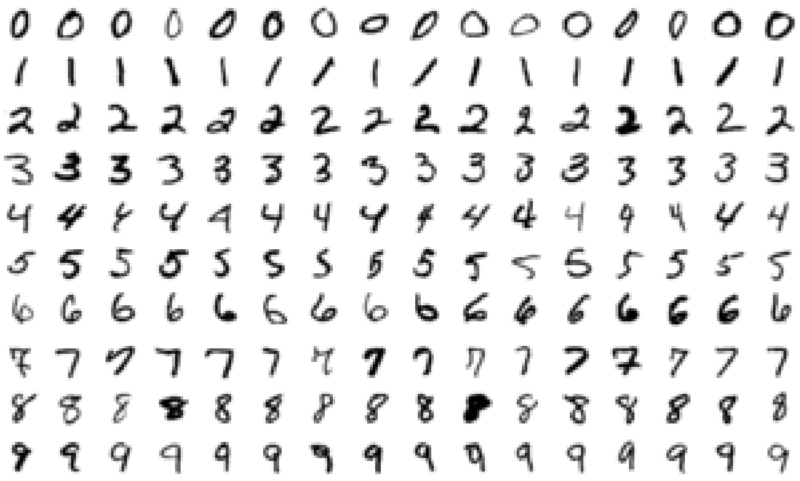
# Image Processing

# Handwritten Digit Classification

Vasileios-Marios Panagakis – 1115201600123
National and Kapodistrian University of Athens
Athens, Greece



ABSTRACT. In the field of digital image processing, pattern recognition is the research area that studies the operation and design of systems that recognize patterns in data. Its primary goal is the classification of objects into a number of categories or classes. Depending on the use, these objects can be images, signal waveforms or any other type of measurements that need to be classified. Common applications of pattern recognition are automatic speech recognition, classification of text into several categories (e.g., spam/non-spam email messages) and image classification. In this project we tackle the handwritten digit recognition problem, a subproblem of the image classification task, where some handwritten digits must be assigned into one of 10 classes using some classification method. Our purpose is to produce the best possible classification results using the most modern machine learning techniques and programming tools. We perform our experiments on the well-known MNIST database that contains binary-level images. Finally, we manage to produce fast and accurate results by exploiting the Python programming language and the Keras API on the Google Colab platform.
**Keywords: digit, classification, images, MNIST, python**

## Contents

# 1. Introduction

## 1.1 Image Classification

Image Classification is a fundamental task that attempts to comprehend an entire image as a whole. The goal is to classify the image by assigning it to a specific label. Typically, image classification refers to images, in which only one class appears and is analyzed. Instances of a class can be "car", "animal", "building" and so on.

Classifying images is no big deal for humans. That is why it is a perfect example of Moravec's paradox [1] when it comes to machines. (The things we find easy are difficult for Artificial Intelligence). Early image classification relied on raw pixel data. This meant that computers would break down images into individual pixels. The problem is that two pictures of the same object can look very different. They can have different backgrounds, angles, poses, etc. This made it quite the challenge for computers to correctly "see" and categorize images, until Deep learning came into play.

## 1.2 Deep learning

Deep learning [2] is a type of machine learning, a subset of Artificial Intelligence (AI) that allows machines to learn from data. Deep learning involves the use of computer systems known as neural networks (Figure 1). In neural networks, the input is filtered through hidden layers of nodes. Each one of these nodes processes the input and communicates its results to the next layer of nodes. This repeats until the information reaches an output layer, and the machine provides its answer. There are different types of neural networks based on how the hidden layers work. Image classification with deep learning most often involves convolutional neural networks, or CNNs. In CNNs, the nodes in the hidden layers do not always share their output with every node in the next layer (known as convolutional layers). Deep learning allows machines to identify and extract features from images. This means they can learn the features to look for in images by analyzing lots of pictures. So, programmers do not need to enter these filters by hand.

Therefore, we have to exploit the most modern Deep learning techniques to carry through with our handwritten digit recognition problem. As we mentioned above, our purpose in this project is to produce the best possible classification results, meaning to predict correctly as many as possible handwritten digits of MNIST's test set (t10k-images.idx3-ubyte). To achieve this, we will build a complex Neural Network as follows:

1. Create and train an Autoencoder Neural Network.
2. Create and train a Classifier (Feed-Forward Neural Network), using the Encoder part of the Autoencoder.
3. Evaluate the Classifier's accuracy based on the provided labels of the test set (10k-labels.idx1-ubyte).

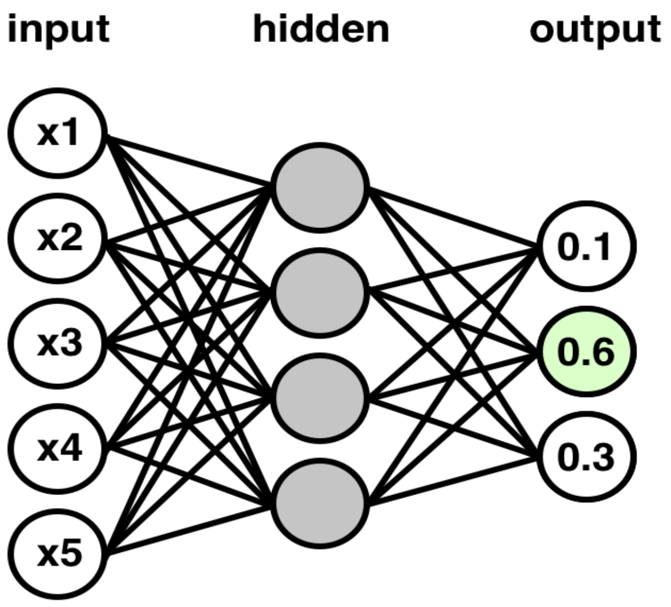


***Figure 1: Multilayer Perceptron with one hidden layer.***

## 2. Dataset

The MNIST database (*Modified National Institute of Standards and Technology database*) is a large database of handwritten digits (0 through 9) that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine learning. It is said, that MNIST is the “hello world” of machine learning. Data scientists will train an algorithm on the MNIST dataset simply to test new architectures or frameworks, to ensure that they work. MNIST database contains 60,000 training images and 10,000 testing images. Half of the training set and half of the test set were taken from NIST's training dataset, while the other halves were taken from NIST's testing dataset. Digits are size-normalized and centered in an image of size $28 \times 28$ by computing the center of mass of pixels and translating the image to locate this center point at the center of the $28 \times 28$ field. The MNIST set contains gray level images as a result of the antialiasing technique used by the normalization algorithm conducted by the creators of the dataset [3].

## 3. Network Architecture

### 3.1 Autoencoder

An Autoencoder [4] is an unsupervised artificial neural network that firstly learns how to efficiently compress and encode data and then learns how to reconstruct the encoded data to a representation that is as close to the original input as possible. By design, the Autoencoder reduces data dimensionality by learning how to ignore the noise in the data.

