ANN Final Project

Kaggle competition:

In this competition you are challenged with a digits classification task. The images are similar to MNIST, however they contain the background noise.

28x28 greyscale images are flattened into a 784-elements array. The train.csv contains also the target as the last column.

28x28 greyscale images are flattened into a 784-elements array. The train.csv contains also the target as the last column.

File descriptions

- *train.csv - the training set*
- *test.csv - the test set*
- *sampleSubmission.csv - a sample submission file in the correct format*

1 Introduction

The training set was relatively small only containing 12000 entries and the test set was large, containing 50000 entries. Normally the proportions are the other way around for the training and test datasets. So, the question was how to handle this image classification task given such a small dataset?

The first thing done was to test out different models against the training set. Then a neural network model was implemented to see if better results could be yielded. A baseline neural network was implemented and then individual steps were taken to see what could be done to improve the model.

Initially it was thought that the dataset was already flattened so no convolutions were implemented. But after clarification with Nadiya it seemed all that was required was some reshaping in order to restore the images.

A baseline CNN model was implemented which achieved much better results than the non-CNN model. Then steps were taken to see how the features and hyperparameters of the model could be adjusted in order to obtain the best results.

The goal was to achieve a model with the best accuracy and the lowest loss possible which could then be used to make predictions for the test dataset and then submitted to the Kaggle competition.

This document outlines the above approach and all the steps taken to achieve the best model.

At the time of writing the best personal score on Kaggle is 0.98126 which is no. 5 on the leaderboard.

2 Testing of Different Models

Initially, different types of models were assessed against the training dataset to see which ones gave a good result. Kfolds validation was used with a split of 5 folds.

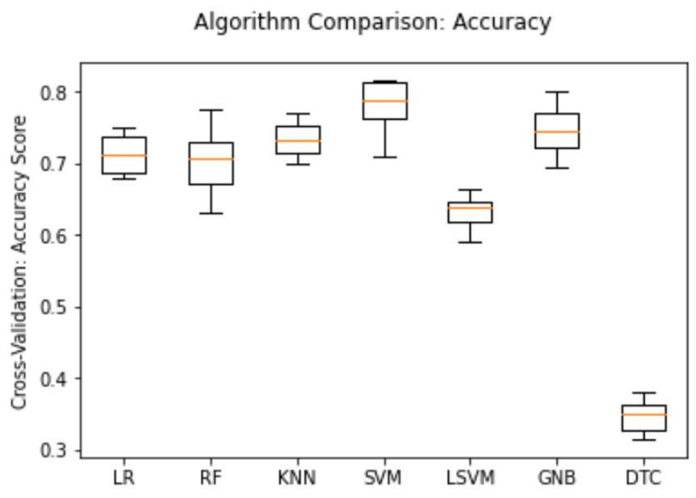
```python
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC, LinearSVC
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt

def compareDifferentModelsAccuracy(X_train,Y_train,X_test,Y_test):
    print('\nCompare Multiple Classifiers:')
    print('\nK-Fold Cross-Validation Accuracy:\n')
    models = []
    models.append(('LR', LogisticRegression(max_iter=4000)))
    models.append(('RF', RandomForestClassifier()))
    models.append(('KNN', KNeighborsClassifier()))
    models.append(('SVM', SVC(max_iter=4000)))
    models.append(('LSVM', LinearSVC(max_iter=4000)))
    models.append(('GNB', GaussianNB()))
    models.append(('DTC', DecisionTreeClassifier()))

    resultsAccuracy = []
    names = []
    for name, model in models:
        model.fit(X_train,Y_train)
        kfold = KFold(n_splits=10)
        accuracy_results = cross_val_score(model, X_test, Y_test, cv=kfold, scoring='accuracy')
        resultsAccuracy.append(accuracy_results)
        names.append(name)
        accuracyMessage = "%s: %f (%f)" % (name, accuracy_results.mean(), accuracy_results.std())
        print(accuracyMessage)

        # boxplot algorithm comparison
    fig = plt.figure()
    fig.suptitle('Algorithm Comparison: Accuracy')
    ax = fig.add_subplot(111)
    plt.boxplot(resultsAccuracy)
    ax.set_xticklabels(names)
    ax.set_ylabel('Cross-Validation: Accuracy Score')
    plt.show()
    return
```

```python
compareDifferentModelsAccuracy(X_train,y_train,X_test,y_test)
```

Below are the results obtained from running the above code:

```
Compare Multiple Classifiers:

K-Fold Cross-Validation Accuracy:

LR: 0.713500 (0.026557)
RF: 0.704000 (0.043635)
KNN: 0.734000 (0.023108)
SVM: 0.783000 (0.032496)
```

```
LSVM: 0.632000 (0.021471)
GNB: 0.747000 (0.032726)
DTC: 0.348500 (0.022028)
```



The best result for the accuracy was obtained with the SVM model but it was thought that something better could be obtained running with a neural network configuration.

