**ÉCOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE ET DE MATHÉMATIQUES APPLIQUÉES**

# Recognizing Digits using Neural Networks

GROUP 15

JOÃO VITOR VARGAS SOARES
NICOLAU PEREIRA ALFF


Academic Year 2020 – 2021

# 1 Introduction

In this paper, the main goal was to use the MNIST set and neural networks to build models that recognize handwritten digits. In order to do that, two types of networks were chosen: fully connected multi-layer network and multi-layer Convolutional network. The development and the results of each of the networks are described in the next topics of this document.

# 2 Fully connected multi-layer network

The implementation of the fully connected multi-layer network was made in five parts that will be explained in the following topics.

## 2.1 Data preparation

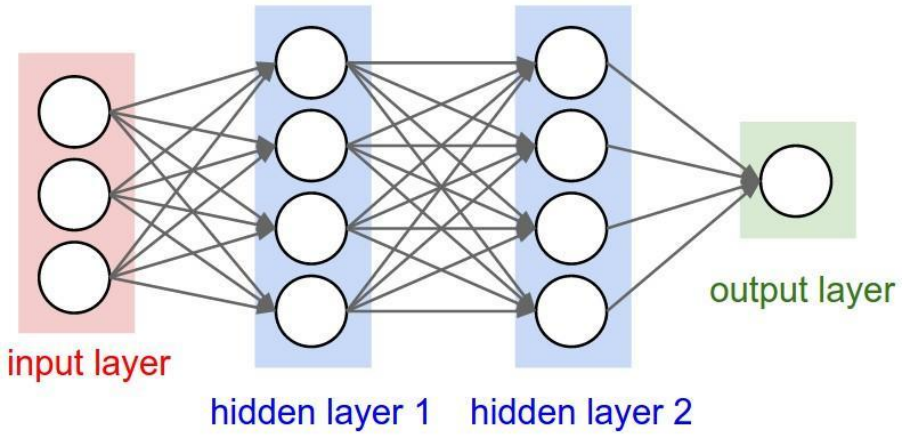
The first thing to do was to upload the MNIST dataset into the program. The dataset was divided between test and train, but since it was asked for us to divide the dataset into train, test and validation, the two given sets were joined and redivided twice with the function train_test_split, and the final percentage for the sets were: 80% for training, 10% for validation and 10% for testing. We also normalized the RGB coordinates by dividing them by 255 (0 is black and 255 is white) and reshaped the data into a two-dimensional array. All the images had 784 pixels, so the array dimension before dividing the sets was (70000, 784). The last thing that was done in this function was to change the value of the classes into binary vector, which means that if the number was 7, we converted it to a bit vector with 1 in the 8$^{th}$ position and 0 in the others, for example.

```
1.  def prepareData():
2.      # Upload dataset
3.      (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
4.
5.      num_classes = 10 # 10 numbers, from 0 to 9
6.
7.      # Concatenate training and testing sets to redivide later
8.      x = np.array(np.concatenate((x_train, x_test)))
9.      y = np.array(np.concatenate((y_train, y_test)))
10.
11.     # Scale images to the [0, 1] range
12.     x = x.astype("float32") / 255
13.
14.     # Reshaping x from (70000, 28, 28) to (70000, 784)
15.     x = x.reshape(x.shape[0], 784)
16.
```

```
    # Dividing total set into test set (10%), training set (80%) and validation set (10%)
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.1 , random_state=42)
    x_train, x_valid, y_train, y_valid = train_test_split(x_train, y_train, test_size=1/9 , random_state=42)

    # Saving class vector for further use
    y_test_classes = y_test

    # convert class vectors to binary class matrices
    y_train = keras.utils.to_categorical(y_train, num_classes)
    y_valid = keras.utils.to_categorical(y_valid, num_classes)
    y_test = keras.utils.to_categorical(y_test, num_classes)

    return x_test, x_train, x_valid, y_test, y_train, y_valid, y_test_classes

```

2.2 Implementing the fully connected multi-layer network model

The next function was the one used to implement the fully connected multi-layer network. To do that, we initialized the model and then added its layers. The input and the hidden layers were given as parameters of the function. The input_shape of the input layer is the number of pixels of each image, the units is the number of neurons of each layer and the activation parameter is the activation function, which we chose Rectified Linear Unit (ReLU) for the input and hidden layers and Softmax for the output layer. For the output layer, the number of neurons chosen was the number of classification classes, which were the numbers from 0 to 9 (ten classes).




The activation function defines the output of a node given an input or set of inputs. The ones used in this project are described in the topics below:

→ The Rectified Linear Unit (ReLU) is the most commonly used activation function in deep learning. The function returns 0 if the input is negative, but for any positive input, it returns that value back;

→ Softmax is an activation function that scales numbers/logits into probabilities. The output of a Softmax is a vector with probabilities of each possible outcome. The probabilities in this vector sums to one for all possible outcomes or classes and it is commonly used as the output layer activation function.

Another parameter added to the model was the dropout which refers to ignoring units during the training phase of a certain set of neurons which is chosen at random, i.e., the units are not considered during a particular forward or backward pass, and it is used to prevent overfitting.

After setting the model, we compiled it using an optimization function that will be explained further in this document, and monitoring the loss (between the labels and predictions) and accuracy from both validation and train sets.

Before fitting our model, we created a function using the EarlyStopping class so that we can monitor the accuracy of the model and stop when it diverges. To do that, we have set the mode of the function to 'max' and the patience to 8, which means that if the function stops increasing after 8 cycles of epochs, we'll stop fitting the model.

