SCHOOL OF ECONOMICS AND MANAGEMENT

Building a Convolutional Neural Network Model to Identify the American Sign Language (ASL)

Final Project in Deep Learning and Artificial Intelligence Methods

By Pinyapat Manasboonpermpool, Stella Säfström and Vendela Meyerhöffer

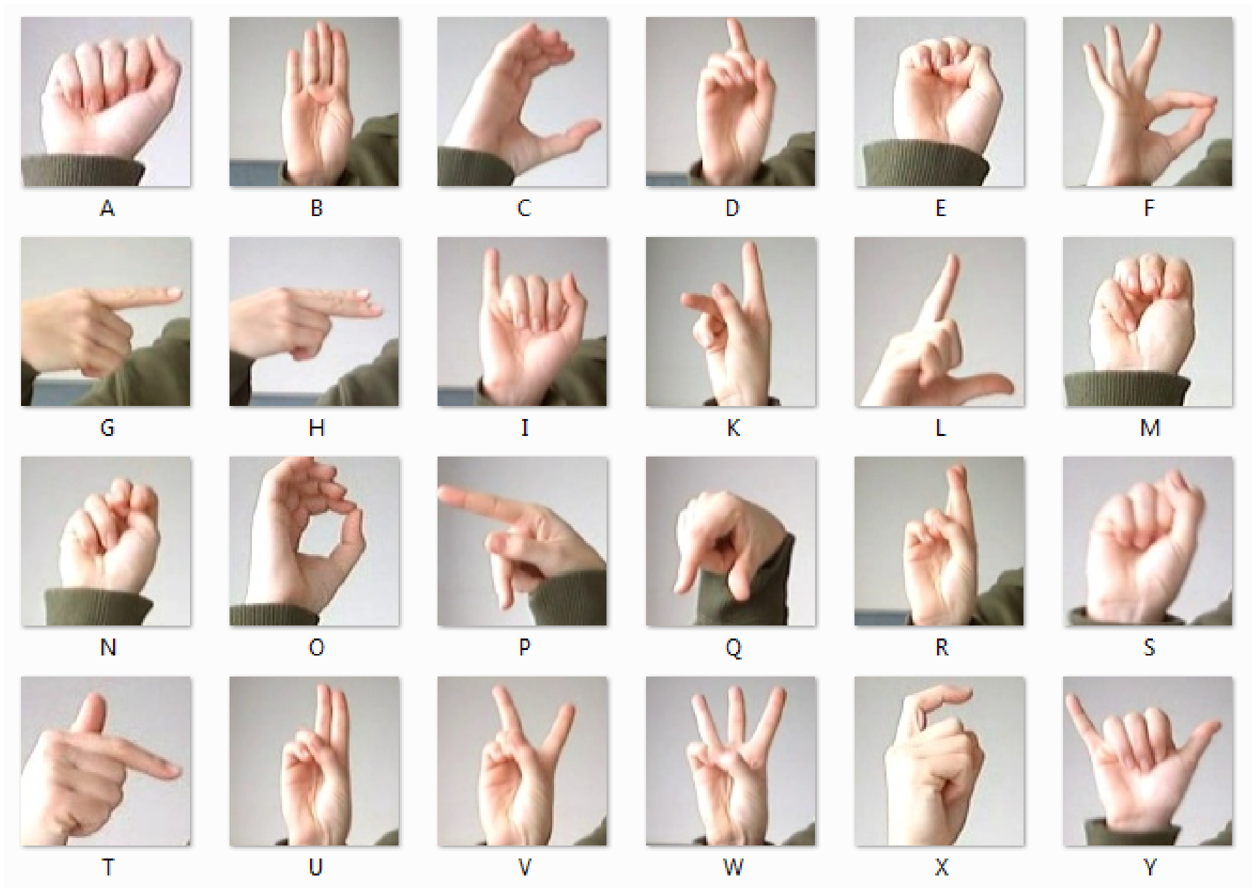
March 2022

1 Introduction

According to **Human Rights Watch** , more than 70 million deaf people use sign language in order to communicate. Many of them use the American Sign Language alphabet in their communication. In this report, we have chosen to work with the Sign Language MNIST data set, which consists of a set of pictures depicting hand gestures that represent 24 letters of the American Sign Language alphabet. All but two letters of the alphabet are present in the data set. The letters (J=9) and (Z=25) are missing, due to that these letters require hand motions and thus cannot be captured in a single image.

In this report, we aim at correctly classifying the 24 labels (hand gestures/ letters) by building a suitable convolutional neural network. Throughout the project we are continuously tweaking and changing the model to improve the test accuracy. This is an interesting problem, seeing as it could improve the communication between deaf and hard-of-hearing through computer vision applications.

The data consists of 27,455 training cases and 7,172 test cases. The data set is similar to MNIST with a header row for the label and all pixel values ranging from pixel1 to pixel784. These values represent a single 28x28 pixel image in grayscale (thus with values between 0 - 255).



1.1 Method

Convolutional neural networks (CNN) are a special type of neural networks that work especially well in image recognition. In very simple terms, convolutional neural networks are able to identify local features in images, which are combined to output a classification. The first layers of a convolutional neural network are often able to identify low-level features such as edges, circles and shapes. In the next layers more complex features such as eyes, fur and teeth are identified.

The model is able to classify images by using convolutional layers and pooling layers. Convolutional layers search for features and patterns in the images. The layer consists of many convolutional filters (also known as feature detectors) which determine if there is a certain feature in an image. This is done by convolutions. We will not go into the mathematical details of how this works.

The pooling layer condenses large images into smaller versions that summarize its information. The max pooling layer (which we choose to use in this project) does this by taking the maximum valued pixel in each non-overlapping 2x2 block in the image.

We can choose how many convolutional and pooling layers we see fit for our model. There are also several tuning parameters to take into account. We can for instance use dropout learning and regularization methods. In this project we will attempt to tweak and tune a basic CNN model in order to reduce the test loss as much as possible.

(Source: An Introduction to Statistical Learning, 2021)

1.2 Importing the libraries

We begin this project by loading all the necessary libraries for the analysis.

```
[ ]: # Use seaborn for pairplot
     !pip install -q seaborn
```

```
[ ]: from __future__ import absolute_import, division, print_function,␣
      ↪unicode_literals

     import pathlib

     import matplotlib.pyplot as plt
     import pandas as pd
     import seaborn as sns
     import numpy as np
     import keras
     from keras.models import Sequential
     from keras.layers import Dense, Conv2D , MaxPool2D , Flatten , Dropout ,␣
      ↪BatchNormalization
     from keras.preprocessing.image import ImageDataGenerator
     from sklearn.model_selection import train_test_split
     from sklearn.metrics import classification_report,confusion_matrix,␣
      ↪accuracy_score
     from keras.callbacks import ReduceLROnPlateau
     from keras.utils.np_utils import to_categorical
     import math

     try:
       # %tensorflow_version only exists in Colab.
       %tensorflow_version 2.x
     except Exception:
       pass
     import tensorflow as tf

     from tensorflow import keras
     from tensorflow.keras import layers


     import logging
     logger = tf.get_logger()
     logger.setLevel(logging.ERROR)


     print(tf.__version__)
```

```
2.8.0
```

1.3 Exploring the data

We start the analysis by loading the Sign Language MNIST data into the working directory. The data has already been divided into training and test data sets, so there is no need for us to divide the data ourselves.

[ ]:
```
#Load Data
sign_lang_train = pd.read_csv("sign_mnist_train.csv")
sign_lang_test = pd.read_csv("sign_mnist_test.csv")
```

Next, we take a look at the data types in our training data. We can see that we have one label and 784 pixel values. The data consists of integers of bit size 64.

[ ]:
```
#Explore the type of the data
sign_lang_train.dtypes
```

[ ]:
```
label       int64
pixel1      int64
pixel2      int64
pixel3      int64
pixel4      int64
            ...
pixel780    int64
pixel781    int64
pixel782    int64
```

```
pixel783    int64
pixel784    int64
Length: 785, dtype: object
```

```
[ ]: #Observe the training data (27455 , 785)
     print(sign_lang_train.head(-5))
     sign_lang_train.shape
```

```
[27450 rows x 785 columns]
```

```
[ ]: (27455, 785)
```

```
[ ]: #Observe the test data (7172 , 785)
     print(sign_lang_test.head(-5))
     sign_lang_test.shape
```

```
[ ]: (7172, 785)
```

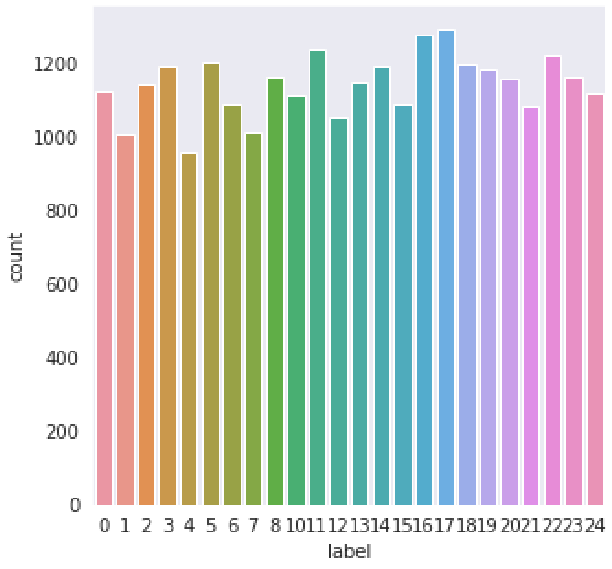
In order to get a better grasp of the data, we plot a histogram to see the overall distribution from both train and test labels.

