JAHANGIRNAGAR UNIVERSITY
Department of Statistics
SAVAR, BANGLADESH

Final Project

Course Name: Big Data
Course No: WM-ASDS19 (Section: B)
Masters in
Applied Statistics and Data Science, 9th Batch
Summer 2023

by
Sk. Md. Rashid Abrar
ID: 20229048

Submitted to
Dr. Md. Rezaul Karim
Associate Professor, Department of Statistics
JAHANGIRNAGAR UNIVERSITY

Contents

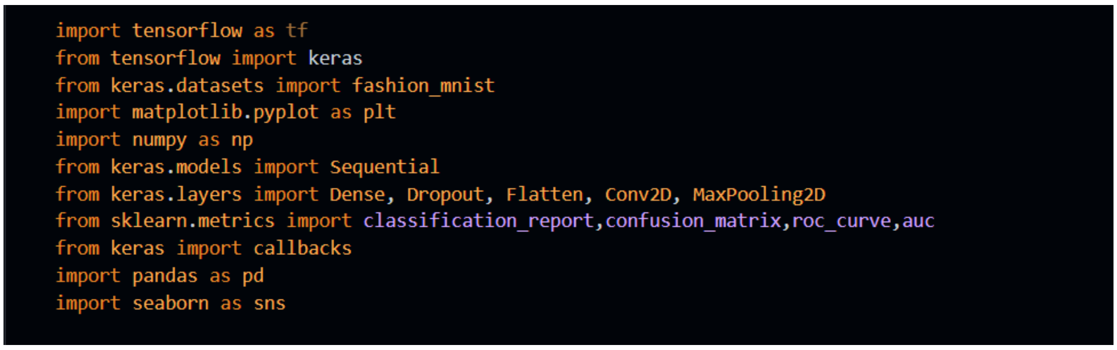
Chapter 1

Working With the Data

The Fashion MNIST dataset includes 70,000 grayscale images in 10 categories. Individual articles of clothing are depicted in low quality (28 by 28 pixels) in the photographs. We utilize ANN and CNN to train the network models on 60,000 images, and we use 10,000 images to evaluate how well the network learned to classify images. Here, it is a multi-class classification problem, and classes 0 to 9 refer to it as a T-shirt or top, trousers, pullovers, dresses, coats, sandals, shirts, sneakers, bags, and ankle boots, respectively. The outputs of both models will then be compared to determine the best categorization model for this dataset.

1.1 Importing necessary libraries

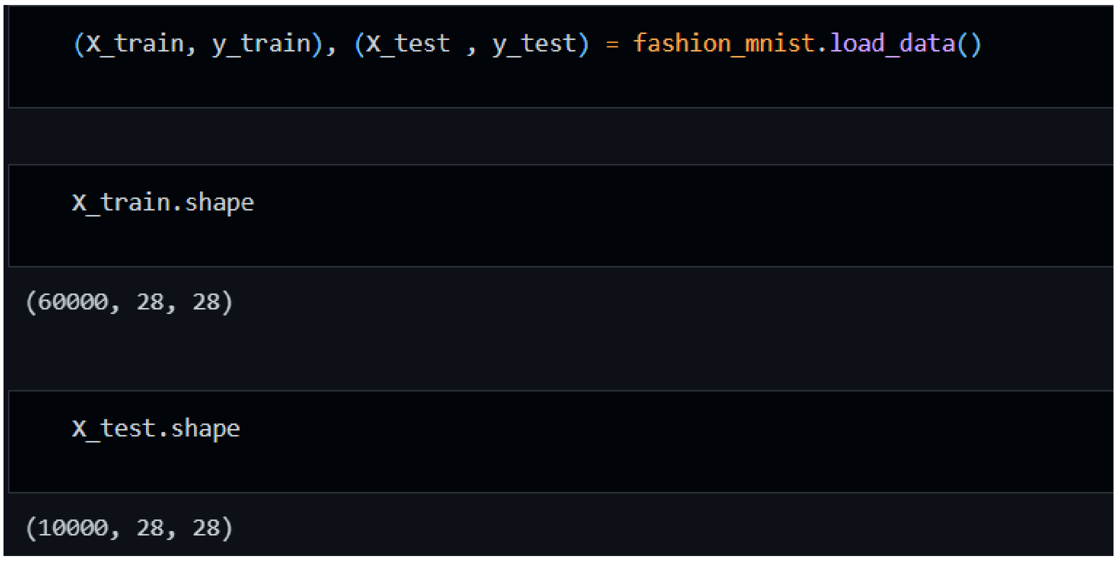
Firstly, we need to import the necessary libraries for this classification problem. Here, TensorFlow and Keras are being used to solve this neural network problem.



```
import tensorflow as tf
from tensorflow import keras
from keras.datasets import fashion_mnist
import matplotlib.pyplot as plt
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
from sklearn.metrics import classification_report,confusion_matrix,roc_curve,auc
from keras import callbacks
import pandas as pd
import seaborn as sns
```

1.2 Data Loading

The data is loaded from the Keras dataset. Upon loading, it was split into training and testing data. Then the respective training and testing shapes are shown. The training dataset has 60,000 images, and the test dataset has 10,000 images. We will need to work with 60,000 images for image classification.



```
(X_train, y_train), (X_test , y_test) = fashion_mnist.load_data()
```

```
X_train.shape
```

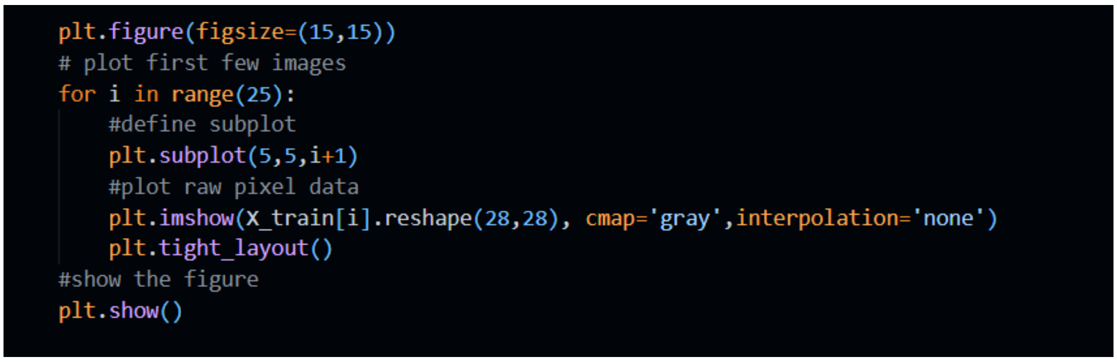
```
(60000, 28, 28)
```

```
X_test.shape
```

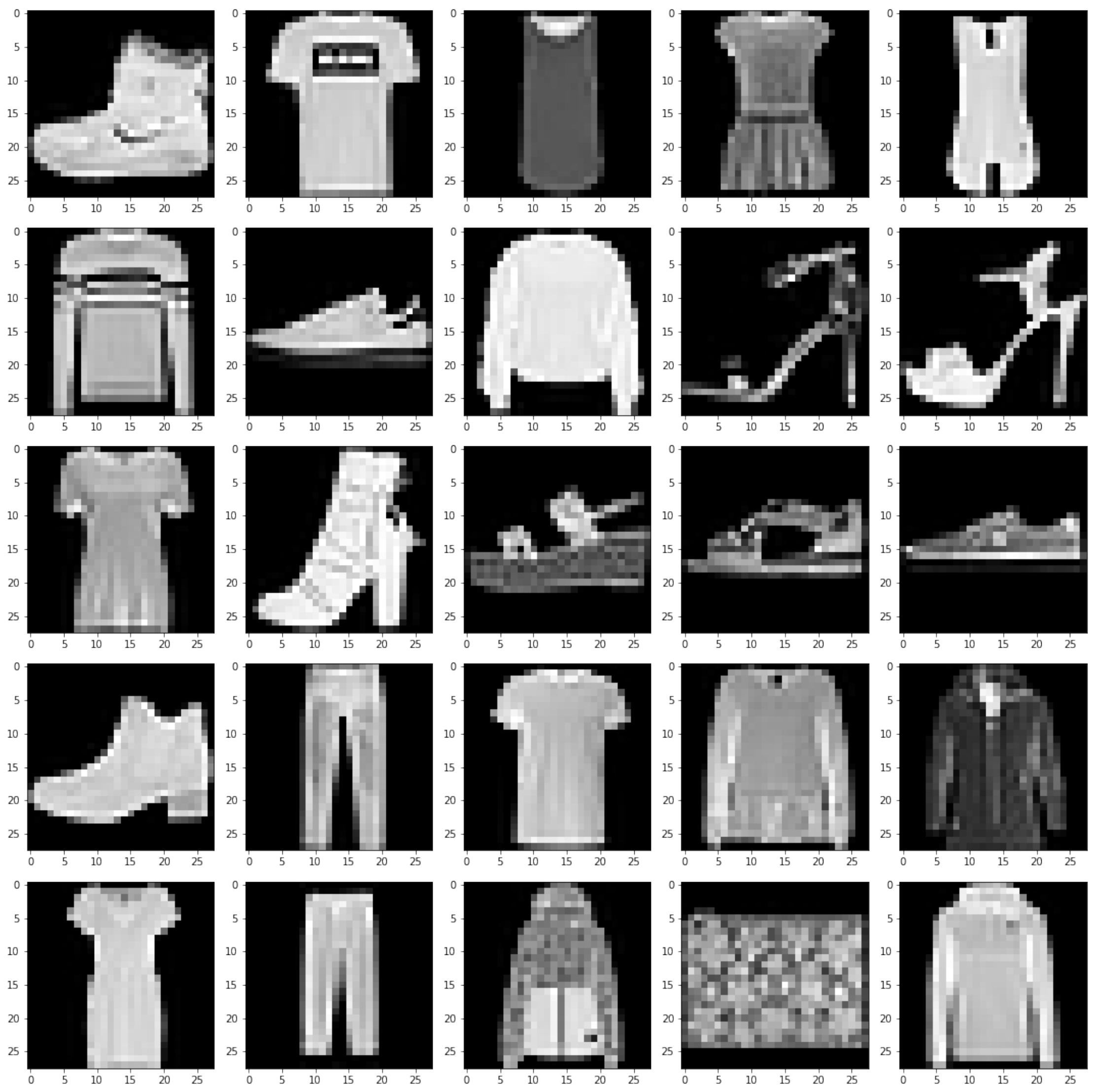
```
(10000, 28, 28)
```

1.3 Data Visualization

Plotting 25 images in order to understand and look at the dataset images. Here the plotting code and the output are shown.

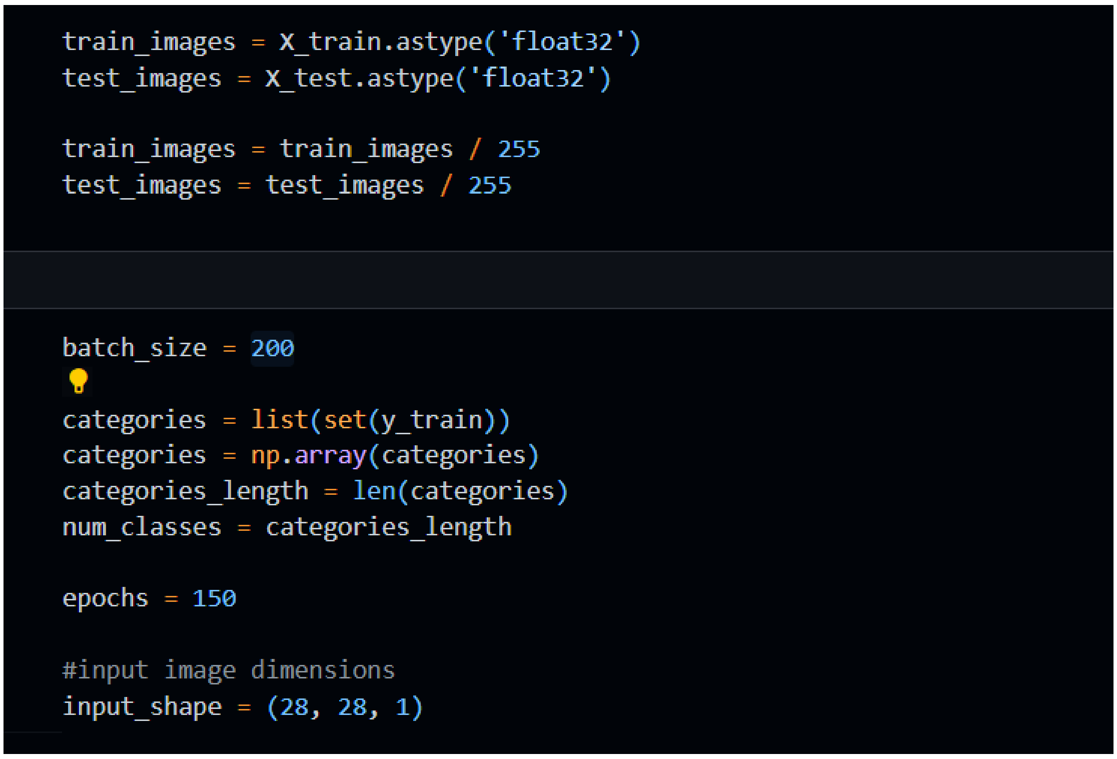


```
plt.figure(figsize=(15,15))
# plot first few images
for i in range(25):
    #define subplot
    plt.subplot(5,5,i+1)
    #plot raw pixel data
    plt.imshow(X_train[i].reshape(28,28), cmap='gray',interpolation='none')
    plt.tight_layout()
#show the figure
plt.show()
```


0
5
10
15
20
25

1.4 Data Preprocessing

This code snippet indicates a slight change in the training dataset by normalising the image pixels by dividing 255. Also, we have taken the number of classes, epochs, input shape and batch size for our model building purposes.



```
train_images = X_train.astype('float32')
test_images = X_test.astype('float32')

train_images = train_images / 255
test_images = test_images / 255
```

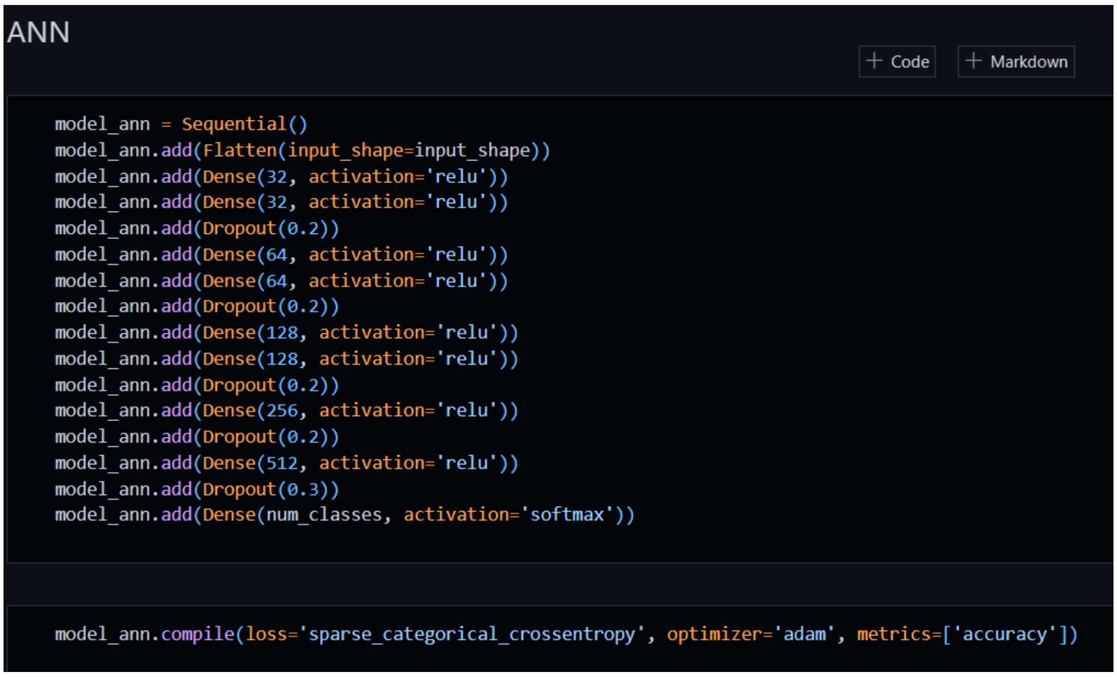
```
batch_size = 200

categories = list(set(y_train))
categories = np.array(categories)
categories_length = len(categories)
num_classes = categories_length

epochs = 150

#input image dimensions
input_shape = (28, 28, 1)
```

Chapter 2

Working with the Models (ANN)

2.1 Model Building and compiling for ANN

Firstly, for ANN, we are going to build a model that consists of 8 layers, excluding output layers. Additionally, we compiled the model using sparse categorical cross-entropy, as the classes start from 0 to 9. The classes are not binary. Adam is used as an optimizer. These parameters are also used for the upcoming CNN model.