An Autoencoders consists of 4 main components:

1. **Encoder** - A stack of several layers, where the model learns how to reduce the input dimensions and compress the input data into an encoded representation.
2. **Bottleneck** - A layer that contains the compressed representation of the input data. This is the lowest possible dimensionality of the input data.

3. **Decoder** - A stack of several layers, where the model learns how to reconstruct the data from the encoded representation to be as close as possible to the original input.
4. **Reconstruction Loss** - A method that measures how well the decoder is performing and how close the output is to the original input.

The two main applications of Autoencoders are dimensionality reduction and information retrieval. Dimensionality reduction was one of the first deep learning applications. Its objective is to find a proper projection method that maps data from a high feature space to a lower feature space. Representing data in a lower-dimensional space can improve performance on tasks such as classification. Information retrieval is another task that benefits particularly from dimensionality reduction as it can become more efficient in certain kinds of low dimensional spaces. Autoencoders were proposed [5] and applied to semantic hashing. By training the algorithm to produce a low-dimensional binary code, all database entries could be stored in a hash table mapping binary code vectors to entries. This table would then support information retrieval by returning all entries with the same binary code as the query.

Apart from the traditional applications of Autoencoders some modern variations were proven successful when applied in machine learning tasks, that emerged over the years. Some instances of these tasks are Anomaly detection, Image processing, Drug discovery, Popularity prediction and Machine translation. A simple Autoencoder representation of our handwritten digit classification task is depicted in Figure 2.

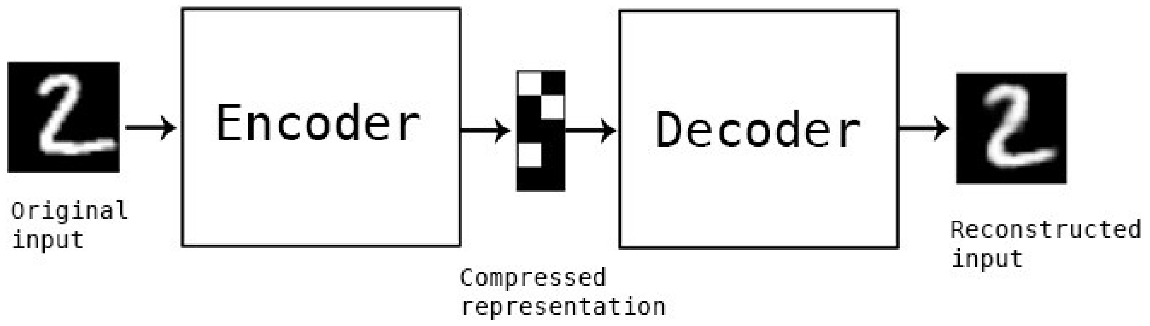


*Figure 2: Autoencoder representation on MNIST dataset.*

### 3.1.1 Encoder & Decoder

If we observe Figure 2, we realize that the most important components of an Autoencoder are the Encoder and the Decoder. As we mentioned before an Encoder is a stack of several layers, where the model learns how to reduce the input dimensions and compress the input data into an encoded representation. We can stack 2 types of layers in an Encoder implementation.

- **Convolution layer** – Responsible for the extraction of high-level features such as edges, from the input image.
- **Pooling layer** - Responsible for the decrease of the computational power required to process the data through dimensionality reduction. Furthermore, it is useful for extracting dominant features which are rotational and positional invariant, thus they help secure the effective model training.

After every Convolution layer we must apply the Batch Normalization technique. Batch normalization is a method used to make artificial neural networks faster and more stable through normalization of the layers' inputs by re-centering and re-scaling. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks. The extensive layer architecture of our Encoder is depicted in Figure 3.

A Decoder is a stack of several layers, where the model learns how to reconstruct the data from the encoded representation to be as close to the original input as possible. Inside an Autoencoder model the Decoder is implemented with the exact same layers as the Encoder of the model, but upside down.