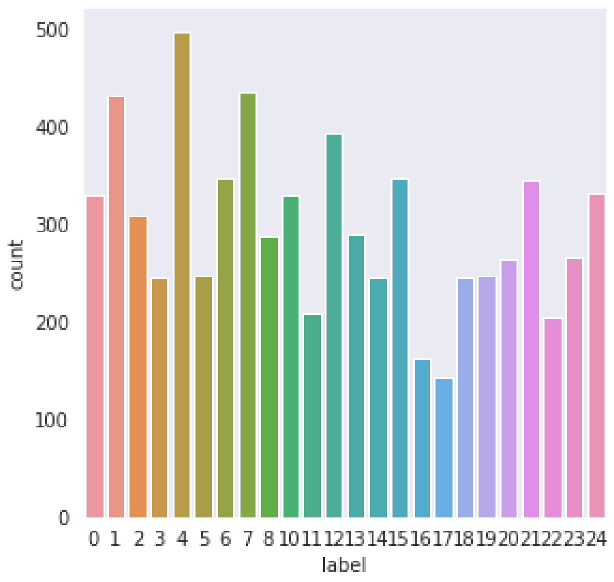
As one can see, each training label is considered to be well-balanced which indicates that there are adequate training examples to build a model. Apart from this, the test label appears to be perfect for further evaluation on a model due to its unbalanced distribution.

```
[ ]: plt.figure(figsize = (5,5)) # Label Count for Train, quiet well distributed
     sns.set_style("dark")
     sns.countplot(sign_lang_train['label'])

     plt.figure(figsize = (5,5)) # Label for Test, quiet unbalanced
     sns.set_style("dark")
     sns.countplot(sign_lang_test['label'])
```

```
[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3f531a8550>
```





[ ]:
```
#Unique to find out the total numbers of layers
labels = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "K", "L", "M", "N", "O",␣
 ↪"P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y"]
unique_val = sign_lang_train['label']
np.unique(unique_val)
```

[ ]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24])
```

1.4 Preprocessing the data

In order to perform CNN, we need to normalize, reshape and encode the labels to so called one hot vectors. We also shuffle the training data so that the model cannot learn anything from the order of the data.

1.4.1 Shuffle the training data

We then shuffle the training data to reduce any variances and that they can represent the overall distribution of the data before building the CNN model.

[ ]:
```
sign_lang_train = sign_lang_train.sample(frac = 1)
sign_lang_train.head()
```

[ ]:
```
       label  pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  pixel8  \
13049     18     191     193     195     198     200     201     204     205
25786      4     194     194     196     197     198     198     199     199
2372      12     175     176     176     176     175     176     177     174
13225      2     173     174     173     173     174     174     174     173
3578      22     101     117     131     139     142     147     155     159

       pixel9  ...  pixel775  pixel776  pixel777  pixel778  pixel779  \
13049     206  ...       206       246       239       241       241
25786     198  ...        18         3       195       252       237
2372      173  ...       205       202       202       201       201
13225     173  ...       217       216       216       215       215
3578      165  ...       110       106       107        81        46

       pixel780  pixel781  pixel782  pixel783  pixel784
13049       242       242       242       242       242
25786       238       237       236       236       236
2372        199       197       196       194       193
13225       214       212       211       196       166
3578         43        30        27        31        29

[5 rows x 785 columns]
```

1.4.2 One-Hot Encoding

We begin by converting the labels into one-hot vectors. If we fail to do this, the model can incorrectly assume that there is an ordinal relationship between the labels. This can result in poor performance of our model according to **machinelearningmastery.com** . Therefore, we convert the labels into one hot vectors as seen below using the label binarizer function.

[ ]:
```
#Extracting y, i.e. the label values
y_train = sign_lang_train['label'].values
y_test = sign_lang_test['label'].values

print(y_train)
```

```
[18  4 12 ... 19 21  4]
```

```
[ ]: #Binary Classifer / One hot encoding

     from sklearn.preprocessing import LabelBinarizer
     label_binarizer = LabelBinarizer()
     y_train = label_binarizer.fit_transform(y_train)
     y_test = label_binarizer.fit_transform(y_test)

     print(y_train)
     print(y_test)
```

```
[[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]
 [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 1 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 1 ... 0 0 0]]
```

```
[ ]: #Number of labels
     num_labels = y_train.shape[1]
     print(f"There are {num_labels} labels in the dataset")
```

```
There are 24 labels in the dataset
```

1.4.3 Normalization and reshaping

Next, we normalize the training and test data. We do this in order to make the algorithm converge faster.

Simultaneously we reshape the data into 28x28x1 matrices.

```
[ ]: #Construct x_train and x_test
     x_train = sign_lang_train.drop(columns = ['label']).to_numpy().
      ↪reshape((sign_lang_train.shape[0], 28, 28,1)).astype('float64')/255.0
     x_test = sign_lang_test.drop(columns = ['label']).to_numpy().
      ↪reshape((sign_lang_test.shape[0], 28,28,1)).astype('float64')/255.0

     x_train.shape
```

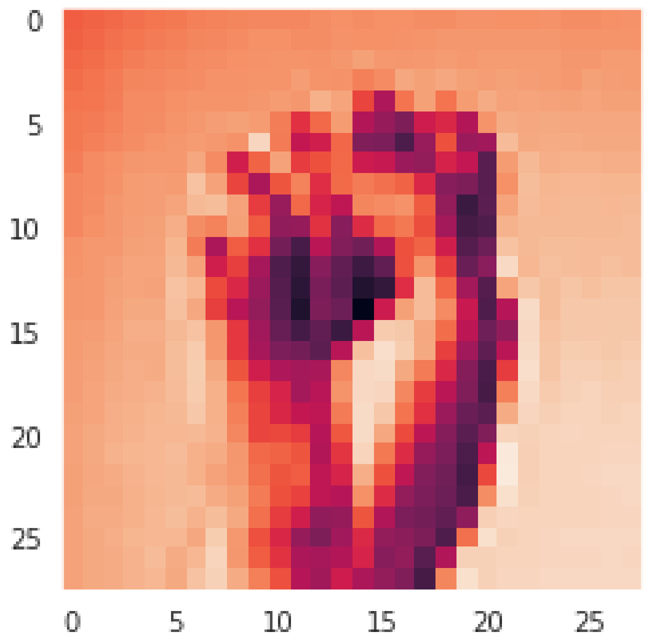
```
[ ]: (27455, 28, 28, 1)
```

1.5 Examples of pictures in the data

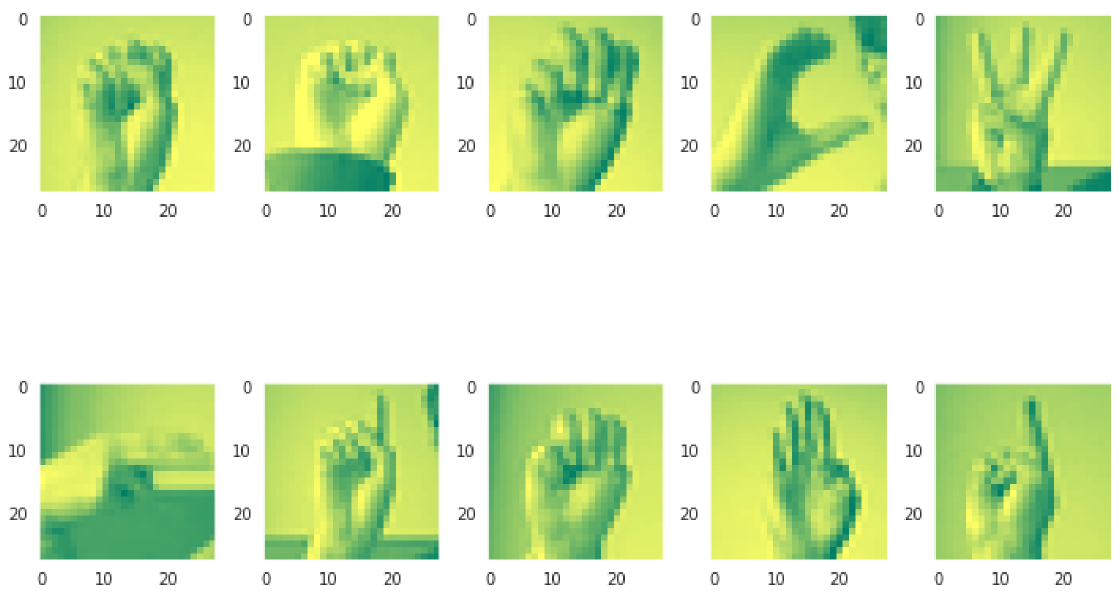
We can also take a look at some examples of pictures in the data.

```
[ ]: plt.imshow(x_train[0].reshape(28,28))
```

```
[ ]: <matplotlib.image.AxesImage at 0x7f3f52ba56d0>
```



```
[ ]: #Overview of images
     f, ax = plt.subplots(2,5)
     f.set_size_inches(10, 10)
     z = 0
     for i in range(2):
         for j in range(5):
             ax[i,j].imshow(x_train[z].reshape(28, 28) , cmap = "summer")
             z += 1
         plt.tight_layout()
```



2 Designing model 1

After exploring and pre-processing the data, we are now ready to start designing our Convolutional Neural Network. Below we are building seven models with different tweaks and changes, aiming to find a model that performs the best in terms of validation accuracy (computed on the test data).

We are starting with a rather basic CNN model with two convolutional layers (`tf.keras.layers.Conv2D`) and 128 neurons in the output layer (`tf.keras.layers.Dense`). The nodes in the output layer is equal to the number of labels (y) and hence the number of signs/letters.

Apart from this we apply the keras layer `tf.keras.layers.MaxPooling2D` to the network. Max pooling reduces dimensionality in our data, which also helps to control overfitting. It also makes feature detection scale and orientation invariant. (See Deep Learning Book, chapter 9.3)

We use ReLU as activation function in the hidden layers and softmax in the ouput layer, due to that we want to perform a multiclass classification.

Below the code block, there is a printed summary of the model to show a better overview of the structure of the model in terms of number of parameters and in which layer these are found.

[ ]:
```
#Design network: setting and building the model with 128-neuron (first try)
model = tf.keras.models.Sequential([
    # Layer 1
    tf.keras.layers.Conv2D(32, (3,3), padding = 'same', input_shape = x_train[0].
 ↪shape, activation = 'relu'),
    tf.keras.layers.MaxPooling2D((2,2), strides=2),
    # Layer 2
    tf.keras.layers.Conv2D(64, (3,3), padding = 'same', activation = 'relu'),
    tf.keras.layers.MaxPooling2D((2,2), strides=2),
    tf.keras.layers.Flatten(),
    # Output layer
    tf.keras.layers.Dense(128, activation = 'relu'),
    tf.keras.layers.Dense(num_labels, activation = 'softmax')
])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 32)        320

 max_pooling2d (MaxPooling2D  (None, 14, 14, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 14, 14, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 7, 7, 64)         0
 2D)

 flatten (Flatten)           (None, 3136)              0

 dense (Dense)               (None, 128)               401536

 dense_1 (Dense)             (None, 24)                3096

=================================================================
Total params: 423,448
Trainable params: 423,448
Non-trainable params: 0
_________________________________________________________________
```

2.1 Compiling the model

Before training the model it needs to be compiled. Here we are setting the optimizer to 'adam', the loss funciton to 'categorical_crossentropy' (which is used for multiclass classification) and the metrics to 'accuracy'. For the latter we use 'accuracy' in order to monitor the extent of which the model predicts images correctly, measured in percentage.