In the fitting, we added our train and validation set, our callback, which is the EarlyStopping function, the verbose (used to print the method logs), the batch size (the number of training samples in one forward/backward pass. The higher the batch size, the more memory space is needed) and the number of epochs (the number chosen is high because it will stop if the condition in the callback attribute is achieved).

```
31. def fullyConnectedMultiLayerNetwork(layers, x_train, y_train, x_valid,
    y_valid, optimizer):
32.     # Initializing model
33.     model = Sequential()
34.
35.     #Adding input layer with first_units as the number of neurons, input shape
    (784,) and activation
36.     model.add(Dense(units=layers[0]["units"], input_shape=(x_train.shape[1],),
    activation=layers[0]["activation"]))
37.
38.     # Using dropout to avoid overfitting
39.     model.add(Dropout(layers[0]["dropout"]))
40.
41.     # Adding hidden layers
42.     for layer in layers[1:-1]:
43.         model.add(Dense(units=layer["units"], activation =
    layer["activation"]))
44.         model.add(Dropout(layer["dropout"]))
45.
46.     # Adding output layer
47.     model.add(Dense(10))
```

```
    model.add(Activation('softmax'))

    # Compiling model
    model.compile(loss='categorical_crossentropy', metrics=['accuracy'],
optimizer=optimizer)

    # Early stop when accuracy diverges
    es = EarlyStopping(monitor='val_accuracy',
                       patience=8,
                       min_delta=0.001,
                       mode='max')

    # model fit with maximum of 500 epochs
    history = model.fit(x_train, y_train,
            batch_size=128, epochs=500,
            verbose=2,
            validation_data=(x_valid, y_valid),
            callbacks=es)
    return model, history
```

2.3 Prediction and metrics

After creating our model, the predictions using the test set were made using the function model.predict. We divided it into the probabilities of the prediction (y_prediction_probs) and the class prediction (y_prediction_classes). Next, we calculated the metrics for the actual model using the functions provided by the library sklearn.metrics, including accuracy, precision, recall, f1 score and AUC for ROC curves.

```
# This function makes the predictions and metrics for a determined model
def predictionsAndMetrics(model, x_test, y_test_classes):

    # predict probabilities for test set
    y_prediction_probs = model.predict(x_test, verbose=0)
    # predict crisp classes for test set
    y_prediction_classes = np.argmax(model.predict(x_test), axis=-1)

    # accuracy: (tp + tn) / (p + n)
    accuracy = accuracy_score(y_test_classes, y_prediction_classes)
    print('Accuracy: %f' % accuracy)
    # precision tp / (tp + fp)
    precision = precision_score(y_test_classes, y_prediction_classes,
average='macro')
    print('Precision: %f' % precision)
    # recall: tp / (tp + fn)
    recall = recall_score(y_test_classes, y_prediction_classes,
average='macro')
    print('Recall: %f' % recall)
    # f1: 2 tp / (2 tp + fp + fn)
    f1 = f1_score(y_test_classes, y_prediction_classes, average='macro')
    print('F1 score: %f' % f1)
    # ROC AUC
    auc = roc_auc_score(y_test_classes, y_prediction_probs, multi_class='ovr')
    print('ROC AUC: %f' % auc)
    return accuracy, precision, recall, f1, auc

```

2.4 Variation of model parameters

In order to compare the importance of each parameter in the model, we varied them, including the number of layers of the network, the number of

neurons in each layer of the network, the learning rate of the chosen optimization function, the types of optimizers, and the percentage of dropout.

To vary the number of layers, we started with a three-layer model, including the input layer, one hidden layer and the output layer. After each iteration, we added one more hidden layer and saved its metrics.

The varying units function was used to vary the number of neurons of each layer, using as a base a three-layer model defined in the main function. The numbers of neurons were increased after each iteration, from 32 to 512 units.

In the varyingLearningRate function, we used as a base the optimizer RMSprop and varied its learning rate from 1 to 0.0001. The learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated. The RMSprop function will be explained in the next paragraph.

Since there are a lot of types of optimizers that can be used in the model compilation, we chose five to do a comparison. The ones chosen were the following:

→ RMSprop: Root Mean Square Propagation optimization function maintains a moving average of the square of gradients and divides the gradient by the root of this average;

→ Adam: Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments. It is computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters;

→ SGD: Stochastic Gradient Descent is a simple yet very efficient approach to fitting linear classifiers and regressors under convex loss functions;

→ Adadelta: Adadelta optimization is a stochastic gradient descent method that is based on adaptive learning rate per dimension to address two drawbacks: the continual decay of learning rates throughout training and the need for a manually selected global learning rate;

→ Adamax: It is a variant of Adam based on the infinity norm. Adamax is sometimes superior to Adam, especially in models with embeddings.

The last parameter that was varied for comparisons was the percentage of dropout of each layer, which was varied from 10% to 50%.

```
# This function varies the number of hidden layers from 1 to 5 for comparison
def varyingLayers(x_test, x_train, x_valid, y_test, y_train, y_valid, y_test_classes, layers, optimizer):
    accuracy = np.zeros(5)
    precision = np.zeros(5)
    recall = np.zeros(5)
    f1 = np.zeros(5)
    auc = np.zeros(5)
    for i in range(5):
        model, history = fullyConnectedMultiLayerNetwork(layers, x_train, y_train, x_valid, y_valid, optimizer)
        accuracy[i], precision[i], recall[i], f1[i], auc[i] = predictionsAndMetrics(model, x_test, y_test_classes)
        layers = np.append(layers, [{"units": 128, "activation": 'relu', "dropout": 0.2}])
    metrics = {"accuracy": accuracy, "precision": precision, "recall": recall, "f1": f1, "auc": auc}
    return metrics

 # This function varies the number of units from 32 to 512 for comparison
def varyingUnits(x_test, x_train, x_valid, y_test, y_train, y_valid, y_test_classes, layers, optimizer):
    accuracy = np.zeros(5)
    precision = np.zeros(5)
    recall = np.zeros(5)
    f1 = np.zeros(5)
    auc = np.zeros(5)
    for i in range(5):
        model, history = fullyConnectedMultiLayerNetwork(layers, x_train, y_train, x_valid, y_valid, optimizer)
        accuracy[i], precision[i], recall[i], f1[i], auc[i] = predictionsAndMetrics(model, x_test, y_test_classes)
        layers[0]["units"] = layers[0]["units"] * 2
        layers[1]["units"] = layers[1]["units"] * 2
    metrics = {"accuracy": accuracy, "precision": precision, "recall": recall, "f1": f1, "auc": auc}
    return metrics

# This function varies the learning rate in the optimizer function Adam from 1 to 0.00001 for comparison
def varyingLearningRate(x_test, x_train, x_valid, y_test, y_train, y_valid, y_test_classes, layers, optimizer):
    accuracy = np.zeros(5)
    precision = np.zeros(5)
    recall = np.zeros(5)
    f1 = np.zeros(5)
    auc = np.zeros(5)
    lr = 1
    for i in range(5):
        optimizer = keras.optimizers.RMSprop(learning_rate=lr)
        model, history = fullyConnectedMultiLayerNetwork(layers, x_train, y_train, x_valid, y_valid, optimizer)
        accuracy[i], precision[i], recall[i], f1[i], auc[i] = predictionsAndMetrics(model, x_test, y_test_classes)
        lr = lr/10
    metrics = {"accuracy": accuracy, "precision": precision, "recall": recall, "f1": f1, "auc": auc}
    return metrics

# This function varies the type of optimizer used in the model compilation for comparison
def varyingOptimizer(x_test, x_train, x_valid, y_test, y_train, y_valid, y_test_classes, layers):
    accuracy = np.zeros(5)
```

```
        precision = np.zeros(5)
        recall = np.zeros(5)
        f1 = np.zeros(5)
        auc = np.zeros(5)
        lr = 1
        optimizers = np.array(['rmsprop', 'adam', 'sgd', 'adadelta',
    'adamax'])
        for i in range(5):
            model, history = fullyConnectedMultiLayerNetwork(layers, x_train,
    y_train, x_valid, y_valid, optimizers[i])
            accuracy[i], precision[i], recall[i], f1[i], auc[i] =
    predictionsAndMetrics(model, x_test, y_test_classes)
            lr = lr/10
        metrics = {"accuracy": accuracy, "precision": precision, "recall":
    recall, "f1": f1, "auc": auc}
        return metrics

    # This function varies the percentage of dropout from 0.1 to 0.5 for
    comparison
    def varyingDropout(x_test, x_train, x_valid, y_test, y_train, y_valid,
    y_test_classes, layers, optimizer):
        accuracy = np.zeros(5)
        precision = np.zeros(5)
        recall = np.zeros(5)
        f1 = np.zeros(5)
        auc = np.zeros(5)
        dropout = 0.1
        for i in range(5):
            layers[0]["dropout"] = dropout
            layers[1]["dropout"] = dropout
            model, history = fullyConnectedMultiLayerNetwork(layers, x_train,
    y_train, x_valid, y_valid, optimizer)
            accuracy[i], precision[i], recall[i], f1[i], auc[i] =
    predictionsAndMetrics(model, x_test, y_test_classes)
            dropout += 0.1
        metrics = {"accuracy": accuracy, "precision": precision, "recall":
    recall, "f1": f1, "auc": auc}
        return metrics

