, and applies to it the activation function f (which is used to determine whether or not the neuron exceeds the sensitivity threshold / bias) thus providing the value. Mathematically, this can be represented in one of three following manners:

- ✓ $f(w_1x_1 + w_2x_2 + \cdots + w_nx_n)$ (2.1)
- ✓ $f(\sum_{i=1}^{n} w_ix_i)$ (2.2)

✓ $f(v)$, where $v = \sum_{i=1}^{n} w_i x_i$ (2.3)

Regardless of how the output value is articulated, we can say that we simply take the sum of the inputs to which certain weight coefficients are associated and apply the desired activation function f to it.

2.1.1. ACTIVATION FUNCTIONS

Definition: Function $f: R \to R$ which maps the activation value of a neuron to its output is called an activation function if that function is continuous in parts.
The simplest activation function is the step function otherwise known as the Heaviside step function, which is used by the perceptron algorithm.

$$f(v) = \begin{cases} 1 & v \geq 0 \\ 0 & v < 0 \end{cases} \tag{2.4}$$

As we can observe from the above equation, this is a very plain threshold function. If the activation value, i.e. the sum of the input data conjoined with the weighting coefficients is $v = \sum_{i=1}^{n} w_i x_i \geq 0$ we have an output 1, otherwise the output is 0. However, although it is intuitive and easy to use, the step function is not differentiable which can lead to problems when employing the gradient descent method and training the neural network. Instead, a more common activation function utilized in neural networks is the unipolar sigmoid function:

$$v = \sum_{i=1}^{n} w_i x_i \qquad f(v) = \frac{1}{1 + e^{-\lambda v}} \tag{2.5}$$

The sigmoid function is a better choice for learning than a simple step function, because it is continuous and differentiable everywhere. This function is symmetric around the y - axis, and asymmetrically approaches the saturation values. The main advantage here is that the curve of the sigmoid function has such a shape that there are no rough transitions and thus it makes it easier to devise learning algorithms. Nonetheless, there are two major problems with the sigmoid function. The first problem is that the outputs of the sigmoidal function are not centered towards zero, while the second problem is that the saturated neurons essentially destroy the gradient, because the deltas of the gradient will be tremendously small. The hyperbolic tangent function / bipolar sigmoid function or $tanh$ function was also widely used as an activation function until the late 1990s, this function can be represented by the following equation:

$$f(v) = \tanh(v) = \frac{1 - e^{-\lambda v}}{1 + e^{-\lambda v}} = \frac{2}{1 + e^{-\lambda v}} - 1 \tag{2.6}$$

The parameter $\lambda > 0$ in sigmoidal functions determines their slope around the point $v = 0$. When the parameter lambda approaches infinity $\lambda \to \infty$, then from the unipolar sigmoidal function we can obtain the step function, and from the bipolar sigmoidal function we can acquire the unipolar sigmoidal function.

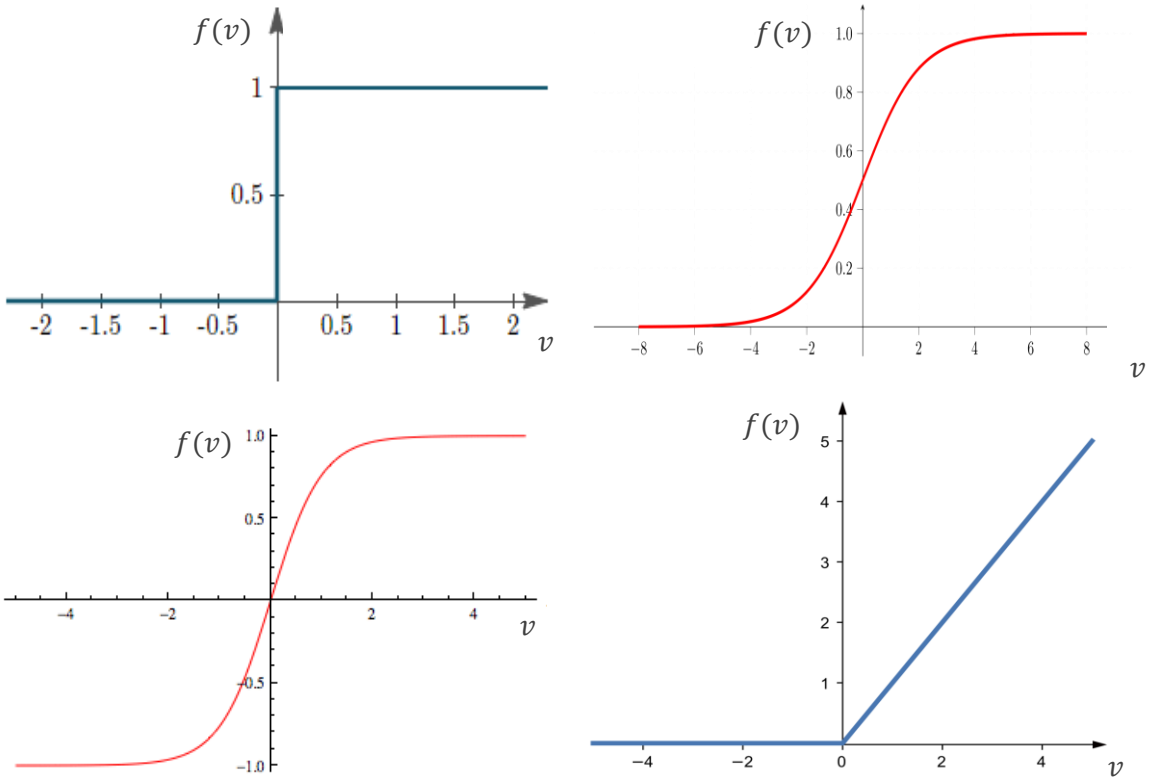


Figure 2.1.1.- 1 Activation functions: step function, unipolar sigmoid function, bipolar sigmoid function, ReLU function (from left to right, top to bottom)

$Tanh$ the activation function is centered at zero, but the gradients approach the zero when the neurons become saturated. At the moment we comprehend that there are better choices for the activation function than the sigmoid and $tanh$ functions. In 2000. Hahnloser introduced the Rectified Linear Unit / ReLU in the paper [10], which is defined as:

$$f(v) = \max(0, v) \tag{2.7}$$

The ReLU functions are also called "ramp functions" because of what they look like when drawn (Figure 2.1.1.-1). Note that the function is equal to zero for negative inputs, but then increases linearly for positive values. The ReLU function is not saturated and is also extremely computationally efficient. Empirically, the activation function of ReLU inclines to surpass both the sigmoid function and the tanh function, and functions in almost all applications.

In combination with Hahnloser's and Seung's works, a paper was published in 2003 [11] and it was found that the activation function of ReLU has stronger biological motivations than previous activation functions, as well as, more comprehensive mathematical justifications. Since 2015, ReLU has been the most widespread activation function utilized in deep learning. However, the problem arises when we have a zero - the gradient descent cannot be calculated.
Subsequently there is a variation of ReLU, called the Leaky ReLU, which allows a small gradient without a zero value when the unit is not active:

$$f(v) = \begin{cases} v & v > 0 \\ \alpha \cdot v & v \leq 0 \end{cases} \tag{2.8}$$

Another variation of ReLU is the ELU function. In this function, exponential linear units attempt to bring the mean value of the activation function closer to zero, which speeds up the learning process. It has been displayed that the ELU functions can obtain higher classification accuracy than the ReLU functions:

$$f(v) = \begin{cases} v & v > 0 \\ \alpha \cdot (e^{v} - 1) & v \leq 0 \end{cases} \tag{2.9}$$

Here $\alpha \geq 0$ denotes a hyperparameter that needs to be adjusted during the training of the neural network.

2.1.2. FEEDFORWARD NEURAL NETWORKS

The most common neural network architecture is the feedforward neural network architecture. In this type of architecture, the connection between nodes is authorized merely between nodes in layer i to nodes in layer $i + 1$. There are no inter-layer or backward connections. Once a feedforward neural network includes feedback (output connections that move back to the inputs) it is called a recurrent neural network. In this paper, we concentrate on advanced neural networks because they are the mainspring of deep learning. Convolutional neural networks are actually nothing more than a special case of a neural networks with feedforward neural network.
In the representation of the following Figure 2.1.2.- 1 we have a feedforward network / network with advanced signal flow with layers 3-2-3-2.

- ✓ Layer 0: the zero layer contains three inputs, with values of x i.e. input data. These can be pixels of intensity of the image or some characteristic vector extracted from the image.
- ✓ Layers 1 and 2: these are the hidden layers containing two and three nodes.
- ✓ Layer 3: the third layer is the output layer or visible layer — there we get the entire output based on the network classification. The output layer habitually has as many nodes as the class labels; one node for each potential output. For example, if we were to create a neural network to classify a handwritten digit, our output layer would consist of ten nodes, one for each digit $0,1,2, \ldots ,7,8,9$.

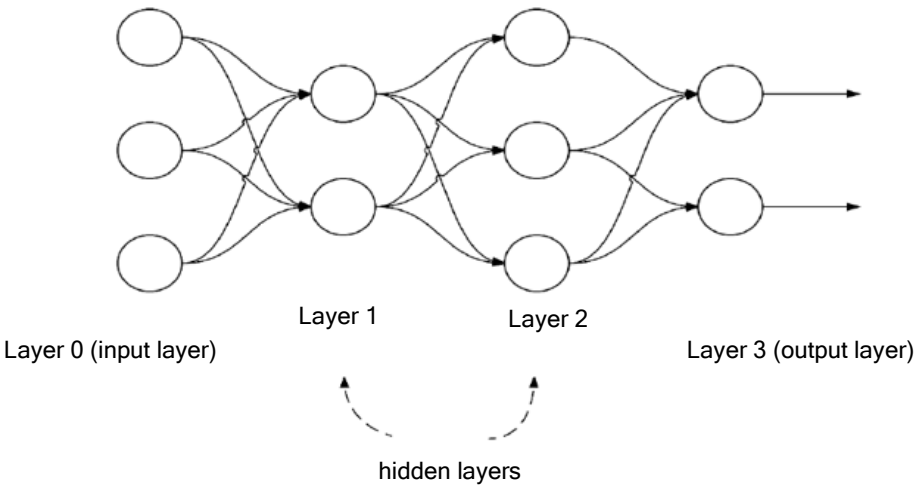


Figure 2.1.2.- 1 Feedforward neural network with 3-2-3-2 layers

2.2. PERCEPTRON AND THE DELTA RULE

The expression perceptron was first introduced by Rosenblatt in 1958. Perceptron [12] is both the oldest and plainest type of artificial neural network. It was later displayed that the single-layer perceptron is not capable in separating nonlinear data points. Since most real-world datasets are nonlinearly separable, it seemed at the time that perceptron, along with other neural network research, could come to a premature end. The training of perceptron is a fairly simple operation. The objective is to obtain a set of weights that accurately classifies each instance in our training input set. In order to train the perceptron, we iteratively bring a training data set to the neural network input multiple times. Iteration involves a step in the algorithm during the training process in which the adjustment of the weights takes place, whereby every time the network moves over the whole set of input data and performs weight adjustment, we articulate that an entire epoch has passed.

It is habitually necessary to have many epochs for a weight vector $\boldsymbol{w}$ to learn to linearly separate our two data set classes. Primary, we convey the sample $\boldsymbol{x}$ to the input, which will then provide the activation value by elementary multiplication with the weight matrix. This activation value then goes through an activation function that will return 1 if $v \geq 0$, and if $v < 0$ it will return a value of 0. Forthwith, we ought to adjust the weight vector $\boldsymbol{w}$ so as to step in the direction closer to the correct classification. This adjustment of the weight vector transpires with the aid of the delta rule.

The expression $(\boldsymbol{d} - \boldsymbol{y})$ determines whether or not the output classification is correct. Hence, if the classification is correct, then that difference will be zero. Otherwise, the difference will be either positive or negative, giving us a direction in which the weight will adjust and conveying us closer to the correct classification. Then we multiply $(\boldsymbol{d} - \boldsymbol{y})$ with $\boldsymbol{x}$, in doing so we are approaching the correct classification.

Let us summarize the learning, i.e. the training of perceptron:
1. we initialize a weight vector $\boldsymbol{w}$ with small random values
2. neural network training procedure:
   - ✓ let us pass over each characteristic vector $\boldsymbol{x}$ and the precise class label in our training data set $\boldsymbol{d}$
   - ✓ by taking the $\boldsymbol{x}$ and passing it through the neural network, as well as, calculating the output value:

$$\boldsymbol{y} = f(\boldsymbol{w} \cdot \boldsymbol{x})$$

   - ✓ "the delta rule": adjusting the weights $\boldsymbol{w}(t+1) = \boldsymbol{w}(t) + \eta(\boldsymbol{d} - \boldsymbol{y})\boldsymbol{x}$ for all characteristics $0 \leq i \leq n$

The parameter α is the learning rate and controls how large (or how small) step we choose. It is critical that this value is set correctly, since it is usually kept small in order to ensure the convergence and prevent oscillations. At the beginning of the neural network training, the parameter α should be much higher in order for the weight coefficients to move faster to the point of minimum. Smaller values of the learning rate α are suitable at the final stages of training, in order to avoid jumping over the global

minimum. Precisely, set in this manner the parameter α permits us not to bypass the local / global minimum.

2.2.1. SOLVING THE GENERALIZED DELTA RULE FOR THE OUTPUT LAYER OF NEURAL NETWORK

Initially, let us define the current error function for the neural network as the sum of squared errors of all of the output neurons:

$$E = \frac{1}{2}\sum_{j=1}^{m}\big(d_j(k) - y_j(k)\big)^2 = \frac{1}{2}\sum_{j=1}^{m}\Big(e_j(k)\Big)^2 \tag{2.10}$$

How about we say that the neural network we are observing has M layers. Consider a neuron $neuron(M,j)$ in the output layer, where j is the number of neurons in the output layer, E is the function of the current error described by (2.10), and its partial derivative, i.e. the element error signal is displayed as:

$$\delta_j^{(M)} = -\frac{\partial E}{\partial v_j^{(M)}} \qquad j = 1,2,\ldots.,m \tag{2.11}$$

$$\delta_j^{(M)} = -\frac{1}{2}\frac{\partial \sum_{l=1}^{m}(d_l - y_l)^2}{\partial v_j^{(M)}} = -\frac{1}{2}\frac{\partial \sum_{l=1}^{m}\Big(d_l - f(v_l^{(M)})\Big)^2}{\partial v_j^{(M)}} \tag{2.12}$$

$$\delta_j^{(M)} = -\frac{1}{2}\sum_{l=1}^{m}\frac{\partial\Big(d_l - f(v_l^{(M)})\Big)^2}{\partial v_j^{(M)}} \tag{2.13}$$

Since $l \neq j$ are mutually independent, we obtain:

$$\delta_j^{(M)} = \Big(d_l - f(v_l^{(M)})\Big)\frac{\partial f(v_l^{(M)})}{\partial v_j^{(M)}} = e_j^{(M)} f'(v_j^{(M)}) \tag{2.14}$$

The delta error which corresponds to the neuron in the output layer $\delta_j^{(M)}$ is equal to the product of the errors of the output $e_j^{(M)}$ and the differential signal of the nonlinear activation function $f'\left(v_j^{(M)}\right)$. In order to obtain an expression for the weight adjustment, the gradient of the error function E with respect to the weight vector $\boldsymbol{W}_j^{(M)}$ should be calculated.

$$\nabla_{\boldsymbol{W}_j^{(M)}} E = \frac{\partial E}{\partial \boldsymbol{W}_j^{(M)}} = \sum_{l=1}^{m}\frac{\partial E}{\partial v_l^{(M)}}\frac{\partial v_l^{(M)}}{\partial \boldsymbol{W}_j^{(M)}} \tag{2.15}$$

$$\nabla_{\boldsymbol{W}_j^{(M)}} E = \frac{\partial E}{\partial v_j^{(M)}}\frac{\partial v_j^{(M)}}{\partial \boldsymbol{W}_j^{(M)}} \tag{2.16}$$

It is introduced for $v_j^{(M)} = \left(\boldsymbol{W}_j^{(M)}\right)^T \boldsymbol{Z}$, where we know that $\boldsymbol{Z}$ is the extended input vector for the output layer M:

$$\nabla_{\boldsymbol{W}_j^{(M)}} E = -\delta_j^{(M)}\boldsymbol{Z} = -e_j^{(M)} f'(v_j^{(M)})\boldsymbol{Z} \tag{2.17}$$

The law of the weight vector update for the output layer is provided by:

$$\boldsymbol{W}_j^{(M)}(k+1) = \boldsymbol{W}_j^{(M)}(k) - \alpha \nabla_{\boldsymbol{W}_j^{(M)}} E(k) \tag{2.18}$$

$$\boldsymbol{W}_j^{(M)}(k+1) = \boldsymbol{W}_j^{(M)}(k) + \alpha \delta_j^{(M)}(k)\boldsymbol{Z}(k) \tag{2.19}$$

$$\boldsymbol{W}_j^{(M)}(k+1) = \boldsymbol{W}_j^{(M)}(k) + \alpha e_j^{(M)}(k) f'(v_j^{(M)})(k)\boldsymbol{Z}(k) \tag{2.20}$$

Equation (2.20) is called the generalized delta rule, because the calculation of the delta derivative is included in the algorithm.

2.3. BACKPROPAGATION ALGORITHM

Theback propagation algorithm / BP algorithm is undoubtedly the most important algorithm in the history of neural networks. Without an efficient BP algorithm, it would be impossible to train deep neural networks to the depths that are being implemented nowadays. The BP algorithm can be considered as the cornerstone of modern neural networks and deep learning. The original backpropagation algorithm was introduced in the 1970s. However, only on the basis of a seminar paper [13] in 1986, whose authors were Rumelhart, Hinton and Williams, it was found to be conceivable to devise a faster algorithm, more skillful and more suitable for deeper training of neural networks.

The backpropagation algorithm consists of two phases:

1. The feedforward neural network where the input data are transmitted through the neural network, thus obtaining the intended output.
2. Backpropagation, where the gradient of the loss function at the final layer (i.e., the predictive layer) of the neural network is calculated. Then we use this gradient descent for recursive adjustment of weights within the neural network (also known as the weight update phase).

2.3.1. BACKPROPAGATION ALGORITHM: MATHEMATICAL APPROACH

On the example of a simple multilayer neural network, we will elucidate the BP algorithm, i.e., the backpropagation algorithm, as well as its implementation. Foremost, consider a neural network that contains two nodes (neurons), in each of its layers, the input, output, and hidden layer. We shall again omit the bias for convenience. Figure 2.3.1.- 1 displays an example of this neural network.

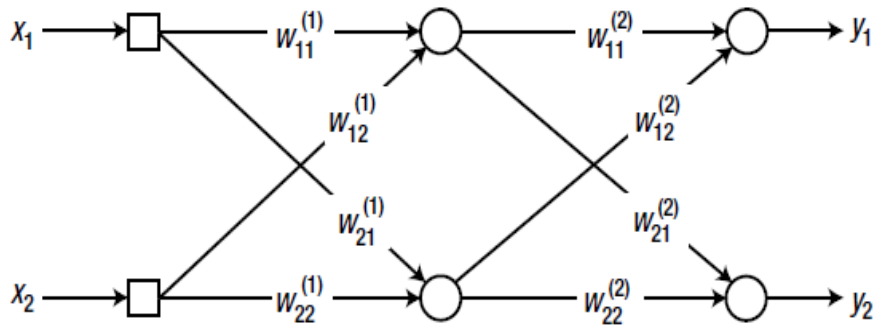


Figure 2.3.1.- 1 Example of a neural network

To acquire an error based on the output, we first need a neural network output based on the input data. In this example, the neural network has one hidden layer and we need two manipulations of the input data before we can calculate the output data.
The activation value of the hidden node is calculated as:

$$\begin{bmatrix} v_1^{(1)} \\ v_2^{(1)} \end{bmatrix} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \triangleq W_1 \cdot x$$

When we calculate this activation value, we insert it into the activation function and then get the output from the hidden nodes. Thenceforth, in a similar way, the weighted sum of the output nodes is calculated.

$$\begin{bmatrix} y_1^{(1)} \\ y_2^{(1)} \end{bmatrix} = \begin{bmatrix} \varphi(v_1{}^{(1)}) \\ \varphi(v_2{}^{(1)}) \end{bmatrix}$$

$$\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} \\ w_{21}^{(2)} & w_{22}^{(2)} \end{bmatrix} \cdot \begin{bmatrix} y_1^{(1)} \\ y_2^{(1)} \end{bmatrix} \triangleq W_2 \cdot y^{(1)}$$

As soon as we put this weighted sum into the activation function the neural network provided the output.

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \varphi(v_1) \\ \varphi(v_2) \end{bmatrix}$$

We will now train the neural network applying the backpropagation algorithm. Primary, in order to perform the adjustment of the weight coefficients it is compulsory to calculate the value of the magnitude δ of the error of each node, based on the previously clarified delta rule.

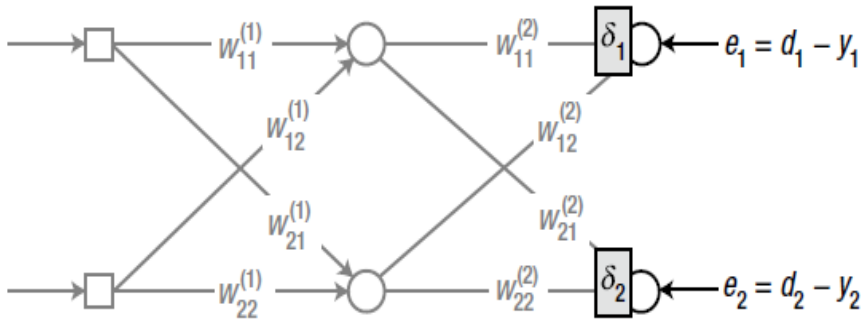


Figure 2.3.1.- 2 Generalized delta rule

The backpropagation algorithm uses a modification of the delta rule, which is previously elucidated and is dubbed the generalized delta rule, where $\varphi'(\cdot)$ is the first derivative of the activation function of the output layer, while y_i is the output data from the output node, while d_i is the exact value established on the basis of the input data, and v_i is the weight sum of the corresponding node.

$$\mathrm{e}_1 = \mathrm{d}_1 - \mathrm{y}_1$$
$$\delta_1 = \varphi'(\mathrm{v}_1)\mathrm{e}_1$$
$$\mathrm{e}_2 = \mathrm{d}_2 - \mathrm{y}_2$$

$$\delta_2 = \varphi'(v_2)e_2$$

Seeing as we have δ errors for every output node, we proceed further left to the hidden nodes and calculate the δ errors. Redundant connections are obscured for convenience in Figure 2.3.1 - 3.

In the backpropagation algorithm a node is defined as a weighted sum of δ values that propagates backward from the layer, the immediate right side (in this case the output layer). Whilst an error is obtained, the calculation of the delta from the node is displayed using the following equation.

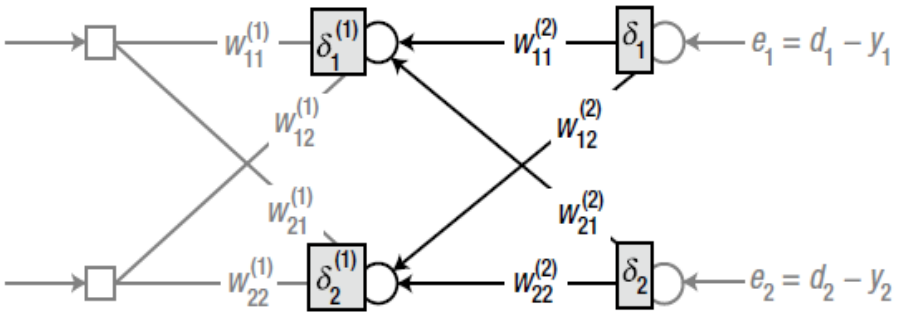


Figure 2.3.1.- 3 Demonstration of computing the delta rule

$$e_1{}^{(1)} = w_{11}^{(2)}\delta_1 + w_{21}^{(2)}\delta_2$$
$$\delta_1{}^{(1)} = \varphi'(v_1{}^{(1)})e_1{}^{(1)}$$
$$e_2{}^{(1)} = w_{12}^{(2)}\delta_1 + w_{22}^{(2)}\delta_2$$
$$\delta_2{}^{(1)} = \varphi'(v_2{}^{(1)})e_2{}^{(1)}$$

Summarizing, the error of the hidden layer is calculated backwards as a weighted sum of δ errors, and the δ node error is the product of the error and the derivative of the activation function. This process commences at the output of the layer and is then repeated for all hidden layers. The two formulas for calculating the error are combined within the matrix and can be represented by the ensuing equation:

$$\begin{bmatrix} e_1^{(1)} \\ e_2^{(1)} \end{bmatrix} = \begin{bmatrix} w_{11}^{(2)} & w_{21}^{(2)} \\ w_{12}^{(2)} & w_{22}^{(2)} \end{bmatrix} \cdot \begin{bmatrix} \delta_1 \\ \delta_2 \end{bmatrix}$$

Formulate a juxtaposition of this equation with the output of the neural network. The matrix of the preceding equation is the result of the transpose of the weight matrix W, so the equation can be written as:

$$\begin{bmatrix} e_1^{(1)} \\ e_2^{(1)} \end{bmatrix} = W_2^T \cdot \begin{bmatrix} \delta_1 \\ \delta_2 \end{bmatrix}$$

This equation implies that we can attain the error as the product of the transposed weight matrix and the δ error. This is a very useful attribute and makes the implementation of the algorithm much more straightforward. If we have additional hidden layers, we shall

just repeat the same backwards progression and process each hidden layer, as well as calculate all of the deltas.
Once all of the deltas have been calculated, we can train the neural network, and hence just use the succeeding equation to adjust the weights of the corresponding layers:

$$\Delta w_{ij} = \alpha \delta_i x_j$$
$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$$

An example of the weight update may be depicted with $w_{21}^{(2)}$:

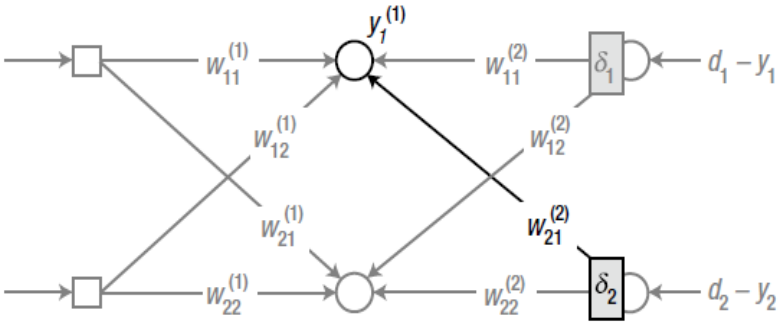


Figure 2.3.1.- 4 Example of weight adjustment $w_{21}^{(2)}$

$$w_{21}{}^{(2)} \leftarrow w_{21}{}^{(2)} + \alpha \delta_2 y_1{}^{(1)}$$

An example of the weight update may be depicted with $w_{11}^{(1)}$:

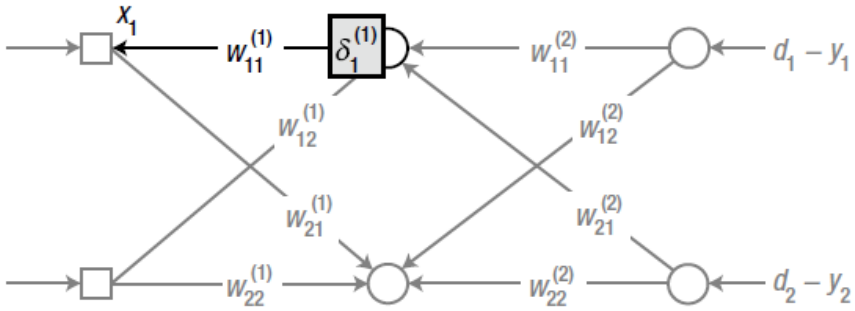


Figure 2.3.1.- 5 Example of weight adjustment $w_{11}^{(1)}$

$$w_{11}{}^{(1)} \leftarrow w_{11}{}^{(1)} + \alpha \delta_1 x_1{}^{(1)}$$

3. CONVOLUTIONAL NEURAL NETWORKS

Convolutional neural networks originated from the study of the visual cortex of the brain and have been utilized to recognize images since the 1980s.
In recent couple of years, thanks to the increase in speed and performance of today's computers, convolutional neural networks have been able to achieve superhuman performance on some complex visual tasks. Convolution neural networks / CNNs are not limited to visual perception, they are also prosperous in other tasks such as voice recognition or natural language processing. Nevertheless, we will concentrate on their visual applications within this paper, but before that let us look at how this came about.

