Instituto Superior Técnico

Neural Networks

Machine Learning

Autors:	*Numbers:*
Manuel Freitas	85246
Guilherme Lopes	87016

Professor: Maria Margarida Campos da Silveira

November 2019

Contents

Chapter 1: Digit Recognition

In this assignment we were presented with a problem of image classification to be solved via neural networks. We then created and trained a multilayer perceptron(MLP) and a convolutional neural network(CNN) to classify those images and compared the results of both approaches.

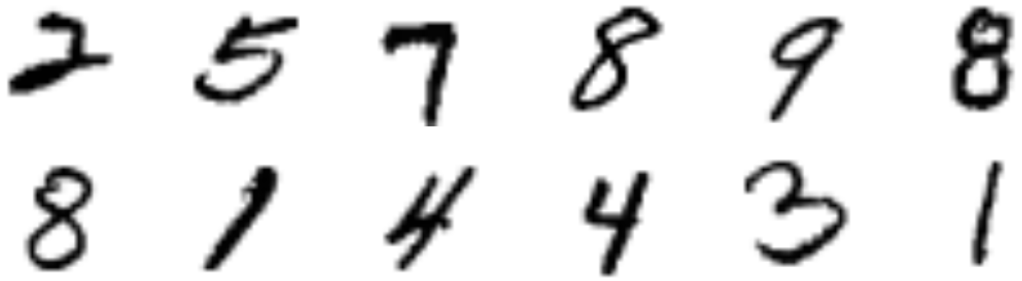


Figure 1.1: Digit examples

Chapter 2: Data

The data examples previously presented were obtained from a well know data set (MNIST). We will also use a small portion of this data set to train and test our neural networks. We will consider 3000 training images and 500 test images. Below we present the code used to load the data files and check the size of the inputs, display digits from the data set (and the respective images the code presents) and convert the labels to one-hot encoding. The validation data splitting is in fact done in a later stage, as the validation_split= 0.3 parameter in the MLP.fit() function.

```
X_test = np.load("mnist_test_data.npy")
Y_test = np.load("mnist_test_labels.npy")
X_train = np.load("mnist_train_data.npy")
Y_train = np.load("mnist_train_labels.npy")

print("Random image from train data set")
plt.imshow(X_train[random.randrange(1,3001,1)].squeeze(), cmap = "gray")
plt.show()
print("Random image from test data set")
plt.imshow(X_test[random.randrange(1,501,1)].squeeze(), cmap = "gray")
plt.show()


X_test=X_test/255
X_train=X_train/255
Y_train_1hot = to_categorical(Y_train)
Y_test_1hot = to_categorical(Y_test)
```

The variables *X_train*,and *X_test* are simply the result of diving each of the values by 255 so that we can turn the images from the MNIST data set (RGB) into grayscale ones.

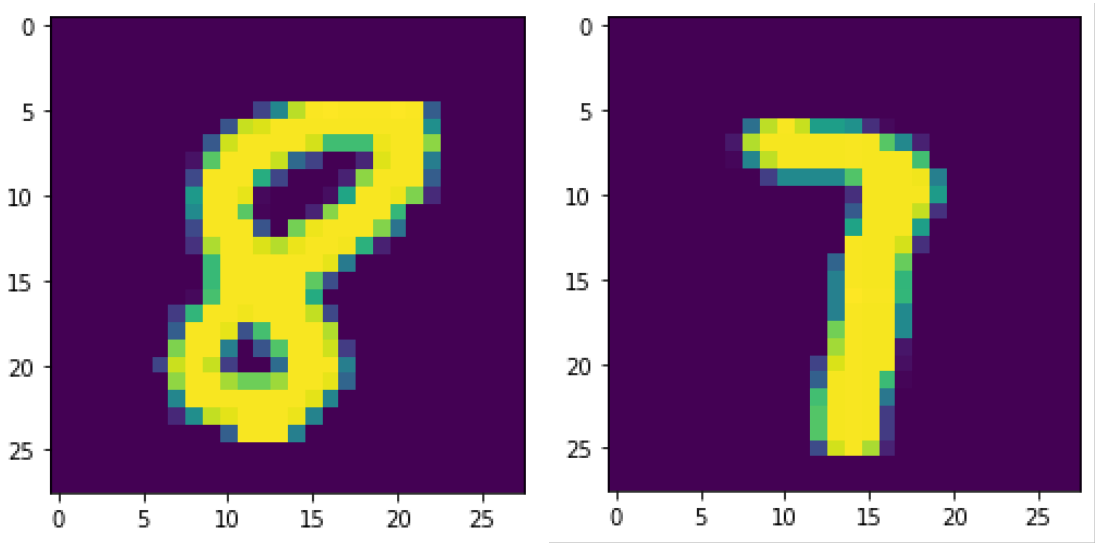


Figure 2.1: Data files before dividing by 255 (RGB)