```

## 2.5 Main function

In the main function, the base model was defined, which was a three-layer model with 128 neurons in the first two layers and 10 in the last one, activation function ReLU, dropout of 20%, RMSprop optimization function with 0.001 as the learning rate.

```
    x_test, x_train, x_valid, y_test, y_train, y_valid, y_test_classes =
    prepareData()
    layers = np.array([
                        {"units": 128, "activation": 'relu', "dropout": 0.2},
                        {"units": 128, "activation": 'relu', "dropout": 0.2},
                        ])
    optimizer = keras.optimizers.RMSprop(learning_rate=0.001)

    metrics_layers = varyingLayers(x_test, x_train, x_valid, y_test, y_train,
    y_valid, y_test_classes, layers, optimizer)
    metrics_units = varyingUnits(x_test, x_train, x_valid, y_test, y_train,
    y_valid, y_test_classes, layers, optimizer)
    metrics_learningRate = varyingLearningRate(x_test, x_train, x_valid,
    y_test, y_train, y_valid, y_test_classes, layers, optimizer)
    metrics_optimizer = varyingOptimizer(x_test, x_train, x_valid, y_test,
    y_train, y_valid, y_test_classes, layers)
    metrics_dropout = varyingDropout(x_test, x_train, x_valid, y_test,
    y_train, y_valid, y_test_classes, layers, optimizer)

    print(metrics_layers)
    print(metrics_units)
    print(metrics_learningRate)
    print(metrics_optimizer)
```

```
print(metrics_dropout)
```

## 3 Results for the fully connected multi-layer network

After computing the results, it was possible to analyze which of the different parameters makes the better performance for the fully connected multi-layer network. In the following tables, the comparison of the outputs varying the chosen parameters.

| Dropout | Accuracy | Precision | Recall | F1 Score | AUC |
|---|---|---|---|---|---|
| 0.1 | 0.98085714 | 0.98071844 | 0.98097437 | 0.9807922 | 0.99961915 |
| 0.2 | 0.98014286 | 0.98017712 | 0.98028013 | 0.98017551 | 0.99961565 |
| 0.3 | 0.98042857 | 0.98036316 | 0.98053219 | 0.98039725 | 0.9997069 |
| 0.4 | 0.98242857 | 0.98217885 | 0.98256779 | 0.98234521 | 0.9996366 |
| 0.5 | 0.98171429 | 0.98154649 | 0.98172645 | 0.98162166 | 0.99955359 |

*Table 1: Metrics for dropout variation*

| #Layers | Accuracy | Precision | Recall | F1 Score | AUC |
|---|---|---|---|---|---|
| 3.0 | 0.97714286 | 0.97714492 | 0.97710102 | 0.97709119 | 0.99941992 |
| 4.0 | 0.97771429 | 0.97769637 | 0.97773986 | 0.97765557 | 0.99941324 |
| 5.0 | 0.97785714 | 0.9777514 | 0.97796677 | 0.97783423 | 0.99922791 |
| 6.0 | 0.97785714 | 0.97776834 | 0.97771733 | 0.97771705 | 0.99921848 |
| 7.0 | 0.97685714 | 0.97666715 | 0.97670108 | 0.97666848 | 0.99907801 |

*Table 2: Metrics for number of layers variation*

| Learning Rate | Accuracy | Precision | Recall | F1 Score | AUC |
|---|---|---|---|---|---|
| 1.0 | 0.454 | 0.71370207 | 0.44217129 | 0.47487017 | 0.78888396 |
| 0.1 | 0.88842857 | 0.92865724 | 0.88948368 | 0.89545517 | 0.98588866 |
| 0.01 | 0.976 | 0.97581947 | 0.97600039 | 0.975877 | 0.99914585 |
| 0.001 | 0.98028571 | 0.98011692 | 0.9803134 | 0.98018134 | 0.99958006 |
| 0.0001 | 0.97985714 | 0.97974605 | 0.97984632 | 0.97978106 | 0.99966164 |