```
[ ]: model.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics =␣
      ↪['accuracy'])
```

2.2 Training the model

In the next step, we train the model by calling `.fit()` which iterate over the entire set of data for a given number of epochs. We have chosen 10 epochs to start off with, however this is a hyperparameter of gradient descent so it might need to be tuned. In general, using more than ten epochs will improve the accuracy since the model then has more iterations to adjust the parameters, and hence, to learn from the data. However, adding epochs is computationally costly so there is always a question of balance between performance and cost.

The full test data is used as validation data, in order to monitor validation loss at the end of each epoch.

```
[ ]: history = model.fit(
         x_train,
         y_train,
         epochs = 10,
         validation_data = (x_test, y_test))
```

```
Epoch 10/10
858/858 [==============================] - 39s 46ms/step - loss: 4.3517e-05 -
accuracy: 1.0000 - val_loss: 0.4256 - val_accuracy: 0.9271
```

2.3 Evaluating the model

By then calling `.evaluate()` we can see how well the model performed on the test data in terms of loss and accuracy. As stated below, the validation accuracy is 92.71 %.

```
[ ]: print("The accuracy of the model is - " , model.evaluate(x_test,y_test)[1]*100 ,␣
      ↪"%")
```

```
225/225 [==============================] - 3s 12ms/step - loss: 0.4256 -
accuracy: 0.9271
The accuracy of the model is -  92.70775318145752 %
```

2.4 Plotting accuracy and loss

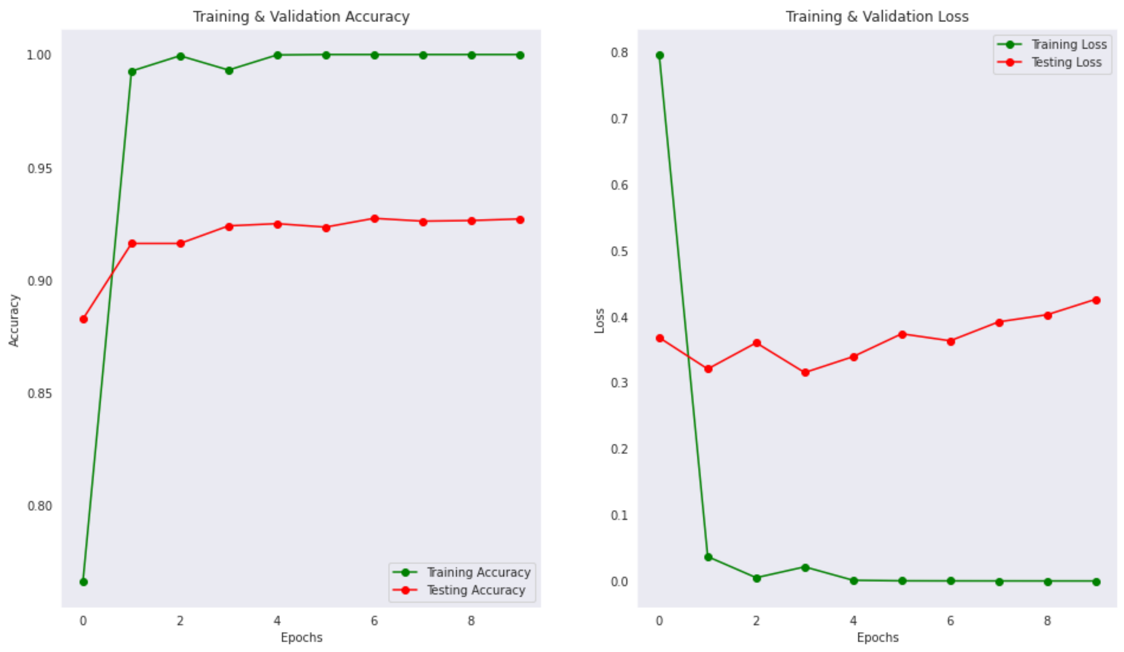
We can now plot the training and validation accuracy (left) as well as the training and validation loss (right) to create a visual presentation of how it evolves over the number of epochs.

```
[ ]: epochs = [i for i in range(10)]
     fig , ax = plt.subplots(1,2)
     train_acc = history.history['accuracy']
     train_loss = history.history['loss']
     val_acc = history.history['val_accuracy']
     val_loss = history.history['val_loss']
     fig.set_size_inches(16,9)

     ax[0].plot(epochs , train_acc , 'go-' , label = 'Training Accuracy')
     ax[0].plot(epochs , val_acc , 'ro-' , label = 'Testing Accuracy')
     ax[0].set_title('Training & Validation Accuracy')
```

```
ax[0].legend()
ax[0].set_xlabel("Epochs")
ax[0].set_ylabel("Accuracy")

ax[1].plot(epochs , train_loss , 'g-o' , label = 'Training Loss')
ax[1].plot(epochs , val_loss , 'r-o' , label = 'Testing Loss')
ax[1].set_title('Training & Validation Loss')
ax[1].legend()
ax[1].set_xlabel("Epochs")
ax[1].set_ylabel("Loss")
plt.show()
```



2.5 Overfitting

Looking at the graphs above, we can see that the model is clearly overfitting. The testing accuracy is well below the training accuracy. When looking at the loss, we can see a huge difference. The loss in the test data is over 35 % while the training loss is close to 0. We need to makes some changes in our model in order to combat this problem.

3 Improving the model

We have now seen that we have an issue regarding overfitting in our model. We thus need to make some changes to our model to combat this issue and also to improve the validation accuracy. In this section we will try to improve the model by adding a batch size, increasing the number of neurons, adding a batch normalization layer, adjusting the learning rate, using the regularization technique "dropout", data augmentation, increasing the number of epochs and finally by redesigning the model.

3.1 Batch size

In order to try to prevent overfitting and hence improve the validation accuracy, we specify a `batch_size` inside the `.fit()` call. This is a hyperparameter of gradient descent, splitting the data into a set of mini-batches and regulates the number of training samples to iterate over before updating the model parameters. This is not to confuse with epochs, which controls the number of **complete** passes through the training dataset.

Popular batch sizes include samples of 32, 64 and 128. To begin with, we are trying with a batch size of 32:

```
[ ]: model.fit(x_train, y_train, epochs = 10, batch_size = 32, validation_data =␣
      ↪(x_test, y_test))
```

```
Epoch 10/10
858/858 [==============================] - 38s 45ms/step - loss: 0.0220 -
accuracy: 0.9936 - val_loss: 0.3726 - val_accuracy: 0.9202
```

```
[ ]: <keras.callbacks.History at 0x7f3f4bdc00d0>
```

```
[ ]: print("The accuracy of the model is - " , model.evaluate(x_test,y_test)[1]*100 ,␣
      ↪"%")
```

```
225/225 [==============================] - 3s 12ms/step - loss: 0.3726 -
accuracy: 0.9202
The accuracy of the model is -  92.02454090118408 %
```

Since specifying a batch size of 32 slightly improves the accuracy from 92.71% to 92.02%, there was no significant difference. Thus, we are increasing it to the size of 64 to see the performance:

```
[ ]: model.fit(x_train, y_train, epochs = 10, batch_size = 64, validation_data =␣
      ↪(x_test, y_test))
```

```
Epoch 10/10
429/429 [==============================] - 37s 86ms/step - loss: 1.1062e-05 -
accuracy: 1.0000 - val_loss: 0.3873 - val_accuracy: 0.9340
```

```
[ ]: <keras.callbacks.History at 0x7f3f4bd27750>
```

```
[ ]: print("The accuracy of the model is - " , model.evaluate(x_test,y_test)[1]*100 ,␣
      ↪"%")
```

```
225/225 [==============================] - 3s 12ms/step - loss: 0.3873 -
accuracy: 0.9340
The accuracy of the model is -  93.40490698814392 %
```

This gave some improvement! The validation accuracy increased from 92.02 % to 93.40 %. Let's try with a batch size of 128:

```
[ ]: model.fit(x_train, y_train, epochs = 10, batch_size = 128, validation_data =␣
      ↪(x_test, y_test))
```

```
Epoch 10/10
215/215 [==============================] - 32s 150ms/step - loss: 3.6396e-06 -
accuracy: 1.0000 - val_loss: 0.4149 - val_accuracy: 0.9342
```

```
[ ]: <keras.callbacks.History at 0x7f3f4bc97f50>
```

3.1.1 Conclusion: Batch size

When trying with three different batch sizes (32, 64 and 128), a batch size of 128 generated the best validation accuracy of 93.42 %. We will hence use this setting when moving forward with building the model.

3.2 Increasing the number of neurons

We will now build a similar CNN model, however with an increased number of neurons in the output layer. We are now setting them to the maximum value of 512 in order to increase the depth of the neural network. We are then repeating the same steps as for the previous model in order to obtain the validation accuracy.