Hubel and Wissel performed experiments on cats in 1958 and 1959 (and then a not many years later on monkeys), with which they managed to give a crucial insight into the structure of the visual cortex. They showed that many neurons in the visual cortex have a small local receptive field, meaning that they only respond to visual stimuli located in a limited area of the visual field. The receptive fields of numerous neurons can overlap, and join together to form the entire visual field. Moreover, they illustrated that some neurons merely react to images of horizontal lines, while others react simply to the lines with different orientations, ie. vertical lines. This means that two neurons can have the same receptive field, but react to different line orientations. They also observed that some neurons have larger receiving fields and respond to more complex patterns that are combinations of lower-level patterns. These observations led to the idea that higher-level neurons are based on the outputs of neighboring lower-level neurons, and that this powerful architecture is able to detect all kinds of complex patterns in any area of the visual field.

These studies of the visual cortex inspired the “neocognitron”, that was introduced in 1980 and then gradually evolved into what we nowadays convene convolutional neural networks. An notable turning point for all of this occurred in 1998 when the work of LeCun, Boto, Beng, and Hafner was published, and in so exhibited the famous $LeNet$ architecture (which we will demonstrate in Chapter 6). This architecture has some blocks that we have already explained in the previous chapter, such as fully connected layers and the activation function, but two novel main layers are introduced, namely the convolutional layer and the pooling layer.

3.1. FILTERS

If we observe the image as a large matrix and the filter / kernel as a small matrix (relative to the whole image of course), as the following Figure 3.1.- 1 depicts, we can declare that we slide with the filter (red cube) from left to right and above then down in the original image. At each (x, y) coordinate of the original image, we stop and examine the pixels located in the center of the image filter. Then we take this square of pixels combine them with a filter and get a unique output value. The output value is simultaneously stored in the output image at the same (x, y) coordinates as the center of the filter.

Let us foremost demonstrate what a filter / kernel looks like:

$$K = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Figure 3.1.- 1 Demonstration of a squared matrix filter the size of 3×3, which can be of arbitrary rectangular size $M \times N$, provided that both M and N are odd integers.

We use an odd filter size to determine that there is a valid integer (x, y) - the coordinate system is located in the center of the image. On the left we have a matrix the size of 3×3. The center of the matrix is at $x = 1,\ y = 1$ because the upper left corner of the matrix is utilized as the source and we are acquainted with the fact that in python the coordinates are zero indexed. However, if we had a matrix the size of 2×2 on the right, then the center of this matrix would be located at a point $(0.5;\ 0.5)$, and since we know that without interpolation there is no pixel location $(0.5;\ 0.5)$ - the pixel coordinates must be integers. This is exactly why we use odd filters, to ensure that there is suitable (x, y) coordination at the center of the larger matrix.

3.2. CONVOLUTIONAL LAYER

The most important block of the convolutional neural networks is the convolutional layer / CONV layer. The neurons in the first convolutional layer are not connected to each individual pixel in the input image, but only to the pixels in their receiving fields (the fields they reach).
Then, each neuron in the second convolutional layer is connected only to neurons located in a small rectangle within the first layer (as demonstrated in Figure 3.2.-1).

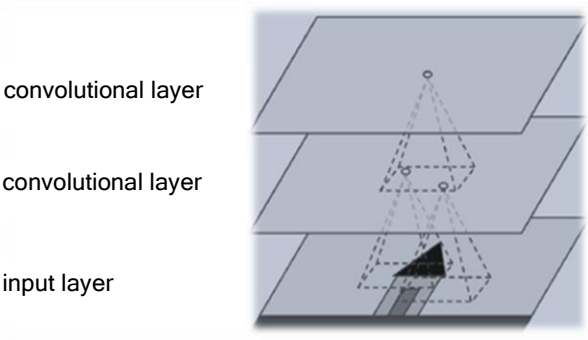


Figure 3.2.- 1 Hierarchical structure of convolutional layers

This architecture allows the neural network to concentrate on low-level functions in the first hidden layer and then assemble them into higher-level functions within the next hidden layer, and so on.
This hierarchical structure is conventional in images that appear as real world problems, which is one of the reasons why convolutional neural networks function so well for image recognition. The neuron located in row i, column j of the given layer is connected to the outputs of neurons from the previous layer, which is located in rows from i to $i + f_h - 1$, and columns from j to $j + f_w - 1$, where f_h and f_w are the height and the width of the receptive field. In order for the current convolutional layer to have the same height and width as the previous layer, it is common to add zeros around the input, as indicated in Figure 3.2.-2. This is known as the zero padding method.

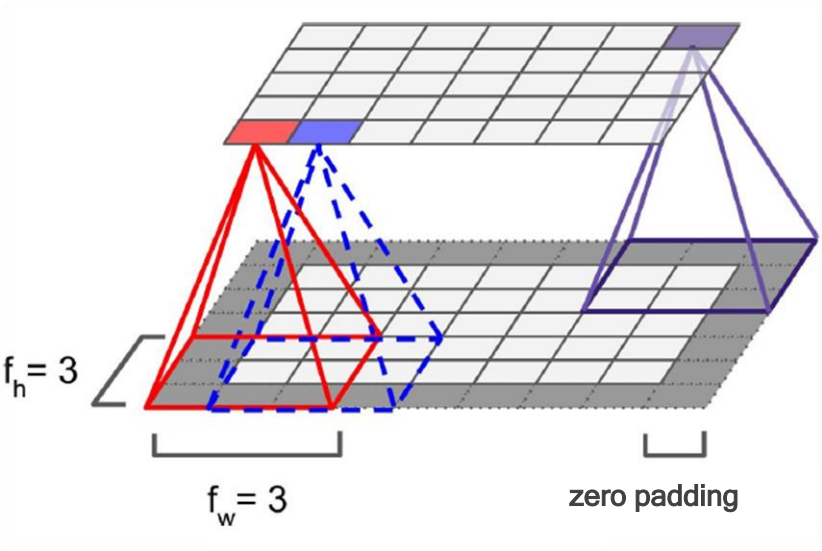


Figure 3.2.- 2 Demonstration of the zero padding method

It is also doable to connect a large input layer to a much smaller layer by arranging the receiving fields, as disclosed in Figure 3.2.- 3 The distance between two consecutive receiving fields is called the stride. In the diagram, an input layer the size of 5×7 (plus zero as an appendix) is connected to a layer the size of 3×4, using receiving fields the size of 3×3 and a stride of 2 (in this example, the stride in both directions is the same, however it does not have to be). The neuron located in the row i and column j in the upper layer is connected to the outputs of neurons from the previous layer, which is located in rows ixs_h to $ixs_h + f_h - 1$, columns $jxs_w + f_w - 1$, where s_h and s_w are vertical and horizontal strides.

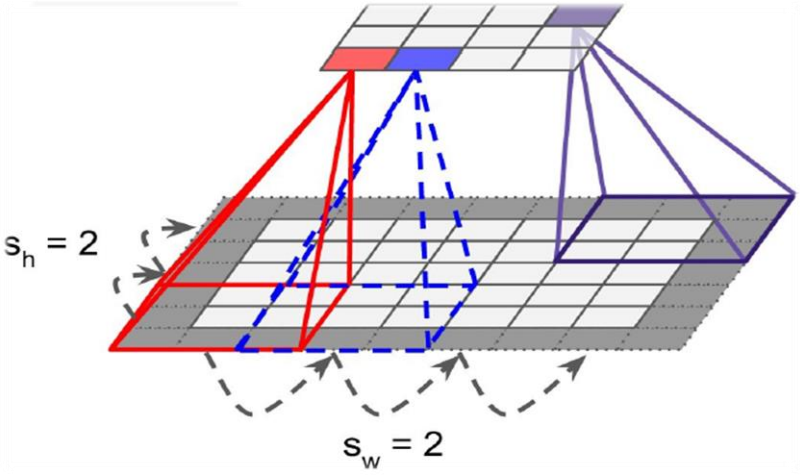


Figure 3.2.- 3 Demonstration of the stride inside the layer

3.2.1. CONVOLUTIONAL LAYER: MATHEMATICAL APPROACH

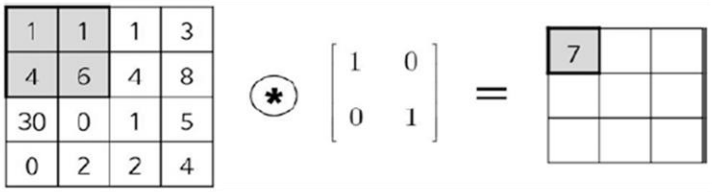
Given that it is very difficult to represent the convolution layer, we will be using a simple mathematical example to explain how the convolution layer essentially functions. Let us consider an image the size of 4×4 pixeles that is conveyed as a matrix and depicted in Figure 3.2.1.- 1 We will closely disclose how our previously clarified filter moves along this image.

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

1	1	1	3
4	6	4	8
30	0	1	5
0	2	2	4

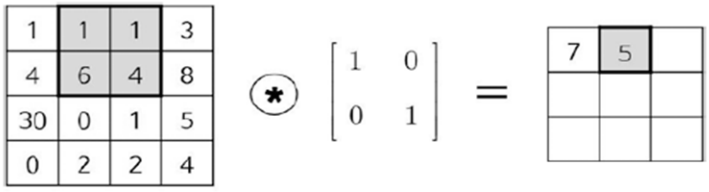
Figure 3.2.1.- 1 An example where the convolutional layer is employed

Subsequently, since we can say that one filter / kernel consists of two convolutional filters the size of $F \times F$, we can conclude that we actually use two filters in this example. We start by moving the first filter. The convolution operation begins in the upper left corner at the part of the matrix that is the same size as the convolution filter.

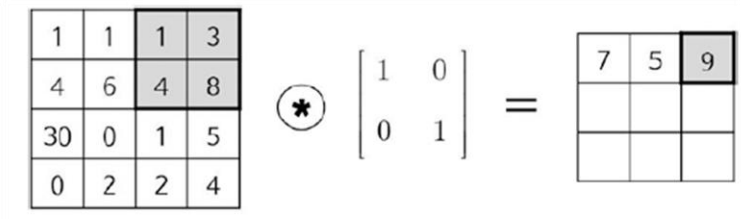


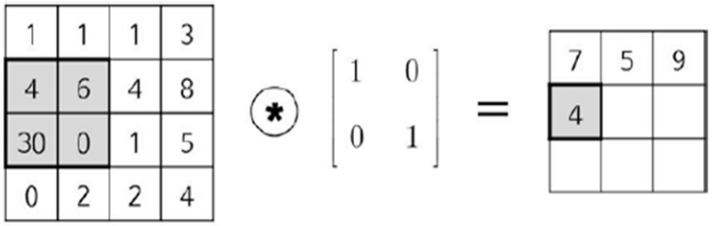
A convolutional operation is the sum of the products of elements that are located at the same positions in both matrices and is calculated as follows:

$$(1 \times 1) + (1 \times 0) + (4 \times 0) + (6 \times 1) = 7$$

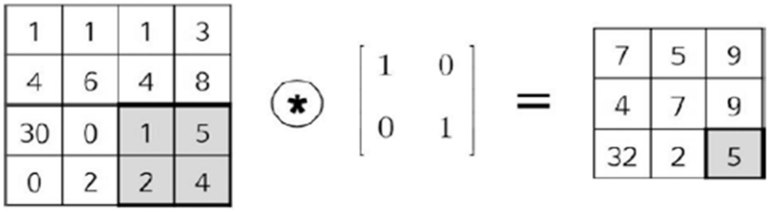


For the ensuing submatrix another convolutional operation was performed and immediately after it, a third convolutional operation was carried out.

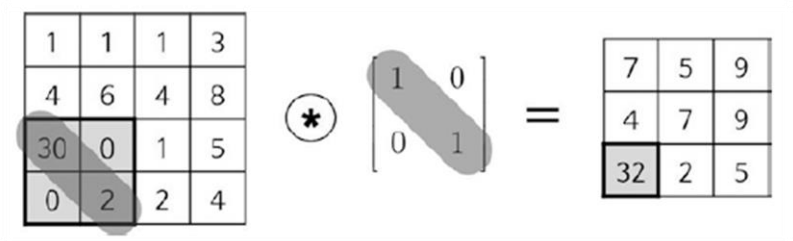




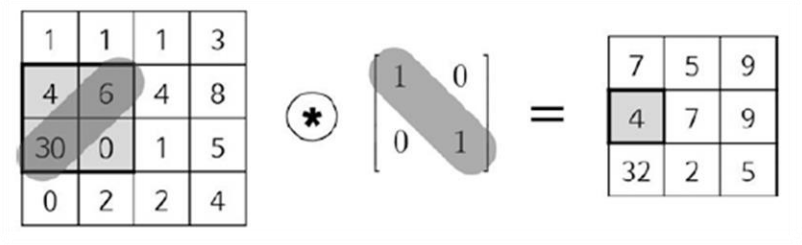
After the upper row is finished, the subsequent row starts again on the left side.



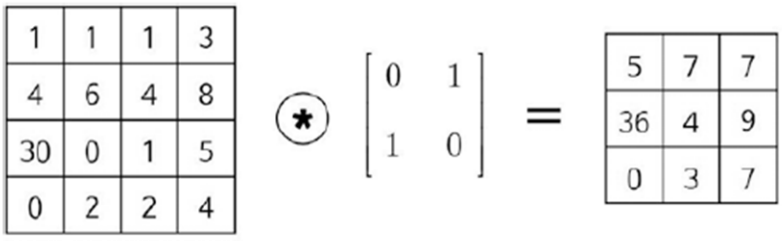
The same procedure is repeated until the whole map of the characteristics of the given filter is displayed. We see that the element $(3,1)$ on the map indicates the largest value.



On the figure it is visible that the image submatrix matches the convolutional filter; both are diagonal matrices with significant numbers on the same matrix fields. The convolution operation produces large values when the input matches the filter.



In contrast, the same significantly higher number 30 does not affect the convolution result, on the contrary the result is only 4. This is because the image matrix does not match the filter, the significant image elements and the matrix are aligned in the wrong direction. In the same manner by processing the second convolution filter we get the following map.



Similar to the first convolution operation, the values in the elements of this matrix depend on whether or not the image matrix matches the convolution filter. In short, the convolution layer manages the convolution filters on the entered images and produces feature maps. The features that are extracted vary depending on which filter is used. The feature map created by the convolution filter is processed through preprocessing and an activation function (ReLU or Softmax) before it appears in the output layer.

3.2.2. DEPTH

The depth of the output layer controls the number of neurons (i.e. filters) in the CONV layer that connects to the local region of the input volume. Every filter produces an activation map that is "activated" in the presence of oriented edges, spots or color.
For a particular CONV layer, the depth of the feature map shall be $K = F \times F$, or simply the number of filters we are learning in the contemporary layer. Let us articulate that our convolutional layer utilizes $64 \times 64 = F \times F$ filters, in this case the depth would be 64 filters. The set of filters located at the same (x, y) location is called the column depth. Note that this does not mean the depth of the image (whether the black and white image has a $depth = 1$ or the color image has a $depth = 3$), but the depth created by the convolution filter when creating the previously described feature map.

3.2.3. STRIDE

The stride in the perspective of convolutional neural networks can have the following principle - for each step we create a new depth step around the local region of the image, into which we insert each of the $K = F \times F$ (kernels) filters. When creating our CONV layers we commonly use a stride step size S which is either $S = 1$ or $S = 2$. Smaller strides will result in overlapping receptive fields and larger output images. Conversely, larger steps will result in less overlap of receptive fields and smaller output images.

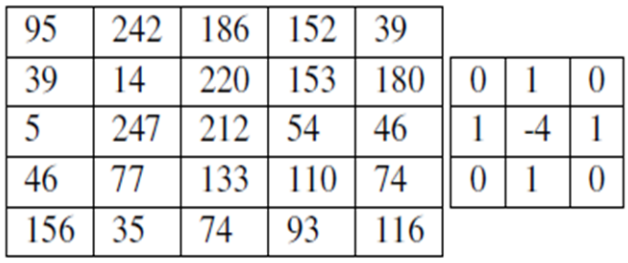


Figure 3.2.3- 1 The image before preprocessing (on the left), and filter (on the right)

Let us consider the Figure 3.2.3.- 1, where we have an input image the size of 5×5 and overlap it with a filter the size of 3×3 using a stride $S = 1$. The filter slides from left to right, and from top to bottom, one pixel at a time, creating the subsequent output that is depicted in Figure 3.2.3.- 2. However, if we were to apply an equivalent operation, this time with a stride of $S = 2$, then we would have skipped two pixels simultaneously (two pixels along the x axis and two pixels along the y axis), creating a smaller output volume, which is displayed in Figure 3.2.3.- 2. Here we establish that convolutional layers can be used to reduce the spatial dimensions of the input image, by simply modifying the stride of the kernel / filter.

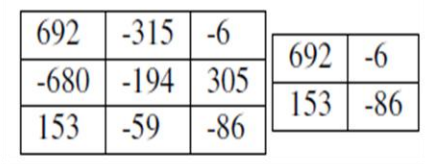


Figure 3.2.3- 2 The image after the first convolutional layer over which the filter was applied with the stride 1 (on the left) and stride 2 (on the right)

3.2.4. ZERO PADDING

As we are acquainted with the explanation of the convolutional layer, we must retain the shape, i.e. the size of the original image so that we can further employ the convolution to it. In using zero padding, we can adjust the input along the boundaries so that the size of the output image matches with the size of the input image. The amount of zeros we apply is controlled by parameter P. This technique is especially critical when we begin to observe deep convolutional neural networks architectures applied with multiple CONV layers and filters (stacked one after the other). Without zero padding, the spatial dimensions of the input image would shrink too quickly, and we will not be capable to train deep neural networks, because the input image would be too small to learn any useful pattern. In combining all of these parameters, we can calculate the size of the output image as a function of the size of the input volume (W, assuming that the input images are squared, which they almost always are), the size of the receptive field F, stride S and the amount zero padding P. In order to construct a valid CONV layer, we primarily must ensure that the ensuing equation is an integer.

$$\frac{(W - F + 2 \cdot P)}{S} + 1 \tag{3.1}$$

If it is not an integer, then the strides are set incorrectly and the neurons cannot be brought to an adequate stage to add zeros.

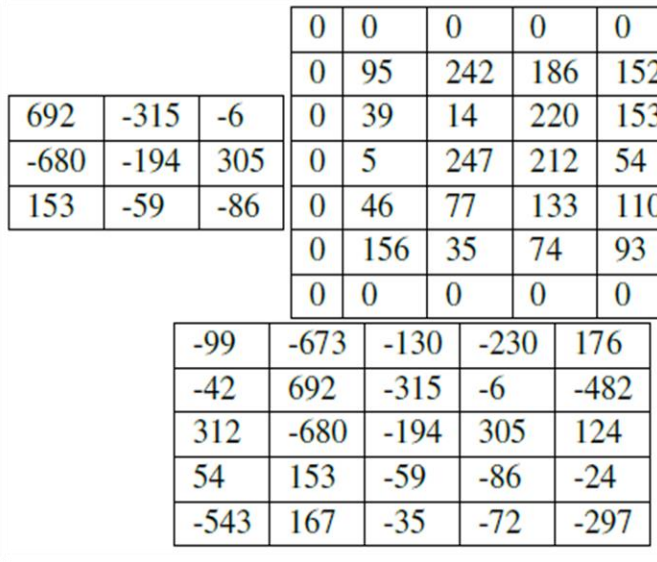


Figure 3.2.4.- 1 Zero padding method and the kernel ($F \times F = K$) i.e., filter, application of the first convolutional layer

On the Figure 3.2.4.- 1 it is clearly shown that if on the original image, which has dimensions the size of 5×5, as depicted in the Figure 3.2.3.- 1 from the previous chapter, we initially apply the zero padding it will become an image with dimensions the size of 7×7. Then, we can slide the filter the size of 3×3 and we will receive an output that has retained the original shape of the image from the input.
Briefly, the description of the CONV layer:

- ✓ Accepts the input image as $W \times H \times D$ (where the image is frequently $W = H$).

- ✓ Four parameters are needed: the number of filters K, the size of the receptive field, i.e. filter $F \times F$ ($K = F \times F$), stride S, amount of zero padding P.

✓ The output of the CONV layer is then positioned as $W \times H \times D$, where:

$$W = \frac{(W - F + 2 \cdot P)}{S} + 1 \tag{3.2}$$

$$H = \frac{(H - F + 2 \cdot P)}{S} + 1 \tag{3.3}$$

$$D = K \tag{3.4}$$

3.3. ACTIVATION LAYER

After each CONV layer within the convolutional neural network, we employ a nonlinear activation function, such as ReLU, ELU, Leaky ReLU, or some other variant. Furthermore, within this paper, we denote the layers with the activation function as ReLU for the reason that we will only implement them (otherwise it would be something else). Activation layers are technically not layers (due to the fact that there are no parameters or weights which are learned within the activation layer) and are sometimes omitted from the neural network architecture diagram, because it is assumed that the activation is immediately followed by the CONV layer.

An example of the layout of a convolutional neural network:

$$INPUT \implies CONV \implies ReLU \implies FC$$

3.4. POOLING LAYER

The objective of the pooling layer is to reduce the input images in order to lessen the computer load, memory usage, as well as, the number of parameters (thus limiting the risk of overfitting). Reducing the size of the input image makes the neural network tolerate a small image shift, i.e. the possible invariance of the location. Just as in convolutional layers, each neuron in the pooling layer is connected to the limited outputs of the neurons from the previous layer, which are located within a small rectangular receptive field.

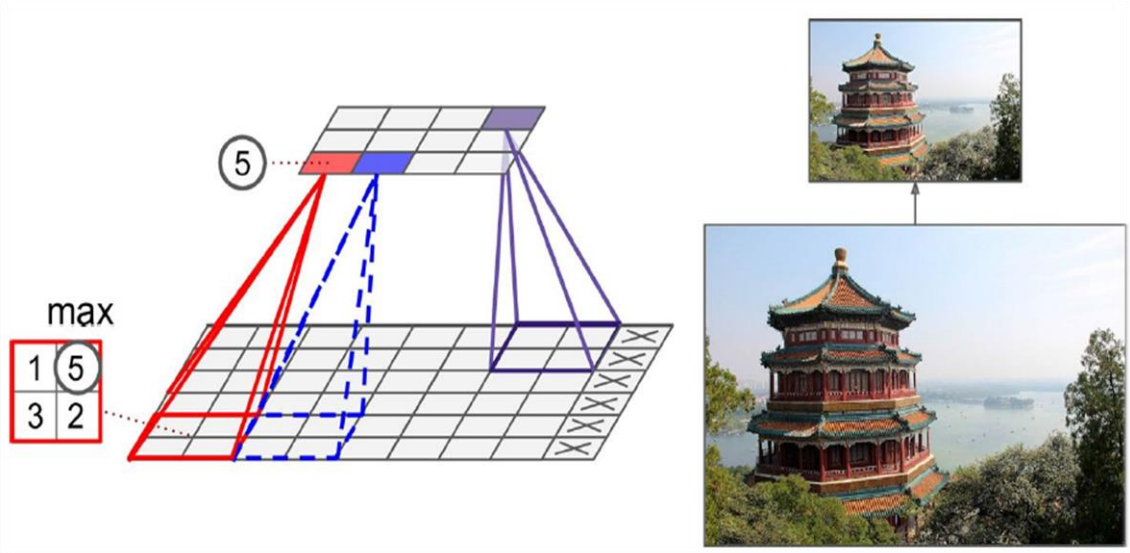


Figure 3.4.- 1 Demonstration of the pooling layer

Nonetheless, the accompanying neuron has no weights, all it does is aggregate the input using a pooling function such as maximum value or mean value. Figure 3.4.- 1 exhibits the maximum pooling layer, which is the most common type of pooling layer. In this example, we use an aggregation filter the size of 2×2, with a stride of 2, and we have no zero padding. So, only the largest input value is taken and then the filter passes through the subsequent layer, while the other inputs are discarded.

This is obviously a very destructive type of layer even with a filter the size of 2×2 that is small and a stride of 2. The output will be twice as small in both directions (so that its surface will be four times diminished) by simply dropping 75% of the input values. The pooling layer normally works on each of the input channels independently so that the output depth is the same as the input depth.

3.4.1. POOLING LAYER: MATHEMATICAL APPROACH

The pooling layer reduces the size of the image for the reason that it combines the adjacent pixels of a certain area of the image into one representative value. In order to perform operations in the pooling layer we should determine how to select pixels from the image in order to acquire a representative value. Adjacent pixels are frequently selected from a square matrix and the number of combined pixels varies from problem to problem. A representative value is habitually set as the mean value or maximum value of the selected pixels. The work of the pooling layer is surprisingly simple and we will show it in the subsequent examples.

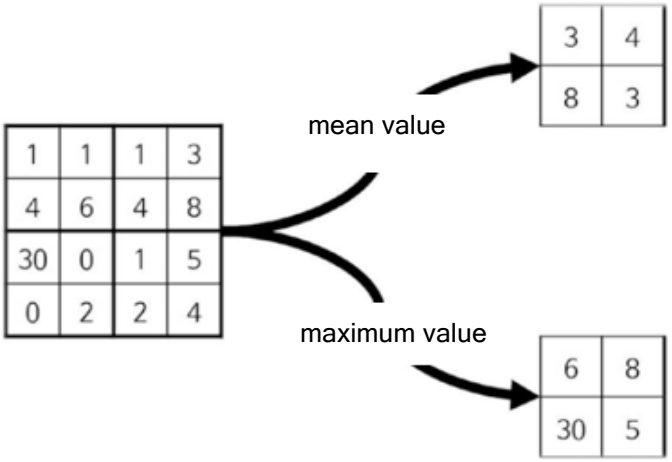


Figure 3.4.1.- 1 The pooling layer based on mean or maximum value

Let us first and foremost observe the Figure 3.4.1.- 1 on it we have an image with 4×4 input pixels. We combine the pixels of the input image into a matrix the size of 2×2 without overlapping elements. Once the input image passes through the pooling layer, it is reduced to this size, and this is achieved in this example by using either mean pooling or maximum pooling.

In the Figure 3.4.1.- 2 we can perceive an example where we used two variations of the maximum pooling. Type 1: $F = 3; S = 2$, called overlapping pooling is often applied to images with large spatial dimensions. Type 2: $F = 2,; S = 2$ which is dubbed non-overlapping pooling. This is in turn the most common type of pooling and is applied to

images with smaller spatial dimensions. For neural network architectures that accept smaller input images (in the range of 32644 pixels), $F = 2;\ S = 1$ can also be utilized.

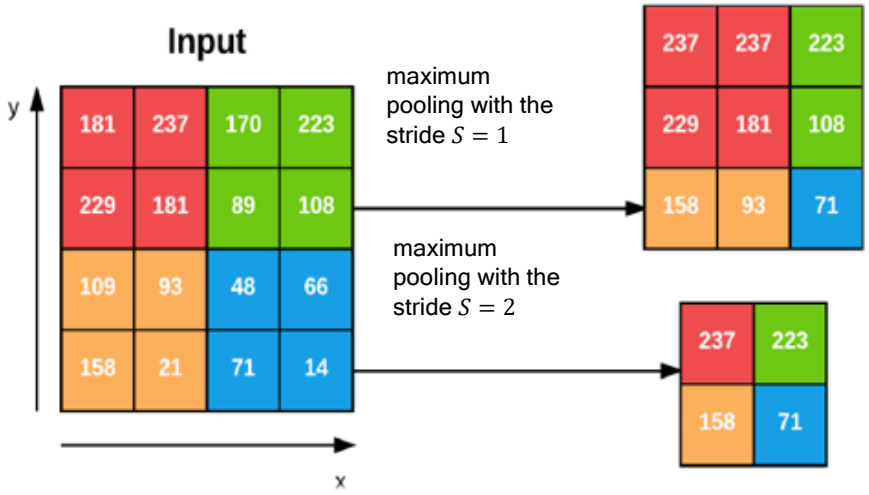


Figure 3.4.1.-2 Maximum pooling with the stride of $S = 1$ and $S = 2$

3.5. FULLY CONNECTED LAYER

Neurons in the FC (fully-connected) layers are fully connected with all activations from the previous layer, FC layers are always placed at the end of the neural network. It is quotidian to use one or two FC layers before applying the Softmax classifier, let us demonstrate the following simplified architecture:

$$INPUT \Longrightarrow CONV \Longrightarrow ReLU \Longrightarrow POOL \Longrightarrow CONV \Longrightarrow ReLU \Longrightarrow POOL \Longrightarrow FC \Longrightarrow FC$$

Further, here we apply two fully connected layers before our (default) Softmax classifier that will calculate our final output probabilities for each class.

