INTEL UNNATI INDUSTRIAL TRAINING

FINAL REPORT

TOPIC - CONQUERING FASHION MNIST WITH CNNs USING COMPUTER VISION
TEAM NAME - Ashutosh Jha
MEMBER - Ashutosh Jha
REGISTRATION NUMBER - 210907370
COLLEGE - MANIPAL INSTITUTE OF TECHNOLOGY (MAHE)
SUBMISSION DATE – 15/07/2023

Email : ashutosh.jha2@learner.manipal.edu

CONQUERING FASHION MNIST WITH CNNs USING COMPUTER VISION

Fashion-MNIST is a dataset commonly used for image classification tasks in the field of computer vision. It consists of 60,000 training images and 10,000 testing images, each of which is a grayscale 28x28 pixel image belonging to one of ten different fashion categories.

And in this problem statement we have to make a CNN architecture which will help us in classifying the clothing items and then we have to make use of the optimization provided by intel Devcloud platform to optimize our model and then compare the BASELINE INFERENCE LATENCY and ACCURACY with and without the optimization.

Introduction

- Convolutional Neural Networks (CNNs) have demonstrated exceptional performance in image classification tasks due to their ability to capture spatial relationships within images. In this study, I employ CNNs to conquer the Fashion-MNIST dataset, leveraging the power of computer vision techniques to accurately classify fashion items.

- MOTIVATION : The Fashion-MNIST dataset is used as a standard for assessing how well image classification models perform. Classifying fashion items presents a difficult task that is applicable to real-world applications. I want to try to add to the body of knowledge in the field of computer vision by creating a CNN model for Fashion-MNIST.

My Approach

I basically followed the following steps in order to come up with the final convolution neural network model, I will talk in detail about each step ahead in the report.

1. Data Preprocessing : Before we feed the data to our CNN architecture we need to process it as per the requirement of our model .
2. Making the model architecture from scratch : We need to make our Model layer by layer from scratch in order to make it capable of learning and predicting .
3. Model training : After our model is ready we need to train it in order to make it efficient enough to successfully predict with high accuracy.
4. Hyperparameter Tuning : This step is done to find a balance between all the parameters , which ensures highest accuracy and best working of the model without overfitting.
5. Model Evaluation : After our model is trained we need to check its metrics like accuracy and loss to find out how good our model is . Sometimes we also look at precision , recall and confusion matrix.
6. Model optimisation : This step is done to make the model's inference more accurate and faster.
7. Model Deployment : After a model is ready , we need to deploy it to other platforms to use it .

Data Preprocessing

- Loading the dataset : First of all we need to load the required dataset to our model . This can be done in 2 ways , directly from the keras or by uploading it manually to the notebook and then loading it using it’s directory path.

  sample code : train = pd.read_csv('fashion-mnist_train.csv’)
  test = pd.read_csv('fashion-mnist_test.csv')

- Determining of Shape : After we are ready with the test and train data we need to make sure they are of correct shape for our architecture.

  Sample code : X_train.shape, y_train.shape
  X_test.shape, y_test.shape

- Determining of dimensions : In some cases we need to change the dimension of our data as per the requirement of our CNN architecture.

  Sample code : X_train = np.expand_dims(X_train, -1)
  X_test = np.expand_dims(X_test, -1)

- Scaling : we need to do this step so that we get a normalized value, doing this sometimes also affects the accuracy by a lot. This step basically incudes dividing the data by 255.

  Sample code : X_train = X_train/255
  X_test = X_test/255

- Making classes : we need to assign numbers to all different types of classes .

Making the model architecture from scratch

To initiate making the architecture we use the code : model = Sequential()

After this we need to add layer by layer to make a whole CNN architecture . We will now discuss different types of Layers available and how to use them .

Convolution layer : convolutional layers are designed to leverage the inherent spatial structure of data, capture local patterns, and learn hierarchical representations, making them well-suited for tasks involving images, videos, and other grid-like data.

Sample code :model.add(Conv2D(filters=32, kernel_size=(3, 3), activation='relu', strides=1, padding='same',data_format='channels_last', input_shape=(28,28,1)))

Here we can also change the kernel size , activation function , unit of strides, padding etc for our layer.

Batch normalization layer : Batch normalization is a technique commonly used in CNN architectures to improve the training and performance of the model. It addresses the internal covariate shift problem and provides several benefits

Sample code : model.add(BatchNormalization())

Making the model architecture from scratch

Dropout layer : The dropout layer is a regularization technique commonly used in CNN model architectures to prevent overfitting and improve generalization. It introduces random noise and imposes a form of regularization during training by temporarily "dropping out" a random subset of units/neurons from the network

Sample code : model.add(Dropout(0.25))

Max Pooling layer : The max pooling layer is a fundamental component of CNN model architectures that provides several benefits for feature extraction and spatial downsampling

Sample code : model.add(MaxPooling2D(pool_size=(2, 2)))

Flatten layer : The flatten layer is used in CNN model architectures to reshape the multidimensional feature maps produced by the convolutional and pooling layers into a one-dimensional vector

Sample code : model.add(Flatten())

Dense layer : The dense layer, also known as a fully connected layer, is used in CNN model architectures to perform high-level reasoning and decision-making based on the features extracted by the convolutional and pooling layers.