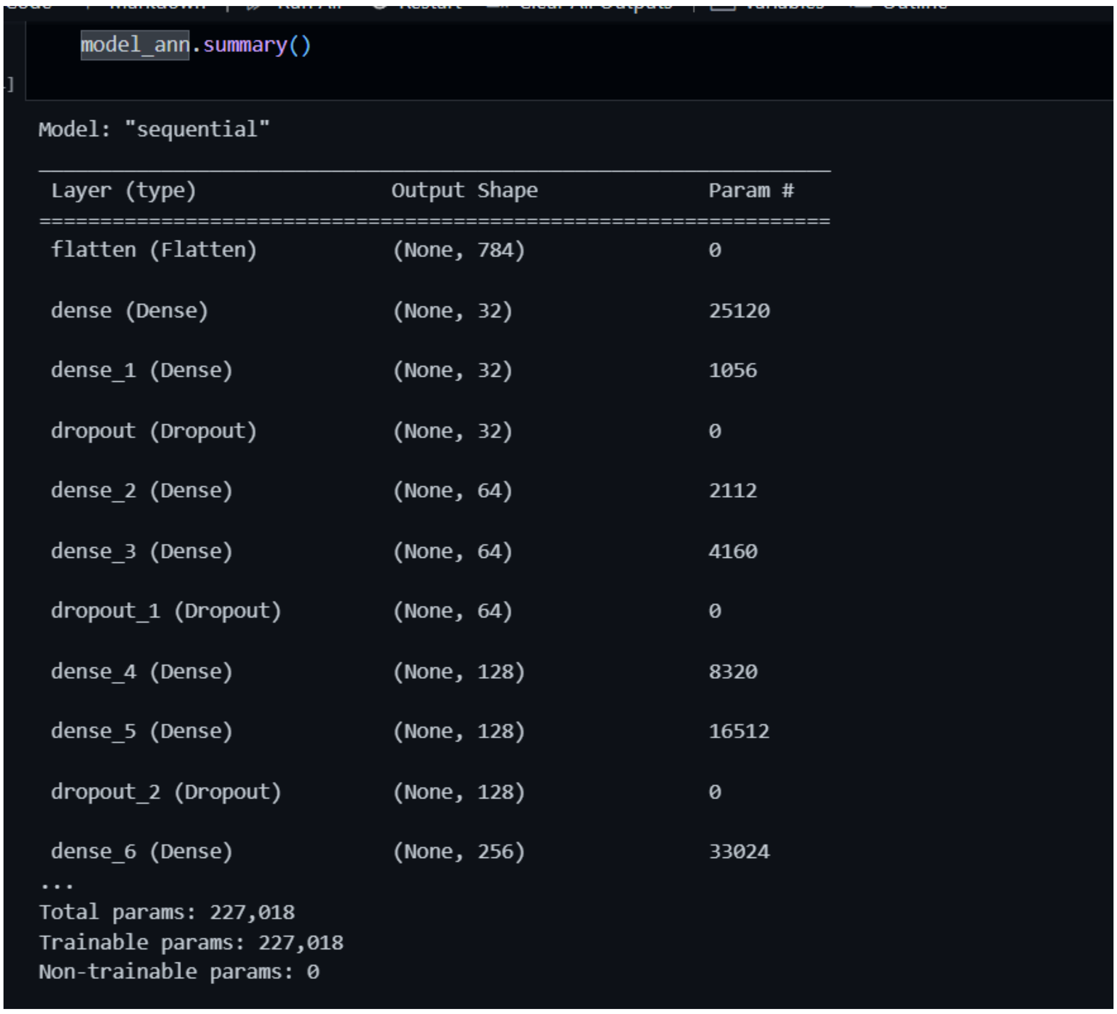


```
model_ann = Sequential()
model_ann.add(Flatten(input_shape=input_shape))
model_ann.add(Dense(32, activation='relu'))
model_ann.add(Dense(32, activation='relu'))
model_ann.add(Dropout(0.2))
model_ann.add(Dense(64, activation='relu'))
model_ann.add(Dense(64, activation='relu'))
model_ann.add(Dropout(0.2))
model_ann.add(Dense(128, activation='relu'))
model_ann.add(Dense(128, activation='relu'))
model_ann.add(Dropout(0.2))
model_ann.add(Dense(256, activation='relu'))
model_ann.add(Dropout(0.2))
model_ann.add(Dense(512, activation='relu'))
model_ann.add(Dropout(0.3))
model_ann.add(Dense(num_classes, activation='softmax'))
```

```
model_ann.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

2.2 Model Summary for ANN

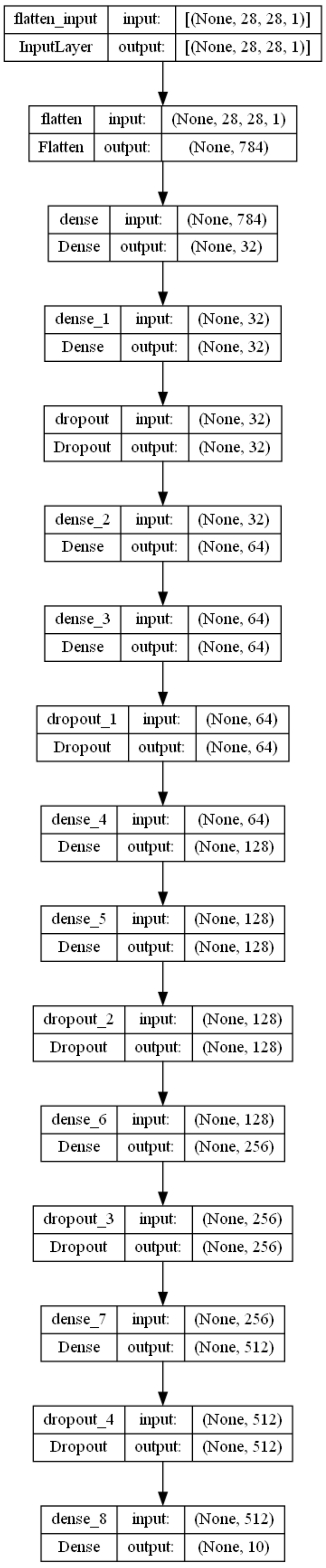
The model summary shows us the architecture of the model that was built earlier. Here we can see the shapes in each layer and also the parameters. It might help in adjusting the models if needed and also give insights into how model layers are changing with respect to shapes and parameters.



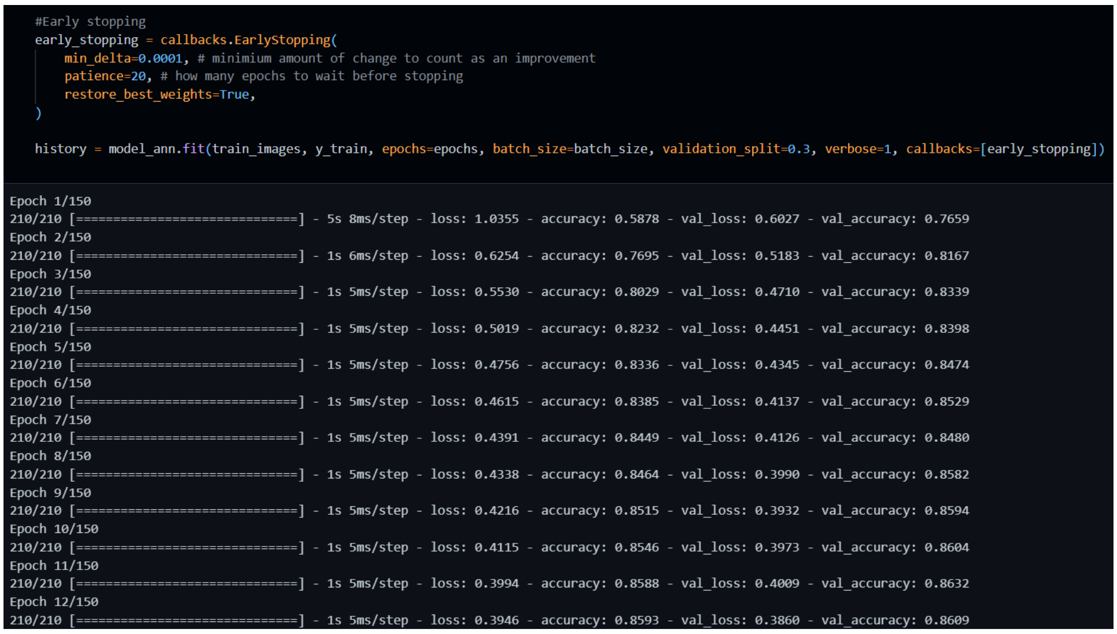
```
model_ann.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 32)                25120

 dense_1 (Dense)             (None, 32)                1056

 dropout (Dropout)           (None, 32)                0

 dense_2 (Dense)             (None, 64)                2112

 dense_3 (Dense)             (None, 64)                4160

 dropout_1 (Dropout)         (None, 64)                0

 dense_4 (Dense)             (None, 128)               8320

 dense_5 (Dense)             (None, 128)               16512

 dropout_2 (Dropout)         (None, 128)               0

 dense_6 (Dense)             (None, 256)               33024
...
Total params: 227,018
Trainable params: 227,018
Non-trainable params: 0
```

2.3 Plotting Model for ANN


flatten_input | input: | [(None, 28, 28, 1)]
InputLayer | output: | [(None, 28, 28, 1)]
flatten | input: | (None, 28, 28, 1)
Flatten | output: | (None, 784)
dense | input: | (None, 784)
Dense | output: | (None, 32)
dense_1 | input: | (None, 32)
Dense | output: | (None, 32)
dropout | input: | (None, 32)
Dropout | output: | (None, 32)
dense_2 | input: | (None, 32)
Dense | output: | (None, 64)
dense_3 | input: | (None, 64)
Dense | output: | (None, 64)
dropout_1 | input: | (None, 64)
Dropout | output: | (None, 64)
dense_4 | input: | (None, 64)
Dense | output: | (None, 128)
dense_5 | input: | (None, 128)
Dense | output: | (None, 128)
dropout_2 | input: | (None, 128)
Dropout | output: | (None, 128)
dense_6 | input: | (None, 128)
Dense | output: | (None, 256)
dropout_3 | input: | (None, 256)
Dropout | output: | (None, 256)
dense_7 | input: | (None, 256)
Dense | output: | (None, 512)
dropout_4 | input: | (None, 512)
Dropout | output: | (None, 512)
dense_8 | input: | (None, 512)
Dense | output: | (None, 10)

2.4 Fitting the Model for ANN

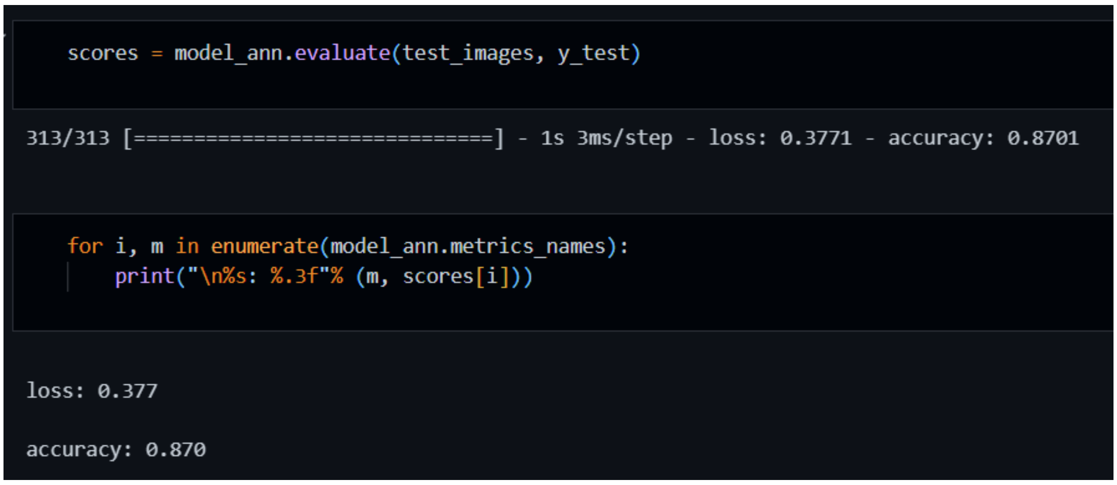


```
#Early stopping
early_stopping = callbacks.EarlyStopping(
    min_delta=0.0001, # minimium amount of change to count as an improvement
    patience=20, # how many epochs to wait before stopping
    restore_best_weights=True,
)

history = model_ann.fit(train_images, y_train, epochs=epochs, batch_size=batch_size, validation_split=0.3, verbose=1, callbacks=[early_stopping])
```