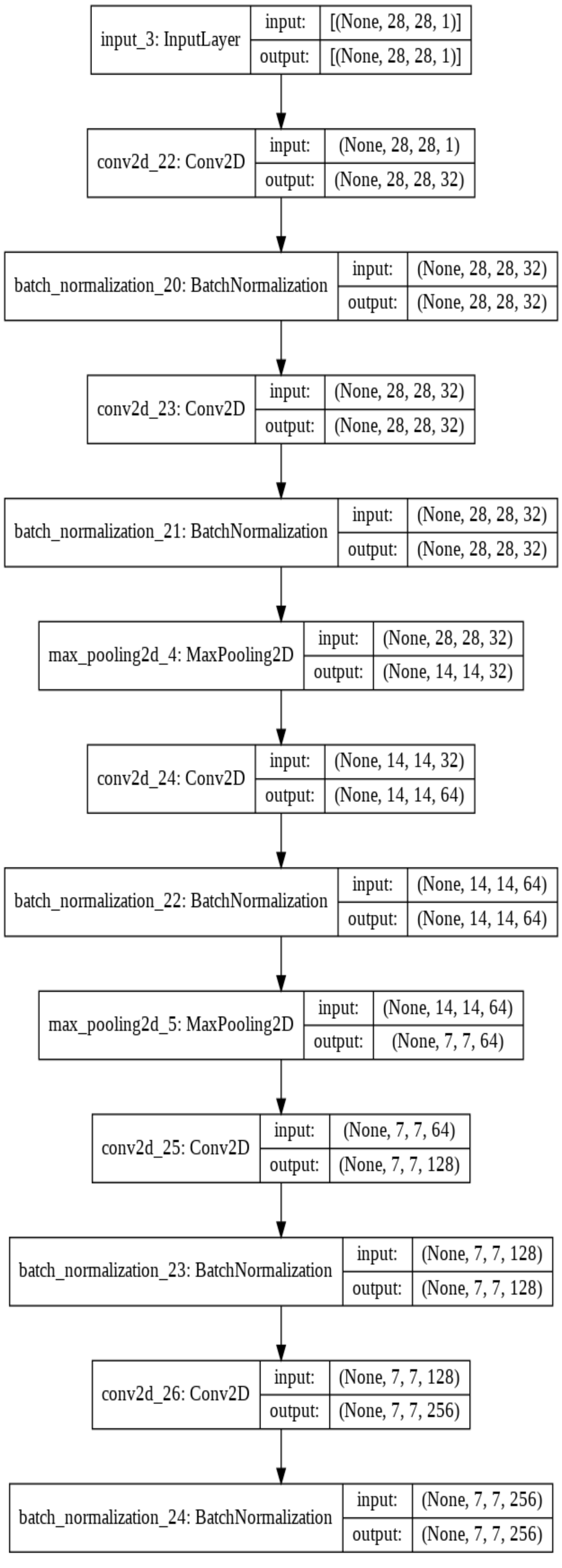

input_3: InputLayer
input: [(None, 28, 28, 1)]
output: [(None, 28, 28, 1)]
conv2d_22: Conv2D
input: (None, 28, 28, 1)
output: (None, 28, 28, 32)
batch_normalization_20: BatchNormalization
input: (None, 28, 28, 32)
output: (None, 28, 28, 32)
conv2d_23: Conv2D
input: (None, 28, 28, 32)
output: (None, 28, 28, 32)
batch_normalization_21: BatchNormalization
input: (None, 28, 28, 32)
output: (None, 28, 28, 32)
max_pooling2d_4: MaxPooling2D
input: (None, 28, 28, 32)
output: (None, 14, 14, 32)
conv2d_24: Conv2D
input: (None, 14, 14, 32)
output: (None, 14, 14, 64)
batch_normalization_22: BatchNormalization
input: (None, 14, 14, 64)
output: (None, 14, 14, 64)
max_pooling2d_5: MaxPooling2D
input: (None, 14, 14, 64)
output: (None, 7, 7, 64)
conv2d_25: Conv2D
input: (None, 7, 7, 64)
output: (None, 7, 7, 128)
batch_normalization_23: BatchNormalization
input: (None, 7, 7, 128)
output: (None, 7, 7, 128)
conv2d_26: Conv2D
input: (None, 7, 7, 128)
output: (None, 7, 7, 256)
batch_normalization_24: BatchNormalization
input: (None, 7, 7, 256)
output: (None, 7, 7, 256)

***Figure 3: Encoder layer architecture.***

### 3.1.2 Autoencoder Training

The process before and during the Autoencoder model training can be summarized in the following algorithm:

Step 1: Read all pixels from the train dataset.
Step 2: Form separate images of 28 × 28 pixels each.
Step 3: Normalize the images in range [0, 1] for better training results.
Step 4: Define the training parameters.
Step 5: Apply an 80-20 split on the train dataset and keep the smallest part for validation.
Step 6: Train the model with the defined parameters, until it converges.

We train our Autoencoder model using Keras, which is a framework built on top of TensorFlow 2.0. In other words, Keras is a neural network library while TensorFlow is the open-source library and they both help us simplify, optimize and accelerate the execution of various machine learning tasks. These libraries combined with the use of a GPU, which is available in the Google Colab platform reinforce the model reach a really fast convergence. We should point out that a machine learning model reaches convergence when it achieves a state during training, in which loss settles to within an error range around the final value. In other words, a model converges when additional training will not improve the model. Our Autoencoder model converges after almost 40 epochs. This can be verified in Figure 4.

Based on the experiments we conducted and that we display in the "Results" section later on, the best parameters for the Autoencoder training are the following:
Batch size = 64
Kernel = 3x3
Epochs = 40
Hidden layers (type, activation function, size) = {(convolution, relu, 32),
(convolution, relu, 32),
maxpool,
(convolution, relu, 64),
maxpool,
(convolution, relu, 128),
(convolution, relu, 256)}
Last activation function = sigmoid

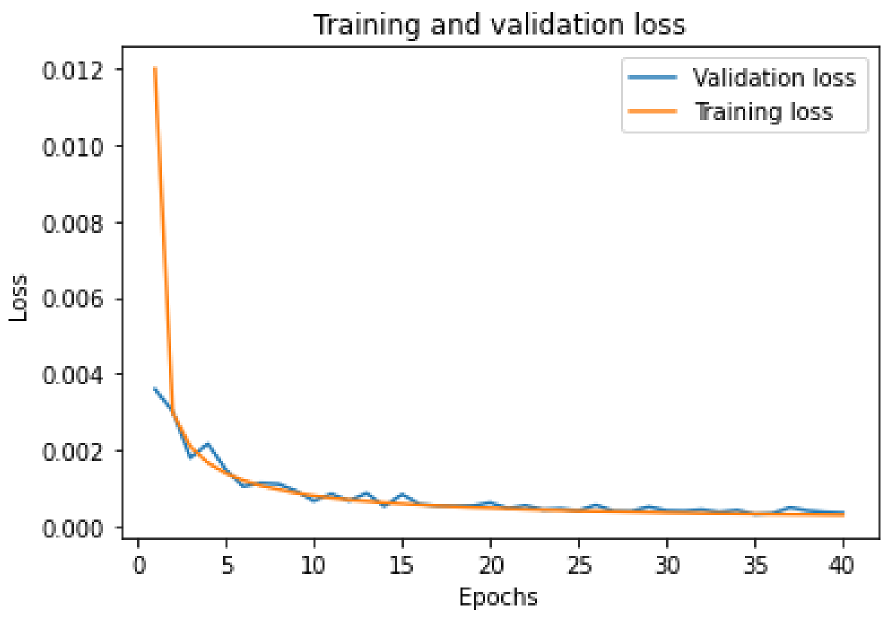


*Figure 4: Autoencoder - Loss-Epochs.*

## 3.2 Classifier

A Classifier is a neural network programmed to carry out a classification task. For the classification task of our project, we chose to implement a Multilayer Perceptron. A classic Multilayer Perceptron (MLP) is a Feed-Forward Neural Network composed of fully connected layers. It consists of three types of layers: an input layer, a hidden layer and an output layer, as shown in Figure 1. The input layer receives the input signal to be processed. The required task, either a prediction or a classification is performed by the output layer. An arbitrary number of hidden layers that are placed in between the input and output layer form the computational engine of the MLP. Like a Feed-Forward network, in a MLP the data flow in the forward direction from input to output layer. The neurons in the MLP are trained with the backpropagation learning algorithm. MLPs are designed to approximate any continuous function and they can solve problems which are not linearly separable. The major use cases of MLP are pattern classification, recognition, prediction, and approximation. The extensive layer architecture of our Multilayer Perceptron Classifier is depicted in Figure 6.

### 3.2.1 Feed-Forward Neural Network

A Feed-Forward Neural Network (FFNN) [6] is a network with no recurrent connections. This means that the connections between its nodes do not form a cycle. As such, the information moves in only one direction, forward, from the input nodes, through the hidden nodes and to the output nodes. That is why we separate Feed-Forward Neural Networks from Recurrent Neural Networks (RNNs), where recurrent connections are allowed. It is an important distinction because in a FFNN the gradient is clearly defined and computable through backpropagation, whereas in a RNN the gradient computation requires, potentially, a huge number of operations, that makes it more expensive.

### 3.2.2 Fully Connected Neural Network

A Fully Connected Neural Network [7] is a network, where the architecture is such that all the nodes, or neurons, in one layer are connected to the neurons in the next layer. The major advantage of Fully Connected networks is that they are "structure agnostic", meaning there are no special assumptions needed to be made about the input. Although, being structure agnostic makes Fully Connected networks very broadly applicable, such networks do tend to have weaker performance than special-purpose networks tuned to the structure of a problem space.