Sample code : model.add(Dense(10, activation='softmax'))

Model training

- Before training the model we need to compile the model , this includes specifying the optimizer , which loss calculating algorithm is used and the metrics.

  Sample Code : cnn_model3.compile(optimizer='adam', loss= 'sparse_categorical_crossentropy', metrics=['accuracy’])

- Training the model is basically teaching it how to predict and we can change some parameters such as epochs , batch size and verbose to make sure its trained in a better way.

  Sample Code : cnn_model3.fit(X_train, y_train, epochs=40, batch_size=64, verbose=1, validation_data=(X_validation, y_validation))

Hyperparameter Tuning

- Hyperparameter Tuning is basically experimenting with different hyperparameters, such as learning rate, batch size, number of filters, and number of layers, to find the optimal configuration. Utilize techniques like learning rate schedules or early stopping to improve model performance.

Sample code :
```
optimizer = Adam(lr=0.001, beta_1=0.9, beta_2=0.999 )
reduce_lr = LearningRateScheduler(lambda x: 1e-3 * 0.9 ** x)
```

Model Evaluation

- Model Evaluation: Evaluating your trained model on the testing set to assess its performance and Calculating metrics like accuracy, precision, recall, and F1 score to measure the model's classification performance.

Sample code 1:
```
score = model.evaluate(x_test, y_test)
print('Loss: {:.4f}'.format(score[0]))
print('Un-Optimized Accuracy: {:.4f}'.format(score[1]))
```

Sample code 2:
```
print(classification_report(Y_true, Y_pred_classes, target_names = classes))
```

Model optimisation and Deployment

- Model Optimisation: Here I have done optimization of our model with the help of Intel Devcloud integrated oneDNN . It helps us increase accuracy and reduce the inference time.

  Sample code : TF_ENABLE_ONEDNN_OPTS= 1

- Model Deployment: Here I have completed the conversion of .h5 format model to IR format so that it can be used for openvino and then deployed with the help of openvino

  Sample code : !python3 openvino/model-optimizer/mo_tf.py --
  fashion_mnist_cnn_model.h5 --input_shape=\[1,28,28\]