```
Epoch 1/150
210/210 [==============================] - 5s 8ms/step - loss: 1.0355 - accuracy: 0.5878 - val_loss: 0.6027 - val_accuracy: 0.7659
Epoch 2/150
210/210 [==============================] - 1s 6ms/step - loss: 0.6254 - accuracy: 0.7695 - val_loss: 0.5183 - val_accuracy: 0.8167
Epoch 3/150
210/210 [==============================] - 1s 5ms/step - loss: 0.5530 - accuracy: 0.8029 - val_loss: 0.4710 - val_accuracy: 0.8339
Epoch 4/150
210/210 [==============================] - 1s 5ms/step - loss: 0.5019 - accuracy: 0.8232 - val_loss: 0.4451 - val_accuracy: 0.8398
Epoch 5/150
210/210 [==============================] - 1s 5ms/step - loss: 0.4756 - accuracy: 0.8336 - val_loss: 0.4345 - val_accuracy: 0.8474
Epoch 6/150
210/210 [==============================] - 1s 5ms/step - loss: 0.4615 - accuracy: 0.8385 - val_loss: 0.4137 - val_accuracy: 0.8529
Epoch 7/150
210/210 [==============================] - 1s 5ms/step - loss: 0.4391 - accuracy: 0.8449 - val_loss: 0.4126 - val_accuracy: 0.8480
Epoch 8/150
210/210 [==============================] - 1s 5ms/step - loss: 0.4338 - accuracy: 0.8464 - val_loss: 0.3990 - val_accuracy: 0.8582
Epoch 9/150
210/210 [==============================] - 1s 5ms/step - loss: 0.4216 - accuracy: 0.8515 - val_loss: 0.3932 - val_accuracy: 0.8594
Epoch 10/150
210/210 [==============================] - 1s 5ms/step - loss: 0.4115 - accuracy: 0.8546 - val_loss: 0.3973 - val_accuracy: 0.8604
Epoch 11/150
210/210 [==============================] - 1s 5ms/step - loss: 0.3994 - accuracy: 0.8588 - val_loss: 0.4009 - val_accuracy: 0.8632
Epoch 12/150
210/210 [==============================] - 1s 5ms/step - loss: 0.3946 - accuracy: 0.8593 - val_loss: 0.3860 - val_accuracy: 0.8609
```

2.5 Evaluating the Model for ANN

After fitting the model, we get to evaluate the ANN model based on the 10,000 test data. Here we got 38% loss and 87% accuracy.



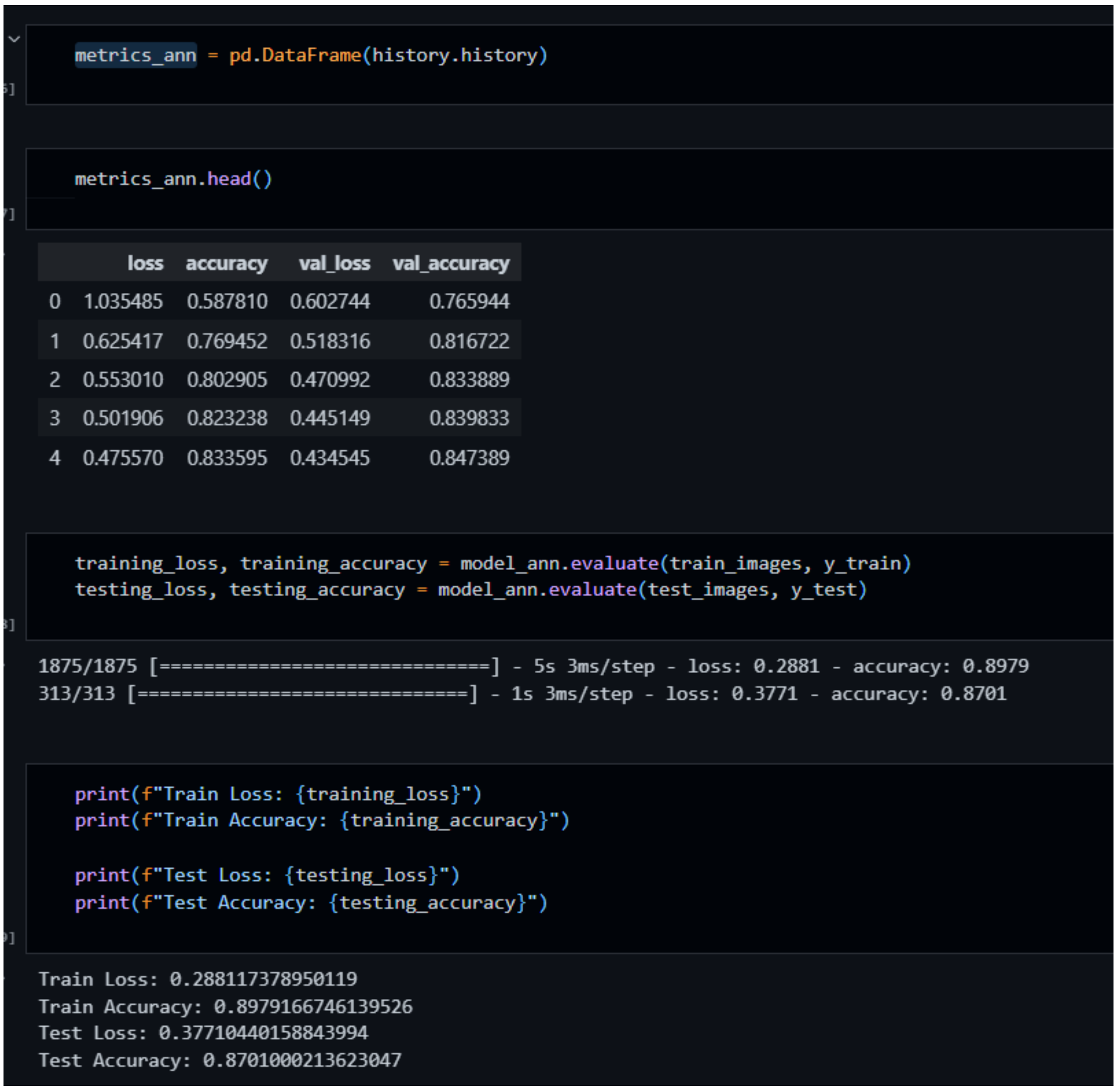
```
scores = model_ann.evaluate(test_images, y_test)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.3771 - accuracy: 0.8701
```

```
for i, m in enumerate(model_ann.metrics_names):
    print("\n%s: %.3f"% (m, scores[i]))
```

```

loss: 0.377

accuracy: 0.870
```

Also, show in detail the train loss, train accuracy, validation loss, and validation accuracy.



```
metrics_ann = pd.DataFrame(history.history)
```

```
metrics_ann.head()
```

	loss	accuracy	val_loss	val_accuracy
0	1.035485	0.587810	0.602744	0.765944
1	0.625417	0.769452	0.518316	0.816722
2	0.553010	0.802905	0.470992	0.833889
3	0.501906	0.823238	0.445149	0.839833
4	0.475570	0.833595	0.434545	0.847389

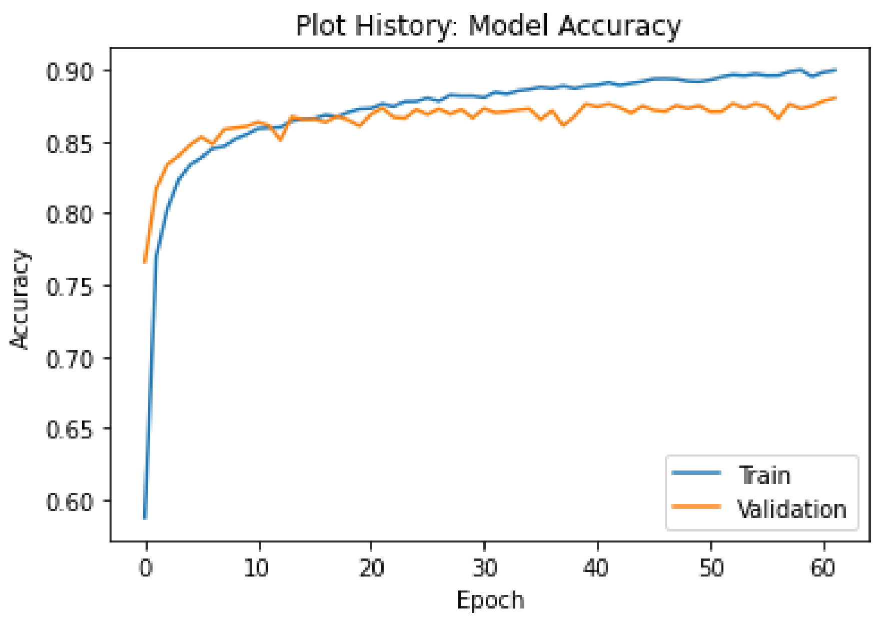
```
training_loss, training_accuracy = model_ann.evaluate(train_images, y_train)
testing_loss, testing_accuracy = model_ann.evaluate(test_images, y_test)
```

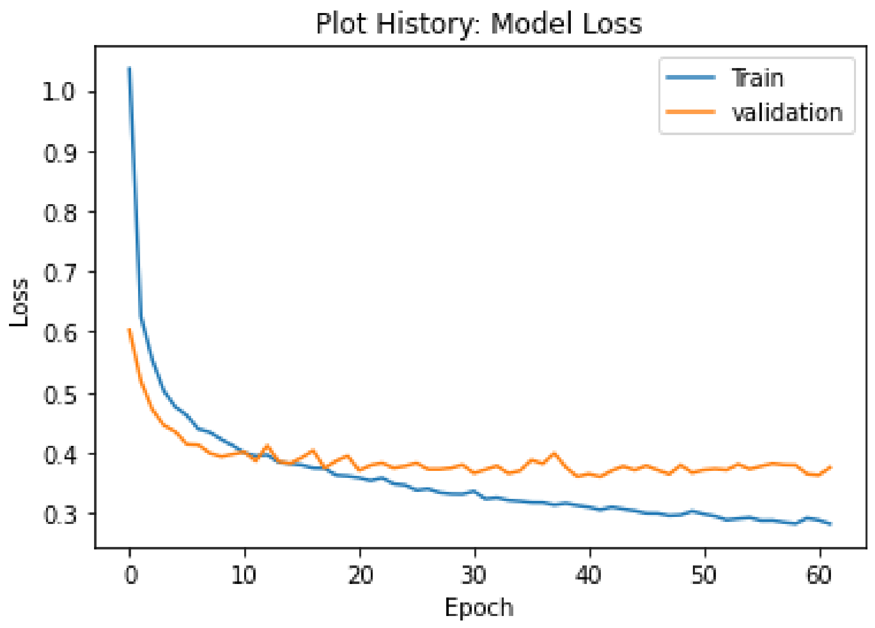
```
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2881 - accuracy: 0.8979
313/313 [==============================] - 1s 3ms/step - loss: 0.3771 - accuracy: 0.8701
```

```
print(f"Train Loss: {training_loss}")
print(f"Train Accuracy: {training_accuracy}")

print(f"Test Loss: {testing_loss}")
print(f"Test Accuracy: {testing_accuracy}")
```

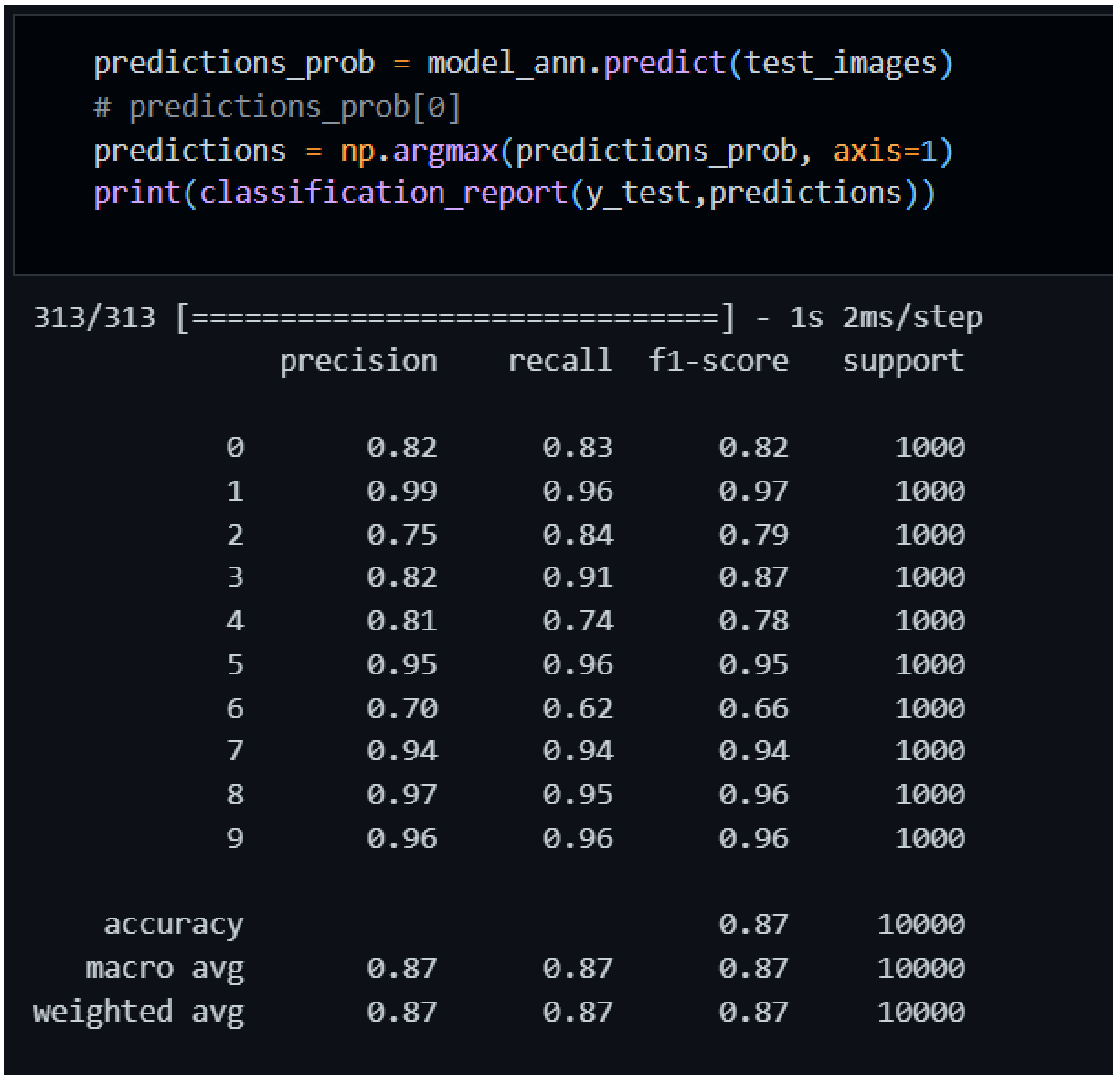
```
Train Loss: 0.288117378950119
Train Accuracy: 0.8979166746139526
Test Loss: 0.37710440158843994
Test Accuracy: 0.8701000213623047
```

From this model evaluation training accuracy vs validation accuracy and training loss vs validation loss can also be depicted.





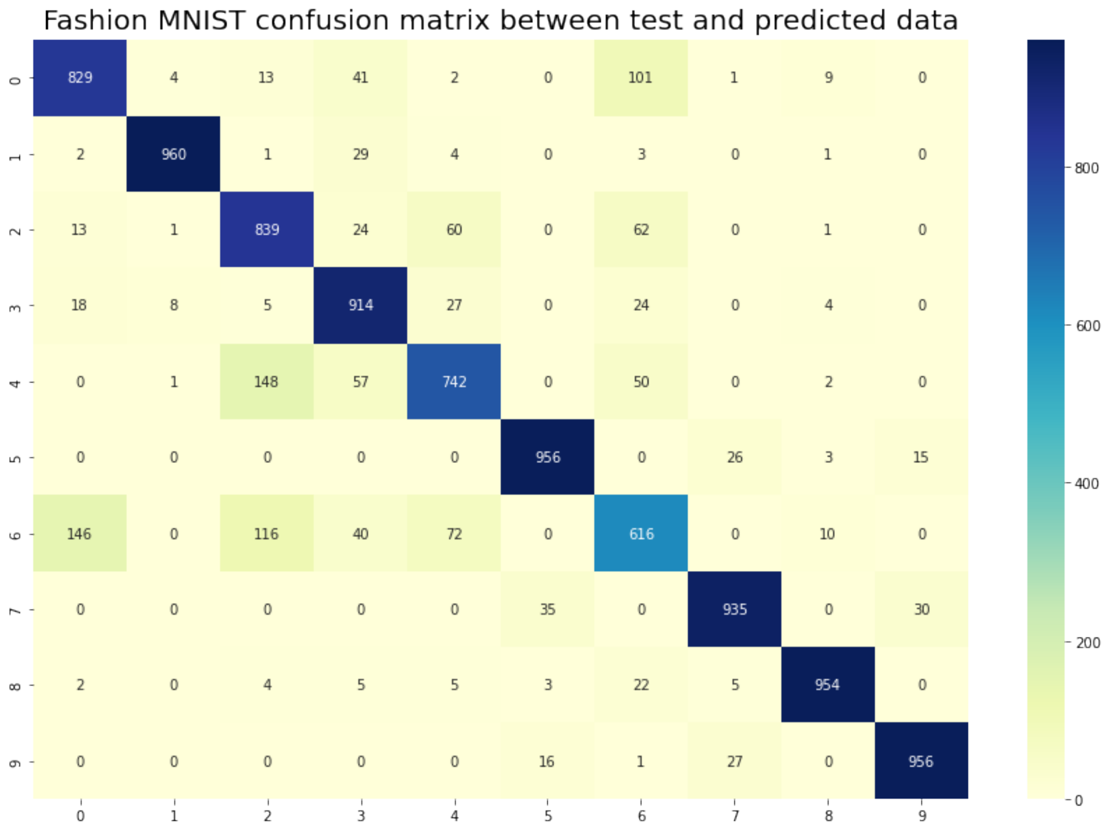
Here we are showing the classification report consisting of precision recall and f-1 score for every class.