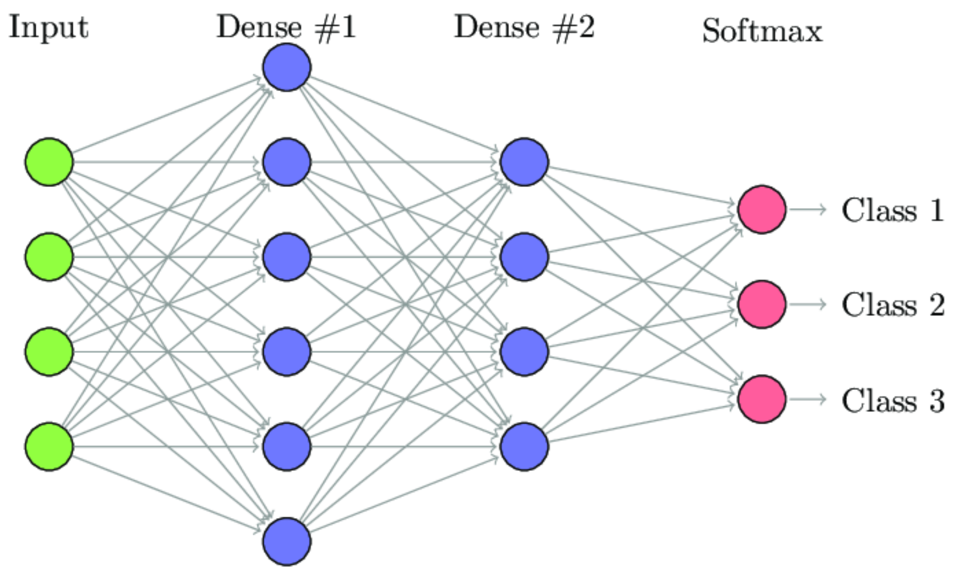


*Figure 5: Fully Connected Neural Network.*

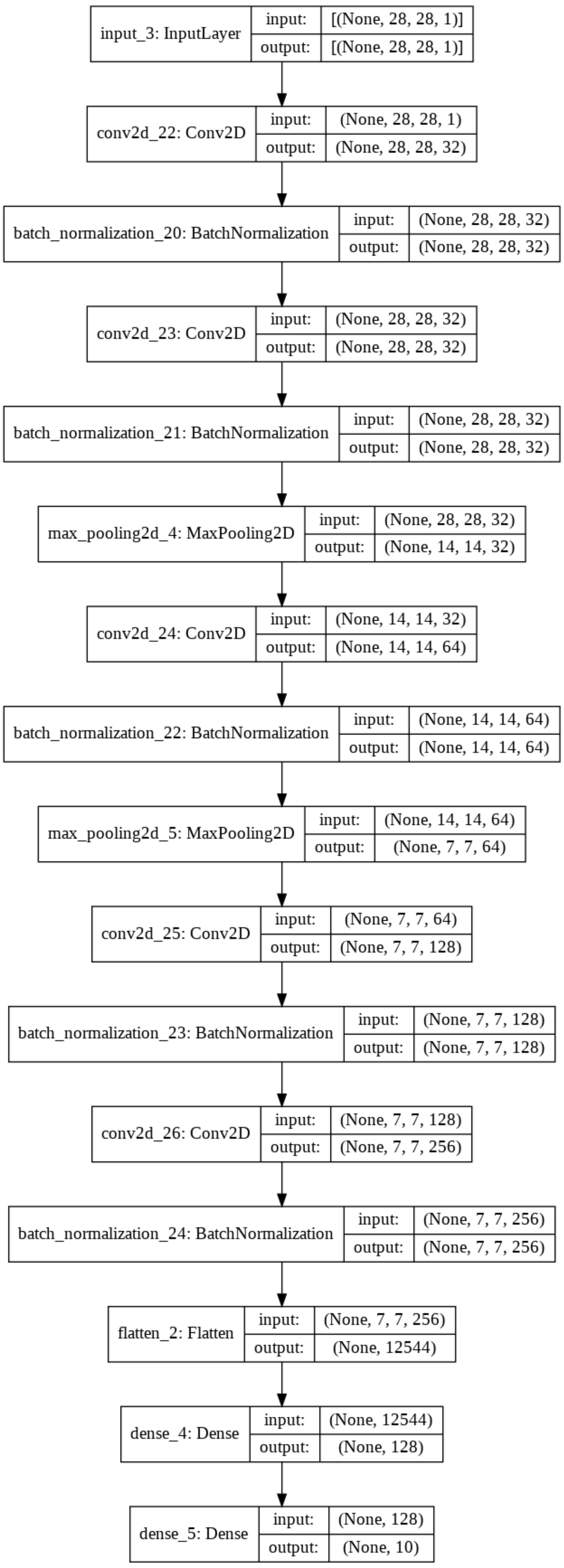

input_3: InputLayer | input: [(None, 28, 28, 1)] | output: [(None, 28, 28, 1)]
conv2d_22: Conv2D | input: (None, 28, 28, 1) | output: (None, 28, 28, 32)
batch_normalization_20: BatchNormalization | input: (None, 28, 28, 32) | output: (None, 28, 28, 32)
conv2d_23: Conv2D | input: (None, 28, 28, 32) | output: (None, 28, 28, 32)
batch_normalization_21: BatchNormalization | input: (None, 28, 28, 32) | output: (None, 28, 28, 32)
max_pooling2d_4: MaxPooling2D | input: (None, 28, 28, 32) | output: (None, 14, 14, 32)
conv2d_24: Conv2D | input: (None, 14, 14, 32) | output: (None, 14, 14, 64)
batch_normalization_22: BatchNormalization | input: (None, 14, 14, 64) | output: (None, 14, 14, 64)
max_pooling2d_5: MaxPooling2D | input: (None, 14, 14, 64) | output: (None, 7, 7, 64)
conv2d_25: Conv2D | input: (None, 7, 7, 64) | output: (None, 7, 7, 128)
batch_normalization_23: BatchNormalization | input: (None, 7, 7, 128) | output: (None, 7, 7, 128)
conv2d_26: Conv2D | input: (None, 7, 7, 128) | output: (None, 7, 7, 256)
batch_normalization_24: BatchNormalization | input: (None, 7, 7, 256) | output: (None, 7, 7, 256)
flatten_2: Flatten | input: (None, 7, 7, 256) | output: (None, 12544)
dense_4: Dense | input: (None, 12544) | output: (None, 128)
dense_5: Dense | input: (None, 128) | output: (None, 10)

***Figure 6: Classifier layer architecture.***

### 3.2.3 Classifier Training

The process before and during the Classifier model training can be summarized in the following algorithm:

Step 1: Read all pixels from the train dataset and all pixels from the test dataset.
Step 2: Form separate images of 28 × 28 pixels each., for each dataset.
Step 3: Read all labels from the train dataset and all labels from the test dataset.
Step 4: Normalize the images and the labels for better training results.
Step 5: Ask from user to give the training parameters or use the defined ones.
Step 6: Apply an 80-20 split on the train dataset and keep the smallest part for validation.
Step 7: Train the model with the defined/given parameters, without using the Encoder part of the Autoencoder.
Step 8: Train the model with the defined/given parameters, including the Encoder part of the Autoencoder.

