“Gheorghe Asachi” Technical University of Iasi
Faculty of Electronics, Telecommunications and Information Technology

Bachelor thesis

Irina Pavel

Handwritten digits recognition using artificial neural networks

Thesis supervisor: Dr. Lecturer Eng. Nicolae Cleju

Study Program: Telecommunication Technologies and Systems

2019

I declare that I wrote this Bachelor thesis by myself and using only referenced sources.

Date: Signature:

3rd of July, 2019 Irina Pavel

Table of Contents

Title: Handwritten digits recognition using artificial neural networks

Author: Irina Pavel

Department: Department of Telecommunication

Program of study: Telecommunication Technologies and Systems

Thesis supervisor: Dr. Lecturer Eng. Nicolae Cleju

Abstract

Recognition of handwritten characters, whether they are letters from the English alphabet, symbols or digits, has been one of the most active and challenging domains in the field of image processing and pattern recognition, including also artificial neural networks.

In this project we will focus on handwritten digits recognition from an image acquired using a Smartphone camera with the help of neural networks. The neural network will be trained using MATLAB Neural Network Toolbox, which will eventually be used to recognize the digits. Firstly, the image will be acquired followed by some pre-processing procedures such as conversion to gray scale, gray scale to binary conversion and normalization, or resizing, and plotted in MATLAB. After that, the neural network that has been trained will be run in order to be able to classify and recognize the class of the handwritten digit from the acquired image; the result will be displayed in the command window in MATLAB.

We mention that the neural network will be trained using MATLAB Neural Network Toolbox with MNIST Data Base that helps us with the needed dataset of handwritten images labeled by an integer from 0 to 9.

Keywords: handwritten digits, pattern recognition, MATLAB, artificial neural networks

Introduction

"*Recognition of Handwritten text has been one of the active and challenging areas of research in the field of image processing and pattern recognition. It has numerous applications which include reading aid for blind, bank cheques and conversion of any hand written document into structural text form.*"[1]

The purpose of this project is to take handwritten digits as an input data set, to process the data, train the neural network algorithm, to recognize the pattern and to display it in a live application.

Recently, the use of artificial neural network has been converted from theoretical approach to a widely-used technology with applications that come as solutions to different problems. Neural networks are particularly useful for solving problems that cannot be expressed as a series of steps, such as recognizing patterns, classifying the into groups, series prediction and data mining. Pattern recognition is one of the most used neural network tools. The neural network is presented with a target vector and a vector that contains the pattern information, a handwritten digit image, in this case. Afterwards, the neural network attempts to determine if the input data matches a pattern that it has memorized from the previous trainings.

"*The human visual system is one of the wonders of the world. Considering a random sequence of handwritten digits, 5 4 3 2 9, most of the people would recognize the digits effortlessly, at a glance, as a reflex, due to an entire series of visual cortices - V1, V2, V3, V4 and V5 – doing progressively complex image processing. But the process of visual pattern recognition suddenly becomes extremely difficult when we attempt to write a computer program to recognize digits like those above. Our simple intuition about how we recognize shapes – "a 6 has a loop at the bottom and a vertical inverted rod at the top left – turns out to be not so simple when we want to express algorithmically. It may seem hopeless.*

Neural networks approach the problem in a different way. The idea is to take a large number of handwritten digits, known as training examples, and then develop a system which can learn from those training examples. In other words, the neural network uses the examples to automatically generate rules for recognizing handwritten digits."[2]

[1] Source: *Character Recognition Using Matlab's Neural Network Toolbox* , Kuleshwar Prasad, Devvrat C. Nigam, AshmikaLakhotiya and DheerenUmre

[2] Source: *Neural Networks and Deep Learning*, Michael Nielsen

The thesis paper is divided in three main chapters. The first chapter entitled *Fundamental elements of Neural Networks* gives an introduction to the meaning of artificial neural networks. It will be explained the model of an artificial neuron, also known as perceptron, how the artificial neurons can be organized into networks and we will find out how they learn to achieve the desired output, or just simply to achieve the target.

In the second chapter, *A simple network to recognize handwritten digits*, it is presented the practical part of the thesis, that is divided in two sub-parts: simulations and the physical application. In this chapter, one can find out how to build and train the neural network that will be used to recognize the handwritten digits. The training will be made by using the MNIST Data Base and eventually, after the model has learned the patterns of the handwritten digits, it will be tested in real situations where it will receive a live image of a digit.

And in the third chapter, *Live Application*, the live application will be presented: the block diagram of the application and the steps of the whole process. There will be attached some screenshots which underlie the fact that the application is working adequately.

Chapter 1: Fundamental elements of Neural Networks

1.1 What is a neural network?

"Even though there is no general definition for the artificial neural networks, the majority of the researchers reached a common definition where they said that the artificial neural networks represent assemblies or blocks of simple processing elements that intercommunicate through communication channels by propagating numerical information.

In information technology (IT), a neural network is a system of hardware and/or software patterned after the operation of neurons in the human brain. Neural networks, also called artificial neural networks, are a variety of deep learning technology, which also falls under the umbrella of artificial intelligence, or AI."[3]

Another way of understanding the notion of artificial neural network is by taking as guidance how Simon Haykin (McMaster University Hamilton, Ontario, Canada) defines a neural network in his book (*Neural Networks: A Comprehensive Foundation*, 2nd Edition, Prentice-Hall, 1999):

"A neural network is a massively parallel distributed processor made up of simple processing units, which has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:

1. *Knowledge is acquired by the network from its environment through a learning process.*

2. *Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge."*

[3] Source: *Artificial Neural Networks,* Dr. Professor Eng. Iulian Ciocoiu

1.2 Learning process of a neural network

The way neural networks learn is similar to how people learn in their regular lives and activities: they perform an action and it is either accepted or corrected by a trainer, and in this way we understand how to get better at a certain task. Similarly, neural networks require a trainer in order to describe what should have been produced as a response to the input. Based on the difference between the actual value and the value that was outputted by the network, an error value is computed and sent back through the system (the so called cost function, we will talk about it in the following pages). For each layer of the network, the error value is analyzed and used to adjust the threshold value and the weights for the next input. In this way, the error keeps becoming marginally smaller each run as the network learns how to analyze values. A learning algorithm, also known as a learning rule or a training algorithm, can be defined as a procedure of modifying the weights and the biases of a network.
There are a few types of learning algorithms based on the desired output of the neural network, such as supervised learning, unsupervised learning, reinforcement learning.

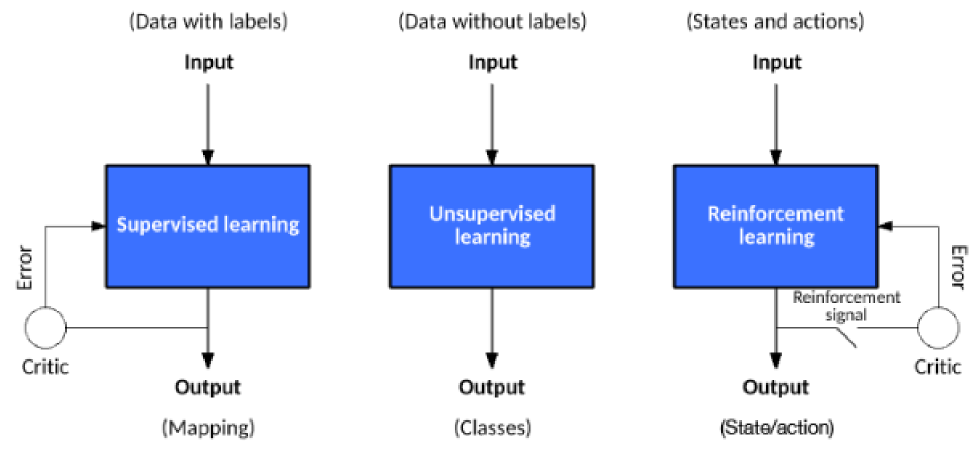


Figure 1.1: Learning models for algorithms[4]

As the name suggests, *supervised learning* occurs under the supervision of a teacher or a trainer. During the training of an artificial neural network under supervised learning, the input, having vector format, is presented to the network, which will produce an output vector. The output vector is compared with the target or the desired output. An error signal is generated if there is a difference between the actual output and the desired output. Based on this error signal, the weights should be adjusted until the actual output is matched with the desired output. It is used to predict the output based on the historical data. Being a predictive model, it is given clear instructions regarding to what it has to be learnt.

[4] Image source: *https://developer.ibm.com/articles/cc-models-machine-learning/*

In this case, the learning mode is provided with a set of examples, a training data set, of proper network behavior. The fact that the dataset is labeled provides the necessary information that the algorithm can use to evaluate its accuracy. This method of learning is usually used in applications that do not require big data sets. The perceptron learning rule falls in this supervised learning category.

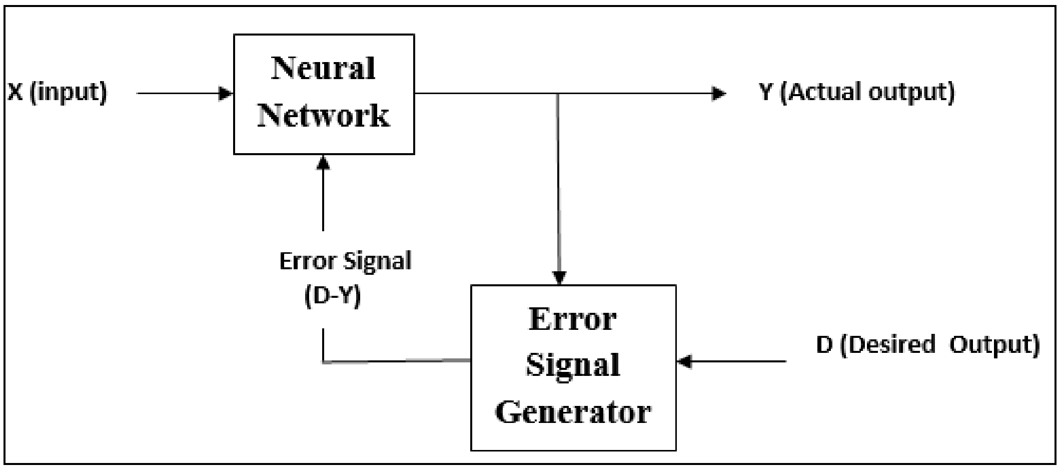


Figure 1.2: Supervised Learning[5]

Unsupervised learning is done without the supervision of a teacher. Traditional unsupervised feature selection algorithms usually assume the data instances are identically distributed and there is no dependency between them. During the training of an artificial neural network under unsupervised learning, the input vectors of similar type are combined to form clusters. When a new input pattern is applied, then the neural network gives an output response indicating the class to which input pattern belongs. In this, there would be no feedback from the environment as to what should be the desired output and whether it is correct or incorrect. Hence, in this type of learning the network itself must discover the patterns, features from the input data and the relation for the input data over the output.

Most of these algorithms perform clustering operations providing unlabeled data that the algorithm tries to makes sense of by extracting features and patterns on its own.

[5] Image source: *https://www.tutorialspoint.com/artificial_neural_network/artificial_neural_network_supervised_learning.htm*

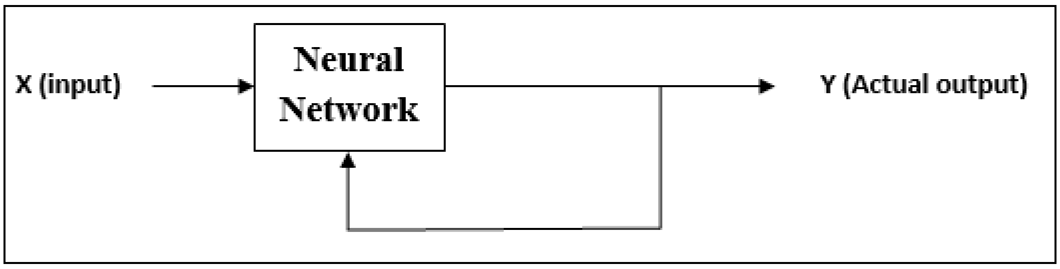


Figure 1.3: Unsupervised Learning[6]

If an artificial neural network is being trained under *reinforcement learning,* it receives some response from the environment. Yet, the feedback is estimative and not instructive as in the case of supervised learning. Based on this evaluative response received from the environment, the network adjusts the weights to obtain better precision in future. This learning algorithm is similar to supervised learning but the difference is that the model will have access to less information. If we take a look to the block diagram shown in Figure 1.4, we can deduce that reinforcement learning trains the algorithm with a reward system, the block "Error Signal Generator" provides some feed-back when an artificial intelligence agent performs the best action in a particular situation to accomplish the specific goal or to improve on a specific task. In this situation, the learning algorithm works as a causal system, because to make its choices, the agent learns from its past experience.

The overall aim: predict the best next step to take to earn the biggest final reward.

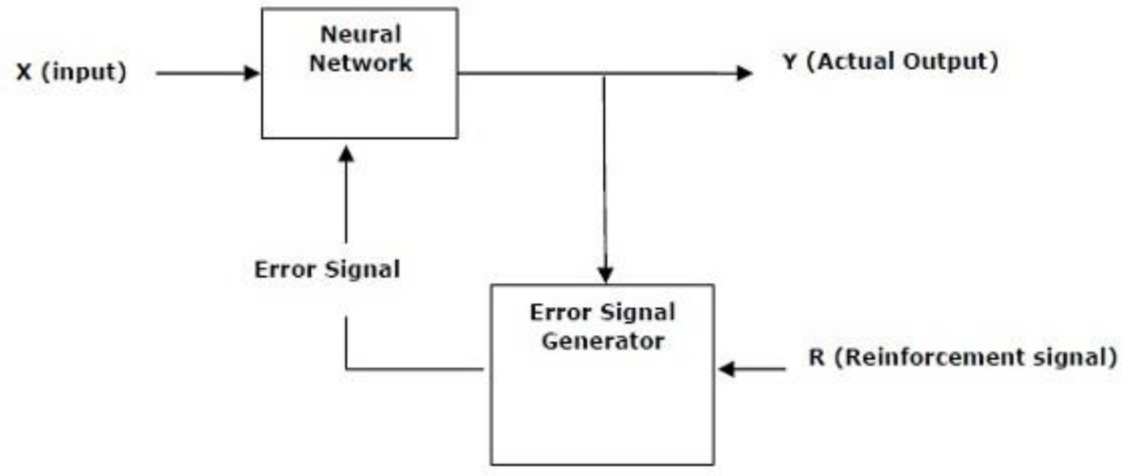


Figure 1.4: Reinforcement Learning[4]

[3] Image source: *https://www.tutorialspoint.com/artificial_neural_network/index.htm*

Important Note!

Even though they seem to be alike, there is a slight difference between supervised learning and reinforcement learning. Usually, reinforcement learning involves learning by interacting with an environment, collecting the information from states and actions. A reinforcement algorithm agent learns from its past experience, rather from its continual try-out and error learning process in comparison to supervised learning where an external supervisor provides labeled examples.

1.3 Logistic Regression

The classic application of logistic regression model is binary classification. Logistic regression is useful if we are working with a dataset that is relatively (very) small in size. Thus, it is a great model for simple classification tasks. Neural networks are somewhat related to logistic regression. Basically, we can think of logistic regression as a one layer neural network.

We want to model a function that can take in any number of inputs and constrain the output to be between 0 and 1. We can associate this with a neuron, armed with the sigmoid activation function.

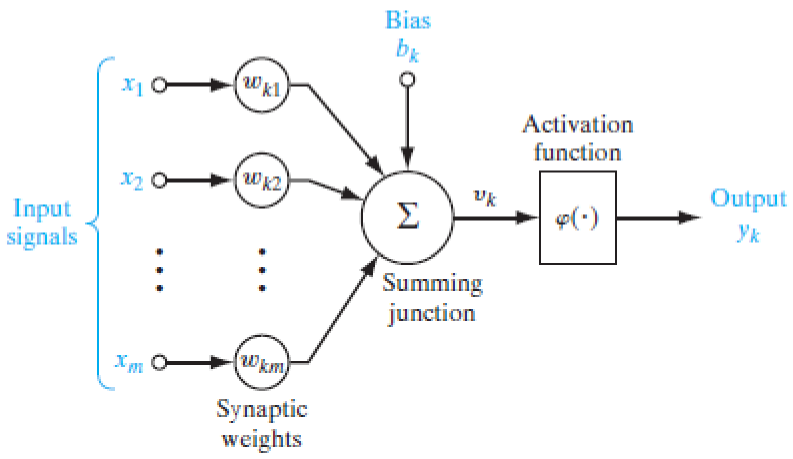


Figure 1.5: Nonlinear model of a neuron, labeled k[7]

If logistic regression is the best performing classifier, then this idea can be extended to neural networks. In terms of its structure, logistic regression can be thought as a neural network with no hidden layer, and just one output node.

[7]Image source: *Neural Networks and Learning Machines*, Third Edition, by Simon Haykin

1.3.1 Single Layer Perceptron (SLP)

"From biological point of view, a neuron is a specialized cell transmitting nerve impulses; a nerve cell. Therefore, we can define an artificial neuron, also called a perceptron, as an information-processing unit that is fundamental to the operation of a neural network. Perceptrons were developed in the 1950s and 1960s by the scientist Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts."[8]

"The simplest type of layered neural network has a single layer of weights connecting the inputs and output. In this way, it can be considered the simplest kind of feed-forward network, where the information always moves in one direction; it never goes backwards. This network architecture is also known as single-layer perceptron.

The perceptron shown in Figure takes several binary inputs, x_1, x_2, x_3 and produces a single binary output:

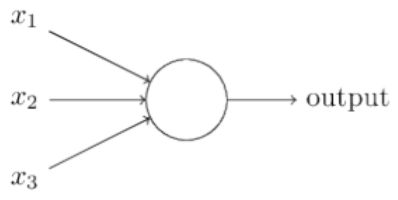


Figure1.6: Single-layer perceptron[9]

Rosenblatt proposed a simple rule to compute the output. He introduced *weights*, w_1, w_2, …,w_n, real numbers expressing the importance of the respective inputs to the output. The neuron's output, 0 or 1, is determined by whether the weighted sum $\sum_j w_j \cdot x_j$ is less than or greater than some *threshold value*. Just like the weights, the threshold is a real number which is a parameter of the neuron.

$$\text{output} = \begin{cases} 0 \; if \sum_j w_j \cdot x_j \leq \text{treshold} \\ 1 \; if \sum_j w_j \cdot x_j \geq \text{treshold} \end{cases}$$

From mathematical point of view, we can describe a neuron, using Figure (the one with blue), as follows:

[8]Source: *https://skymind.ai/wiki/neural-network*
[9] Image source: *Neural Networks and Deep Learning*, Michael Nielsen

$$u_k = \sum_{j=1}^{n} w_{kj} \cdot x_j$$

and

$$y_k = \varphi(u_k + b_k)$$

where $x_1, x_2, \ldots, x_n$ are the input signals; $w_{k1}, w_{k2}, \ldots, w_{kn}$ represent the weights of the neuron; u_k is the linear output due to the input signals; b_k is the bias; $\varphi(\cdot)$ is the activation function and y_k is the output signal.

When talking about neural networks, to each input provided to the artificial neuron, a weight is associated. Weight increases the steepness of the activation function, in other words, the gradient becomes relatively high. This means weight decides how fast the activation function will trigger whilst the bias is used to delay the triggering of the activation function.

The bias is an additional external parameter and it focuses on applying an affine transformation to the linear output u_k. It is used to adjust the output along the weighted sum of the inputs to the neuron to fit the prediction with the data better.”[10]

1.3.2 Activation function

Activation function decides whether a neuron should be activated or not by calculating weighted sum and further adding bias with it. The purpose of the activation function is to introduce non-linearity into the output of a neuron.

A neural network has neurons that work in correspondence of *weight, bias* and their respective activation function. In a neural network, we would update the weights and biases of the neurons on the basis of the error at the output. This process is known as *back-propagation*. Activation functions make the back-propagation possible since the gradients are supplied along with the error to update the weights and biases.

[10]Source: *Neural Networks and Deep Learning*, Michael Nielsen

1.3.2.1 Why do we need Non-linear activation functions?

A neural network without an activation function is essentially just a linear regression model. The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks.

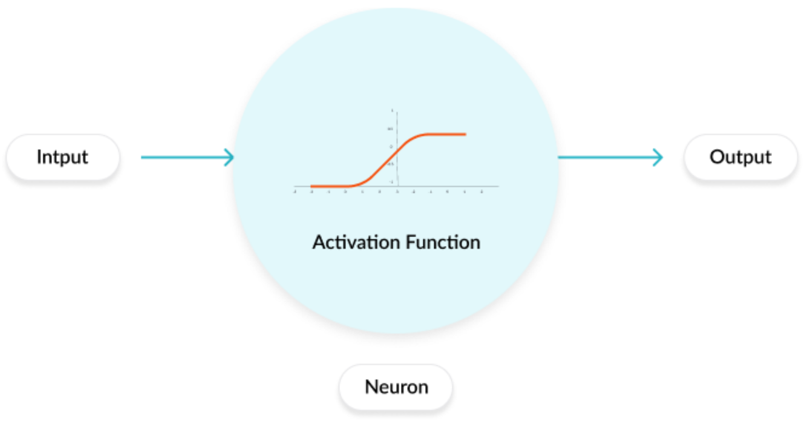


Figure 1.7: Activation function[11]

The activation function takes into account the interaction effects in different parameters and does a transformation after which it gets to decide which neuron passes forward the value into the next layer.

Let's consider a neuron as follows:

$$Y = \sum_j w_{kj} \cdot x_j + b_k, \text{ where } Y \in (-\infty, +\infty)$$

The neuron calculates a weighted sum of its inputs, adds a bias and then it decides whether it should be "fired" or not. At this point, the neuron does not know the bounds of the output value.

A selection of commonly used activation functions for artificial neurons are concisely presented in Figure 1.8:

[11]Image source: *https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/*

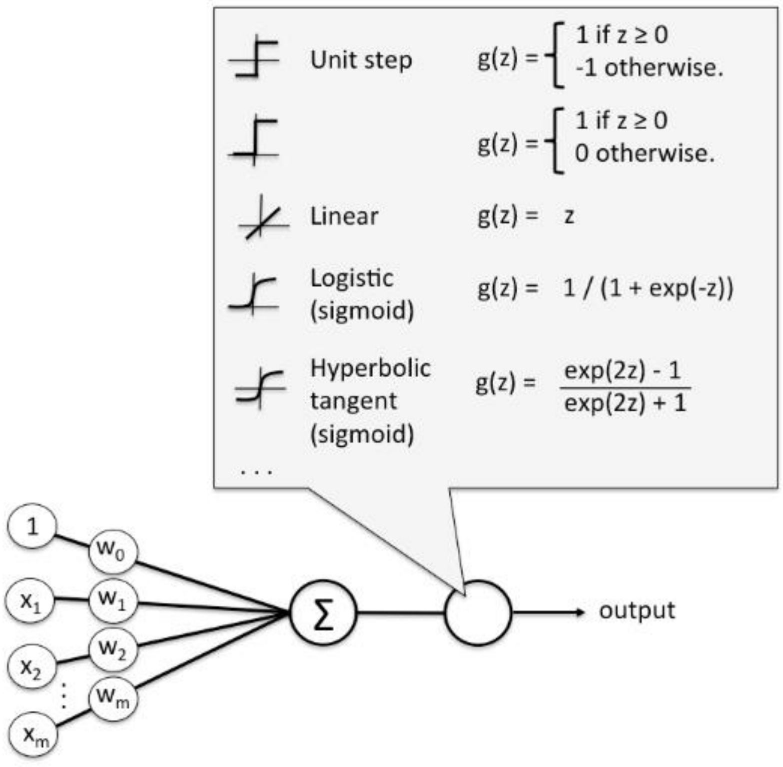

Unit step
g(z) = { 1 if z ≥ 0, -1 otherwise.
g(z) = { 1 if z ≥ 0, 0 otherwise.
Linear
g(z) = z
Logistic (sigmoid)
g(z) = 1 / (1 + exp(-z))
Hyperbolic tangent (sigmoid)
g(z) = (exp(2z) - 1) / (exp(2z) + 1)
. . .
1
x1
x2
xm
w0
w1
w2
wm
Σ
output

Figure 1.8: Commonly used activation functions for artificial neurons[12]

A. Threshold function (Binary Step function or Unit Step function)

A binary step function is also known as either binary step function or unit step function and is generally used in the perceptron linear classifier.

From mathematical point of view, it can be described with the following expression:

$$\varphi_1 (z) = \begin{cases} 1 & if z \geq 0 \\ 0 & otherwise \end{cases} \qquad \varphi_2 (z) = \begin{cases} 1 & if z \geq 0 \\ -1 & otherwise \end{cases}$$

Figure 1.9 indicates that either the neuron is fully active or not. However, this function is not differentiable which is quite vital when using the back-propagation algorithm.