```
predictions_prob = model_ann.predict(test_images)
# predictions_prob[0]
predictions = np.argmax(predictions_prob, axis=1)
print(classification_report(y_test,predictions))
```

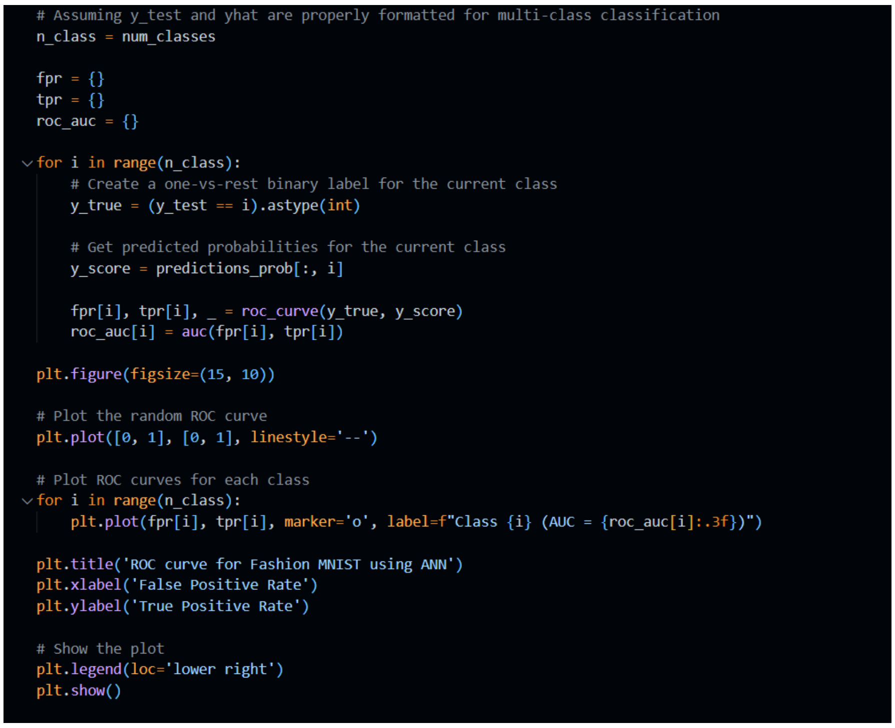
```
313/313 [==============================] - 1s 2ms/step
              precision    recall  f1-score   support

           0       0.82      0.83      0.82      1000
           1       0.99      0.96      0.97      1000
           2       0.75      0.84      0.79      1000
           3       0.82      0.91      0.87      1000
           4       0.81      0.74      0.78      1000
           5       0.95      0.96      0.95      1000
           6       0.70      0.62      0.66      1000
           7       0.94      0.94      0.94      1000
           8       0.97      0.95      0.96      1000
           9       0.96      0.96      0.96      1000

    accuracy                           0.87     10000
   macro avg       0.87      0.87      0.87     10000
weighted avg       0.87      0.87      0.87     10000
```

Confusion Matrix for True data that comes from test data and our models predicted data

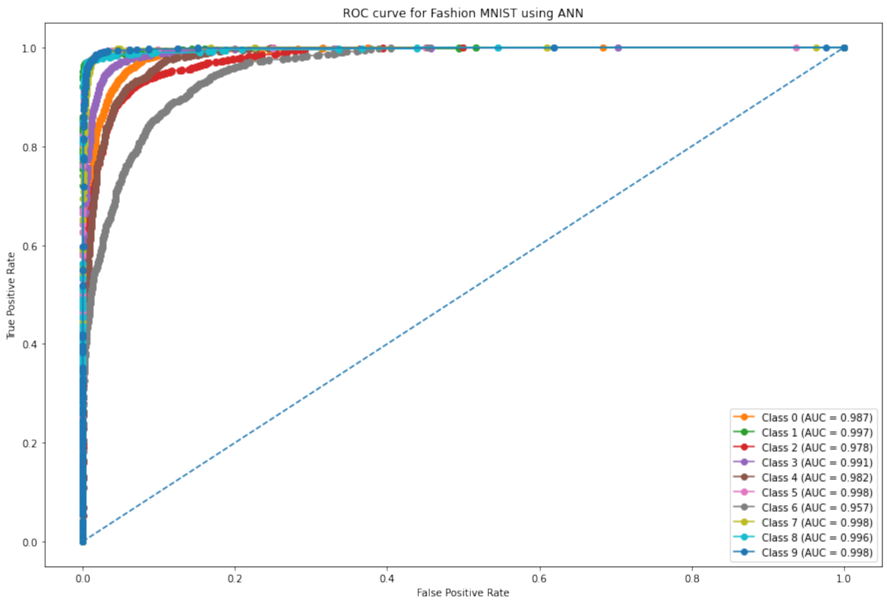


	0	1	2	3	4	5	6	7	8	9
0	829	4	13	41	2	0	101	1	9	0
1	2	960	1	29	4	0	3	0	1	0
2	13	1	839	24	60	0	62	0	1	0
3	18	8	5	914	27	0	24	0	4	0
4	0	1	148	57	742	0	50	0	2	0
5	0	0	0	0	0	956	0	26	3	15
6	146	0	116	40	72	0	616	0	10	0
7	0	0	0	0	0	35	0	935	0	30
8	2	0	4	5	5	3	22	5	954	0
9	0	0	0	0	0	16	1	27	0	956

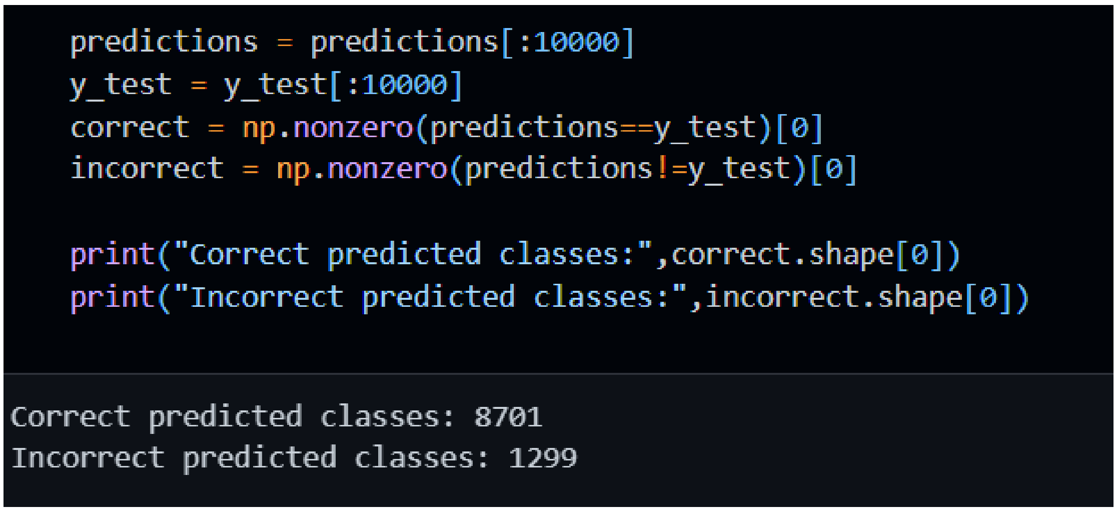
Now we will present the ROC curve for this model. In order to understand if a ROC is good or bad, we need to know if the true positive rate, or sensitivity, will increase, and if the area under the curve (AUC) is close to 1, the model is performing well. So, from the below code and plot, it can be seen that the model is performing well for class 0 to 9.



```python
# Assuming y_test and yhat are properly formatted for multi-class classification
n_class = num_classes

fpr = {}
tpr = {}
roc_auc = {}

for i in range(n_class):
    # Create a one-vs-rest binary label for the current class
    y_true = (y_test == i).astype(int)

    # Get predicted probabilities for the current class
    y_score = predictions_prob[:, i]

    fpr[i], tpr[i], _ = roc_curve(y_true, y_score)
    roc_auc[i] = auc(fpr[i], tpr[i])

plt.figure(figsize=(15, 10))

# Plot the random ROC curve
plt.plot([0, 1], [0, 1], linestyle='--')

# Plot ROC curves for each class
for i in range(n_class):
    plt.plot(fpr[i], tpr[i], marker='o', label=f"Class {i} (AUC = {roc_auc[i]:.3f})")

plt.title('ROC curve for Fashion MNIST using ANN')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')