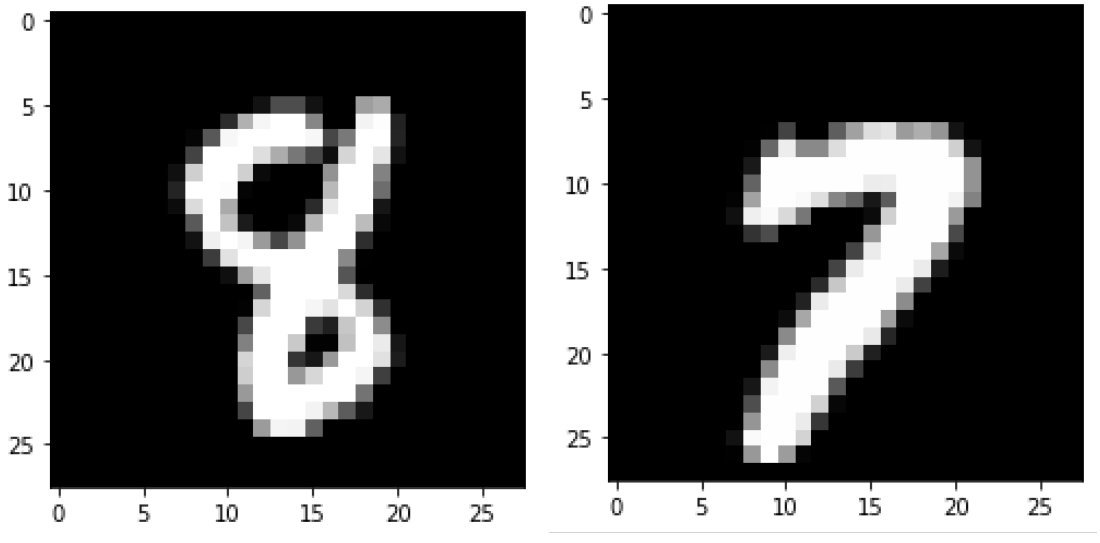
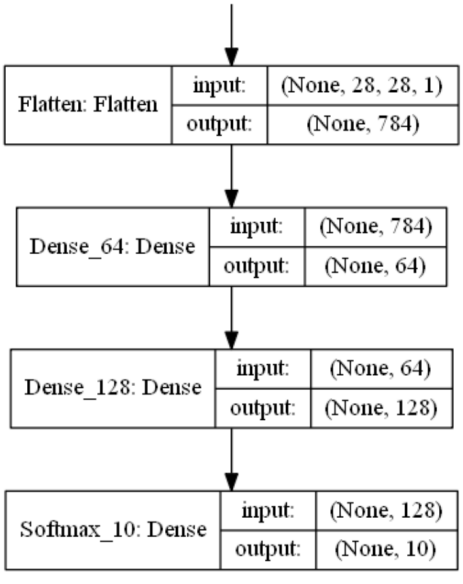


Figure 2.2: Data files after dividing by 255 (grayscale)

Chapter 3: MLP

Based of the the graph plot of the MLP model given below, we created our MLP in python(as the code shows below). Then via the summary() function we verified if the generated MLP was as expected. Comparing the return of the summary() function and the graph plot given in the assignment we concluded that was true.

```
MLP = Sequential()
MLP.add(Flatten(input_shape=(28,28,1)))
MLP.add(Dense(units=64,activation='relu'))
MLP.add(Dense(units=128,activation='relu'))
MLP.add(Dense(10, activation='softmax'))
MLP.summary()
ES = EarlyStopping(patience=15,restore_best_weights=True)
MLP.compile(loss='categorical_crossentropy',optimizer='Adam')
MLPhistory=MLP.fit(x=X_train,y=Y_train_1hot ,batch_size=300,epochs=400,callbacks
    =[ES],validation_split=0.3)
plt.figure()
plt.plot(MLPhistory.history['loss'], label='train')
plt.plot(MLPhistory.history['val_loss'], label='test')
Y_predicted= MLP.predict(X_test)
score=accuracy_score(Y_test_1hot.argmax(axis=1),Y_predicted.argmax(axis=1))
cm = confusion_matrix(Y_test_1hot.argmax(axis=1),Y_predicted.argmax(axis=1))
```



Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0
dense_3 (Dense)	(None, 64)	50240
dense_4 (Dense)	(None, 128)	8320
dense_5 (Dense)	(None, 10)	1290

Figure 3.1: Graph plot and summary() return

The theory behind the processes of a multilayer perceptron is that each neuron or unit x is

connected to the next y via a weight w_{xy}.The value of each neuron z is determined via the weighted sum of the input from all the neurons from the previous layer and an offset w_0, as show in the equation below:

$$s_y = w_{0y} + \sum_{x \in previous\ layer} w_{xy} z_x \tag{3.1}$$

$$z_y = g(s_y) \tag{3.2}$$

This resulted is parsed through an activation function(in this case the rectifier function) and that gives us the value of that neuron on that specific iteration. The rectifier function gives us Rectified Linear Units (ReLU) and is given by:

$$g(x) = max0, x \tag{3.3}$$

The final softmax layer function takes as input the vector of 10 numbers from the final layer, and normalizes it into a probability distribution consisting of probabilities proportional to the exponentials of the input. That is, prior to applying softmax, some vector components could be negative, or greater than one; and might not sum to 1; but after applying softmax, each component will be in the interval (0,1) and the components will add up to 1, so that they can be interpreted as probabilities. The softmax function is given by:

$$\sigma(z)_x = \frac{e^{z_x}}{\sum_{y=1}^{10} e^{z_y}}, x = 1, ..., 10 \text{ and } z = (z_1, ..., z_{10}) \in \mathbb{R}^{10} \tag{3.4}$$

We then created an EarlyStopping monitor to stop training if there was no relevant improvement in the validation loss. This would cause the neural network to be faster at achieving a reasonably good state.

We can now fit our MLP to the training and validation data. We used the *categoricalcrossentropy* formula to determine the loss value.This means that we use the softmax function and a cross-entropy loss. That is given by:

$$CE = -log(\frac{e^{s_x}}{\sum_y^C e^{s_y}}) \tag{3.5}$$

We can also evaluate the accuracy and obtain the confusion matrix for this training set:

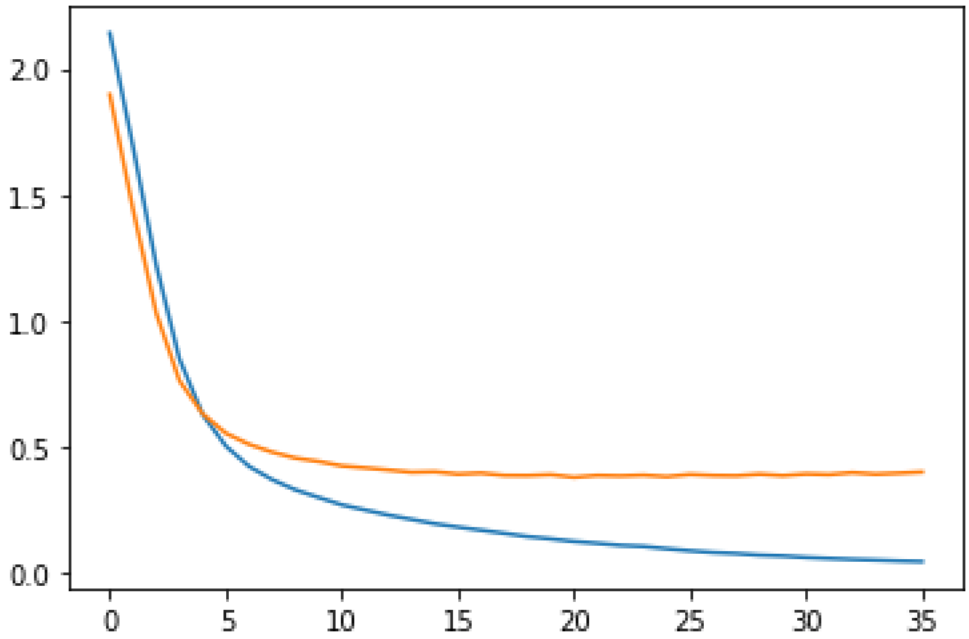


Figure 3.2: val_loss(orange) and loss(blue) plots with early stopping

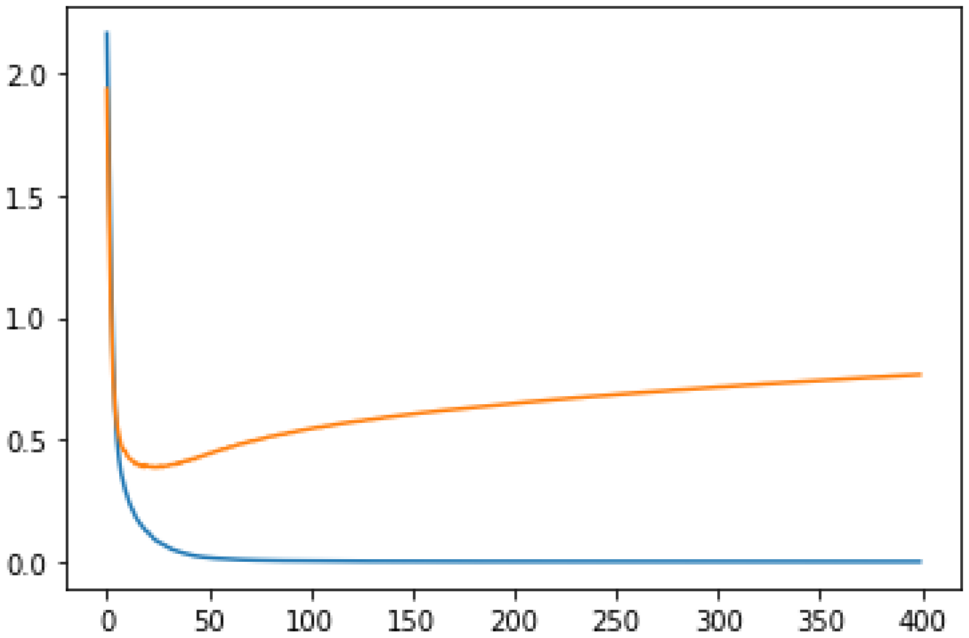


Figure 3.3: val_loss(orange) and loss(blue) plots without early stopping

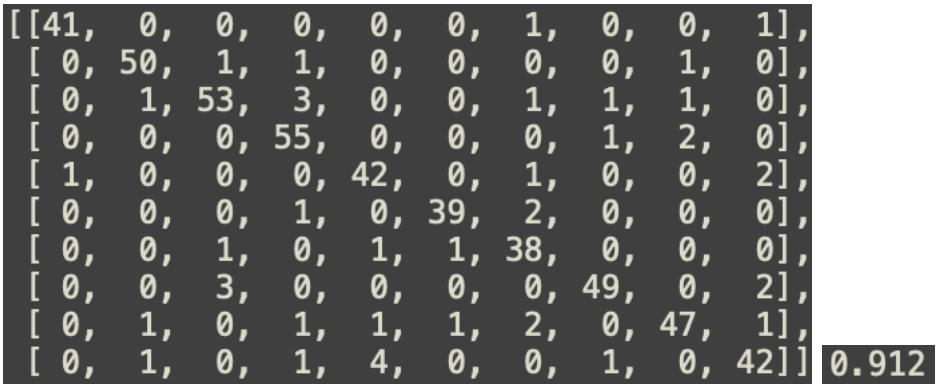


Figure 3.4: Accuracy score and confusion matrix with early stopping

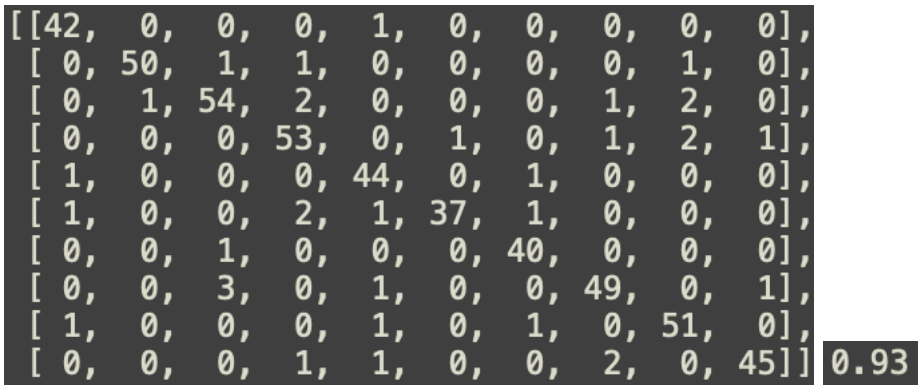
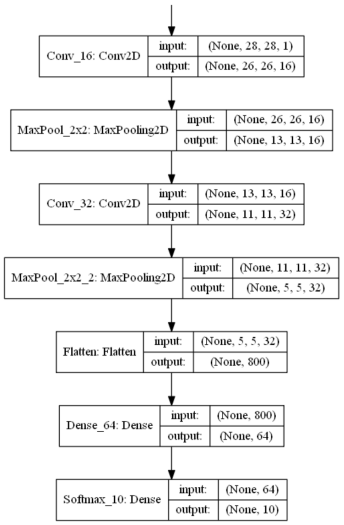