*Table 3: Metrics for learning rate variation in the RMSprop optimizer*

| Optimizer | Accuracy | Precision | Recall | F1 Score | AUC |
|---|---|---|---|---|---|
| RMSprop | 0.98128571 | 0.98107462 | 0.9813375 | 0.98118632 | 0.99961775 |
| Adam | 0.98185714 | 0.98144191 | 0.98207278 | 0.98166093 | 0.99972027 |
| SGD | 0.96514286 | 0.96517442 | 0.9650107 | 0.96505965 | 0.99885425 |
| Adadelta | 0.91528571 | 0.91486565 | 0.91455433 | 0.91453688 | 0.99318023 |
| Adamax | 0.97885714 | 0.97862221 | 0.97891256 | 0.97874168 | 0.99971716 |

*Table 4: Metrics for optimizer variation*

| #Neurons | Accuracy | Precision | Recall | F1 Score | AUC |
|---|---|---|---|---|---|
| 32.0 | 0.95914286 | 0.95920847 | 0.95909429 | 0.95911098 | 0.99822763 |
| 64.0 | 0.97285714 | 0.97273743 | 0.97276611 | 0.97272533 | 0.99904859 |
| 128.0 | 0.97885714 | 0.97863842 | 0.97893691 | 0.9787558 | 0.99942827 |
| 256.0 | 0.97971429 | 0.97951777 | 0.9796957 | 0.97959133 | 0.99959793 |
| 512.0 | 0.98 | 0.97980602 | 0.97995066 | 0.97984981 | 0.99967485 |

*Table 5: Metrics for number of neurons variation*

In table 1, it is possible to see that the best percentage for the dropout value is between 30% and 40%, but we can also see that this parameter doesn't have so much influence in the metrics, only in the prevention of overfitting the model.

Analyzing table 2, it is possible to conclude that the optimal number of layers is between 5 and 6 layers in terms of the metrics, but since when increasing the number of layers, you also increase the computational cost, a good number of layers for this model would be between 3 to 4 layers, because as it is shown in the table, the metrics don't change that much by increasing the number of layers.

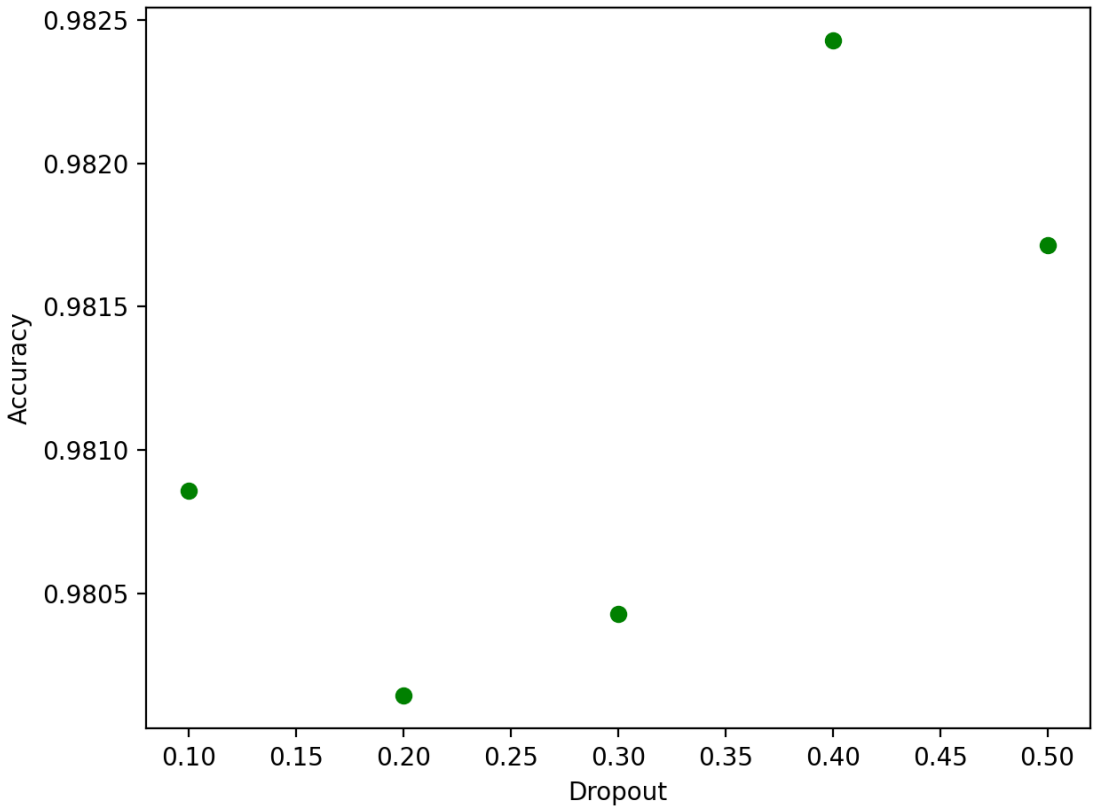
Table 3 shows us that the learning rate can drastically change the metrics for the model, and a good value for it would be near 0.001.

The analysis of the type of the optimizer in table 4 shows us that the one that has the best metrics is the Adam optimizer.