3.6. COMPOSITION OF THE CONVOLUTIONAL NEURAL NETWORK

For simplicity's sake, each convolutional layer in the previous chapters was presented as a thin two-dimensional layer, however in reality it consists of several feature maps of equal size so it is more accurately presented in a three-dimensional space.

In the Figure 3.6.-1 we can unmistakably see that within one feature map all neurons have the same parameters (weights), however maps of different features may have different parameters. The receptive field of neurons is the same as described earlier, but extends to all maps of all previous layers. In short, the convolutional layer simultaneously applies multiple filters to its inputs, making it capable of detecting multiple functions anywhere in its inputs. Moreover, the input images also consist of several layers: one per color channel. There are three: red, green and blue (RGB). Black and white images have only one channel, but some images can have many more - for example, satellite images that capture additional light frequencies (such as infrared radiation).

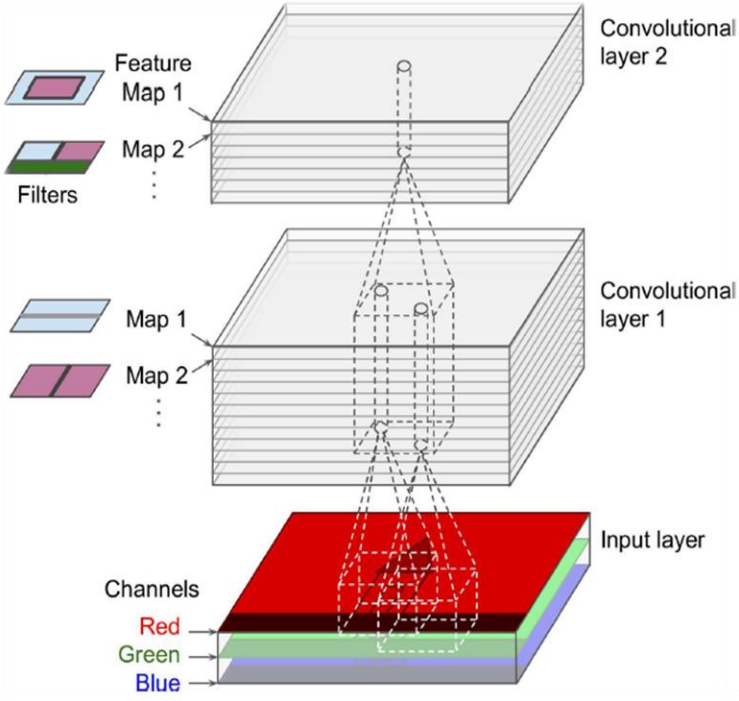


Figure 3.6.-1 Representation of a complex convolutional neural network

In particular, the neuron is positioned in the row i, the column j of the feature map k in the given convolutional layer l it is connected to the outputs of the neuron from the previous layer $l-1$, which is in the rows from $i \times s_w$ to $i \times s_w + f_w - 1$ and in columns from $j \times s_h$ to $j \times s_h + f_h - 1$, on all of the feature maps (in the layer $l-1$). We should keep in mind that all neurons that are in the same row i and column j, but in different feature maps are connected to the outputs of precisely the same neurons from the previous layer. If inside the code it is programmed to "SAME", the convolutional layer utilizes zero padding if needed.

3.6.1. MEMORY REQUIREMENTS

One of the problems with convolutional neural networks is that the convolutional layers need a huge amount of RAM (random-access memory), especially during training, because the backpropagation algorithm requires that all intermediate values be calculated during the time the information moves through the feedforward neural network. For example, let us consider a convolutional layer with filters the size of 5×5, which emits 200 feature maps the size of 150×100 with a stride of 1 and a "SAME" zero padding setting. If the input image is the size of 150×100 and if it is in color that means that it has three channels (RGB), at that juncture the number of parameters $(5 \times 5 \times 3 + 1) \times 200 = 15200$ ($+1$ corresponds to the bias) and it is quite small compared to the fully connected layer. However, each of the 200 feature maps contains 150×100 neurons, and each of these neurons must calculate the weight sum of its $5 \times 5 \times 3 = 75$ inputs: that comes to a total of 225 million multiplications. For a fully connected layer this is not so damaging, howbeit, it is still quite computationally intensive. Moreover, if the feature maps are represented by 32-bit real values then the output of the convolutional layer occupies $200 \times 150 \times 100 \times 32 = 96$ million bits (about 11.4 MB)

of RAM. However, this is just for one case. If the training batch contains 100 copies then this layer will deplete more than 1GB of RAM.

3.7. DROPOUT METHOD

Dropout / DO is actually a form of regularization that endeavours to prevent overfitting by increasing the accuracy of the validation, perhaps to the disadvantage of the accuracy of neural network training. For each mini-batch within the training set the discarded layers (dropout with a probability p) will randomly exclude inputs originating from the previous layer and at the same time extend to the subsequent layer in the neural network architecture. In the Figure 3.8.- 1 we have a demonstration of random dropout with a probability $p = 0.5$ of the connection between two FC layers for a provided mini-batch. After the back and forth propagation is calculated for the mini-batch we reconnect the breached connections and then sample to remove another series of connections. The reason we apply dropout is the necessity to reduce neural network overfitting by unambiguously changing the neural network architecture during neural network training. Random disconnection ensures that there are no nodes in the neural network that are responsible for the activation when provided a given form. Instead, dropout ensures that there are more redundant nodes that will be activated if similar inputs are presented to them — this in turn helps the model to generalize learning. The most commonly placed layers have a dropout with a probability of $p = 0.5$ between FC layers in the neural network architecture and there the final FC layer in question is assumed to essentially be a Softmax classifier:

$$\ldots . CONV \Longrightarrow ReLU \Longrightarrow POOL \Longrightarrow FC \Longrightarrow DO \Longrightarrow FC \Longrightarrow DO \Longrightarrow FC(Softmax)$$

Nevertheless, we can employ dropout with less probability, but we should always take into account whether or not it changes the situation for the better or for the worse.

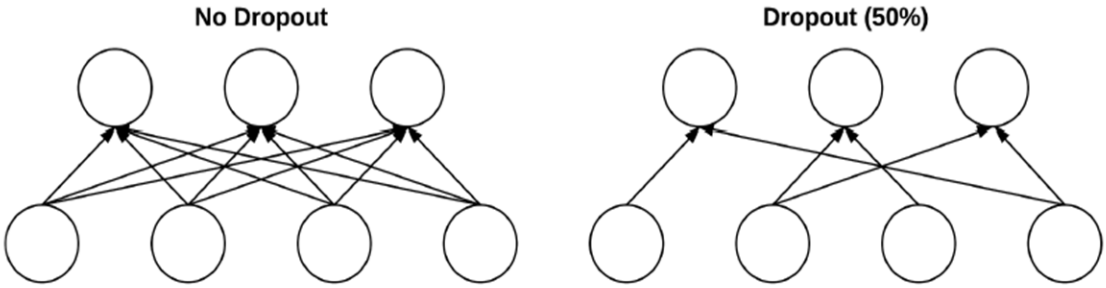


Figure 3.8.-1 An example with and without dropout

3.7.1. DROPOUT METHOD: MATHEMATICAL APPROACH

Deep neural network models have an astronomical degree of accurate classification, but they can also be overfitted very easily. Although this overfitting problem can be solved by using a very large amount of data, large data sets are not available at all times. Therefore, dropout is a cheap but powerful method of regularization.

We will currently present a dropout algorithm in the zero multiplication example for simplicity, but this method can be trivially modified in order to be adapted to work with other operations that remove parts of the neural network. For many classes of models that do not have nonlinear hidden units the weight adjustment method is very accurate. As a simple example we employ the regression of the Softmax classifier with n input data that are represented with the vector x:

$$P(Y = y|\boldsymbol{x}) = Softmax\left(\boldsymbol{w^T x} + \theta\right) \tag{3.5}$$

We can then index this into the submodel family by matrix multiplication of the inputs with the binary vector $\boldsymbol{d}$:

$$P(Y = 1|\boldsymbol{x}; \boldsymbol{d}) = Softmax\left(\boldsymbol{w^T d} \odot \boldsymbol{x} + \theta\right) \tag{3.6}$$

The predictor is then defined by renormalizing the geometric mean value for the whole set of predictors:

$$P_1(Y = y|\boldsymbol{x}) = \frac{\widetilde{P_1}(Y = y|\boldsymbol{x})}{\sum_{y'} \widetilde{P_1}(Y = y'|\boldsymbol{x})} \tag{3.7}$$

By further simplifications $\widetilde{P_1}$ is normalized, and by ignoring some constant parameters the previous equation is reduced to the following:

$$\widetilde{P_1}(Y = y|\boldsymbol{x}) = exp\left(\frac{1}{2^n} \sum_{d \in \{0,1\}^n} \boldsymbol{w^T}_{\boldsymbol{y},:} \boldsymbol{d} \odot \boldsymbol{x} + \theta\right) \tag{3.8}$$

$$\widetilde{P_1}(Y = y|\boldsymbol{x}) = exp\left(\frac{1}{2}\, \boldsymbol{w^T}_{\boldsymbol{y},:} \boldsymbol{x} + \theta\right) \tag{3.9}$$

It has been demonstrated that the interruption of work, i.e. the dropout of the parts of the neural network is much more efficient than other regularization methods (such as decay, limitation of the filter norm, and infrequent regulation of activity) and is very convenient for saving time during the neural network training.

3.8. BATCH NORMALIZATION

Batch normalization was initially introduced in the paper [14] in 2015. BN (batch normalization) layers / batch normalization layers are utilized to normalize the activation of the given input volume before conveying it to the next layer in the convolutional neural network.

If we contemplate that x is a mini-batch of activations, then we can compute the normalized $\hat{x}$ via the ensuing equation:

$$\widehat{x_i} = \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \varepsilon}} \tag{3.10}$$

For the duration of training, we can calculate μ_β and σ_β for each mini-batch β, where:

$$\mu_\beta = \frac{1}{M}\sum_{i=1}^{m} x_i \qquad \sigma_\beta^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x_i - \mu_\beta\right)^2 \tag{3.11}$$

We set ε so that it is equal to a small positive value, such as $1\mathrm{e}-7$ so as not to take the square root of zero. At the time of neural network testing we replace the mini-batch μ_β and σ_β with the average of μ_β and σ_β, which were calculated during the training process. This ensures that we can pass images through our neural network and acquire accurate predictions without the fear of the "predisposition" of μ_β and σ_β from the final mini-batch as we go through the neural network at the time of training. The normalization of the batch has been revealed to be extremely effective in reducing the number of epochs required to train the neural network. Batch normalization likewise has the added advantage of stabilizing the training, thus allowing for a greater number of learning and regulation parameters. The following question that arises is how do we distinguish whether or not we should insert batch normalization.

The batch normalization distributes the functions that represent the output from the CONV layer. Some of these characteristics may be negative and in those we will be adjusting to a zero by a nonlinear activation function such as ReLU. If the normalization is performed before the activation function we essentially include negative values within the normalizations.

Our zero focused characteristics are then passed through a ReLU where we remove any activation that is less than zero (which includes functions that may not have been negative before normalization) — this layer sequence completely defeats the purpose of applying batch normalization. Instead, if we set the the batch normalization after ReLU we shall normalize the positive characteristics without statistically linking to the characteristics that would not be present otherwise within the succeeding CONV layer.

3.9. LAYERS OF THE CONVOLUTIONAL NEURAL NETWORK AND IMPLEMENTATION

As we have already clearly established, convolutional neural networks consist of four primary layers: CONV, POOL, ReLU and FC. Taking these layers and arranging them in a certain pattern makes the CNN architecture, that is the architecture of the convolutional neural networks. The CONV and FC (and BN) layers are the only network layers that actually learn the parameters — the other layers are purely responsible for performing a particular operation. The activation layers such as ReLU and DO are not technically layers, but then again are often included in the CNN architecture.

By far the most common form of the CNN architecture is stacking several CONV and ReLU layers and succeeding them with a POOL operation. This sequence is repeated until the width and the height of the image volume are small at which point we apply one or more FC layers. Therefore, we can conclude that the most common pattern of CNN architecture is as follows:

$$INPUT \implies \left[[CONV \implies ReLU] * N \implies POOL\right] * M \implies [FC \implies ReLU] * K \implies FC$$

Here the operator $*$ denotes more than one layer. We often choose:

$$0 \leq N \leq 3$$
$$M \geq 0$$
$$0 \leq K \leq 2$$

In the usual course of events we apply deeper neural network architectures when we have many classes within the input data set and when the classification problem is quite challenging. Stacking multiple CONV layers before applying the POOL layer allows CONV layers to develop more complex characteristics before the destructive pooling is performed.

Using square inputs allows us to take advantage of the linear libraries in order to optimize algebras. Conventional sizes of the input layer include 32×32, 64×64, 224×224, 227×227 and 229×229. Secondly, the input layer should be able to split at least twice after the operation of the first CONV layer has been applied. We can do this by adjusting the size and stride of the filter. The "rule of division with two" allows the spatial inputs in our neural network to be conveniently reduced through a sample of POOL operations, i.e. in an efficient manner.

In particular, CONV layers should utilize smaller filter sizes, such as 3×3 и 5×5. Small filters the size of 1×1 are depleted in order to learn local functions, but only in more advanced neural network architectures.

Larger filter sizes such as 7×7 and 11×11 can be used as the first CONV layer in the neural network (so as to reduce the spatial size of the input provided that the images are larger than 200×200 pixels). However, after this initial CONV layer, the size of the filter should drop drastically otherwise we will reduce the spatial dimensions of the image too quickly.

4. TRAINING OF THE CONVOLUTIONAL NEURAL NETWORK: CIFAR–10

In this chapter we shall explain in detail the implementation of the code on the CIFAR-10 data set and the manner in which we arrive to the solution, i.e. the ability to recognize images based on class.
Furthermore, we will also employ all the theoretical knowledge from the previous chapters in the finest conceivable manner by breaking down the code and offering a detailed explanation of that same code. In doing so in this chapter, we will be allowed to present clarifications later on for other types and techniques of implementation such as $LeNet$ and feature extracting when utilizing other data sets.

4.1. SHALLOW CONVOLUTIONAL NEURAL NETWORK

Let us begin the implementation of the shallow convolutional neural network / shallow CNN. First, in the following manner by demonstrating the skeleton of the directory and subdirectory and explaining the codes within them. This type of code layout permits an easy correlation to the main code. Within the $utilites$ directory there are three subdirectories the $dataset\_loader,\ nn$ and $preprocessing$:

1. $dataset\_loader$ (loading the data set)
2. neural network/nn (neural network training technique)
3. $preprocessing$ (preprocessing of images i.e. input data set)

animals
utilities
cifar-10-python.tar
shallownet_animals.py
shallownet_cifar10.py

Table 4.1.-1 – The main code $shallownet\_cifar10$, which contains the data set that we necessitate within the $.tar$ extension, as well as the subdirectory $utilities$ (green for the python script, blue for the data sets i.e. images, yellow for the subdirectories)

4.1.1. DATA PREPROCESSING

The first part of the code that we process is called the $simple\_preprocessor$, in Table 4.1.1.- 1 the path to this subdirectory and the code itself is illustrated. Different algorithms in machine learning, as well as deep learning employ this kind of code, even convolutional neural networks.

animals	__pycache__	__pycache__
utilities	dataset_loader	__init__.py
cifar-10-python.py	nn	imagetoarray_preprocessor.py
shallownet_animals.py	preprocessing	simple_preprocessor.py
shallownet_cifar10.py	__init__.py	

Table 4.1.1.- 1 - Implementation of a $simple\_preprocessor$ and an $imagetoarray\_preprocessor$ (red directory developed during the code execution in command prompt)

As we have previously mentioned convolutional neural networks require that their input data set, i.e. the images be of fixed size, i.e. that the matrix vector be constant. This means that our images must be preprocessed and reduced to have identical widths and heights. This image preprocessor actually reduces the size, i.e. image dimensions in so ignoring the image proportion.

As a part of the $SimplePreprocessor$ function we insert the $cv2$ library, which was actually developed within the $OpenCV$ (which is used for digital image processing), that is in charge of importing images and processing them in the way that we have previously elucidated. It requires three arguments, one of which is optional and is utilized as a command for selecting the interpolation algorithm that we will apply to change the image size (later on we will get acquainted with other types of algorithms in Chapter 8).
It is concluded that this preprocessor is by definition very simple — all we do is receive the image input, set it to a fixed dimension and then return it back to the code.

4.1.2. PREPROCESSOR: IMAGE TO ARRAY

In practice the backpropagation algorithm / BP algorithm is very demanding to apply and not only because of the errors that occur in the calculation of the gradient descent method, but it is very complicated to make such an algorithm to be efficient and not contain any additional library for optimization. Therefore, instead of implementing the code (which we will certainly depict in a simple example in Chapter 6 of this paper) we frequently use libraries such as $Keras$, $TensorFlow$ and $mxnet$ with this algorithm. These libraries already have the correct implementation of the backpropagation algorithm and utilize optimization strategies.

In order to best explain how the $imagetoarray\_preprocessor$, essentially works, we must initially explain what the $keras.json$ file is and how it affects the input of the input data and images, and also how it influences the convolutional neural network itself. Within this file there are two very important configurations, $image\_data\_format$ and $backend$. The $Keras$ library automatically utilizes $TensorFlow$ for a numerical calculation within the code. There is also an option to use theano instead of the $TensorFlow$ library.

$Keras$ does a fantastic job and $backend$, in so it allows you to write deep learning code that will be compatible with any $backend$, and we will more often than not find that both computational backgrounds deliver the equivalent result. Finally, we have an $image\_data\_format$ that can accept two values $channel\_last$ or $channel\_first$.

As we have already stated, the images are loaded using the $OpenCV$ library and are presented in the subsequent order ($rows, columns, channels$), and this is termed $channel\_last$ in the $Keras$ library (because the channels are the last in the series). Nevertheless, we can set the $image\_data\_format$ to be represented in such a manner that our input data i.e. images be placed in the following order ($channels, rows, columns$), and this is termed $channel\_first$ in the $Keras$ library (because the channels are the first in the series). The problematic behind this and why we introduce $imagetoarray\_preprocessor$ arises in that $Keras$ was first compatible with the theano library, and not with the $TensorFlow$ library.

Over the course of time and due to the great implementation possibilities $TensorFlow$ has been implemented in such a way that the $Keras$ library accepts the input image, and then replaces the places with channels in a correct manner. Further, we will now implement another image preprocessor entitled $imagetoarray\_preprocessor$ that accepts input as an image and then converts it to a $NumPy$ array that $Keras$ can work with.

We utilize the $ImageToArrayPerprocessor$ function (which we previously imported from the $Keras$ library) it primarily accepts an optional parameter dubbed $data\_format$, sets its value to zero, which actually induces the need to use the $keras.json$ format. We also need to specify the $channel\_last$ and the $channel\_first$, Nevertheless, within the code it is the most beneficial to let it be implemented by the $Keras$ library, as well as, to determine what is utilizes based on the image file configuration. Finally, when it processes the image it returns a novel $NumPy$ array with properly arranged channels. The advantage of defining a class for handling this type of image preprocessing is that when we call this function in the main code it will sequentially pass over all images that will be entered as input data and perform this type of adjustment.

Ultimately, it is necessary to write pieces within this directory inside the $\_\_init\_\_.py$ script to call these two main codes for image preprocessings. This way we will create a file that will be directly linked to the main directory and our code will be capable to easily extract the necessary functions. We will then repeat this further within all subdirectories of our code since in this manner we can obtain a clear code concept the easiest. Correspondingly, some of the neural networks we created can then be tested on another data set with minor modifications within the main code and inside the code where we use methods such as $LeNet, VGGnet, Shallownet$ (the case of this Chapter).

4.1.3. LOADING OF THE INPUT DATA SET

animals	__pycache__	__pycache__
utilities	dataset_loader	__init__.py
cifar-10-python.py	nn	simple_dataset_loader.py
shallownet_animals.py	preprocessing	
shallownet_cifar10.py	__init__.py	

Table 4.1.3. -1 – Implementation of a $simple\_dataset\_loader$

At the present, we will explain the implementation of the program for loading the images from the disk, this preprocessor the $simple\_dataset\_loader$ will permit us to quickly load the images and along the way process the database from the disk, all the while allowing us to move quickly while working with the classification of images.

Here we employ the ensuing libraries os, $cv2$, and $NumPy$. The os library is used to extract the names of the images from a given directory with a data set, while the $NumPy$ library is used for numerical calculations, i.e. processing. Inside the $SimpleDatasetLoader$ we have two functions. The first function that deals with extracting the image and translating it through the appropriate preprocessors (in this case, only one preprocessor comes to mind and it was explained in the previous Chapter). The second function is utilized to initialize the list of input data, i.e. images together with their

corresponding classes, in addition within it there is a loop function that goes over all of our images.
In all of this, we should have in mind a precaution that the assumption is that our data sets are organized on the disk in accordance with the following directory structure:
`/the_name_of_the_folder_where_the_images_are_located/class/the_name_of_an_image.jpg`

Established based on this hierarchical directory structure we can keep data sets tidy and organized. Once all of the images are called to the function, we can perform preprocessing if necessary. The manually formed set of images within the Flowers-17 data set is organized exactly as described. It should be noted that this procedure for loading data assumes that all of the images in the data set can fit into the main memory at the same time. In this paper this is the case, but with much larger data sets of images this is sometimes impossible to do on a single computer.

4.1.4. SHALLOW NEURAL NETWORK

$ShallowNet$ (shallow neural network) architecture contains only a few layers — the entire neural network architecture can be represented as follows:

$$INPUT => CONV => ReLU => FC$$

This simple architecture of a convolutional neural network will allow us to later differentiate between the importance of the layers within the neural network and how many there should be and how the filters should be arranged within them in order for the accuracy to be as great as possible.

animals	__pycache__	__pycache__	__pycache__
utilities	dataset_loader	cnn	__init__.py
cifar-10-python.py	nn		shallownet.py
shallownet_animals.py	preprocessing		
shallownet_cifar10.py	__init__.py		

Table 4.1.4.- 1 – Implementation of a $shallownet$

This part of the code together with the previous one forms a skelet without which the main code could not function properly. Further, within it we use the subsequent classes from the $Keras$ library $Sequential$, $Conv2D$, $Activation$, $Flatten$, $Dense$ and K. The $Sequential$ class implies that our neural network will be a feedforward neural network and that the layers will be appended sequentially to the classes of the images, one on top of the other. The $Activation$ class deals with the application of the activation function to our input data set.

The $Flatten$ class takes our multidimensional matrix and "flattens" it into a one-dimensional array, before we enter the input data set into the $Dense$ class.
The $Dense$ class is utilized to implement FC (fully connected) layers. The K class is introduced as a $backend$ and is necessary in order for the $Keras$ library to be compatible with the $TensorFlow$ library. The main function will accept a certain number of parameters and construct a neural network architecture and then return it to the call function. In this case, the compilation method requires four parameters for the width (i.e. the number of columns in the matrix), the height (i.e., number of rows in the matrix), the depth (i.e., number of image channels — for black and white images it is 1, whereas for

images in color which is the case here, that number is equal to 3). It is a common practice that almost every convolutional neural network has a section where it first assumes that the channels are set as the last and at the same time checks whether they are set as the "first channel", and if so, performs an adjustment. This is completed in order to ensure that the convolutional neural network will work no matter how the user sets the image channels.

Let us now return briefly to the CIFAR-10 data set. Accordingly, this data set consists of 60000 images whose vector dimension is $3072\ (32 \times 32 \times 32)$, with a total of 10 classes. Since in this example we have only one convolution layer, it contains $K = 32$ filters and each of them has an $F \times F = 3 \times 3$ filter shape. We insert "same padding" so as to ensure that the output from the convolution layer is the same as the input and we employ the ReLU activation function. We arrange the images and further reduce the number of nodes at the output so as to be the same as the number of classes, that is in this case 10. Finally, the activation function Softmax is applied and it will give us the probability of membership to each class.

4.1.5. IMPLEMENTATION ON CIFAR-10 DATA SET

It is time to print a command for the main code that will link all of the previous codes, it will be dubbed the $shallow\_cifar10$ and we have previously extracted the data and the functions from all its subdirectories. First of all, it should be noted that the auxiliary library for Cifar-10 is used here, which is actually built into the $Keras$ library and performs complete image preprocessing. The implementation was done without this library based on the previously mentioned codes and the result was almost identical. Calling the Cifar-10 data set, automatically loads a set of data from the disk, which has already been segmented into a data set for training and testing. If this is the first time we are calling the Cifar-10, the $cifar10.load\_data()$ function will download the data set for us. This file has 170 MB and once the file is downloaded it will not need to be downloaded again. If there is a problem when connecting to the internet, as can happen if the internet is not fast enough (for example with a jupyter online notebook — this happens more often than not) we can insert it directly into the python directory located on the disk `C:/users/our_name/sckit_learn_data/mldata`, and thus solve it.

At this point, for the first time in this paper we encounter the $LabelBinarizer$ library used for "$one\ hot\ encoding$", which is nothing more but the representation of integers as vector notations. "$One\ hot\ encoding$" transforms categorical labels from an integer to a vector.

"One hot encoding"

Each data point in the Cifar-10 data set has an entire notation in the range $[0, 9]$, one for each of the possible ten input quantities. Let us suppose that the label in place number 8 indicates that the corresponding image contains an airplane. Accordingly, first we have to transform these integer labels into vector labels, where the index vector for the label will be set to 1, and everything else will be set to 0.

$$[0, 0, 0, 0, 0, 0, 0, 1, 0]$$

This "$one\ hot\ encoding$" process, where input integers become vector labels will be performed first on the training data set and then on the testing data set.

The optimization method utilized here is the stochastic gradient descent method, i.e. the SGD method (previously theoretically explained). We initialize the optimization employing the learning rate $\alpha = 0.01$. The $ShallowNet$ architecture was introduced as follows, to have a width and height of 32 pixels, along with a depth of 3 — this implies that the images are 32×32 pixels with three channels (i.e., in color, RGB channels are the ones in question). Given that the Cifar-10 data set have ten class labels, we set the classes to 10. We use the previously defined cross-entropy function as our loss function. The $ShallowNet$ neural network will be trained with 40 epoch utilizing mini-batches the size of 32 (which actually means that 32 images will be presented to the neural network at the same time, and full feedforward and backpropagation be completed in order to update the neural network parameters).

	precision	recall	f1 score	support
airplane	0.63	0.60	0.62	1000
automobile	0.72	0.69	0.71	1000
bird	0.37	0.56	0.44	1000
cat	0.50	0.26	0,34	1000
deer	0.60	0.34	0,44	1000
dog	0.41	0.63	0.50	1000
frog	0.75	0.59	0.66	1000
horse	0.60	0.69	0.65	1000
ship	0.71	0.72	0.72	1000
truck	0.66	0.66	0.66	1000
accuracy			0.57	10000
macro avg	0.60	0.57	0.57	10000
weighted avg	0.60	0.57	0.57	10000

Table 4.1.5.- 1 - Demonstration of the accuracy and the loss over time on the training data set and the testing data set in the form of results that are depicted within the Command Prompt

The table clearly demonstrates the accuracy of our trained convolutional neural network, let us first and foremost clarify what each column represents.

precision

Precision is the ability of a classifier not to denote an image as a positive if it is actually negative, i.e. not to classify it in the wrong class. For each class, the image is defined as the ratio of all true positives and the sum of true positives and false positives. We can define this in another way; this is the ability of the classifier not to denote positively a data that is negative.

recall

Recall is the ability of a classifier to find all of the positive images. For each class, the image is defined as the ratio of all true positives and the sum of true positives and false negatives. We can define this in another way; this is the ability of the classifier to find all the positive data.

f1 - score

f1-score is a weighted harmonic mean of precision and return, such that the best result is 1.0 and the worst 0.0. As a general rule, f1 scores are lower than accuracy measures because they include accuracy and return within their computation. According to the "rule of thumb", the sum of the average of f1-score should be used to compare the model of the classifier, instead of the global accuracy.

support

Support is the number of actual class occurrences in the specified data set. Unbalanced support within a training data set may indicate structural insufficiencies in reported classifier results and may also indicate the need for stratified checks or rebalances. Support does not change between models, but the evaluation process does, i.e., it is analyzed.