```
[ ]: model1 = tf.keras.models.Sequential([
         # Layer 1
         tf.keras.layers.Conv2D(32, (3,3), padding = 'same', input_shape = x_train[0].
      ↪shape, activation = 'relu'),
         tf.keras.layers.MaxPooling2D((2,2), strides=2),
         # Layer 2
         tf.keras.layers.Conv2D(64, (3,3), padding = 'same', activation = 'relu'),
         tf.keras.layers.MaxPooling2D((2,2), strides=2),
         tf.keras.layers.Flatten(),
         # Output layer
         tf.keras.layers.Dense(512, activation = 'relu'),
         tf.keras.layers.Dense(num_labels, activation = 'softmax')
     ])
     model1.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics =␣
      ↪['accuracy'])
     model1.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)           (None, 28, 28, 32)        320

 max_pooling2d_2 (MaxPooling  (None, 14, 14, 32)       0
 2D)

 conv2d_3 (Conv2D)           (None, 14, 14, 64)        18496
 max_pooling2d_3 (MaxPooling  (None, 7, 7, 64)         0
 2D)

 flatten_1 (Flatten)         (None, 3136)              0

 dense_2 (Dense)             (None, 512)               1606144

 dense_3 (Dense)             (None, 24)                12312

=================================================================
Total params: 1,637,272
Trainable params: 1,637,272
Non-trainable params: 0
_________________________________________________________________
```

```
[ ]: history1 = model1.fit(x_train, y_train, epochs = 10, batch_size = 128,␣
      ↪validation_data = (x_test, y_test))
```

```
Epoch 10/10
215/215 [==============================] - 40s 187ms/step - loss: 3.8265e-05 -
accuracy: 1.0000 - val_loss: 0.5015 - val_accuracy: 0.9017
```

```
[ ]: print("Accuracy of the model 1 is - " , model1.evaluate(x_test,y_test)[1]*100 ,␣
      ↪"%")
```

```
225/225 [==============================] - 6s 28ms/step - loss: 0.5015 -
accuracy: 0.9017
Accuracy of the model 1 is -  90.1701033115387 %
```

3.2.1 Conclusion: Increasing neurons

Increasing the neurons from 128 to 512 does not result in a higher validation accuracy. However, it might increase only the depth of the neural network. We noticed that the result remains high although it appears to be overfitting. We will keep this setting when proceeding with the next model when applying regularization methods to prevent overfitting.

3.3 Adding Batch Normalization layer and controlling learning rate

In the following model, we are using the keras layer `tf.keras.layers.BatchNormalization()` to our existing convolutional layers. This normalizes the inputs by applying a transformation that will keep the mean output close to zero and the output standard deviation close to 1. Batch normalization can aid the optimization process and increase prediction performance according to **Kaggle.com.**

Additionally, before training the model, we are specifying a reduced learning rate in order to keep the model from learning patterns in the data too quickly, and hence, to prevent overfitting. The reduced learning rate is added into the `.fit()` call using the `callbacks` input.

Again, we are repeating the steps of compiling, training and evaluating the model in order to obtain the validation accuracy.

```
[ ]: model2 = tf.keras.models.Sequential([
         # Layer 1
         tf.keras.layers.Conv2D(32, (3,3), padding = 'same', input_shape = x_train[0].
      ↪shape, activation = 'relu'),
         tf.keras.layers.BatchNormalization(),
         tf.keras.layers.MaxPooling2D((2,2), strides=2),
         # Layer 2
         tf.keras.layers.Conv2D(64, (3,3), padding = 'same', activation = 'relu'),
         tf.keras.layers.BatchNormalization(),
         tf.keras.layers.MaxPooling2D((2,2), strides=2),
         tf.keras.layers.Flatten(),
         # Output layer
         tf.keras.layers.Dense(512, activation = 'relu'),
         tf.keras.layers.Dense(num_labels, activation = 'softmax')
     ])
     model2.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics =␣
      ↪['accuracy'])
     model2.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)           (None, 28, 28, 32)        320

 batch_normalization (BatchN  (None, 28, 28, 32)       128
 ormalization)

 max_pooling2d_4 (MaxPooling  (None, 14, 14, 32)       0
 2D)

 conv2d_5 (Conv2D)           (None, 14, 14, 64)        18496

 batch_normalization_1 (Batc  (None, 14, 14, 64)       256
 hNormalization)

 max_pooling2d_5 (MaxPooling  (None, 7, 7, 64)         0
 2D)

 flatten_2 (Flatten)         (None, 3136)              0

 dense_4 (Dense)             (None, 512)               1606144

 dense_5 (Dense)             (None, 24)                12312
```

```
=================================================================
Total params: 1,637,656
Trainable params: 1,637,464
Non-trainable params: 192
_________________________________________________________________
```

Before applying the learning rate, we observed the performance without the learning but applied only the batch normalization in the network in order to first see what this does to the performance.

```
[ ]: history2_nolr = model2.fit(x_train, y_train, epochs = 10, batch_size = 128,␣
      ↪validation_data = (x_test, y_test)) #93.43
```

```
Epoch 10/10
215/215 [==============================] - 55s 254ms/step - loss: 2.1492e-05 -
accuracy: 1.0000 - val_loss: 0.3303 - val_accuracy: 0.9388
```

The result appears that there was some improvement which next we will apply the learning rate to see whether the accuracy can be enhanced.

Reduce learning rate is applied to assist a metric has stopped improving. This part we experiment by monitoring the validation accuracy.

```
[ ]: learning_rate_reduction = ReduceLROnPlateau(monitor='val_accuracy', patience =␣
      ↪2, verbose=1,factor=0.5, min_lr=0.00001)
```

```
[ ]: history2 = model2.fit(x_train, y_train, epochs = 10, batch_size = 128,␣
      ↪validation_data = (x_test, y_test), callbacks = [learning_rate_reduction])
```

```
Epoch 10/10
215/215 [==============================] - 56s 259ms/step - loss: 3.7831e-06 -
accuracy: 1.0000 - val_loss: 0.3761 - val_accuracy: 0.9403 - lr: 5.0000e-04
```

```
[ ]: print("Accuracy of the model 2 is - " , model2.evaluate(x_test,y_test)[1]*100 ,␣
      ↪"%")
```

```
225/225 [==============================] - 5s 23ms/step - loss: 0.3761 -
accuracy: 0.9403
Accuracy of the model 2 is -  94.03234720230103 %
```

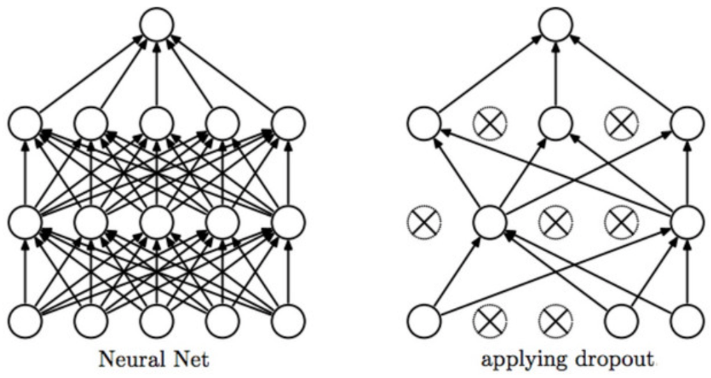
3.3.1 Conclusion: Learning rate

It turns out that the decreased learning rate is improving the performance to the model providing the accuracy at 94.03%.

3.4 Dropout (Regurarization)

Overfitting is an unneglectable problem in deep CNNs, which can be effectively reduced by regularization. In this section, we introduce an effective regularization technique: Dropout.

For our next model, we will try to improve the validation accuracy further by adding the keras layer `tf.keras.layers.Dropout()`. Dropout is a regularization technique to handle overfitting, where random neurons are “ignored” during the training process. As a result, the other neurons have to make up for the (temporarily) missing neurons, which makes the network become less sensitive to the specific weights of neurons. In this case, dropout is implemented by randomly selecting nodes to be dropped-out with a given probability of 20% each weight update cycle.



[ ]:
```
model3 = tf.keras.models.Sequential([
    # Layer 1
    tf.keras.layers.Conv2D(32, (3,3), padding = 'same', input_shape = x_train[0].
 ↪shape, activation = 'relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D((2,2), strides=2),
    # Layer 2
    tf.keras.layers.Conv2D(64, (3,3), padding = 'same', activation = 'relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D((2,2), strides=2),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Flatten(),
    # Output layer
    tf.keras.layers.Dense(512, activation = 'relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(num_labels, activation = 'softmax')
])
model3.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics =␣
 ↪['accuracy'])
model3.summary()
```

```
Model: "sequential_5"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_10 (Conv2D)          (None, 28, 28, 32)        320


 batch_normalization_6 (Batc  (None, 28, 28, 32)       128
 hNormalization)
 max_pooling2d_10 (MaxPoolin  (None, 14, 14, 32)       0
 g2D)

 conv2d_11 (Conv2D)          (None, 14, 14, 64)        18496

 batch_normalization_7 (Batc  (None, 14, 14, 64)       256
 hNormalization)

 max_pooling2d_11 (MaxPoolin  (None, 7, 7, 64)         0
 g2D)

 dropout_2 (Dropout)         (None, 7, 7, 64)          0

 flatten_5 (Flatten)         (None, 3136)              0

 dense_10 (Dense)            (None, 512)               1606144

 dropout_3 (Dropout)         (None, 512)               0

 dense_11 (Dense)            (None, 24)                12312

=================================================================
Total params: 1,637,656
Trainable params: 1,637,464
Non-trainable params: 192
_________________________________________________________________
```

[ ]:
```
history3 = model3.fit(x_train, y_train, epochs = 10, batch_size = 128,␣
 ↪validation_data = (x_test, y_test), callbacks = [learning_rate_reduction])
```

```
Epoch 10/10
215/215 [==============================] - 54s 252ms/step - loss: 2.3139e-04 -
accuracy: 1.0000 - val_loss: 0.3328 - val_accuracy: 0.9374 - lr: 5.0000e-04
```

[ ]:
```
print("Accuracy of the model 3 is - " , model3.evaluate(x_test,y_test)[1]*100 ,␣
 ↪"%")
```

```
225/225 [==============================] - 4s 19ms/step - loss: 0.3328 -
accuracy: 0.9374
Accuracy of the model 3 is -  93.7395453453064 %
```

3.4.1 Conclusion: Drop out

We can see that using the dropout technique on the output layer is not improving the validation accuracy. However, we will keep this regularization in our model and apply the optimization method data augmentation to see if we can arrive at any improvement.

3.5 Data Augmentation (Optimization)

Data augmentation consists in transforming the available data into new data without altering their nature. Data augmentation is particularly effective for object recognition.

As sometimes, the Neural Network model may memorize the images in the training set and may not be able to recognize the input of the test set. Data augmentation is one of the methods to prevent the overfitting as it can transform by many features such as sampling, mirroring, flipping, translation, rotation, changing brightness, adding noises etc. to make the images more generalized as some new images. As a result, our model can learn to understand the images from the training set and apply to any new sets rather than memorizing.