The problem with a step function is that it does not allow multi-value outputs, for example, it cannot support classifying the inputs into one of several categories, therefore, this activation

[12]Image source: *https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f*

function is useful when the input pattern can only belong to one of two groups, for example binary classification.

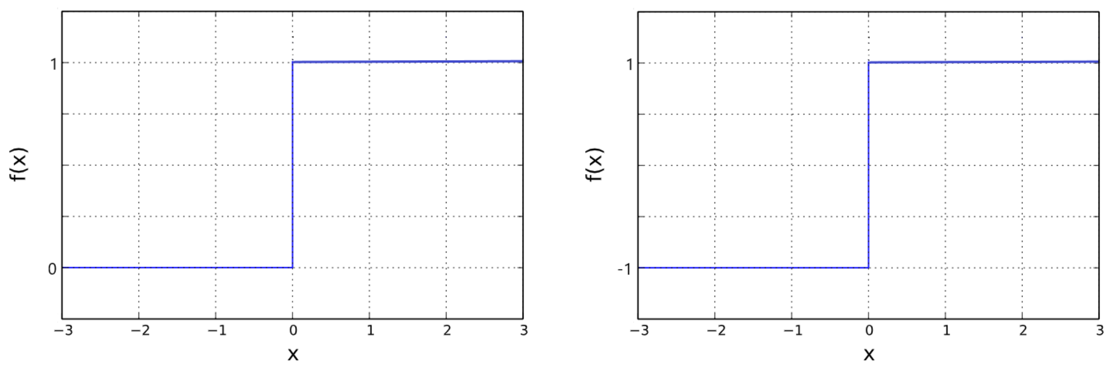


Figure 1.9: Threshold activation functions

B. Linear Function

The linear activation function is a straight line function where activation is proportional to input (which is the weighted sum from the neuron). This way, it gives a range of activations, so it is not binary activation so we can definitely connect a few neurons together.

Let's define the mathematical expression as f(x) = c·x, derivative with respect to x is c, a constant. That means, the gradient has no relationship with x. It is a constant gradient and the descent is going to be on constant gradient. If there is an error in prediction, the changes made by back-propagation are constant and not depending on the changes made at the input. Each layer is activated by a linear function. That activation in turn goes into the next level as input and the second layer calculates weighted sum on that input and it in turn, fires based on another linear activation function.

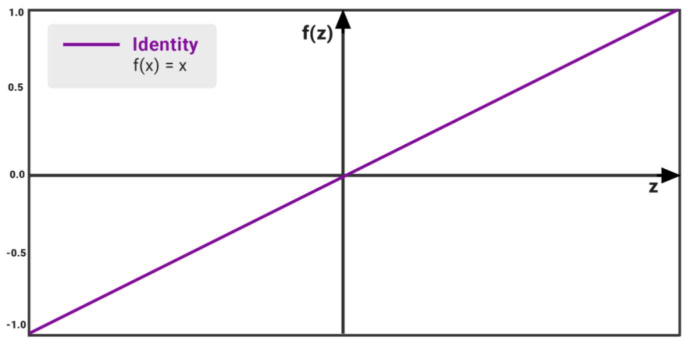


Figure 1.10: Linear Activation Function[13]

[13]Image source: *https://towardsdatascience.com/activation-functions-in-neural-networks-83ff7f46a6bd*

C. Logistic Function (Sigmoid Function)

The sigmoid function is a logistic function bounded by 0 and 1, as with the threshold function, but this activation function is continuous and differentiable.

The function has a graph that is shaped as the "S" letter. The sigmoid function is a common form of activation function used in the construction of neural networks. It is defined as a strictly increasing function that exhibits a graceful balance between linear and nonlinear behavior. An example of the sigmoid function is the *logistic function*, defined by the following equation:

$$\sigma(z) = \frac{1}{1+e^{(-z)}}$$

where α is the slope parameter of the function.

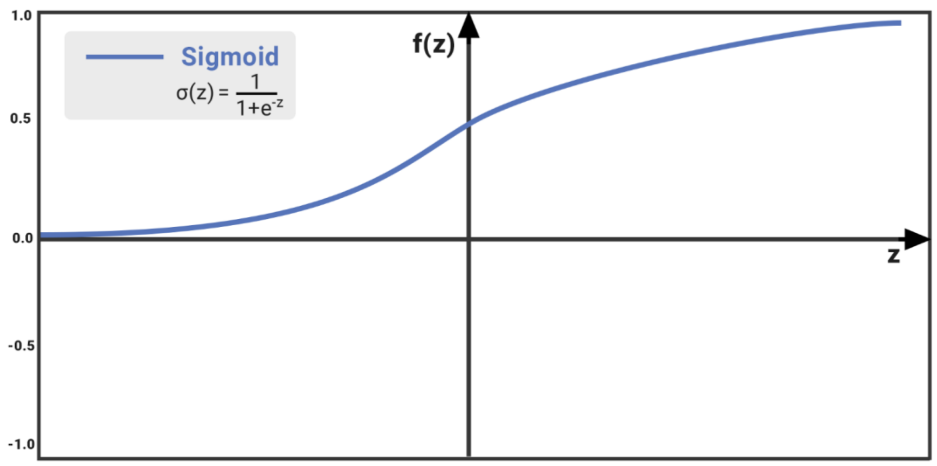


Figure 1.11: Sigmoid Activation Function[14]

D. Hyperbolic Tangent Function (Sigmoid Function)

$$\tanh(z) = \frac{2}{1+e^{(-2z)}}$$

This function enables activation functions to range from -1 to +1.

[14]Image source: *https://towardsdatascience.com/activation-functions-in-neural-networks-83ff7f46a6bd*

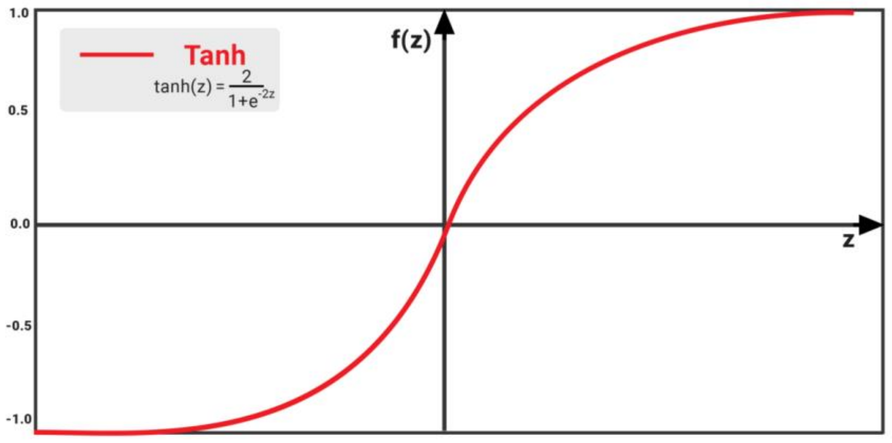


Figure 1.12: Hyperbolic Tangent Activation Function[15]

E. Rectified Linear Activation function (ReLU)

ReLUs are the smooth approximation to the sum of many logistic units and produce sparse activity vectors. Below is the equation of the function:

R(z) = max(0, z); $y = max\{0, b + \sum_{i=1}^{k} x_i \cdot w_i\}$

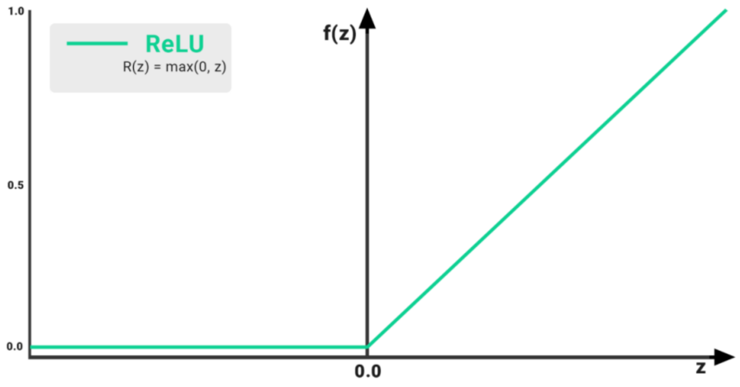


Figure 1.13: ReLU Activation Function[16]

[15]Image source: *https://towardsdatascience.com/activation-functions-in-neural-networks-83ff7f46a6bd*
[16]Image source: *https://towardsdatascience.com/activation-functions-in-neural-networks-83ff7f46a6bd*

1.3.3 Loss Function

A fundamental feature with respect to the learning process of the neural networks is the purpose for which they are used. Therefore, for applications using logistic regression the main goal is to model probability density of the output values (target) accustomed to the input data. On the other hand, when we talk about classification applications, as in our case, recognition of the handwritten digits, it is intended to estimate the probabilities that the input variables belong to one of the available categories. Achieving these goals is made possible by optimizing cost features conveniently defined according to neural network parameters, with the observation that the two types of applications typically require the use of specific cost functions.
To better understand the loss function, also named as cost function, let's do an analysis using the squared loss error cost function, as presented in the following formula:

$$L(\hat{y}, y) = \frac{1}{2\text{m}} \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

Where $\boldsymbol{\hat{y}}^{(i)}$ is the expected value or the hypothesis, briefly said , the target; $\boldsymbol{y}^{(i)}$ is the actual value.

A cost function is something we want to minimize. For example, the cost function might be the sum of squared errors over your training set. Gradient descent is a method for finding the minimum of a function of multiple variables. Cost function can be also defined as:

$$J(w, b) = \frac{1}{2\text{m}} \sum_{i=1}^{m} f(y - h(w^T x))^2$$

Where **h(z)** is hypothesis function, or the expected value of the output, **m** is the number of samples; in case of logistic regression, h(z) = $\frac{1}{1+e^{-z}}$.

An illustration of a cost function $J(w,\ b)$ is depicted in the figure below. The cost function has a clear minimum located for a particular set of values ***w*** and ***b***, for which the cost is minimal. The training procedure aims to find these optimal values, using an iterative approach.

J is a surface above the horizontal axis which can be seen also as the height of the surface.

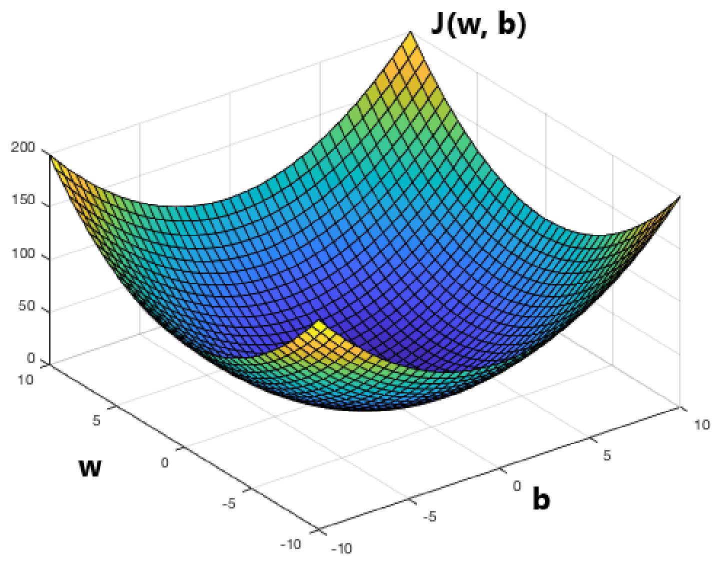


Figure 1.14: The cost function[17]

1.3.1.4 Gradient Descent

As mentioned in the previous section, gradient descent is used to minimize the loss function. Gradient descent is an iterative process that occurs in the backpropagation phase where the goal is to continuously resample the gradient of the model's parameter in the opposite direction based on the weight ***w***, updating consistently until we reach the global minimum of the cost function ***J(w)***.

$$J(w,b) = \frac{1}{\mathrm{m}}\sum_{i=1}^{m} f(\hat{y}^{(i)}, y^{(i)})$$

$$\nabla J(W) = \left(\frac{dJ}{dw_1}, \frac{dJ}{dw_2}, \dots, \frac{dJ}{dw_N}\right)$$

$$\mathrm{W{:} = W - \alpha \nabla J\ (W)}$$

Where **α** is the learning rate, also called the learning step; $\boldsymbol{\frac{dJ}{dw_k}}$ is the derivative of the cost function with respect to **w** parameter, the weight of the model, it represents the slope.

An illustration of Gradient Descent is depicted in Figure 1.15 below. Starting from an initial value w_0, the gradient descent step will update the value towards minimizing the loss function, yielding value w_1. This is an iterative process. At the next iteration, the loss function is computed again, and

[17] Image source: *https://pythonmachinelearning.pro/complete-guide-to-deep-neural-networks-part-1/*

the value w_1 is updated once more towards minimizing the loss function, yielding the value w_2. After a large number of steps, the value of *w* will reach the value which minimizes the loss.

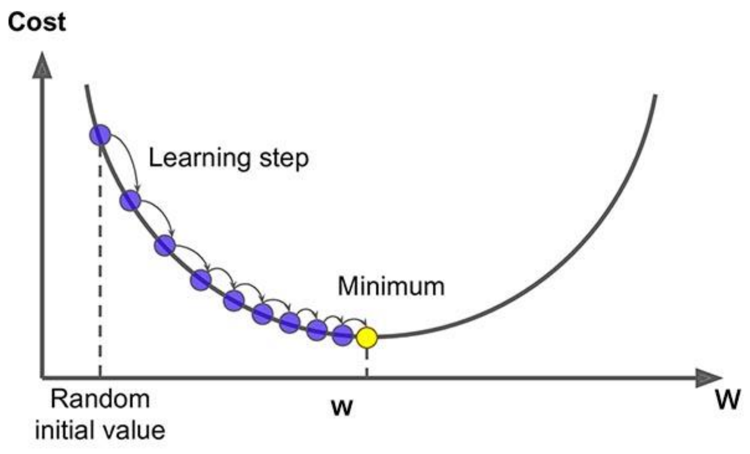


Figure 1.15: Schematic of Gradient Descent[18]

Note that the learning steps taken must be of small value in order for the gradient to converge (see Figure 1.16); otherwise the minimum point might be exceeded, which can lead to the instability of the whole process, it will diverge out of control (see Figure 1.17).

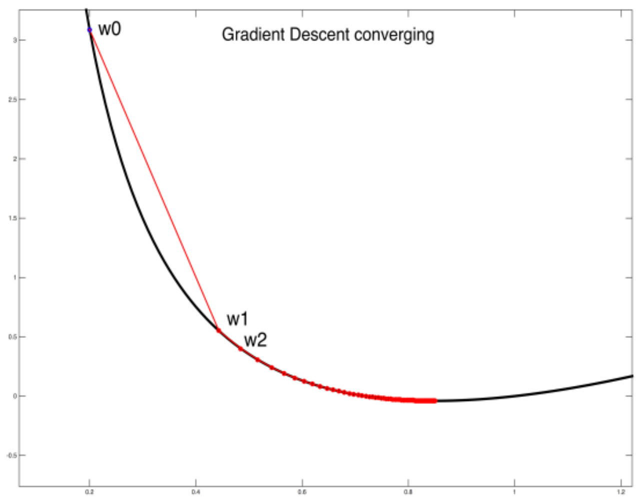


Figure 1.16: Small learning rate[19]

[18] Image source: *https://saugatbhattarai.com.np/what-is-gradient-descent-in-machine-learning/*
[19] Image source: *http://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote07.html*

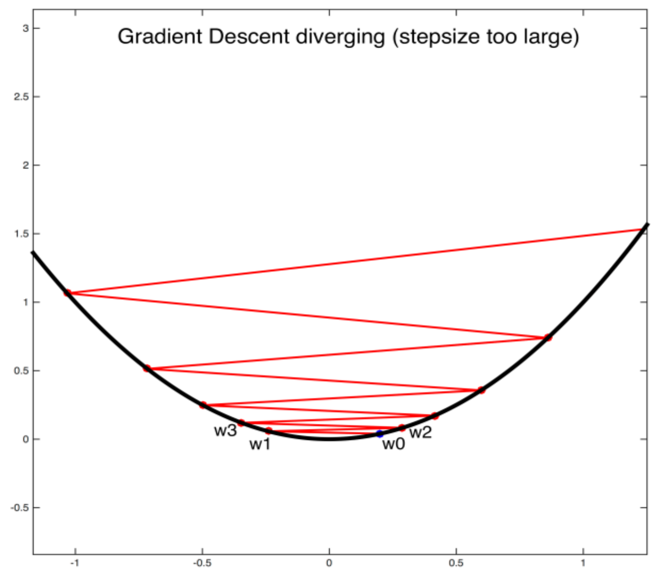


Figure 1.17: Large learning rate[20]

For gradient descent to converge to a global minimum, it has to be a convex function. But as we defined J(w) in the previous subchapter, it is a concave function. Therefore, we will apply logarithm function to J(w) as follows:

→ We have the mathematical equation of the cost function:

$$J(w) = \frac{1}{2\text{m}} \sum_{i=1}^{m} f(y - h(w^T x))^2$$

→ We apply *log* function to remodel the cost function for logistic regression, so that it will be a convex function satisfying all constraints, and the result is:

$$\text{J(w)} = \frac{1}{2\text{m}} \sum_{i=1}^{m} f\left[ylog\left(h(w^T x)\right) + (1-y)\log(1-h(w^T x)\right)$$

[20] Image source: *http://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote07.html*

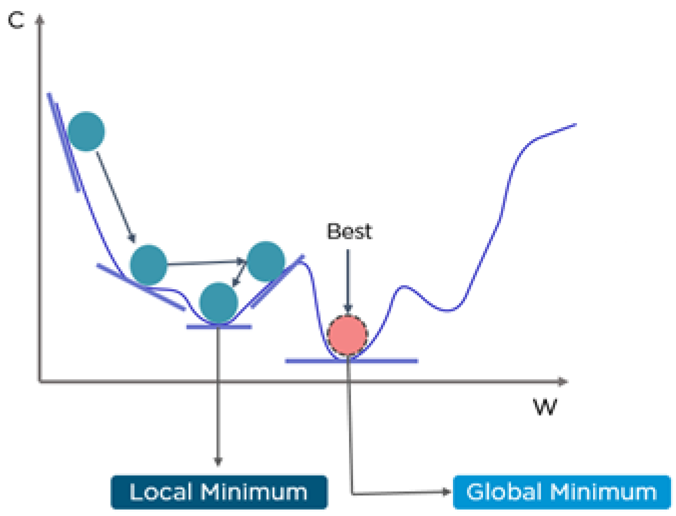


Figure 1.18: Local minimum vs. Global minimum

The *local minimum* of the cost function regarding the gradient descent has the following property: the loss function evaluates to a greater value at every other point in a neighborhood around the local minimum (a neighborhood in this case can correspond to a "ball" around the minimum) than the local minimum itself.

In different circumstances, the *global minimum* of the cost function minimizes the function on its entire domain, not only on a neighborhood of the minimum. In other words, the function evaluated at the global minimum is less than or equal to the function evaluated at any other point.

The reason convex optimization problems are highly respected in machine learning is because convexity guarantees a global minimum.

1.4 Shallow Networks

Shallow neural networks are non-deep feedforward neural networks. Besides an input layer and an ouput layer, a shallow neural network has intermediate layers, also be called hidden layers or encoders. Shortly said, shallow neural networks is a term used to describe neural networks that usually have only one hidden layer when, on the other hand, deep neural networks have several hidden layers, often of various types.
Shallow neural networks can also be described as multi-layer perceptrons. A multi-layer perceptron, also known as feed-forward neural networks, consists of a sequence of layers each fully connected to the next one.

As we can see from Figure 1.19, the architecture of an MLP can be characterized as follows:

- *input layer***:** brings the initial data into the system for further processing by subsequent layers of artificial neurons.
- *hidden layer*: a layer in between input layers and output layers, where artificial neurons take in a set of weighted inputs and produce an output through an activation function.
- *output layer***:** the last layer of neurons that produces given outputs for the program.

Each input layer feature is interconnected with every hidden layer feature.

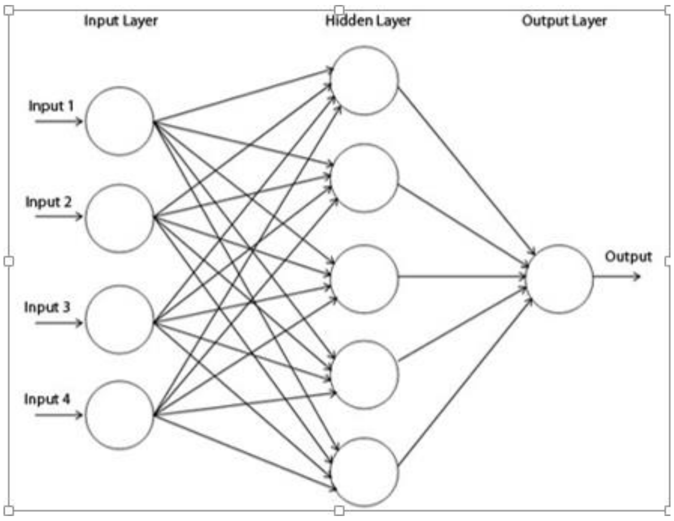


Figure 1.19: Multilayer perceptron[21]

Each layer functions as described in the previous subchapters (activation, loss function, gradient descent).

All the above network models were so called feedforward neural networks, this means that output from one layer is used as input to the next layer. Briefly, there are no loops in the network. The information is always fed forward and never fed back. In the case of a feedback loop, the models are called recurrent neural networks which are less influential than feedforward networks, the main reason for that is that the learning algorithms are less powerful.

[21]Image source:
https://www.analyticsvidhya.com/blog/2016/08/evolution-core-concepts-deep-learning-neural-networks/

Chapter 2: A simple network to recognize handwritten digits

2.1 A short introduction

If in the previous chapter we have seen mostly theoretical notions about neural networks, let’s see a brief neural network architecture used to recognize handwritten digits, see Figure 2.1.

Our main target is to design an application which recognizes handwritten digits. The goal is to determine the correct digit (0, 1,…, 9) given an input image acquired with a phone camera. In other words, we want to classify the images into 10 classes, one class for each digit. This may seem a simple application, but once you try it in real life, there can appear a lot of unexpected factors that may affect the way the application works. Each new handwritten digit can have its own little variations, so using a fixed representation of a handwritten digit will not result in a good accuracy. However, machine learning is data-driven, and we can apply it to solve our problem. In particular, we’ll be applying neural networks.

Fortunately, regarding the data set with which the neural network will be trained, we will not have to go out and collect this data ourselves. In fact, there’s a very famous dataset, called MNIST that we will be using. In the earlier days, it was used for training the first modern convolutional neural network.

2.1.1 Objectives

The main objectives for this project are as follows:

- We want to provide an easy user interface to input the object image
- The image should be easily acquired with a phone camera
- The system should be able to process the given input to suppress the background
- The system should be able to detect the digit region present in the image
- The system should be able to display the present digit in the image and plot it to the user
- The system should be able to recognize the digit from the image and display it in the command window in MATLAB

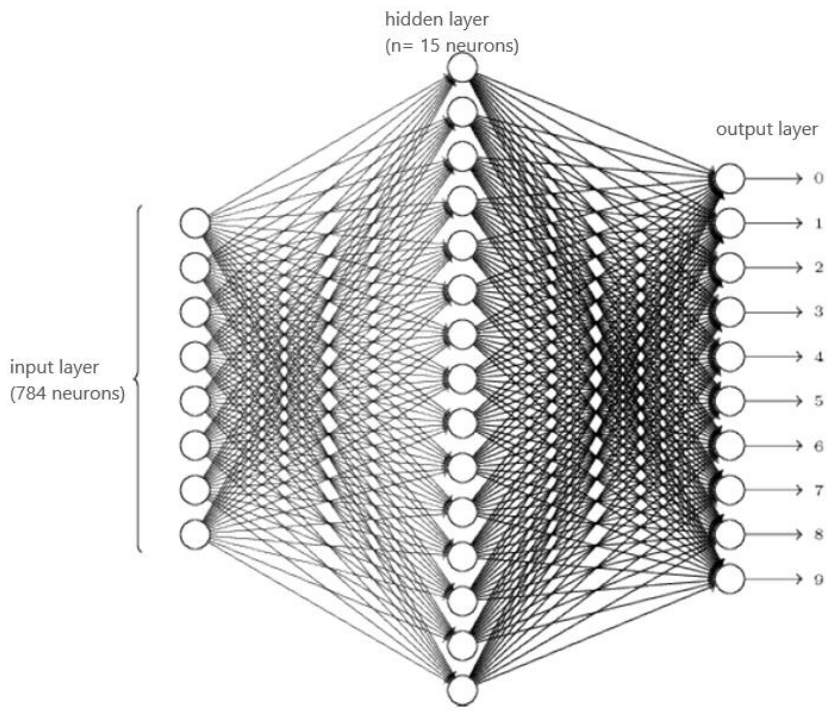


Figure 2.1: The architecture of a neural network with 15 neurons and a hidden layer[22]

The input layer of the network contains neurons encoding the values of the input pixels. Our training data for the network will consist of many 28 by 28 pixel images taken from MNIST Data Base, we will talk about it in the next sub-chapter, and so the input layer contains 784 = 28·28 neurons. The input pixels are grey scale, with a value of 1.0 representing white, a value of 0.0 representing black, and in between values representing gradually darkening shades of grey. The second layer of the network is a hidden layer, containing just n = 15 neurons, in this example. The output layer of the network contains 10 neurons. If the first neuron fires, i.e., has an output equal to 1, or a value closer to 1, then that will indicate that the network thinks the digit is a 0. If the second neuron fires then that will indicate that the network thinks the digit is a 1. And so on. A little more precisely, we number the output neurons from 0 through 9, and figure out which neuron has the highest activation value.