Figure 4.1.5.- 2 - Display of the accuracy and the loss over the course of time on the training data set and testing data set

The conclusion is that our convolutional neural network is trained for about 26 seconds on average, yet we have in mind that the CPU is utilized singularly here, and not the GPU, although it would be much faster to employ the GPU. It is also clear to us that we have obtained an accuracy of about 60%. Our loss has been decreasing all the time when it comes to the training data set, nevertheless we observe a clear divergence when testing on a validation data set for loss and accuracy. This probably happened because of the learning rate α , which is quite large, and for the reason that we did not use some of the regularization methods that would have prevented this. It is rather clear that we can see this from the final graph, as well as, in the Command Prompt window during the neural network training process. We unambiguously notice the problem on the testing data set after only 5 epochs.

It is evident from Figure 4.1.5- 2 that our convolutional neural network has an overfitting problem. So, the optimal solution is to regularize it by introducing more layers (which we will explain afterwards in the subsequent Chapter) and prevent this as best we can.

4.2. DEEP CONVOLUTIONAL NEURAL NETWORKS

The $VGGNet$ neural network was first introduced in paper [15] which was introduced by Simonyan and Zisserman in 2014. The main contribution of their work was that they demonstrated that the neural network architecture with very small filters (type 3×3 filters) can be trained and implemented at very large depths (16 to 19 layers) and then have a outstanding classification on different data sets.

Formerly, neural network architectures used a combination of altered filter sizes in the deep learning literature. The first layer of the convolutional neural networks commonly included filter sizes between7×7 and 11×11. From there, the size of the filters was progressively reduced to 5×5, and finally, only on the deepest layers of the neural network the 3×3 filters were employed. The $VGGNet$ neural network was more exceptional in that it used only 3×3 filters in its whole architecture.

In this section, we will use batch normalization / BN and dropout / DO, which we have previously theoretically explained, and monitor their impact on the neural network performance. We will also implement a neural network that resembles the $VGGNet$ in structure but is far simpler.

Let us first summarize what makes the $VGGNet$ neural network:

1. All of the CONV layers in the neural network use only 3×3 filters.
2. Stacking multiple sets of $CONV \Rightarrow ReLU$ layers (where the number of consecutive $CONV \Rightarrow ReLU$ layers usually increases as we move deeper through the neural network) and of course all this implementation is done before applying the POOL operation.

Meanwhile, in the previous chapter in the shallow network we employed a series of layers $CONV \Rightarrow ReLU \Rightarrow POOL$. Nevertheless, we will now place multiple $CONV \Rightarrow ReLU$ layers before applying one POOL layer. This permits the neural network to learn more plentiful functions from the CONV layers before lowering the spatial input size of the image matrix via the POOL operation.

Therefore, our neural network consists of three groups of layers $CONV \Rightarrow ReLU \Rightarrow CONV \Rightarrow ReLU \Rightarrow POOL$, followed by a set of layers $FC \Rightarrow ReLU \Rightarrow FC \Rightarrow Softmax$. The first two $CONV$ layers will learn 32 filters the size of $F \times F = 3 \times 3$, the second two $CONV$ layers will learn 64 filters the size of $F \times F = 3 \times 3$ and lastly the third $CONV$ layer will learn 128 filters, also the size of $F \times F = 3 \times 3$. Our $POOL$ layers will perform maximum pooling over 2×2, with a stride $S = 2$.

We will also insert layers for batch normalization / BN after the activation function, along with the DO / dropout layers that we introduce after the POOL and the FC layers.

Yet again, we note that the layers of the batch normalization and dropout layers are included in the neural network architecture, based on the rules we explained earlier in detail in Chapter 3. Applying the batch normalization will help reduce the neural network overfitting effects and will increase the classification accuracy on the Cifar-10 data set.

Mathematical approach:

* The first convolutional layer:

$$INPUT\ (32 \times 32 \times 3)\ \Longrightarrow$$
$$\Longrightarrow\ [\ CONV\ (32 \times 32 \times 32)\ \Longrightarrow\ ReLU\ (32 \times 32 \times 32) \Longrightarrow\ BN\ (32 \times 32 \times 32)\ ] \times 2\ \Longrightarrow$$
$$\Longrightarrow\ POOL\ (16 \times 16 \times 32)\ \Longrightarrow\ DO\ (16 \times 16 \times 32)\ \Longrightarrow$$

* whereby here 32 filters the size of 3×3 act on the CONV layer, and stride 2×2 acts on the POOL layer.

* The second convolutional layer:

$$\Longrightarrow\ [\ CONV\ (16 \times 16 \times 64)\ \Longrightarrow\ ReLU\ (16 \times 16 \times 64) \Longrightarrow\ BN\ (16 \times 16 \times 64)\ ] \times 2\ \Longrightarrow$$
$$\Longrightarrow\ POOL\ (8 \times 8 \times 64)\ \Longrightarrow\ DO\ (8 \times 8 \times 64)\ \Longrightarrow$$

* whereby here 64 filters the size of 3×3 act on the CONV layer, and stride 2×2 acts on the POOL layer.

* The third convolutional layer:

$$\Longrightarrow\ [\ CONV\ (8 \times 8 \times 128)\ \Longrightarrow\ ReLU\ (8 \times 8 \times 128) \Longrightarrow\ BN\ (8 \times 8 \times 128)\ ] \times 2\ \Longrightarrow$$
$$\Longrightarrow\ POOL\ (4 \times 4 \times 128)\ \Longrightarrow\ DO\ (4 \times 4 \times 128)\ \Longrightarrow$$

* whereby here 128 filters the size of 3×3 act on the CONV layer, and stride 2×2 acts on the POOL layer.

$$\Longrightarrow\ FC\ (512)\ \Longrightarrow\ ReLU\ (512) \Longrightarrow\ BN\ (512)\ \Longrightarrow DO\ (512)\ \Longrightarrow$$
$$\Longrightarrow FC\ (10)\ \Longrightarrow\ Softmax\ (10)$$

4.2.1. A MORE COMPLEX NEURAL NETWORK RESEMBLING THE VGG_NET

utilities	__pycache__	__pycache__	__pycache__
minivggnet_cifar10.py	datasets	cnn	__init__.py
	nn		minivggnet.py
	preprocessing		
	__init__.py		

Table 4.2.1.- 1 – Implementation of a $minivggnet$

Whilst forming a complex convolutional neural network, we utilize identical libraries almost the same as for a shallow network, with a couple of additions.

The libraries we use are $Sequential$, $BatchNormalization$, $Conv2D$, $MaxPooling2D$, $Activation$, $Flatten$, $Dropout$, $Dense$ and K. In the subsequent explanations, we shall only be focusing on the modifications.

One of the main differences is the introduction of the --chanDim variable, which is a channel dimension index. Batch normalization works through channels, so in order to apply the BN, i.e., batch normalization, we need to know through which axis it is necessary to normalize. Further, setting to channel_dim = -1 implies that the index is the dimension of the channel, which is the last in the form of an entry (i.e., the last channel order). However, if we utilize the channels that are set as first, we need to update the input_shape and поставимо channel_dim = 1, because the channel dimension is now the first entry in the matrix.

Primarily, we define a CONV layer with 32 filters, each of which has a 3×3 filter size. Then we apply the ReLU activation function, which is then implemented through batch normalization and positioning relative to zero is accomplished. However, instead of applying the POOL layer and thus reducing the spatial dimensions of our input, we instead apply two more layers $CONV \implies ReLU \implies BN$, which in turn allows our network to learn better features, and thus this is also a common solution for deep learning. We use $MaxPooling2D$ with a stride size of 2×2. Given that we do not set the stride explicitly, our neural network assumes that it is equal to the maximum pooling size of 2×2. Dropout with a probability of $p = 0.25$ is employed, which means that the node from the POOL layer will be randomly disconnected from the next layer with a probability of 25% during neural network training. We utilize dropout in order to reduce the effects of the neural network retraining. Then sets of 64 filters (all the size of 3×3) are loaded into the neural network. It is common to increase the number of filters, because the spatial size of the input decreases as we move deeper through the neural network. Finally, sets of 128 filters (all the size of 3×3) are loaded into the network.

In order to implement the fully connected / FC layer, we must first convert a multidimensional representation to a one-dimensional list. This is done by utilizing the operation $Flatten$. Then we have the $Dense$ layer, that will use the same number of nodes 512, because that is how much nodes our fully connected layer has. Finally, we apply the ReLU activation function and dropout, increasing the probability to about 50% - we usually have a dropout with a probability $p = 0.5$ that is inserted between the FC layers. Then, again, we have the $Dense$ layer where we will use the same number of output nodes as the number of classes. Subsequentially, on it we infiltrate the Softmax activation function, which will give us the probability of membership to each class.

4.2.2. IMPLEMENTATION ON CIFAR-10 DATA SET

At the moment we are familiar with a lot of the elements within the Chapter 4.1.5, in the following proclamations we will skip the parts of the code that have been clarified already and explain only the parts of the code that are new to us.

As part of the implementation of this neural network, we also use the $argparse$ module, which automatically generates messages for help and use, and reports an error, if it is necessary, in our code. The $Parser$ is formed and a new parameter is adjoined with

$add\_argument()$. Our code requires only one switch --output, that represents the path of the output trained data, i.e., images, as well as, the loss function.
We will be utilizing SGD as our optimizer with the learning rate set to $\alpha = 0.01$ and the momentum $\eta = 0.9$. Setting the $nestrov = True$ indicates that we want to apply the Nesterov acceleration gradient to the SGD optimizer.

The decay parameter is one type of optimizer. This argument is depleted in order to slowly reduce the learning rate over the course of time during the training process of the neural network. Decreasing the learning rate is useful for reducing the impact of the neural network overfitting and for achieving superior classification accuracy, for the reason that the lower the learning rate, the less weight updates there will be. It is common for the decline to divide the initial learning rate by the total number of epochs. In this case, we are training a neural network with 40 epochs, with an initial learning rate of $\alpha = 0.01$.

The decay can essentially be represented as $decay = 0.01 \,/\, 40$. The convolutional neural network will be trained with 40 epochs utilizing the mini-batch the size of 64 filters (which in turn means that 64 images will be presented to the neural network at the same time, and a feedforward and backpropagation will be done in order to adjust the neural network parameters).

	precision	recall	f1 score	support
airplane	0.89	0.83	0.86	1000
automobile	0.95	0.89	0.92	1000
bird	0.82	0.73	0.77	1000
cat	0.73	0.65	0,69	1000
deer	0.76	0.86	0,80	1000
dog	0.78	0.78	0.78	1000
frog	0.82	0.93	0.87	1000
horse	0.90	0.89	0.90	1000
ship	0.90	0.93	0.91	1000
truck	0.89	0.92	0.90	1000
accuracy			0.84	10000
macro avg	0.84	0.84	0.84	10000
weighted avg	0.84	0.84	0.84	10000

Table 4.2.2.- 1 - Demonstration of the accuracy and the loss over time on the training data set and the testing data set in the form of results that are depicted within the Command Prompt with three layers and with batch normalization / BN

In using the batch normalization / BN we were able to obtain an accuracy of 84%. The training lasted about 6 minutes per epoch. On the other hand, the implementation of the neural network with the batch normalization is more stable. Although both the loss and the accuracy of the training data set begin to balance after the 35th epoch, there is no drastic overfitting of the neural network, accordingly we observe this to be quite a solid solution. (Table 4.2.2.-1 and Figure 4.2.2.-2).

In the next attachment, we will explain what happened by removing 3 layers within the neural network and omitting the batch normalization, and what lead us to the conclusion that this type of neural network training with an adequate regularization and depth is still more stable and suitable.

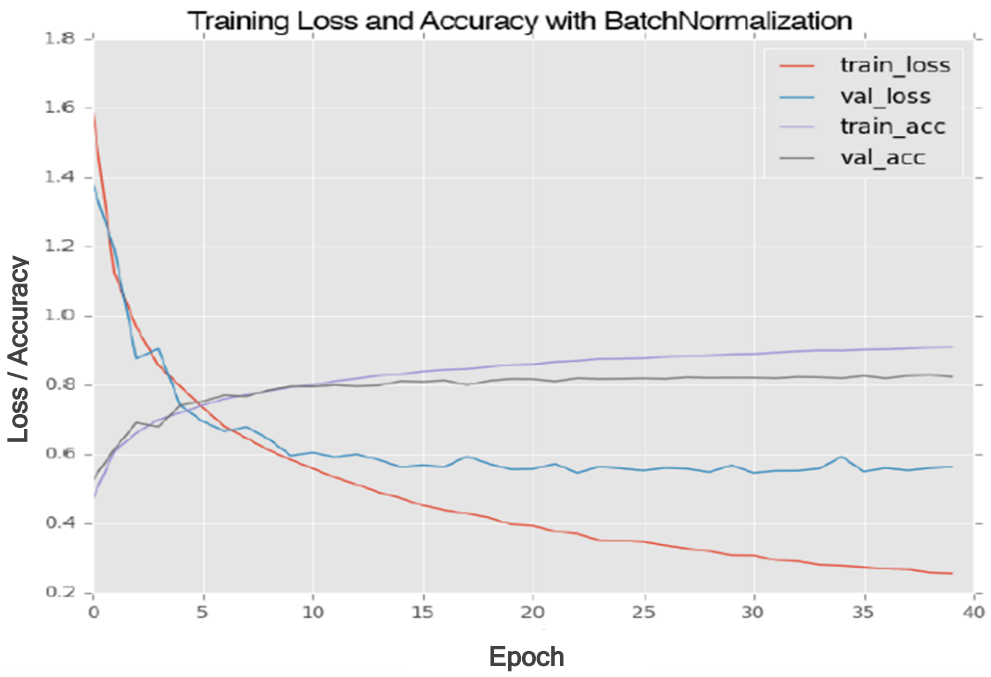


Figure 4.2.2.-2 - Display of the accuracy and the loss over the course of time on the training data set and testing data set with batch normalization

	precision	recall	f1 score	support
airplane	0.86	0.85	0.85	1000
automobile	0.93	0.92	0.93	1000
bird	0.79	0.72	0.75	1000
cat	0.70	0.66	0,68	1000
deer	0.77	0.86	0,81	1000
dog	0.76	0.76	0.76	1000
frog	0.85	0.90	0.88	1000
horse	0.89	0.88	0.88	1000
ship	0.91	0.91	0.91	1000
truck	0.88	0.91	0.89	1000
accuracy			0.84	10000
macro avg	0.83	0.84	0.83	10000
weighted avg	0.83	0.84	0.83	10000

Table 4.2.2.- 3 - Demonstration of the accuracy and the loss over time on the training data set and the testing data set in the form of results that are depicted within the Command Prompt with three layers and without batch normalization / BN

Once, we disconnected the batch normalization from the code, the accuracy was 83%, and the training was reduced to an average of 3 minutes per epoch. It is observed that the loss for the neural network begins to increase after the 30th epoch, which in turn indicates that the neural network is overfitting on the basis of the training data. This can also be clearly perceived from the accuracy of the test data set which became quite saturated by the time it reached the 25th epoch (Table 4.2.2.- 3 and Figure 4.2.2.- 4).

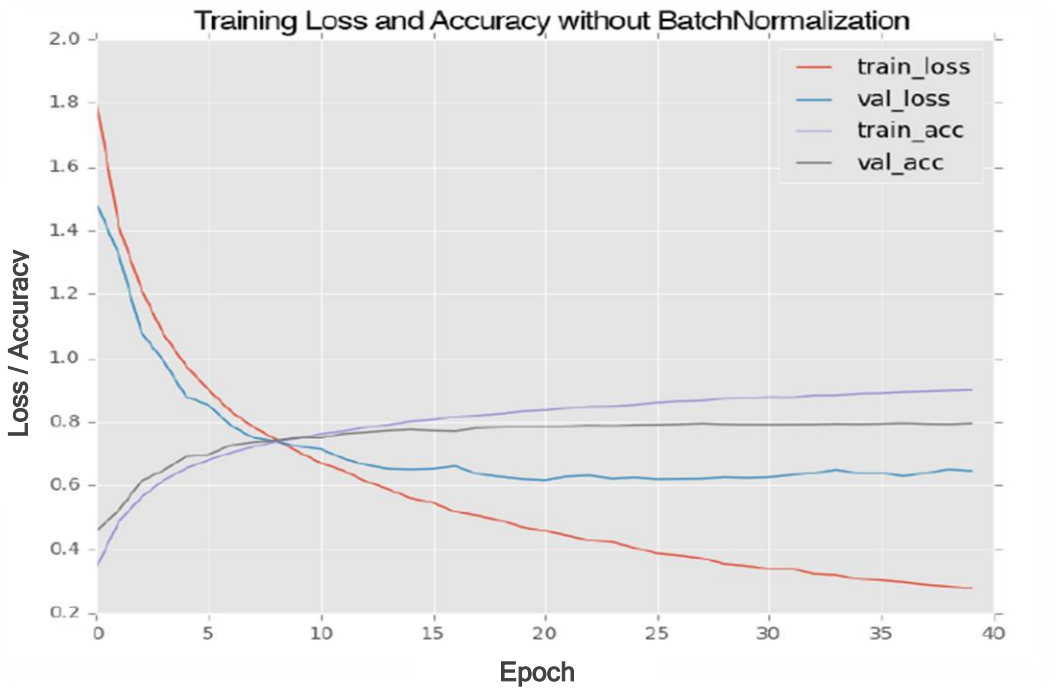


Figure 4.2.2.- 4 - Display of the accuracy and the loss over the course of time on the training data set and testing data set without batch normalization

When removing the third layer and the batch normalization, the performance is reduced to 78%, and the training per epoch is reduced to an average of 1 minute. It is observed that the loss for the neural network, without the batch normalization, begins to increase after the 26th epoch, which designates that the neural network is overfitting on the basis of the training data set. All of this can also be perceived based on the divergence of the loss on the testing data and the training data, since they have an increasing generalization gap (Table 4.2.2.- 5 and Figure 4.2.2.- 6).

	precision	recall	f1 score	support
airplane	0.83	0.80	0.82	1000
automobile	0.89	0.88	0.90	1000
bird	0.74	0.69	0.71	1000
cat	0.64	0.57	0,61	1000
deer	0.73	0.81	0,78	1000
dog	0.69	0.72	0.70	1000
frog	0.81	0.88	0.82	1000
horse	0.85	0.83	0.84	1000
ship	0.90	0.86	0.89	1000
truck	0.83	0.89	0.86	1000
accuracy			0.78	10000
macro avg	0.78	0.78	0.78	10000
weighted avg	0.78	0.79	0.78	10000

Table 4.2.2.- 5 - Demonstration of the accuracy and the loss over time on the training data set and the testing data set in the form of results that are depicted within the Command Prompt with two layers and without batch normalization / BN

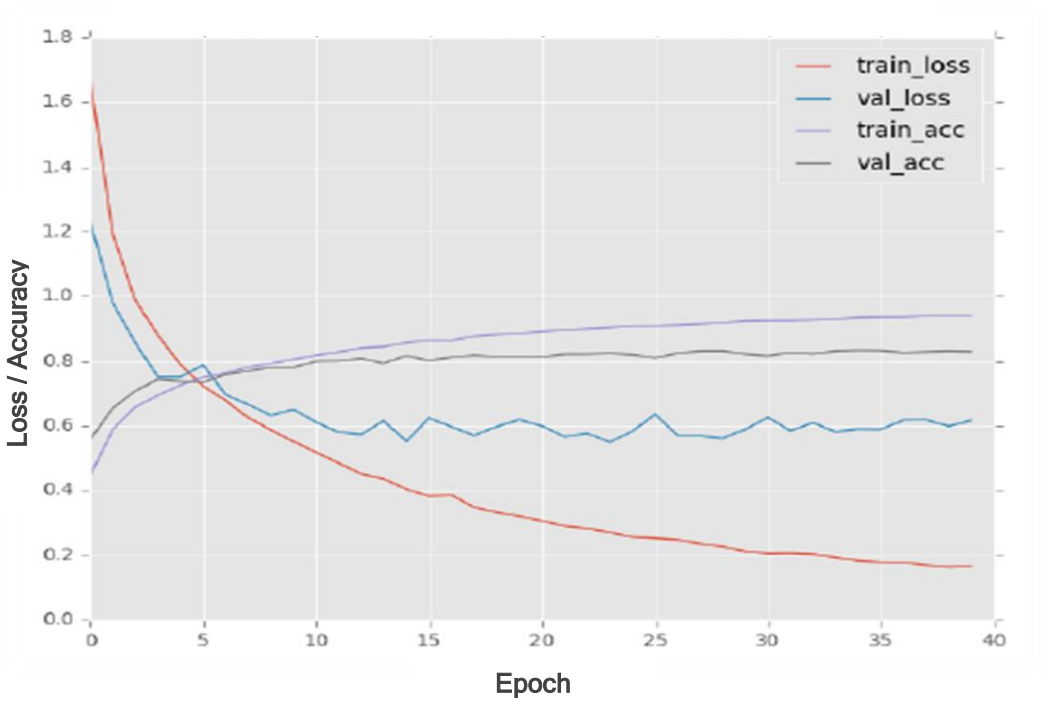


Figure 4.2.2.-6 - Display of the accuracy and the loss over the course of time on the training data set and testing data set without batch normalization and with 2 hidden layers

4.3. LEARNING RATE SCHEDULES WITH MINIVGGNET

The method of adaptive learning rates indicates adjusting our learning rate after each epoch. This way we can reduce the loss, increase the accuracy, and even in certain situations reduce the total amount of computational time it takes to train the convolutional neural network.
The simplest and most widespread methods of adaptive learning rate schedulers are those that progressively reduce the learning rate over the course of time.
Let us recall the weight adjustment formula from Chapter 3 (2.18):

$$W += -args[\alpha] * gradient \tag{4.1}$$

Let us bring to mind that the learning rate controls the "step" we take along the slope of the gradient descent. Elevated values of the learning rate mean that we take larger steps, while lower values mean that we take small steps — if the learning rate α is zero, the neural network cannot make any steps at all (since the gradient multiplied by zero is zero). In the previous examples of this paper, the learning rate was constant — we frequently set it to $\alpha = \{0.1; 0.01\}$ and then train the neural network for a fixed number of epochs without changing the learning parameter, once it is set. This method is adequate in some situations, but more often than so it is more useful to reduce the learning rate over time. In practice, we are looking for a point that does not have to be a global minimum or a local minimum, but only necessities to be in a portion with relatively low losses and that is considered to be good enough.
Consecutively, if we constantly maintain a high learning rate, we could potentially surpass those areas with small losses, since we have taken quite large steps and bypassed that portion. Instead, it is convenient to reduce the learning rate, allowing the

neural network to take smaller steps — such a reduced learning rate permits the neural network to descend into areas of the loss that are more optimal (these areas would be completely ignored with a large learning rate).

The process of reducing the learning rates can be viewed as:

- ✓ Finding a set of adequate weights in the process of early neural network training while still utilizing a relatively large learning rate.
- ✓ Adjusting these weights later during the training process in order to find more optimal weights empoying a smaller learning rate.

There are two fundamental types of learning rates:
1. Learning rates that gradually decay based on the number of epochs (such as a linear, polynomial, or exponential functions).
2. Learning rates that decay based on a certain epoch.

4.3.1. THE STANDARD DECAY SCHEDULE

The standard decay is essentially the fastest descent method that we have explained during the first chapter. The $Keras$ library contains a time-based learning rate — which is regulated via the decay parameter of the optimizer (such as the stochastic gradient descent / SGD). In the previous chapter, within the code, we initialized a SGD optimizer with a learning rate $\alpha = 0.01$, a momentum $\eta = 0.9$ and we pointed out that we utilize the Nesterov acceleration gradient. Afterwards, we set the decay to actually be the quotient of the learning rate with the total number of epochs, which resulted in a $\text{decay} = 0.01/40 = 0.00025$. The $Keras$ library employs the following arrangement of learning rates to adjust the learning rate after each epoch:

$$\alpha_{e+1} = \alpha_e \times \frac{1}{1 + \eta * e} \tag{4.2}$$

Further, if we set the decay parameter to zero (which is the default value for the $Keras$ library optimizers, unless we explicitly change it) we will notice that there is no effect on the learning rate (here we arbitrarily set our current epoch to $e = 1$ in order to demonstrate this point):

$$\alpha_{e+1} = 0.01 \times \frac{1}{1 + 0.0 * 1} = 0.01 \tag{4.3}$$

Nevertheless, if instead of that we utilize the decay, $\text{decay} = 0.01/40$, we shall notice that the learning rate commences to decrease after each epoch. By the means of this time-based decay in the learning rate, the $MiniVGGNet$ model gained 84% of classification accuracy.

This result has led us to a much better learning accuracy. Amongst using the learning rates with decreasing values we can often not only improve the accuracy of classification, but also reduce the impacts of the neural network overfitting, thereby increasing the ability of our model to generalize.

4.3.2. STEP-BASED DECAY METHOD

Another prevalent method for adjusting the learning rate is step-based decay, where we systematically drop the learning rate after a certain number of epochs during training. Here, the learning rate is completely constant for a couple of epochs, and then it decreases, so it is constant again and decreases again, and so on, etc.

In applying the step decay to a learning rate, we can devise two options:

- ✓ We define an equation that models the piecewise decay of the learning rate we want to achieve.
- ✓ We utilize the "$ctrl + C$" method (the convolutional neural network stop) to train a deep learning neural network for which we train a certain number of epochs at a certain learning rate, and ultimately determine the performance of the test data set.

When we employ a decay step, we often lower the learning rate either by half or by a certain size, after a certain number of epochs. Let us declare that our initial learning rate is $\alpha = 0.1$. After 10 epochs the learning rate will be $\alpha = 0.05$, after ten more epochs it will decline to a learning rate of $\alpha = 0.025$, this means that the learning rate decreases as follows $\alpha = 0.1 * 0.5 = 0.05; \alpha = 0.05 * 0.5 = 0.025$.

4.3.3. IMPLEMETATION ON CIFAR-10 DATA SET

The $Keras$ library contains the $LearningRateScheduler$ class, which permits you to define a function that is adapted to the learning rate, and then to apply it automatically during the neural network training process. This function should take a number of epochs as an argument and then calculate our desired learning rate based on the previously defined function.
In this chapter, we will define a function that will reduce the learning rate by a certain F factor after every D epoch.
The equation shall obtain the following shape:

$$\alpha_{E+1} = \alpha_1 \times F^{\frac{1+E}{D}} \tag{4.4}$$

Here is our initial learning rate is α_1, whilst F is the value of the factor that controls the rate at which the learning parameter decreases, D is the drop value of each epoch, whereas E is the current epoch. The higher the F factor, the slower the learning rate decreases. Conversely, the smaller the F factor, the faster the learning rate will decrease. We import the $LearingRateScheduler$ class from the $Keras$ library, this class will allow us to define the learning rate.

Let us first and foremost define the $step\_decay$ function that receives only one parameter, and that is the current epoch. Then we define the learning rate $\alpha = 0.01$, the decay factor 0.5, and we set that the decline occurs every 5 epochs. This means that our learning rate will decrease in the following manner $0.01 * 0.5 = 0.005$ после 5 epochs.