```
[ ]: datagen = tf.keras.preprocessing.image.ImageDataGenerator(
                 featurewise_center=False,
                 samplewise_center=False,
                 featurewise_std_normalization=False,
                 samplewise_std_normalization=False,
                 zca_whitening=False,
                 rotation_range= 10,
                 width_shift_range=0.0,
                 height_shift_range=0.0,
                 brightness_range=None,
                 shear_range=0.0,
                 zoom_range=0.1,
                 fill_mode='nearest',
                 horizontal_flip=False, vertical_flip=False
     )

     datagen.fit(x_train)
```

```
datagen = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0
over the dataset
        samplewise_center=False,  # set each sample
mean to 0
        featurewise_std_normalization=False,  # divide
inputs by std of the dataset
        samplewise_std_normalization=False,  # divide
each input by its std
        zca_whitening=False,  # apply ZCA whitening
        rotation_range=10,  # randomly rotate images in
the range (degrees, 0 to 180)
        zoom_range = 0.1, # Randomly zoom image
        width_shift_range=0.1,  # randomly shift images
horizontally (fraction of total width)
        height_shift_range=0.1,  # randomly shift images
vertically (fraction of total height)
        horizontal_flip=False,  # randomly flip images
        vertical_flip=False)  # randomly flip images
```

```
[ ]: model4 = tf.keras.models.Sequential([
         # Layer 1
         tf.keras.layers.Conv2D(32, (3,3), padding = 'same', input_shape = x_train[0].
      ↪shape, activation = 'relu'),
         tf.keras.layers.BatchNormalization(),
         tf.keras.layers.MaxPooling2D((2,2), strides=2),
         # Layer 2
         tf.keras.layers.Conv2D(64, (3,3), padding = 'same', activation = 'relu'),
         tf.keras.layers.BatchNormalization(),
         tf.keras.layers.MaxPooling2D((2,2), strides=2),
         tf.keras.layers.Dropout(0.2),
         tf.keras.layers.Flatten(),
         # Output layer
         tf.keras.layers.Dense(512, activation = 'relu'),
         tf.keras.layers.Dropout(0.2),
         tf.keras.layers.Dense(num_labels, activation = 'softmax')
     ])
```

```
model4.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics =␣
 ↪['accuracy'])
model4.summary()
```

```
Model: "sequential_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_14 (Conv2D)          (None, 28, 28, 32)        320

 batch_normalization_10 (Bat  (None, 28, 28, 32)       128
 chNormalization)

 max_pooling2d_14 (MaxPoolin  (None, 14, 14, 32)       0
 g2D)

 conv2d_15 (Conv2D)          (None, 14, 14, 64)        18496

 batch_normalization_11 (Bat  (None, 14, 14, 64)       256
 chNormalization)

 max_pooling2d_15 (MaxPoolin  (None, 7, 7, 64)         0
 g2D)

 dropout_5 (Dropout)         (None, 7, 7, 64)          0

 flatten_7 (Flatten)         (None, 3136)              0

 dense_14 (Dense)            (None, 512)               1606144

 dropout_6 (Dropout)         (None, 512)               0

 dense_15 (Dense)            (None, 24)                12312

=================================================================
Total params: 1,637,656
Trainable params: 1,637,464
Non-trainable params: 192
_________________________________________________________________
```

[ ]:
```
history4 = model4.fit(datagen.flow(x_train,y_train, batch_size = 128), epochs =␣
 ↪10, validation_data = (x_test, y_test), callbacks = [learning_rate_reduction])
```

```
Epoch 10: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
215/215 [==============================] - 60s 278ms/step - loss: 0.0027 -
accuracy: 0.9994 - val_loss: 0.2261 - val_accuracy: 0.9564 - lr: 5.0000e-04
```

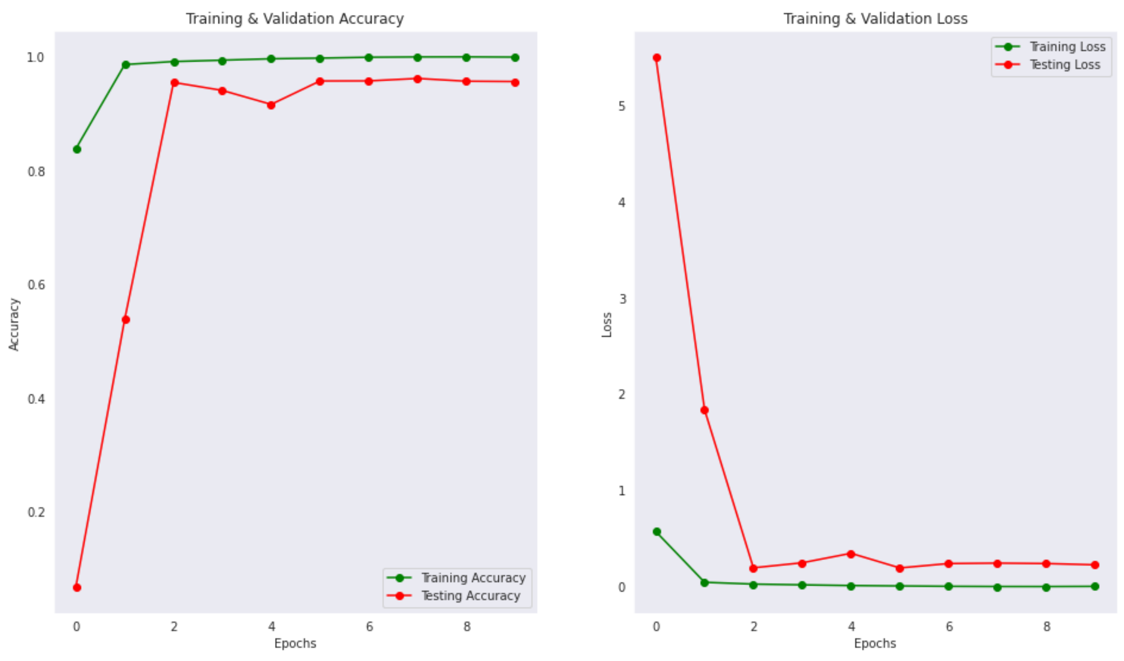
```
[ ]: print("Accuracy of the model 4 is - " , model4.evaluate(x_test,y_test)[1]*100 ,␣
     ↪"%")
```

```
225/225 [==============================] - 5s 22ms/step - loss: 0.2261 -
accuracy: 0.9564
Accuracy of the model 4 is -  95.63580751419067 %
```

```
[ ]: epochs = [i for i in range(10)]
     fig , ax = plt.subplots(1,2)
     train_acc = history4.history['accuracy']
     train_loss = history4.history['loss']
     val_acc = history4.history['val_accuracy']
     val_loss = history4.history['val_loss']
     fig.set_size_inches(16,9)

     ax[0].plot(epochs , train_acc , 'go-' , label = 'Training Accuracy')
     ax[0].plot(epochs , val_acc , 'ro-' , label = 'Testing Accuracy')
     ax[0].set_title('Training & Validation Accuracy')
     ax[0].legend()
     ax[0].set_xlabel("Epochs")
     ax[0].set_ylabel("Accuracy")

     ax[1].plot(epochs , train_loss , 'g-o' , label = 'Training Loss')
     ax[1].plot(epochs , val_loss , 'r-o' , label = 'Testing Loss')
     ax[1].set_title('Training & Validation Loss')
     ax[1].legend()
     ax[1].set_xlabel("Epochs")
     ax[1].set_ylabel("Loss")
     plt.show()
```



3.5.1 Conclusion: Data augmentation

As a result of applying data augmentation, the accuracy is further improved. Hence, we will keep this setting when building the next model.

3.6 Adjusting the Learning Rate

Next, we aim to monitor the loss rather than the accuracy and set the mode as min so that the learning rate is reduced when the quantity of loss monitored has stopped, min_delta for the threshold for measuring the new optimum and focus on purely significant changes, verbose to 1 to update the messages, factor by which the learning rate will be reduced by and the min_lr to set the lower bound on the learning rate.

With these changes, we aim to see the increase of performance accuracy.