[22]Image source: *Neural Networks and Deep Learning*, Michael Nielsen

2.2 The MNIST Data Base

For this application, we decided to use the well-known MNIST Data Base. This is a very popular database used in evaluation of neural networks applications, containing a large number of images of handwritten digits, preprocessed and ready to use for the training process.
A detailed description of the database is presented below.

"The MNIST database (Modified National Institute of Standards and Technology Data Base) is a large database of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of learning of handwritten digits. It has a training set of 60,000 examples and a test set of 10,000 examples of images. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image.

The original black and white images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. The images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field.

With some classification methods (particularly template-based methods, such as SVM and K-nearest neighbors), the error rate improves when the digits are centered by bounding box rather than center of mass. There have been a number of scientific papers on attempts to achieve the lowest error rate; one paper, using a hierarchical system of convolutional neural networks, manages to get an error rate on the MNIST Data Ba00se of 0.23%. The original creators of the database keep a list of some of the methods tested on it. In their original paper, they use a support vector machine to get an error rate of 0.8%. An extended dataset similar to MNIST called EMNIST has been published in 2017, which contains 240,000 training images, and 40,000 testing images of handwritten digits and characters. "[23]

2.2.1 File formats for the MNIST Data Base:

"The data is stored in a very simple file format designed for storing vectors and multidimensional matrices. General info on this format is given at the end of this page, but you don't need to read that to use the data files.

All the integers in the files are stored in the MSB first (high endian) format used by most non-Intel processors. Users of Intel processors and other low-endian machines must flip the bytes of the header.

There are 4 files:

[23]Source: *http://yann.lecun.com/exdb/mnist/*

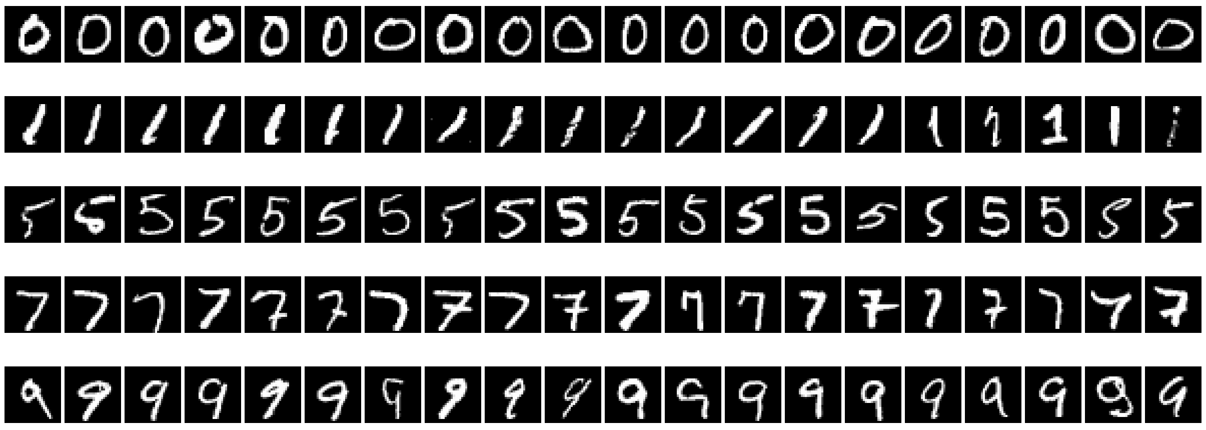
train-images-idx3-ubyte: training set images
train-labels-idx1-ubyte: training set labels
t10k-images-idx3-ubyte: test set images
t10k-labels-idx1-ubyte: test set labels

The training set contains 60000 examples, and the test set 10000 examples. The first 5000 examples of the test set are taken from the original NIST training set. The last 5000 are taken from the original NIST test set. The first 5000 are cleaner and easier than the last 5000."[24]

The term *endian* refers to denoting or relating to a system of ordering bytes in a word, or bits in a byte, in which the most significant (or least significant) item is put first. Big-endian, or high-endian, is an order in which the "big end" (most significant value in the sequence) is stored first (at the lowest storage address). Little-endian, or low-endian, is an order in which the "little end" (least significant value in the sequence) is stored first.[25]

These images are scanned handwriting samples from 250 people, half of whom were US Census Bureau employees, and half of whom were high school students.

Examples of images from the MNIST Data Base:



2.3.Data Pre-processing

Since the images in MNIST Data Base are preprocessed, we don't need to do this step, we can take the images as they are, ready to be trained. However, for the live image acquisition and preprocessing using a phone, we will need to apply all these steps.

[24]Source: *http://yann.lecun.com/exdb/mnist/*

[25]Source: *https://searchnetworking.techtarget.com/definition/big-endian-and-little-endian*

The images have already undergone the following proprocessing steps:

→ *segmentation:* the digits are cropped from a larger image and appear in the center of the image.

→ *resizing*: all images have fixed size: 28 by 28 pixels.

→ *binarization:* the background is converted to black and the foreground to white; initially, the image was with white background and black foreground.

For our MNIST dataset, we have an image, not an input vector. So we have to convert our 2D images into 1D vectors. This can be done by flattening the 2D images as shown in the figure bellow:

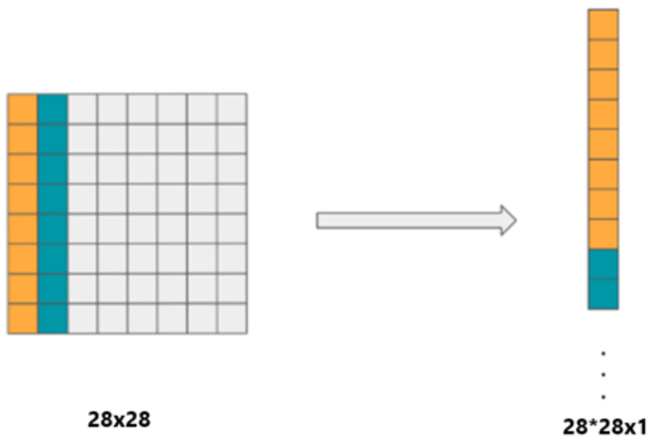


Figure 2.2: Convertion of 2D images to 1D vectors[26]

To better understand the process of the convertion, we take the second row represented with blue and we pin it on to the first one; we pin on the third row to the concatenation of the first two and so on. By concatenation we refere to a series of interconnected events; connections, as in a chain. Carrying on, we end up with a 1D vector. This is how the input data will be presented to the neural network. Since our MNIST dataset has 28 x 28 pixeled images, they are converted into 28 ·28 x 1 = 784 x 1 vectors, which is exctly the size of the input layer.

Observation!

As in every application, there are negative aspects to this approach; by converting 2D images to 1D vectors, we lose information. There is a type of neural network tailored for images called a *convolutional neural networks* where there is no need to flatten the input image. These tend to do better at image tasks than regular neural networks. This is a short observation, we will not use

[26]Image source: *https://pythonmachinelearning.pro/complete-guide-to-deep-neural-networks-part-1/*

convolutional neural networks, despite their performances, they are more complex and hard to work with.

2.4 Training the neural network

Patternet recognition networks are feedforward networks that can be trained to classify inputs according to target classes. To train the networks, Neural Net Pattern tool from MATLAB is used by applying the command *nprtool()*in the command window to open it.
In the next steps, the Input data that will be presented the network and the Target data defining the desired network ouput is selected. To be mentioned that the samples used are built in the form of matrix columns and there are three kinds of samples:

→ *Training set:* These are presented to the network during training, and the network is adjusted according to its error. It used for computing the gradient and updating the network weights and biases

→ *Validation set:* These are used to measure network generalization, and to halt (to stop) the training when generalization stops improving. The error on the validation set is monitored during the training process. The validation error normally decreases during the initial phase of training, as does the training set error. However, when the network begins to overfit the data, the error on the validation set typically begins to rise. The network weights and biases are saved at the minimum of the validation set error.

→ *Testing set:* These have no effect on training and so provide an independent measure of network performance during and after training. The test set error is not used during training, but it is used to compare different models. It is also useful to plot the test set error during the training process. If the error on the test set reaches a minimum at a significantly different iteration number than the validation set error, this might indicate a poor division of the data set.

The selected samples are divided in percentages as follows: 70% training samples, 15% validation samples, and 15% testing samples. Values are taken from literature and they are mostly used in this combiantions, usually, the highest percentage must be assigned to the training set.

An important step for the training process is creating the architecture of the network by setting, first of all, the number of the hidden neurons used in each layer. The architecture of the used network is presented in Figure 2.2.

The architecture from Figure 2.2 is just one example where the hidden layer has only 10 neurons. Since the project is about the recognition of the handwritten digits, the value 10 of the output layer is pretty much self explanatory.

All this information can be found in the official technical MATLAB documentation. (*https://ww2.mathworks.cn/help/deeplearning/gs/classify-patterns-with-a-neural-network.html*)

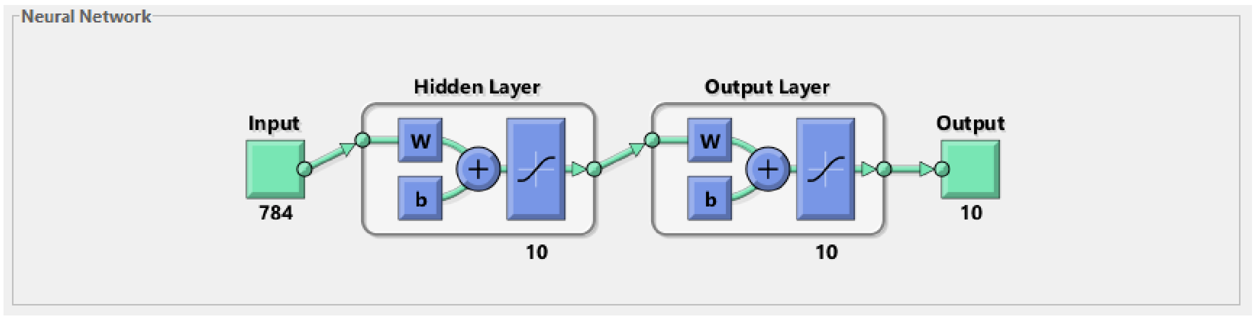


Figure 2.2: Block diagram of the neural network implemented in Matlab

The above figure shows a two-layer feed-forward network, with sigmoid hidden and output neurons, given 10 neurons in its hidden layer that will be trained with scaled conjugate gradient back-propagation, as follows:

- Input: the network receives a 28x28 pixel image, hence 28x28=784 input pixel values. Thus, the input layer has 784 inputs.

- Hidden Layer: in this example, it has 10 neurons. The neuron calculates a weighted sum of its inputs, adds a bias and then it decides whether it should be “fired” or not. It can be observed that a sigmoid activation function is used to decide whether a neuron should be activated or not by calculating weighted sum and further adding bias with it.

- Output Layer: the last layer of neurons that produces given outputs for the program.

- Output: the network outputs 10 values, corresponding to the probabilities that the input image belongs in the 10 digit classes

2.5 Presentation and interpretation of the results

In this subchapter the results of the simulations are going to be presented using some simple figures such as graphs, Confusion Matrices, Receiver Operating Characteristic (ROC) and Mean Square Error (MSE) from MATLAB R2018b program. The simulations were taken with respect to the number of the data set and the number of hidden neurons. In all the simulations only one hidden layer of neurons was used, the main reasons why only one hidden layer was used is the RAM memory of the computer and the simulation time.

2.5.1 Confusion Matrix

Results in a classification problem are usually illustrated in the form of a *Confusion Matrix.* A Confusion Matrix is a performance measurement for machine learning classification problem where the output can be two or more classes. It is a table with 4 different combinations of predicted and actual values. In other words, a confusion matrix can be defined as a table used to describe the performance of a classification model on a set of test data for which the true values are known.

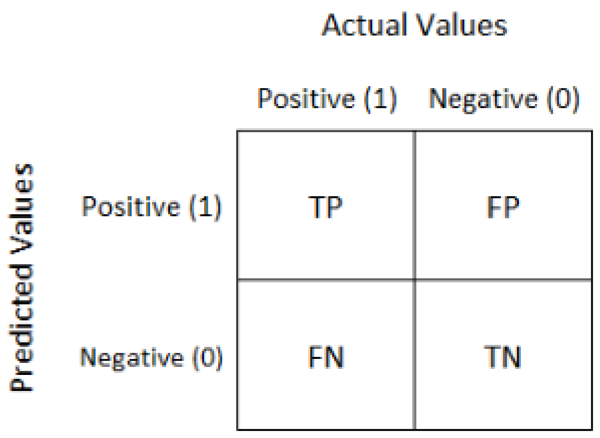


Figure 2.3: Example of Confusion Matrix[27]

A simple example of a Confusion Matrix can be depicted in Figure 2.3. Let's analyze the terms that define the cells of the table (TP, TN, FP, FN), therefore, they can be exaplained as follows

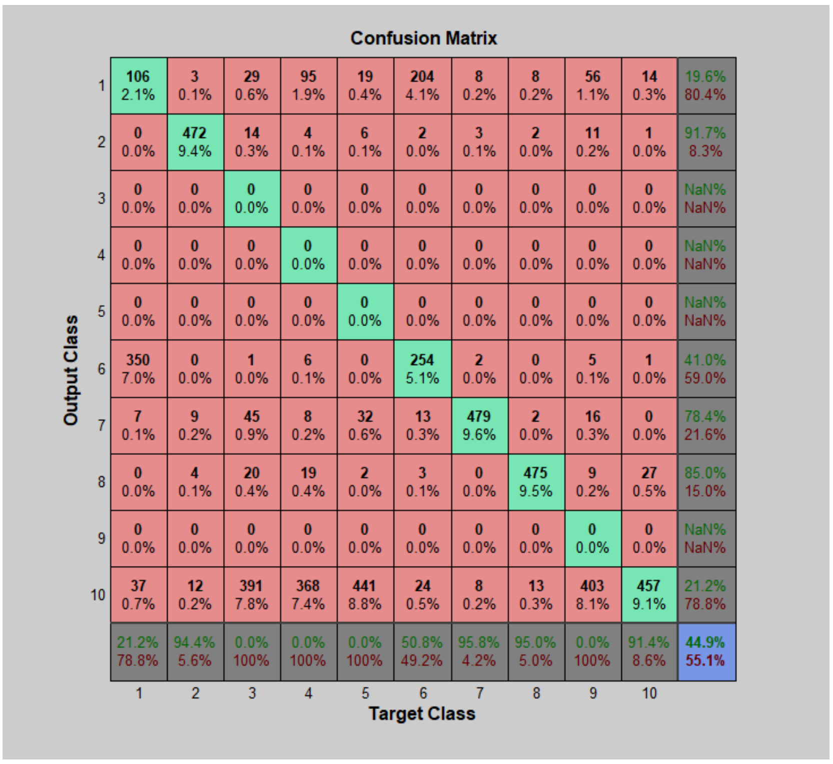
- → *True Positive (TP):* You predicted positive and it's true.
- → *True Negative (TN):* You predicted negative and it's true.
- → *False Positive (FP),* also known as *Type 1 Error:* You predicted positive and it's false.
- → *False Negative (FN),* also known as *Type 2 Error*: You predicted negative and it's false.

A good classification result will have large values on the main diagonal of the matrix (approaching 1), and close to zero values in the rest of the matrix.

Multiple simulations have been done with different sizes of the input data set and the following Confusion Matrices, Mean Squared Error (MSE) and Receiver Operating Characteristic (ROC) curves for the three kinds of data combined (training, testing and validation) were generated.

The first set of simulations was made with an input data set of 500 images modifying the number of hidden neurons at a time to 10 hidden neurons, 30 hidden neurons and 80 hidden neurons.

[27]Image source: *https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62*



Confusion Matrix

Output Class	1	2	3	4	5	6	7	8	9	10	
1	106 2.1%	3 0.1%	29 0.6%	95 1.9%	19 0.4%	204 4.1%	8 0.2%	8 0.2%	56 1.1%	14 0.3%	19.6% 80.4%
2	0 0.0%	472 9.4%	14 0.3%	4 0.1%	6 0.1%	2 0.0%	3 0.1%	2 0.0%	11 0.2%	1 0.0%	91.7% 8.3%
3	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	NaN% NaN%
4	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	NaN% NaN%
5	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	NaN% NaN%
6	350 7.0%	0 0.0%	1 0.0%	6 0.1%	0 0.0%	254 5.1%	2 0.0%	0 0.0%	5 0.1%	1 0.0%	41.0% 59.0%
7	7 0.1%	9 0.2%	45 0.9%	8 0.2%	32 0.6%	13 0.3%	479 9.6%	2 0.0%	16 0.3%	0 0.0%	78.4% 21.6%
8	0 0.0%	4 0.1%	20 0.4%	19 0.4%	2 0.0%	3 0.1%	0 0.0%	475 9.5%	9 0.2%	27 0.5%	85.0% 15.0%
9	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	NaN% NaN%
10	37 0.7%	12 0.2%	391 7.8%	368 7.4%	441 8.8%	24 0.5%	8 0.2%	13 0.3%	403 8.1%	457 9.1%	21.2% 78.8%
	21.2% 78.8%	94.4% 5.6%	0.0% 100%	0.0% 100%	0.0% 100%	50.8% 49.2%	95.8% 4.2%	95.0% 5.0%	0.0% 100%	91.4% 8.6%	44.9% 55.1%

Target Class

Figure 2.4: Confusion Matrix for 500 images per class and 10 hidden neurons per layer. One can see that the results are not very good (many classes have 0% recognition rate).

Let's analyze the Confusion Matrix for a data set of 500 images, one hidden layer with 10 neurons. The diagonal cells show the number of cases that were correctly classified, for example, in the case of the digit 0, only 106 images out of 500 have been classified correctly, having a 21.2% validation percentage. The off-diagonal cells show the misclassified cases. The blue cell in the bottom right shows the total percent of correctly classified cases (in green) and the total percent of misclassified cases (in red).

We can conclude from the first attempt to train the neural network that the results are poor and we need more accuracy. Therefore, there are a few steps that we can take into account for better performances:

- We can reset the initial network weights and biases to new values and train again.
- We can increase the number of training vectors.
- *We can increase the number of hidden neurons.*
- *We can increase the number of input values, if more relevant information is available.*

In our case, the last two approaches fit better and it can be observed that the results improve along the simulations as it can be seen in the next pages.

Observation!

Each time a neural network is trained, the output can result in a different value due to different initial weight and bias values and different divisions of data into training, validation, and test sets. As a consequence, different neural networks trained on the same problem can give different outputs for the same input. To make sure that of the good accuracy of the neural network, the model should be retrained several times.

2.5.2 Mean Square Error (MSE)

"The performance of a trained ANN is usually measured in terms of the so-called validation error, which is calculated in the same way as the training error but on entirely new data set that has not been part of the network training. This means that when comparing networks that have been trained using different error functions the validation error is the common measure to assess performance in terms of accuracy and computational effort."[28]

Mean Square Error is just as it says: The *mean* (average) magnitude of the *squares* of the *error:* the distance between the model's estimate of the test values and the actual test value.

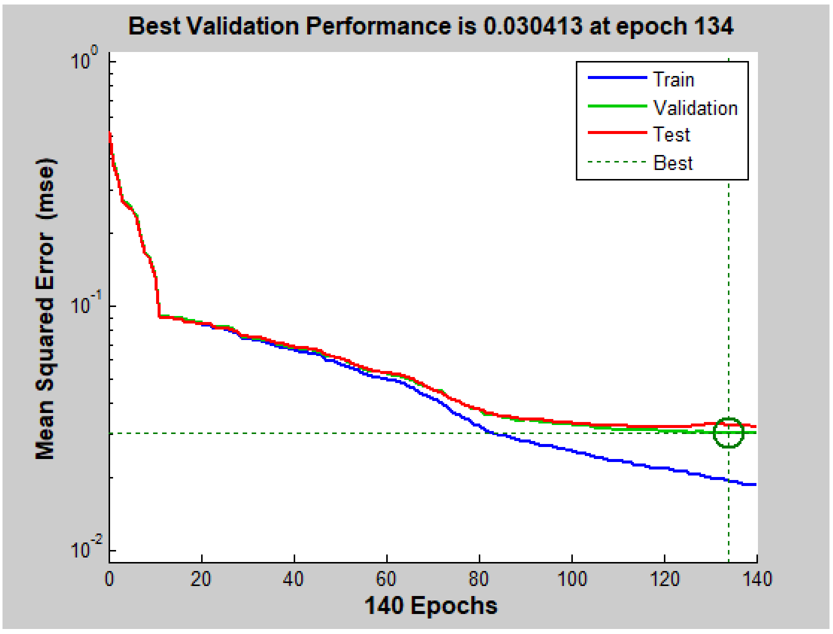


Figure 2.5: Mean Square Error plot for 500 images per class for a one hidden network with 10 neurons

[28]Source: *https://www.hindawi.com/journals/jam/2014/759834/*

Let’s analyze the Mean Square Error plot for a data set of 500 images, one hidden layer with 10 neurons. Firstly, we will define the meaning of *epochs*. In the official documentation and literature, there can be found two ways of defining this term:

→ An epoch is the presentation of the set of training (input and/or target) vectors to a network and the calculation of new weights and biases. Note that training vectors can be presented one at a time or all together in a batch.

→ An epoch is a measure of the number of times all of the training vectors are used once to update the weights. For batch training all of the training samples pass through the learning algorithm simultaneously in one epoch before weights are updated.

In figure 2.5, there are three main lines that represent the three sets used by the neural network:

→ the blue, represented by line the training set

→ the green line, represented by the validation set

→ the red line, represented by the test set

→ the dashed line, that represents the best behavior of the model with respect to the mean squared error and the number of epochs.

The main purpose of a plot performance is to obtain the plot of train record error values against the number of training epochs. In this example, from Figure 2.5, when the neural network was trained, the Best Validation Performance was found at epoch 134 with the value 0.030413, even though, at the end of training, the total number of epochs was 140. Generally, the error reduces after more epochs of training, but might start to increase on the validation data set as the network starts overfitting the training data. In the default setup, the training stops after six consecutive increases in validation error, and the best performance is taken from the epoch with the lowest validation error.

2.5.3 Receiver Operating Characteristic (ROC)

When we need to analyze and to verify the performance of the multi - class classification problem, we use *Area Under the Curve* (AUC) *Receiver Operating Characteristic (*ROC) curve. It is also known in literature as AUROC (*Area Under the Receiver Operating Characteristic*).
A Receiver Operating Characteristic, or ROC, is a plot of the true positive rate against the false positive rate for the different possible cut points of a diagnostic test. It is one of the most important evaluation metrics for checking any classification model’s performance. ROC is a probability curve and it tells how the much model is capable of distinguishing between classes. The higher the Area Under the Receiver Operating Characteristic curve, the better the model is at predicting.

“An ROC curve demonstrates several things:

- It shows the tradeoff between sensitivity and specificity (any increase in sensitivity will be accompanied by a decrease in specificity).
- The closer the curve follows the left-hand border and then the top border of the ROC space, the more accurate the test.