The neural network was trained with an accuracy of 84%, in the same manner as in the previous chapter. By changing the learning rates over the epochs, we were able to speed

up the learning process to about 5 minutes per epoch. Based on the Figure 4.3.3.- 2, we can clearly observe that the neural network continues to learn even after the $25 - 30$ epoch, up until the loss begins to stagnate on testing data after the 30th epoch, there is the learning rate is also quite small and is not able to make any significant weight adjustments, nor to affect the loss / accuracy of test data.

	precision	recall	f1 score	support
airplane	0.87	0.85	0.86	1000
automobile	0.93	0.92	0.93	1000
bird	0.82	0.74	0.78	1000
cat	0.74	0.68	0,71	1000
deer	0.80	0.86	0,83	1000
dog	0.77	0.78	0.77	1000
frog	0.83	0.92	0.87	1000
horse	0.89	0.86	0.88	1000
ship	0.91	0.92	0.91	1000
truck	0.89	0.91	0.90	1000
accuracy			0.84	10000
macro avg	0.84	0.84	0.84	10000
weighted avg	0.84	0.84	0.84	10000

Table 4.3.3.- 1 - Demonstration of the accuracy and the loss over time on the training data set and the testing data set in the form of results that are depicted within the Command Prompt with three layers and with batch normalization / BN

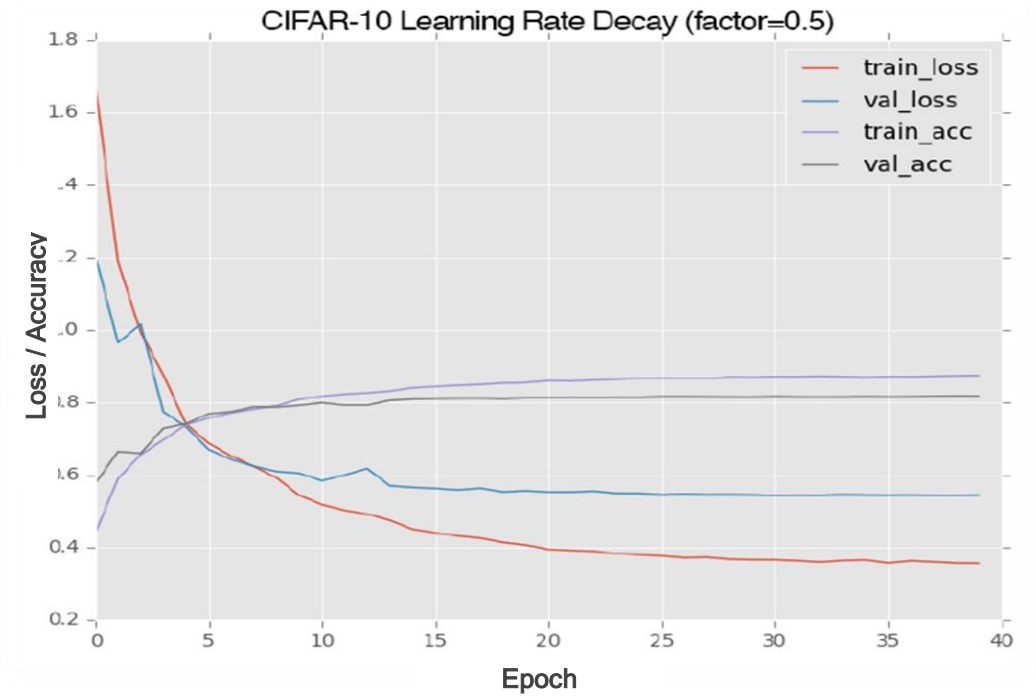


Figure 4.3.3.- 2 - Display of the accuracy and the loss over the course of time on the training data set and testing data set with batch normalization and a decay factor 0.5

5. TRAINING OF THE CONVOLUTIONAL NEURAL NETWORK: ANIMALS

The principal topic of this Chapter is to gaze at a simple shallow neural network and survey how the model is saved and how the image recognition is performed based on the novel input data. The Animals data set consists of 3000 images that have the size of 32×32 pixels and that are arranged in 3 classes: a dog, a cat and a panda. Figure 4.1.1.- 1 demonstrates how the schematic of the directories and subdirectories, which are necessary for this code to work, look like. It should be noted, that the main directory contains a set of input images in a directory labeled with the name animals.
We can also articulate that the tree that pulls our set of images and classifies them is identical to that in Chapter 4.1. the shallow convolutional neural network, the only difference being that for the main code, instead of $shallownet\_cifar10.py$ we utilize $shallownet\_animals.py$. The codes for the preprocessors are almost identical to those in subsections 4.1.1. и 4.1.2., the code for loading input data is the same as in the subsection 4.1.3., And lastly the code for implementing a shallow neural network is the same as inside subsection 4.1.5. So we will skip their re-explanations and focus only on the implementation of the $shallownet\_animals.py$ master code.

5.1. IMPLEMENTATION ON ANIMALS DATA SET

Once, we load images, we also insert preprocessors within the main code that will be applied in sequential order. First, the size of the provided input data will change, i.e. images will be reduced to 32×32 pixels. Then the resized image will be edited according to our channels and for that purpose the $keras.json$ configuration file is utilized. Now that the data and labels have been loaded, one can train and test the neural network with “one hot encoding”.

75% of the data will be used for neural network training and 25% for testing. First and foremost, we initialize the SGD (stochastic gradient descent method) optimizer using the learning rate $\alpha = 0.005$. The $ShallowNet$ architecture was introduced in the following manner: to have the width and the height of 32 pixels, together with a depth of 3 — this entails that the images are 32×32 pixels with three channels (i.e., the channels are in color RGB). Since animal data set has three classes of labels, we set the classes to 3. The model is then compiled and here we will use the cross-entropy as a loss function. The neural network will be trained with a 100 epochs using a mini-batches the size of 32 (which actually means that 32 images will be presented to the neural network at the same time, and feedforward and backpropagation will be done in order to update the neural network parameters).

After the training process, a classification accuracy of 68% was obtained, with the neural network being trained for an average of 2 seconds per epoch. This probably happened because after a certain epoch the learning rate parameter is a bit higher, which means that overfitting of the neural network occurs, which is also very visible after the 20 epoch. We will elucidate the solution to this problem in Chapter 8, where we will use data augmentation on an adequate example.

	precision	recall	f1 score	support
cat	0.59	0.77	0.67	262
dog	0.63	0.40	0.49	249
panda	0.83	0.87	0.85	239
accuracy			0.68	750
macro avg	0.68	0.68	0.67	750
weighted avg	0.68	0.68	0.67	750

Table 5.1.- 1 - Demonstration of the accuracy and the loss over time on the training data set and the testing data set in the form of results that are depicted within the Command Prompt

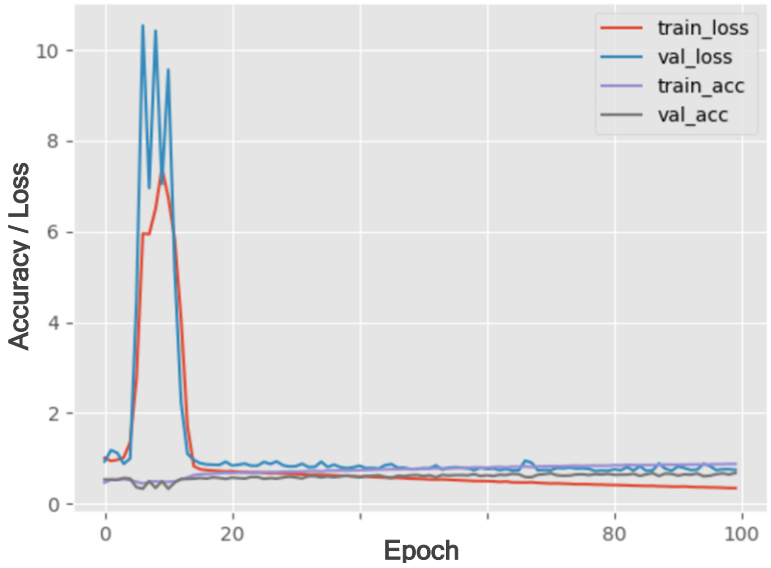


Figure 5.1.- 2 – Display of the accuracy and the loss over the course of time on the training data set and testing data set

5.2. SAVING AND LOADING A TRAINED NEURAL NETWORK

After the training of the neural network, it is necessary to save the network for its further use, i.e. in order to classify new data. In the previous chapter, we explained the approach utilized on the animals data set. However, we shall at the moment implement the two codes $shallownet\_train.py$ and $shallownet\_load.py$, and demonstrate their results. One is for training of the convolutional neural network, and the other is for the testing on randomly selected novel input data. The previous code required only one argument at hand the --dataset, which is the path to the directory with the Animals data set. This code requires another argument --model that is the path to the directory, i.e. the locations where we would be partial to saving the neural network after completing the neural network training process.

Now that the convolutional neural network is trained, we can store it in a directory as follows; in using the $model.save$ command we are essentially providing a path to be saved to our local disk. This method takes the weights and state of the optimizer and then serially implements them on the disk in the form of the $hdf5$ format. We can obviously see that the accuracy of the trained network is 68%, the most common error occurs when it comes to recognizing and distinguishing dogs and cats. Further, it can

now clearly be observed, within our main directory, that a directory dubbed the $shallownet\_weights.hdf5$ has been saved, and it represents a serialized network.

animals
utilities
shallownet_load.py
shallownet_train.py
shallownet_weights.hdf5

Table 5.2.- 2 Depicts the main directory where the code for saving and loading the previously saved neural network is located, everything else is the same as in chapter 4.1. (previously trained and saved model is marked in purple)

Now that the model is trained and serially implemented, we need to load it from the disk. In the next section, we will explain how to make classified images appear on the screen. The function used to load a trained model from disk is $load\_model$. This function is responsible for using the path of the trained neural network (i.e. our $hdf5$ file), as well as, for setting the weights within the neural network architecture so that we can either continue training the network or use this file to classify new images i.e., novel input sizes. For the test images, 10 random images are taken from the Animals data set directory for further classification. Each of these ten images is then preprocessed, and then extracted from the directory for additional classification. A very important point is to preprocess these ten randomly selected images in the exactly same way as we preprocessed the images during the training of the neural network itself. If we do not do this, it can lead to poor classification, i.e. incorrect classifications because our neural network will receive patterns that it is unable to recognize. The conclusion is that images that are tested and trained should always be preprocessed in an identical manner. The $model.predict$ method of the model will return a list of probabilities for each image from the input data set, one probability for each class label. Utilizing $argmax$ on the $axis = 1$, the index of the class with the highest probability is found for each of the images.

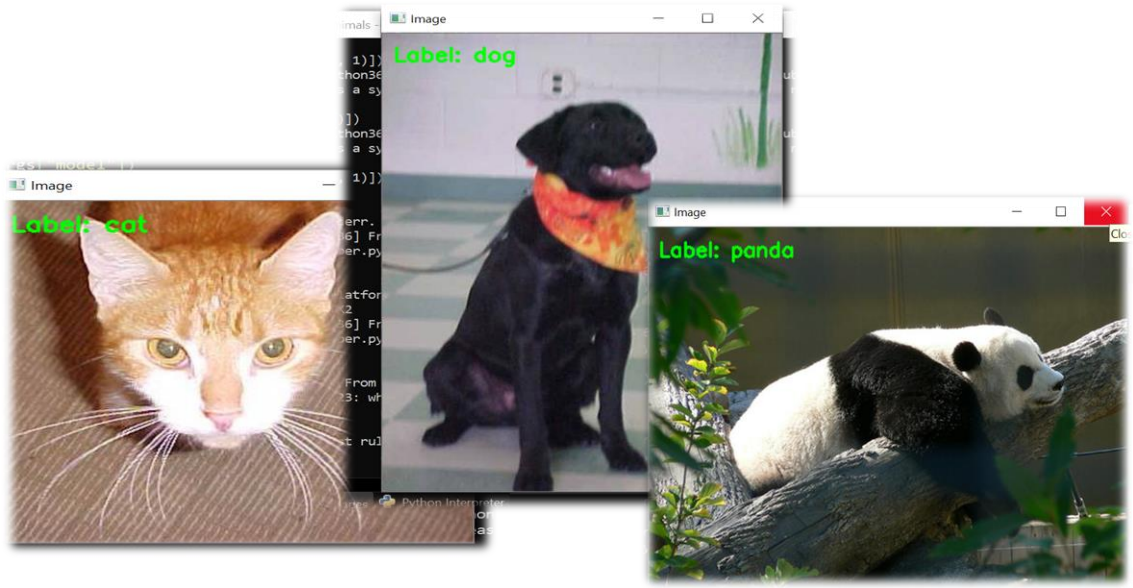


Figure 5.2.- 3 Depiction of the image recognition with an accuracy of 68%, the code was able to recognize accurately 10 out of 8 images

Given that we discern that the accuracy of the classification is 68%, we can utter that every third image from the input data set will probably be incorrectly classified.

6. TRAINING OF THE CONVOLUTIONAL NEURAL NETWORK: MNIST

Momentarily, is time to look back at what is the fundamental of any convolutional neural network, the backpropagation algorithm. We have previously briefly elucidated theoretically what the significance of the backpropagation algorithm is and why we use the $Keras$ library instead of the BP algorithm. Nevertheless, in this Chapter, we will explain how and why it is actually better, on one of the crucial and basic data sets. You may view this explanation as a multiplication table in the world of deep learning (the basis for any further understanding). We will furthermore explain the $LeNet$ technique as we did with the convoluted $VGGNet$.

Let us recollect that MNIST, which is also built into python within the $Scikit-learn$ library (although only part of the data set in this case), is a data set containing 70000 images. This means that we have 7000 images per number, i.e., digit, and the digits are in the range from $0 - 9$. Each digit is displayed in the form of a 784 dimensional vector, i.e., in the form of $28 \times 28 \times 1$, where we know that 28 is the width and the height, while 1 is for the channel (we have only one because these are black and white images).

6.1. BACKPROPAGATION ALGORITHM

In order to apply the backpropagation algorithm, our activation function must be differentiable, so that we can calculate the partial derivative of the error with respect to a given weight $w_{i,j}$, loss (E), output from the nod o_j and neural network output net_j.

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j}\frac{\partial net_j}{\partial w_{i,j}} \tag{6.1}$$

Given that, we already have a picture of the appearance of the main code from the previous two Chapters, we will emphasis only on the implementation of brand-new parts of the code, considering that loading images, as well as, determining the epochs and learning rates, certainly takes place in a similar fashion. In this particular example, we will explain the most crucial things within the $neuralnetwork.py$ code that implements the BP algorithm.

utilities	__pycache__	__pycache__	__pycache__
bp_mnist.py	datasets	cnn	__init__.py
keras_mnist.py	nn		neuralnetwork.py
	preprocessing		
	__init__.py		

Table 6.1.- 1 - Implementation of a $neuralnetwork$

By defining the $NeuralNetwork$ function, we conclude that we must have one mandatory argument, and another one that is optional. The first argument $layers$ is an integer list that represents the actual feedforward neural network architecture. For example, the value $[2, 2, 1]$ implies that the input layer has two nodes, the hidden layer has two nodes, and the output layer has one node (this is regulated within the main code). The second argument is alpha — α and it represents the learning rate of the neural network, this value

was applied during the weight adjustment phase. Primarily, we initialize the weigh coefficients (the initialization on its own is completely random) and if necessary, we apply the "bias trick". Then we define the sigmoid activation function, as well as, the derivative of the sigmoid function as ensues: $return\ 1.0\ /\ (1\ +\ np.exp(-x)), x * (1 - x)$. Inside this code, we also specify the number of epochs we want to apply to our neural network as for example 1000 and show the changes after every 100 epoch.

In utilizing the following portion of the code: $net\ =\ A\ [\ layer\ ].dot(self.W\ [\ layer\ ])$ within the for loop for each layer of the neural network, we take the elementary product between the activation and the weight matrix, then the output from this neural network layer is calculated by passing it through a nonlinear sigmoid activation function and we proceed to the succeeding layer.

The first phase of the backpropagation algorithm is to calculate the error or difference between the predicted label and the correct label. It is necessary to start compiling a list of deltas, D, deltas will be used to adjust the weight matrices, scaled for the α learning rate. The first entry in the delta list is the error of the output layer multiplied by the derivative of the sigmoidal output value $D\ =\ [\ error\ *\ self.sigmoid\_deriv(\ A[-1]\ )\ ]$.

Then we commence the loop over each of the layers in the neural network (ignoring the previous two layers, i.e. the last and penultimate layer, because we have already calculated their error before) in reverse order, i.e. backwards to calculate and adjust the deltas for each layer.

The delta for the current layer is equal to the delta of the previous layer, $D[-1]$ multiplied by the multiplication method by the weight matrix of the current layer. To complete the delta calculation, we multiply the delta by the derivative of the sigmoid activation function of the layer. Simply put, we observe that the backpropagation algorithm is iterative — we merely take the deltas from the previous layer, and multiply them by the weight matrices of the current layer, and then multiply them by the derivative of the activation function.

	precision	recall	f1 score	support
0	1.00	1.00	1.00	47
1	0.93	1.00	0.96	51
2	0.94	0.98	0.96	46
3	0.98	0.98	0,98	51
4	1.00	0.98	0,99	47
5	1.00	0.95	0.98	42
6	1.00	1.00	1.00	49
7	1.00	0.97	0.99	36
8	1.00	0.92	0.96	48
9	0.94	1.00	0.97	33
accuracy			0.98	450
macro avg	0.98	0.98	0.98	450
weighted avg	0.98	0.98	0.98	450

Table 6.1.- 2 – Demonstration of the accuracy and the loss over time on the training data set and the testing data set in the form of results that are depicted within the Command Prompt

This process is then repeated until we reach the first layer in the neural network. In the end, we go forward again expanding through our neural network and multiply our deltas and current layers with the learning rate, and afterwards we adjust the weights. The procedure described earlier is actually the neural network training.

The value of the loss function is initially very large, but then declines rapidly during the training process through all of the 1000 epochs (Figure 6.1.-3). Since in this example we utilize a smaller MNIST data set from the $Scikit-learn$ library, we had images with dimensions $8 \times 8 = 64$, accordingly the first layer had 64 nodes, the second layer had 32 nodes, while the third layer had 16 nodes, and the last of course 10 nodes (identical to the number of classes according to which we arrange our digits). The classification report displays that the testing obtained a classification accuracy of 98%, however, we have had some problems with the classification of digits 4 and 5 (the accuracy was 95% and 94%). This served only as a demonstrative example, so no attention was paid to training on the entire data set.

Figure 6.1.- 3 – Display of the loss function employing the feedforward neural network

6.2. LIBRARY KERAS INSTEAD OF THE BACKPROPAGATION ALGORITHM

Here we employ the $Keras$ library instead of the backpropagation algorithm. Firstly, we will observe the crucial differences here, and then we will implement a convolutional neural network and perceive significant improvements. All of the images have dimensions $28 \times 28 = 784$ and the data set dabbles with 7000 per class. The following libraries $LabelBinariyer,\ Sequential,\ Dense,\ SGD$... are handled within the code.

The implementation is done in the same manner as in the previous Chapters, it involves downloading and preprocessing images to certain dimensions, and then "$one\ hot\ encoding$", i.e. translating integers into vectors. With the help of the $Keras$

library, we can easily define the architecture of our neural network. We start with the $Sequential$ class which implies that the layers will be stacked on top of each other, so that the output of the previous layer will be the input of the current layer. Then we define the first fully connected layer in the neural network and the input shape is set to 784, the dimensionality of each point in the MNIST data set. Afterwards, we adjust the 256 weights in this layer and apply the sigmoidal activation function. The subsequent layer (the hidden layer) sets 128 weights. Finally, another fully connected layer is applied, this time only adjusting 10 weights, which corresponds to the number of output classes — ten (digits from 0 to 9). Instead of the sigmoid activation function, we shall utilize the Softmax activation function in the last layer in order to obtain a normalized class probability for each prediction.

	precision	recall	f1 score	support
0	0.95	0.97	0.96	1740
1	0.94	0.97	0.96	1944
2	0.92	0.90	0.91	1799
3	0.89	0.90	0,90	1766
4	0.92	0.93	0,93	1688
5	0.88	0.84	0.86	1567
6	0.93	0.96	0.94	1713
7	0.95	0.91	0.93	1822
8	0.89	0.88	0.88	1690
9	0.89	0.91	0.90	1771
accuracy			0.92	17500
macro avg	0.92	0.92	0.92	17500
weighted avg	0.92	0.92	0.92	17500

Table 6.2.- 1 – Demonstration of the accuracy and the loss over time on the training data set and the testing data set in the form of results that are depicted within the Command Prompt

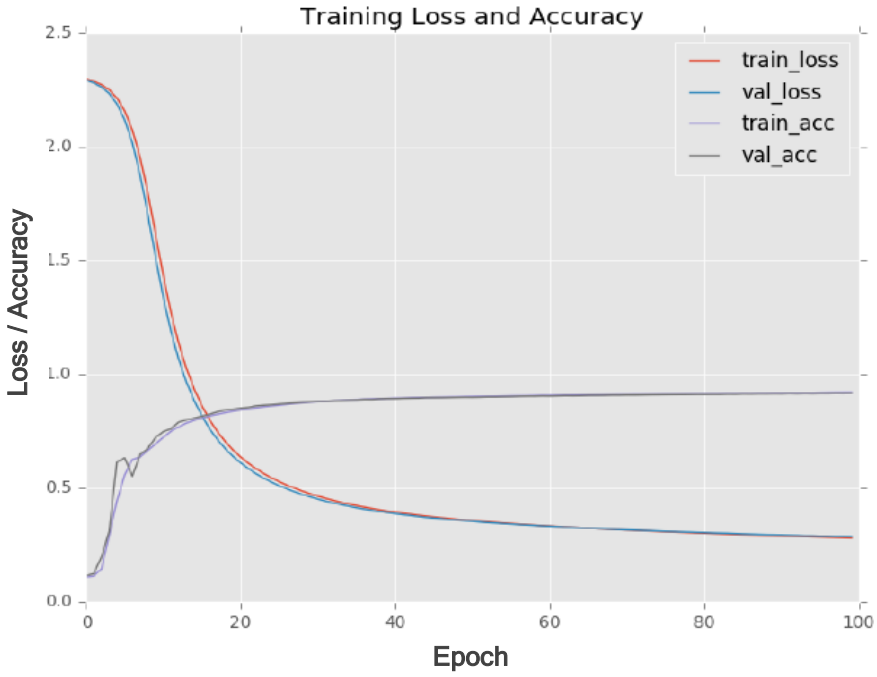


Figure 6.2.- 2 – Display of the loss function employing the $Keras$ library

Thenceforth, we initialize the SGD optimizer with the learning rate of $\alpha = 0.01$. The loss function we use is the cross-entropy function. This is also the principal reason why "$one\ hot\ encoding$" must be applied. As a standard in this paper, we divide the training data and the testing data to 75% and 25%. In most circumstances, for example, when adjusting hyperparameters or deciding on a model architecture, we will specifically create a validation set, which will not be the same as the test data set within those 25%.

The neural network utilized the batch size of 128 points per iteration and was trained on 100 epochs. As we can establish from the results, we have achieved an accuracy of 92%. On average, we trained one epoch for 1 second. Also, the training and testing curves (Figure 6.2.-2) are almost identical to each other and it is quite clear that they coincide, which indicates that there is no neural network overfitting or any other problems during the neural network training process. Howbeit, this accuracy is still very small, in the subsequent Chapter we will illustrate how to maintain good stability and accuracy of the data set.

6.3. COMPLEX NEURAL NETWORK WITH LENET

The $LeNet$ architecture is one of the fundamental articles of deep learning, and was first introduced by LeCun in his paper [16] in 1998. The paper primarily dealt with optical character recognition. The $LeNet$ architecture is simple and small (in terms of the memory footprint), which makes it very suitable for utilization in the convolutional neural networks with a small data set. We will use the previous technique in this Chapter as an inspiration and apply it to the MNIST data set.

The architecture of our neural network consists of the following layers. The input layer has image input data with 28 rows, 28 columns and with one depth channel (because a black and white image is in question). In the first convolution layer, we apply 20 filters the size of $5 \times 5 = F \times F$. The convolution layer is then followed by the ReLU activation function, followed by a pooling layer / POOL layer that does the pooling with a stride 2×2.

The next convolutional layer follows this same pattern, only now learning 50 filters. It is common for the number of convolutional layers to increase as we go deeper into the neural network. At the very end we have two FC layers. The first FC layer contains 500 hidden nodes, followed by the ReLU activation function, while the final FC controls the number of output class labels ($0 - 9$, one for each of the possible ten digits). Finally, we apply the Softmax activation function to obtain the class probability.
Let us present this with a simple mathematical approach:

* The first convolutional layer:

$$INPUT\ (28 \times 28 \times 1) \implies$$
$$\implies \quad CONV\ (28 \times 28 \times 20) \implies ReLU\ (28 \times 28 \times 20) \implies POOL\ (14 \times 14 \times 20)$$

* whereby here 20 filters the size of 5×5 act on the CONV layer, and stride 2×2 acts on the POOL layer.

* The second convolutional layer:

$$\Longrightarrow CONV\ (14 \times 14 \times 50) \Longrightarrow ReLU\ (14 \times 14 \times 50) \Longrightarrow POOL\ (7 \times 7 \times 50)$$

* whereby here 50 filters the size of 5×5 act on the CONV layer, and stride 2×2 acts on the POOL layer.

$$\Longrightarrow FC\ (500) \Longrightarrow ReLU\ (500) \Longrightarrow FC\ (10) \Longrightarrow Softmax\ (10)$$

6.3.1. LENET ARCHITECTURE

The libraries enveloped by this architecture are $Sequential,\ Conv2D,\ MaxPooling2D,\ Activation,\ Flatten,\ Dense,\ K$ (all elucidated beforehand). This architecture was then only implemented in the code in a similar manner as the complex convolutional neural network was implemented in the previous Chapter. Further explanations are superfluous, the only thing that matters is that in the next Chapter we will use this same system of $LeNet$ architecture to recognize a smile.

utilities	__pycache__	__pycache__	__pycache__
lenet_mnist.py	datasets	cnn	__init__.py
	nn		lenet.py
	preprocessing		
	__init__.py		

Table 6.3.1.- 1 Implementation of the $LeNet$

6.3.2. IMPLEMENTATION ON MNIST DATA SET

Now, after a couple of implementations, we already have a quite sharp picture of how convolutional neural network training is done:

- ✓ we train the neural network architecture
- ✓ we select an optimizer (in this case it is the SGD)
- ✓ we choose the function we shall use to separate the input data into the training and the testing data sets
- ✓ we insert a function that gives a detailed report on the classification, so that we can evaluate the work of the classifier

The MNIST data set has already been processed and classified, so we imported them using the $datasets.fetch\_mldata$, although we could train them in the same manner as the previous examples and have an identical result. Then we call the $model.predict$ function. For each sample in $test\_x$, the batch sizes of 128 are moved through the neural network and are classified. Once it passes over all of the input data set, the $predictions$ variable is returned. The variable $predictions$ is actually a $NumPy$ array with the shape of $(len\ (test\_x), 10)$ implying that at the moment we have 10 probabilities associated with each class label for each data in the $test\_x$. Taking into the bargain, the $predictions.argmax(axis = 1)$ in the classification report, the label index is found most likely to belong to a certain class. Then we can compare the classification of the neural network with the factual one i.e. the correct labels.

The neural network achieved a classification accuracy of 98%, after 10 minutes of training over 20 epochs, utilizing the cross-entropy function as a loss function. From the Figure 6.3.2.- 2 we can undoubtedly observe that $LeNet$ reached a 96% classification accuracy after only five epochs. The loss of the training data and the testing data continues to drop with only a few minor leaps, due to the learning rate, which remained constant (i.e. we did not employ the decay method). At the end of the twentieth epoch we reach a 98% accuracy on the testing data set.

	precision	recall	f1 score	support
0	0.99	0.99	0.99	1677
1	0.99	0.99	0.99	1935
2	0.99	0.98	0.98	1767
3	0.93	0.99	0,96	1766
4	0.99	0.99	0,99	1691
5	0.99	0.97	0.98	1653
6	0.99	0.99	0.99	1754
7	0.98	0.99	0.98	1846
8	0.98	0.96	0.97	1702
9	0.98	0.97	0.98	1709
accuracy			0.98	17500
macro avg	0.98	0.98	0.98	17500
weighted avg	0.98	0.98	0.98	17500

Table 6.3.2.-1 – Demonstration of the accuracy and the loss over time on the training data set and the testing data set in the form of results that are depicted within the Command Prompt

The graph displaying the loss and the accuracy of $LeNet$ at the MNIST data set has evidently shown that the loss and the accuracy of the training and testing data almost coincide with each other, leading us to conclude that the overfitting is minimal. We only used the MNIST data set as a basis to later on master the Smiles data set more easily.