3 Baseline NN Model

Initially it was thought that the dataset was already flattened so the neural network model in the below code was attempted.

```
from numpy import mean
from numpy import std
from matplotlib import pyplot
from sklearn.model_selection import KFold
from keras.datasets import mnist
from keras.utils import to_categorical


def create_model():

    model = Sequential()
    model.add(Dense(512, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(10, activation='softmax'))

    opt = SGD(lr=0.01, momentum=0.9)
    # compile model
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
    # compiling the sequential model

    return model
```

```
from sklearn.model_selection import KFold
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import train_test_split

# evaluate a model using k-fold cross-validation
def fit_and_evaluate_model(dataX, dataY, n_folds=5):
    scores, histories = list(), list()

    # prepare cross validation
    kfold = KFold(n_folds, shuffle=True, random_state=1)
    # enumerate splits

    # define model
    model = create_model()

    for train_ix, test_ix in kfold.split(dataX):

        # select rows for train and test
        trainX, testX = dataX.iloc[train_ix], dataX.iloc[test_ix]
        trainY, testY = dataY[train_ix], dataY[test_ix]

        # fit model
        history = model.fit(trainX, trainY, epochs=20, batch_size=128, validation_data=(testX, testY), verbose=0)

        # evaluate model
        _, acc = model.evaluate(testX, testY, verbose=0)
        print('> %.3f' % (acc * 100.0))
        # stores scores
        scores.append(acc)
        histories.append(history)
        print("#################################")
    loss, accuracy = model.evaluate(X_test, Y_test, verbose=0)
    print()
    print(f'Test loss: {loss:.3}')
    print(f'Test accuracy: {accuracy:.3}')
    return scores, histories
```

```
from numpy import mean
from numpy import std
from matplotlib import pyplot
# plot loss and accuracy curves
def plot_loss_accuracy(histories):
    for i in range(len(histories)):
        # plot loss
        pyplot.subplot(2, 1, 1)
        pyplot.title('Cross Entropy Loss')
        pyplot.plot(histories[i].history['loss'], color='blue', label='train')
        pyplot.plot(histories[i].history['val_loss'], color='orange', label='test')

        # plot accuracy
        pyplot.subplot(2, 1, 2)
        pyplot.title('\nClassification Accuracy')
        pyplot.plot(histories[i].history['accuracy'], color='blue', label='train')
        pyplot.plot(histories[i].history['val_accuracy'], color='orange', label='test')
    pyplot.show()

# plot model performance
def plot_performance(scores):
    # print summary
    print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores)*100, std(scores)*100, len(scores)))
    # boxplot of results
    pyplot.boxplot(scores)
    pyplot.show()
```

```
scores, histories = fit_and_evaluate_model(X_train, Y_train)
# plot loss and accuracy curves
plot_loss_accuracy(histories)
#plot estimated performance
plot_performance(scores)
```

The initial baseline model contained a dense layer with 512 nodes using a rectified linear unit (Relu) activation function followed by an output layer containing 10 nodes and a softmax activation function.

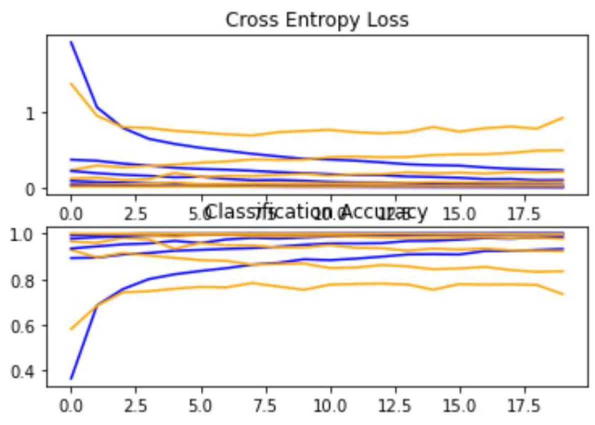
The optimizer implemented was SGD with a learning rate of 0.01 and a momentum of 0.9.

Kfolds validation was used with a split of 5.

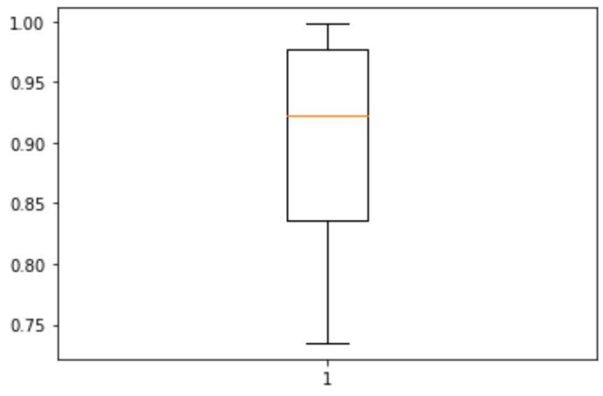
The results obtained from the above model were the following:

```
> 73.500
#################################
> 83.550
#################################
> 92.300
#################################
> 97.750
#################################
> 99.800
#################################
```

The results initially looked promising with the last run achieving an accuracy of 99.8%



```
Accuracy: mean=89.380 std=9.730, n=5
```



However, running the model against the actual test set returned a much lower accuracy of only 76% which is slightly less than that achieved by the SVM model (78%). Steps were then taken to see if this model could be improved.

```
Test loss: 1.11
Test accuracy: 0.761
```

4 Improving the NN Model

A number of factors were tested in order to try and improve the previous model such as:

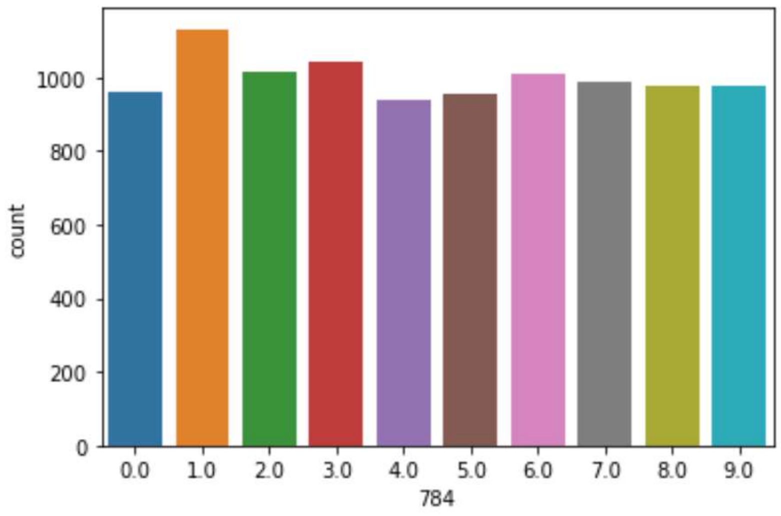
1. Ensuring the classes were distributed evenly
2. Different variations of Kfolds and train_test_split
3. Trying out 5 different optimizers
4. Playing with the learning rate of the optimizer
5. Different configurations for the layers of the model
6. Regularization by changing the drop-out rate
7. Batch normalization
8. Changing the no. of epochs and batch size

4.1 Class Distribution

Having an uneven class distribution can really affect the results of the model. Therefore, a count was done of each of the classes in the training dataset.

The classes looked as they were more or less evenly distributed.

```
g = sns.countplot(y_train)
y_train.value_counts()
```



4.2 KFolds vs train_test_split

After trying KFolds with splits of 5 and 10, the train_test_split method was used. It was found that this method gave the most flexibility with regards to testing. Ultimately it was found that good results could be obtained using a test_size of 0.1.