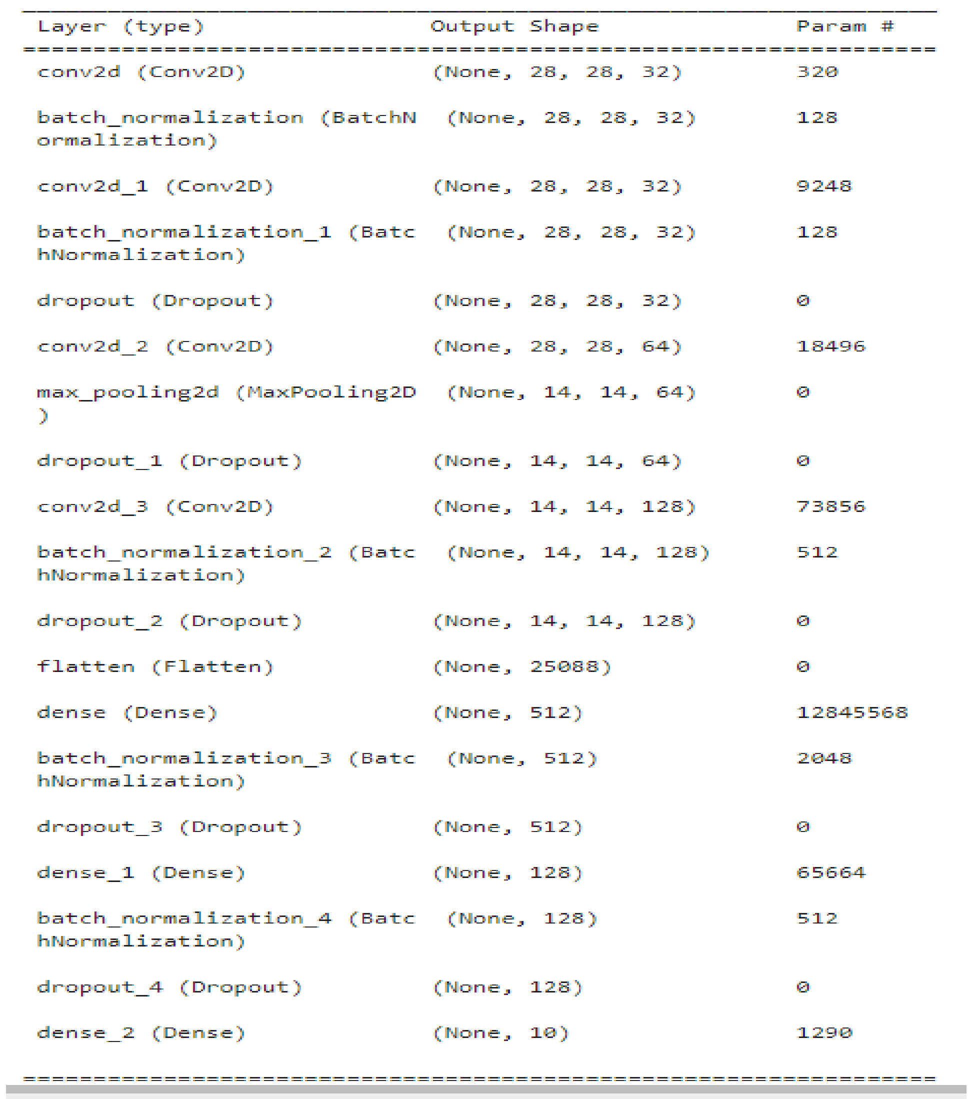
CNN Models made by me

During this training period I was able to make 3 different CNN architectures from scratch, I have used different type of dataset pre processing for all of them along with different numbers of different types of layers in the model architecture to ensure all 3 of them are unique.

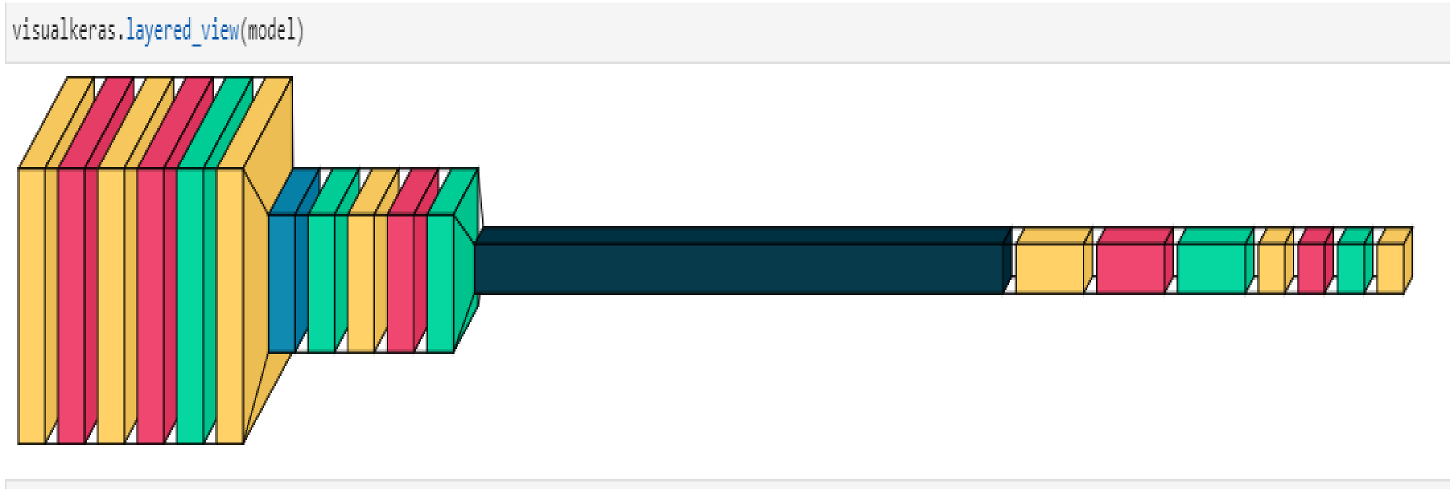
Name of my models are as follows

- Model 1 : MAIN_Fashionmnistcnn_AshutoshJha
- Model 2 : 1stBackup_Ashutoshjha
- Model 3 : 2ndBackup_cnnAshutosh