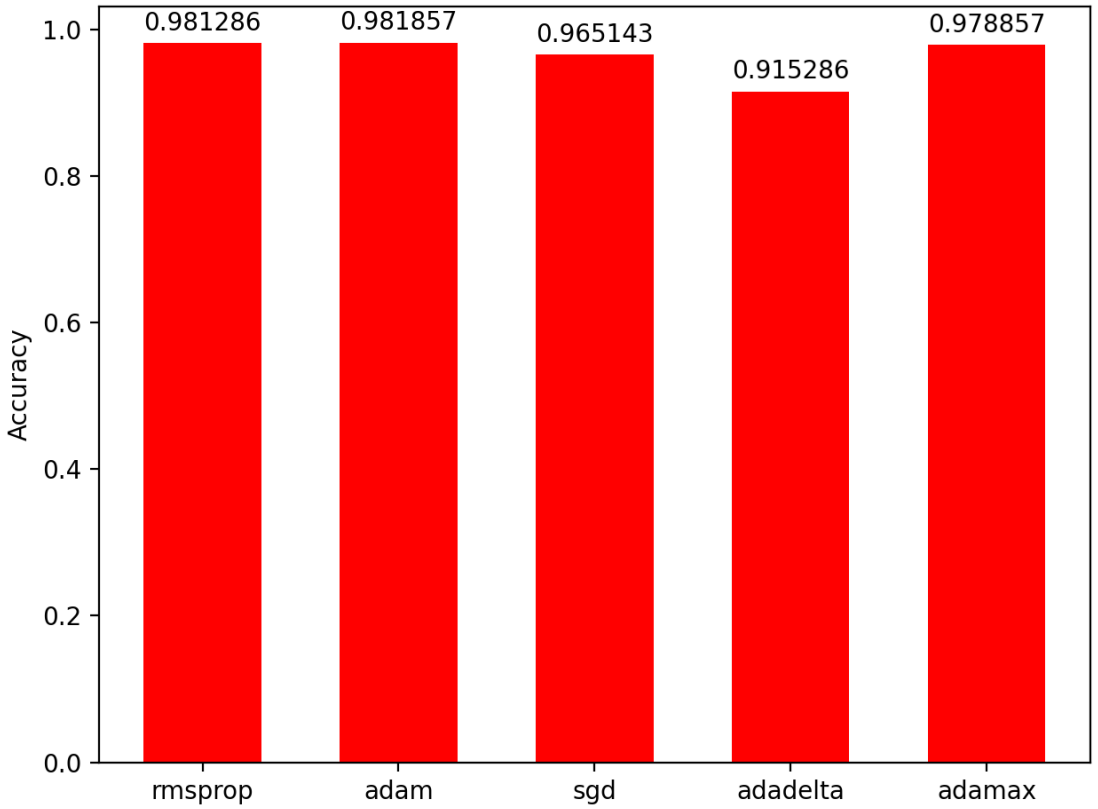
The last table, in which is compared the number of neurons in each layer shows us that a good value for it would be between 256 and 512, but since the

increase between the two values is not that significant and the computational cost increases a lot by doubling its value, the value chosen as the best would be around 256 neurons.

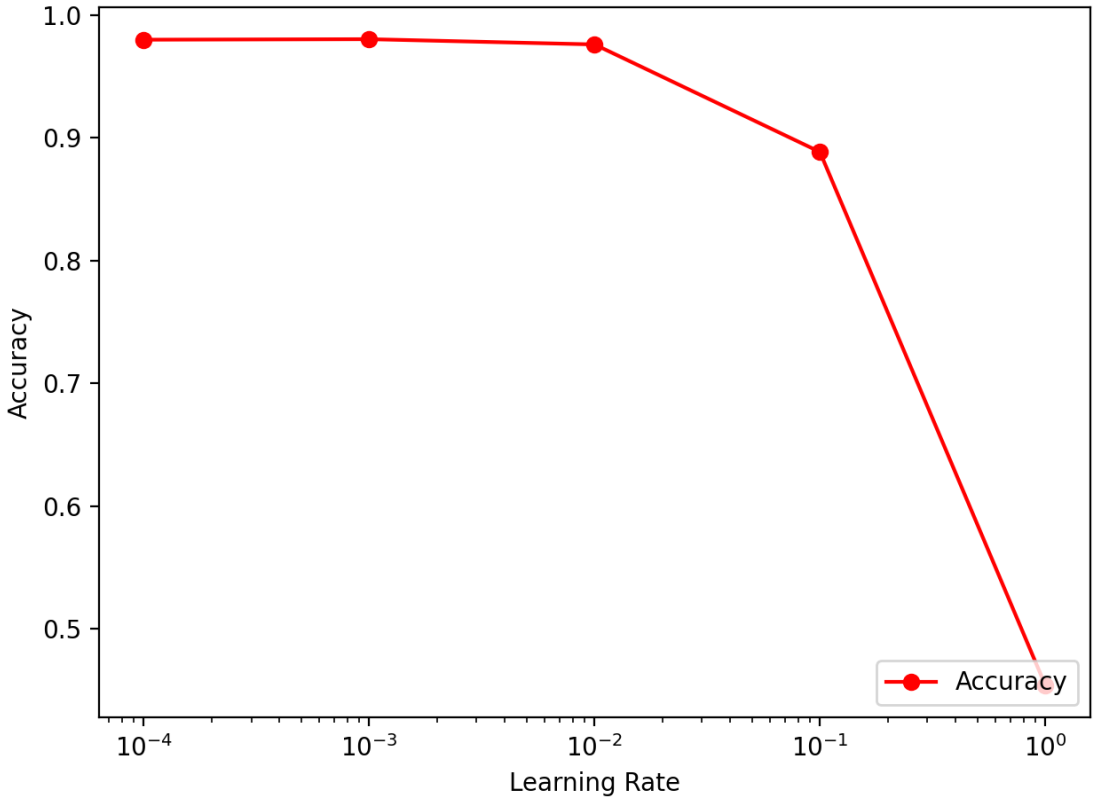
To graphically show the changes between each parameter variation, we have chosen the metric accuracy and plotted some graphics shown in the next pages.