4.3 Different Optimizers

The correct optimizer must be selected when trying to create the best model as it can really have an impact on the results obtained. The following optimizers were tested:

- SGD
- RMSProp
- Adam

- Adadelta
- Adagrad

RMSProp and Adam gave the best results, but it in the end the decision was made to go with RMSProp as it gave fractionally better results.

The loss function chosen was **categorical cross-entropy** as the output had been one-hot encoded.

4.4 Optimizer Learning Rate

After choosing the RMSProp optimizer, the learning rate was tested, changing it first by a factor of 10 e.g. 0.1, 0.01, 0.001 and then smaller changes until eventually the best result was obtained with a learning rate of 0.05.

```
optimizer = RMSprop(lr=0.05, rho=0.1, epsilon=1e-08, decay=0.0)
```

4.5 Model Layer Configuration

Initially the following configuration was attempted

```
model = Sequential()
model.add(layers.Dense(1024, activation='relu', input_shape=(28 * 28,| )))
model.add(layers.Dense(10, activation='softmax'))
```

Which gave the following result:

```
Test loss: 0.835
Test accuracy: 0.802
```

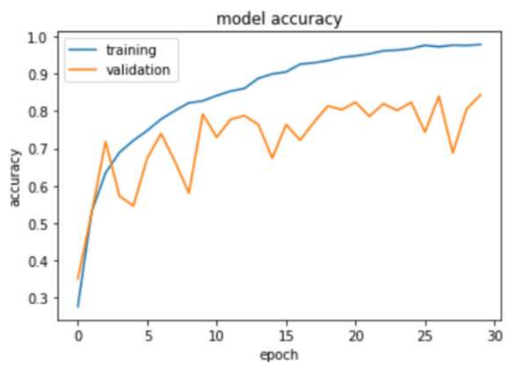
Then the following configuration was tested:

```
model = Sequential()
model.add(layers.Dense(1024, activation='relu', input_shape=(28 * 28, )))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

Which gave a better loss and accuracy:

```
Test loss: 0.75
Test accuracy: 0.849
```

But the plot showed some erratic behaviour for the validation run:

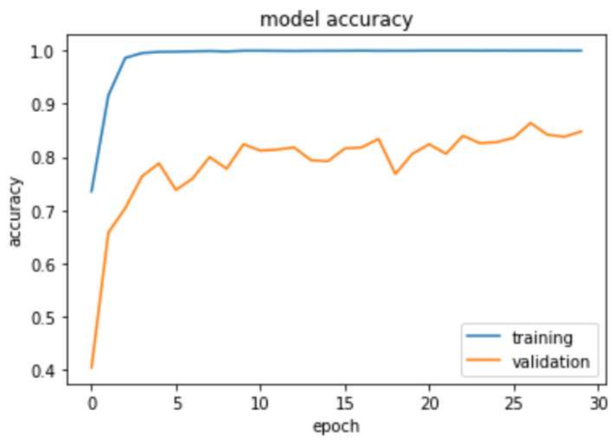


4.6 Batch Normalization

To correct the erratic behaviour seen in the last step, Batch Normalization was applied to the layers. After applying the Batch Normalization, the accuracy increased. The following result was obtained:

```
Test loss: 0.502
Test accuracy: 0.854
```

The plot of the validation also looked better.

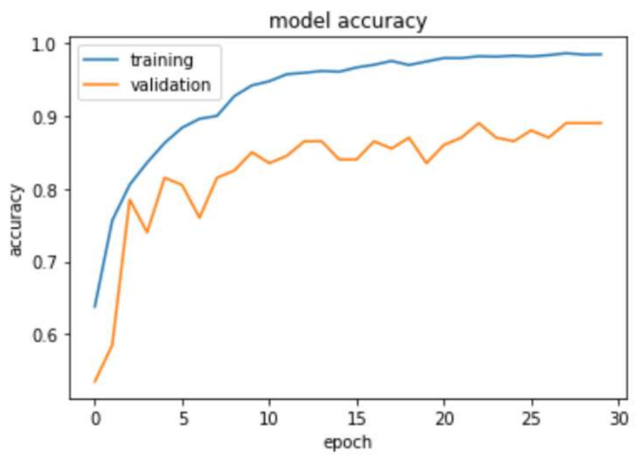


4.7 Regularization using Dropout

Various dropout rates were testing for the first two layers of the model. In the end, 0.7 and 0.5 were found to give the best results. The results of the model with the dropouts added can be found below.