- The closer the curve comes to the 45-degree diagonal of the ROC space, the less accurate the test.
- The slope of the tangent line at a cut point gives the likelihood ratio (LR) for that value of the test.
- The area under the curve is a measure of text accuracy”[29]

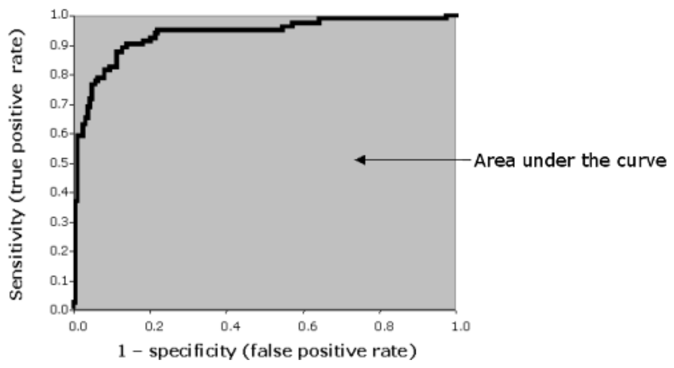


Figure 2.6: Example of AUC-ROC[30]

To explain the plotted AUC-ROC from Figure is important to have understood the terms TP, TN, FP, FN that were defined in the section *2.5.1 Confusion Matrix*. Let’s see how the terms used in AUC and ROC curve are defined:

→ *True Positive Rate (TPR),* also known as *Sensitivity* or *Recall:*

$$\text{TPR} = \frac{TP}{TP+FN}$$

→ *Specificity:*

$$\text{Specificity} = \frac{TN}{TN+FN}$$

→ *False Positive Rate (FPR):*

$$\text{FPR} = 1 - \text{Specificity}$$

$$\text{FPR} = \frac{FP}{TN+FP}$$

A model with excellent behavior or performances has Area Under the Curve near to 1, that being said, it means that it has a good measure of separability. A model with poor performances has AUC near to 0, which means it has worst measure of separability. And in the case when AUC is 0.5, or around 0.5, it means that the model has no capacity to separate the classes between them, this is

[29]Source: *http://gim.unmc.edu/dxtests/roc2.htm*

[30]Image source: *https://acutecaretesting.org/en/articles/roc-curves-what-are-they-and-how-are-they-used*

the worst scenario that can happen, the model has no discrimination capacity to distinguish between positive class and negative class.
Briefly, as a rule of categorization to describe an ROC curve, the following table can be used:

AUROC	**Category**
0.9 - 1.0	Very Good
0.8 - 0.9	Good
0.7 - 0.8	Fair
0.6 – 0.7	Poor
0.5 – 0.6	Fail

Table 2.1: Categorization of ROC curves

Now let's analyze the ROC presented in Figure 2.6 for our first set of simulations, 500 images per class, one hidden layer neural network with 10 neurons.

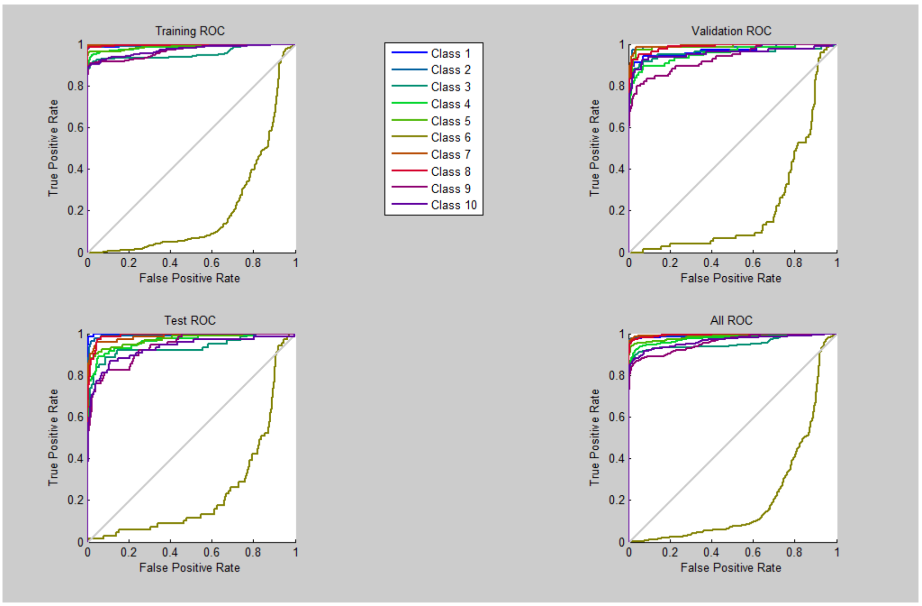


Figure 2.7: Receiver Operating Characteristic curves for 500 images per class for a one hidden network with 10 neurons

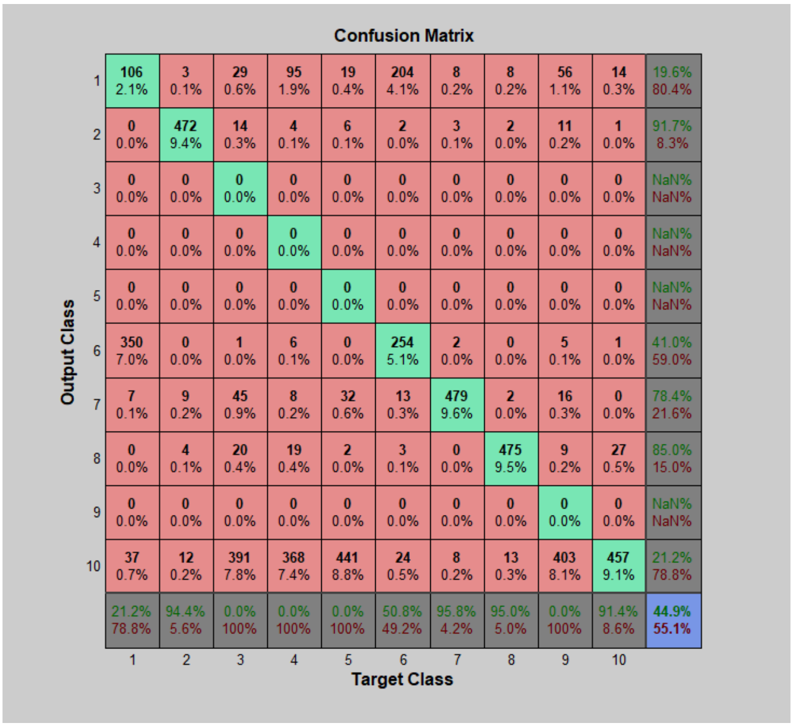
At a first sight, it can be easily noticed that the images from class 6 were not classified as expected; the AUC assigned to class 6 tends to 0 rather than 1, resulting a failure in all the cases: validation, training and testing.
In the case of *Training ROC*, all the ROC curves, except the curve for class 6, are above the value 0.8 on the True Positive Rate axis, tending to value 1, leading us to the conclusion of good and very good behavior of the model.

In the case of *Validation ROC*, the situation is similar, all the curves are above 0.8 value on TPR axis, leading to a good and very good behavior, the exceptions being the curve for class 9 that has the smallest AUC, hence, the model has a fair response.

In the case of *Test ROC*, class 1, class 2, class 7, class 8 are superior to class 3, class 4, class 5, class 6, class 9, class 10 because at all cut-offs the True Positive Rate is higher and the False Positive Rates is lower. The area for the superior classes is larger than the area of the other classes. *The closer the ROC curve is to the upper left corner, the more efficient is the model of the neural network.* It can be seen on the graph that the values of the ROC curves for class 9 and class 10 are depicted between the values 0.6 – 0.7 which means that the model has a poor performance of the classification. On the other hand, for class 1, class 2, class 3, class 4, class 5, class 7 and class 8, the values of the ROC curves are between 0.8 – 1.0, leading to good and very good performance.

All the cases are summed up in the last graph *All ROC*, where most of the curves are above 0.8 on the TPR axis, meaning that overall the model has a good behavior.

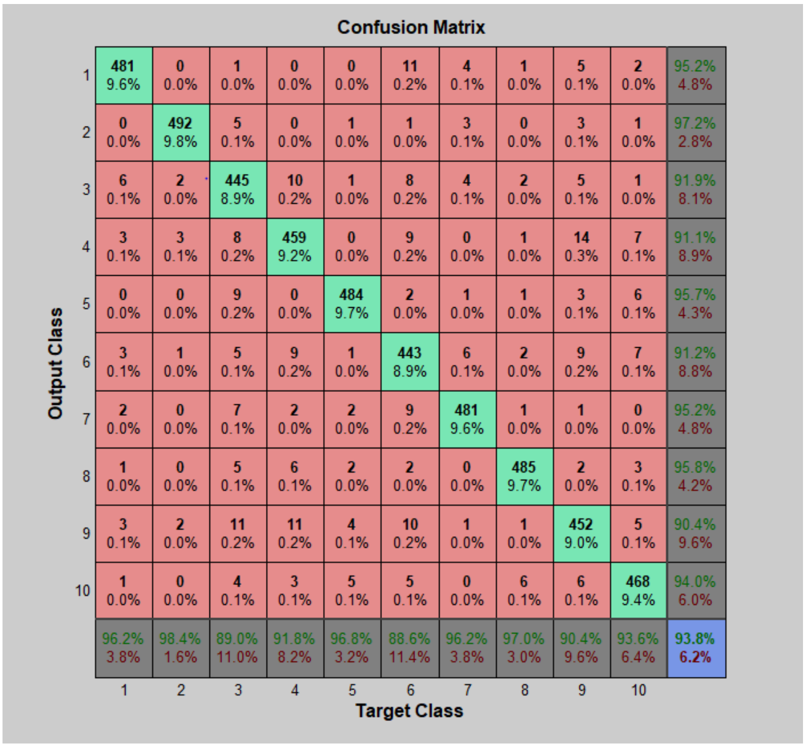
2.5.4 Simulation results: Confusion Matrix



Confusion Matrix

Output Class \ Target Class	1	2	3	4	5	6	7	8	9	10	
1	106 2.1%	3 0.1%	29 0.6%	95 1.9%	19 0.4%	204 4.1%	8 0.2%	8 0.2%	56 1.1%	14 0.3%	19.6% 80.4%
2	0 0.0%	472 9.4%	14 0.3%	4 0.1%	6 0.1%	2 0.0%	3 0.1%	2 0.0%	11 0.2%	1 0.0%	91.7% 8.3%
3	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	NaN% NaN%
4	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	NaN% NaN%
5	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	NaN% NaN%
6	350 7.0%	0 0.0%	1 0.0%	6 0.1%	0 0.0%	254 5.1%	2 0.0%	0 0.0%	5 0.1%	1 0.0%	41.0% 59.0%
7	7 0.1%	9 0.2%	45 0.9%	8 0.2%	32 0.6%	13 0.3%	479 9.6%	2 0.0%	16 0.3%	0 0.0%	78.4% 21.6%
8	0 0.0%	4 0.1%	20 0.4%	19 0.4%	2 0.0%	3 0.1%	0 0.0%	475 9.5%	9 0.2%	27 0.5%	85.0% 15.0%
9	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	NaN% NaN%
10	37 0.7%	12 0.2%	391 7.8%	368 7.4%	441 8.8%	24 0.5%	8 0.2%	13 0.3%	403 8.1%	457 9.1%	21.2% 78.8%
	21.2% 78.8%	94.4% 5.6%	0.0% 100%	0.0% 100%	0.0% 100%	50.8% 49.2%	95.8% 4.2%	95.0% 5.0%	0.0% 100%	91.4% 8.6%	44.9% 55.1%

Figure 2.8: Confusion Matrix for 500 images per class, one hidden layer with 10 neurons

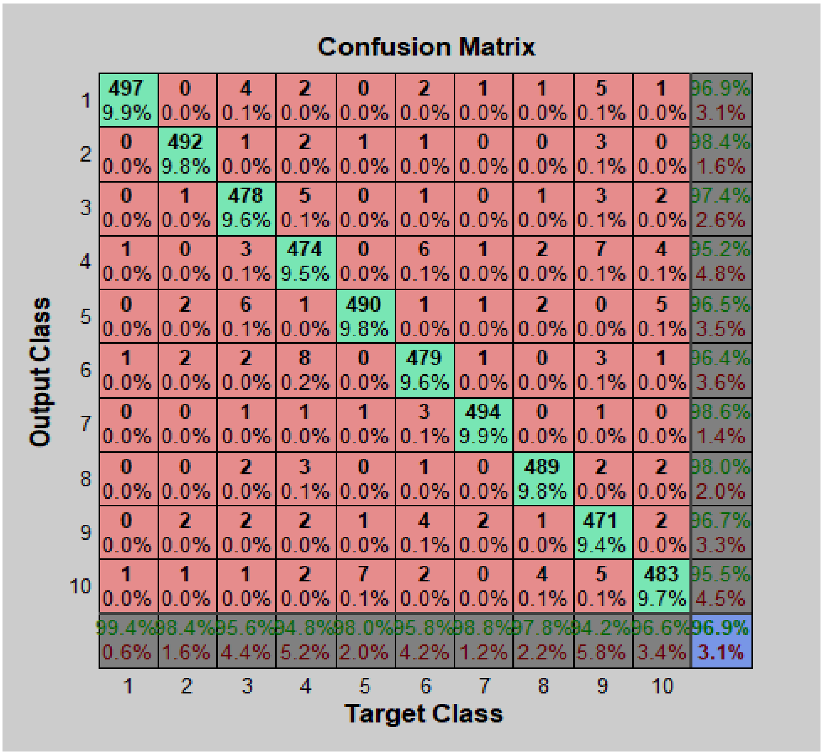


Confusion Matrix

Output Class	1	2	3	4	5	6	7	8	9	10	
1	481 9.6%	0 0.0%	1 0.0%	0 0.0%	0 0.0%	11 0.2%	4 0.1%	1 0.0%	5 0.1%	2 0.0%	95.2% 4.8%
2	0 0.0%	492 9.8%	5 0.1%	0 0.0%	1 0.0%	1 0.0%	3 0.1%	0 0.0%	3 0.1%	1 0.0%	97.2% 2.8%
3	6 0.1%	2 0.0%	445 8.9%	10 0.2%	1 0.0%	8 0.2%	4 0.1%	2 0.0%	5 0.1%	1 0.0%	91.9% 8.1%
4	3 0.1%	3 0.1%	8 0.2%	459 9.2%	0 0.0%	9 0.2%	0 0.0%	1 0.0%	14 0.3%	7 0.1%	91.1% 8.9%
5	0 0.0%	0 0.0%	9 0.2%	0 0.0%	484 9.7%	2 0.0%	1 0.0%	1 0.0%	3 0.1%	6 0.1%	95.7% 4.3%
6	3 0.1%	1 0.0%	5 0.1%	9 0.2%	1 0.0%	443 8.9%	6 0.1%	2 0.0%	9 0.2%	7 0.1%	91.2% 8.8%
7	2 0.0%	0 0.0%	7 0.1%	2 0.0%	2 0.0%	9 0.2%	481 9.6%	1 0.0%	1 0.0%	0 0.0%	95.2% 4.8%
8	1 0.0%	0 0.0%	5 0.1%	6 0.1%	2 0.0%	2 0.0%	0 0.0%	485 9.7%	2 0.0%	3 0.1%	95.8% 4.2%
9	3 0.1%	2 0.0%	11 0.2%	11 0.2%	4 0.1%	10 0.2%	1 0.0%	1 0.0%	452 9.0%	5 0.1%	90.4% 9.6%
10	1 0.0%	0 0.0%	4 0.1%	3 0.1%	5 0.1%	5 0.1%	0 0.0%	6 0.1%	6 0.1%	468 9.4%	94.0% 6.0%
	96.2% 3.8%	98.4% 1.6%	89.0% 11.0%	91.8% 8.2%	96.8% 3.2%	88.6% 11.4%	96.2% 3.8%	97.0% 3.0%	90.4% 9.6%	93.6% 6.4%	93.8% 6.2%
	1	2	3	4	5	6	7	8	9	10	

Target Class

Figure 2.9: Confusion Matrix for 500 images per class, one hidden layer with 50 neurons

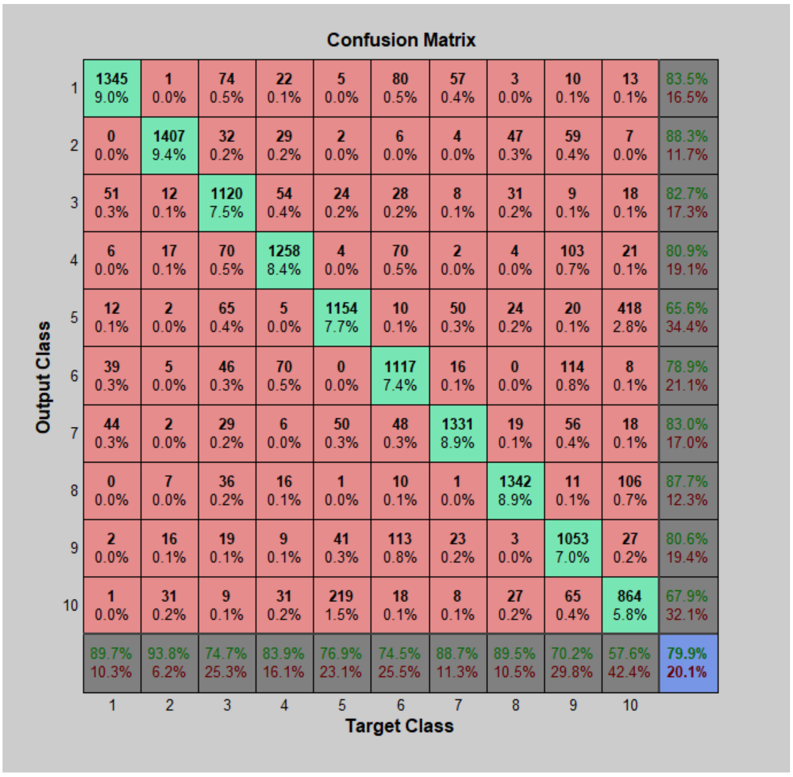


Confusion Matrix

Output Class	1	2	3	4	5	6	7	8	9	10	
1	497 9.9%	0 0.0%	4 0.1%	2 0.0%	0 0.0%	2 0.0%	1 0.0%	1 0.0%	5 0.1%	1 0.0%	96.9% 3.1%
2	0 0.0%	492 9.8%	1 0.0%	2 0.0%	1 0.0%	1 0.0%	0 0.0%	0 0.0%	3 0.1%	0 0.0%	98.4% 1.6%
3	0 0.0%	1 0.0%	478 9.6%	5 0.1%	0 0.0%	1 0.0%	0 0.0%	1 0.0%	3 0.1%	2 0.0%	97.4% 2.6%
4	1 0.0%	0 0.0%	3 0.1%	474 9.5%	0 0.0%	6 0.1%	1 0.0%	2 0.0%	7 0.1%	4 0.1%	95.2% 4.8%
5	0 0.0%	2 0.0%	6 0.1%	1 0.0%	490 9.8%	1 0.0%	1 0.0%	2 0.0%	0 0.0%	5 0.1%	96.5% 3.5%
6	1 0.0%	2 0.0%	2 0.0%	8 0.2%	0 0.0%	479 9.6%	1 0.0%	0 0.0%	3 0.1%	1 0.0%	96.4% 3.6%
7	0 0.0%	0 0.0%	1 0.0%	1 0.0%	1 0.0%	3 0.1%	494 9.9%	0 0.0%	1 0.0%	0 0.0%	98.6% 1.4%
8	0 0.0%	0 0.0%	2 0.0%	3 0.1%	0 0.0%	1 0.0%	0 0.0%	489 9.8%	2 0.0%	2 0.0%	98.0% 2.0%
9	0 0.0%	2 0.0%	2 0.0%	2 0.0%	1 0.0%	4 0.1%	2 0.0%	1 0.0%	471 9.4%	2 0.0%	96.7% 3.3%
10	1 0.0%	1 0.0%	1 0.0%	2 0.0%	7 0.1%	2 0.0%	0 0.0%	4 0.1%	5 0.1%	483 9.7%	95.5% 4.5%
	99.4% 0.6%	98.4% 1.6%	95.6% 4.4%	94.8% 5.2%	98.0% 2.0%	95.8% 4.2%	98.8% 1.2%	97.8% 2.2%	94.2% 5.8%	96.6% 3.4%	96.9% 3.1%
	1	2	3	4	5	6	7	8	9	10	

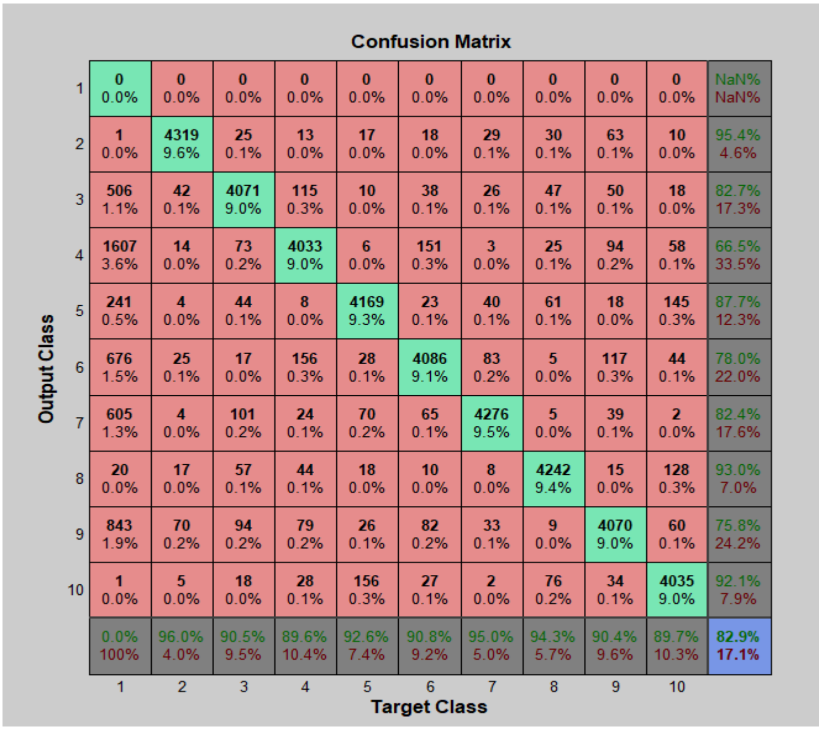
Target Class

Figure 2.10: Confusion Matrix for 500 images per class, one hidden layer with 80 neurons


Confusion Matrix
Output Class
Target Class

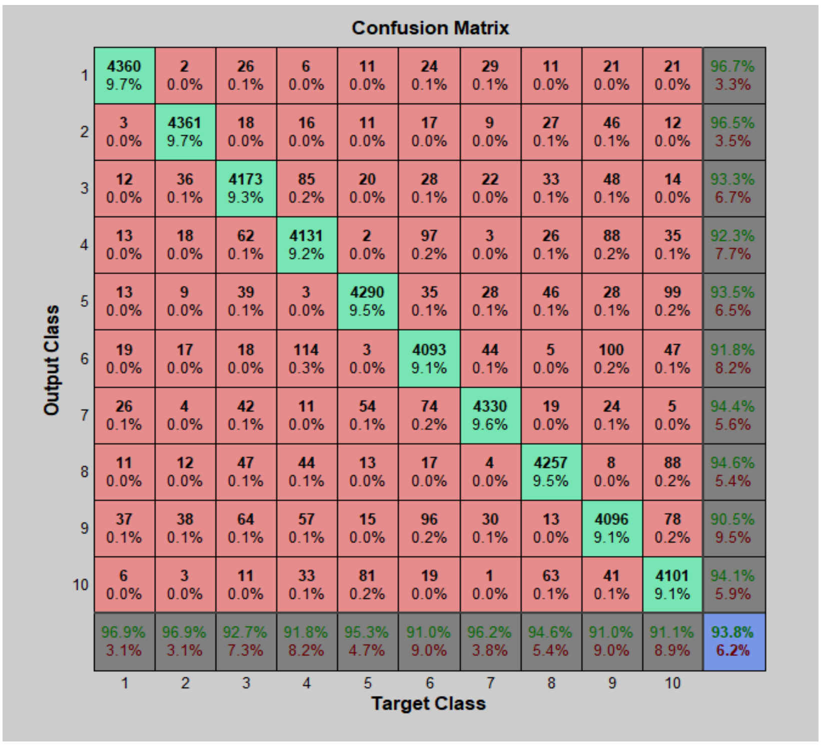
	1	2	3	4	5	6	7	8	9	10	
1	1345 9.0%	1 0.0%	74 0.5%	22 0.1%	5 0.0%	80 0.5%	57 0.4%	3 0.0%	10 0.1%	13 0.1%	83.5% 16.5%
2	0 0.0%	1407 9.4%	32 0.2%	29 0.2%	2 0.0%	6 0.0%	4 0.0%	47 0.3%	59 0.4%	7 0.0%	88.3% 11.7%
3	51 0.3%	12 0.1%	1120 7.5%	54 0.4%	24 0.2%	28 0.2%	8 0.1%	31 0.2%	9 0.1%	18 0.1%	82.7% 17.3%
4	6 0.0%	17 0.1%	70 0.5%	1258 8.4%	4 0.0%	70 0.5%	2 0.0%	4 0.0%	103 0.7%	21 0.1%	80.9% 19.1%
5	12 0.1%	2 0.0%	65 0.4%	5 0.0%	1154 7.7%	10 0.1%	50 0.3%	24 0.2%	20 0.1%	418 2.8%	65.6% 34.4%
6	39 0.3%	5 0.0%	46 0.3%	70 0.5%	0 0.0%	1117 7.4%	16 0.1%	0 0.0%	114 0.8%	8 0.1%	78.9% 21.1%
7	44 0.3%	2 0.0%	29 0.2%	6 0.0%	50 0.3%	48 0.3%	1331 8.9%	19 0.1%	56 0.4%	18 0.1%	83.0% 17.0%
8	0 0.0%	7 0.0%	36 0.2%	16 0.1%	1 0.0%	10 0.1%	1 0.0%	1342 8.9%	11 0.1%	106 0.7%	87.7% 12.3%
9	2 0.0%	16 0.1%	19 0.1%	9 0.1%	41 0.3%	113 0.8%	23 0.2%	3 0.0%	1053 7.0%	27 0.2%	80.6% 19.4%
10	1 0.0%	31 0.2%	9 0.1%	31 0.2%	219 1.5%	18 0.1%	8 0.1%	27 0.2%	65 0.4%	864 5.8%	67.9% 32.1%
	89.7% 10.3%	93.8% 6.2%	74.7% 25.3%	83.9% 16.1%	76.9% 23.1%	74.5% 25.5%	88.7% 11.3%	89.5% 10.5%	70.2% 29.8%	57.6% 42.4%	79.9% 20.1%