# Show the plot
plt.legend(loc='lower right')
plt.show()
```

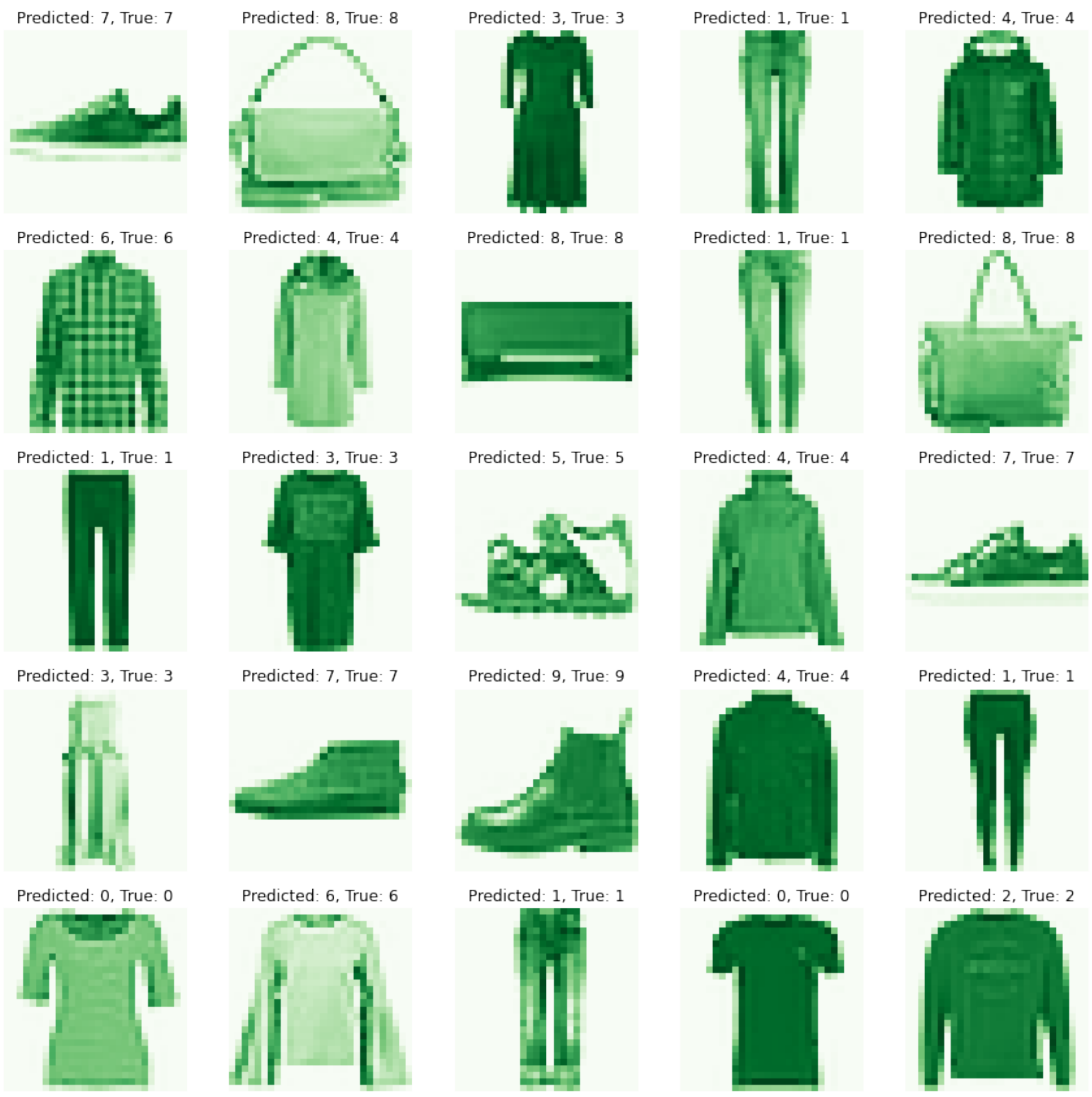


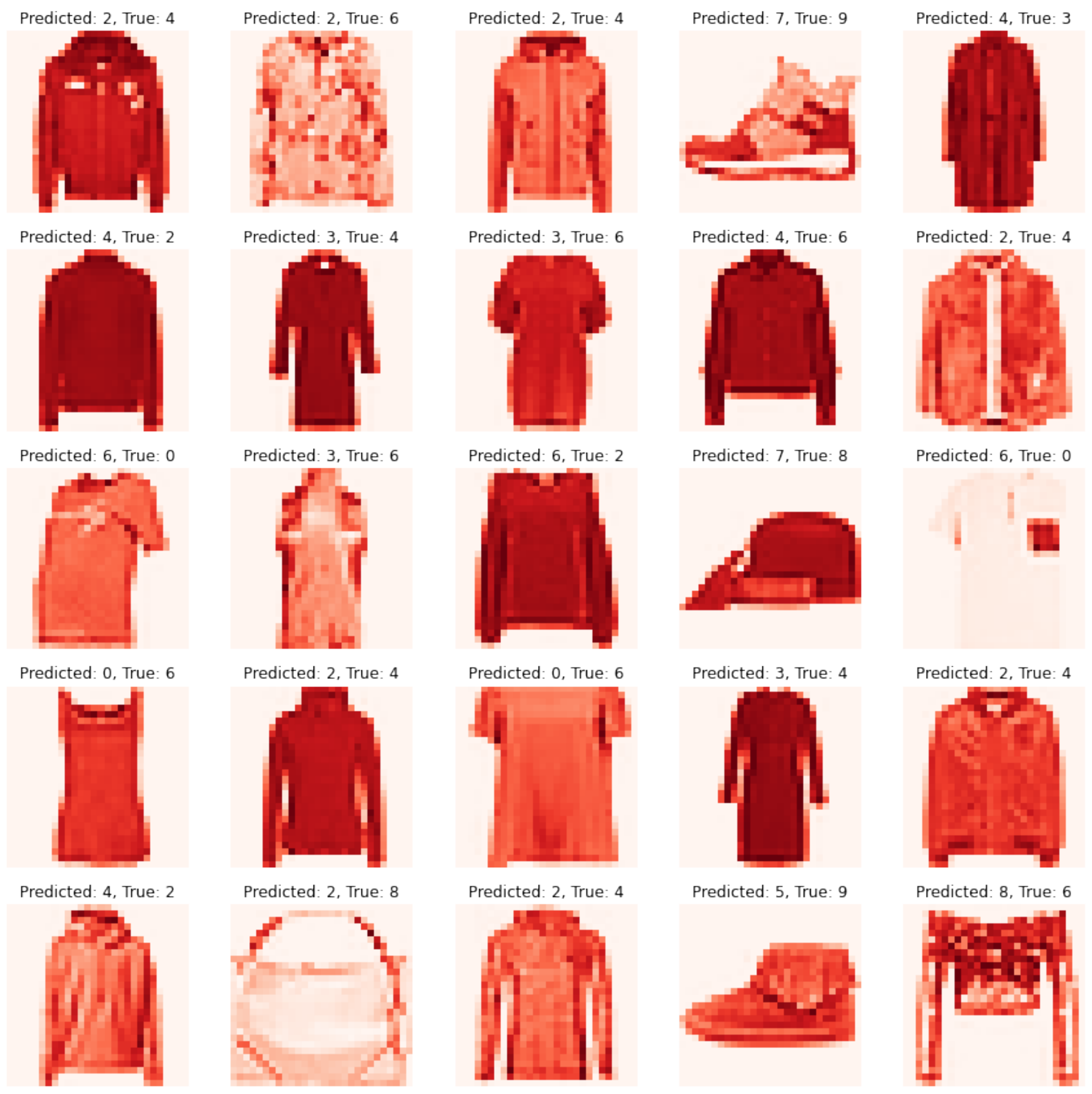
To calculate and visualise correctly predicted and miss-classified images, we performed the below code. From here, it can be seen that this ANN model predicts 8701 images correctly and 1299 images incorrectly out of 10,000.



```
    predictions = predictions[:10000]
    y_test = y_test[:10000]
    correct = np.nonzero(predictions==y_test)[0]
    incorrect = np.nonzero(predictions!=y_test)[0]

    print("Correct predicted classes:",correct.shape[0])
    print("Incorrect predicted classes:",incorrect.shape[0])
```

```
Correct predicted classes: 8701
Incorrect predicted classes: 1299
```



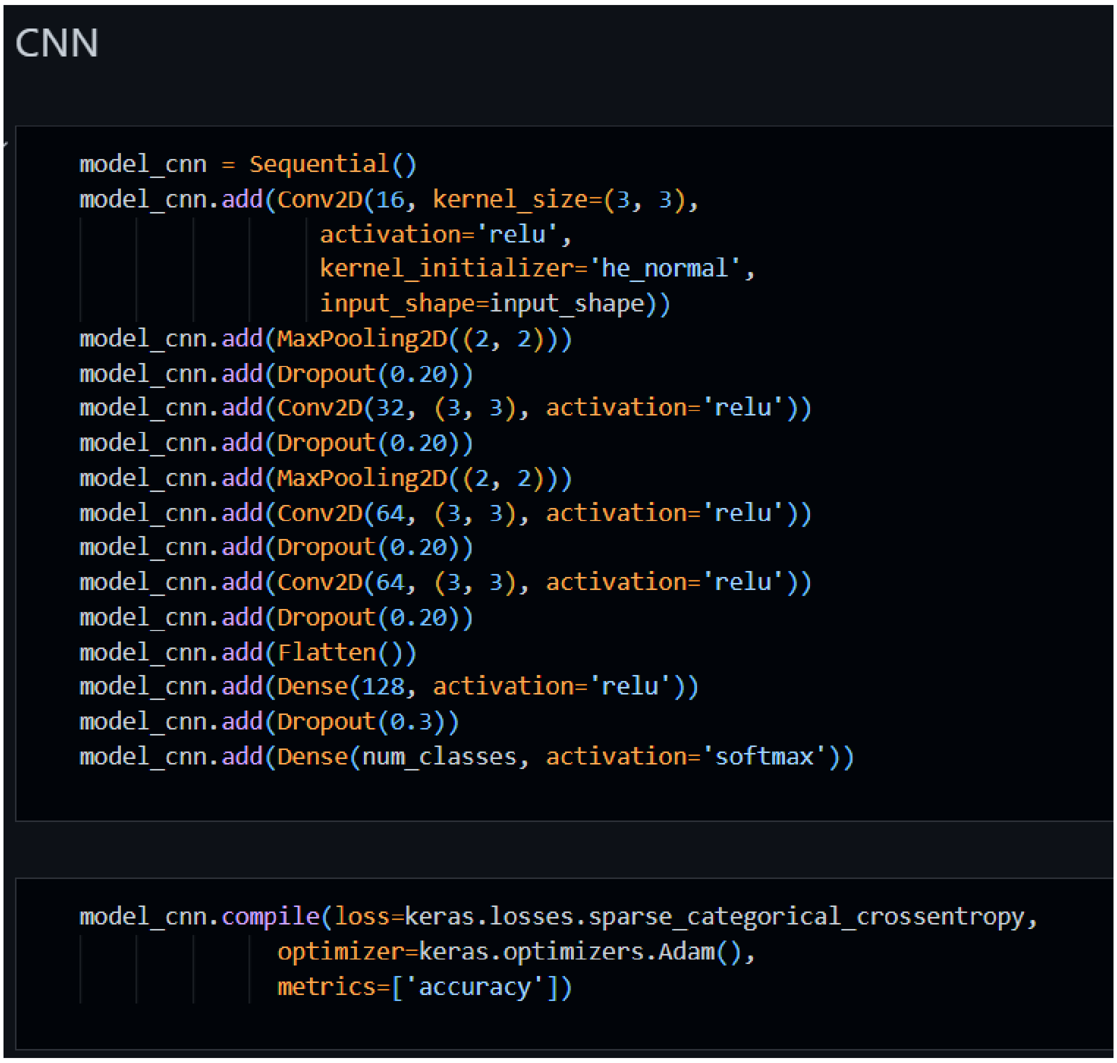

Predicted: 2, True: 4
Predicted: 2, True: 6
Predicted: 2, True: 4
Predicted: 7, True: 9
Predicted: 4, True: 3
Predicted: 4, True: 2
Predicted: 3, True: 4
Predicted: 3, True: 6
Predicted: 4, True: 6
Predicted: 2, True: 4
Predicted: 6, True: 0
Predicted: 3, True: 6
Predicted: 6, True: 2
Predicted: 7, True: 8
Predicted: 6, True: 0
Predicted: 0, True: 6
Predicted: 2, True: 4
Predicted: 0, True: 6
Predicted: 3, True: 4
Predicted: 2, True: 4
Predicted: 4, True: 2
Predicted: 2, True: 8
Predicted: 2, True: 4
Predicted: 5, True: 9
Predicted: 8, True: 6

Chapter 3

Working with Models (CNN)

3.1 Model Building and compiling for CNN

Firstly, for CNN, we are going to build a model that consists of 5 layers, excluding output layers. Additionally, we compiled the model using sparse categorical cross-entropy, as the classes start from 0 to 9. The classes are not binary. Adam is used as an optimizer. These parameters are also used for the upcoming CNN model.



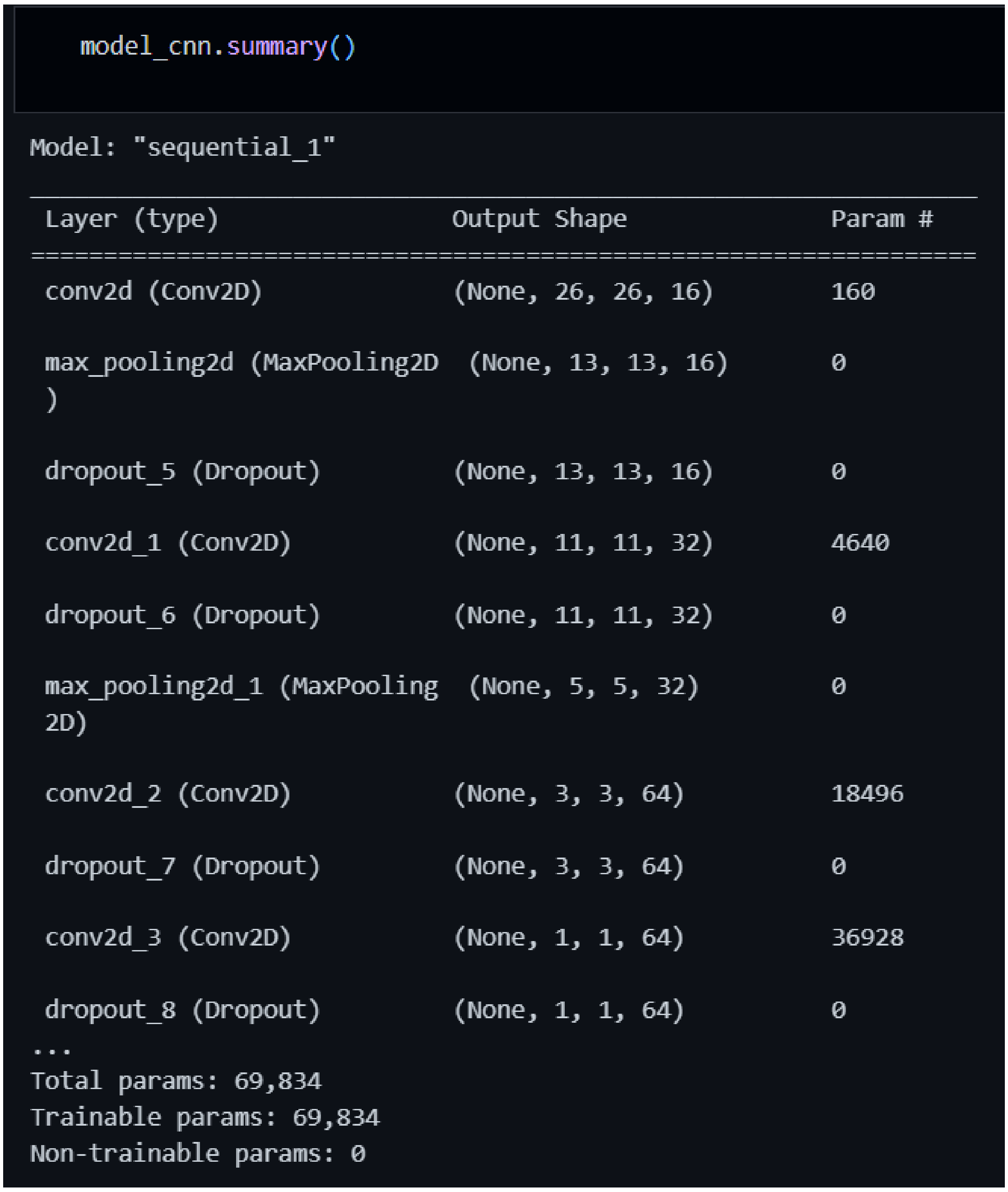
CNN

```
model_cnn = Sequential()
model_cnn.add(Conv2D(16, kernel_size=(3, 3),
                     activation='relu',
                     kernel_initializer='he_normal',
                     input_shape=input_shape))
model_cnn.add(MaxPooling2D((2, 2)))
model_cnn.add(Dropout(0.20))
model_cnn.add(Conv2D(32, (3, 3), activation='relu'))
model_cnn.add(Dropout(0.20))
model_cnn.add(MaxPooling2D((2, 2)))
model_cnn.add(Conv2D(64, (3, 3), activation='relu'))
model_cnn.add(Dropout(0.20))
model_cnn.add(Conv2D(64, (3, 3), activation='relu'))
model_cnn.add(Dropout(0.20))
model_cnn.add(Flatten())
model_cnn.add(Dense(128, activation='relu'))
model_cnn.add(Dropout(0.3))
model_cnn.add(Dense(num_classes, activation='softmax'))
```