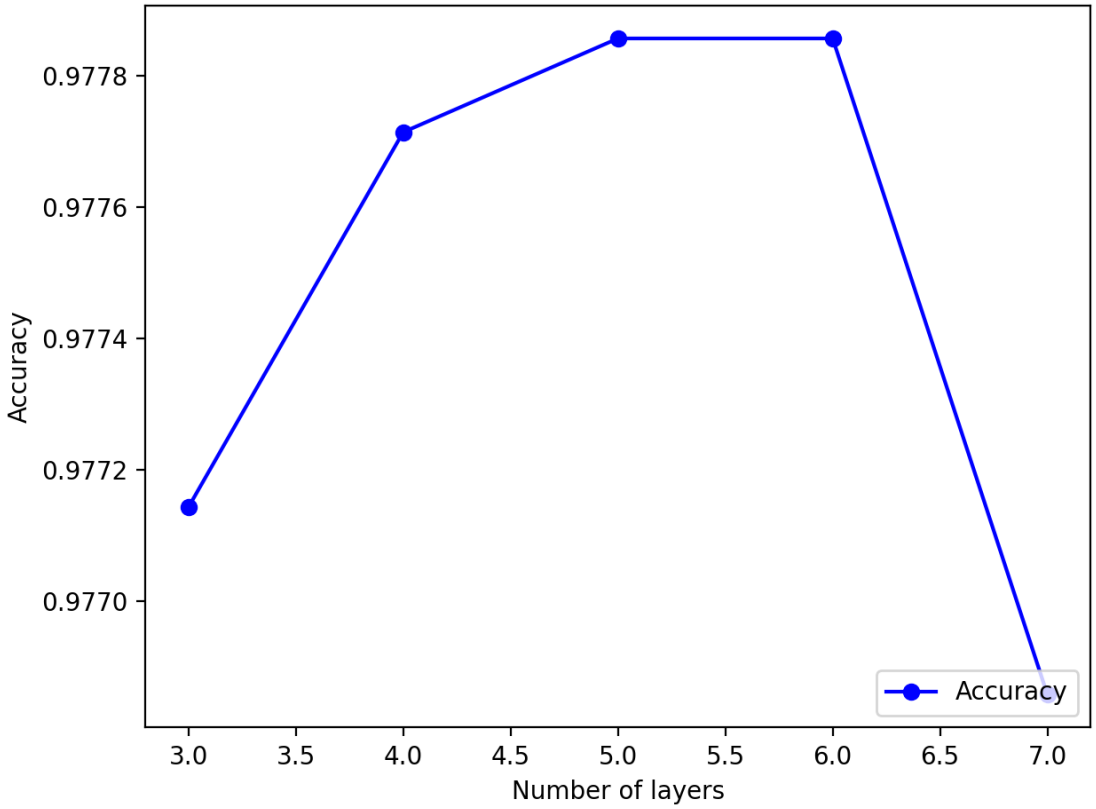

*Graphic 1: Dropout variation vs accuracy of the model*



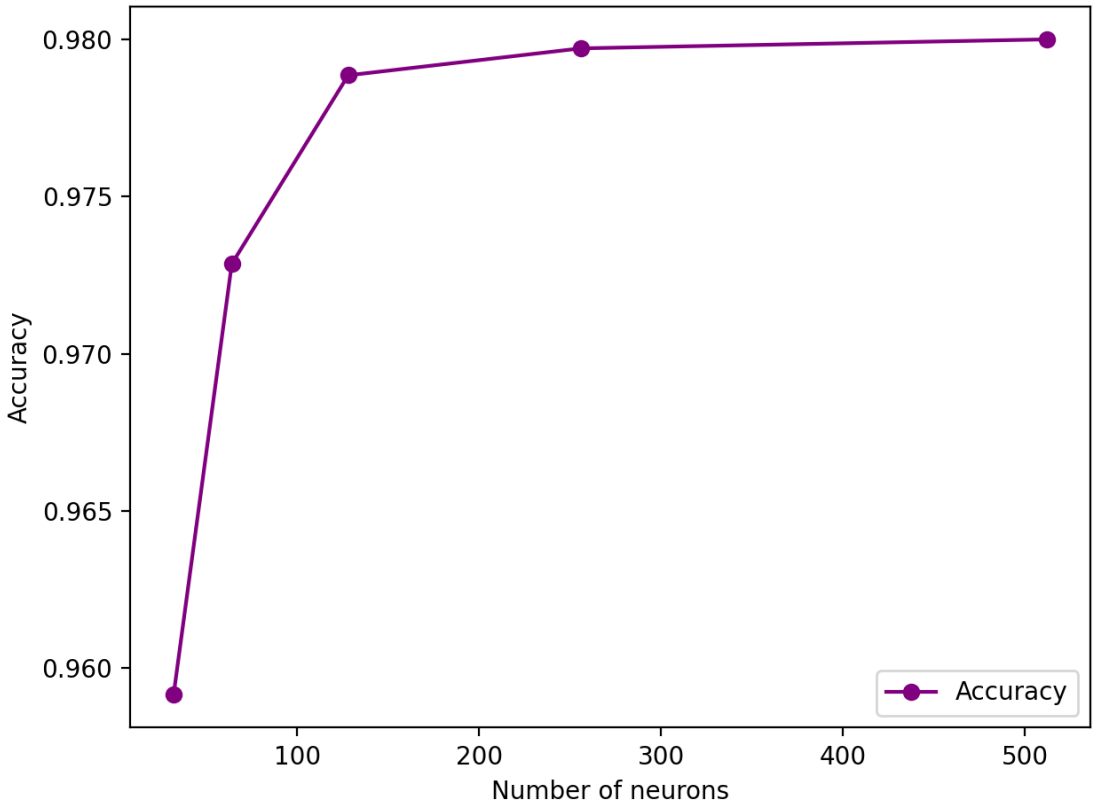

*Graphic 2: Optimizer variation vs accuracy of the model*




*Graphic 3: Learning rate variation vs accuracy of the model (x scaled logarithmically)*




*Graphic 4: Number of layers variation vs accuracy of the model*




*Graphic 5: Number of neurons variation vs accuracy of the model*

4 Multi-layer Convolutional network

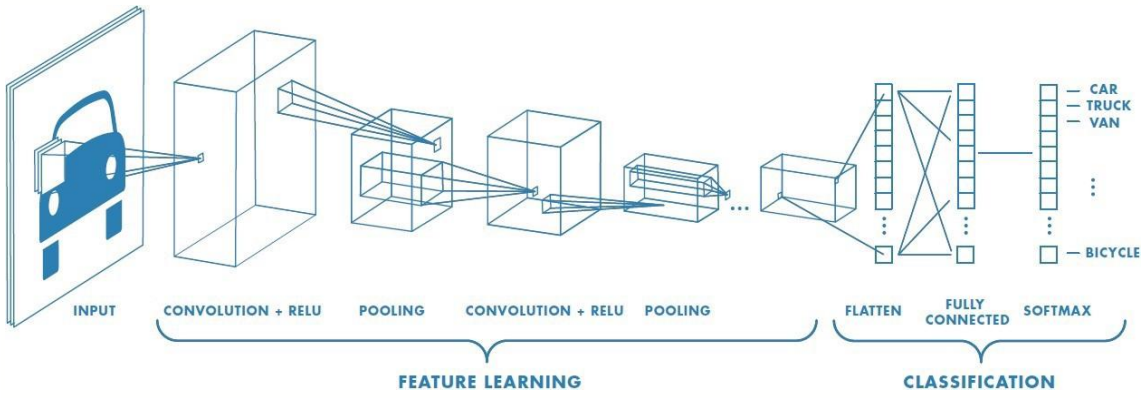
Some parts of the Multi-layer Convolutional network implementation are very similar to fully connected Multi-layer network, so we will focus on what's different in this implementation.