Figure 3.5: Accuracy score and confusion matrix without early stopping

Chapter 4: CNN

Now, based on the graph plot given for the CNN, we will do as before. Create a convolutional neural network and then, via summary() check if it is as intended.

```
CNN = Sequential()
CNN.add(Conv2D(filters=16,kernel_size=(3,3),activation='relu',input_shape
    =(28,28,1)))
CNN.add(MaxPooling2D(pool_size=(2,2)))
CNN.add(Conv2D(filters=32,kernel_size=(3,3),activation='relu'))
CNN.add(MaxPooling2D(pool_size=(2,2)))
CNN.add(Flatten())
CNN.add(Dense(units=64,activation='relu'))
CNN.add(Dense(10, activation='softmax'))
CNN.summary()
CNN.compile(loss='categorical_crossentropy',optimizer='Adam')
CNNhistory=CNN.fit(x=X_train,y=Y_train_1hot ,batch_size=300,epochs=400,callbacks=
    None,validation_split=0.3)
plt.figure()
plt.plot(CNNhistory.history['loss'], label='train')
plt.plot(CNNhistory.history['val_loss'], label='test')
Y_predicted_CNN= CNN.predict(X_test)
score_CNN=accuracy_score(Y_test_1hot.argmax(axis=1),Y_predicted_CNN.argmax(axis
    =1))
cm_CNN = confusion_matrix(Y_test_1hot.argmax(axis=1),Y_predicted_CNN.argmax(axis
    =1))
```



Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 16)	160
max_pooling2d (MaxPooling2D)	(None, 13, 13, 16)	0
conv2d_1 (Conv2D)	(None, 11, 11, 32)	4640
max_pooling2d_1 (MaxPooling2	(None, 5, 5, 32)	0
flatten_5 (Flatten)	(None, 800)	0
dense_15 (Dense)	(None, 64)	51264
dense_16 (Dense)	(None, 10)	650

Figure 4.1: Graph plot and summary() return

The theory behind a convolutional neural network is based on the tought of simplifying something or reducing it to useful information. This is specially effective in imagery, which is the case we are dealing with, and we will use a simpler case to explain the way this kind of neural networks work. Lets take a 5x5(in reality it is a 5x5x1 square as each position has a weight associated to it) square like the one below. To simplify it we can transform it into a 3x3 square that carries information related to it. Applying the Kernel filter gives us the convolved feature above. We then shift this filter and apply it to the rest of the 5x5 square.The aim is to extract the features from the input and obtain high-level features.

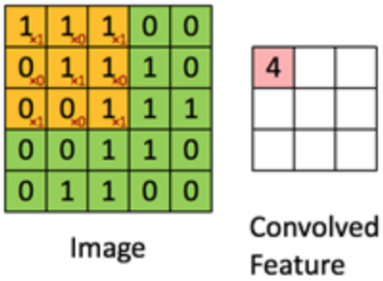


Figure 4.2: Kernel filter example

The type of Pooling we used for this network was MaxPooling. This means that, from the complete convolved feature, when moving to higher levels, we would take the MaxValue from where the filter hovers (another technique could be AvgPooling, that would take the average instead).

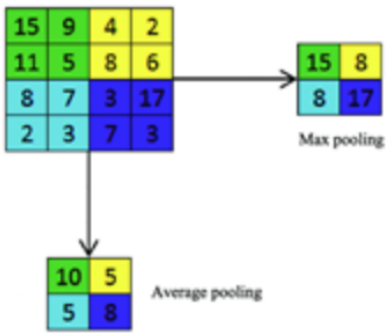


Figure 4.3: MaxPooling and AvgPooling

We now want to reduce the dimension of each layer we are creating in relation to the previous one. After repeating this process we will then pass this onto a classification "layer" (in our case three layers that are very similar to the multilayer perceptron previously explained).