```
Test loss: 0.599
Test accuracy: 0.861
```

The test accuracy has improved slightly compared to the previous run.

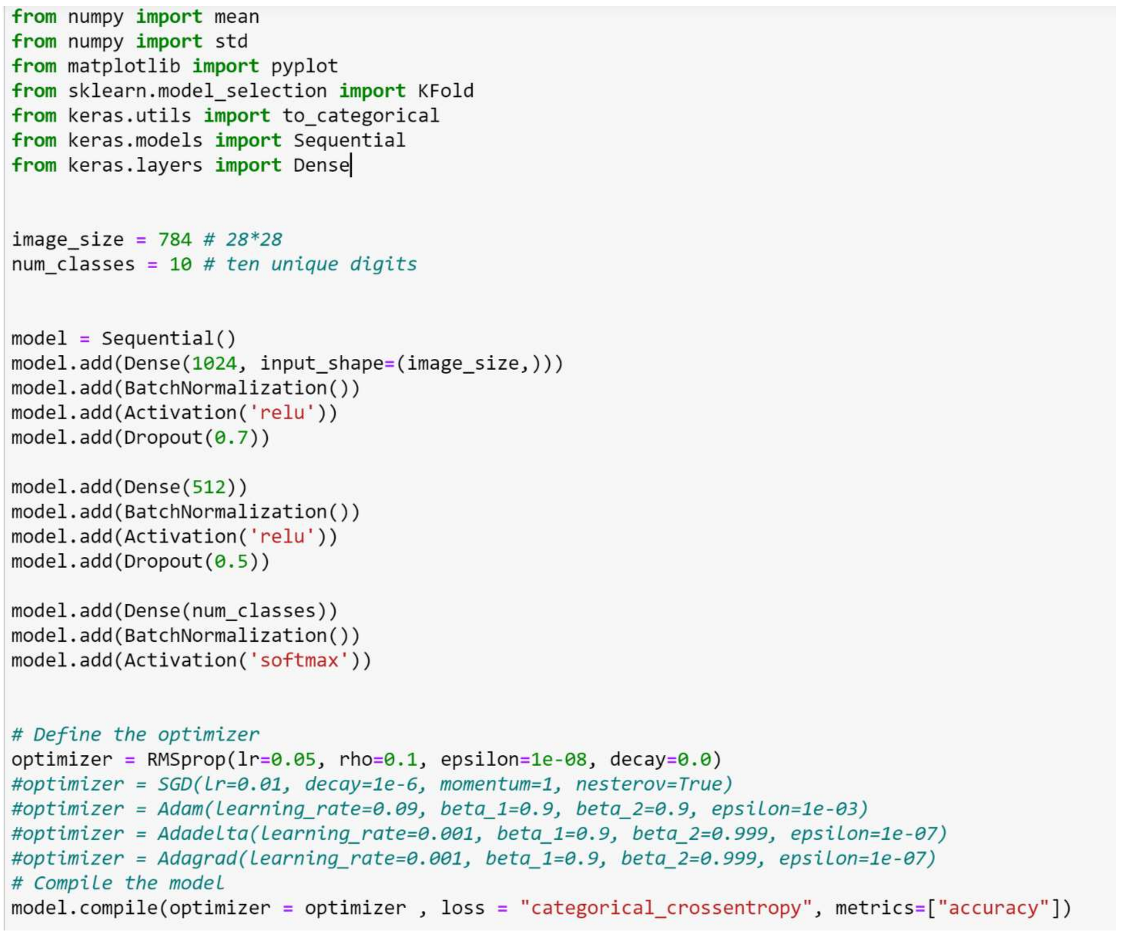


4.8 Epochs and Batch Size

Epochs were tested whilst keeping batch size constant at 32. Starting at 20 epochs and then increasing to 25, 30, 50, 60, 80, 100, 150, 200, 300. 30 epochs were found to be the optimum number of epochs.

Batch size was tested whilst keeping the number of epochs constant. Starting at 10, then 16, 32, 64, and finally 128. The optimum batch size was found to be 64.

4.9 Final NN Configuration



```python
from numpy import mean
from numpy import std
from matplotlib import pyplot
from sklearn.model_selection import KFold
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense


image_size = 784 # 28*28
num_classes = 10 # ten unique digits


model = Sequential()
model.add(Dense(1024, input_shape=(image_size,)))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(0.7))

model.add(Dense(512))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(num_classes))
model.add(BatchNormalization())
model.add(Activation('softmax'))


# Define the optimizer
optimizer = RMSprop(lr=0.05, rho=0.1, epsilon=1e-08, decay=0.0)
#optimizer = SGD(lr=0.01, decay=1e-6, momentum=1, nesterov=True)
#optimizer = Adam(learning_rate=0.09, beta_1=0.9, beta_2=0.9, epsilon=1e-03)
#optimizer = Adadelta(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07)
#optimizer = Adagrad(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07)
# Compile the model
model.compile(optimizer = optimizer , loss = "categorical_crossentropy", metrics=["accuracy"])
```

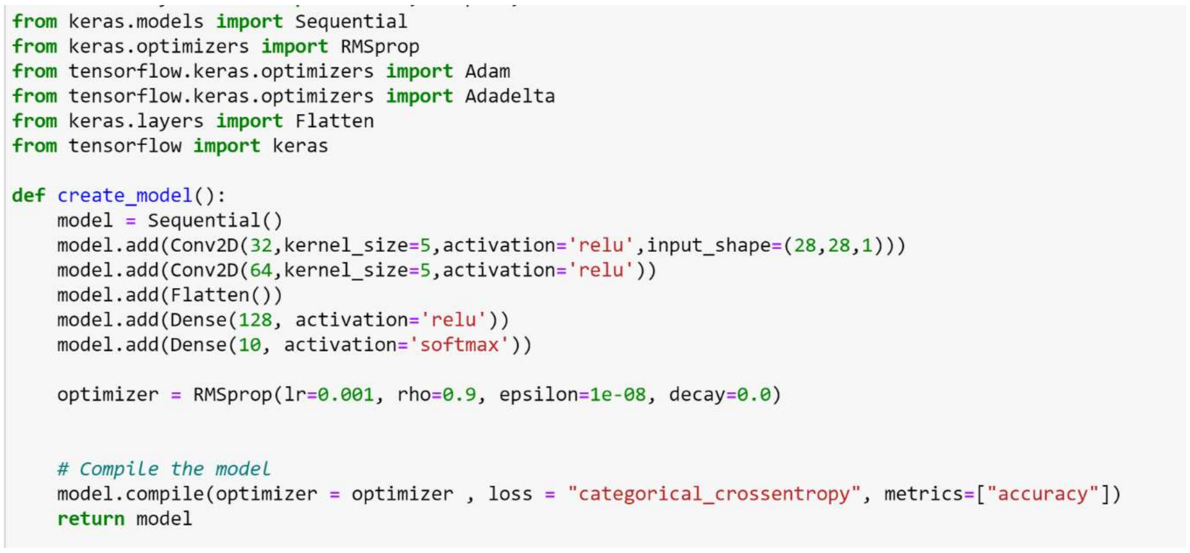
After trying all these things, the accuracy of the model could not be improved beyond 86%. As the data was already flattened it was thought that it was not possible to work with images and therefore convolutions would not be possible.

After discussion with Nadiya it was determined that all that was required was to reshape the data in order to work with images again.

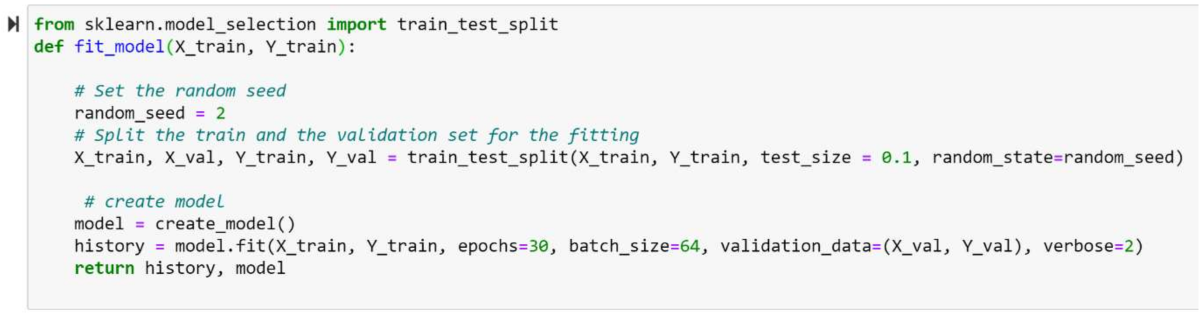
Now that images were available, a baseline ConvNet model could be created.

5 Baseline CNN Model

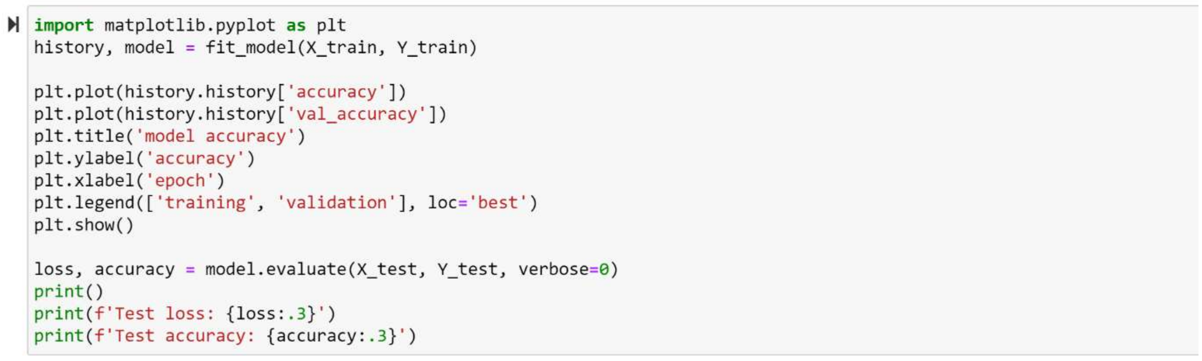
A basic CNN model was created with the following code:



```python
from keras.models import Sequential
from keras.optimizers import RMSprop
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.optimizers import Adadelta
from keras.layers import Flatten
from tensorflow import keras

def create_model():
    model = Sequential()
    model.add(Conv2D(32,kernel_size=5,activation='relu',input_shape=(28,28,1)))
    model.add(Conv2D(64,kernel_size=5,activation='relu'))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(10, activation='softmax'))

    optimizer = RMSprop(lr=0.001, rho=0.9, epsilon=1e-08, decay=0.0)


    # Compile the model
    model.compile(optimizer = optimizer , loss = "categorical_crossentropy", metrics=["accuracy"])
    return model
```



```python
from sklearn.model_selection import train_test_split
def fit_model(X_train, Y_train):

    # Set the random seed
    random_seed = 2
    # Split the train and the validation set for the fitting
    X_train, X_val, Y_train, Y_val = train_test_split(X_train, Y_train, test_size = 0.1, random_state=random_seed)

     # create model
    model = create_model()
    history = model.fit(X_train, Y_train, epochs=30, batch_size=64, validation_data=(X_val, Y_val), verbose=2)
    return history, model
```

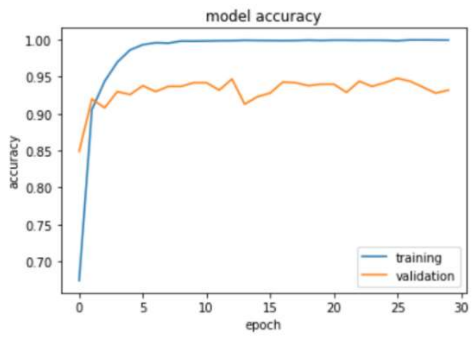


```python
import matplotlib.pyplot as plt
history, model = fit_model(X_train, Y_train)

plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['training', 'validation'], loc='best')
plt.show()

loss, accuracy = model.evaluate(X_test, Y_test, verbose=0)
print()
print(f'Test loss: {loss:.3}')
print(f'Test accuracy: {accuracy:.3}')
```

An accuracy of 93.8% was achieved! Which was much better than the 86% achieved with the non-CNN model.