4.1 Implementing the Multi-layer Convolutional network

The function that implements the CNN model, it's already initialized with some convolutional layers, pooling layers and in the end with the Flatten layer followed by a fully connected layer that's the output layer, later we will expand this part to a fully-connected multi-layer.

To implement a convolutional neural network then there will be a feature learning part where there is a sequence of layers of a convolutional neural network. But these layers are divided in two parts: a convolutional layer followed by a pooling layer. The objective for this part of the algorithm is to get the features from the image, such as edges and colors. The Convolutional layers use filters to generate feature maps, and then we use the pooling layers to select the largest values on the map to use as input to the next convolutional layer.

After all this part of the algorithm, we will have a classification part and between these parts we use a flatten layer to prepare the data to the next part. That's where we use a fully connected multi-layer network, with the objective to classify the original input with the given features selected by the convolutional neural network. Here's a good image that could illustrate it:



In the definition of this model we must decide the numbers of filters, the size of the filters,

The function the we implemented for this modeling part is:

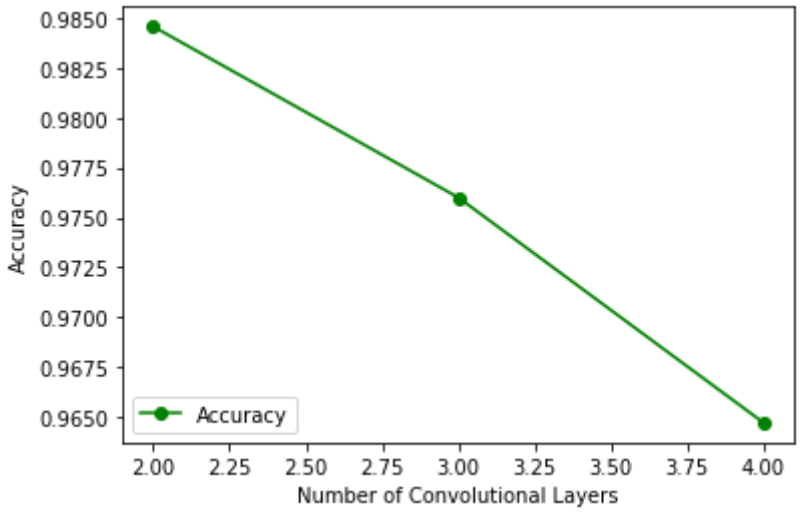
```
def CNNModel(layers, x_train, y_train, x_valid, y_valid, optimizer):
    # Initializing model
    model = Sequential()
    model.add(Conv2D(32,(3,3), activation = 'relu',
kernel_initializer='he_uniform', input_shape=(28,28,1)))
    model.add(MaxPooling2D((2,2)))
    model.add(Conv2D(64, (3,3), activation = 'relu',
kernel_initializer='he_uniform'))
    model.add(MaxPooling2D((2,2)))

    # Adding hidden layers
    for layer in layers[1:-1]:
        model.add(Conv2D(layer["filters"],layer["kernelSize"], activation =
layer["activation"], kernel_initializer='he_uniform'))
        model.add(MaxPooling2D((2,2)))

    # Adding output layer
    model.add(Flatten())
    # Using dropout to avoid overfitting
    model.add(Dropout(layers[0]["dropout"]))
    model.add(Dense(10))
    model.add(Activation('softmax'))

    # Compiling model
    model.compile(loss='categorical_crossentropy', metrics=['accuracy'],
optimizer=optimizer)
    model.summary()

    # Early stop when accuracy diverges
    es = EarlyStopping(monitor='val_accuracy',
                       patience=8,
                       min_delta=0.001,
                       mode='max')

    # model fit with maximum of 500 epochs
    history = model.fit(x_train, y_train,
            batch_size=128, epochs=500,
            verbose=2,
            validation_data=(x_valid, y_valid),
            callbacks=es)


    return model, history
```

5 Results for the multi-layer Convolutional network

After computing the results, it is possible to analyze how which parameters affect the performance for the multi-layer convolutional network. In the following tables, the comparison of the outputs varying the chosen parameters.

5.1 Different Convolutional Layers

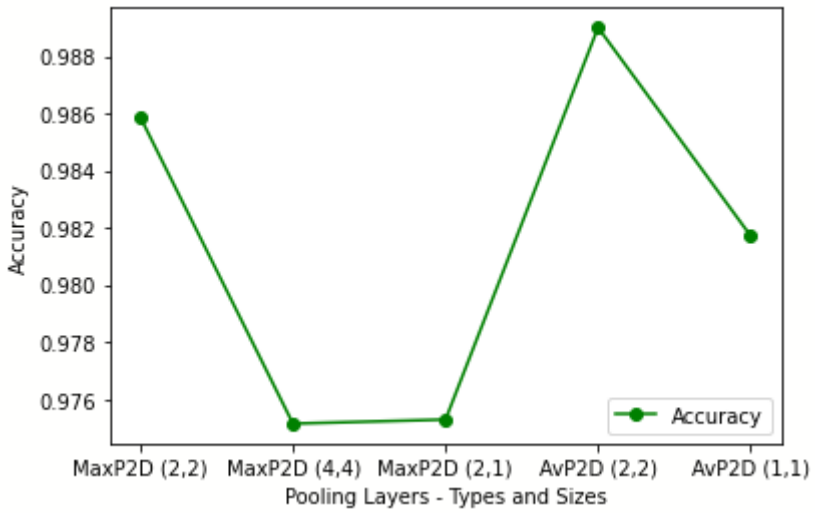
| Number of Convolutional Layers | Accuracy | Precision | Recall | F1 score | AUC |
|---|---|---|---|---|---|
| 2 | 0,984571 | 0,984522 | 0,98447 | 0,984477 | 0,999873 |
| 3 | 0,976 | 0,975812 | 0,975956 | 0,97578 | 0,999632 |
| 4 | 0,964714 | 0,964246 | 0,964766 | 0,9643 | 0,998621 |




The number of Convolutional layers are very important for this task to be done. From the results we could analyze that there is a drop in all metrics for each layer inserted. We believed that it would not be what had happened, however through the tests carried out it was what we were able to analyze, with 2 convolutional layers being the number of layers that would be the best for this application.