```
model_cnn.compile(loss=keras.losses.sparse_categorical_crossentropy,
                  optimizer=keras.optimizers.Adam(),
                  metrics=['accuracy'])
```

3.2 Model Summary for CNN

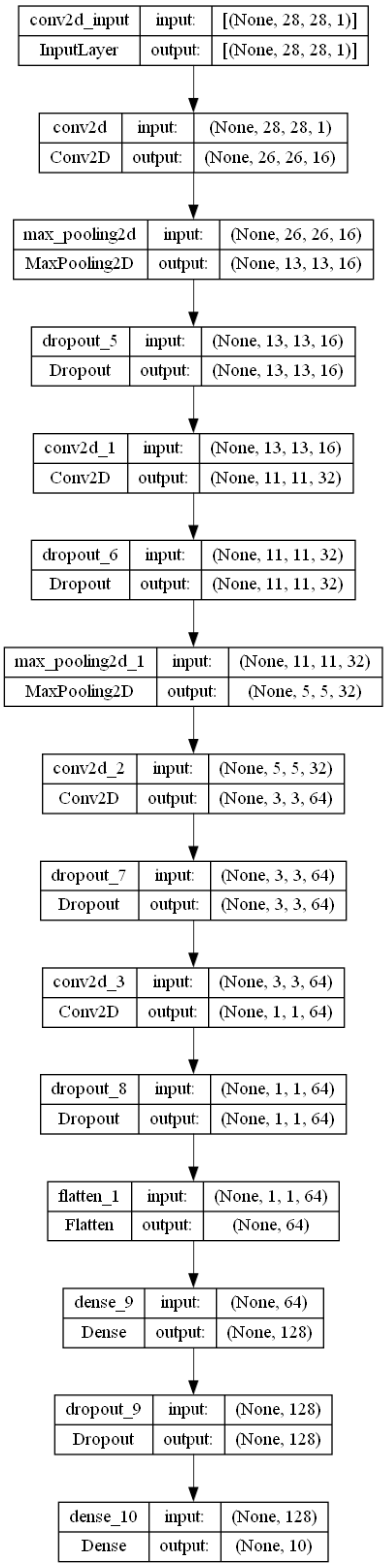
The model summary illustrates the architecture of the model that was previously generated. The shapes in each layer, as well as the parameters, are visible here. It may aid in model adjustment if necessary, as well as provide insight into how model layers change in relation to shapes and parameters.



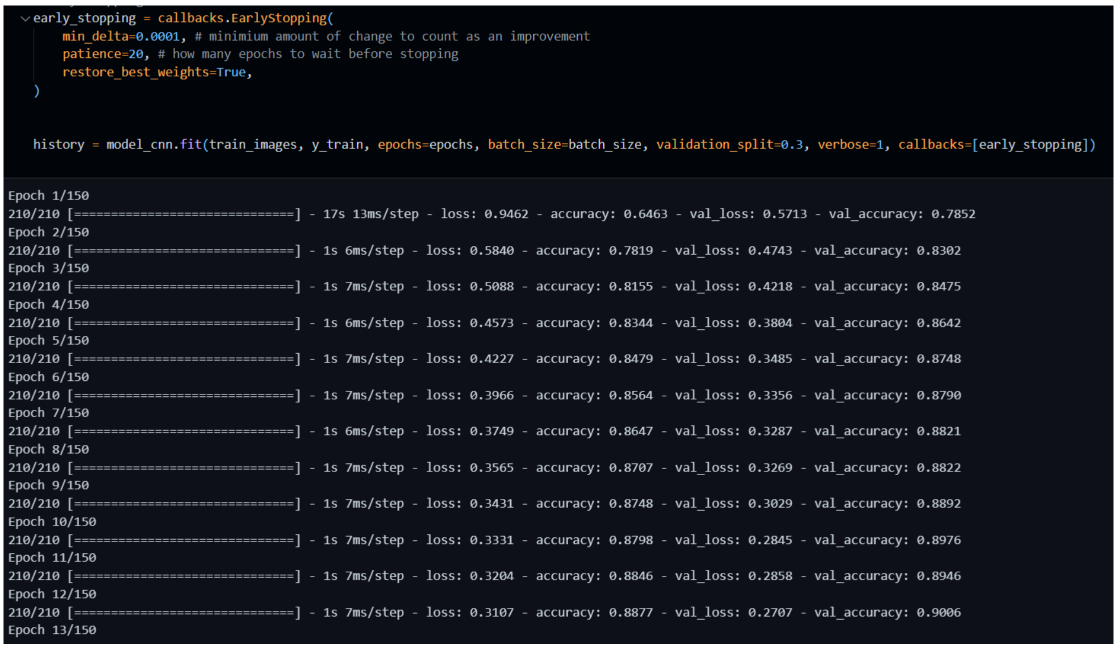
```
model_cnn.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 16)        160

 max_pooling2d (MaxPooling2D  (None, 13, 13, 16)       0
 )

 dropout_5 (Dropout)         (None, 13, 13, 16)        0

 conv2d_1 (Conv2D)           (None, 11, 11, 32)        4640

 dropout_6 (Dropout)         (None, 11, 11, 32)        0

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 32)         0
 2D)

 conv2d_2 (Conv2D)           (None, 3, 3, 64)          18496

 dropout_7 (Dropout)         (None, 3, 3, 64)          0

 conv2d_3 (Conv2D)           (None, 1, 1, 64)          36928

 dropout_8 (Dropout)         (None, 1, 1, 64)          0
...
Total params: 69,834
Trainable params: 69,834
Non-trainable params: 0
```

3.3 Plotting Model for CNN


conv2d_input | input: | [(None, 28, 28, 1)]
InputLayer | output: | [(None, 28, 28, 1)]
conv2d | input: | (None, 28, 28, 1)
Conv2D | output: | (None, 26, 26, 16)
max_pooling2d | input: | (None, 26, 26, 16)
MaxPooling2D | output: | (None, 13, 13, 16)
dropout_5 | input: | (None, 13, 13, 16)
Dropout | output: | (None, 13, 13, 16)
conv2d_1 | input: | (None, 13, 13, 16)
Conv2D | output: | (None, 11, 11, 32)
dropout_6 | input: | (None, 11, 11, 32)
Dropout | output: | (None, 11, 11, 32)
max_pooling2d_1 | input: | (None, 11, 11, 32)
MaxPooling2D | output: | (None, 5, 5, 32)
conv2d_2 | input: | (None, 5, 5, 32)
Conv2D | output: | (None, 3, 3, 64)
dropout_7 | input: | (None, 3, 3, 64)
Dropout | output: | (None, 3, 3, 64)
conv2d_3 | input: | (None, 3, 3, 64)
Conv2D | output: | (None, 1, 1, 64)
dropout_8 | input: | (None, 1, 1, 64)
Dropout | output: | (None, 1, 1, 64)
flatten_1 | input: | (None, 1, 1, 64)
Flatten | output: | (None, 64)
dense_9 | input: | (None, 64)
Dense | output: | (None, 128)
dropout_9 | input: | (None, 128)
Dropout | output: | (None, 128)
dense_10 | input: | (None, 128)
Dense | output: | (None, 10)

3.4 Fitting the Model for CNN

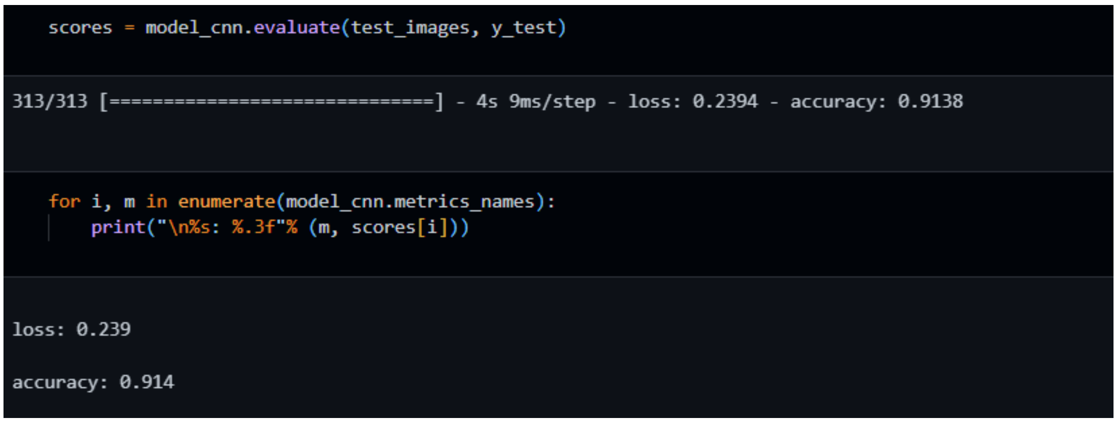


```
early_stopping = callbacks.EarlyStopping(
    min_delta=0.0001, # minimium amount of change to count as an improvement
    patience=20, # how many epochs to wait before stopping
    restore_best_weights=True,
)


history = model_cnn.fit(train_images, y_train, epochs=epochs, batch_size=batch_size, validation_split=0.3, verbose=1, callbacks=[early_stopping])
```

```
Epoch 1/150
210/210 [==============================] - 17s 13ms/step - loss: 0.9462 - accuracy: 0.6463 - val_loss: 0.5713 - val_accuracy: 0.7852
Epoch 2/150
210/210 [==============================] - 1s 6ms/step - loss: 0.5840 - accuracy: 0.7819 - val_loss: 0.4743 - val_accuracy: 0.8302
Epoch 3/150
210/210 [==============================] - 1s 7ms/step - loss: 0.5088 - accuracy: 0.8155 - val_loss: 0.4218 - val_accuracy: 0.8475
Epoch 4/150
210/210 [==============================] - 1s 6ms/step - loss: 0.4573 - accuracy: 0.8344 - val_loss: 0.3804 - val_accuracy: 0.8642
Epoch 5/150
210/210 [==============================] - 1s 7ms/step - loss: 0.4227 - accuracy: 0.8479 - val_loss: 0.3485 - val_accuracy: 0.8748
Epoch 6/150
210/210 [==============================] - 1s 7ms/step - loss: 0.3966 - accuracy: 0.8564 - val_loss: 0.3356 - val_accuracy: 0.8790
Epoch 7/150
210/210 [==============================] - 1s 6ms/step - loss: 0.3749 - accuracy: 0.8647 - val_loss: 0.3287 - val_accuracy: 0.8821
Epoch 8/150
210/210 [==============================] - 1s 7ms/step - loss: 0.3565 - accuracy: 0.8707 - val_loss: 0.3269 - val_accuracy: 0.8822
Epoch 9/150
210/210 [==============================] - 1s 7ms/step - loss: 0.3431 - accuracy: 0.8748 - val_loss: 0.3029 - val_accuracy: 0.8892
Epoch 10/150
210/210 [==============================] - 1s 7ms/step - loss: 0.3331 - accuracy: 0.8798 - val_loss: 0.2845 - val_accuracy: 0.8976
Epoch 11/150
210/210 [==============================] - 1s 7ms/step - loss: 0.3204 - accuracy: 0.8846 - val_loss: 0.2858 - val_accuracy: 0.8946
Epoch 12/150
210/210 [==============================] - 1s 7ms/step - loss: 0.3107 - accuracy: 0.8877 - val_loss: 0.2707 - val_accuracy: 0.9006
Epoch 13/150
```

3.5 Evaluating the Model for CNN

After fitting the model, we get to evaluate the CNN model based on the 10,000 test data. Here we got 24% loss and 91% accuracy.



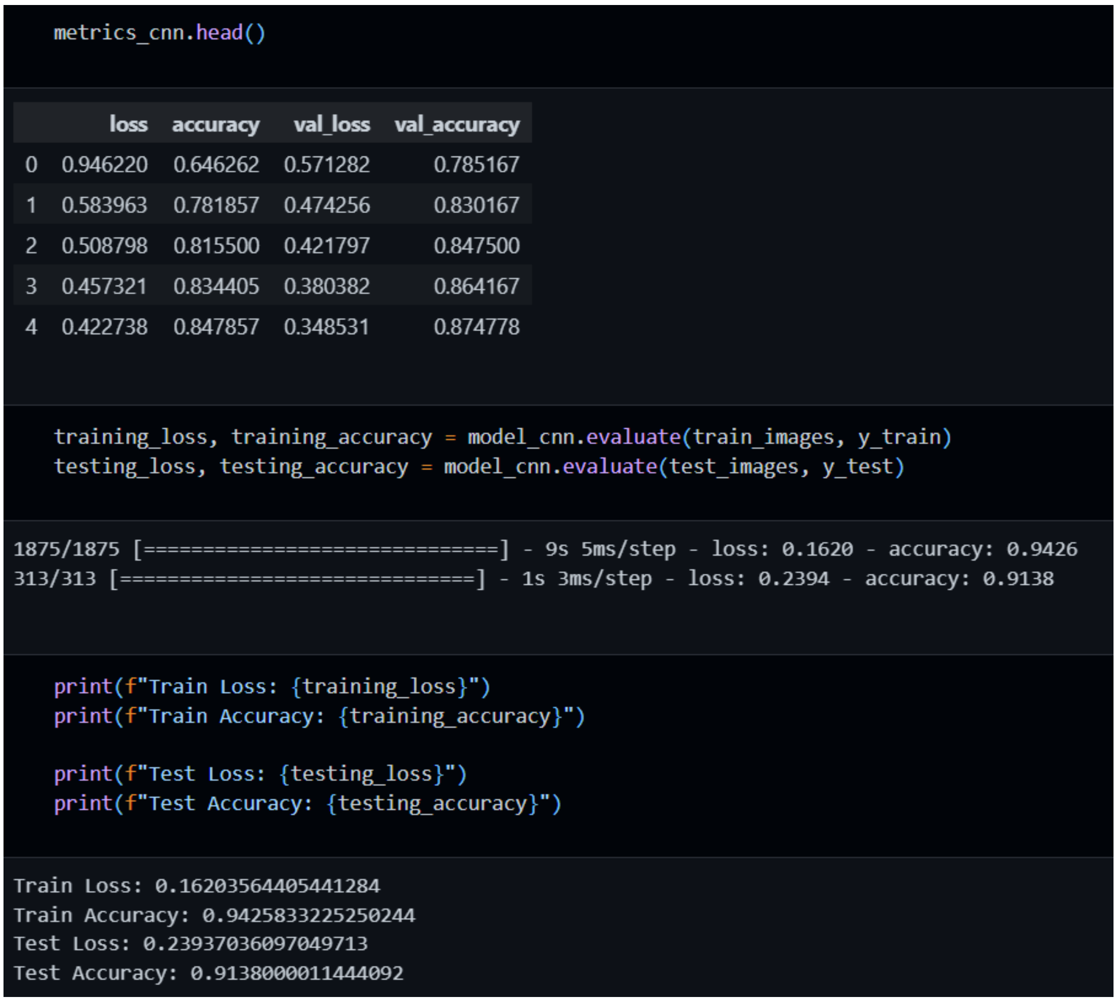
```
scores = model_cnn.evaluate(test_images, y_test)
```

```
313/313 [==============================] - 4s 9ms/step - loss: 0.2394 - accuracy: 0.9138
```

```
for i, m in enumerate(model_cnn.metrics_names):
    print("\n%s: %.3f"% (m, scores[i]))
```

```

loss: 0.239

accuracy: 0.914
```

Also, show in detail the train loss, train accuracy, validation loss, and validation accuracy.