Figure 2.11: Confusion Matrix for 1500 images per class, one hidden layer with 10 neurons


Confusion Matrix
Output Class
Target Class

	1	2	3	4	5	6	7	8	9	10	
1	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	NaN% NaN%
2	1 0.0%	4319 9.6%	25 0.1%	13 0.0%	17 0.0%	18 0.0%	29 0.1%	30 0.1%	63 0.1%	10 0.0%	95.4% 4.6%
3	506 1.1%	42 0.1%	4071 9.0%	115 0.3%	10 0.0%	38 0.1%	26 0.1%	47 0.1%	50 0.1%	18 0.0%	82.7% 17.3%
4	1607 3.6%	14 0.0%	73 0.2%	4033 9.0%	6 0.0%	151 0.3%	3 0.0%	25 0.1%	94 0.2%	58 0.1%	66.5% 33.5%
5	241 0.5%	4 0.0%	44 0.1%	8 0.0%	4169 9.3%	23 0.1%	40 0.1%	61 0.1%	18 0.0%	145 0.3%	87.7% 12.3%
6	676 1.5%	25 0.1%	17 0.0%	156 0.3%	28 0.1%	4086 9.1%	83 0.2%	5 0.0%	117 0.3%	44 0.1%	78.0% 22.0%
7	605 1.3%	4 0.0%	101 0.2%	24 0.1%	70 0.2%	65 0.1%	4276 9.5%	5 0.0%	39 0.1%	2 0.0%	82.4% 17.6%
8	20 0.0%	17 0.0%	57 0.1%	44 0.1%	18 0.0%	10 0.0%	8 0.0%	4242 9.4%	15 0.0%	128 0.3%	93.0% 7.0%
9	843 1.9%	70 0.2%	94 0.2%	79 0.2%	26 0.1%	82 0.2%	33 0.1%	9 0.0%	4070 9.0%	60 0.1%	75.8% 24.2%
10	1 0.0%	5 0.0%	18 0.0%	28 0.1%	156 0.3%	27 0.1%	2 0.0%	76 0.2%	34 0.1%	4035 9.0%	92.1% 7.9%
	0.0% 100%	96.0% 4.0%	90.5% 9.5%	89.6% 10.4%	92.6% 7.4%	90.8% 9.2%	95.0% 5.0%	94.3% 5.7%	90.4% 9.6%	89.7% 10.3%	82.9% 17.1%

Figure 2.12: Confusion Matrix for 4500 images per class, one hidden layer with 10 neurons

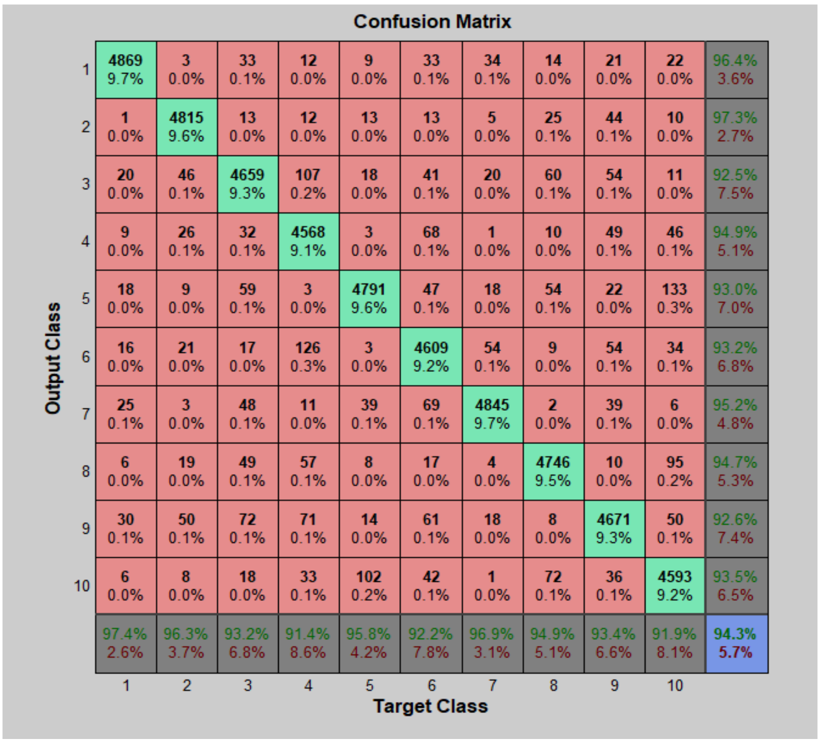


Confusion Matrix

Output Class	1	2	3	4	5	6	7	8	9	10	
1	4360 9.7%	2 0.0%	26 0.1%	6 0.0%	11 0.0%	24 0.1%	29 0.1%	11 0.0%	21 0.0%	21 0.0%	96.7% 3.3%
2	3 0.0%	4361 9.7%	18 0.0%	16 0.0%	11 0.0%	17 0.0%	9 0.0%	27 0.1%	46 0.1%	12 0.0%	96.5% 3.5%
3	12 0.0%	36 0.1%	4173 9.3%	85 0.2%	20 0.0%	28 0.1%	22 0.0%	33 0.1%	48 0.1%	14 0.0%	93.3% 6.7%
4	13 0.0%	18 0.0%	62 0.1%	4131 9.2%	2 0.0%	97 0.2%	3 0.0%	26 0.1%	88 0.2%	35 0.1%	92.3% 7.7%
5	13 0.0%	9 0.0%	39 0.1%	3 0.0%	4290 9.5%	35 0.1%	28 0.1%	46 0.1%	28 0.1%	99 0.2%	93.5% 6.5%
6	19 0.0%	17 0.0%	18 0.0%	114 0.3%	3 0.0%	4093 9.1%	44 0.1%	5 0.0%	100 0.2%	47 0.1%	91.8% 8.2%
7	26 0.1%	4 0.0%	42 0.1%	11 0.0%	54 0.1%	74 0.2%	4330 9.6%	19 0.0%	24 0.1%	5 0.0%	94.4% 5.6%
8	11 0.0%	12 0.0%	47 0.1%	44 0.1%	13 0.0%	17 0.0%	4 0.0%	4257 9.5%	8 0.0%	88 0.2%	94.6% 5.4%
9	37 0.1%	38 0.1%	64 0.1%	57 0.1%	15 0.0%	96 0.2%	30 0.1%	13 0.0%	4096 9.1%	78 0.2%	90.5% 9.5%
10	6 0.0%	3 0.0%	11 0.0%	33 0.1%	81 0.2%	19 0.0%	1 0.0%	63 0.1%	41 0.1%	4101 9.1%	94.1% 5.9%
	96.9% 3.1%	96.9% 3.1%	92.7% 7.3%	91.8% 8.2%	95.3% 4.7%	91.0% 9.0%	96.2% 3.8%	94.6% 5.4%	91.0% 9.0%	91.1% 8.9%	93.8% 6.2%

Target Class

Figure 2.13: Confusion Matrix for 4500 images per class, one hidden layer with 30 neurons

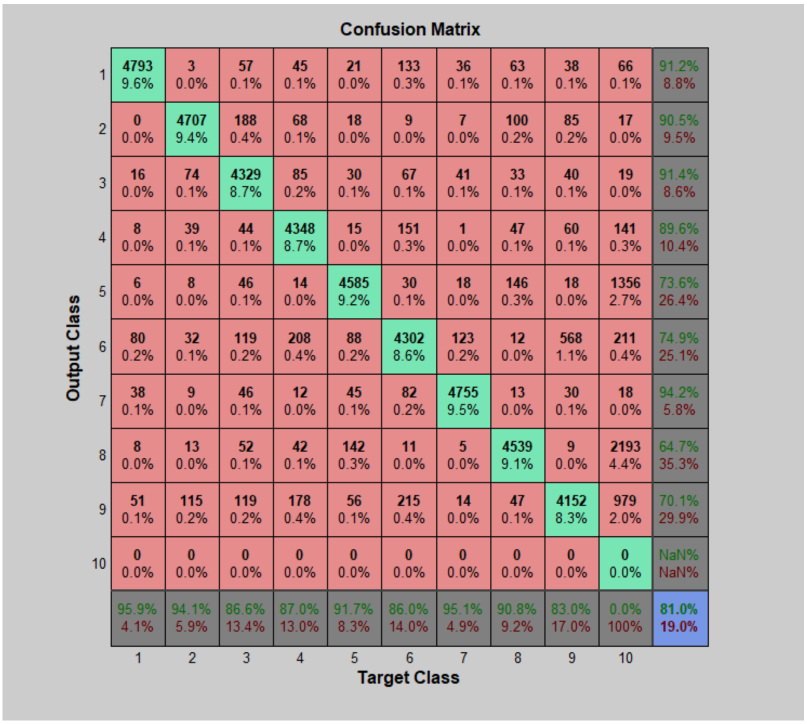


Confusion Matrix

Output Class	1	2	3	4	5	6	7	8	9	10	
1	4869 9.7%	3 0.0%	33 0.1%	12 0.0%	9 0.0%	33 0.1%	34 0.1%	14 0.0%	21 0.0%	22 0.0%	96.4% 3.6%
2	1 0.0%	4815 9.6%	13 0.0%	12 0.0%	13 0.0%	13 0.0%	5 0.0%	25 0.1%	44 0.1%	10 0.0%	97.3% 2.7%
3	20 0.0%	46 0.1%	4659 9.3%	107 0.2%	18 0.0%	41 0.1%	20 0.0%	60 0.1%	54 0.1%	11 0.0%	92.5% 7.5%
4	9 0.0%	26 0.1%	32 0.1%	4568 9.1%	3 0.0%	68 0.1%	1 0.0%	10 0.0%	49 0.1%	46 0.1%	94.9% 5.1%
5	18 0.0%	9 0.0%	59 0.1%	3 0.0%	4791 9.6%	47 0.1%	18 0.0%	54 0.1%	22 0.0%	133 0.3%	93.0% 7.0%
6	16 0.0%	21 0.0%	17 0.0%	126 0.3%	3 0.0%	4609 9.2%	54 0.1%	9 0.0%	54 0.1%	34 0.1%	93.2% 6.8%
7	25 0.1%	3 0.0%	48 0.1%	11 0.0%	39 0.1%	69 0.1%	4845 9.7%	2 0.0%	39 0.1%	6 0.0%	95.2% 4.8%
8	6 0.0%	19 0.0%	49 0.1%	57 0.1%	8 0.0%	17 0.0%	4 0.0%	4746 9.5%	10 0.0%	95 0.2%	94.7% 5.3%
9	30 0.1%	50 0.1%	72 0.1%	71 0.1%	14 0.0%	61 0.1%	18 0.0%	8 0.0%	4671 9.3%	50 0.1%	92.6% 7.4%
10	6 0.0%	8 0.0%	18 0.0%	33 0.1%	102 0.2%	42 0.1%	1 0.0%	72 0.1%	36 0.1%	4593 9.2%	93.5% 6.5%
	97.4% 2.6%	96.3% 3.7%	93.2% 6.8%	91.4% 8.6%	95.8% 4.2%	92.2% 7.8%	96.9% 3.1%	94.9% 5.1%	93.4% 6.6%	91.9% 8.1%	94.3% 5.7%

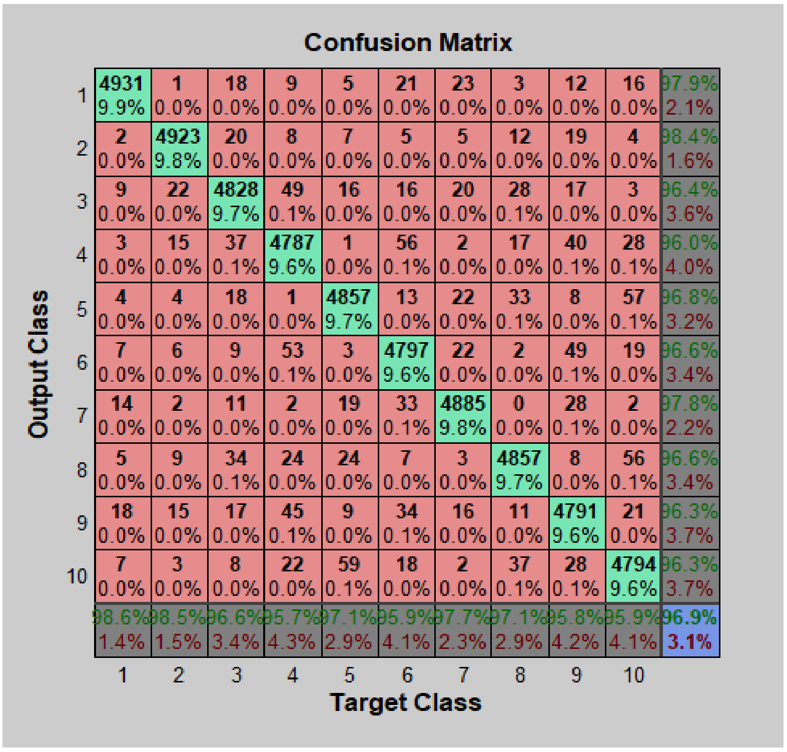
Target Class

Figure 2.14: Confusion Matrix for 5000 images per class, one hidden layer with 30 neurons


Confusion Matrix

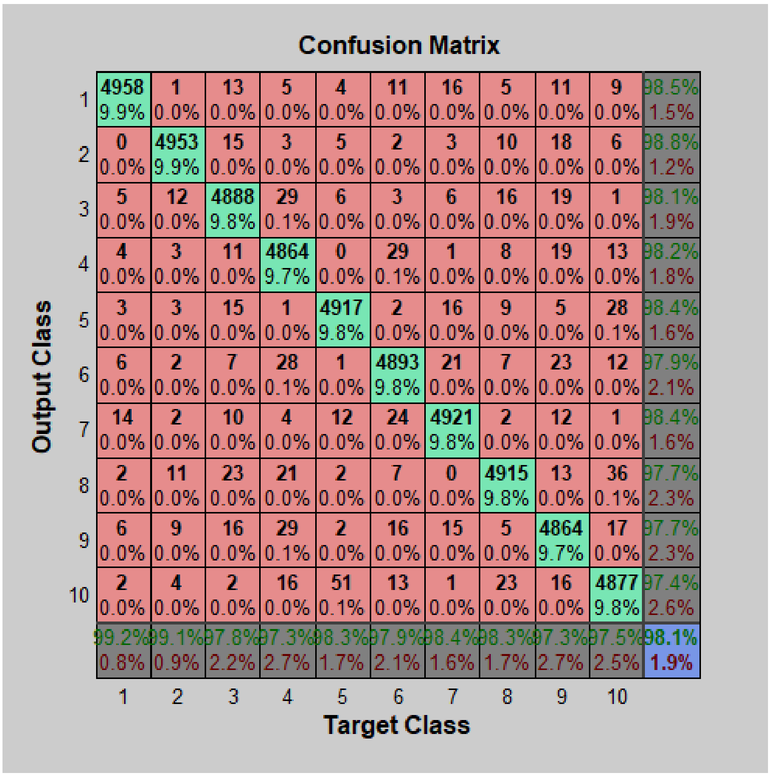
Output Class \ Target Class	1	2	3	4	5	6	7	8	9	10	
1	4793 9.6%	3 0.0%	57 0.1%	45 0.1%	21 0.0%	133 0.3%	36 0.1%	63 0.1%	38 0.1%	66 0.1%	91.2% 8.8%
2	0 0.0%	4707 9.4%	188 0.4%	68 0.1%	18 0.0%	9 0.0%	7 0.0%	100 0.2%	85 0.2%	17 0.0%	90.5% 9.5%
3	16 0.0%	74 0.1%	4329 8.7%	85 0.2%	30 0.1%	67 0.1%	41 0.1%	33 0.1%	40 0.1%	19 0.0%	91.4% 8.6%
4	8 0.0%	39 0.1%	44 0.1%	4348 8.7%	15 0.0%	151 0.3%	1 0.0%	47 0.1%	60 0.1%	141 0.3%	89.6% 10.4%
5	6 0.0%	8 0.0%	46 0.1%	14 0.0%	4585 9.2%	30 0.1%	18 0.0%	146 0.3%	18 0.0%	1356 2.7%	73.6% 26.4%
6	80 0.2%	32 0.1%	119 0.2%	208 0.4%	88 0.2%	4302 8.6%	123 0.2%	12 0.0%	568 1.1%	211 0.4%	74.9% 25.1%
7	38 0.1%	9 0.0%	46 0.1%	12 0.0%	45 0.1%	82 0.2%	4755 9.5%	13 0.0%	30 0.1%	18 0.0%	94.2% 5.8%
8	8 0.0%	13 0.0%	52 0.1%	42 0.1%	142 0.3%	11 0.0%	5 0.0%	4539 9.1%	9 0.0%	2193 4.4%	64.7% 35.3%
9	51 0.1%	115 0.2%	119 0.2%	178 0.4%	56 0.1%	215 0.4%	14 0.0%	47 0.1%	4152 8.3%	979 2.0%	70.1% 29.9%
10	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	NaN% NaN%
	95.9% 4.1%	94.1% 5.9%	86.6% 13.4%	87.0% 13.0%	91.7% 8.3%	86.0% 14.0%	95.1% 4.9%	90.8% 9.2%	83.0% 17.0%	0.0% 100%	81.0% 19.0%

Figure 2.15: Confusion Matrix for 5000 images per class, one hidden layer with 10 neurons


Confusion Matrix

Output Class \ Target Class	1	2	3	4	5	6	7	8	9	10	
1	4931 9.9%	1 0.0%	18 0.0%	9 0.0%	5 0.0%	21 0.0%	23 0.0%	3 0.0%	12 0.0%	16 0.0%	97.9% 2.1%
2	2 0.0%	4923 9.8%	20 0.0%	8 0.0%	7 0.0%	5 0.0%	5 0.0%	12 0.0%	19 0.0%	4 0.0%	98.4% 1.6%
3	9 0.0%	22 0.0%	4828 9.7%	49 0.1%	16 0.0%	16 0.0%	20 0.0%	28 0.1%	17 0.0%	3 0.0%	96.4% 3.6%
4	3 0.0%	15 0.0%	37 0.1%	4787 9.6%	1 0.0%	56 0.1%	2 0.0%	17 0.0%	40 0.1%	28 0.1%	96.0% 4.0%
5	4 0.0%	4 0.0%	18 0.0%	1 0.0%	4857 9.7%	13 0.0%	22 0.0%	33 0.1%	8 0.0%	57 0.1%	96.8% 3.2%
6	7 0.0%	6 0.0%	9 0.0%	53 0.1%	3 0.0%	4797 9.6%	22 0.0%	2 0.0%	49 0.1%	19 0.0%	96.6% 3.4%
7	14 0.0%	2 0.0%	11 0.0%	2 0.0%	19 0.0%	33 0.1%	4885 9.8%	0 0.0%	28 0.1%	2 0.0%	97.8% 2.2%
8	5 0.0%	9 0.0%	34 0.1%	24 0.0%	24 0.0%	7 0.0%	3 0.0%	4857 9.7%	8 0.0%	56 0.1%	96.6% 3.4%
9	18 0.0%	15 0.0%	17 0.0%	45 0.1%	9 0.0%	34 0.1%	16 0.0%	11 0.0%	4791 9.6%	21 0.0%	96.3% 3.7%
10	7 0.0%	3 0.0%	8 0.0%	22 0.0%	59 0.1%	18 0.0%	2 0.0%	37 0.1%	28 0.1%	4794 9.6%	96.3% 3.7%
	98.6% 1.4%	98.5% 1.5%	96.6% 3.4%	95.7% 4.3%	97.1% 2.9%	95.9% 4.1%	97.7% 2.3%	97.1% 2.9%	95.8% 4.2%	95.9% 4.1%	96.9% 3.1%

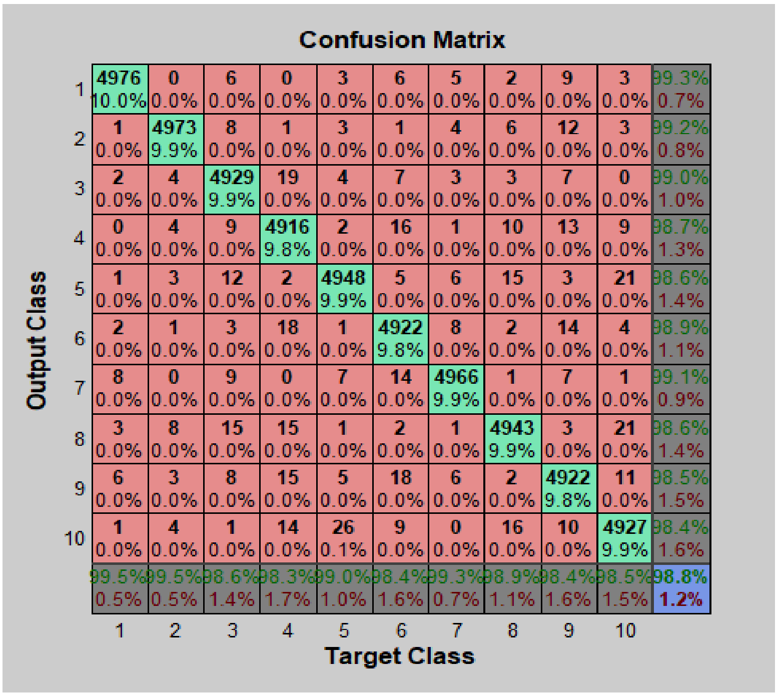
Figure 2.16: Confusion Matrix for 5000 images per class, one hidden layer with 50 neurons


Output Class
Target Class

Confusion Matrix

	1	2	3	4	5	6	7	8	9	10	
1	4958 9.9%	1 0.0%	13 0.0%	5 0.0%	4 0.0%	11 0.0%	16 0.0%	5 0.0%	11 0.0%	9 0.0%	98.5% 1.5%
2	0 0.0%	4953 9.9%	15 0.0%	3 0.0%	5 0.0%	2 0.0%	3 0.0%	10 0.0%	18 0.0%	6 0.0%	98.8% 1.2%
3	5 0.0%	12 0.0%	4888 9.8%	29 0.1%	6 0.0%	3 0.0%	6 0.0%	16 0.0%	19 0.0%	1 0.0%	98.1% 1.9%
4	4 0.0%	3 0.0%	11 0.0%	4864 9.7%	0 0.0%	29 0.1%	1 0.0%	8 0.0%	19 0.0%	13 0.0%	98.2% 1.8%
5	3 0.0%	3 0.0%	15 0.0%	1 0.0%	4917 9.8%	2 0.0%	16 0.0%	9 0.0%	5 0.0%	28 0.1%	98.4% 1.6%
6	6 0.0%	2 0.0%	7 0.0%	28 0.1%	1 0.0%	4893 9.8%	21 0.0%	7 0.0%	23 0.0%	12 0.0%	97.9% 2.1%
7	14 0.0%	2 0.0%	10 0.0%	4 0.0%	12 0.0%	24 0.0%	4921 9.8%	2 0.0%	12 0.0%	1 0.0%	98.4% 1.6%
8	2 0.0%	11 0.0%	23 0.0%	21 0.0%	2 0.0%	7 0.0%	0 0.0%	4915 9.8%	13 0.0%	36 0.1%	97.7% 2.3%
9	6 0.0%	9 0.0%	16 0.0%	29 0.1%	2 0.0%	16 0.0%	15 0.0%	5 0.0%	4864 9.7%	17 0.0%	97.7% 2.3%
10	2 0.0%	4 0.0%	2 0.0%	16 0.0%	51 0.1%	13 0.0%	1 0.0%	23 0.0%	16 0.0%	4877 9.8%	97.4% 2.6%
	99.2% 0.8%	99.1% 0.9%	97.8% 2.2%	97.3% 2.7%	98.3% 1.7%	97.9% 2.1%	98.4% 1.6%	98.3% 1.7%	97.3% 2.7%	97.5% 2.5%	98.1% 1.9%