5.2 Different Types and Sizes of Pooling Layer

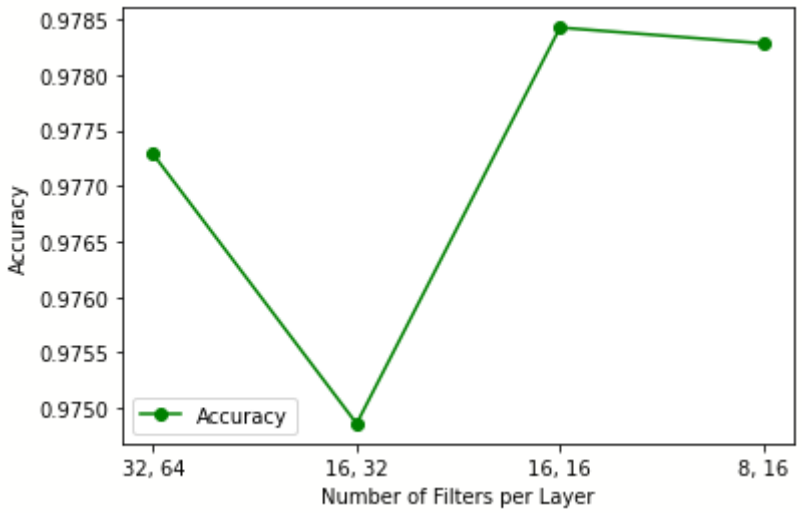
| Pooling Layer Type and Size | Accuracy | Precision | Recall | F1 score | AUC |
|---|---|---|---|---|---|
| Baseline - MaxPooling2D (2,2) | 0,985857 | 0,985885 | 0,985741 | 0,985792 | 0,999833 |
| MaxPooling2D (4,4) | 0,975143 | 0,975079 | 0,974831 | 0,974922 | 0,999580 |
| MaxPooling2D (2,1) | 0,975286 | 0,975184 | 0,975245 | 0,975151 | 0,999607 |
| AveragePooling2D (2,2) | 0,989000 | 0,988871 | 0,989005 | 0,988922 | 0,999932 |
| AveragePooling2D (1,1) | 0,981714 | 0,981661 | 0,981789 | 0,981671 | 0,999706 |




From this table we can conclude that the Average Pooling is a better choice for this project. Especially using the size (2,2) as the size, because in this configuration we could get the best accuracy of all and a AUC of 0.999932, also the best in the tests. Another conclusion is that the size (2,2) was the best independently from the pooling layer type.

5.3 Different Numbers of Filters per layer

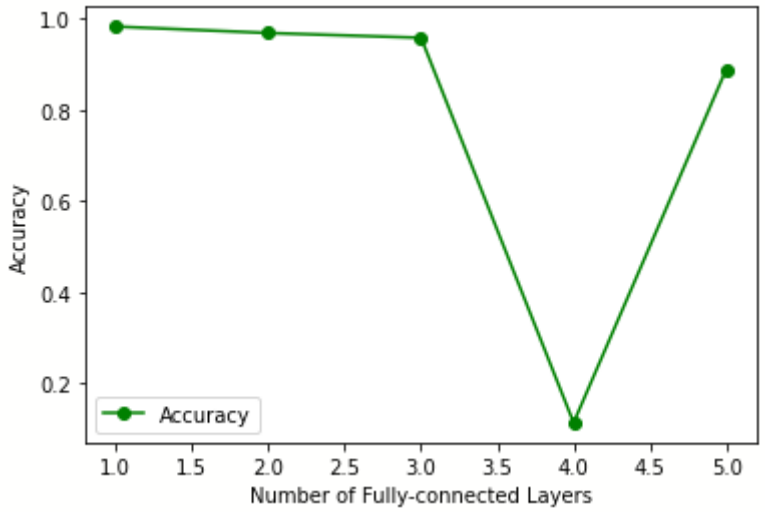
| Number of Filters per Layer | Accuracy | Precision | Recall | F1 score | AUC |
|---|---|---|---|---|---|
| 32, 64 | 0,977286 | 0,977314 | 0,97734 | 0,977271 | 0,999523 |
| 16, 32 | 0,974857 | 0,975214 | 0,974608 | 0,97485 | 0,999546 |
| 16, 16 | 0,978429 | 0,978117 | 0,978482 | 0,978254 | 0,999707 |
| 8, 16 | 0,978286 | 0,978098 | 0,978365 | 0,97818 | 0,999638 |




The results from this variation of the convolutional network made us conclude that finding the optimum number of filters per layer could be a challenge, because as we can see there's not a very clear pattern as "more filters is better", nor the other way. For this dataset the implementation that was the best was using layers of 16 filters each.

5.4 Number of Fully-connected Layers

| Number of Fully-connected Layers | Accuracy | Precision | Recall | F1 score | AUC |
|---|---|---|---|---|---|
| 1 | 0,982286 | 0,982124 | 0,982222 | 0,982149 | 0,999703 |
| 2 | 0,968143 | 0,967937 | 0,968127 | 0,967920 | 0,999098 |
| 3 | 0,957714 | 0,957432 | 0,958296 | 0,957620 | 0,998179 |
| 4 | 0,114286 | 0,011429 | 0,100000 | 0,020513 | 0,500000 |
| 5 | 0,888143 | 0,889485 | 0,887476 | 0,887712 | 0,989106 |




Using a fully-connected multi-layer network too deep also didn't seem very effective by our results. Actually by the tests done, we could conclude that the best accuracy was gotten from the cnn network that after the flatten layer got directly to the output layer for the classification part.

6 Conclusions

Analyzing all the results from our tests we can't declare a better model between the fully connected multi-layer network and the multi-layer convolutional network. But we can guarantee that the choice of the parameters and how to use them are very important, and making the right choices can lead us to an optimum result with metrics about 99%.

All our implementations can be found in the notebook attached with this report and in this Google Colab notebook. This was our first experience using Python and Keras to implement a neural network, as well as using a collaborative development tool, Google Colab, where we were able to implement and execute our project. And for us it was a great experience.

## 7 References


How to Develop a CNN for MNIST Handwritten Digit Classification

Understand the Impact of Learning Rate on Neural Network Performance

How Do Convolutional Layers Work in Deep Learning Neural Networks?

A Gentle Introduction to k-fold Cross-Validation

MNIST Handwritten Digit Recognition in Keras

Keras for Beginners: Building Your First Neural Network

MNIST - Deep Neural Network with Keras

Introduction to Multilayer Neural Networks with TensorFlow's Keras API