Figure 6.3.2.- 2 – Display of the loss function employing the $LeNet$ neural network architecture

7. TRAINING OF THE CONVOLUTIONAL NEURAL NETWORK: SMILES

It is time to apply the $LeNet$ architecture to a set of images that contain the faces of people who are smiling and who are not smiling. Once we trained the neural network, we shall create a distinctive script within python, which will recognize faces in images using the $OpenCV$ library that has a built-in $Haar\ cascade$ face detector. We will first take a picture of a face using a webcam, then we will call the model we trained with the face picture and run that image through the $LeNet$ neural network architecture in order to detect a smile.

The Smiles data set consists of images of people's faces that are either smiling or are not smiling. There is a total of 13165 black and white images in the data set, with each image having a size of 64×64 pixels. The images within this data set are firmly cropped around the face, which facilitates the training process because we will be able to learn directly from the input images, without additional modifications. Nevertheless, narrow cutting is a problem during testing — since our input images will not only contain the face, but will also contain the background. So first, we need to locate the face, then capture t and isolate it before passing it through our neural network to detect a smile. We will easily achieve this with the $Haar\ cascade$ face detector.

Another issue we need to address in the Smiles data set is the class imbalance. We have 13165 images in the data set, however 9475 of these examples are images without a smile, while only 3690 images are with a smile. Of course, our neural network can choose the label "not smiling" since the distributions are uneven and there are several examples of what a person looks like when he is not smiling. The ratio of images without a smile and the images with a smile is $2.5:1$.

7.1. MODEL TRAINING

On the Figure 7.1.- 1 the main directory with our codes is depicted. We will first consider the code $train\_smile.py$. Two arguments are required within the code: --dataset path to the directory where Smiles data set is located --model path to the place where $LeNet$ weights will be stored after training.
First we need to load all of the images from the directory, then set them all to be black and white (if they are not already), and finally change the size of all the images so that the input size is 28×28 pixels. The images should be converted to an array that is compatible with the $Keras$ library, and the classes should be arranged accordingly.

smiles
utilities
detect_smile.py
haarcascade_frontalface_default.xml
lenet.hdf5
train_model.py

Table 7.1.- 1 – Depicts the main code $train\_model.py$, in which we train the neural network, and then implement it with $detect\_smile.py$

One line of the code calculates the total number of samples per class. In this case, it will be an array $[9475, 3690]$ for images without a smile and with a smile. We performed this

addition in order to form an array of the shape $[1, 2.56]$, with the assist of the $classWeight$ class, which is utilized for handling the class division.

This weighting of data suggests that our neural network will treat all examples with a smile as 2.56 instances, and the examples that are not smiling as an 1 instances. This also assists to combat the problem of class imbalance, increasing the loss per instance for a bigger weight when you detect examples with a smile. Now that we have calculated the weights of the class, we can move on to dividing our data for training and testing, the separation is done using 80% of the data for training and 20% of the data for testing. The $LeNet$ architecture will accept 28×28 images of one channel, it is logical that we have two classes. We will train the neural network with 15 epochs with batch sizes of 64. We shall also use binary cross-entropy rather than the categorical cross-entropy as a loss function. It should be noted that the categorical cross-entropy is utilized solemnly when the number of classes is greater than two. So far we have employed the SGD optimizer to train our convolutional neural network, now we will be using the Adam classifier instead.

Adam (The Adam Classifier)

Firstly, we must describe the RMSprop optimizer, which is the direct precursor to the Adam optimizer.

$$cache = decay_{rate} * \ cache + (1 - decay_{rate}) * (dW ** 2) \tag{7.1}$$

$$W+= \ -lr * \frac{dW}{(np.sqrt(cache) \ + eps)} \tag{7.2}$$

The decay parameter is habitually defined as ρ, and it is a hyperparameter that is usually set to $\rho = 0.9$. Here we can see that the previous entries in the cache can be weighted and therefore significantly smaller. This aspect of the average movement allows the cache to "leak" the old square gradients and replace them with newer ones. The algorithm relies on the cache to be decreasing exponentially, which permits us to avoid a monotonous reduction of the learning rate.

$$m = beta1 * m + (1 - beta1) * dW \tag{7.3}$$

$$v = beta2 * v + (1 - beta2) * (dW ** 2) \tag{7.4}$$

$$x+= \ -lr * \frac{m}{(np.sqrt(v) + eps)} \tag{7.5}$$

The values of m and v are similar to the momentum SGD optimizer, relying on its previous values from the time $t - 1$. The value m represents the first moment (mean value) of the gradient, while v represents the second moment. It is typical for $beta1$ to be set to 0.9, and for $beta2$ to be set to 0.999, these values predictably never change. In fact, the weight adjustment is almost identical to the previously mentioned RMSprop optimizer, the only difference being that instead of the dW gradient we utilize a milder "ironed" version of m.

The neural network managed to achieve a classification accuracy of 91%, after 15 epochs. Training per epoch lasted on average about 15 seconds. Figure 7.1- 3 clearly demonstrates that after 6 epochs, the loss of the test data set starts to stagnate, which leads us to the conclusion that if we continued to train the neural network for more than 15 epochs, we would fall into overfitting.

	precision	recall	f1 score	support
not_smiling	0.95	0.91	0.93	1895
smiling	0.79	0.88	0.84	738
accuracy			0.90	2633
macro avg	0.87	0.90	0.88	2633
weighted avg	0.91	0.90	0.90	2633

Table 7.1.- 2 – Demonstration of the accuracy and the loss over time on the training data set and the testing data set in the form of results that are depicted within the Command Prompt

In Chapter 8, we will portray a data augmentation technique, which can be used to obtain greater accuracy and to avoid overfitting after the 15th epoch. However, the accuracy of 91% is quite fine, so we did not perform further experiments.

Figure 7.1.- 3 – Display of the loss function employing the $LeNet$ neural network architecture

The graph clearly displays that the curves of the loss functions of the test data set and training data set match, i.e. they are almost identical, with minor deviations. Also, the accuracy curve of the test data set and the training data set almost match. This tells us that although there is overfitting after the 15 epochs, our system will still be partially stable for some 5 epochs and will perform well.

7.2. IMPLEMENTATION ON SMILES DATA SET

The first argument, --cascade is the path to $Haars\ cascade$ which is used to detect faces in images. First published in 2001, Viola and Jones describe the $Haars\ cascade$ in detail in their paper [17]. The $Haars\ cascade$ algorithm can detect objects in images, regardless of their location and size. Perhaps most importantly, this detector can work in real time.

The second argument of the line --model, specifies the path to the serialized $LeNet$ architecture on disk in $hdf5$ format. When the detection of the face using the webcam is completed, we switch to black and white images from the trained data set. We indicate that a particular region will be considered a face and must have a minimum width of 30×30 pixels. Then the implementation is done in the neural network architecture and we can clearly observe if we have adequately detected the smile.

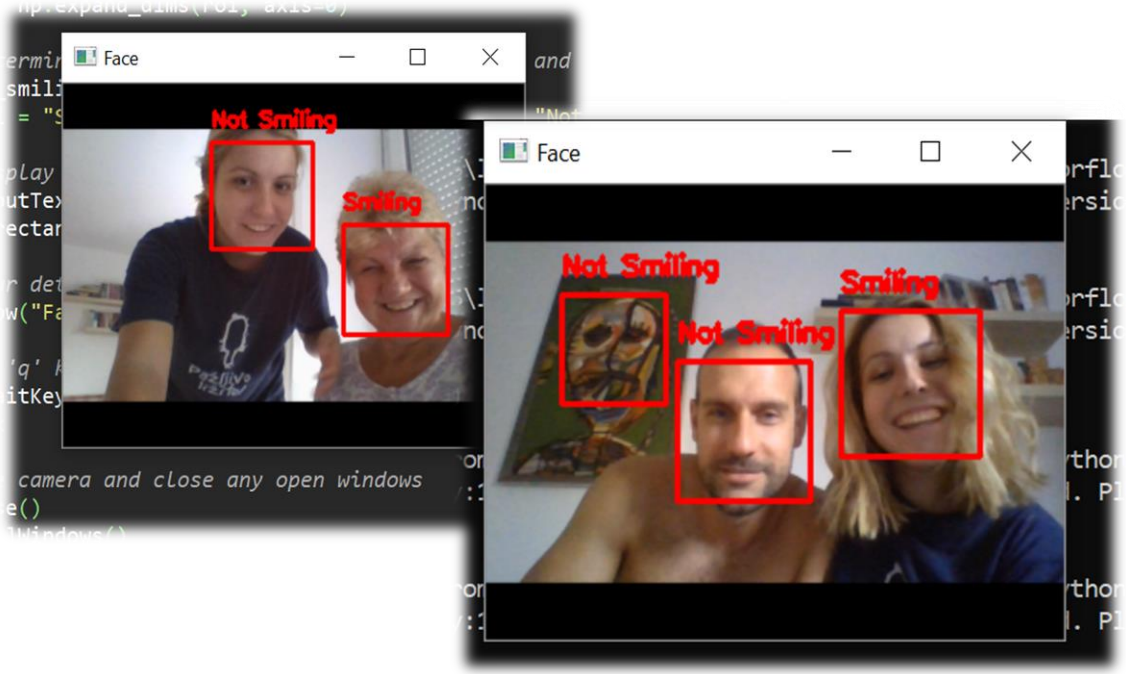


Figure 7.2.- 1 – Loading the trained neural network

From the Figure 7.2- 1 we can plainly perceive what it looks like when our neural network is started and tested. Let us pay attention first to the left picture, we observe that our neural network clearly distinguishes between a person who smiles (there are lines around the lips that are formed during a smile), and a person who simply just showed her teeth. On the other hand, if we look at the right picture, we notice that the neural network detected a smile and a face without a smile, however, in the background we notice that it also detected a picture of something that looked like a person who is not smiling. Just as the left image evidently shows the advantages of this convolutional neural network, the right one visibly indicates the disadvantages.

8. TRAINING OF THE CONVOLUTIONAL NEURAL NETWORK: FLOWERS-17

8.1. DATA AUGMENTATION OF THE CONVOLUTIONAL NEURAL NETWORK

Regularization strives to reduce the testing error, perhaps to the disadvantage by increasing the training error (nonetheless to a very small extent). We have already used different forms of regularization, which were parameterized forms of regularization, that required us to adjust the loss function.

We can utter that there are two types of regularization:
1. changing the neural network architecture itself
2. data augmentation that is transmitted to the neural network for training

Dropout is a stupendous example of a modification of the neural network architecture, and it achieves greater generalization. A layer is inserted here that randomly discontinuities the nodes connected from the previous layer to the following layer, ensuring that no single node is responsible for learning how to represent a given class. We are now considering another type of regularization called data augmentation. This method intentionally interferes with the samples, i.e. the images utilized to train the neural network by changing their appearance slightly, before transferring them to the neural network architecture. The end result is that the neural network constantly sees “new” training inputs generated from the original training data set, mitigating the requirement to collect more training data (although generally, collecting more learning data can only improve the neural network training). Our objective when applying data augmentation is to improve the model generalization.

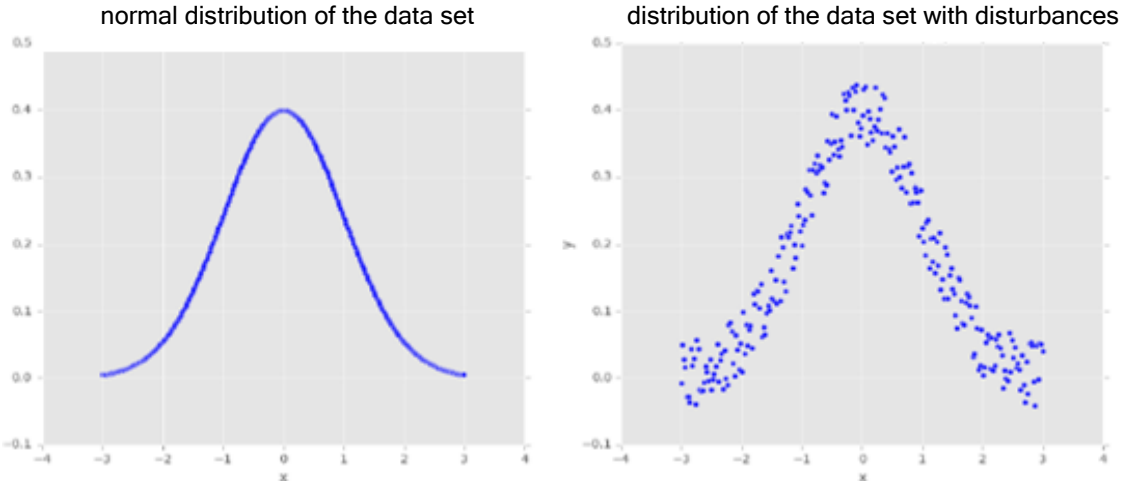


Figure 8.1.- 1 Normal distribution of the data set and distribution with disturbances of the data set

Consider the Figure 8.1.- 1, normal distributions with the zero mean value and a unit variance. Training the models of machine learning on this data can result in us accurately modeling the distribution. However, in real applications, the data rarely accompanies such an orderly distribution. As an alternative, to increase the generalization of the

classifier, we can primary randomly disrupt the distribution by expanding some values of ε extracted from the random distribution.

The function on the graph still follows an approximately normal distribution, but it is not a perfect distribution as on the left side. The model trained on the basis of these data certainly has a better generalization. For example, we can obtain supplementary training data from the original images by applying simple geometric transformations to them, such as:

- ✓ Translations
- ✓ Rotations
- ✓ Changes in the image scale, i.e. proportion
- ✓ Cropping out the edges of the image
- ✓ Horizontal or vertical rotations

Employing a small amount of these transformations to the input image will significantly change its appearance without changing the label class — thus increasing the data seems a natural and simple way to regularize within the deep learning for computer vision tasks. More advanced techniques, which we will not apply within this paper, in order to increase data include random color change in a provided color space and nonlinear geometric distortions.

8.1.1. ASPECT RATIO IN IMAGE PREPROCESSING

The portion of the code that we use this time together with the main code is termed $apectawarepreprocessor$, in Table 8.1.1.- 2 the path to the directory and the code itself is demonstrated. We previously processed the images by adjusting them to a fixed size, and in that process we were neglecting the proportions of the image. In some situations, especially for fundamental data sets with reference values, this is acceptable. However, for more challenging data sets, we should still strive to resize to a fixed size, whilst maintaining the proportion. An example of this is noticeably seen in Figure 8.1.1.- 1, and maintaining a consistent image ratio allows the convolutional neural network to learn discriminant and consistent characteristics.

Figure 8.1.1.- 1 - the original image (410×310) - on the right, the image where we ignore the proportion (256×256) - in the middle and the image where we maintain the proportion (256×256) - on the left

17flowers	__pycache__	__pycache__
utilities	datasets	__init__.py
minivggnet_flowers17.py	nn	imagetoarray_preprocessor.py
minivggnet_flowers17_data_aug.py	preprocessing	simple_preprocessor.py
	__init__.py	aspectawarepreprocessor.py

Table 8.1.1.- 2 – Implementation of an $aspectawarepreprocessor$

The libraries we use here are $cv2$ (which we have previously clarified) and $imutils$, that are utilized for simple image processing functions such as rotation, translation, resizing,...The beginning of the code requires us to enter two parameters the height and the width of the image, as well as, the interpolation method we shall use in the image processing. $dW = 0$ and $dH = 0$ determine the delta deviation point, which we will use when cutting a larger dimension.

This preprocessing represents a two-step algorithm:

- ✓ determine the smallest dimension and change the size along it.
- ✓ let us crop the image along the largest dimension to achieve the targeted width and height.

Primary, it is checked whether or not the width is less than the height, and if so, the size changes along the width. Otherwise, if the height is less than the width, then we change the size according to the height. Now that the image has been resized, we need to re-arrange the width and the height, using the deltas to cut out the center of the image. When cutting (due to rounding errors) our targeted image dimensions can be spoiled by one pixel each. That is why we call from the $cv2.resize$ library in order to ensure that our output image is the desired width and height.

8.1.2. DATA LABELING ON FLOWERS-17 DATA SET

Flowers-17 is a classification challenge where our task is to identify 17 different types of flowers. Data set, i.e. the images set is quite small and contains only 80 images per class, for a total of 1360 images.

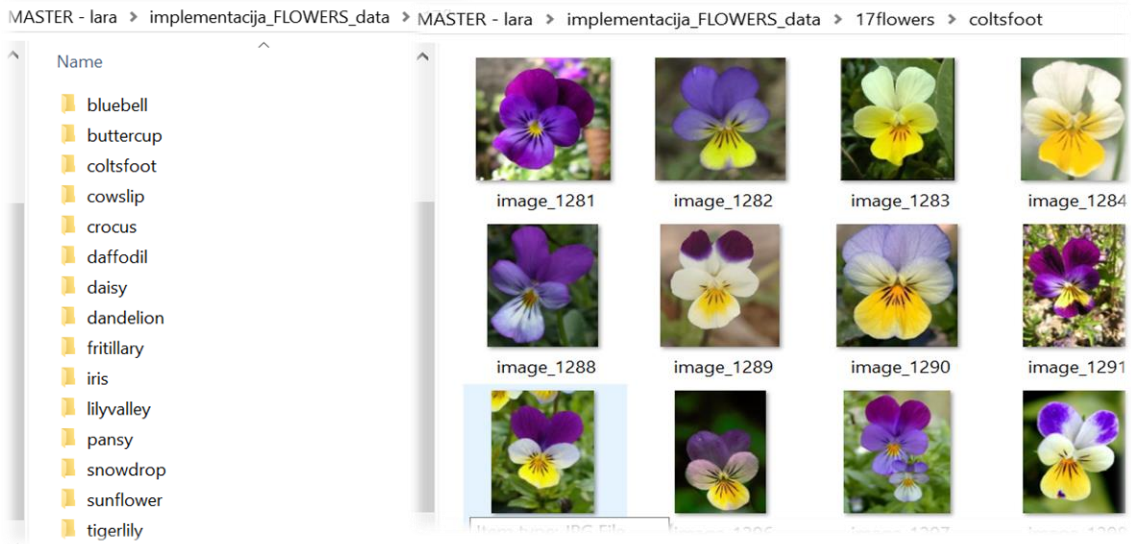


Figure 8.1.2.- 1 – Display of the sorting by class, and image labeling

The widespread rule when applying deep learning, as we mentioned earlier, is to have $1000 - 5000$ examples per class. Since we are in a deficit here, the ideal solution is the previously mentioned data augmentation. This will permit us to improve the accuracy of the classification, as well as, to reduce overfitting. This class is quite difficult to learn due to the large number of colors, there is a lack of opportunities in finding alterations between flowers, since they are all very similar in structure. In the Figure 8.1.2.- 1 the labeling process is clearly depicted, as well as, how the directory itself should be arranged, and all of this must be done before using the code. And this is exactly one of the most difficult things in deep learning, collecting adequate input data and sorting them. This example was not a big problem because it is relatively small, however in other examples with a larger amount of input this would really be a serious problem.

8.1.3. IMPLEMENTATION ON FLOWERS-17 DATA SET

For a detailed explanation of the $VGGNet$ we can refer to Chapter 4.2. Here we will just make an overview of the structure of our neural network. First we need to explain why we use only 2 layers this time instead of 3 layers. In the previous example, during the implementation of the $VGGNet$ onto the Cifar-10 data set, we clearly noticed that although we avoided overfitting in both cases, the neural network with 2 layers still performed training per epoch much faster. Nevertheless, we are taking into consideration that we will maintain the batch normalization / BN to make training more stable and to minimize the effects of overfitting and the impact of error. We also leave that to the dropout / DO layers, which we are inserting after the POOL and FC layers.

So our neural network consists of two groups of layers $CONV \Rightarrow ReLU \Rightarrow CONV \Rightarrow ReLU \Rightarrow POOL$, followed by a set of $FC \Rightarrow ReLU \Rightarrow FC \Rightarrow Softmax$ layers. The first two $CONV$ layers will be learning 32 filters, the size of $F \times F = 3 \times 3$, the second two $CONV$ layers will be learning 64 filters. Our $POOL$ layers will perform maximum pooling over the 2×2 with a stride $S = 2$.

* The first convolutional layer:

$$INPUT\ (32 \times 32 \times 3) \Rightarrow$$
$$\Rightarrow\ [\ CONV\ (32 \times 32 \times 32) \Rightarrow\ ReLU\ (32 \times 32 \times 32) \Rightarrow\ BN\ (32 \times 32 \times 32)\ ] \times 2 \Rightarrow$$
$$\Rightarrow\ POOL\ (16 \times 16 \times 32) \Rightarrow\ DO\ (16 \times 16 \times 32) \Rightarrow$$

*whereby here 32 filters the size of 3×3 act on the CONV layer, and stride 2×2 acts on the POOL layer.

* The second convolutional layer:

$$\Rightarrow\ [\ CONV\ (16 \times 16 \times 64) \Rightarrow\ ReLU\ (16 \times 16 \times 64) \Rightarrow\ BN\ (16 \times 16 \times 64)\ ]x2 \Rightarrow$$
$$\Rightarrow\ POOL\ (8 \times 8 \times 64) \Rightarrow\ DO\ (8 \times 8 \times 64) \Rightarrow$$

*whereby here 64 filters the size of 3×3 act on the CONV layer, and stride 2×2 acts on the POOL layer.

$$\Rightarrow\ FC\ (512) \Rightarrow\ ReLU\ (512) \Rightarrow\ BN\ (512) \Rightarrow\ DO\ (512) \Rightarrow$$

$$\Rightarrow FC\ (17) \Rightarrow Softmax\ (17)$$

We will primarily implement the $VGGNet$ without data augmentation, so that we can compare the results later and see the improvements.
From the $Keras$ library we implement the following necessary classes: $LabelBinarizer$, $train\_test\_split$, $ImageToArrayPreprocessor$, $AspectToAwarePreprocessor$, $SimpleDatasetLoader$, $VGGNet$, SGD, $paths$, np, $argparse$, os. We have already clarified all of these classes before, however we would like to mention that we use only one argument --dataset, which also represents the path to the directory where the Flower-17 dataset is located. Our directory where the images are located contains subdirectories ordered by their classes, as we have already demonstrated in the image above. The core idea being to extract the class name by simply calling the penultimate index and extracting the text. It is apparent to us that we will first initialize the $AspectToAwarePreprocessor$, i.e. to adjust the image to the desired size, and then the other preprocessors. We will then divide the data set into a training data set of 75% and a test data set of 25%. Training will be performed with regularization techniques, the batch normalization and the dropout, and with the SGD optimizer and the learning rate $\alpha = 0.05$. The $VGGNet$ will accept images of the ensuing dimensions $64 \times 64 \times 3$ (3 channels for images in color), and with a total of 17 classes.

	precision	recall	f1 score	support
bluebell	0.59	0.53	0.56	19
buttercup	0.61	0.73	0.67	15
coltsfoot	0.79	0.75	0.77	20
cowslip	0.59	0.43	0,50	23
crocus	0.48	0.58	0,52	19
daffodil	0.48	0.52	0.50	21
daisy	0.74	0.85	0.79	20
dandelion	0.76	0.48	0.59	27
fritillary	0.82	0.88	0.85	16
iris	0.88	0.75	0.81	20
lilyvalley	0.24	0.25	0.24	20
pansy	0.62	0.59	0.60	22
snowdrop	0.94	0.94	0.94	16
sunflower	0.81	0.72	0.76	18
tigerlily	0.58	0.83	0.68	18
tulip	0.71	0.65	0.68	23
windflower	0.50	0.61	0.55	23
accuracy			0.64	340
macro avg	0.66	0.65	0.65	340
weighted avg	0.65	0.64	0.64	340

Table 8.1.3.- 1 – Demonstration of the accuracy and the loss over time on the training data set and the testing data set in the form of results that are depicted within the Command Prompt, the neural network has 3 layers and batch normalization, the input data have no data augmentation

As we can observe from the results, we have managed to obtain a 66% for the classification accuracy, which is quite reasonable considering our limited amount of training data. Further, as we can evidently perceive in the Figure 8.1.3.- 2, the neural network quickly began overfitting after the 20 epoch. The reason for this behavior is that we only have 1.20 training examples with 60 images per class (other images are used for

testing). Keep in mind that ideally we should have between $1000 - 5000$ examples of images per class during convolutional neural network training.

The neural network was trained on 100 epochs each per 15 seconds, it was trained for about an hour on the average. In addition, in the first few epochs the accuracy of training leaps over 95%, in the end a 100% accuracy is obtained in later epochs — this result is an obvious case of overfitting. Due to the lack of significant training data, $VGGNet$ models the basic patterns in the training data too closely and is unable to generalize them on the testing data. We can apply regularization techniques to combat overfitting. Here, this includes data augmentation, dropout and batch normalization in order to further reduce the effects of overfitting.

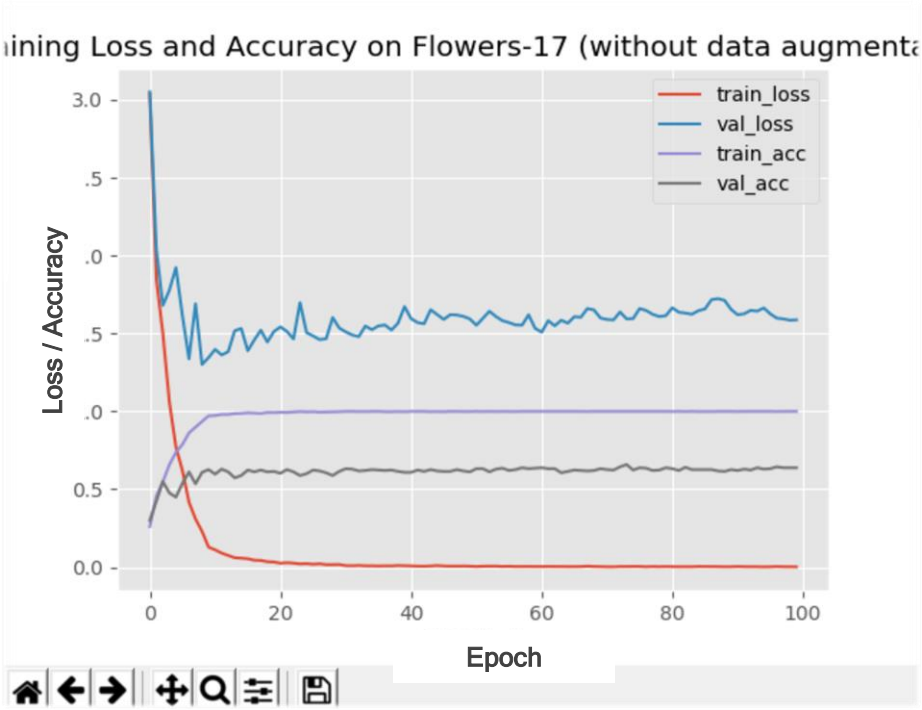


Figure 8.1.3.- 2 – Display of the accuracy and the loss over the course of time on the training data set and testing data set without data augmentation

Now, we shall demonstrate in the same manner how we will improve the results by adding the previously explained preprocessor, i.e., by increasing the number of input data. Everything remains comparable to the previous code except for minor changes, which are that we should allow the brand-new data to be divided into the training data set and testing data set, as well. However, the implementation of the new data is done in the subsequent way.

$aug = ImageDataGenerator(rotation\_range = 30, width\_range = 0.1, height\_shift\_range = 0.1, shear\_range = 0.2, zoom\_range = 0.2, horizontal\_flip = True, fill\_mode = "nearest")$ within this portion of the code we enable our input data i.e. images to rotate randomly ± 30 degrees, move horizontally or vertically by a factor of 0.2, to be shorten by 0.2, zoom in and form uniformly in the range $[0.8, 1.2]$ and rotate randomly horizontally. It is common for rotations to take place between 10 and 30 degrees depending on the input data. The horizontal and the vertical movements are completed in the range from 0.1 to 0.2, the same applies to zooming. However, for these

kind of deviations should be taken care of in order to not deviate too much from the initial data and thus change the class.