Just as we trained our Autoencoder model using Keras and TensorFlow libraries, so too we use them to train our Classifier model. We have already pointed out during the Autoencoder training, that these libraries helped us achieve a fast convergence and a low loss value. This time, we deal with a classification problem. As a result, we don't only seek for a fast convergence but also for a high accuracy, in order to succeed in the classification task. It is now clear, that we are facing a more demanding task. So as to cope with it we have to train our model twice and not once, as we did with our previous task. The first time, we only train the Fully Connected part of the Classifier for 100 epochs, skipping the Encoder part taken from the Autoencoder. The reason why we do this is to help the Classifier learn gradually the information and optimize its results after the second training. The second time, we train the whole Classifier including the Encoder part of the Autoencoder. Indeed, thanks to the first training the Classifier only needs 21 epochs to converge. After all, our model needs 121 epochs in total to converge. Concerning the accuracy results, the training accuracy reaches 100%, meaning that the Classifier can predict the digits of the training dataset images with absolute precision. The validation accuracy ranges between 98-99%. The above results can be verified in Figure 7 and Figure 8.

Based on the experiments we conducted and that we display in the "Results" section later on, the best parameters for the Classifier training are the following:
Batch size = 256
Epochs = 121 (100 + 21)
Fully Connected Layer size = 128
Fully Connected Layer activation function = relu
Last activation function = softmax

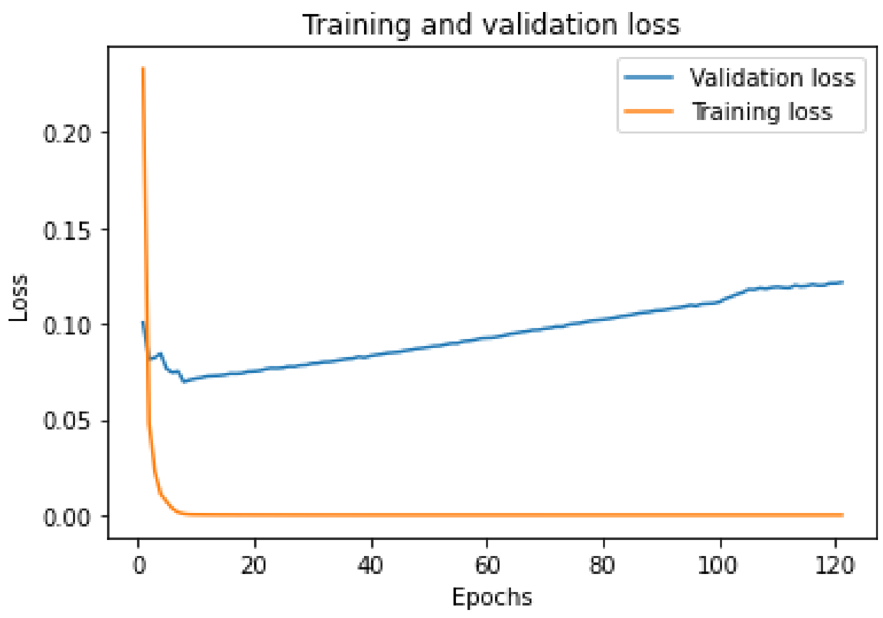


*Figure 7: Classifier - Loss-Epochs.*

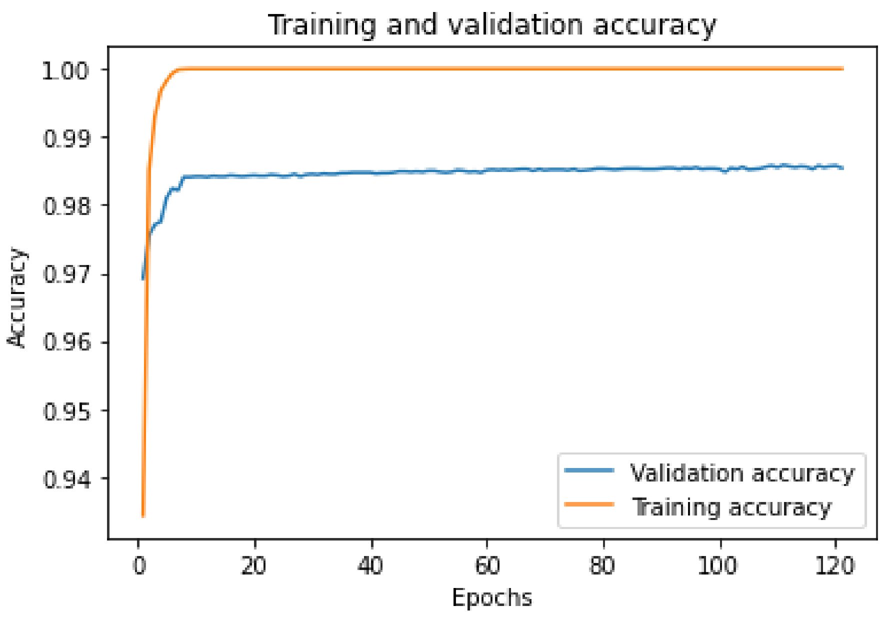


*Figure 8: Classifier - Accuracy-Epochs.*

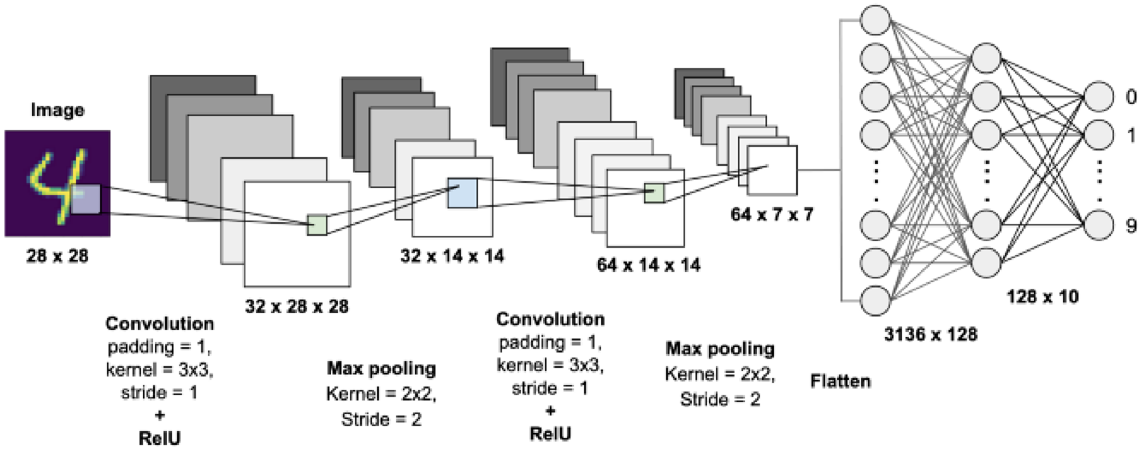


*Figure 9: Total Network Architecture.*

## 4. Results

### 4.1 Summary Figures

- **Loss – Fully Connect Layer Size Figure**

  Batch size = 256
  Epochs = 121 (100 + 21)
  Fully Connected Layer size = n
  Fully Connected Layer activation function = relu
  Last activation function = softmax