```
metrics_cnn.head()
```

	loss	accuracy	val_loss	val_accuracy
0	0.946220	0.646262	0.571282	0.785167
1	0.583963	0.781857	0.474256	0.830167
2	0.508798	0.815500	0.421797	0.847500
3	0.457321	0.834405	0.380382	0.864167
4	0.422738	0.847857	0.348531	0.874778

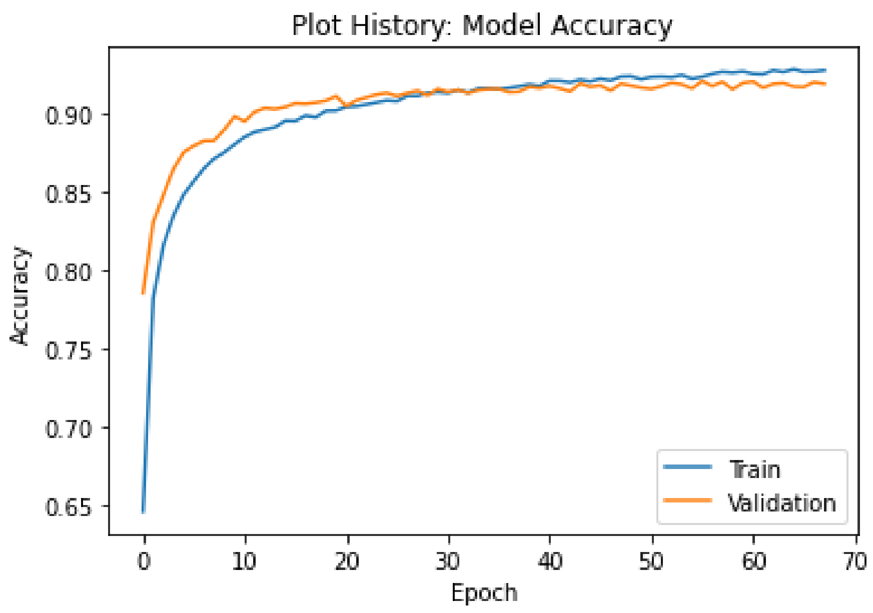
```
training_loss, training_accuracy = model_cnn.evaluate(train_images, y_train)
testing_loss, testing_accuracy = model_cnn.evaluate(test_images, y_test)
```

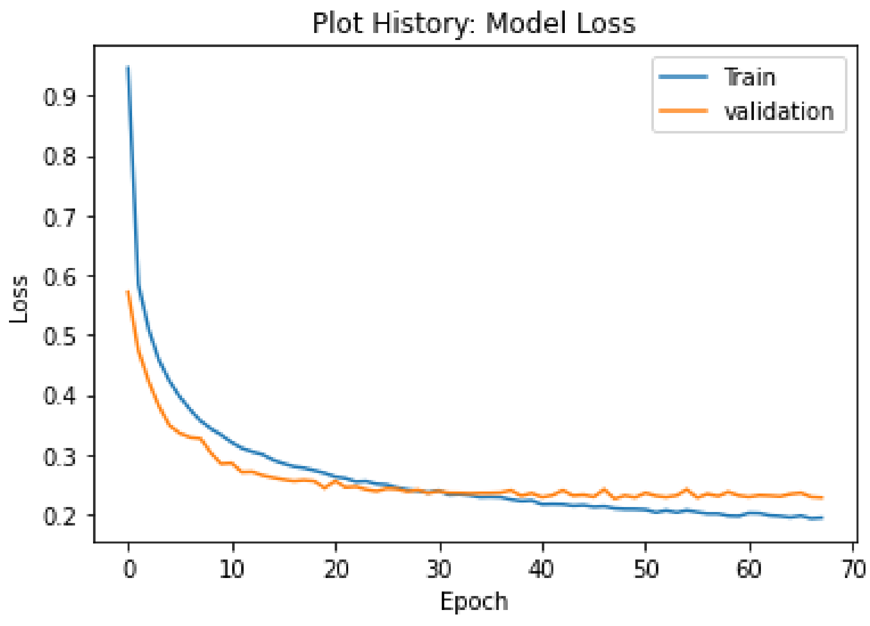
```
1875/1875 [==============================] - 9s 5ms/step - loss: 0.1620 - accuracy: 0.9426
313/313 [==============================] - 1s 3ms/step - loss: 0.2394 - accuracy: 0.9138
```

```
print(f"Train Loss: {training_loss}")
print(f"Train Accuracy: {training_accuracy}")

print(f"Test Loss: {testing_loss}")
print(f"Test Accuracy: {testing_accuracy}")
```

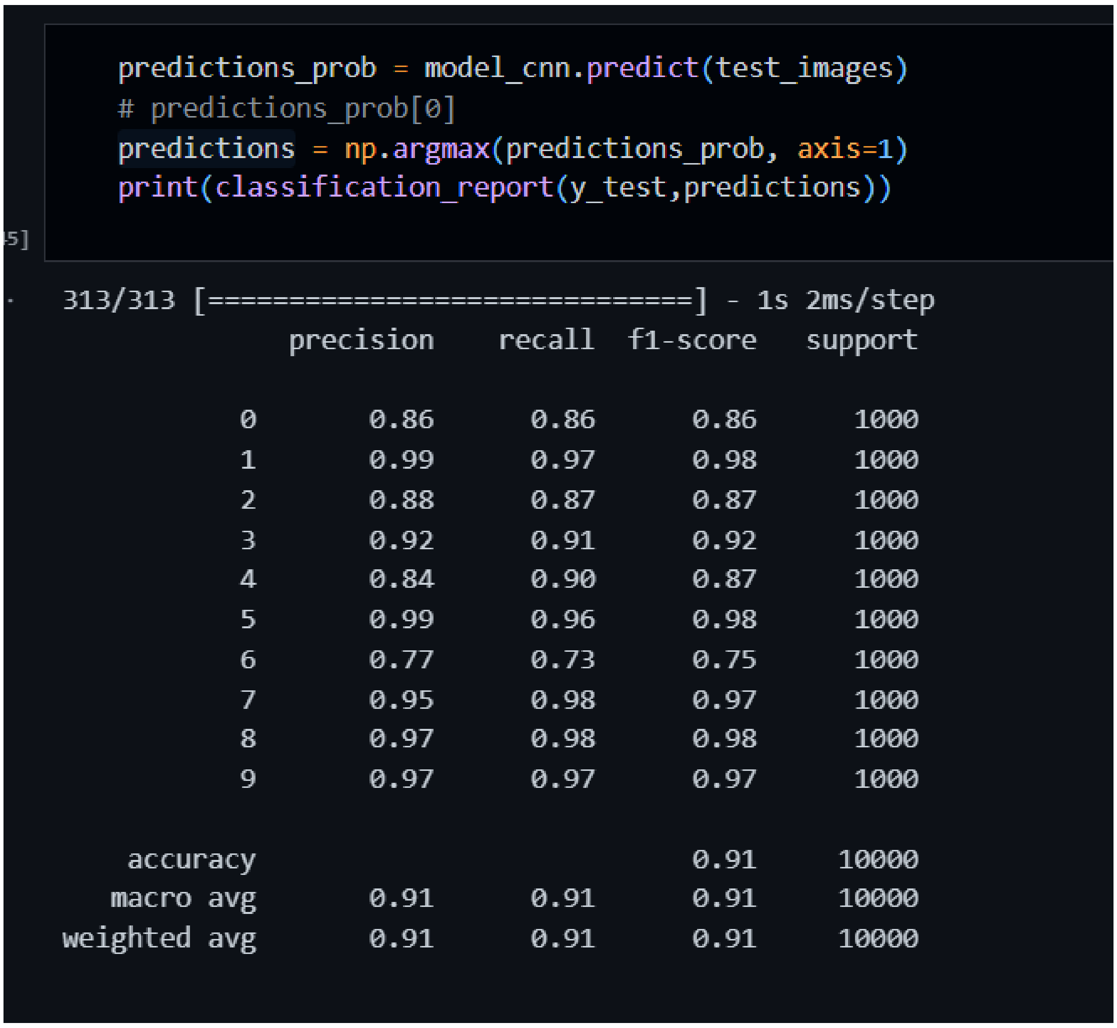
```
Train Loss: 0.16203564405441284
Train Accuracy: 0.9425833225250244
Test Loss: 0.23937036097049713
Test Accuracy: 0.9138000011444092
```

From this model evaluation training accuracy vs validation accuracy and training loss vs validation loss can also be depicted.





Here we are showing the classification report consisting of precision recall and f-1 score for every class.

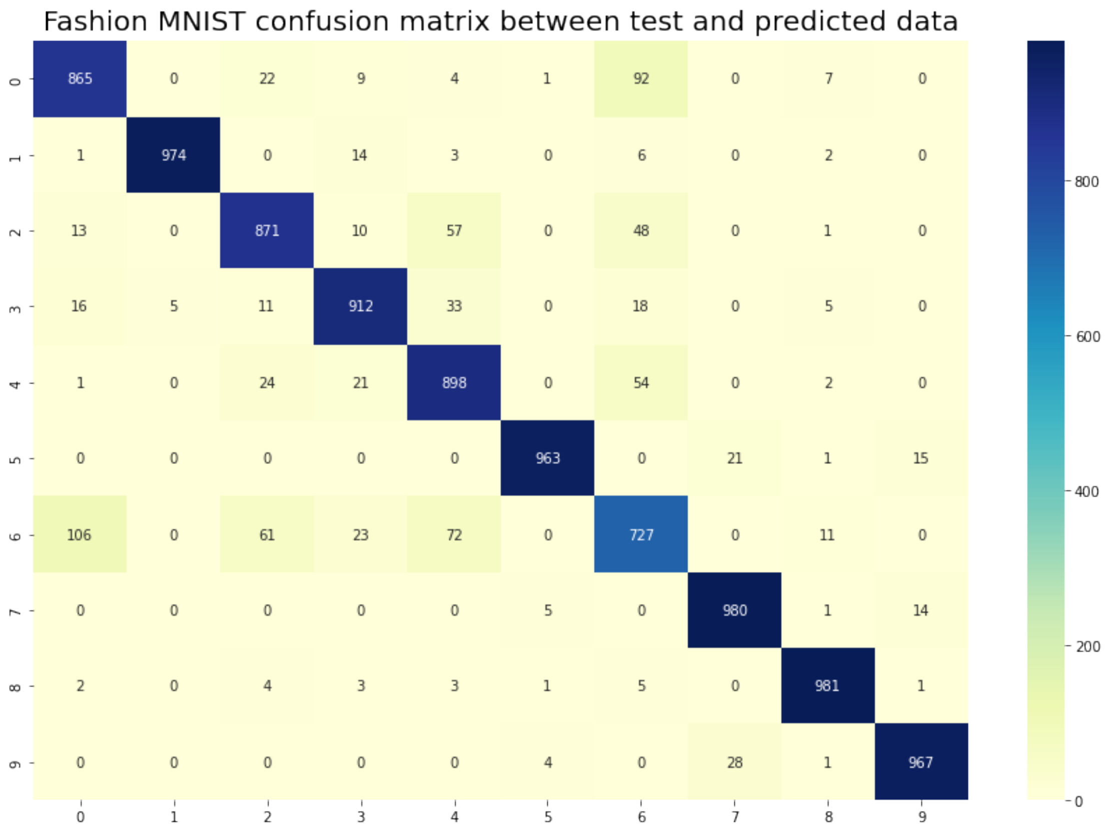


```
predictions_prob = model_cnn.predict(test_images)
# predictions_prob[0]
predictions = np.argmax(predictions_prob, axis=1)
print(classification_report(y_test,predictions))
```

```
5]
```

```
313/313 [==============================] - 1s 2ms/step
              precision    recall  f1-score   support

           0       0.86      0.86      0.86      1000
           1       0.99      0.97      0.98      1000
           2       0.88      0.87      0.87      1000
           3       0.92      0.91      0.92      1000
           4       0.84      0.90      0.87      1000
           5       0.99      0.96      0.98      1000
           6       0.77      0.73      0.75      1000
           7       0.95      0.98      0.97      1000
           8       0.97      0.98      0.98      1000
           9       0.97      0.97      0.97      1000