The accuracy of our neural network was 75%, and one epoch was trained for about 21 seconds, which means that it took us an average of 35 minutes for the entire neural network. We conclude that the training lasted longer, both because we appended the dropout and the batch normalization, in addition to data augmentation, as regularization methods.

	precision	recall	f1 score	support
bluebell	0.85	0.58	0.69	19
buttercup	0.75	0.60	0.67	15
coltsfoot	0.84	0.80	0.82	20
cowslip	0.67	0.35	0,46	23
crocus	0.64	0.84	0,73	19
daffodil	0.57	0.62	0.59	21
daisy	0.84	0.80	0.82	20
dandelion	0.92	0.44	0.60	27
fritillary	0.82	0.88	0.85	16
iris	0.95	0.95	0.95	20
lilyvalley	0.29	0.45	0.35	20
pansy	0.89	0.77	0.83	22
snowdrop	0.88	0.94	0.91	16
sunflower	0.69	1.00	0.82	18
tigerlily	0.59	0.89	0.71	18
tulip	0.89	0.74	0.81	23
windflower	0.68	0.83	0.75	23
accuracy			0.72	340
macro avg	0.75	0.73	0.73	340
weighted avg	0.75	0.72	0.72	340

Table 8.1.3.- 3 – Demonstration of the accuracy and the loss over time on the training data set and the testing data set in the form of results that are depicted within the Command Prompt, the neural network has 2 layers and batch normalization, the input data have data augmentation

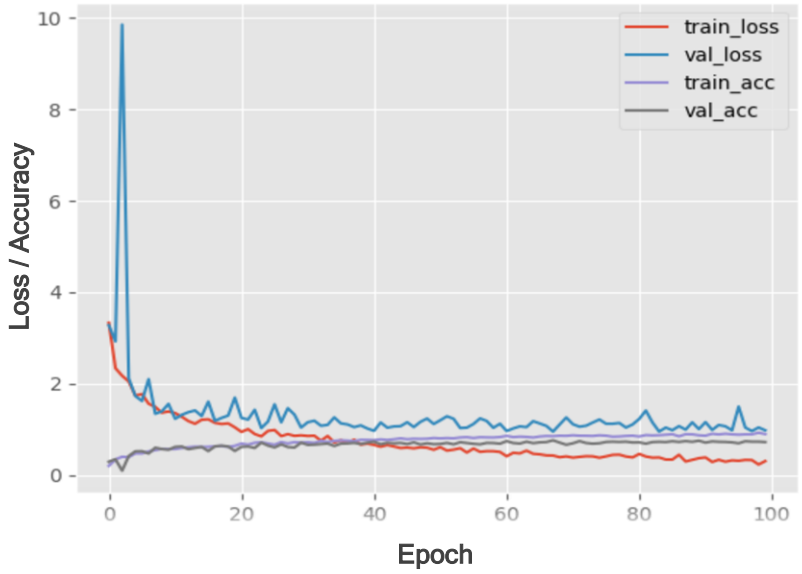


Figure 8.1.3.- 4 – Display of the accuracy and the loss over the course of time on the training data set and testing data set with data augmentation

However, the actual question is whether the data augmentation abetted in preventing overfitting. We can easily answer this if we observe the Figure 8.1.3.- 4. Although overfitting is still happening, the effect is significantly reduced by using data augmentation. We were able to increase the testing accuracy by improving the model generalization, despite decreasing the training accuracy.

	precision	recall	f1 score	support
bluebell	0.71	0.53	0.61	19
buttercup	0.75	0.80	0.77	15
coltsfoot	0.93	0.65	0.76	20
cowslip	0.65	0.57	0,60	23
crocus	0.59	0.68	0,63	19
daffodil	0.73	0.76	0.74	21
daisy	1.00	0.70	0.82	20
dandelion	0.74	0.74	0.74	27
fritillary	0.83	0.94	0.88	16
iris	0.95	0.95	0.95	20
lilyvalley	0.55	0.55	0.55	20
pansy	0.76	0.86	0.81	22
snowdrop	0.94	0.94	0.94	16
sunflower	0.86	1.00	0.92	18
tigerlily	0.63	0.94	0.76	18
tulip	0.75	0.78	0.77	23
windflower	0.75	0.65	0.70	23
accuracy			0.76	340
macro avg	0.77	0.77	0.76	340
weighted avg	0.77	0.76	0.76	340

Table 8.1.3.-5 – Demonstration of the accuracy and the loss over time on the training data set and the testing data set in the form of results that are depicted within the Command Prompt, the neural network has 2 layers and batch normalization, the input data have data augmentation, and we introduce $decay = 0.5$

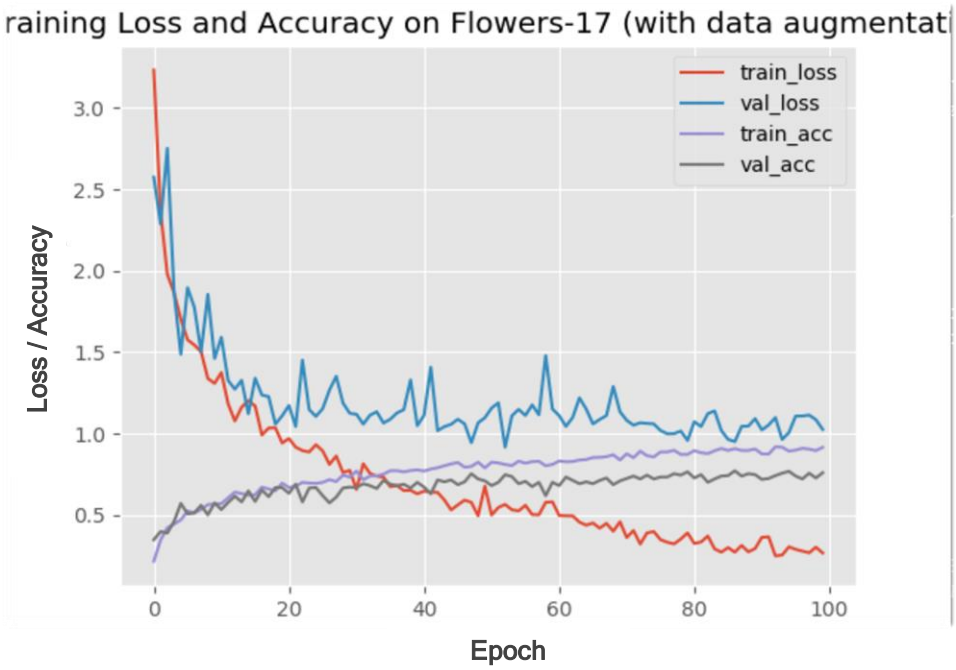


Figure 8.1.3.- 6 – Display of the accuracy and the loss over the course of time on the training data set and testing data set with data augmentation

Now that we have determined that our method functions, several experiments have been done. Let us first introduce, in addition to all of this, the decay, and it being 0.5 per epoch. And let us perceive at how it all affected the neural network. We grasp that the accuracy of the classification that we have now obtained is 77%, and that the neural network was trained 32 seconds per epoch. This accuracy works well, but let us observe at what happened. The neural network trained for an average of an hour.

From the Figure 8.1.3- 6, we can undoubtedly see that the principal change is that by adding a decay, we have made a significant change in the fact that the loss function in the testing data has from the very beginning up to 3 times less the loss. This means that we have really reduced the overfitting to a minimum.

Let us do another test, now we will use everything as in the previous example data augmentation, batch normalization, dropout, only we will implement another layer in the $VGGNet$, and we will not insert the decay.

	precision	recall	f1 score	support
bluebell	1.00	0.58	0.73	19
buttercup	0.59	0.87	0.70	15
coltsfoot	1.00	0.75	0.86	20
cowslip	0.63	0.52	0,57	23
crocus	0.59	0.84	0,70	19
daffodil	0.76	0.62	0.68	21
daisy	0.94	0.85	0.89	20
dandelion	0.76	0.48	0.59	27
fritillary	0.78	0.88	0.82	16
iris	0.83	1.00	0.91	20
lilyvalley	0.42	0.50	0.45	20
pansy	0.95	0.82	0.88	22
snowdrop	0.89	1.00	0.94	16
sunflower	0.71	0.94	0.81	18
tigerlily	0.76	0.89	0.82	18
tulip	0.79	0.96	0.86	23
windflower	0.83	0.65	0.73	23
accuracy			0.76	340
macro avg	0.78	0.77	0.76	340
weighted avg	0.78	0.76	0.76	340

Table 8.1.3.-7 – Demonstration of the accuracy and the loss over time on the training data set and the testing data set in the form of results that are depicted within the Command Prompt, the neural network has 3 layers and batch normalization, the input data have data augmentation

* The third convolutional layer:

$$\Longrightarrow [\ CONV\ (8 \times 8 \times 128) \Longrightarrow\ ReLU\ (8 \times 8 \times 128) \Longrightarrow\ BN\ (8 \times 8 \times 128)\ ] \times 2 \Longrightarrow$$
$$\Longrightarrow\ POOL\ (4 \times 4 \times 128) \Longrightarrow\ DO\ (4 \times 4 \times 128) \Longrightarrow$$

* whereby here 128 filters the size of 3×3 act on the CONV layer, and stride 2×2 acts on the POOL layer.

$$\Rightarrow \quad FC\ (512) \Rightarrow \ ReLU\ (512) \Rightarrow \ BN\ (512) \quad \Rightarrow DO\ (512) \Rightarrow$$
$$\Rightarrow FC\ (17) \Rightarrow \ Softmax\ (17)$$

We succeeded in getting the best classification accuracy of 78%, and the neural network was trained only 15 seconds per epoch. This accuracy works well, but let us observe at what happened. The neural network was trained for a total of 20 minutes on average. And from the Figure 8.1.3.- 8 we can clearly perceive that the central change is that we have added 3 layers. This means that we have reduced the overfitting again, but in this process we have managed to maintain and even improve the accuracy of the classification.

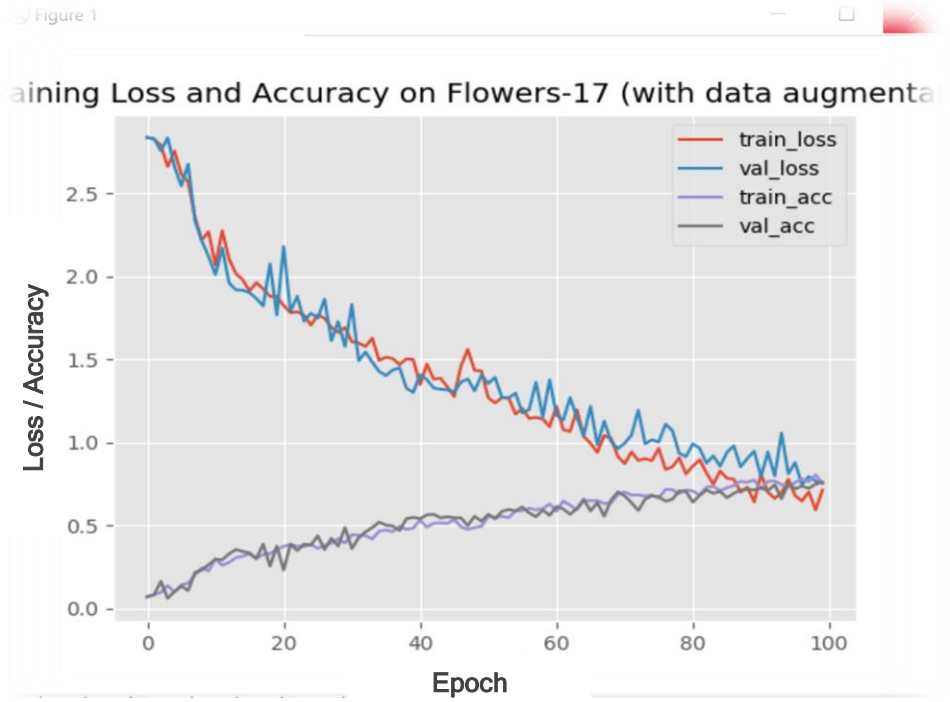


Figure 8.1.3.- 8 – Display of the accuracy and the loss over the course of time on the training data set and testing data set with data augmentation

If we now compare this with the original training at the beginning of this Chapter which gave only 66% of accuracy, we realize that we have reduced overfitting and increased accuracy by as much as 12%! We can also point out that there are large overlaps between the loss functions for the testing data and the training data, as well as, the accuracy functions. In the subsequent Chapter, we will incorporate a novel method, which does not require neural network training and elucidate how we can obtain a good classification on it.

8.2. FEATURE EXTRACTION METHOD OF THE CONVOLUTIONAL NEURAL NETWORK

The concept of transfer learning is the possibility of using the previously trained models as a "shortcut" for learning patterns from data sets on which our neural network was not originally trained.
There are two types of transfer learning within the deep learning:

- ✓ treating the neural network as an arbitrary feature extraction

- ✓ removing the fully connected layers of the existing neural network and placing a new fully connected / FC layer on top of the convolutional neural network, and fine-tuning the weights (from the previous layers as well) to identify which class our input data belong to

The first method of transfer learning is to treat the neural networks as a feature extractor, and in this Chapter we will focus exclusively on this. So far, in previous Chapters, we have treated convolutional neural networks only as image classifiers from the very beginning and the training was followed by the ensuing steps: we entered the images into the neural network, images moved along the network, we got the final classification probabilities on the output of the neural network. On the other hand, what does not exist is a "rule" that says we must allow the image to propagate with the feedforward through the entire neural network. Instead, we can discontinue spreading on an arbitrary layer, such as a pooling layer or an activation layer, to extract values from the neural network at this point, and then use them as vector feature extractors. For example, consider the $VGG16$ neural network architecture of Simonian and Ziserman. The original network will have the classification probabilities for each of the 1000 $ImageNet$ classes.

* The first convolutional layer:

$$INPUT\ (224 \times 224 \times 3)\ \Longrightarrow$$
$$\Longrightarrow\ [\ CONV\ (112 \times 112 \times 128)\ \Longrightarrow\ ReLU\ (112 \times 112 \times 128)$$
$$\Longrightarrow\ BN\ (112 \times 112 \times 128)\ ] \times 2\ \Longrightarrow\ POOL\ (56 \times 56 \times 128)$$
$$\Longrightarrow\ DO\ (56 \times 56 \times 128)\ \Longrightarrow$$

* The second convolutional layer:

$$\Longrightarrow\ [\ CONV\ (56 \times 56 \times 256)\ \Longrightarrow\ ReLU\ (156 \times 56 \times 256) \Longrightarrow\ BN\ (56 \times 56 \times 256)\ ]$$
$$\times 2\ \Longrightarrow\ POOL\ (28 \times 28 \times 256)\ \Longrightarrow\ DO\ (28 \times 28 \times 256)\ \Longrightarrow$$

* The third convolutional layer:

$$\Longrightarrow\ [\ CONV\ (28 \times 28 \times 512)\ \Longrightarrow\ ReLU\ (28 \times 28 \times 512) \Longrightarrow\ BN\ (28 \times 28 \times 512)\ ] \times 3$$
$$\Longrightarrow\ POOL\ (14 \times 14 \times 512)\ \Longrightarrow\ DO\ (14 \times 14 \times 512)\ \Longrightarrow$$

* The fourth convolutional layer:

$$\Longrightarrow\ [\ CONV\ (14 \times 14 \times 512)\ \Longrightarrow\ ReLU\ (14 \times 14 \times 512) \Longrightarrow\ BN\ (14 \times 14 \times 512)\ ] \times 3$$
$$\Longrightarrow\ POOL\ (7 \times 7 \times 512)\ \Longrightarrow\ DO\ (7 \times 7 \times 512)\ \Longrightarrow$$

* The fifth convolutional layer:

$$\Longrightarrow\ [\ CONV\ (7 \times 7 \times 512)\ \Longrightarrow\ ReLU\ (7 \times 7 \times 512) \Longrightarrow\ BN\ (7 \times 7 \times 512)\ ] \times 3\ \Longrightarrow$$
$$\Longrightarrow\ POOL\ (7 \times 7 \times 512)\ \Longrightarrow\ DO\ (7 \times 7 \times 512)\ \Longrightarrow$$

$$\Longrightarrow [\ \boldsymbol{FC\ (1 \times 1 \times 1000)\ \Longrightarrow\ ReLU\ (1 \times 1 \times 1000) \Longrightarrow\ BN\ (1 \times 1 \times 1000)}$$
$$\boldsymbol{\Longrightarrow\ DO\ (1 \times 1 \times 1000)\ ] \times 3 \Longrightarrow}$$
$$\boldsymbol{\Longrightarrow\ FC(1000)\ \Longrightarrow\ Softmax(1000)}$$

Removing the FC layers from the $VGG16$ neural network architecture and then leaving the pooling layer to be the last one i.e., the output layer from the neural network. (In the previous assessment, we remove the layer that is colored green).

When we train neural networks as feature extractor functions, we are essentially chopping the neural network at an arbitrary point (usually before the fully connected layers, although this depends on what data set we are training). Now the last layer in the neural network is the maximum pooling layer (which will have an output of the shape $7 \times 7 \times 512$, this means that we have 512 filters the size of 7×7. If we continue to move with images through this feedforward neural network, but without the last (green layer) we are left with $7 \times 7 \times 512$ that are activated or not depending on the content of the image, i.e. input data. So, we can take this data $7 \times 7 \times 512 = 25088$ and consider them as a vector feature that multiplies the contents of the image. If we repeat this procedure for a whole data set of images (including data sets on which $VGG16$ was not trained on), we will be left with a matrix of N images, each with 25088 columns utilized to quantify the content (i.e. feature vectors). This means that we can now train the Logistic Regression to recognize brand-new images i.e., data.

We keep in mind that the convolutional neural networks are not capable to recognize new classes on their own, but are only an intermediary and are used as feature extractors in this example. The classifier we learned while using machine learning will take care of the learning of these patterns. We are able to achieve a very good classification in this manner, in a very rapid time, with very little effort. The trick is to extract these features and efficiently place them inside the code.

8.2.1. HDF5_DATASET_WRITER

$HDF5$ is a binary data format for storing huge numerical data sets on disk (i.e., data sets that are too large to be easily stored within the memory). This format provides and at the same time facilitating access and computation in the rows of the data sets.
The data in $HDF5$ is stored hierarchically, much like a system stores data. Data is first defined in groups, where a group is a structure that can hold data sets, as well as other groups.

17flowers	__pycache__	__pycache__
utilities	datasets	__init__.py
extract_feature.py	io	hdf5datasetwriter.py
train_model.py	nn	
features.hdf5	preprocessing	
17flowers.cpickle	__init__.py	

Table 8.2.1.- 1 – Implementation of a $hdf5datasetwriter$

Once the group is defined, a data set can be created in the group. A data set can be viewed as a multidimensional array (i.e., a $NumPy$ array) of a homogeneous data type (integer, real number / float or unicode). $HDF5$ is written inside the C programming language, however, with the help of the $h5py$ module, we can access and use it in the python programming language. Now let us first create the $hdf5datasetwriter.py$ which we will use to convert the images from the $NumPy$ array to the $HDF5$ format.

Here we use two libraries os and $h5py$. The $HDF5DatasetWriter$ function accepts four parameters, two of which are optional. The dims parameter controls the dimensions or the shape of the data that we shall store within our data set. If we desire to store the Flowers-17 images for example, then we would set $dims = (1360, 32, 32, 3)$, because there are 1360 images in total, and each is presented as $32 \times 32 \times 3$ RGB color image.

The output of the final pooling layer / POOL layer is $512 \times 7 \times 7$, which when flattened, gives a feature extractor vector of the length $512 \times 7 \times 7 = 25088$. Therefore, when we utilize $VGG16$ as a feature extractor we set $dims = (N, 25088)$, where N is the total number of images within the data set. outputPath — this is the path to the directory where the $HDF5$ file will be stored and saved to disk. The elective $dataKey$ is a parameter that represents the name of the data set from which our algorithm will learn. The $bufSize$ controls the size of the buffer memory, and will initially have 1000 vector functions (based on the previous explanation, it is clear why).

We then form an add function that requires two parameters: rows that we will add to the data set, along with the classes. Then the rows, as well as, the classes are assigned to the buffer (which is nothing more than a parameter that connects matrices, arrays, numbers). We will also define a function called $storeClassLabels$ that will be a repository for class label array names and insert them into a separate data set.

8.2.2. FEATURE EXTRACTION

Feature extraction is used to extract features from an arbitrary data set (provided that the data entry follows a specific directory structure on the disk). Our $extract\_features.py$ code requires four parameters to extract features, two of which are optional.

Argument --dataset controls the path to our image directory where we want to use the feature extraction method. The -output argument specifies the path to our $HDF5$ output data. The --batch size argument represents the number of images in a group that will pass through the $VGG16$ at the same time. We will utilize 32, although if we had a stronger computer, we could use a larger batch. The --buffer-size switch controls the number of extracted features, which we will store in the memory before writing a buffer for the $HDF5$ data set.

Once, we extract images from the memory, we will intentionally shuffle them. In the previous examples, the division of data for testing and training was performed, before the training of classifier, i.e., our convolutional neural network. However, since the idea here is to work with data that is too large to fit into the memory, we cannot do the shuffling directly as before, but we have to do it before the feature extraction takes place. So, we actually train 75% of the data first, and then the network itself assumes that the rest of the data is the testing data.

As soon as, we extract the classes, we have to employ "$one\ hot\ encoding$" on them in order to translate the classes from string to integer. The preparation of images for the feature extraction function is performed in the same manner as the preparation for

classification via the convolutional neural networks. All of the images are taken from the disk and converted into compatible arrays, and then added to the previously formed $batchImages$. The most important thing is to add another dimension to the image matrix and subtract the mean value of the pixel intensity from the $ImageNet$ data set.

In the end, the most important thing is to vertically compare our data set as $(N, 224, 224, 3)$, where the number of batches is $N = 32$. Since, our pooling layer has a shape of $(N, 512,7,7)$, which informs us that we have 512 filters the size of 7×7, and in order to treat this as feature extractors, we need to translate it into an array of the shape $(N, 25088)$.

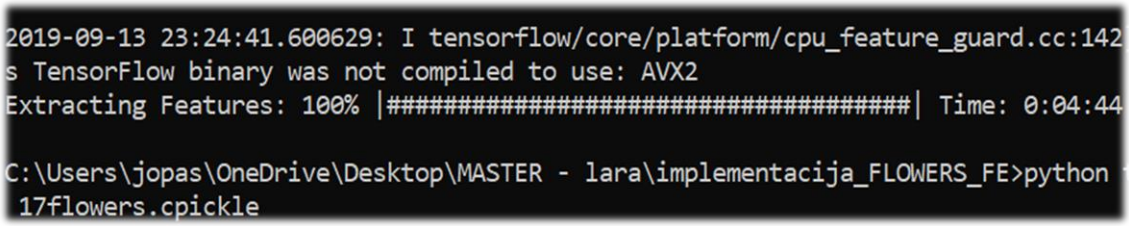


Figure 8.2.2.- 1 – Successfully applied feature extraction after only 4 minutes and 44 seconds

Finally, when we apply feature extraction, we get an 100% accuracy in under 5 minutes. If we look at the $features.hdf5$ file for Flowers-17, we shall observe that each of the 1360 images in the data set are quantified using a vector the size of 25088, the dimensions of the vector features. It is time to implement it within the main code and perceive what classification accuracy we will obtain with this method, unlike the classical methods of convolutional neural networks.

8.2.3. IMPLEMENTATION ON FLOWERS-17 DATA SET

In the subsequent, we will demonstrate how our neural network performs classification on the Flowers-17 data set, keeping in mind, of course, that the $VGG16$ convolutional neural network was previously trained on the ImageNet data set and is only used here for feature extraction. Further, we use four new $Keras$ libraries $GridSearchCV$, $pickle$, $h5py$and$LogisticRegression$. As previously pointed out, we set the parameters using the $LogisticRegression$.

We can aspect this code in more detail in the appendix if we aspire to, howbeit here we will only display the results. What is important to emphasize are the commands $db[\text{“}features\text{”}][:i]$ and $db[\text{“}labels\text{”}][:i]$, as well as, $db[\text{“}features\text{”}][i:]$ and $db[\text{“}labels\text{”}][i:]$, meaning, all of the values before i will belong to the training data set, whereas all the values after i will belong to the testing data set.

In the previous Chapter, we barely managed to get an accuracy of 78%, while here we get very easily and efficiently that the accuracy of the classification is 92%. This clearly indicates to us that the neural networks like VGG are capable of performing transfer learning, by encoding their features into the output activation functions that we may utilize in order to train custom image classifiers.

	precision	recall	f1 score	support
bluebell	0.90	0.86	0.88	21
buttercup	1.00	0.89	0.94	19
coltsfoot	1.00	0.89	0.94	18
cowslip	0.92	0.55	0,69	22
crocus	0.88	1.00	0,93	21
daffodil	0.96	0.96	0.96	25
daisy	0.88	1.00	0.93	21
dandelion	0.88	0.93	0.90	15
fritillary	1.00	0.95	0.97	19
iris	1.00	1.00	1.00	18
lilyvalley	0.85	0.89	0.87	19
pansy	1.00	0.95	0.97	20
snowdrop	0.96	0.93	0.94	27
sunflower	1.00	0.95	0.97	20
tigerlily	0.84	0.84	0.84	19
tulip	0.90	0.86	0.88	21
windflower	0.54	0.93	0.68	15
accuracy			0.92	340
macro avg	0.91	0.93	0.92	340
weighted avg	0.92	0.92	0.92	340

Table 8.2.3.- 1 – Demonstration of the accuracy and the loss over time on the training data set and the testing data set in the form of results that are depicted within the Command Prompt

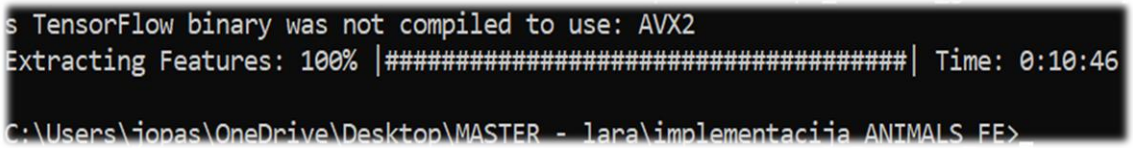


Figure 8.2.3.- 2 – Successfully applied feature extraction after 10 minutes and 46 seconds

Let us also demonstrate how the training on the animal data set that we last mentioned in Chapter 4 went. The feature extraction has been performed within 11 minutes, with an accuracy of a 100%. When we then implemented our master code we got an accuracy of 99%, which is significantly higher than the accuracy we previously got, an accuracy of 68%.

	precision	recall	f1 score	support
cat	0.99	0.99	0.99	229
dog	0.99	0.98	0.99	265
panda	0.99	1.00	1.00	256
accuracy			0.99	750
macro avg	0.99	0.99	0.99	750
weighted avg	0.99	0.99	0.99	750

Table 8.2.3.- 3 – Demonstration of the accuracy and the loss over time on the training data set and the testing data set in the form of results that are depicted within the Command Prompt

9. CONCLUSION AND OBSERVATIONS

This paper has posed the theoretical foundations of machine learning, as well as, deep learning. First, the fundamentals of images and the libraries utilized to load and preprocess them are elucidated. After that, the loss functions, classifiers and optimization methods are clarified. Further, within the optimization method, the primary focus was on the gradient descent method and the decay method. Furthermore, the composition of the neural network and the backpropagation algorithm are analyzed, since it makes a basis of the convolutional neural networks along with feedforward neural networks. Subsequently, after such a detailed introduction, the convolutional neural networks were introduced as the culmination of all this knowledge. The convolutional neural networks are presented by a detailed analysis of their layers — the convolutional layer, the pooling layer and the activation layer.

Various methods have been employed to improve the neural network performance and to prevent overfitting. Particular attention was paid to the careful selection of the shape of the convolutional neural networks, the composition, the density and arrangement of the neural network layers, as well as, the number and distribution of filters that will be used during the training process. Some of these modified methods were $shallownet$, $LeNet$, $VGGNet$. Howbeit, all of these approaches have been applied to different data sets - MNIST, Animals, Cifar-10, Smiles and Flowers-17.