We then repeated the same processes as for the MLP and obtained the following results:

We can also evaluate the accuracy and obtain the confusion matrix for this training set:

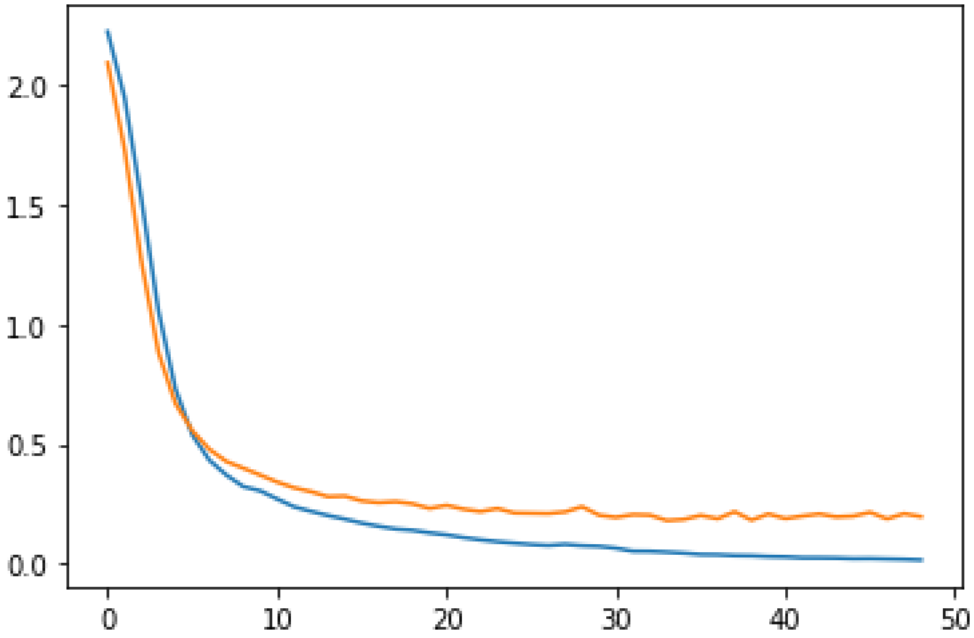


Figure 4.4: val_loss(orange) and loss(blue) plots with early stopping

```
[[43,  0,  0,  0,  0,  0,  0,  0,  0,  0],
 [ 0, 50,  0,  2,  0,  0,  0,  0,  1,  0],
 [ 0,  0, 60,  0,  0,  0,  0,  0,  0,  0],
 [ 0,  0,  0, 55,  0,  0,  0,  1,  1,  1],
 [ 1,  0,  0,  0, 44,  0,  1,  0,  0,  0],
 [ 0,  0,  0,  1,  0, 40,  1,  0,  0,  0],
 [ 0,  0,  0,  0,  0,  0, 41,  0,  0,  0],
 [ 0,  0,  3,  0,  1,  0,  0, 49,  0,  1],
 [ 0,  0,  0,  0,  0,  2,  0,  0, 51,  1],
 [ 0,  0,  0,  0,  1,  0,  0,  1,  0, 47]] 0.96
```

Figure 4.5: Accuracy score and confusion matrix with early stopping

Chapter 5: Comments

Early stopping is a valuable strategy to train a Neural Network . It can be used to prevent overfitting as well as speeding up the time it takes to properly train a network. The criteria is to stop the training process if the loss between training iterations is too small.

If this strategy is not used the model can "learn" the noise in the training set, and overfit the data.

If overfitting happens, upon the validation, the network will output values that have minimum loss compared against the training data, but not when compared against the validation data.

This can be observed, in figures 3.2 and 3.3 .

As for the test data, the accuracy values can be misleading, and it is mainly a reflection of the training data. The validation loss is smaller with early stopping, but if the model is not being well trained, there could be a bigger uncertainty about the label to give to the image. Comparing figures 3.4 and 3.5, we can see this is the case, the early stopping model is underfitting. Without early stopping the model is less "confused", because with more iterations, it was better trained in that regard, even if overfitting. With better training data, the early stopping model would fit better and would produce the best accuracy values.

Between CNN and MLP, with early stopping, the best performance is in fact by the CNN model. This can be traced to multiple reasons : The validation loss is smaller, as observed in figures 3.2 and 4.4. The CNN model is also not underfitting the data, mainly because the max pooling stage, that identifies the most useful features of the image, ultimately filtering the noisy data and consequently resulting in more sparse confusion matrices and better data classification.

As for the number of parameters in both models, CNN has the upper hand, with 56714 weights. It has better performance because it supports parameter and weight sharing, that result in feature extraction. The MLP has 59648 weights, and if some pixels are noisy, this will result in error propagation and ultimately worse classification.