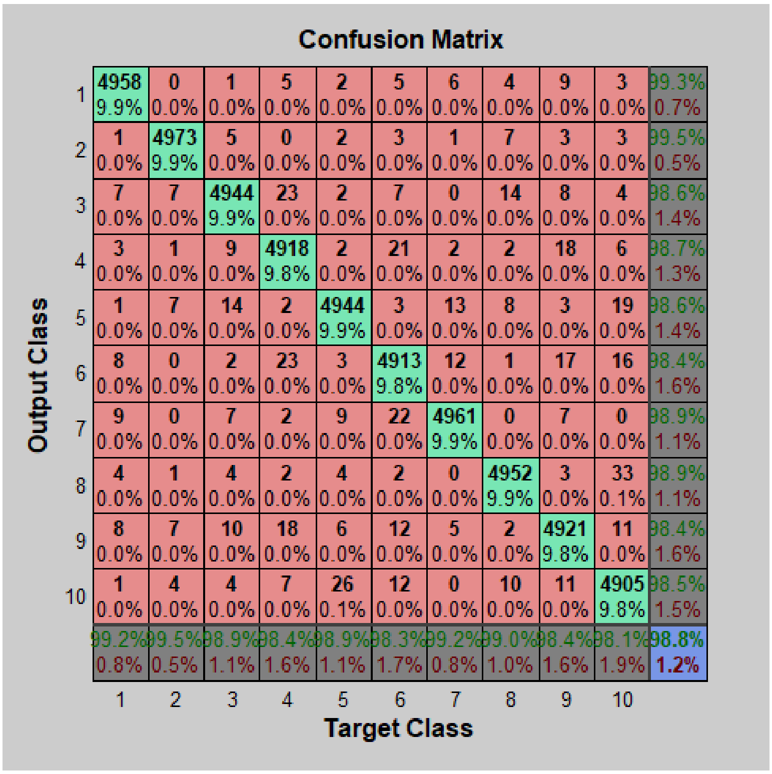
Figure 2.17: Confusion Matrix for 5000 images per class, one hidden layer with 80 neurons


Output Class
Target Class

Confusion Matrix

	1	2	3	4	5	6	7	8	9	10	
1	4976 10.0%	0 0.0%	6 0.0%	0 0.0%	3 0.0%	6 0.0%	5 0.0%	2 0.0%	9 0.0%	3 0.0%	99.3% 0.7%
2	1 0.0%	4973 9.9%	8 0.0%	1 0.0%	3 0.0%	1 0.0%	4 0.0%	6 0.0%	12 0.0%	3 0.0%	99.2% 0.8%
3	2 0.0%	4 0.0%	4929 9.9%	19 0.0%	4 0.0%	7 0.0%	3 0.0%	3 0.0%	7 0.0%	0 0.0%	99.0% 1.0%
4	0 0.0%	4 0.0%	9 0.0%	4916 9.8%	2 0.0%	16 0.0%	1 0.0%	10 0.0%	13 0.0%	9 0.0%	98.7% 1.3%
5	1 0.0%	3 0.0%	12 0.0%	2 0.0%	4948 9.9%	5 0.0%	6 0.0%	15 0.0%	3 0.0%	21 0.0%	98.6% 1.4%
6	2 0.0%	1 0.0%	3 0.0%	18 0.0%	1 0.0%	4922 9.8%	8 0.0%	2 0.0%	14 0.0%	4 0.0%	98.9% 1.1%
7	8 0.0%	0 0.0%	9 0.0%	0 0.0%	7 0.0%	14 0.0%	4966 9.9%	1 0.0%	7 0.0%	1 0.0%	99.1% 0.9%
8	3 0.0%	8 0.0%	15 0.0%	15 0.0%	1 0.0%	2 0.0%	1 0.0%	4943 9.9%	3 0.0%	21 0.0%	98.6% 1.4%
9	6 0.0%	3 0.0%	8 0.0%	15 0.0%	5 0.0%	18 0.0%	6 0.0%	2 0.0%	4922 9.8%	11 0.0%	98.5% 1.5%
10	1 0.0%	4 0.0%	1 0.0%	14 0.0%	26 0.1%	9 0.0%	0 0.0%	16 0.0%	10 0.0%	4927 9.9%	98.4% 1.6%
	99.5% 0.5%	99.5% 0.5%	98.6% 1.4%	98.3% 1.7%	99.0% 1.0%	98.4% 1.6%	99.3% 0.7%	98.9% 1.1%	98.4% 1.6%	98.5% 1.5%	98.8% 1.2%

Figure 2.18: Confusion Matrix for 5000 images per class, one hidden layer with 200 neurons



Confusion Matrix

Output Class \ Target Class	1	2	3	4	5	6	7	8	9	10	
1	4958 9.9%	0 0.0%	1 0.0%	5 0.0%	2 0.0%	5 0.0%	6 0.0%	4 0.0%	9 0.0%	3 0.0%	99.3% 0.7%
2	1 0.0%	4973 9.9%	5 0.0%	0 0.0%	2 0.0%	3 0.0%	1 0.0%	7 0.0%	3 0.0%	3 0.0%	99.5% 0.5%
3	7 0.0%	7 0.0%	4944 9.9%	23 0.0%	2 0.0%	7 0.0%	0 0.0%	14 0.0%	8 0.0%	4 0.0%	98.6% 1.4%
4	3 0.0%	1 0.0%	9 0.0%	4918 9.8%	2 0.0%	21 0.0%	2 0.0%	2 0.0%	18 0.0%	6 0.0%	98.7% 1.3%
5	1 0.0%	7 0.0%	14 0.0%	2 0.0%	4944 9.9%	3 0.0%	13 0.0%	8 0.0%	3 0.0%	19 0.0%	98.6% 1.4%
6	8 0.0%	0 0.0%	2 0.0%	23 0.0%	3 0.0%	4913 9.8%	12 0.0%	1 0.0%	17 0.0%	16 0.0%	98.4% 1.6%
7	9 0.0%	0 0.0%	7 0.0%	2 0.0%	9 0.0%	22 0.0%	4961 9.9%	0 0.0%	7 0.0%	0 0.0%	98.9% 1.1%
8	4 0.0%	1 0.0%	4 0.0%	2 0.0%	4 0.0%	2 0.0%	0 0.0%	4952 9.9%	3 0.0%	33 0.1%	98.9% 1.1%
9	8 0.0%	7 0.0%	10 0.0%	18 0.0%	6 0.0%	12 0.0%	5 0.0%	2 0.0%	4921 9.8%	11 0.0%	98.4% 1.6%
10	1 0.0%	4 0.0%	4 0.0%	7 0.0%	26 0.1%	12 0.0%	0 0.0%	10 0.0%	11 0.0%	4905 9.8%	98.5% 1.5%
	99.2% 0.8%	99.5% 0.5%	98.9% 1.1%	98.4% 1.6%	98.9% 1.1%	98.3% 1.7%	99.2% 0.8%	99.0% 1.0%	98.4% 1.6%	98.1% 1.9%	98.8% 1.2%

Figure 2.19: Confusion Matrix for 5000 images per class, one hidden layer with 2000 neurons

2.5.5 Simulation Results: Mean Square Error (MSE)

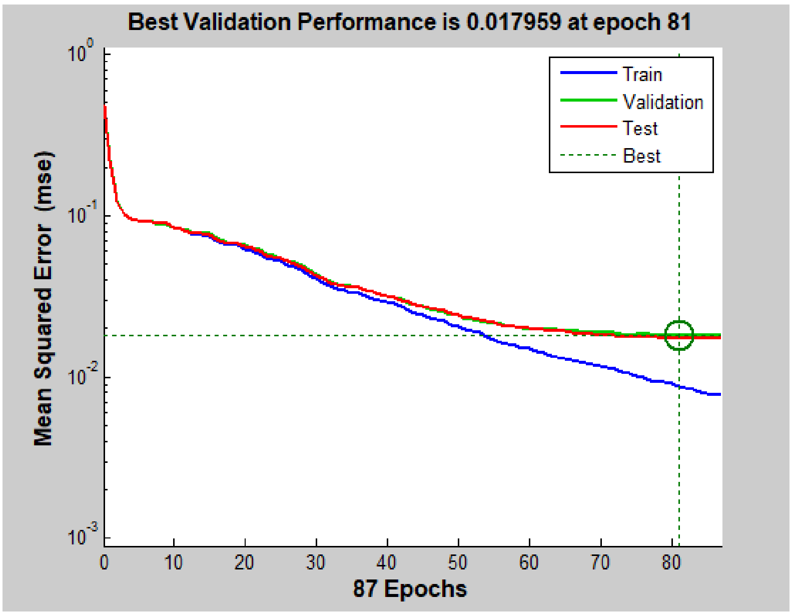


Figure 2.20: Performance of the training, validation, and testing – 500 images per class, one hidden layer network with 30 neurons

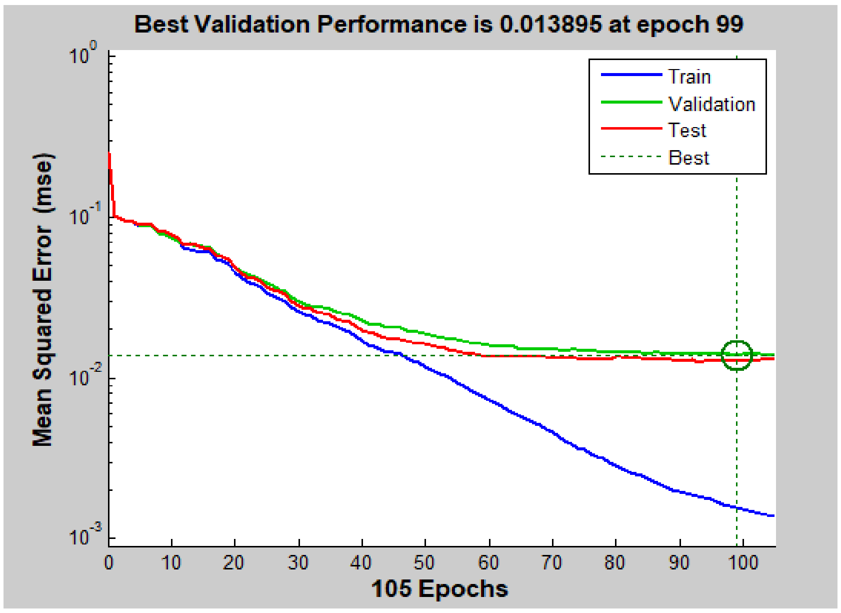


Figure 2.21: Performance of the training, validation, and testing – 500 images per class, one hidden layer network with 80 neurons

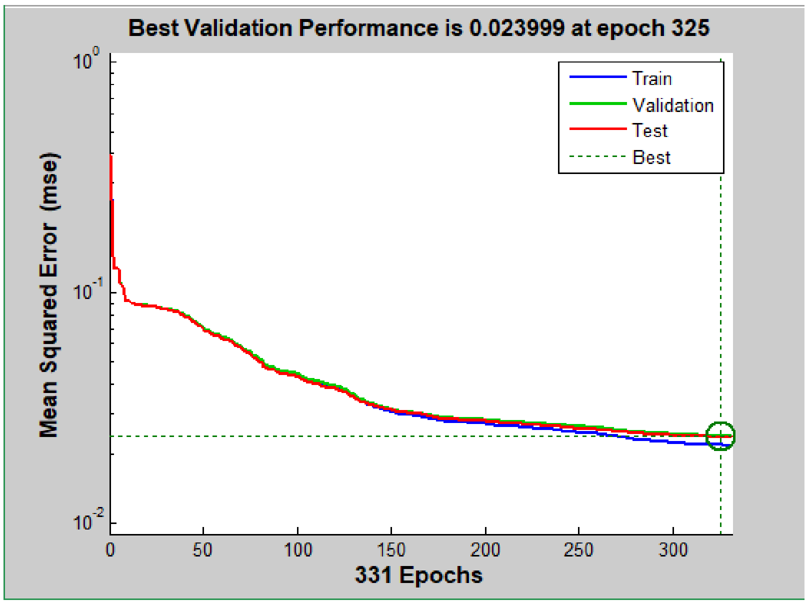


Figure 2.22: Performance of the training, validation, and testing – 4500 images per class, one hidden layer network with 10 neurons

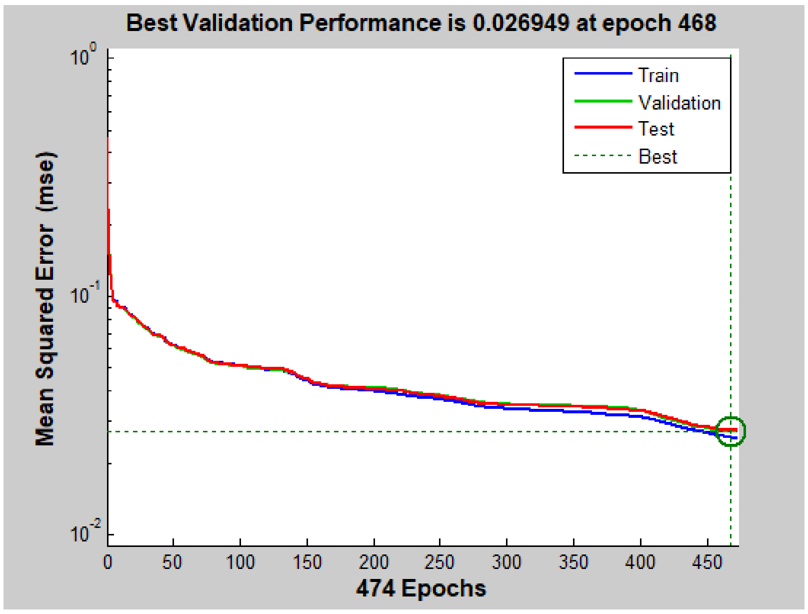


Figure 2.23: Performance of the training, validation, and testing – 5000 images, one hidden layer network with 10 neurons

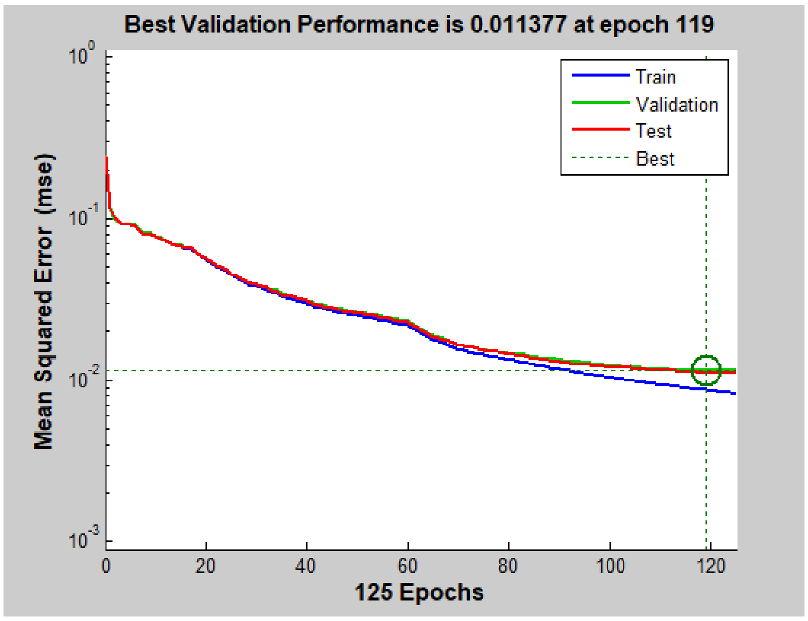


Figure 2.24: Performance of the training, validation, and testing – 5000 images, one hidden layer network with 30 neurons

2.5.6 Simulations Result: Receiver Operating Curve (ROC)

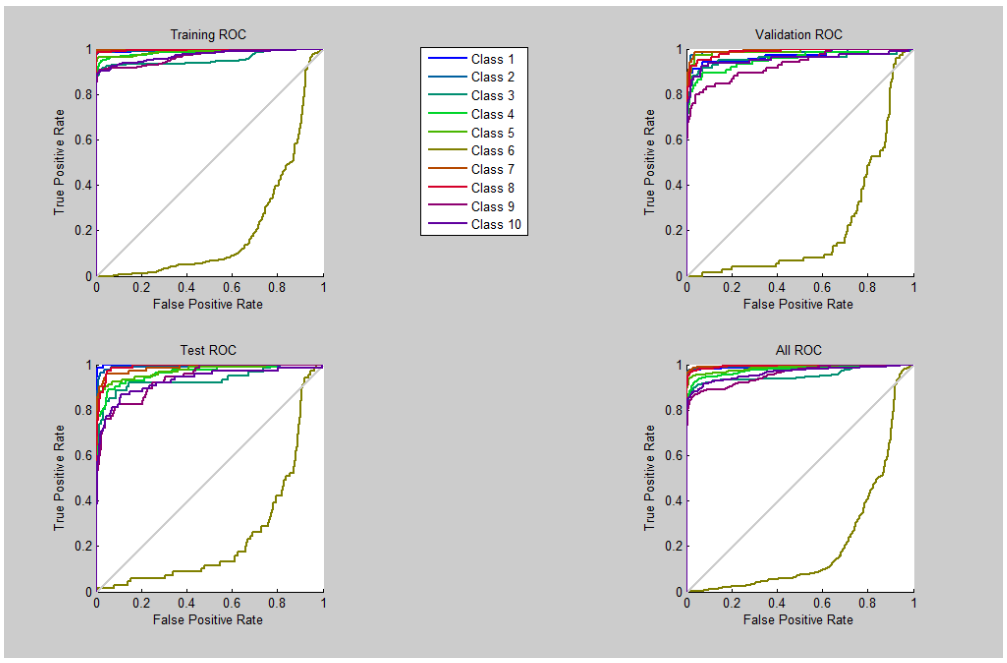


Figure 2.25: ROC curve for500 images, 1 hidden layer network with 10 neurons

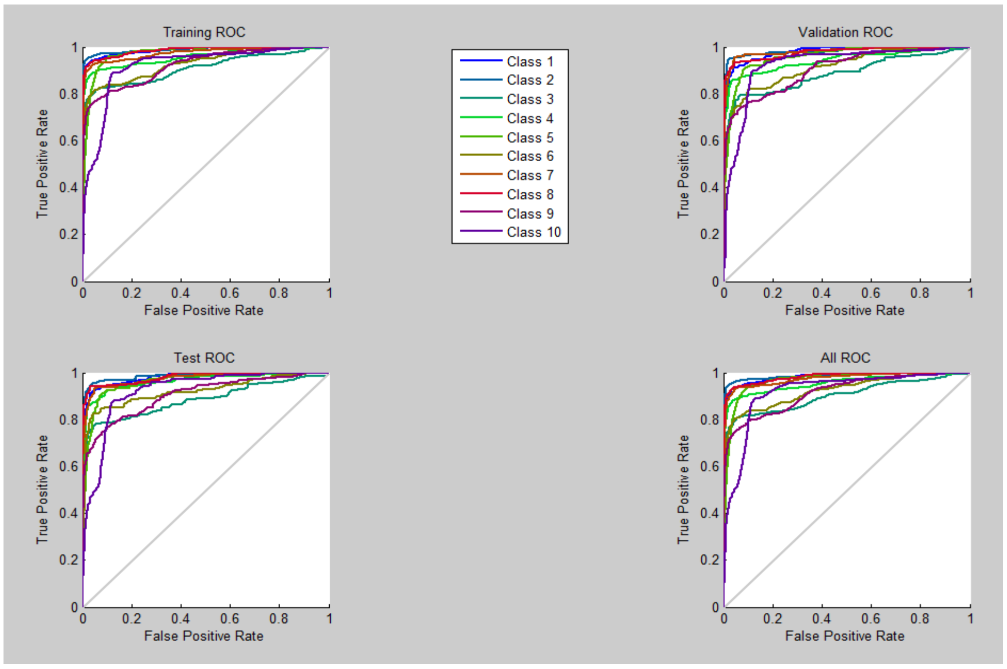


Figure 2.26: ROC curve for 1500 images, 1 hidden layer network with 10 neurons

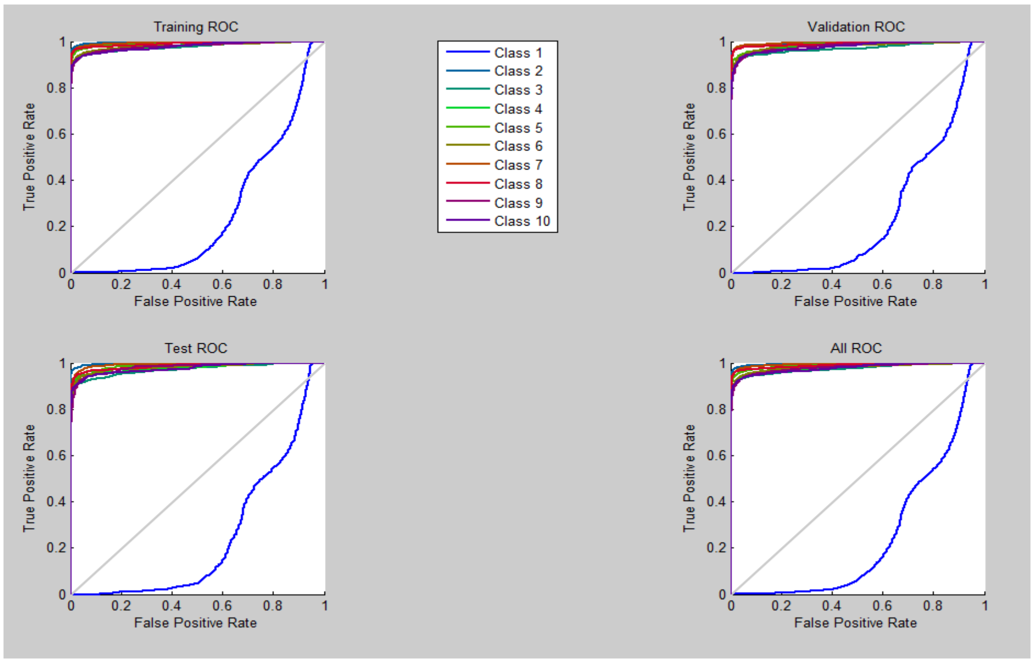


Figure 2.27: ROC curve for4500 images, 1 hidden layer network with 10 neurons

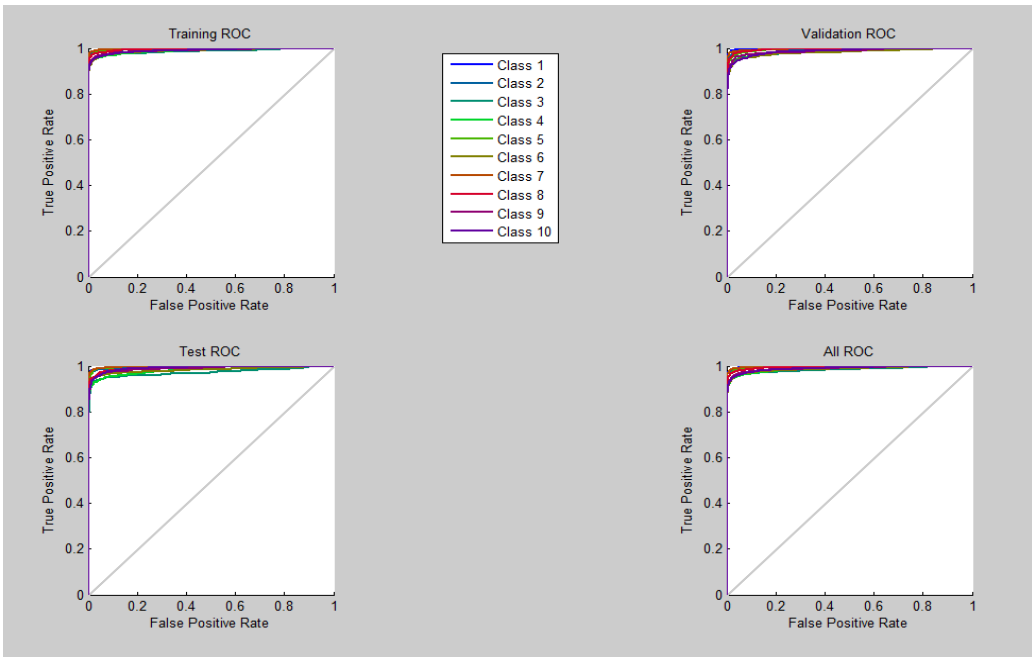


Figure 2.28: ROC curve for5000 images, 1 hidden layer network with 30 neurons

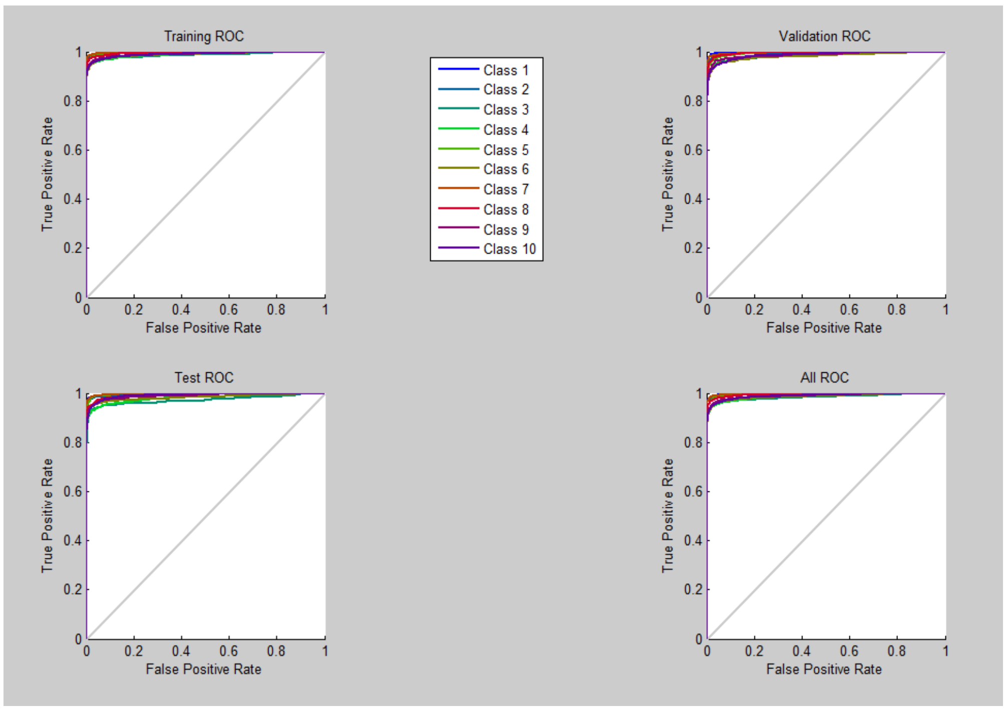


Figure 2.29: ROC curve for5000 images, 1 hidden layer network with 30 neurons

2.5.7 Graphical Interpretation

Summing up the results from all the Confusion Matrices, we have the following results:

Number of neurons in the hidden layer	Validation	Error
10	66.7%	34.3%
50	93.8%	6.2%
80	96.9%	3.1%

Table 2.2: The performance of the neural network woth 500 images as an input data set

Number of neurons in the hidden layer	Validation	Error
10	81%	19%
50	96.9%	3.1%
80	98.1%	1.9%
200	98.8%	1.2%
2000	98.8%	1.2%

Table 2.3: The performance of the neural network woth 5000 images as an input data set

After running the neural network by changing the size of the input data set and the number of neurons from the hidden layer, if we analyze the graphs that encapsulate some brief results, we can easily notice that the best performances are when the number of neurons is bigger. Therefore, the higher the number of neurons, the higher the performance of the artificial neural network is.