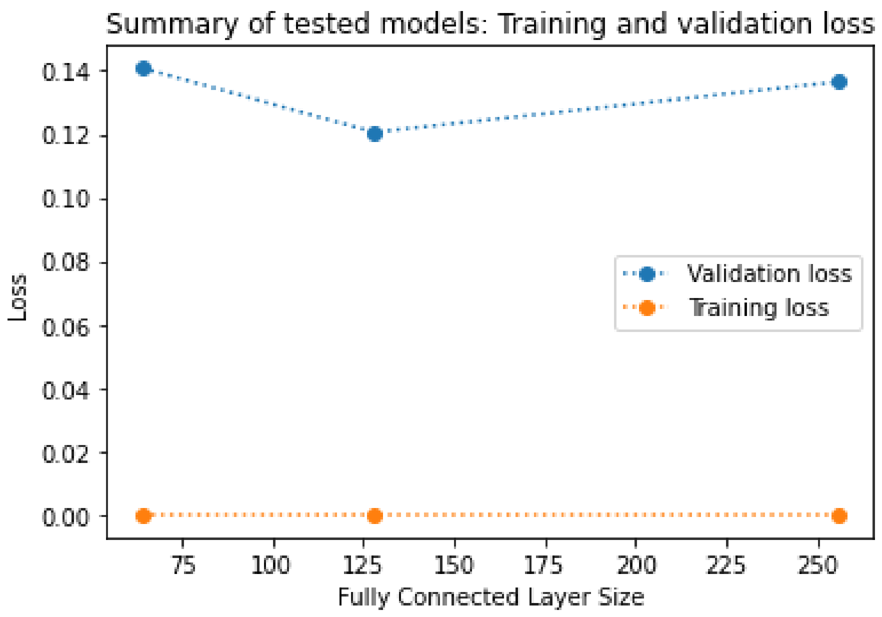


*Figure 10 - Loss-Fully Connected Layer Size.*

- **Accuracy – Fully Connect Layer Size Figure**

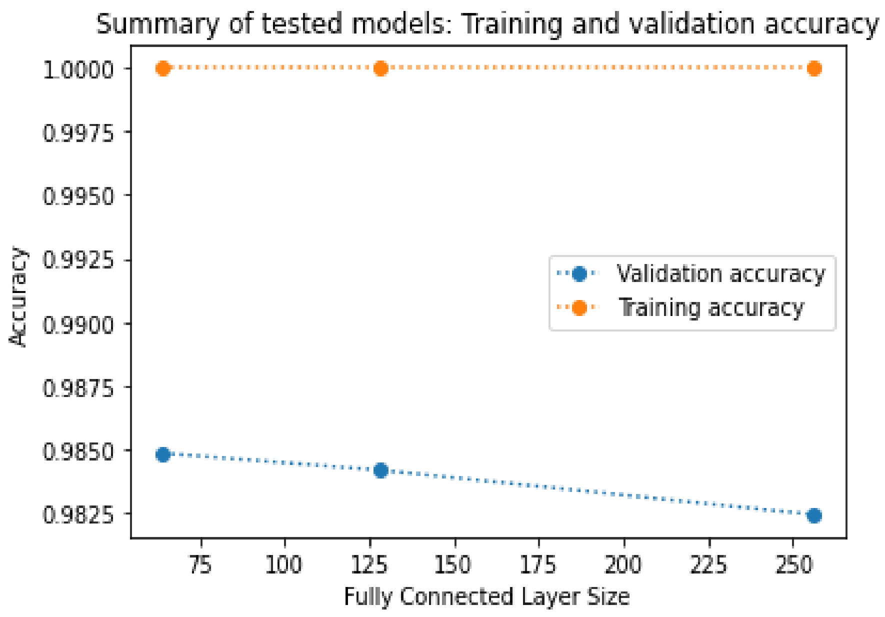


***Figure 11: Accuracy-Fully Connected Layer Size.***

In the Loss – Fully Connected Layer Size Figure we observe that we get the smallest validation loss value when the size of the fully connected layer is 128. Respectively, in the Accuracy – Fully Connected Layer Size Figure, validation accuracy gets its maxiumum values when the size of the fully connected layer is 64 or 128.

Examining the training loss and training accuracy, we can't draw any conclusions, because we get almost perfect results for every size we experimented with. Therefore, the validation loss and validation accuracy are our main leads that help us choose the value 128 as the default size of the fully connected layer.

- **Loss – Epochs Figure**

  Batch size = 256
  Epochs = 2n (n + n)
  Fully Connected Layer size = 128
  Fully Connected Layer activation function = relu
  Last activation function = softmax