    accuracy                           0.91     10000
   macro avg       0.91      0.91      0.91     10000
weighted avg       0.91      0.91      0.91     10000
```

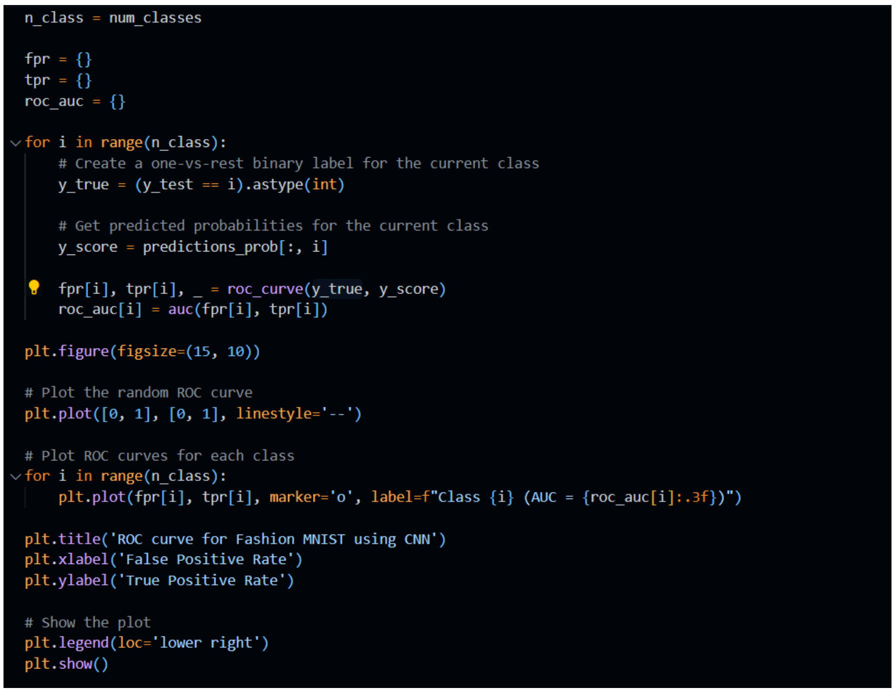
Confusion Matrix for True data that comes from test data and our models predicted data



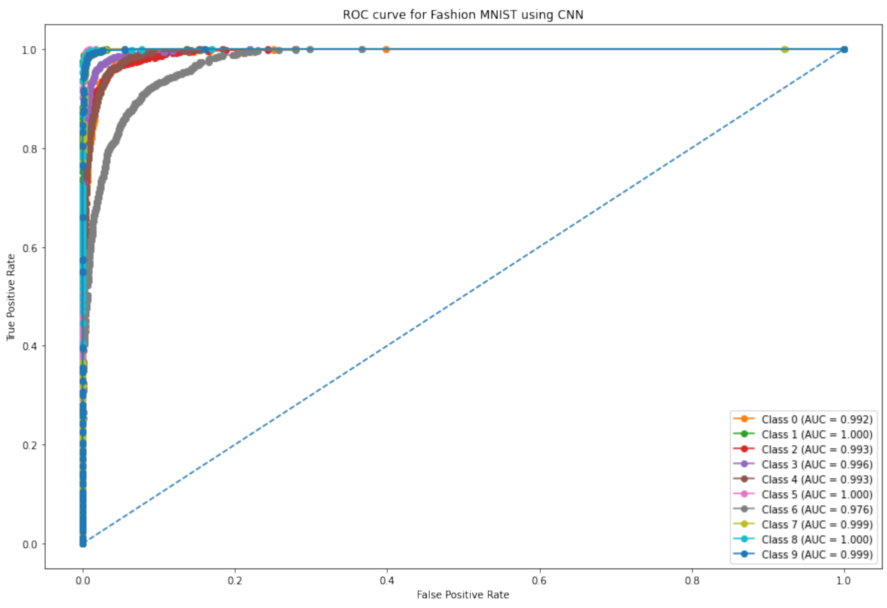
Fashion MNIST confusion matrix between test and predicted data

	0	1	2	3	4	5	6	7	8	9
0	865	0	22	9	4	1	92	0	7	0
1	1	974	0	14	3	0	6	0	2	0
2	13	0	871	10	57	0	48	0	1	0
3	16	5	11	912	33	0	18	0	5	0
4	1	0	24	21	898	0	54	0	2	0
5	0	0	0	0	0	963	0	21	1	15
6	106	0	61	23	72	0	727	0	11	0
7	0	0	0	0	0	5	0	980	1	14
8	2	0	4	3	3	1	5	0	981	1
9	0	0	0	0	0	4	0	28	1	967

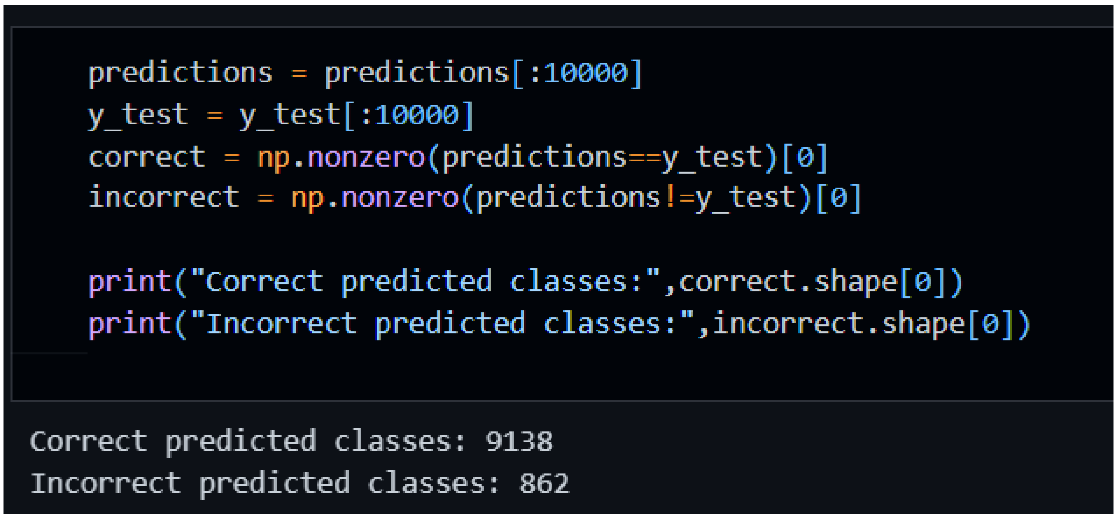
We will now provide the ROC curve for this model. To determine whether a ROC is good or bad, we must first determine whether the actual positive rate, or sensitivity, will rise, and if the area under the curve (AUC) is close to one, the model is doing well. As can be seen from the code and plot below, the model performs well for classes 0 to 9.



```python
n_class = num_classes

fpr = {}
tpr = {}
roc_auc = {}

for i in range(n_class):
    # Create a one-vs-rest binary label for the current class
    y_true = (y_test == i).astype(int)

    # Get predicted probabilities for the current class
    y_score = predictions_prob[:, i]

    fpr[i], tpr[i], _ = roc_curve(y_true, y_score)
    roc_auc[i] = auc(fpr[i], tpr[i])

plt.figure(figsize=(15, 10))

# Plot the random ROC curve
plt.plot([0, 1], [0, 1], linestyle='--')

# Plot ROC curves for each class
for i in range(n_class):
    plt.plot(fpr[i], tpr[i], marker='o', label=f"Class {i} (AUC = {roc_auc[i]:.3f})")

plt.title('ROC curve for Fashion MNIST using CNN')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')

# Show the plot
plt.legend(loc='lower right')
plt.show()
```

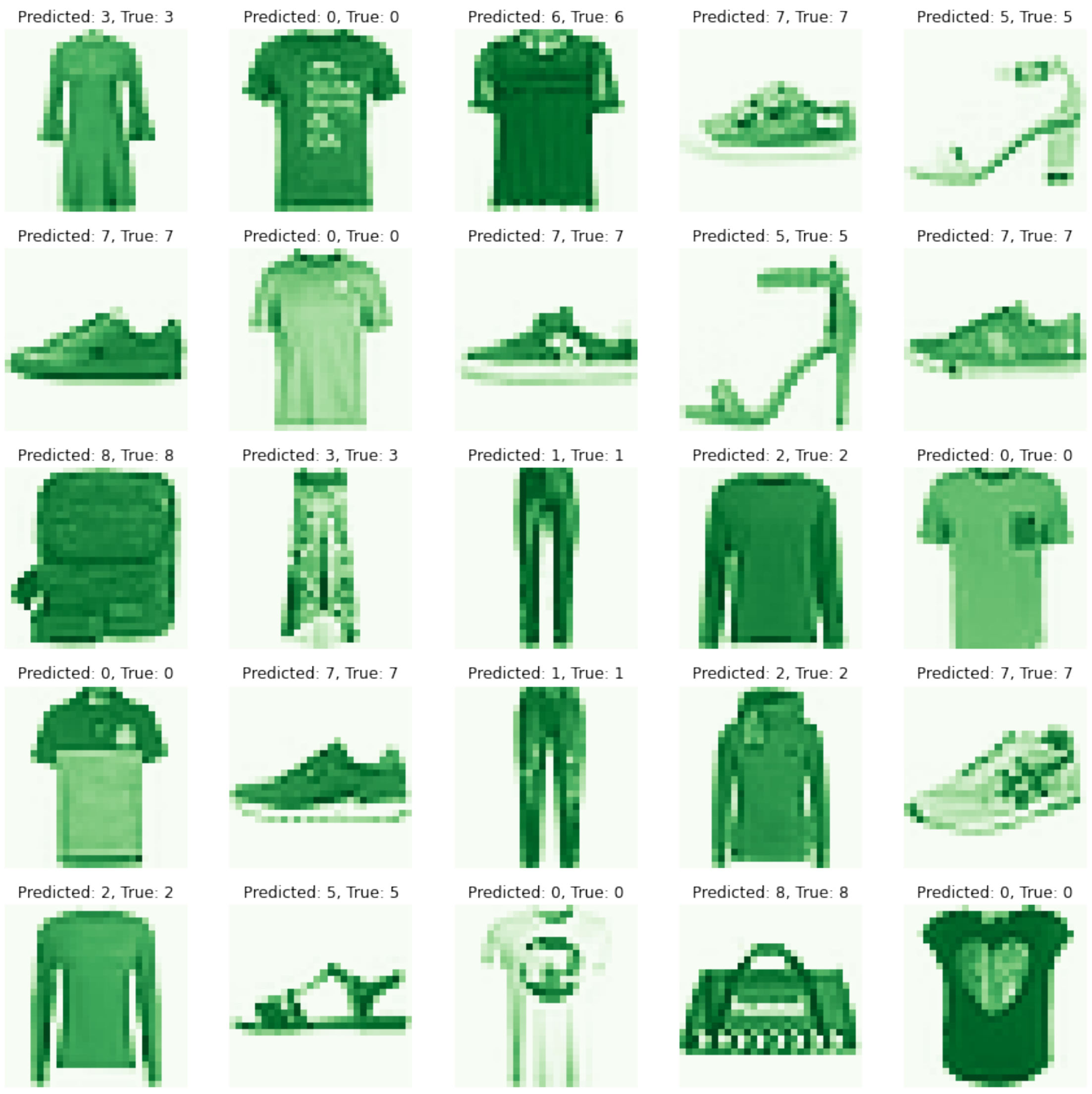


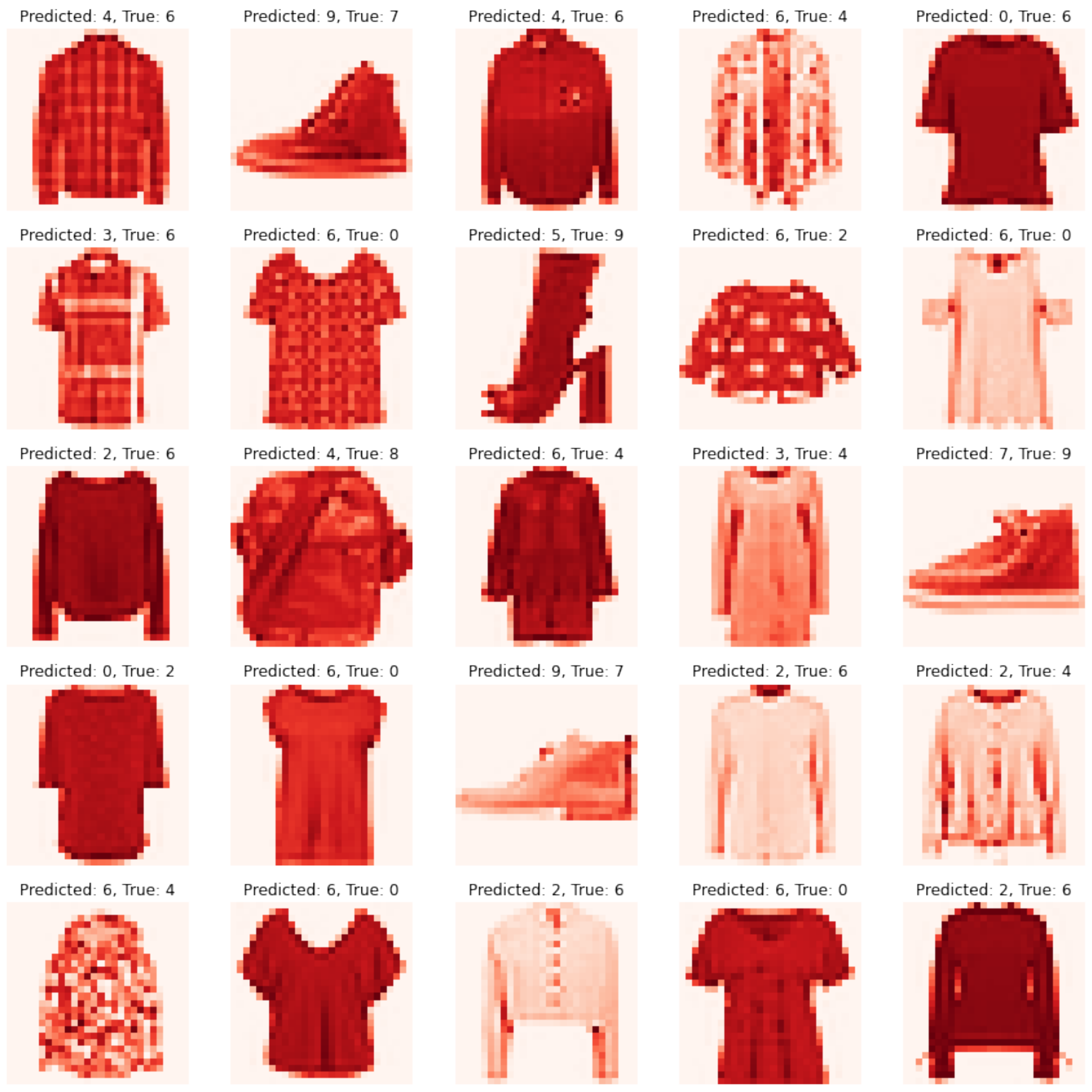
To calculate and visualise correctly predicted and miss-classified images, we performed the below code. From here, it can be seen that this CNN model predicts 9138 images correctly and 862 images incorrectly out of 10,000.



```
predictions = predictions[:10000]
y_test = y_test[:10000]
correct = np.nonzero(predictions==y_test)[0]
incorrect = np.nonzero(predictions!=y_test)[0]

print("Correct predicted classes:",correct.shape[0])
print("Incorrect predicted classes:",incorrect.shape[0])
```

```
Correct predicted classes: 9138
Incorrect predicted classes: 862
```




Predicted: 4, True: 6
Predicted: 9, True: 7
Predicted: 4, True: 6
Predicted: 6, True: 4
Predicted: 0, True: 6
Predicted: 3, True: 6
Predicted: 6, True: 0
Predicted: 5, True: 9
Predicted: 6, True: 2
Predicted: 6, True: 0
Predicted: 2, True: 6
Predicted: 4, True: 8
Predicted: 6, True: 4
Predicted: 3, True: 4
Predicted: 7, True: 9
Predicted: 0, True: 2
Predicted: 6, True: 0
Predicted: 9, True: 7
Predicted: 2, True: 6
Predicted: 2, True: 4
Predicted: 6, True: 4
Predicted: 6, True: 0
Predicted: 2, True: 6
Predicted: 6, True: 0
Predicted: 2, True: 6

Chapter 4

Discussion

The ANN achieved an accuracy of 0.90 on the train set and an accuracy of 0.87 on the test set, which means 87% predictions are correct. The precision for classes 0 to 9 are 0.82, 0.99, 0.75, 0.82, 0.81, 0.95, 0.70, 0.94, 0.97, and 0.96, respectively, which is the percentage of positive predictions that are correct for classes 0 to 9. For recall, class 0 to 9 are 0.83, 0.96, 0.84, 0.91, 0.74, 0.96, 0.62, 0.94, 0.95, and 0.96, respectively, which means positive cases are predicted to be positive for class 0 to 9 with this percentage, and F1-scores for each class are 0.82, 0.97, 0.79, 0.87, 0.78, 0.95, 0.66, 0.94, 0.96, and 0.96, respectively. On the other hand, the CNN obtained an accuracy of 0.94 on the train set and 0.91 on the test set, implying that 91% of predictions were correct. The precision for classes 0 to 9 is 0.86, 0.99, 0.88, 0.92, 0.84, 0.99, 0.77, 0.95, 0.97, and 0.97, respectively, which is the percentage of valid positive predictions. Class 0 to 9 recall is 0.86, 0.97, 0.87, 0.91, 0.90, 0.96, 0.73, 0.98, 0.98, and 0.97, respectively, which means positive cases are predicted to be positive with this percentage, and F1-scores are 0.86, 0.98, 0.87, 0.92, 0.87, 0.98, 0.75, 0.97, 0.98, and 0.97, respectively, and finally, the test loss in ANN is 38% whereas the test loss in CNN is 24%.

All these values make us believe that the CNN model works better than the ANN model for this particular dataset, which is Fashion Mnist.