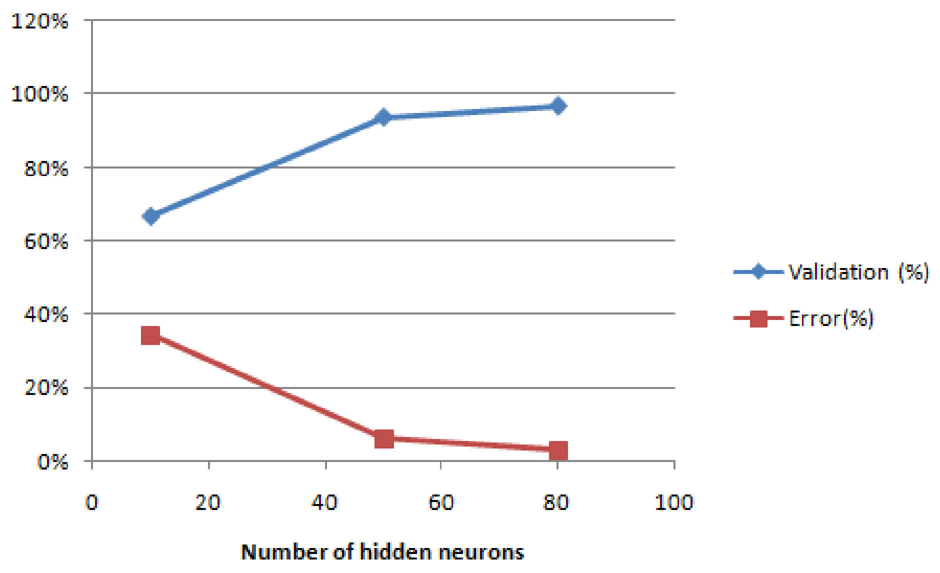


Figure 2.30: The performance of the neural network woth 500 images as an input data set

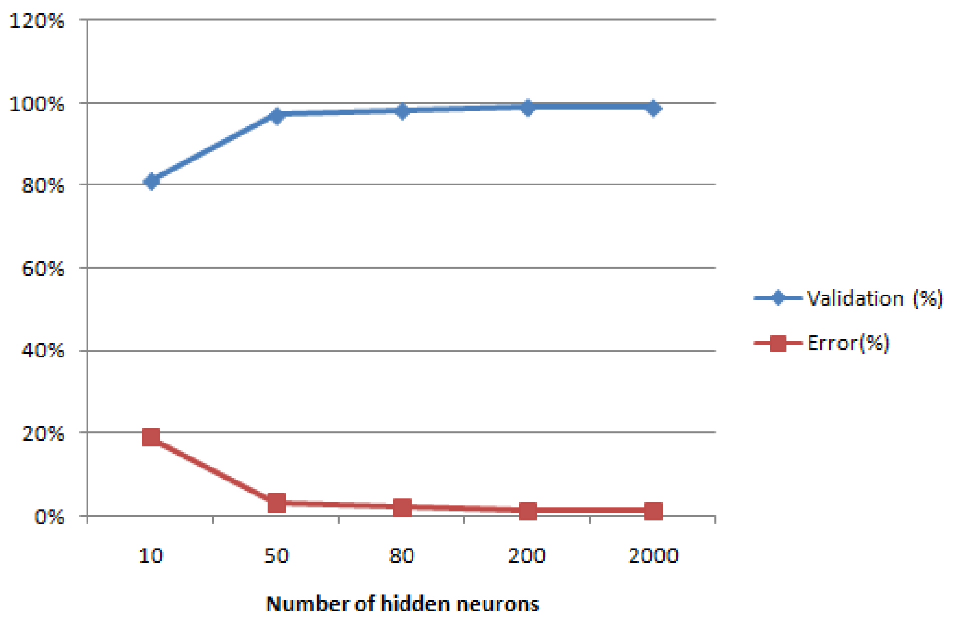


Figure 2.31: The performance of the neural network woth 5000 images as an input data set

Gathering the results from all the best validation performance plots, we can come up with the following table and interpretation:

Input data set	Number of neurons	Epochs	Best Validation Performance
500	30	81/87	0.017959
500	80	99/105	0.013895
4500	10	325/331	0.023999
5000	10	468/474	0.026949
30	30	119/125	0.0113777

Table 2.4: Best Validation Performance

The error rates get better if we increase the number of neurons in the hidden layer up to a certain point. For all that, it is not always increasing with number of hidden neurons. As the number grows, the run time also grows proportionally with the number of neurons in the hidden layers. To decrease the straining time, a PC with at least 8GB RAM can be used. In our situation, the simulation were made with 4GB RAM when the input data set was small and the supported number of neurons in the hidden layer was 80, the rest of the simulations were done on a PC with 16GB RAM.

2.6 Used tools for design: Neural Network Toolbox[31]

Neural Network Toolbox^{TM} provides functions an applications for modeling complex and nonlinear systems. Neural Network Toolbox supports supervised learning with feedforward networls; it also supports unsupervised learning. With this this toolbox, one can design, train, visualize and simulate artificial neural networks.

In our project, we used *Neural Network Pattern Recognition Tool;* to open it, one can use the command *nprtool()*.

[31] Source: Mathworks, MATLAB documentation, MATLAB version R2018a, 2018

Chapter 3: Live Application

3.1 Live Application - Block Diagram

The block diagram of the live application can be depicted in Figure 3.1.

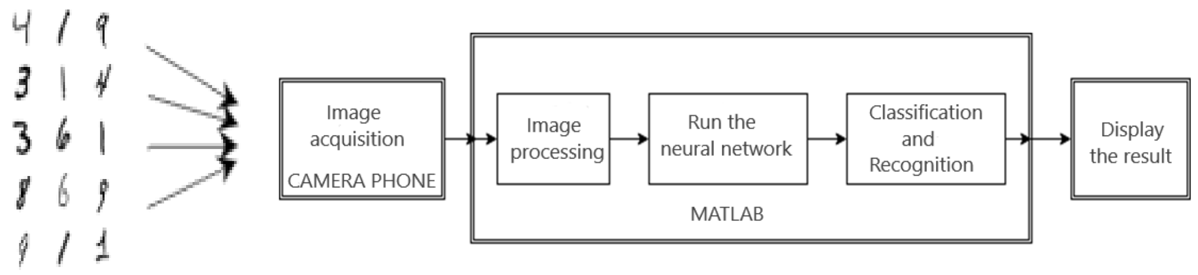


Figure 3.1. : Block diagram of the application

The main blocks of the diagram block are as follows:

- → *Image acquisition* block: it is represented by a phone camera ;
- → *Image processing* block: the image is being processed by MATLAB program;
- → *Run the neural network* block: in this block the neural network is being run in order to be able to classify and recognize the class;
- → *Classification and Recognition* block: after the neural network has been run, in this part of the application reads the outputs of the network and determines the largest value, which indicates the predicted class;
- → *Display the results* block: in this part, the application is displaying in the command line from which class is the acquired image.

During the processing of the image, a series of operations are performed on the acquired image. The role of the processing is to segment the interesting pattern from the background, the handwritten digit. Binarizations process converts a gray scale image into a binary image, meaning a black and white image. The character is uniformly resized into an image with 28x28 pixels.

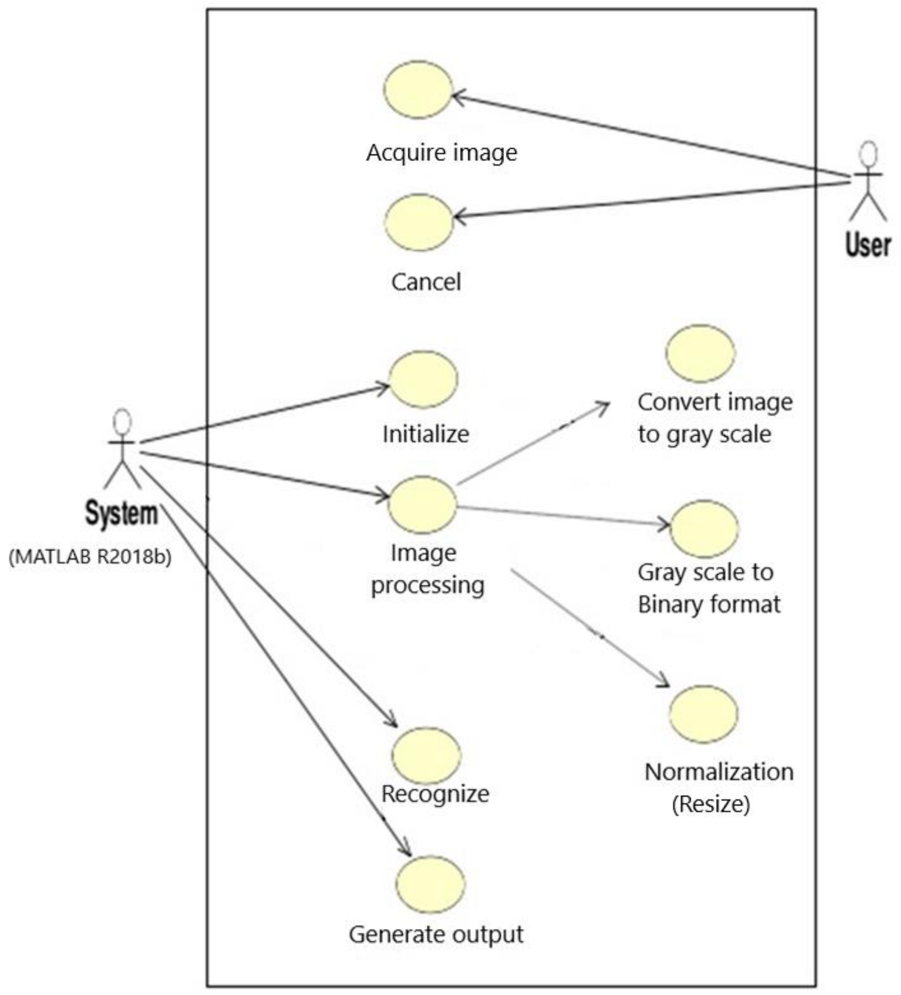


Figure 3.2: Use case diagram

The images are always acquired by the system using a camera that can be the camera of a mobile phone connected to MATLAB via Uniform Resource Locator (URL) address, or web address, which is a reference to a web resource that specifies its location on a computer network and a mechanism for rectifying it. There exists the possibility for the image to not be able to be uploaded from various reasons such as lack of internet connection or if the mobile phone and the laptop are connected to different internet sources. More about possible error sources will be discussed in one of the following sections.

After the images was successfully acquired and read by MATLAB, it is processed. During the processing of the image, a series of operations are performed on the procured image.

Hence, in the processing stage, a series of actions or steps are taken such that the images have the same format as the images from the MNIST Data Base, these steps are fully neccesary for the model to achieve the target:

- The model segments the interesting pattern from the background, the handwritten digit

- The RGB image is converted into a grayscale image
- The binarizations process converts the grayscale image as follows: the background is converted to black and the foreground into white
- The image is uniformly resized into 28 x 28 pixels.

The image is processed and classified as indicated in the block diagram, obtaining information about the digit in the image.

3.2 Possible improvements

Some improvements that can be done for better accuracy:

- A *segmentation* block can be added, if a sequence of digits is written, the image will be segmented into isolated characters using a labeling process which will provide information about the number of the symbols in the image; each individual digit is resized into pixels afterwards.
- A *feature extraction technique* can be used based on the geometry of the character, or digit, for character recognition
- Using more than one hidden layer can also increase the performance and the accuracy of the model for a better pattern recognition
- C*onvolutional neural networks* can be used, they are more popular and more efficient, therefore the accuracy will increase, also the time will be improved

3.3 Possible error sources

The problem of image classification consists in the following process: given an input set of images (the MNIST Data Base) labeled per class, after being trained with the labeled images, the neural network is being asked to predict these categories for a new set of test images, the acquired images with the camera, and to measure the accuracy of the prediction. Yet, there are a variety of cases when things can turn around and we face challenges such as:

- Viewpoint variation: if the camera is not oriented in a good angle, the program will not be able to recognize or to correctly classify the pattern.
- Scale variation: there are chances that the model could meet for the first time an image from a certain position, even though the MNIST Data Base is quite large.

- Image deformation: there could appear local changes occurred in the presence of image object variability.
- Illumination conditions: if it is too dark, the processed image will have a lot of noise and eventually, the model will get confused and will not be able to classify the digit.
- Background disturbances: if the background is not clear, white, there will be some noise in the processing image.

- Matlab is an interpreted language, which means there will always be a problem with the execution speed, this being the main disadvantage of interpreted languages.

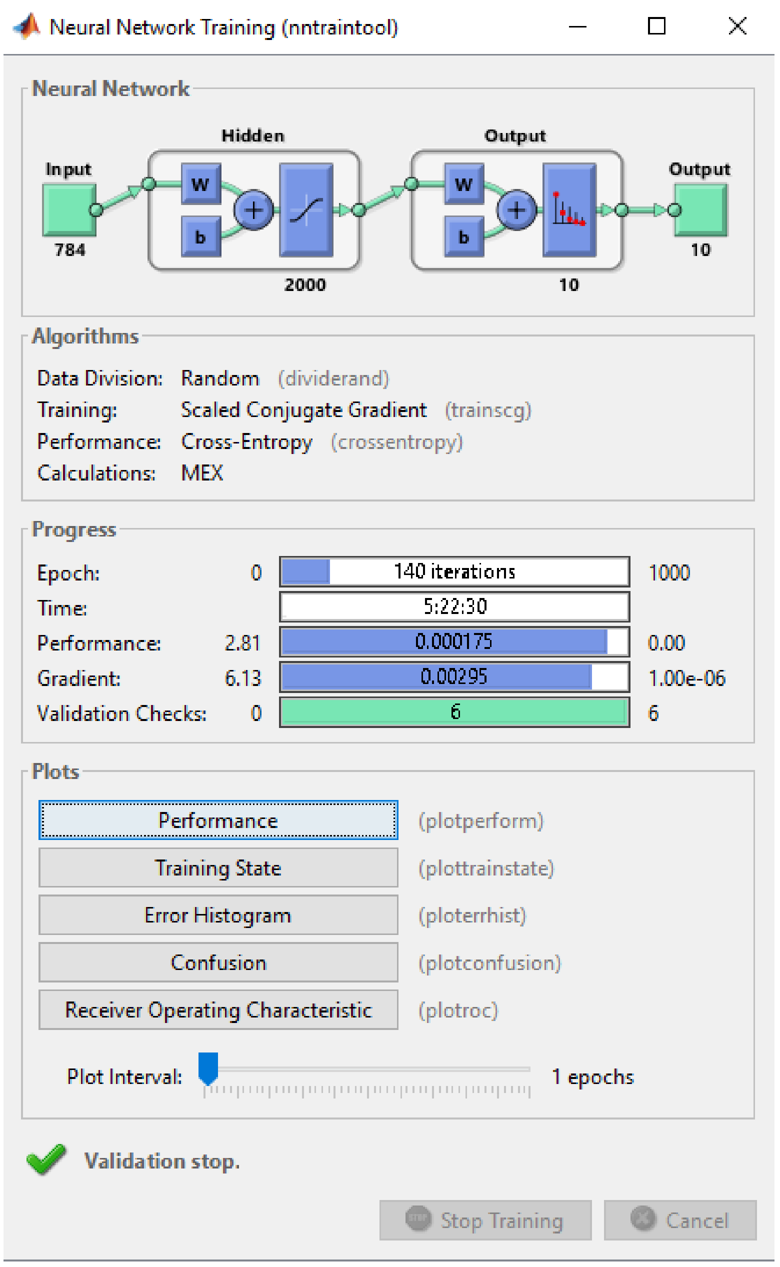
- The differences between the handwrittings; the handwritting among the images from the MNIST Data Base is in American style, which is slighlty different from the European, or Romanian, manner

- If the user moves during the image acquisition, the model might get confused and misclasify the digit.

3.4 How does it work live?

In this subchapter we will talk about the live application and we will show you some screenshots that underlie that it works in real life and the results are reasonable and solid.
Now, let’s see how the whole process works:

Firstly, we run the script *train_neural_network*, this way the model learns and memorizes the patterns of the handwritten digits. As mentioned previously, the neural network is trained using Neural Pattern Recognition Toolbox from MATLAB R2018b and as for the input data, the MNIST Data Base is used. The training process is *time consuming* because each class of digits has at least 5000 images and for better accuracy we will use a neural network with 2000 neurons in the hidden layer. The Neural Network Training window from MATLAB can be seen in Figure 3.3.


Neural Network Training (nntraintool)
Neural Network
Hidden
Output
Input
784
W
b
2000
W
b
10
Output
10

Algorithms

Data Division: Random (dividerand)
Training: Scaled Conjugate Gradient (trainscg)
Performance: Cross-Entropy (crossentropy)
Calculations: MEX

Progress

Epoch:	0	140 iterations	1000
Time:		5:22:30	
Performance:	2.81	0.000175	0.00
Gradient:	6.13	0.00295	1.00e-06
Validation Checks:	0	6	6

Plots

Performance (plotperform)
Training State (plottrainstate)
Error Histogram (ploterrhist)
Confusion (plotconfusion)
Receiver Operating Characteristic (plotroc)

Plot Interval: 1 epochs

Validation stop.

Figure 3.3: Neural Network Training (Time consuming: 5h 22min 30s)

Afterwards, we run the script of the actual application that is used to acquire the image of the hand-drawn digit.

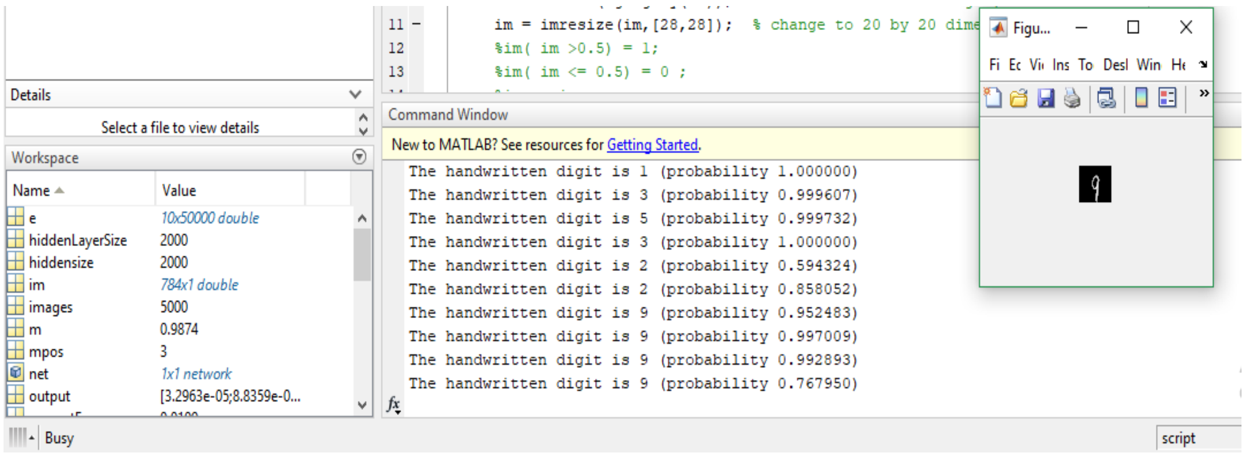
It is important that the digit is clearly written on a white paper with a dark color marker, preferably black. It is better if we use a marker instead of a pen because the character will be more obvious and it can be easily traced by the model. If a normal pen is used, the lines of the character will be too thin, the acquired image will be mostly white and after the processing of the image, the result will be a black image, hence the model will not be able to trace the character. Another important aspect is the light, if the light is poor, some noise will appear and the accuracy will decrease.

The processing operations consist in: converting the image into a grayscale image; converting the grayscale image into a black and white image, process also known as binarization; the image is normalized which means it is resized into a 28 by 28 pixels image, as the MNIST data set of images.

At this point, the model should be able to recognize the digit and classify it in the corresponding class.

3.5 Live Application - Screen shots

Let's take some examples of hand-drawn digits and see how the application behaves:



```
im = imresize(im,[28,28]);  % change to 20 by 20 dime
%im( im >0.5) = 1;
%im( im <= 0.5) = 0 ;
```

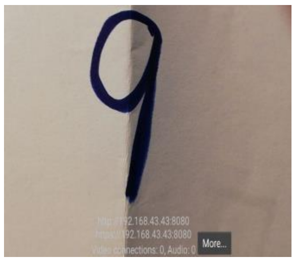
Command Window

New to MATLAB? See resources for Getting Started.

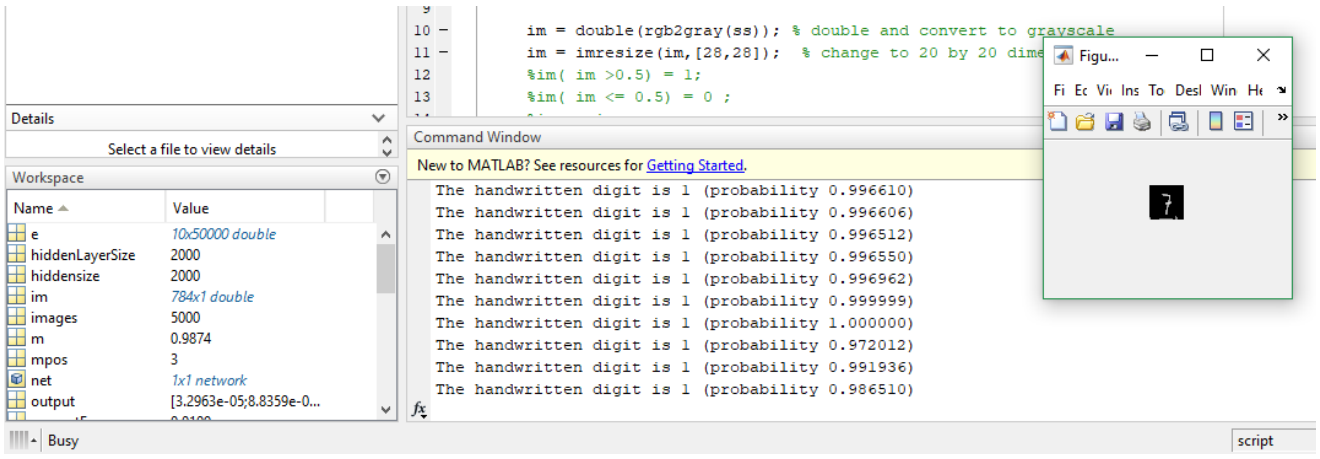
```
The handwritten digit is 1 (probability 1.000000)
The handwritten digit is 3 (probability 0.999607)
The handwritten digit is 5 (probability 0.999732)
The handwritten digit is 3 (probability 1.000000)
The handwritten digit is 2 (probability 0.594324)
The handwritten digit is 2 (probability 0.858052)
The handwritten digit is 9 (probability 0.952483)
The handwritten digit is 9 (probability 0.997009)
The handwritten digit is 9 (probability 0.992893)
The handwritten digit is 9 (probability 0.767950)
```

Name	Value
e	10x50000 double
hiddenLayerSize	2000
hiddensize	2000
im	784x1 double
images	5000
m	0.9874
mpos	3
net	1x1 network
output	[3.2963e-05;8.8359e-0...