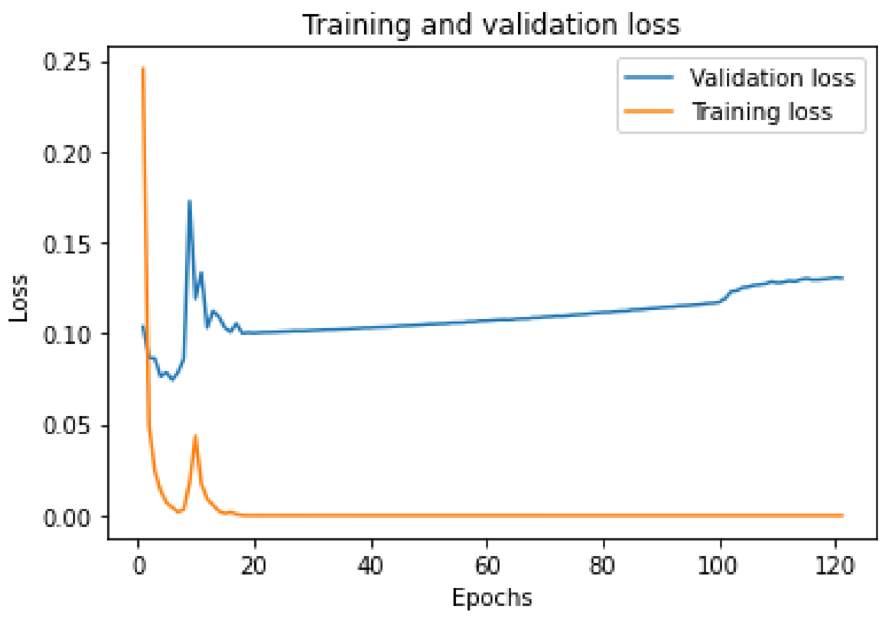


***Figure 12: Loss-Epochs.***

- **Accuracy – Epochs Figure**

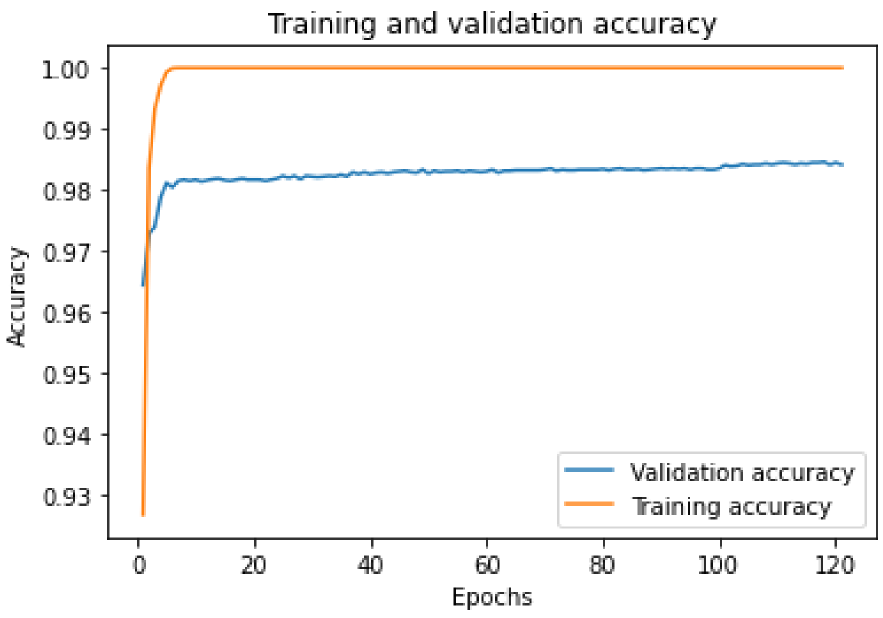


***Figure 13: Accuracy-Epochs.***

During the execution of our experiments we noticed that as the number of epochs used in a model's training increases, so does the total execution time of the training procedure.

In the Loss - Epochs Figure we observe that the training loss value gets its minimum value when we train our model with a small number of epochs. However, we don't get immediately clear results concerning the minimum value of the validation loss. That's why we gradually train our model adding more epochs experiment after experiment. By choosing to train our model with 200 epochs (100 epochs for every fit method) it becomes obvious that the early stopping technique is applied in the 21º epoch of the second fit method. As a result, we conclude that this is the ideal case, in order to achieve the best training outcome for our model.

The use of a satisfactory number of epochs has a significant impact on the prediction on the test set, because we observe that if we increase the number of epochs, then the test accuracy increases as well, to an extent and at the same time the number of incorrectly predicted images decreases.

- **Loss – Batch Size Figure**

  Batch size = n
  Epochs = 121 (100 + 21)
  Fully Connected Layer size = 128
  Fully Connected Layer activation function = relu
  Last activation function = softmax

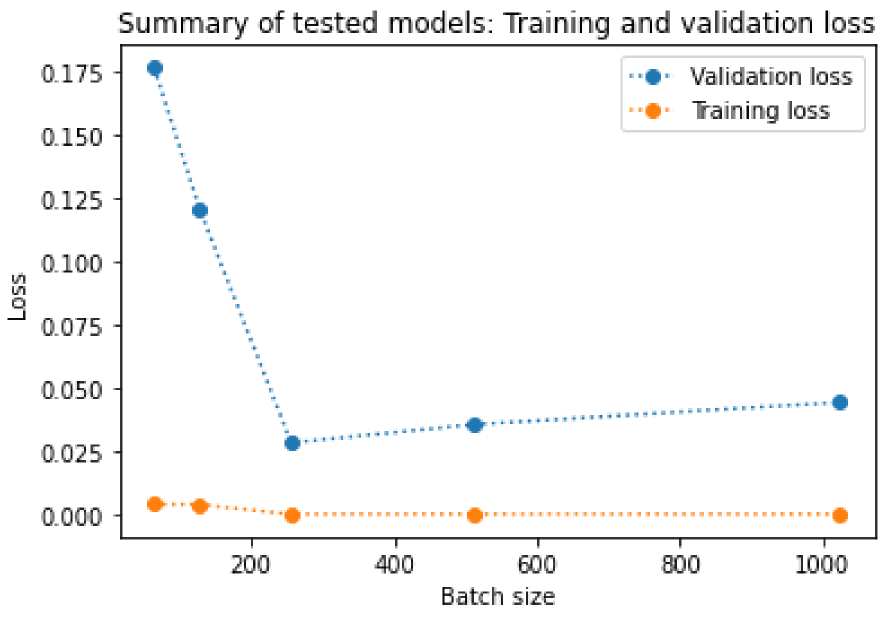


***Figure 14: Loss-Batch size.***

- **Accuracy – Batch Size Figure**

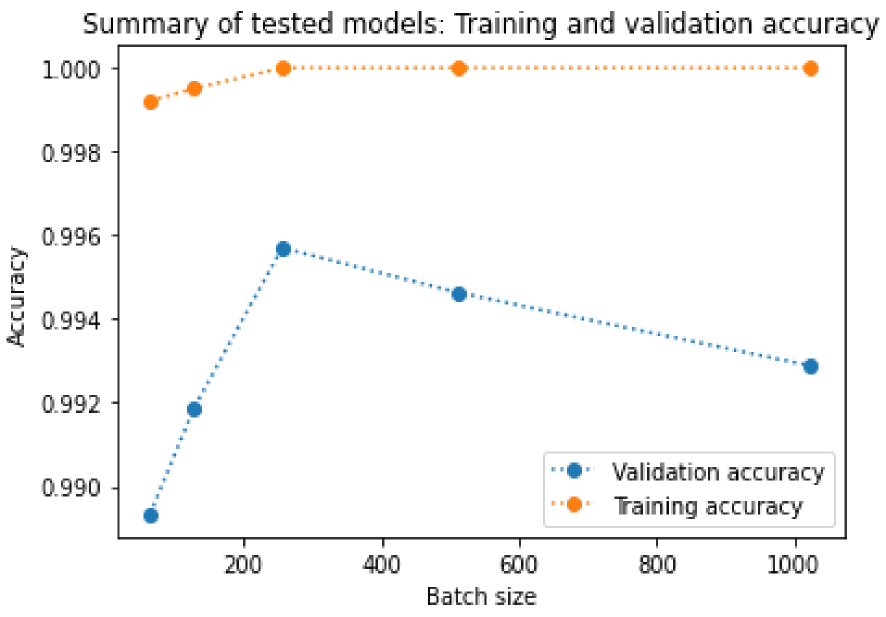


*Figure 15: Accuracy-Batch size.*

During the execution of our experiments we noticed that as the batch size used in a model's training increases, the execution time of each training epoch is decreased.