```
[ ]: #Change the learning rate to monitor loss
     learning_rate_reduction_new = ReduceLROnPlateau(monitor='loss', mode = min,␣
      ↪min_delta=0.01, patience = 2, verbose=1, factor=0.5, min_lr=0.00001)
```

```
[ ]: #Test with new adjustment of learning rate (new) and add drop out
     model5 = tf.keras.models.Sequential([
         # Layer 1
         tf.keras.layers.Conv2D(32, (3,3), padding = 'same', input_shape = x_train[0].
      ↪shape, activation = 'relu'),
         tf.keras.layers.BatchNormalization(),
         tf.keras.layers.MaxPooling2D((2,2), strides=2),
         tf.keras.layers.Dropout(0.2),
         # Layer 2
         tf.keras.layers.Conv2D(64, (3,3), padding = 'same', activation = 'relu'),
         tf.keras.layers.BatchNormalization(),
         tf.keras.layers.MaxPooling2D((2,2), strides=2),
         tf.keras.layers.Dropout(0.2),
         tf.keras.layers.Flatten(),
         # Output layer
         tf.keras.layers.Dense(512, activation = 'relu'),
         tf.keras.layers.Dropout(0.2),
         tf.keras.layers.Dense(num_labels, activation = 'softmax')
     ])
     model5.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics =␣
      ↪['accuracy'])
     model5.summary()
```

```
Model: "sequential_8"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_16 (Conv2D)          (None, 28, 28, 32)        320

 batch_normalization_12 (Bat  (None, 28, 28, 32)       128
 chNormalization)

 max_pooling2d_16 (MaxPoolin  (None, 14, 14, 32)       0
 g2D)

 dropout_7 (Dropout)         (None, 14, 14, 32)        0

 conv2d_17 (Conv2D)          (None, 14, 14, 64)        18496

 batch_normalization_13 (Bat  (None, 14, 14, 64)       256
 chNormalization)

 max_pooling2d_17 (MaxPoolin  (None, 7, 7, 64)         0
 g2D)

 dropout_8 (Dropout)         (None, 7, 7, 64)          0

 flatten_8 (Flatten)         (None, 3136)              0

 dense_16 (Dense)            (None, 512)               1606144

 dropout_9 (Dropout)         (None, 512)               0

 dense_17 (Dense)            (None, 24)                12312

=================================================================
Total params: 1,637,656
Trainable params: 1,637,464
Non-trainable params: 192
_________________________________________________________________
```

```
[ ]: history5 = model5.fit(datagen.flow(x_train,y_train, batch_size = 128), epochs =
     ↪10, validation_data = (x_test, y_test), callbacks =
     ↪[learning_rate_reduction_new])
```

```
Epoch 10/10
215/215 [==============================] - 62s 288ms/step - loss: 0.0043 -
accuracy: 0.9987 - val_loss: 0.1436 - val_accuracy: 0.9644 - lr: 5.0000e-04
```

```
[ ]: print("Accuracy of the model 5 is - " , model5.evaluate(x_test,y_test)[1]*100 ,
     ↪"%")
```

```
225/225 [==============================] - 4s 18ms/step - loss: 0.1436 -
accuracy: 0.9644
Accuracy of the model 5 is -  96.44450545310974 %
```

3.6.1 Conclusion: New Reduced Learning Rate

As a result of applying the new reduced learning rate, the accuracy is further improved. Hence, we will keep this setting when building the next model.

3.7 Increasing the number of Epochs

After having tried different methods and managed to increase the validation accuracy, we will now increase the number of epochs to see if this can improve the accuracy further. With more epochs, the model has more iterations to adjust the parameters and learn from the data. Since adding more epochs is costly, we have chosen to use a lower number of epochs up until now which has given us time to focus on adjusting the model.

Note that too many epochs could cause overfitting, thus we should not increase the number too much.

```
[ ]: # Adding epochs to 20 to see the performance of accuracy
model6 = tf.keras.models.Sequential([
    # Layer 1
    tf.keras.layers.Conv2D(32, (3,3), padding = 'same', input_shape = x_train[0].
  ↪shape, activation = 'relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D((2,2), strides=2),
    tf.keras.layers.Dropout(0.2),
    # Layer 2
    tf.keras.layers.Conv2D(64, (3,3), padding = 'same', activation = 'relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D((2,2), strides=2),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Flatten(),
    # Output layer
    tf.keras.layers.Dense(512, activation = 'relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(num_labels, activation = 'softmax')
])
model6.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics =␣
  ↪['accuracy'])
model6.summary()
```

```
Model: "sequential_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_18 (Conv2D)          (None, 28, 28, 32)        320


 batch_normalization_14 (Bat  (None, 28, 28, 32)       128
 chNormalization)


 max_pooling2d_18 (MaxPoolin  (None, 14, 14, 32)       0
 g2D)

 dropout_10 (Dropout)        (None, 14, 14, 32)        0

 conv2d_19 (Conv2D)          (None, 14, 14, 64)        18496

 batch_normalization_15 (Bat  (None, 14, 14, 64)       256
 chNormalization)

 max_pooling2d_19 (MaxPoolin  (None, 7, 7, 64)         0
 g2D)

 dropout_11 (Dropout)        (None, 7, 7, 64)          0

 flatten_9 (Flatten)         (None, 3136)              0

 dense_18 (Dense)            (None, 512)               1606144

 dropout_12 (Dropout)        (None, 512)               0

 dense_19 (Dense)            (None, 24)                12312

=================================================================
Total params: 1,637,656
Trainable params: 1,637,464
Non-trainable params: 192
_________________________________________________________________
```

[ ]:
```
history6 = model6.fit(datagen.flow(x_train,y_train, batch_size = 128), epochs =␣
 ↪20, validation_data = (x_test, y_test), callbacks =␣
 ↪[learning_rate_reduction_new])
```

```
Epoch 20/20
215/215 [==============================] - 63s 291ms/step - loss: 0.0013 -
accuracy: 0.9996 - val_loss: 0.2035 - val_accuracy: 0.9668 - lr: 1.0000e-05
```

[ ]:
```
print("Accuracy of the model 6 is - " , model6.evaluate(x_test,y_test)[1]*100 ,␣
 ↪"%")
```

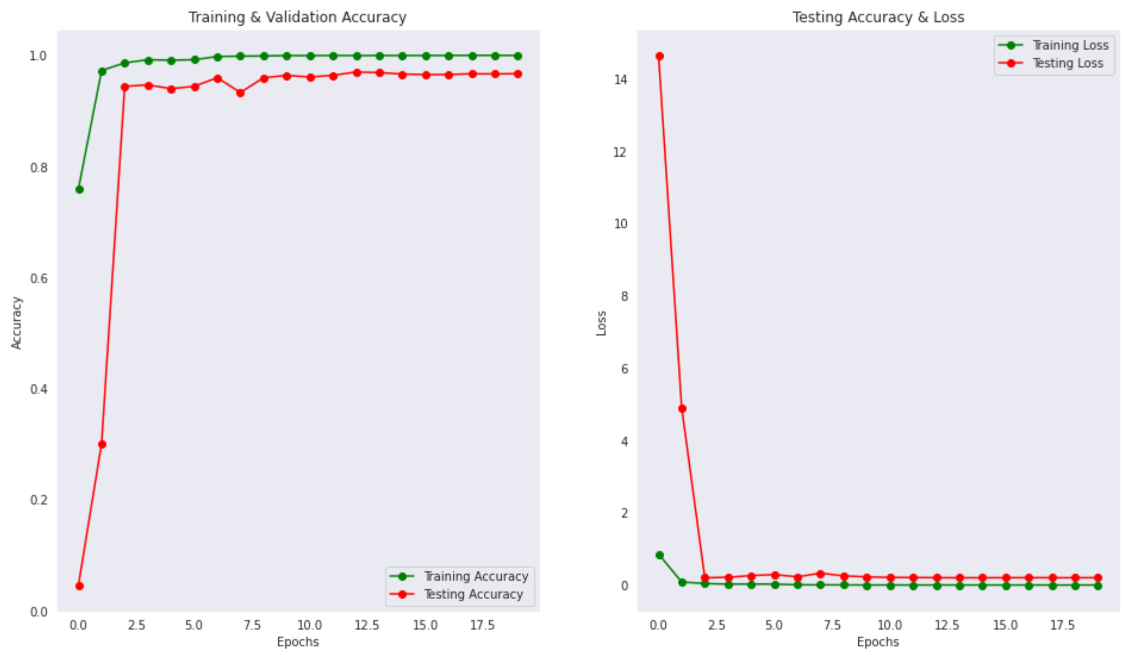
```
225/225 [==============================] - 7s 30ms/step - loss: 0.2035 -
accuracy: 0.9668
Accuracy of the model 6 is -  96.68154120445251 %
```

[ ]:
```
epochs = [i for i in range(20)]
fig , ax = plt.subplots(1,2)
train_acc = history6.history['accuracy']
train_loss = history6.history['loss']
val_acc = history6.history['val_accuracy']
```

```
val_loss = history6.history['val_loss']
fig.set_size_inches(16,9)

ax[0].plot(epochs , train_acc , 'go-' , label = 'Training Accuracy')
ax[0].plot(epochs , val_acc , 'ro-' , label = 'Testing Accuracy')
ax[0].set_title('Training & Validation Accuracy')
ax[0].legend()
ax[0].set_xlabel("Epochs")
ax[0].set_ylabel("Accuracy")

ax[1].plot(epochs , train_loss , 'g-o' , label = 'Training Loss')
ax[1].plot(epochs , val_loss , 'r-o' , label = 'Testing Loss')
ax[1].set_title('Testing Accuracy & Loss')
ax[1].legend()
ax[1].set_xlabel("Epochs")
ax[1].set_ylabel("Loss")
plt.show()
```



3.7.1 Conclusion: Number of Epochs

As expected, increasing the number of epochs resulted in an improved accuracy (quite significantly even).

3.8 Redesigning the model

In our next model, we are redesigning the CNN model by adding another convolutional layer in order to, again, try to improve the accuracy. As with epochs, adding another layer is costly, and thus, we have chosen to work with a simpler model up until now in order to save time.

We have experimented with different amounts of layers and filters in the model, and found that the following model gives the best result.