Figure 3.4: Digit 9 Case



In the case of digit 9, the application firsly classifies it as digit 1, 2, 3, 5. Some possible reasons for that would be the fact that the background has poor light, the paper is bent in the middle and bassically it cuts the digit and this is why it may look as digit 1. Because it has a loop at the top, it may trick the model that it is a 2 or a 3 or a 5. But overall, the result is pretty decent and the model keeps displaying the digit 9 for multiple times with a good probabilty.



```
im = double(rgb2gray(ss)); % double and convert to grayscale
im = imresize(im,[28,28]);  % change to 20 by 20 dime
%im( im >0.5) = 1;
%im( im <= 0.5) = 0 ;
```

Details

Select a file to view details

Workspace

Name	Value
e	10x50000 double
hiddenLayerSize	2000
hiddensize	2000
im	784x1 double
images	5000
m	0.9874
mpos	3
net	1x1 network
output	[3.2963e-05;8.8359e-0...

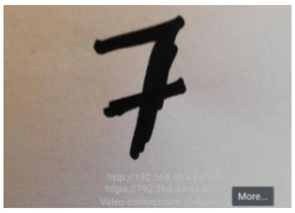
Command Window

New to MATLAB? See resources for Getting Started.

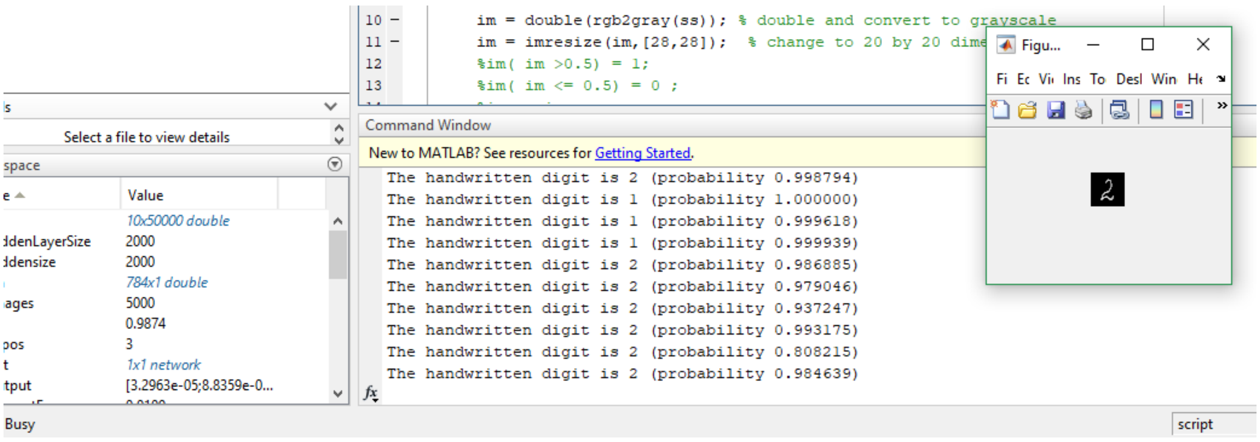
```
The handwritten digit is 1 (probability 0.996610)
The handwritten digit is 1 (probability 0.996606)
The handwritten digit is 1 (probability 0.996512)
The handwritten digit is 1 (probability 0.996550)
The handwritten digit is 1 (probability 0.996962)
The handwritten digit is 1 (probability 0.999999)
The handwritten digit is 1 (probability 1.000000)
The handwritten digit is 1 (probability 0.972012)
The handwritten digit is 1 (probability 0.991936)
The handwritten digit is 1 (probability 0.986510)
```

Busy

Figure 3.5: Digit 7 Case (Error)



This case is a total failure; the model does not recognize the pattern of the digit 7 correctly not even once. Because of the poor light in the background and because the vertical line is thicker than the horizontal line, the model classifies it as digit 1.



```
im = double(rgb2gray(ss)); % double and convert to grayscale
im = imresize(im,[28,28]);  % change to 20 by 20 dime
%im( im >0.5) = 1;
%im( im <= 0.5) = 0 ;
```

Select a file to view details

Value
10x50000 double
2000
2000
784x1 double
5000
0.9874
3
1x1 network
[3.2963e-05;8.8359e-0...

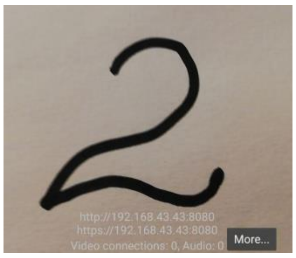
Command Window

New to MATLAB? See resources for Getting Started.

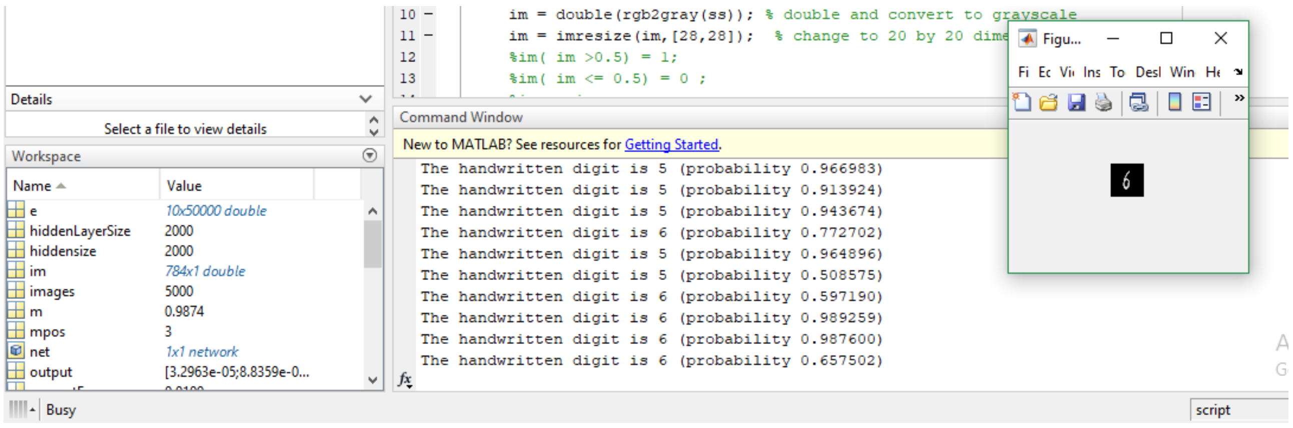
```
The handwritten digit is 2 (probability 0.998794)
The handwritten digit is 1 (probability 1.000000)
The handwritten digit is 1 (probability 0.999618)
The handwritten digit is 1 (probability 0.999939)
The handwritten digit is 2 (probability 0.986885)
The handwritten digit is 2 (probability 0.979046)
The handwritten digit is 2 (probability 0.937247)
The handwritten digit is 2 (probability 0.993175)
The handwritten digit is 2 (probability 0.808215)
The handwritten digit is 2 (probability 0.984639)
```

Busy

Figure 3.6: Digit 2 Case



In the case of digit 2, the result are pretty good, even though for 3 times it displays the digit 1, overall, the model classifies it as digit 2 with a very good probability.



```
im = double(rgb2gray(ss)); % double and convert to grayscale
im = imresize(im,[28,28]);  % change to 20 by 20 dime
%im( im >0.5) = 1;
%im( im <= 0.5) = 0 ;
```

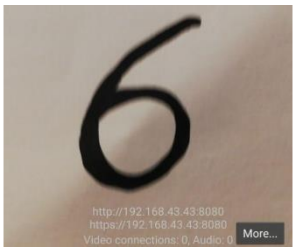
Command Window

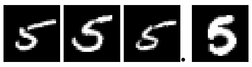
New to MATLAB? See resources for Getting Started.

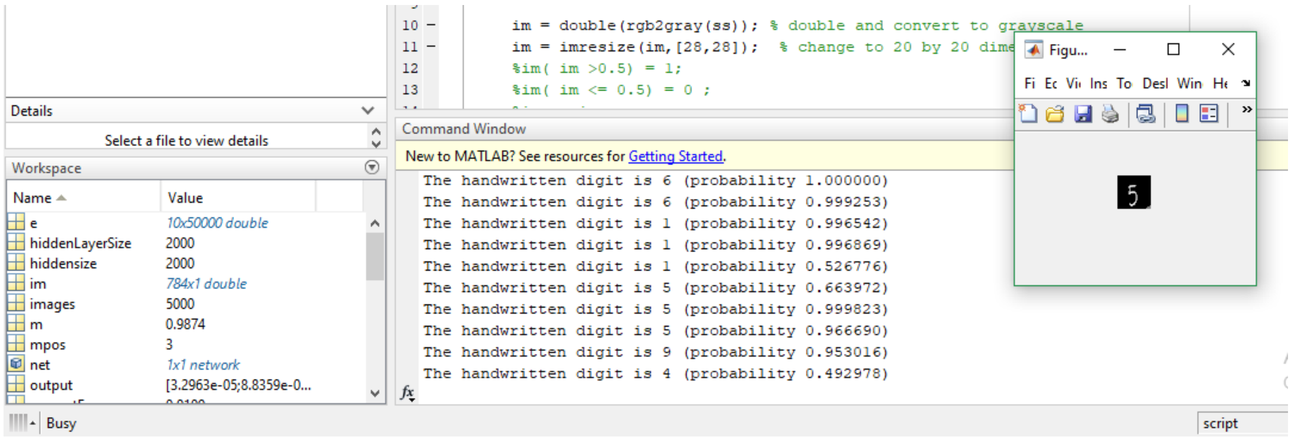
```
The handwritten digit is 5 (probability 0.966983)
The handwritten digit is 5 (probability 0.913924)
The handwritten digit is 5 (probability 0.943674)
The handwritten digit is 6 (probability 0.772702)
The handwritten digit is 5 (probability 0.964896)
The handwritten digit is 5 (probability 0.508575)
The handwritten digit is 6 (probability 0.597190)
The handwritten digit is 6 (probability 0.989259)
The handwritten digit is 6 (probability 0.987600)
The handwritten digit is 6 (probability 0.657502)
```

Name	Value
e	10x50000 double
hiddenLayerSize	2000
hiddensize	2000
im	784x1 double
images	5000
m	0.9874
mpos	3
net	1x1 network
output	[3.2963e-05;8.8359e-0...

Figure 3.7: Digit 6 Case



At the beginning, it classifies the digit 6 as digit 5, the reasons is that in the training data set there are at least 5000 pictures in class 5 and some of them look like digit 6: . But eventually, the handwritten digit is classified as digit 6 with a very good probability.



```
im = double(rgb2gray(ss)); % double and convert to grayscale
im = imresize(im,[28,28]);  % change to 20 by 20 dime
%im( im >0.5) = 1;
%im( im <= 0.5) = 0 ;
```

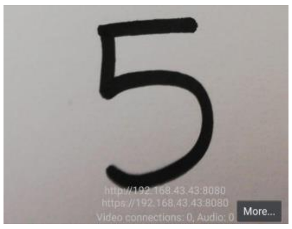
Command Window

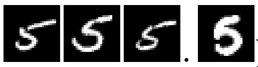
New to MATLAB? See resources for Getting Started.

```
The handwritten digit is 6 (probability 1.000000)
The handwritten digit is 6 (probability 0.999253)
The handwritten digit is 1 (probability 0.996542)
The handwritten digit is 1 (probability 0.996869)
The handwritten digit is 1 (probability 0.526776)
The handwritten digit is 5 (probability 0.663972)
The handwritten digit is 5 (probability 0.999823)
The handwritten digit is 5 (probability 0.966690)
The handwritten digit is 9 (probability 0.953016)
The handwritten digit is 4 (probability 0.492978)
```

Name	Value
e	10x50000 double
hiddenLayerSize	2000
hiddensize	2000
im	784x1 double
images	5000
m	0.9874
mpos	3
net	1x1 network
output	[3.2963e-05;8.8359e-0...

Figure 3.8: Digit 5 Case



We mention that the first 2 lines from the command line are from the previous case, when digit 6 was tested; yet, it is possible for digit 5 to be misclassified as digit 6 (see MNIST training data set examples: )On average, the digit is classified correctly with a good probability.

Conclusions

Pattern recognition using image processing and artificial neural networks, which is among the most active field nowadays, have been in a continuous growth for the last years. One of the reasons of interest in artificial neural networks is the hope that one day they will exceed the limits of basic pattern recognition problems. Even though the domain of artificial intelligence is rooted in the last century and it was present in our lives for many decades, we can easily say that at the present time it is the era of artificial intelligence.

Everyone speculates that with the help of neural networks or some other approaches implying digital computers there will be found a solution used to build thinking machines, which may exceed the intelligence of the humans. Pattern recognition can be done both in normal computers and neural networks. But there is a slight difference between the two methods, yet significant. Computers use conventional arithmetic algorithms to detect whether the given pattern matches an existing one. It is a straightforward method that will say either yes or no. It does not tolerate noisy patterns. On the other hand, neural networks can tolerate noise and, if trained properly, will respond correctly for unknown patterns. Even though neural networks may not perform miracles, if they are built with the proper architecture and trained correctly with a good input data set, they will give amazing results, not only in pattern recognition but also in other scientific and commercial applications that will come in hand, as solutions, to different problems of the mankind.

In this thesis, a simplistic approach for recognition of handwritten digits using artificial neural networks has been described. At the current stage of development, the software does perform well either in terms of speed or accuracy but some improvements could be done. We can say that the algorithm will work adequately in some situations, but it also has limitations if it is used in more complex applications.

The model was able to successfully classify handwritten digits, despite the fact that the images from MNIST Data Base have American writing style and the live application tests the neural network on European writing style, even though it does not seem to be a huge problem, it affects the accuracy of the application. Overall, the results still remained reasonable and solid.

GLOSSARY[32]

bias
Neuron parameter that is summed with the neuron's weighted inputs and passed through the neuron's transfer function to generate the neuron's output.

bias vector
Column vector of bias values for a layer of neurons.

classification
Association of an input vector with a particular target vector.

epoch
Presentation of the set of training (input and/or target) vectors to a network and the calculation of new weights and biases. Note that training vectors can be presented one at a time or all together in a batch.

error jumping
Sudden increase in a network's sum-squared error during training. This is often due to too large a learning rate.

error ratio
Training parameter used with adaptive learning rate and momentum training of back-propagation networks.

error vector
Difference between a network's output vector in response to an input vector and an associated target output vector.

feedback network
Network with connections from a layer's output to that layer's input. The feedback connection can be direct or pass through several layers.

feed-forward network
Layered network in which each layer only receives inputs from previous layers.

[32]Source: *Neural Network Toolbox™6, User's Guide,* Howard Demuth, Mark Beale, Martin Hagan

gradient descent
Process of making changes to weights and biases, where the changes are proportional to the derivatives of network error with respect to those weights and biases. This is done to minimize network error.

hidden layer
Layer of a network that is not connected to the network output (for instance, the first layer of a two layer feed-forward network).

initialization
Process of setting the network weights and biases to their original values.

input layer
Layer of neurons receiving inputs directly from outside the network.

input vector
Vector presented to the network.

input weight vector
Row vector of weights going to a neuron.

input weights
Weights connecting network inputs to layers.

layer
Group of neurons having connections to the same inputs and sending outputs to the same destinations

linear transfer function
Transfer function that produces its input as its output

local minimum
Minimum of a function over a limited range of input values. A local minimum might not be the global minimum.

mean square error function
Performance function that calculates the average squared error between the network outputs **a** and the target outputs **t**.

neuron
Basic processing element of a neural network. Includes weights and bias, a summing junction, and an output transfer function. Artificial neurons, such as those simulated and trained with this toolbox, are abstractions of biological neurons.

output layer
Layer whose output is passed to the world outside the network.

output vector
Output of a neural network. Each element of the output vector is the output of a neuron.

output weight vector
Column vector of weights coming from a neuron or input.

overfitting
Case in which the error on the training set is driven to a very small value, but when new data is presented to the network, the error is large.

pattern
A vector.

pattern association
Task performed by a network trained to respond with the correct output vector for each input vector presented.

pattern recognition
Task performed by a network trained to respond when an input vector close to a learned vector is presented. The network "recognizes" the input as one of the original target vectors.

recognition
The network "recognizes" the input as one of the original target vectors.

perceptron
Single-layer network with a hard-limit transfer function. This network is often trained with the perceptron learning rule.

performance
Behavior of a network.

performance function
Commonly the mean squared error of the network outputs. However, the toolbox also considers other performance functions.

sigmoid
Monotonic S-shaped function that maps numbers in the interval $(-\infty,\infty)$ to a finite interval such as (-1,+1) or (0,1).

sum-squared error
Sum of squared differences between the network targets and actual outputs for a given input vector or set of vectors.

supervised learning
Learning process in which changes in a network’s weights and biases are due to the intervention of any external teacher. The teacher typically provides output targets.

target vector
Desired output vector for a given input vector.

test vectors
Set of input vectors (not used directly in training) that is used to test the trained network.

training
Procedure whereby a network is adjusted to do a particular job. Commonly viewed as an offline job, as opposed to an adjustment made during each time interval, as is done in adaptive training.

training vector
Input and/or target vector used to train a network.

transfer function
Function that maps a neuron’s (or layer’s) net output **n** to its actual output.

unsupervised learning
Learning process in which changes in a network’s weights and biases are not due to the intervention of any external teacher. Commonly changes are a function of the current network input vectors, output vectors, and previous weights and biases.

validation vectors
Set of input vectors (not used directly in training) that is used to monitor **s**training progress so as to keep the network from overfitting.

weight function
Weight functions apply weights to an input to get weighted inputs, as specified by a particular function.

weight matrix
Matrix containing connection strengths from a layer's inputs to its neurons. The element $w_{i,j}$ of a weight matrix W refers to the connection strength from input j to neuron i.

weighted input vector
Result of applying a weight to a layer's input, whether it is a network

vector
Input or the output of another layer.

Bibliography

[1] *Neural Networks*, Dr. Professor Eng. Iulian Ciociu

[2] *Neural Networks, A Comprehensive Foundation (*Second Edition), Simon Haykin, McMaster University, Hamilton, Ontario, Canada

[3] *Neural Networks and Learning Machines* (Third Edition), Simon Haykin, McMaster University, Hamilton, Ontario, Canada

[4] *The perceptron: A probabilistic Model for information storage and organization in the brain*, F. Rossenblatt

[5] *Neural Networks and Deep Learning*, Michael Nielsen, *http://neuralnetworksanddeeplearning.com*

[6] TutorialsPoint: *https://www.tutorialspoint.com/artificial_neural_network/index.htm*

[7] Towards Data Science: *https://towardsdatascience.com*

[8] Stack Overflow: *https://stackoverflow.com*

[9] *Beginner's guide to neural networks for the MNIST dataset using MATLAB*, Bitna Kim, Young Ho Park

[10] Andrew Y. Ng, Course on machine learning, Coursera: *https://www.coursera.org/learn/machine-learning*

[11] MNIST, *http://yann.lecun.com/exdb/mnist/*

[12] Wikipedia, https://en.wikipedia.org/

[13] Mathworks, MATLAB documentation, MATLAB version R2018a, 2018

[14] *Neural Network Toolbox*^{TM}*6, User's Guide, Howard* Demuth, Mark Beale, Martin Hagan

Appendices

A.1 The script used to prepare the data set

```
X = zeros(28*28, 502);
k=1;


for label= 0:9

    cd(num2str(label));
    l = dir('.');
    for i = 3:502
        file = l(i).name;
        y = imread(file); %reads the pics from the folders
        y_converted = double(y)/255;  %  y is an uint
        yy = reshape(y, 28*28, 1);
        X(:,k) = yy;
        k = k+1; %columns
    end
    cd('..');
end

T = full(ind2vec( [ ones(1,500), 2*ones(1,500), 3*ones(1,500), 4*ones(1,500), 5*ones(1,500),
6*ones(1,500), 7*ones(1,500), 8*ones(1,500), 9*ones(1,500),10*ones(1,500) ] ))
%full(T)
save( 'minst.mat','X', 'T' );
```

A.2 The script used to train the neural network

```
% Solve a Pattern Recognition Problem with a Neural Network
% Script generated by Neural Pattern Recognition app
% Created 23-Mar-2019 13:18:31
%
% This script assumes these variables are defined:
%
%   X - input data.
%   T - target data.

clear all
close all

% Parameters
%==============================
load('minst_5000perclass.mat');
```

```
%load('mnist_4000perclass.mat');

hiddensize = 2000;
images = 5000;

%===============================
x = X;
t = T;

% Choose a Training Function
% For a list of all training functions type: help nntrain
% 'trainlm' is usually fastest.
% 'trainbr' takes longer but may be better for challenging problems.
% 'trainscg' uses less memory. Suitable in low memory situations.
trainFcn = 'trainscg';  % Scaled conjugate gradient backpropagation.

% Create a Pattern Recognition Network
hiddenLayerSize = hiddensize;
net = patternnet(hiddenLayerSize, trainFcn);

% Choose Input and Output Pre/Post-Processing Functions
% For a list of all processing functions type: help nnprocess
net.input.processFcns = {'removeconstantrows','mapminmax'};

% Setup Division of Data for Training, Validation, Testing
% For a list of all data division functions type: help nndivision
net.divideFcn = 'dividerand';  % Divide data randomly
net.divideMode = 'sample';  % Divide up every sample
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;

% Choose a Performance Function
% For a list of all performance functions type: help nnperformance
net.performFcn = 'crossentropy';  % Cross-Entropy

% Choose Plot Functions
% For a list of all plot functions type: help nnplot
net.plotFcns = {'plotperform','plottrainstate','ploterrhist', ...
    'plotconfusion', 'plotroc'};

% Train the Network
[net,tr] = train(net,x,t);

% Test the Network
y = net(x);
e = gsubtract(t,y);
performance = perform(net,t,y)
tind = vec2ind(t);
yind = vec2ind(y);
```

```
percentErrors = sum(tind ~= yind)/numel(tind);

% Recalculate Training, Validation and Test Performance
trainTargets = t .* tr.trainMask{1};
valTargets = t .* tr.valMask{1};
testTargets = t .* tr.testMask{1};
trainPerformance = perform(net,trainTargets,y)
valPerformance = perform(net,valTargets,y)
testPerformance = perform(net,testTargets,y)

% View the Network
%view(net)

% Plots
% Uncomment these lines to enable various plots.
figure, plotperform(tr)
%figure, plottrainstate(tr)
%figure, ploterrhist(e)
figure, plotconfusion(t,y)
%figure, plotroc(t,y)

% Deployment
% Change the (false) values to (true) to enable the following code blocks.
% See the help for each generation function for more information.
if (false)
    % Generate MATLAB function for neural network for application
    % deployment in MATLAB scripts or with MATLAB Compiler and Builder
    % tools, or simply to examine the calculations your trained neural
    % network performs.
    genFunction(net,'myNeuralNetworkFunction');
    y = myNeuralNetworkFunction(x);
end
if (false)
    % Generate a matrix-only MATLAB function for neural network code
    % generation with MATLAB Coder tools.
    genFunction(net,'myNeuralNetworkFunction','MatrixOnly','yes');
    y = myNeuralNetworkFunction(x);
end
if (false)
    % Generate a Simulink diagram for simulation or deployment with.
    % Simulink Coder tools.
    gensim(net);
end

savefilename = sprintf('results_%dperclass_hiddensize%d.mat', images, hiddensize);
save(savefilename);
```

A.3 The script used in the live application to acquire and process the image

```
url = 'http://192.168.0.100:8080/shot.jpg'; % home wi-fi
%url = 'http://192.168.43.43:8080/shot.jpg'; % mobile data
ss  = imread(url);
%fh = image(ss);
%fh = image();
figure(1);
while(1)
    ss  = imread(url);

    im = double(rgb2gray(ss)); % double and convert to grayscale
    im = imresize(im,[28,28]);  % change to 20 by 20 dimension
    %im( im >0.5) = 1;
    %im( im <= 0.5) = 0 ;
    %im = ~im;
    im = 255 - im; %Black to white (invers)

    im( im<128 ) = 0;

    figure(1)
    imshow(im/255)
    %set(fh,'CData',im);
    %drawnow;

    im = im(:); % unroll matrix to vector

    output = net(im);
    [m, mpos] = max(output);
    fprintf('The handwritten digit is %d (probability %f)\n', mpos-1, m);
        end
```