In the Loss – Batch Size Figure we observe that as the batch size used in a model's training increases, so does the training loss value. However, the training loss gets immediately a very small value and the optimization is insignificant as much as we increase the number of epochs. As far as the validation loss is concerned, it is evident from Figure 14 that it gets its minimum value for a batch size equal to 256. We can draw a similar conclusion for the maximum accuracy value if we observe Figure 15. For all the above reasons, we conclude that the best batch size is equal to 256.

## 4.2 Prediction on Test Set

*Table 1: Prediction Results.*

| Test Loss | Test Accuracy |
|---|---|
| 0.0903 | 98.67% |

## 4.3 Classification Results

Found 9867 correct labels.
Found 133 incorrect labels.

*Table 2: Classification Report.*

| | Precision | Recall | F1-Score | Support |
|---|---|---|---|---|
| Class 0 | 0.98 | 0.99 | 0.99 | 980 |
| Class 1 | 1.00 | 1.00 | 1.00 | 1135 |
| Class 2 | 0.99 | 0.98 | 0.98 | 1032 |
| Class 3 | 0.98 | 0.99 | 0.99 | 1010 |
| Class 4 | 0.98 | 0.99 | 0.99 | 982 |
| Class 5 | 0.99 | 0.99 | 0.99 | 892 |
| Class 6 | 0.99 | 0.99 | 0.99 | 958 |
| Class 7 | 0.98 | 0.98 | 0.98 | 1028 |
| Class 8 | 0.98 | 0.98 | 0.98 | 974 |
| Class 9 | 0.98 | 0.98 | 0.98 | 1009 |
| | | | | |
| Accuracy | | | 0.99 | 10000 |
| Macro avg | 0.99 | 0.99 | 0.99 | 10000 |
| Weighted Avg | 0.99 | 0.99 | 0.99 | 10000 |

## 4.4 Correctly Predicted Images

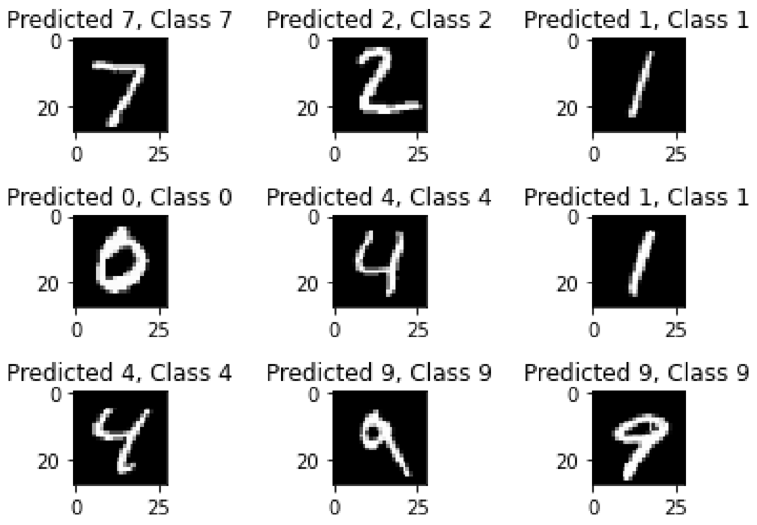


*Figure 16: The first nine images our model predicted correctly.*

## 4.5 Incorrectly Predicted Images

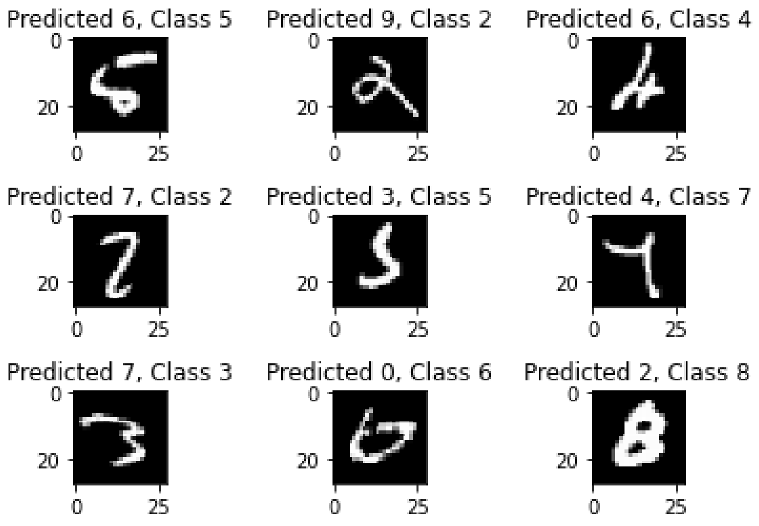


*Figure 17: The first nine images our model predicted incorrectly.*

## 5. Conclusion

To sum up, the MNIST database provides us a relatively simple classification task that helps us explore machine learning and pattern recognition techniques, saving unnecessary efforts on data preprocessing and formatting. Practically, carrying through with the handwritten digit classification problem is the first step, in order to delve into the more complicated tasks of image classification. As we have already analyzed, image classification is perhaps the most important part of digital image processing. Therefore, by reaching a point where we can accurately automate image classification tasks, that involve enormous numbers of images, we make a huge leap forward in improving various fields and applications. The advancements in the field of autonomous driving serve as a great example of the use of image classification in the real-world. Some other applications include automated image organization, stock photography and video websites, visual search for improved product discoverability, image, and face recognition on social networks, and many more.

## 6. References

[1] https://en.wikipedia.org/wiki/Moravec%27s_paradox (accessed 18/04/21)

[2] Schmidhuber, J. (2015). Deep Learning in Neural Networks: An Overview. Neural Networks, 61: 85–117. doi:10.1016/j.neunet.2014.09.003

[3] http://yann.lecun.com/exdb/mnist/ (accessed 18/04/21)

[4] https://towardsdatascience.com/auto-encoder-what-is-it-and-what-is-it-used-for-part-1-3e5c6f017726 (accessed 20/04/21)

[5] Salakhutdinov, R.; H. Geoffrey (2009). Semantic hashing. International Journal of Approximate Reasoning (Special Section on Graphical Models and Information Retrieval), 50 (7): 969–978. doi:10.1016/j.ijar.2008.11.006

[6] https://stackoverflow.com/a/45934649 (accessed 22/04/21)

[7] S.H. Shabbeer Basha, Shiv Ram Dubey, V. Pulabaigari, S. Mukherjee (2019). Impact of Fully Connected Layers on Performance of Convolutional Neural Networks for Image Classification, Neurocomputing 378: 112-119. doi:10.1016/j.neucom.2019.10.008