```
Test loss: 0.669
Test accuracy: 0.938
```

Below is a plot of the training and validation runs against the model.



Although this was a much better result than the non-CNN model, it could still be improved.

6 Improving the CNN Model

The same factors that were tested previously in section 4, needed to be tested again with this model in order to try and improve the baseline CNN model. However, some of the answers were already known such as the distribution of the classes and which optimizers were most suitable.

1. Ensuring the classes were distributed evenly – already known as using same dataset
2. Different variations of Kfolds and train_test_split – already known that train_test_split more flexible
3. Trying out 5 different optimizers – already known that RMSprop and Adam are most suitable.
4. Different configurations for the layers of the model
5. Batch normalization
6. Regularization by changing the drop-out rate
7. Playing with the learning rate of the optimizer
8. Changing the no. of epochs and batch size

In addition, we could also investigate if the following could improve the model:

9. Number of filters
10. Filter size
11. Max pooling
12. Data Augmentation

6.1 Model Layer Configuration, Filters and Max Pooling

The number of layers, filters and filter size in the model can really affect the performance. Therefore, a number of tests were done in order to find the optimum configuration.

In the baseline model there was a convolutional layer with 32 feature maps using a 5 x 5 filter and a stride of 1. This was followed by another convolutional layer with 64 feature maps using a 5 x 5 filter also with a stride of 1 and then finally a fully connected dense layer with 128 units.

Several variations of convolutional sub-sampling and feature maps with and without max pooling were tested:

Test 1

- 2 convolutional layers with 32 feature maps using a 5 x 5 filter and a stride of 1
- Followed by 2 convolutional layers with 64 feature maps using a 5 x 5 filter and a stride of 1 Then fully connected dense layer with 128 units.

784 - **[32C5-32C5] - [64C5-64C5]** - 128 - 10

Test 2

- 2 convolutional layers with 32 feature maps using a 5 x 5 filter and a stride of 1.
- Followed by max pooling using 2 x 2 filter and stride of 2.
- Followed by 2 convolutional layers with 64 feature maps using a 5 x 5 filter and a stride of 1
- Followed by max pooling using 2 x 2 filter and stride of 2.
- And a fully connected dense layer with 128 units.

784 - **[32C5-32C5-P2] - [64C5-64C5-P2]** - 128 - 10

Test 3

- 2 convolutional layers with 32 feature maps using a 5 x 5 filter and a stride of 1
- Followed by max pooling using 2 x 2 filter and stride of 2.
- 2 convolutional layers with 64 feature maps using a 5 x 5 filter and a stride of 1
- Followed by max pooling using 2 x 2 filter and stride of 2
- And a fully connected dense layer with 256 units.

784- **[32C5-32C5-P2] - [64C5-64C5-P2]** - 256 - 10

Test 4

The max pooling was replaced with a 3rd convolutional layer for the 32 and 64 feature maps.

- 3 convolutional layers with 32 feature maps using a 5 x 5 filter and a stride of 1
- 3 convolutional layers with 64 feature maps using a 5 x 5 filter and a stride of 1
- And a fully connected dense layer with 128 units.

784 - **[32C5-32C5-32C5] - [64C5-64C5-64C5]** - 128 - 10

Test 6

- 3 convolutional layers with 32 feature maps, the first 2 layers using a 3 x 3 filter and a stride of 1 and the third layer using a 5 x 5 filter.
- Followed by 3 convolutional layers with 64 feature maps with the first 2 layers using a using a 3 x 3 filter and the third layer a 5 x 5 filter and a stride of 1
- And a fully connected dense layer with 128 units.

784 - **[32C3-32C3-32C5] - [64C3-64C3-64C5]** - 128 - 10

Test 7

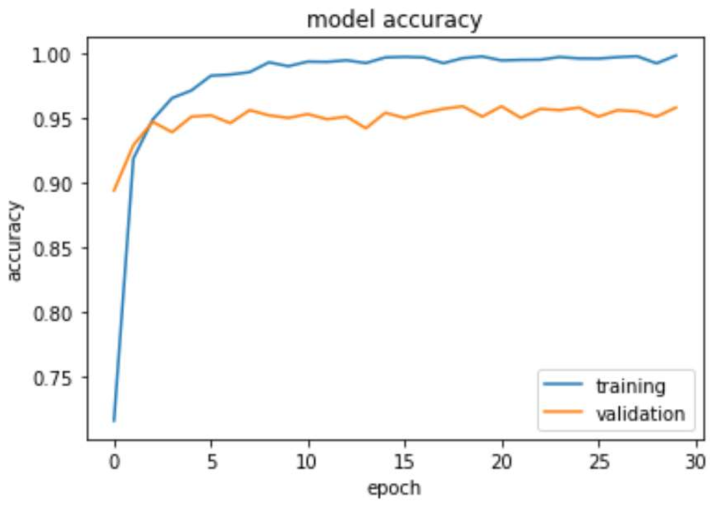
Replace the fully connected dense layer with another convolutional layer with 128 feature maps

- 3 convolutional layers with 32 feature maps, the first 2 layers using a 3 x 3 filter and a stride of 1 and the third layer using a 5 x 5 filter and 'same' padding.
- Followed by 3 convolutional layers with 64 feature maps with the first 2 layers using a using a 3 x 3 filter and the third layer a 5 x 5 filter and a stride of 1 and 'same' padding.
- And a convolutional layer with 128 feature maps with a filter size of 4.

784 - **[32C3-32C3-32C5] - [64C3-64C3-64C5]** – **[128C4]** – 10

This configuration achieved the best result:

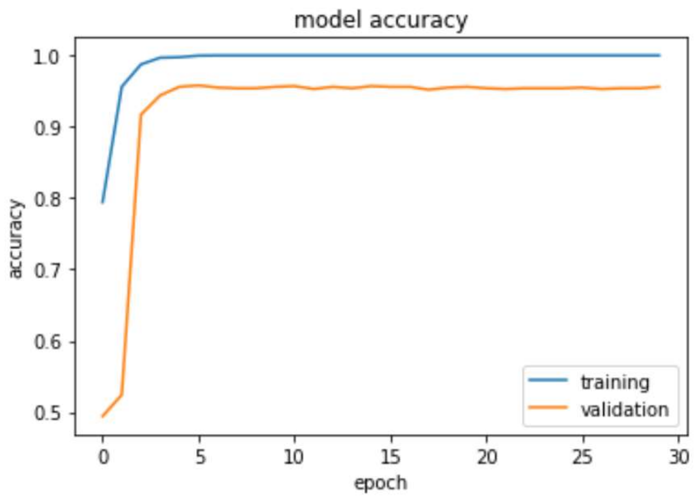
```
Test loss: 0.212
Test accuracy: 0.963
```



6.2 Batch Normalization

Batch normalization stabilizes the learning process and can reduce the number of epochs required to train a model. A batch normalization step was added after each of the convolutional layers and the following results were obtained. Although the accuracy was fractionally lower, the loss was reduced. The validation results which can be seen in the plot below were also a lot smoother compared to the previous plot.

```
Test loss: 0.129
Test accuracy: 0.96
```



6.3 Regularization using Dropout

Dropout prevents the neural network from overfitting and therefore helps the model to generalize better. Previously, for the non-CNN model, adding dropout improved the result.

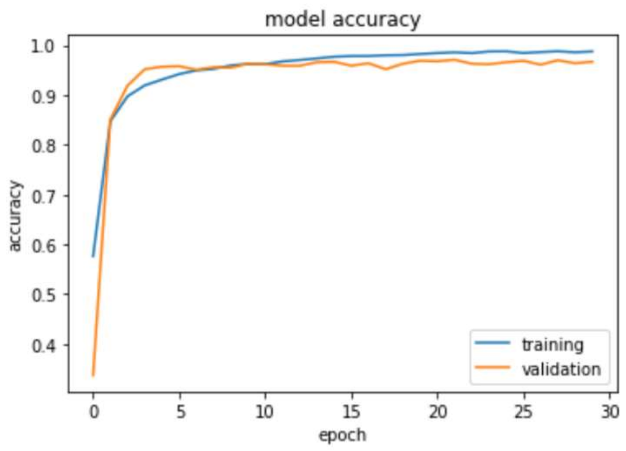
Various dropout levels were tested (10-70%) in different layers. The below configuration obtained the best results.

- 0.6 after the three 32 convolutional layers
- 0.4 after the three 64 convolutional layers
- 0.4 after the single 128 convolutional layer