```
[ ]: # Redesign to see the performance of accuracy
     model7 = tf.keras.models.Sequential([
         # Layer 1
         tf.keras.layers.Conv2D(75, (3,3), padding = 'same', input_shape = x_train[0].
      ↪shape, activation = 'relu'),
         tf.keras.layers.BatchNormalization(),
         tf.keras.layers.MaxPooling2D((2,2), strides=2),
         # Layer 2
         tf.keras.layers.Conv2D(50 , (3,3), padding = 'same' , activation = 'relu'),
         tf.keras.layers.Dropout(0.2),
         tf.keras.layers.MaxPooling2D((2,2), strides=2), #added
         # Layer 3
         tf.keras.layers.Conv2D(25, (3,3), padding = 'same', activation = 'relu'),
         tf.keras.layers.BatchNormalization(),
         tf.keras.layers.MaxPooling2D((2,2), strides=2),
         tf.keras.layers.Dropout(0.2),
         tf.keras.layers.Flatten(),
         # Output layer
         tf.keras.layers.Dense(512, activation = 'relu'),
         tf.keras.layers.Dropout(0.2),
         tf.keras.layers.Dense(num_labels, activation = 'softmax')
     ])
     model7.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics =␣
      ↪['accuracy'])
     model7.summary()
```

```
Model: "sequential_10"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_20 (Conv2D)          (None, 28, 28, 75)        750

 batch_normalization_16 (Bat  (None, 28, 28, 75)       300
 chNormalization)

 max_pooling2d_20 (MaxPoolin  (None, 14, 14, 75)       0
 g2D)

 conv2d_21 (Conv2D)          (None, 14, 14, 50)        33800
 dropout_13 (Dropout)         (None, 14, 14, 50)        0

 max_pooling2d_21 (MaxPoolin  (None, 7, 7, 50)          0
 g2D)

 conv2d_22 (Conv2D)           (None, 7, 7, 25)          11275

 batch_normalization_17 (Bat  (None, 7, 7, 25)          100
 chNormalization)

 max_pooling2d_22 (MaxPoolin  (None, 3, 3, 25)          0
 g2D)

 dropout_14 (Dropout)         (None, 3, 3, 25)          0

 flatten_10 (Flatten)         (None, 225)               0

 dense_20 (Dense)             (None, 512)               115712

 dropout_15 (Dropout)         (None, 512)               0

 dense_21 (Dense)             (None, 24)                12312

=================================================================
Total params: 174,249
Trainable params: 174,049
Non-trainable params: 200
_________________________________________________________________
```

[ ]:
```
history7 = model7.fit(datagen.flow(x_train,y_train, batch_size = 128), epochs =␣
 ↪20, validation_data = (x_test, y_test), callbacks =␣
 ↪[learning_rate_reduction_new])
```

```
Epoch 20/20
215/215 [==============================] - 111s 515ms/step - loss: 0.0028 -
accuracy: 0.9993 - val_loss: 0.0402 - val_accuracy: 0.9914 - lr: 1.5625e-05
```

[ ]:
```
print("Accuracy of the model 7 is - " , model7.evaluate(x_test,y_test)[1]*100 ,␣
 ↪"%")
```

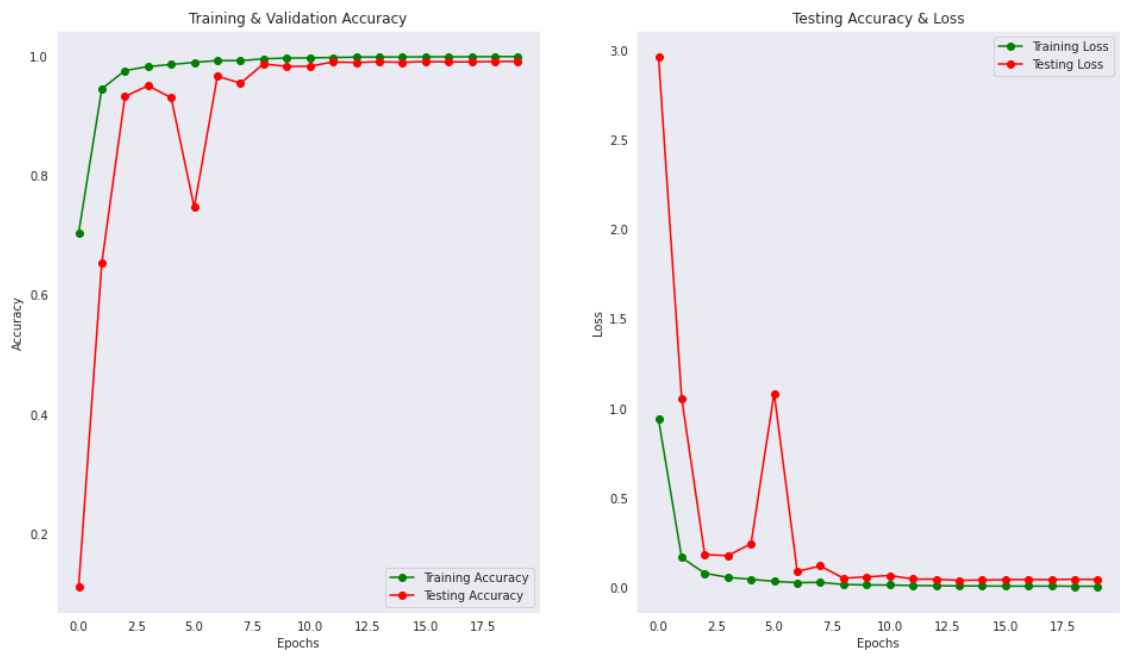
```
225/225 [==============================] - 6s 26ms/step - loss: 0.0402 -
accuracy: 0.9914
Accuracy of the model 7 is -  99.1355299949646 %
```

[ ]:
```
epochs = [i for i in range(20)]
fig , ax = plt.subplots(1,2)
train_acc = history7.history['accuracy']
train_loss = history7.history['loss']
```

```
val_acc = history7.history['val_accuracy']
val_loss = history7.history['val_loss']
fig.set_size_inches(16,9)

ax[0].plot(epochs , train_acc , 'go-' , label = 'Training Accuracy')
ax[0].plot(epochs , val_acc , 'ro-' , label = 'Testing Accuracy')
ax[0].set_title('Training & Validation Accuracy')
ax[0].legend()
ax[0].set_xlabel("Epochs")
ax[0].set_ylabel("Accuracy")

ax[1].plot(epochs , train_loss , 'g-o' , label = 'Training Loss')
ax[1].plot(epochs , val_loss , 'r-o' , label = 'Testing Loss')
ax[1].set_title('Testing Accuracy & Loss')
ax[1].legend()
ax[1].set_xlabel("Epochs")
ax[1].set_ylabel("Loss")
plt.show()
```



3.8.1 Conclusion: Redesigning the model

By adding another layer, the performance in terms of validation accuracy was improved. Since the model now consists of a higher number of parameters (see model summary below) due to the added layer, this was expected.

4 Final Model Evaluation

4.1 Confusion Matrix

In order to evaluate how well our model performs, we construct a confusion matrix. Confusion matrices are used to obtain a summary of prediction results on classification problems. One can observe the number of correct and incorrect predictions with count values and each label is broken down by each label class.

First, we create a new variable called predictions using the np.argmax function to return the indices of the maximum values and obtain an array of new predicted labels based on our selected best model 7.

```
[ ]: predictions = np.argmax(model7.predict(x_test),axis = 1)

     for i in range(len(predictions)):
         if predictions[i] >= 9:
             predictions[i] += 1

     predictions[:5]
```

```
[ ]: array([ 6,  5, 10,  0,  3])
```

Next, we apply the function of `label_binarizer.inverse_transform()` to inverse back the labels in y test for inputting into a confusion matrix.

```
[ ]: y_test_inv_label = label_binarizer.inverse_transform(y_test) #Inverse back
     y_test_inv_label
```

```
[ ]: array([ 6,  5, 10, ...,  2,  4,  2])
```

```
[ ]: confusion_matrix(predictions, y_test_inv_label)
```

```
[ ]: array([[331,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
            [  0, 432,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
            [  0,   0, 310,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
            [  0,   0,   0, 245,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
            [  0,   0,   0,   0, 498,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
            [  0,   0,   0,   0,   0, 247,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
            [  0,   0,   0,   0,   0,   0, 346,  10,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
            [  0,   0,   0,   0,   0,   0,   0, 426,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,  17,   0,   0,   0,   0,   0],
            [  0,   0,   0,   0,   0,   0,   0,   0, 288,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1],
            [  0,   0,   0,   0,   0,   0,   0,   0,   0, 331,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
            [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 209,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
            [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 394,   0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
            [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 291,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
            [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
             246,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
            [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0, 347,   0,   0,   0,   0,   0,   0,   0,   0,   0],
            [  0,   0,   0,   0,   0,   0,   2,   0,   0,   0,   0,   0,   0,
               0,   0, 164,   0,   0,   0,   0,   0,   0,   0,   0],
            [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0, 144,   0,   0,   0,   0,   0,   0,   0],
            [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0, 246,   0,   0,   0,   0,   0,   0],
            [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0, 199,   0,   0,   0,   0,   0],
            [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0, 266,   0,   0,   0,   0],
            [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0, 346,   0,   0,   0],
            [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0, 206,   0,   0],
            [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,  32,   0,   0,   0, 267,   0],
            [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 331]])
```

[ ]:
```
print("The accuracy from the confusion matrix is - ",␣
↪accuracy_score(y_test_inv_label, predictions)*100, "%")
```