The optimization methods included mini-batches (within each example), the decay method (i.e., setting the learning parameter rate), a stochastic gradient descent / SGD optimizer, and an Adam classifier, as well. The regularization methods operated were dropout (exclusion of the portions of the convolutional neural network), batch normalization, data augmentation and feature extraction. The influence of these parameters within each example was analyzed, and the neural network responses before and after the introduction of these improvements were observed in meticulous detail.

After careful consideration of the results analysis, it is not possible to determine or define the optimal method of application, whatsoever it is desirable to employ a combination of several methods. The reason is that each of the methods reacts differently to a certain set of input data. The dropout method and pooling layers have their advantages when the emphasis is on saving time, as well as, space within the memory, however, it is very demanding to distribute them properly, without risking too much disruption within the neural network.

The feature extraction technique is certainly much faster than the classical application of convolutional neural network training, nevertheless, there are many disadvantages if the neural network is not previously trained on a larger and adequate data set. In order to increase the data set the data augmentation was utilized, and it can be said that if applied correctly (i.e., different techniques of rotating, zooming, cropping of the images in order to increase the number of input data) it can significantly improve the neural network performance. Moreover, it is a technique that can almost never lead to irregularities in the classification of samples.

Finally, it is possible to make a brief summary of all that has been presented in this paper, and the conclusions that have been reached. It is transparent from all the above

examples that when training a machine learning or deep learning model, the most important thing is:

- ✓ to reduce the value of the loss function during training as much as possible, and
- ✓ that the difference in the value of the loss function on the training set and the testing set be reasonably small (i.e. that they coincide as much as possible).

The controlling of the overfitting and the underfitting of the training of the neural network model was established using optimization and regularization techniques (decay, dropout, data augmentation, batch normalization). The model can be increased by adding more layers to it or it can be reduced by removing layers, and such changes significantly affect the neural network performance and classification accuracy.

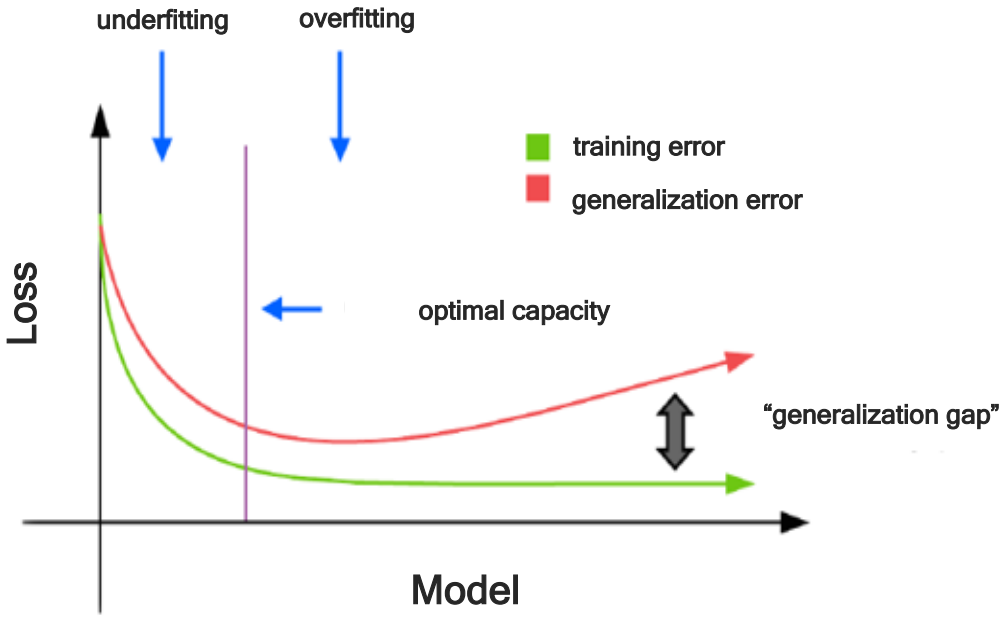


Figure 9.1.- 1 – The relationship between the model and the loss

From the Figure 9.1.- 1 it is noticeably shows that the generalization gap represents the width (i.e., the difference between the neural network training error and the generalization error) that notifies us whether the model is overfitting or underfitting. The grander the generalization gap, the larger the deviations and the model is overfitted. Moreover, when the training error of the neural network and the generalization match, the optimal model is obtained.

It is obvious from the Figure that the beginning of neural network training represents a zone of insufficient training. Then the weights are initialized randomly and the process of training the neural network by iterations begins, then by epochs, and considering the depth of the neural network and the grander number of layers, it is necessary to use the above-mentioned regularization methods. However, if we examine the Chapter 8.1, there is a unblemished example of a three-layer neural network where the decay may be superfluous, because the regularization method termed the data augmentation is already utilized in order to increase the data. The same is accurate for the dropout example,

these regularization methods should be applied carefully because sometimes discarding (droping) neural network layers can create a setback.

It is also of the dire importance to carefully choose the optimizer, as well as, the activation function, because that can be essential to obtaining a better accuracy and classification of the input data, i.e. the images. The learning rate is best chosen in precisely defined ranges, and in this paper it is accomplished so its value was in the range from $\alpha = 0.01$ to $\alpha = 0.1$. For some larger data sets, such things should be thoroughly considered and some other methods should be applied that allow us not to wait for the completion of the training process, but to stop the training process before the overfitting has occurred or perceived.

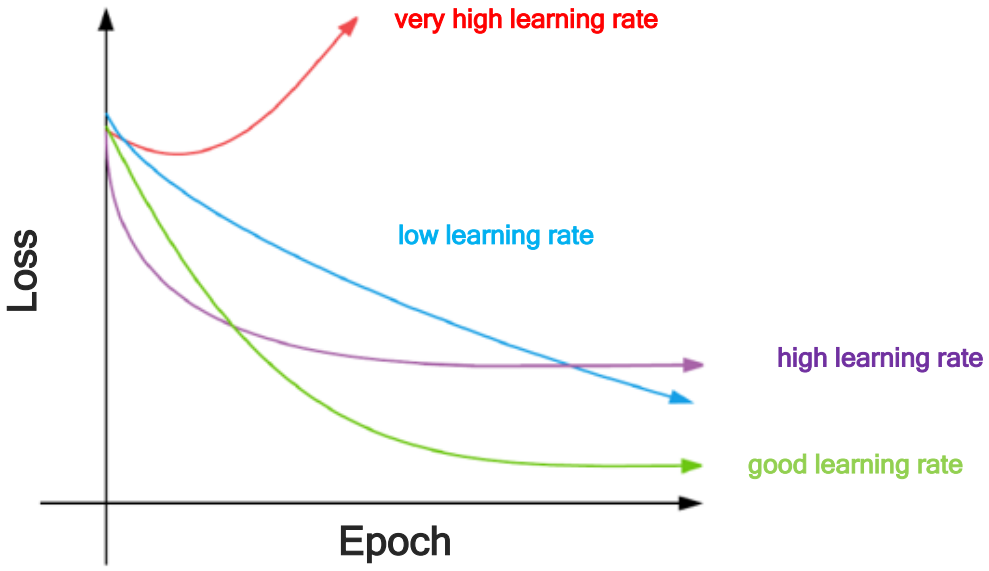


Figure 9.1.- 2 – Demonstration of the learning rates

From the Figure 9.1.- 2 it can be observed that a good learning rate will decrease exponentially, faster than a linear one, but with a lower rate than a high learning speed. This allows us to monitor the loss function.

So, in essence, the graphs depict the loss and the accuracy functions at the same time, out of a desire to make it easier to read and understand the process of the training of the convolutional neural networks. Preferably, as noted in the previous examples, the training data set and the test data set coincide in the loss function and have a small generalization gap, indicating that the neural network overfitting is minimal.

In the ensuing portion some of the codes, which were utilized during the training process of the convolutional neural networks, are depicted. It is mainly a review of the codes used in the last Chapter, since they form in some way a summation of all the knowledge within this paper.

APPENDIX — CODE

```
# simple_preprocessor.py

import cv2

class SimplePreprocessor:
    def __init__(self, width, height, interpolation=cv2.INTER_AREA):

        self.width = width
        self.height = height
                    self.interpolation = interpolation

                def preprocess(self, image):

                    return cv2.resize(image, (self.width, self.height), interp
olation=self.interpolation)
```

```
  # imagetoarray_preprocessor.py

from keras.preprocessing.image import img_to_array

class ImageToArrayPreprocessor:
    def __init__(self, data_format=None):

        self.data_format = data_format

    def preprocess(self, image):

        return img_to_array(image, data_format=self.data_format)
```

```
 # aspectawarepreprocessor.py

import imutils
import cv2

class AspectAwarePreprocessor:
    def __init__ (self, width, height, inter=cv2.INTER_AREA):

        self.width = width
        self.height = height
        self.inter = inter

    def preprocess(self,image):

        (h,w) = image.shape[:2]
        dW = 0
        dH = 0

        if w < h:
            image = imutils.resize(image, width=self.width, inter=self.inter)

            dH = int((image.shape[0] - self.height) / 2.0)

        else:
```

```
                image = imutils.resize(image, height=self.height, inter=self
.inter)
                dW = int((image.shape[1] - self.width) / 2.0)

                (h,w) = image.shape[:2]
                image = image[dH:h - dH, dW:w - dW]

        return cv2.resize(image, (self.width, self.height), interpolation=sel
f.inter)
```

```
   # __init__.py
# utilities.preprocessing

from .simple_preprocessor import SimplePreprocessor
from .imagetoarray_preprocessor import ImageToArrayPreprocessor
from .aspectawarepreprocessor import AspectAwarePreprocessor
```

```
 # simple_dataset_loader.py

import os
import cv2
import numpy as np

class SimpleDatasetLoader:

    def __init__(self, preprocessors=None):

        self.preprocessors = preprocessors

        if self.preprocessors is None:
            self.preprocessors = []

    def load(self, image_paths, verbose=-1):

        data, labels = [], []

        for i, image_path in enumerate(image_paths):
            image = cv2.imread(image_path)
            label = image_path.split(os.path.sep)[-2]

            if self.preprocessors is not None:
                for p in self.preprocessors:
                    image = p.preprocess(image)

            data.append(image)
            labels.append(label)

            if verbose > 0 and i > 0 and (i+1) % verbose == 0:
                print('[INFO]: Processed {}/{}'.format(i+1, len(image_paths))
)

        return (np.array(data), np.array(labels))
```

```
# __init__.py
# utilities.datasets

from .simple_dataset_loader import SimpleDatasetLoader
```

```
# minivggnet.py

from keras.models import Sequential
from keras.layers.normalization import BatchNormalization
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Flatten
from keras.layers.core import Dropout
from keras.layers.core import Dense
from keras import backend as K

class MiniVGGNet:
    @staticmethod
    def build(width, height, depth, classes):

        model = Sequential()
        input_shape = (height, width, depth)
        channel_dim = -1

        if K.image_data_format() == 'channels_first':
            input_shape = (depth, height, width)
            channel_dim = 1

        # First CONV => RELU => CONV => RELU => POOL layer set
        # 32 filters, 3x3 filter size
        model.add(Conv2D(32, (3, 3), padding='same', input_shape=in
put_shape))
        model.add(Activation('relu'))
        model.add(BatchNormalization(axis=channel_dim))
        model.add(Conv2D(32, (3, 3), padding='same'))
        model.add(Activation('relu'))
        model.add(BatchNormalization(axis=channel_dim))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.25))

        # Second CONV => RELU => CONV => RELU => POOL layer set
        # WE ARE LEARNING 2 SETS OF 64 FILTERS (SIZE 3x3)
        model.add(Conv2D(64, (3, 3), padding='same'))
        model.add(Activation('relu'))
        model.add(BatchNormalization(axis=channel_dim))
        model.add(Conv2D(64, (3, 3), padding='same'))
        model.add(Activation('relu'))
        model.add(BatchNormalization(axis=channel_dim))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.25))

        # Third CONV => RELU => CONV => RELU => POOL layer set
        # WE ARE LEARNING 2 SETS OF 64 FILTERS (SIZE 3x3)
        model.add(Conv2D(128, (3, 3), padding='same'))
        model.add(Activation('relu'))
        model.add(BatchNormalization(axis=channel_dim))
        model.add(Conv2D(128, (3, 3), padding='same'))
        model.add(Activation('relu'))
        model.add(BatchNormalization(axis=channel_dim))
```

```
                    model.add(MaxPooling2D(pool_size=(2, 2)))
                    model.add(Dropout(0.25))

                    # First (and only) set of FC => RELU layers
                    # dropout p = 0.5
                    model.add(Flatten())
                    model.add(Dense(512))
                    model.add(Activation('relu'))
                    model.add(BatchNormalization())
                    model.add(Dropout(0.5))
                    # Softmax classifier
                    model.add(Dense(classes))
                    model.add(Activation('softmax'))

                    return model
```

```
  # __init__.py
# utilities.nn.cnn

from .minivggnet import MiniVGGNet
```

```
    # Flowers-17 1360, 32x32 rgb images
# 17 classes - each class is 80 images
# minivggnet_flowers17.py

from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from utilities.preprocessing import ImageToArrayPreprocessor
from utilities.preprocessing import AspectAwarePreprocessor
from utilities.datasets import SimpleDatasetLoader
from utilities.nn.cnn import MiniVGGNet
from keras.optimizers import SGD
from imutils import paths
import matplotlib.pyplot as plt
import numpy as np
import argparse
import os

ap = argparse.ArgumentParser()
ap.add_argument("-d", "--dataset", required=True,
                help="path to input dataset")
args = vars(ap.parse_args())

print("[INFO] loading images...")
imagePaths = list(paths.list_images(args["dataset"]))

classNames = [pt.split(os.path.sep)[-2] for pt in imagePaths]
classNames = [str(x) for x in np.unique(classNames)]

aap = AspectAwarePreprocessor(64, 64)
iap = ImageToArrayPreprocessor()

sdl = SimpleDatasetLoader(preprocessors=[aap, iap])
```

```
34.           (data, labels) = sdl.load(imagePaths, verbose=500)
35.           data = data.astype('float') / 255.0
36.
37.           (train_x, test_x, train_y, test_y) = train_test_split(data, labels,
 test_size=0.25, random_state=42)
38.           train_y = LabelBinarizer().fit_transform(train_y)
39.           test_y = LabelBinarizer().fit_transform(test_y)
40.
41.           print("[INFO]: Compiling model....")
42.
43.           optimizer = SGD(lr=0.05)
44.           model = MiniVGGNet.build(width=64, height=64, depth=3, classes=len(
classNames))
45.           model.compile(loss="categorical_crossentropy", optimizer=optimizer,
 metrics=["accuracy"])
46.           print("[INFO]: Training....")
47.           H = model.fit(train_x, train_y, validation_data=(test_x, test_y), b
atch_size=32, epochs=100, verbose=1)
48.
49.           print("[INFO]: Evaluating....")
50.           predictions = model.predict(test_x, batch_size=32)
51.           print(classification_report(test_y.argmax(axis=1), predictions.argm
ax(axis=1), target_names=classNames))
52.
53.           plt.style.use("ggplot")
54.           plt.figure()
55.           plt.plot(np.arange(0, 100), H.history["loss"], label="train_loss")

56.           plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss"
)
57.           plt.plot(np.arange(0, 100), H.history["acc"], label="train_acc")
58.           plt.plot(np.arange(0, 100), H.history["val_acc"], label="val_acc")

59.           plt.title("Training Loss and Accuracy on Flowers-
17 (without data augmentation")
60.           plt.xlabel("Epoch #")
61.           plt.ylabel("Loss/Accuracy")
62.           plt.legend()
63.           plt.show()
64.
65.           # COMMAND PROMPT
66.           # $ cd \Users\jopas\OneDrive\Desktop\MASTER - lara\implementacija_F
LOWERS_data
67.           # $ python minivggnet_flowers17.py --dataset 17flowers
```

```
1.             # Flowers-17 1360, 32x32 rgb images
2.           # 17 classes - each class is 80 images
3.           # minivggnet_flowers17_data_aug.py
4.
5.           from sklearn.preprocessing import LabelBinarizer
6.           from sklearn.model_selection import train_test_split
7.           from sklearn.metrics import classification_report
8.           from utilities.preprocessing import ImageToArrayPreprocessor
9.           from utilities.preprocessing import AspectAwarePreprocessor
10.          from utilities.datasets import SimpleDatasetLoader
11.
12.          from utilities.nn.cnn import MiniVGGNet
13.          from keras.preprocessing.image import ImageDataGenerator
```

```
14.           from keras.optimizers import SGD
15.           from imutils import paths
16.           import matplotlib.pyplot as plt
17.           import numpy as np
18.           import argparse
19.           import os
20.
21.           def step_decay(epoch):
22.
23.               init_alpha = 0.05
24.               factor = 0.5
25.               drop_every = 5
26.
27.               alpha = init_alpha * (factor ** np.floor((1 + epoch) / drop_eve
ry))
28.
29.               return float(alpha)
30.
31.           ap = argparse.ArgumentParser()
32.           ap.add_argument("-d", "--dataset", required=True,
33.                           help="path to input dataset")
34.           args = vars(ap.parse_args())
35.
36.           print("[INFO] loading images...")
37.           imagePaths = list(paths.list_images(args["dataset"]))
38.           classNames = [pt.split(os.path.sep)[-2] for pt in imagePaths]
39.           classNames = [str(x) for x in np.unique(classNames)]
40.
41.           aap = AspectAwarePreprocessor(64, 64)
42.           iap = ImageToArrayPreprocessor()
43.           sdl = SimpleDatasetLoader(preprocessors=[aap, iap])
44.
45.           (data, labels) = sdl.load(imagePaths, verbose=500)
46.           data = data.astype('float') / 255.0
47.
48.           (train_x, test_x, train_y, test_y) = train_test_split(data, labels,
 test_size=0.25, random_state=42)
49.
50.           train_y = LabelBinarizer().fit_transform(train_y)
51.           test_y = LabelBinarizer().fit_transform(test_y)
52.
53.           aug = ImageDataGenerator(rotation_range=30, width_shift_range=0.1,
height_shift_range=0.1, shear_range=0.2, zoom_range=0.2, horizontal_flip=True, fi
ll_mode="nearest")
54.
55.           print("[INFO]: Compiling model....")
56.
57.           optimizer = SGD(lr=0.05)
58.           model = MiniVGGNet.build(width=64, height=64, depth=3, classes=len(
classNames))
59.           model.compile(loss="categorical_crossentropy", optimizer=optimizer,
 metrics=["accuracy"])
60.
61.           print("[INFO]: Training....")
62.           H = model.fit_generator(aug.flow(train_x, train_y, batch_size=32),
validation_data=(test_x, test_y), steps_per_epoch=len(train_x) // 32, epochs=100,
 verbose=1)
63.
64.           print("[INFO]: Evaluating....")
65.           predictions = model.predict(test_x, batch_size=32)
66.           print(classification_report(test_y.argmax(axis=1), predictions.argm
ax(axis=1), target_names=classNames))
67.
```

```
68.           plt.style.use("ggplot")
69.           plt.figure()
70.           plt.plot(np.arange(0, 100), H.history["loss"], label="train_loss")

71.           plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss"
)
72.           plt.plot(np.arange(0, 100), H.history["acc"], label="train_acc")
73.           plt.plot(np.arange(0, 100), H.history["val_acc"], label="val_acc")

74.           plt.title("Training Loss and Accuracy on Flowers-
17 (with data augmentation")
75.           plt.xlabel("Epoch #")
76.           plt.ylabel("Loss/Accuracy")
77.           plt.legend()
78.           plt.show()
79.
80.           # COMMAND PROMPT
81.           # $ cd \Users\jopas\OneDrive\Desktop\MASTER - lara\implementacija_F
LOWERS_data_N
82.           # $ python minivggnet_flowers17_data_aug.py --dataset 17flowers
```

```
1.              # hdf5datasetwriter.py
2.
3.            import h5py
4.            import os
5.
6.            class HDF5DatasetWriter:
7.                def __init__(self, dims, outputPath, dataKey="images",
8.                    bufSize=1000):
9.
10.             if os.path.exists(outputPath):
11.                     raise ValueError("The supplied `outputPath` already "
12.                         "exists and cannot be overwritten. Manually delete
"
13.                         "the file before continuing.", outputPath)
14.
15.                   self.db = h5py.File(outputPath, "w")
16.                   self.data = self.db.create_dataset(dataKey, dims,
17.                       dtype="float")
18.                   self.labels = self.db.create_dataset("labels", (dims[0],),

19.                       dtype="int")
20.
21.                   self.bufSize = bufSize
22.                   self.buffer = {"data": [], "labels": []}
23.                   self.idx = 0
24.
25.               def add(self, rows, labels):
26.
27.                   self.buffer["data"].extend(rows)
28.                   self.buffer["labels"].extend(labels)
29.
30.                   if len(self.buffer["data"]) >= self.bufSize:
31.                       self.flush()
32.
33.               def flush(self):
34.
35.                   i = self.idx + len(self.buffer["data"])
36.                   self.data[self.idx:i] = self.buffer["data"]
37.                   self.labels[self.idx:i] = self.buffer["labels"]
```

```
                    self.idx = i
                    self.buffer = {"data": [], "labels": []}

                def storeClassLabels(self, classLabels):

                    dt = h5py.special_dtype(vlen=str)
                    labelSet = self.db.create_dataset("label_names",
                        (len(classLabels),), dtype=dt)
                    labelSet[:] = classLabels

                def close(self):

                    if len(self.buffer["data"]) > 0:
                        self.flush()

                    self.db.close()
```

```
  # __init__.py
# utilities.io

from .hdf5datasetwriter import HDF5DatasetWriter
```

```
               # python extract_features.py

            from keras.applications import VGG16
            from keras.applications import imagenet_utils
            from keras.preprocessing.image import img_to_array
            from keras.preprocessing.image import load_img
            from sklearn.preprocessing import LabelEncoder
            from utilities.io import HDF5DatasetWriter
            from imutils import paths
            import numpy as np
            import progressbar
            import argparse
            import random
            import os

            ap = argparse.ArgumentParser()
            ap.add_argument("-d", "--dataset", required=True,
                help="path to input dataset")
            ap.add_argument("-o", "--output", required=True,
                help="path to output HDF5 file")
            ap.add_argument("-b", "--batch-size", type=int, default=32,
                help="batch size of images to be passed through network")
            ap.add_argument("-s", "--buffer-size", type=int, default=1000,
                help="size of feature extraction buffer")
            args = vars(ap.parse_args())

            bs = args["batch_size"]

            print("[INFO] loading images...")
            imagePaths = list(paths.list_images(args["dataset"]))
            random.shuffle(imagePaths)

            labels = [p.split(os.path.sep)[-2] for p in imagePaths]
            le = LabelEncoder()
            labels = le.fit_transform(labels)

```

```
37.          print("[INFO] loading network...")
38.          model = VGG16(weights="imagenet", include_top=False)
39.
40.          dataset = HDF5DatasetWriter((len(imagePaths), 512 * 7 * 7),
41.              args["output"], dataKey="features", bufSize=args["buffer_size"]
)
42.          dataset.storeClassLabels(le.classes_)
43.
44.          widgets = ["Extracting Features: ", progressbar.Percentage(), " ",

45.              progressbar.Bar(), " ", progressbar.ETA()]
46.          pbar = progressbar.ProgressBar(maxval=len(imagePaths),
47.              widgets=widgets).start()
48.
49.          for i in np.arange(0, len(imagePaths), bs):
50.
51.              batchPaths = imagePaths[i:i + bs]
52.              batchLabels = labels[i:i + bs]
53.              batchImages = []
54.
55.              for (j, imagePath) in enumerate(batchPaths):
56.
57.                  image = load_img(imagePath, target_size=(224, 224))
58.                  image = img_to_array(image)
59.
60.                  image = np.expand_dims(image, axis=0)
61.                  image = imagenet_utils.preprocess_input(image)
62.
63.                  batchImages.append(image)
64.
65.              batchImages = np.vstack(batchImages)
66.              features = model.predict(batchImages, batch_size=bs)
67.
68.              features = features.reshape((features.shape[0], 512 * 7 * 7))
69.
70.              dataset.add(features, batchLabels)
71.              pbar.update(i)
72.
73.          dataset.close()
74.          pbar.finish()
75.
76.          # COMMAND PROMPT
77.          # $ cd \Users\jopas\OneDrive\Desktop\MASTER - lara\implementacija_F
LOWERS_FE
78.          # $ python extract_features.py --dataset 17flowers --
output features.hdf5
```

```
#  train_model.py

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
import argparse
import pickle
import h5py

    ap = argparse.ArgumentParser()
    ap.add_argument("-d", "--db", required=True,
        help="path HDF5 database")
    ap.add_argument("-m", "--model", required=True,
        help="path to output model")
    ap.add_argument("-j", "--jobs", type=int, default=-1,
        help="# of jobs to run when tuning hyperparameters")
    args = vars(ap.parse_args())

    db = h5py.File(args["db"], "r")
    i = int(db["labels"].shape[0] * 0.75)

    print("[INFO] tuning hyperparameters...")
    params = {"C": [0.01, 0.1, 1.0, 10.0, 100.0, 1000.0, 10000.0]}
    model = GridSearchCV(LogisticRegression(), params, cv=3,
        n_jobs=args["jobs"])
    model.fit(db["features"][:i], db["labels"][:i])
    print("[INFO] best hyperparameters: {}".format(model.best_params_))

    print("[INFO] evaluating...")
    preds = model.predict(db["features"][i:])
    print(classification_report(db["labels"][i:], preds,
        target_names=db["label_names"]))

    print("[INFO] saving model...")
    f = open(args["model"], "wb")
    f.write(pickle.dumps(model.best_estimator_))
    f.close()

    db.close()

    # COMMAND PROMPT
    # $ cd \Users\jopas\OneDrive\Desktop\MASTER - lara\implementacija_FLOWERS_
FE
    # $ python train_model.py --db features.hdf5 --model 17flowers.cpickle
```

LITERATURE

[1] Phil Kim - “MATLAB Deep Learning: With Machine Learning, Neural Networks and Artificial Intelligence”, Apress (2017)

[2] Rumelhart, David E., McClelland, James L., Group, PDP Resear - “Parallel Distributed Processing, Vol. 1”, A Bradford Book Paperback (1994)

[3] Aurélien Géron - “Hands-On Machine Learning with Scikit-Learn and TensorFlow Concepts, Tools, and Techniques to Build Intelligent Systems”, O’Reilly Media (2017)

[4] Adrian Rosebrock - “Deep Learning for Computer Vision with Python: Starter Bundle”, PyImageSearch (2017)

[5] Radiša Jovanović – “Intelligent Control Systems” - script, Faculty of Mechanical Engineering of the University of Belgrade (2018)

[6] Hamed Habibi Aghdam, Elnaz Jahani Heravi – “Guide to Convolutional Neural Networks: A Practical Application to Traffic-Sign Detection and Classification”, Springer International Publishing (2017) – Selected Chapters

[7] Shengrong Gong, Chunping Liu, Yi Ji, Baojiang Zhong, Yonggang Li, Husheng Dong – “Advanced Image and Video Processing Using MATLAB (Modeling and Optimization in Science and Technologies)”, Springer (2018) – Selected Chapters

[8] https://www.coursera.org/ and https://www.udemy.com/ - fundamentals of python programming language, fundamentals of machine learning and deep learning with python – IBM courses

[9] https://stackoverflow.com/ - assistance with code issues

[10] Richard H. R. Hahnloser, Rahul Sarpeshkar, Misha A. Mahowald, Rodney J. Douglas, H. Sebastian Seung, “Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit” 2000.

[11] Hahnloser RH1, Seung HS, Slotine JJ. “Permitted and forbidden sets in symmetric threshold-linear networks” 2003.

[12] F. Rosenblatt, “The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain” 1985.

[13] D. E. Rumelhart, G. E. Hinton, R. J. Williams, “Learning internal representations by error propagation” 1986.

[14] Sergey Ioffe, Christian Szegedy “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift” 2015.

[15] Karen Simonyan, Andrew Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition” 2014.

[16] Yann LeCu, Leon Bottou, Yoshua Bengio, Patrick Hafner, “Gradient Based Learning Applied to Document Recognition” 1998.

[17] Viola, Jones “Rapid Object Detection using a Boosted Cascade of Simple Features” 2001.

[18] Ian Goodfellow, Yoshua Bengio, Aaron Courville – “Deep Learning” MIT Press (2016) – Selected Chapter