```
model = Sequential()

model.add(Conv2D(32,kernel_size=3,activation='relu',input_shape=(28,28,1)))
model.add(BatchNormalization())
model.add(Conv2D(32,kernel_size=3,activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(32,kernel_size=5,strides=2,padding='same',activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.6))

model.add(Conv2D(64,kernel_size=3,activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(64,kernel_size=3,activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(64,kernel_size=5,strides=2,padding='same',activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.4))

model.add(Conv2D(128, kernel_size = 4, activation='relu'))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dropout(0.4))
model.add(Dense(10, activation='softmax'))
```

The results obtained were the following. The test accuracy increased and the loss also went down.

```
Test loss: 0.101
Test accuracy: 0.979
```



6.4 Data Augmentation

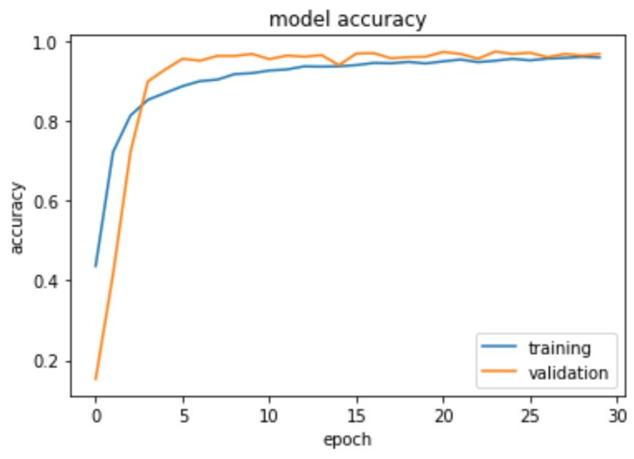
In various texts read about convolutional neural networks it had been mentioned that data augmentation can improve the overall result when processing images and if the dataset is small. However, care must be taken as otherwise the actions taken might have a negative effect if unsuitable such as horizontal flip or large shift and rotations.

The following settings were used for the data augmentation:

```
datagen = tf.keras.preprocessing.image.ImageDataGenerator(
        rotation_range=10,
        zoom_range = 0.10,
        width_shift_range=0.1,
        height_shift_range=0.1)

datagen.fit(X_train)
```

The accuracy did not improve but the loss was lower, so overall the situation did improve.

```
Test loss: 0.0667
Test accuracy: 0.979
```



6.5 Learning Rate

The RMSProp optimizer was used as previously. A learning rate of 0.05 gave the best results.

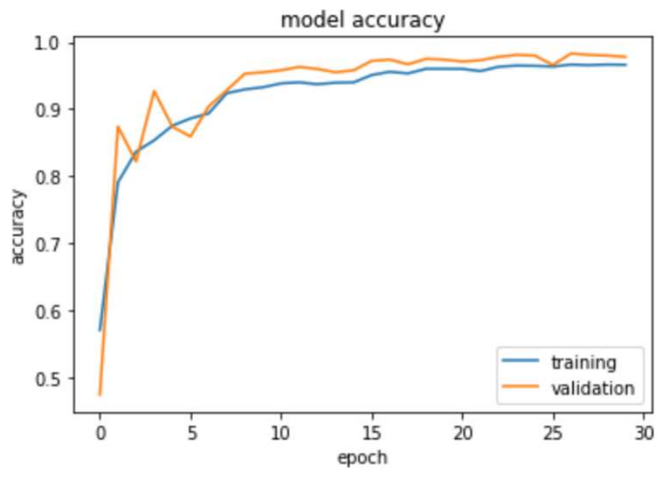
After reading about learning rate annealers which start learning when the rate is high and start decreasing when we get closer to a local minimum, a learning rate annealer was added. This is more flexible than setting the decay rate in the optimizer parameters.

The keras ReduceLROnPlateau annealer was used.

```
# Set a learning rate annealer
learning_rate_reduction = tf.keras.callbacks.ReduceLROnPlateau(monitor='val_accuracy',
                                                               patience=3,
                                                               verbose=1,
                                                               factor=0.5,
                                                               min_lr=0.00001)
```

The following result was achieved:

```
Test loss: 0.048
Test accuracy: 0.984
```



This was the best accuracy and loss achieved so far. Predictions were made against the test dataset and uploaded to Kaggle. The percentage accuracy was slightly lower on Kaggle but this was to be expected as the test dataset contained many more rows than the training dataset.

Later, an accuracy of 98.7% was obtained and predictions for the test dataset were uploaded to Kaggle.

6.5 Epochs and Batch Size

As mentioned in section 4.8, epochs were tested whilst keeping batch size constant at 32. Starting at 20 epochs and then increasing to 25, 30, 50, 60, 80, 100, 150, 200, 300.

30 epochs were found to be the optimum number of epochs.

Batch size was tested whilst keeping the number of epochs and everything else constant. Starting at 10, then 16, 32, 64, and finally 128. The optimum batch size was found to be 64.

The final code can be found in the Appendix.

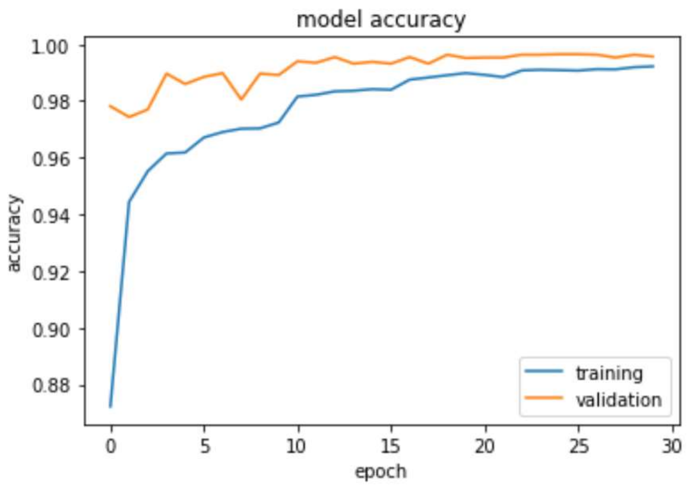
7 Training on a larger MNIST dataset

In articles written about MNIST it is mentioned that when you have a small dataset you can train your dataset with another larger MNIST dataset and then you can make your predictions based on that model.

This was attempted with the MNIST dataset in Python whose training set contains 60000 images:

```
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
```

The accuracy on the MNIST Python dataset was excellent at 99.6%:



```
Test loss: 0.0144
Test accuracy: 0.996
```

This model was then run against the DSTI MNIST train dataset:

```
mytrain = pd.read_csv("C:/Users/yalaz/Documents/DSTI/Modules/ANN/Assignment/train_data.csv", header=None)

my_Y_train = mytrain.iloc[:,-1]#only the last column
my_X_train = mytrain.iloc[:,:-1]#everything except the last column

my_X_train = my_X_train.values.reshape(-1,28,28,1)
my_Y_train = to_categorical(my_Y_train, num_classes = 10)

my_loss, my_accuracy = model.evaluate(my_X_train, my_Y_train, verbose=0)

print()
print(f'Test loss: {my_loss:.3}')
print(f'Test accuracy: {my_accuracy:.3}')
```

```
Test loss: 14.5
Test accuracy: 0.0994
```

The results obtained when run against the DSTI MNIST train dataset were not good, 9.9%. Maybe because the images in the Python dataset were too different from the DSTI ones.

8 Summary

The best model achieved was a CNN model which had its layers and hyperparameters tuned.

The 2 best results were uploaded to Kaggle. On Kaggle the highest personal score achieved was 98.126%.

We have seen that there are many things that can be tuned such as layer configuration, optimizers, learning rates, filters, filter size, data augmentation, dropout rate, batch normalization and so on. The results obtained were very good but also required some time and effort to achieve.

Finally, the dataset was said to be an MNIST like dataset with some noise. If time had permitted, I would have liked to use an autoencoder to try and remove the noise in order to try and get a better result.

9 Appendix

The final code set can be found below.

```python
from keras.utils import to_categorical

import pandas as pd
import numpy as np
import tensorflow as tf

# Load datasets
train = pd.read_csv("C:/Users/yalaz/Documents/DSTI/Modules/ANN/Assignment/Mix1/train_mix1.csv", header=None)
test = pd.read_csv("C:/Users/yalaz/Documents/DSTI/Modules/ANN/Assignment/Mix1/test_mix1.csv", header=None)

# Prepare data
Y_train = train.iloc[:,-1]#only the last column
X_train = train.iloc[:,:-1]#everything except the last column

X_test = test.iloc[:,:-1]#everything except the last column
Y_test = test.iloc[:,-1]#only the last column

X_train = X_train.values.reshape(-1,28,28,1)
X_test = X_test.values.reshape(-1,28,28,1)

Y_train = to_categorical(Y_train, num_classes = 10)
Y_test = to_categorical(Y_test, num_classes = 10)
```

```python
from sklearn.model_selection import train_test_split
# Set the random seed
random_seed = 2
# Split the train and the validation set for the fitting
X_train, X_val, Y_train, Y_val = train_test_split(X_train, Y_train, test_size = 0.1, random_state=random_seed)
```

```python
from keras.layers import BatchNormalization, Conv2D, MaxPool2D
from keras.layers.core import Dense, Dropout, Activation
from keras.models import Sequential
from keras.optimizers import RMSprop
from tensorflow.keras.optimizers import Adam
from keras.layers import Flatten
from tensorflow import keras

# Set the CNN model
model = Sequential()
model.add(Conv2D(32,kernel_size=3,activation='relu',input_shape=(28,28,1)))
model.add(BatchNormalization())
model.add(Conv2D(32,kernel_size=3,activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(32,kernel_size=5,strides=2,padding='same',activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.6))

model.add(Conv2D(64,kernel_size=3,activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(64,kernel_size=3,activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(64,kernel_size=5,strides=2,padding='same',activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.4))

model.add(Conv2D(128, kernel_size = 4, activation='relu'))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dropout(0.4))
model.add(Dense(10, activation='softmax'))

# Define the optimizer
optimizer = RMSprop(lr=0.05, rho=0.9, epsilon=1e-08, decay=0.0)

# Compile the model
model.compile(optimizer = optimizer , loss = "categorical_crossentropy", metrics=["accuracy"])
```

```python
# Set a learning rate annealer
learning_rate_reduction = tf.keras.callbacks.ReduceLROnPlateau(monitor='val_accuracy',
                                            patience=3,
                                            verbose=1,
                                            factor=0.5,
                                            min_lr=0.00001)
```

```python
# Generator for data augmentation
datagen = tf.keras.preprocessing.image.ImageDataGenerator(
        rotation_range=10,
        zoom_range = 0.10,
        width_shift_range=0.1,
        height_shift_range=0.1)

datagen.fit(X_train)
```

```python
# Fit the model
history = model.fit(datagen.flow(X_train,Y_train, batch_size=64),
                              epochs = 30, validation_data = (X_val,Y_val),
                              verbose = 2, steps_per_epoch=None
                              , callbacks=[learning_rate_reduction])
```

```python
import matplotlib.pyplot as plt
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['training', 'validation'], loc='best')
plt.show()

loss, accuracy = model.evaluate(X_test, Y_test, verbose=0)
print()
print(f'Test loss: {loss:.3}')
print(f'Test accuracy: {accuracy:.3}')
```

```python
final = pd.read_csv("C:/Users/yalaz/Documents/DSTI/Modules/ANN/Assignment/test_data.csv", header=None)

X_final = final.values.reshape(-1,28,28,1)

predictions = model.predict(X_final)
results = np.argmax(predictions,axis = 1)
print(results)
```

```python
results = pd.Series(results,name="Category")
submission = pd.concat([pd.Series(range(0,50000),name = "Id"),results],axis = 1)

submission.to_csv("C:/Users/yalaz/Documents/DSTI/Modules/ANN/Assignment/cnn_mnist3.csv",index=False)
```