```
The accuracy from the confusion matrix is -  99.13552704963749 %. Next, we can see␣
↪how well the model redicts the different labels/letters in the data. Here we␣
↪can see that the model has a harder time correctyl predicting some of the␣
↪letters.
```

[ ]:
```
#Showing how well the model predicts the different labels/letters

print(classification_report(y_test_inv_label, predictions, target_names=labels))
```

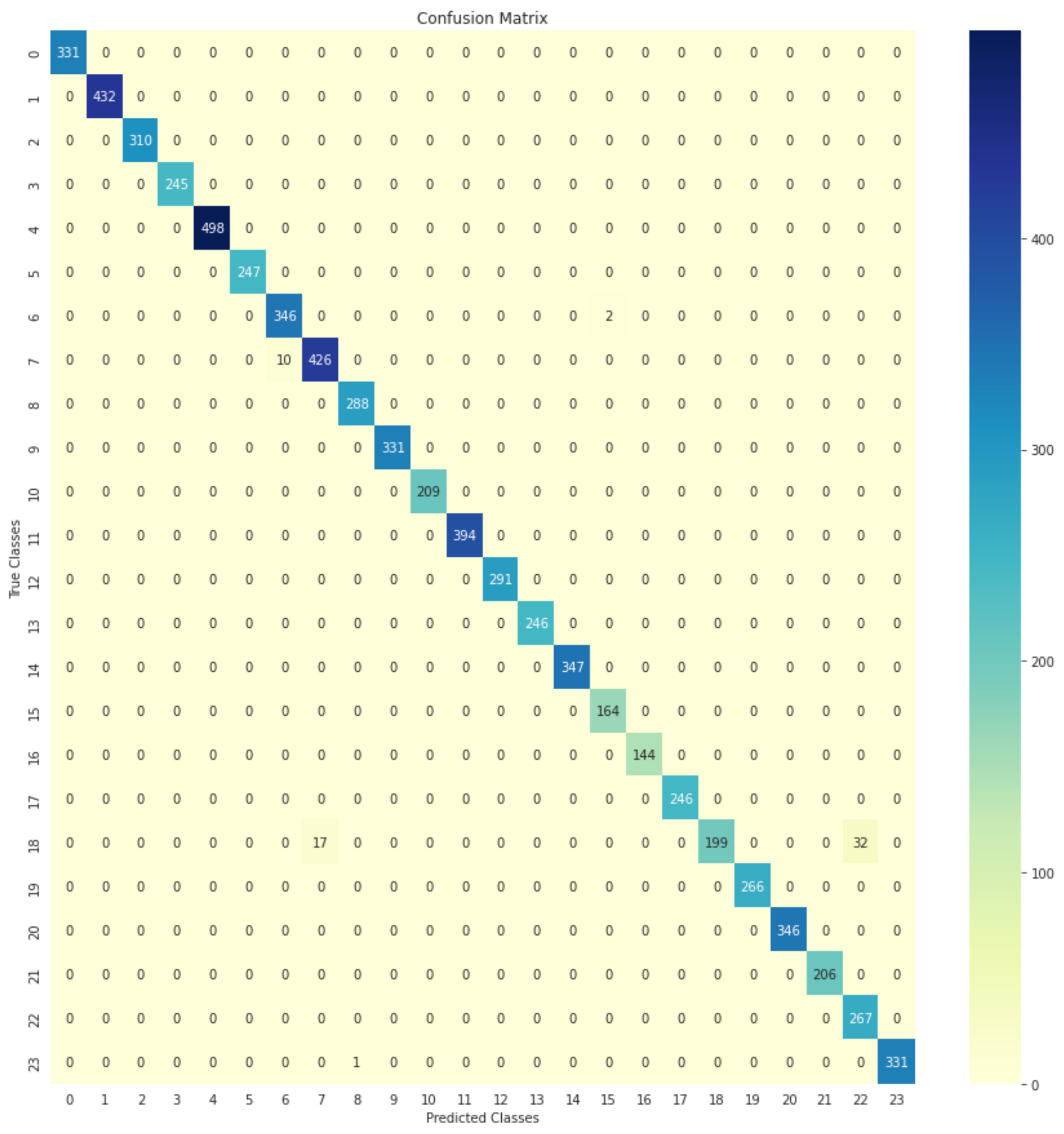
```
              precision    recall  f1-score   support

           A       1.00      1.00      1.00       331
           B       1.00      1.00      1.00       432
           C       1.00      1.00      1.00       310
           D       1.00      1.00      1.00       245
           E       1.00      1.00      1.00       498
           F       1.00      1.00      1.00       247
           G       0.97      0.99      0.98       348
           H       0.96      0.98      0.97       436
           I       1.00      1.00      1.00       288
           K       1.00      1.00      1.00       331
           L       1.00      1.00      1.00       209
           M       1.00      1.00      1.00       394
           N       1.00      1.00      1.00       291
           O       1.00      1.00      1.00       246
           P       1.00      1.00      1.00       347
           Q       0.99      1.00      0.99       164
           R       1.00      1.00      1.00       144
           S       1.00      1.00      1.00       246
           T       1.00      0.80      0.89       248
           U       1.00      1.00      1.00       266
           V       1.00      1.00      1.00       346
           W       1.00      1.00      1.00       206
           X       0.89      1.00      0.94       267
           Y       1.00      1.00      1.00       332

    accuracy                           0.99      7172
   macro avg       0.99      0.99      0.99      7172
weighted avg       0.99      0.99      0.99      7172
```

We can also visualize the confusion matrix.

```
[ ]: #Visualizing the confusion matrix
     from sklearn.metrics import confusion_matrix
     CM = confusion_matrix(y_test_inv_label, predictions)
     plt.figure(figsize = (15,15))
     sns.heatmap(CM, annot=True, cmap="YlGnBu", fmt = 'g')
     plt.xlabel("Predicted Classes")
     plt.ylabel("True Classes")
     plt.title("Confusion Matrix")
     plt.show()
```



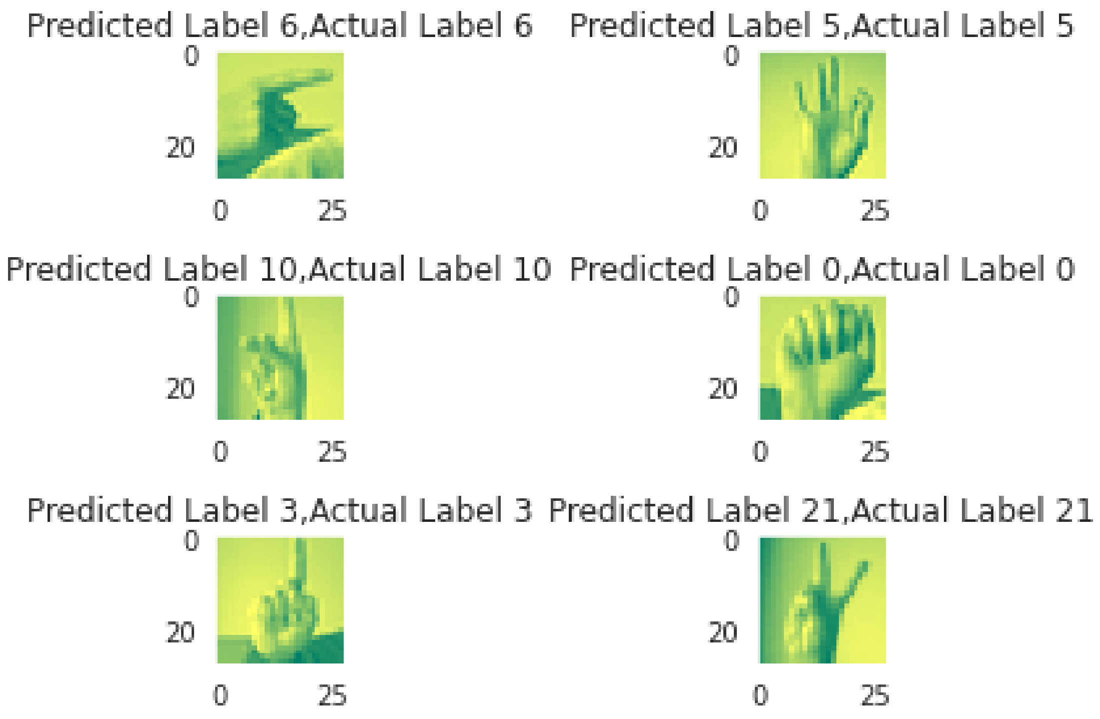
In the confusion matrix above, we can see that each row of the confusion matrix represents the instances of an actual class and each column represents the instances of a predicted class that mostly correct for each label.

4.2 Checking the results

```
[ ]: correct_label = np.nonzero(predictions == y_test_inv_label)[0]
     print(correct_label)
```

```
[   0    1    2 ... 7169 7170 7171]
```

[ ]:
```
i = 0
for c in correct_label[:6]:
    plt.subplot(3,2,i+1)
    plt.imshow(x_test[c].reshape(28,28), cmap="summer", interpolation='none')
    plt.title("Predicted Label {},Actual Label {}".format(predictions[c],␣
 ↪y_test_inv_label[c]))
    plt.tight_layout()
    i += 1
```


Predicted Label 6,Actual Label 6
Predicted Label 5,Actual Label 5
Predicted Label 10,Actual Label 10
Predicted Label 0,Actual Label 0
Predicted Label 3,Actual Label 3
Predicted Label 21,Actual Label 21
0
20
0
25

5 Conclusion

In this report we have built a convolutional neural network in order to classify letters of the American Sign Language alphabet. In the original model that we created we spotted an issue regarding overfitting. The model had a high accuracy when using the training data, but a low accuracy when applying it to the test data. Therefore we chose to try to tweak the model in order to handle the overfitting issue. We have learned that optimization such as Data Augmentation is an important method to increase the accuracy of model performance, as well as applying some callback of ReduceLROnPlateau during the stage of model training.

5.0.1 Further research

There are a myriad of ways to construct and change convolutional neural networks to create a better test accuracy. In this report, we have only focused on a few.

In future research, it would be interesting to see how changing the number of filters and hidden layers affect the model. It would also be interesting to use different regularization methods not used here to avoid overfitting. For example, in this case we did not implement the method of early stopping. During the stage of model fitting, one can explore further for the callback which can apply EarlyStopping, LambdaCallback, LearningRateScheduler to see whether these can help to improve the model performance or not. Similarly, other optimization methods can be explored further such as using Weight Initialization or Shortcut Connections.

Furthermore, in terms of the number of epochs this is a hyperparameter that one can tune by tuning the model up until a set amount of epochs can be selected and then plot the training as well as the validation loss against the number of epochs. Additionally, one can further analyse this by using early stopping as call back to monitor the number of epochs.