Model 1 : MAIN_Fashionmnistcnn_AshutoshJha



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

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

 conv2d_1 (Conv2D)           (None, 28, 28, 32)        9248

 batch_normalization_1 (Batc  (None, 28, 28, 32)       128
 hNormalization)

 dropout (Dropout)           (None, 28, 28, 32)        0

 conv2d_2 (Conv2D)           (None, 28, 28, 64)        18496

 max_pooling2d (MaxPooling2D  (None, 14, 14, 64)       0
 )

 dropout_1 (Dropout)         (None, 14, 14, 64)        0

 conv2d_3 (Conv2D)           (None, 14, 14, 128)       73856

 batch_normalization_2 (Batc  (None, 14, 14, 128)      512
 hNormalization)

 dropout_2 (Dropout)         (None, 14, 14, 128)       0

 flatten (Flatten)           (None, 25088)             0

 dense (Dense)               (None, 512)               12845568

 batch_normalization_3 (Batc  (None, 512)              2048
 hNormalization)

 dropout_3 (Dropout)         (None, 512)               0

 dense_1 (Dense)             (None, 128)               65664

 batch_normalization_4 (Batc  (None, 128)              512
 hNormalization)

 dropout_4 (Dropout)         (None, 128)               0

 dense_2 (Dense)             (None, 10)                1290

=================================================================
```

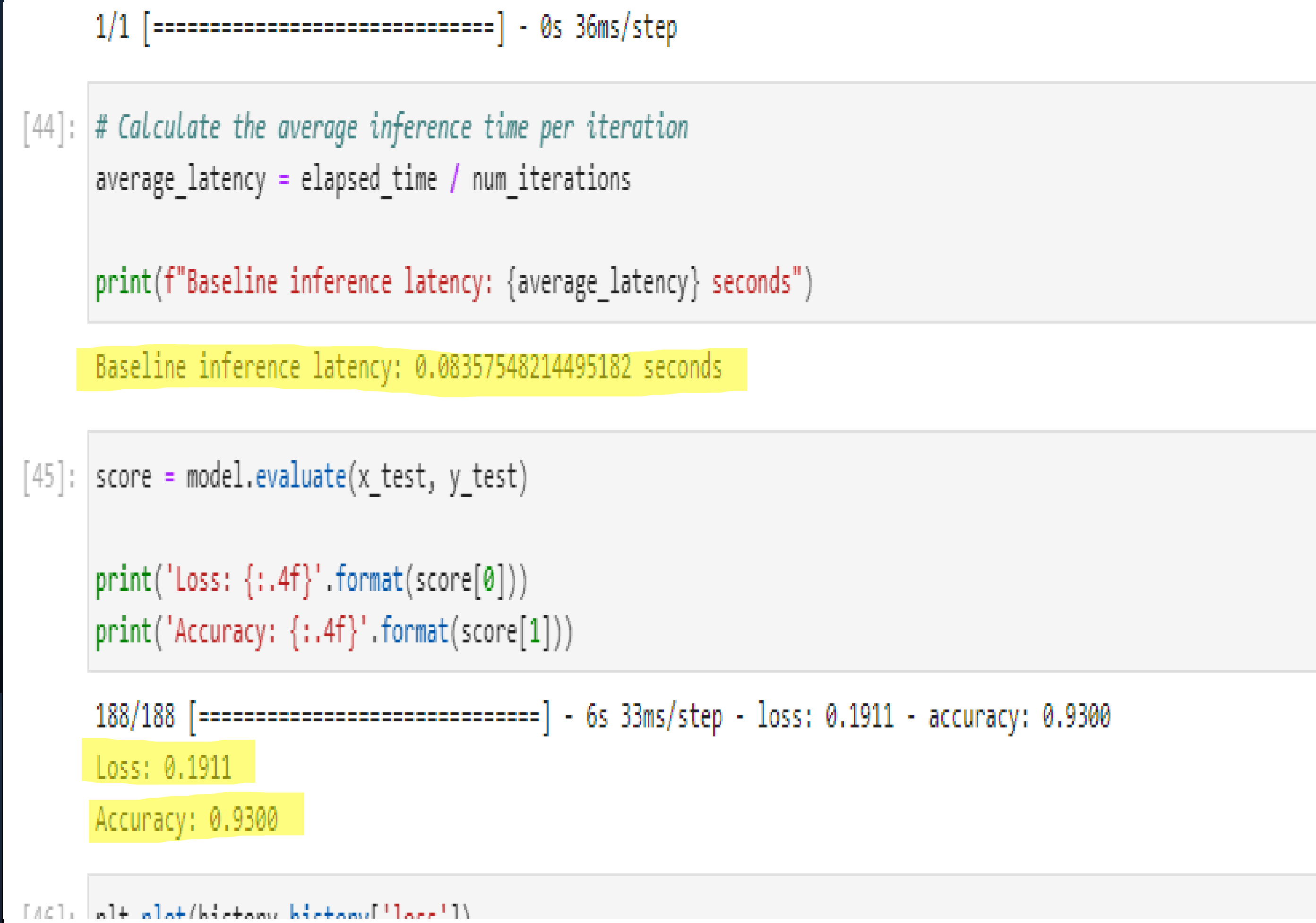


```
visualkeras.layered_view(model)
```

This is the Architecture of my first CNN model, Inference related screenshots for this structure are on the next page

Model 1 : MAIN_Fashionmnistcnn_AshutoshJha

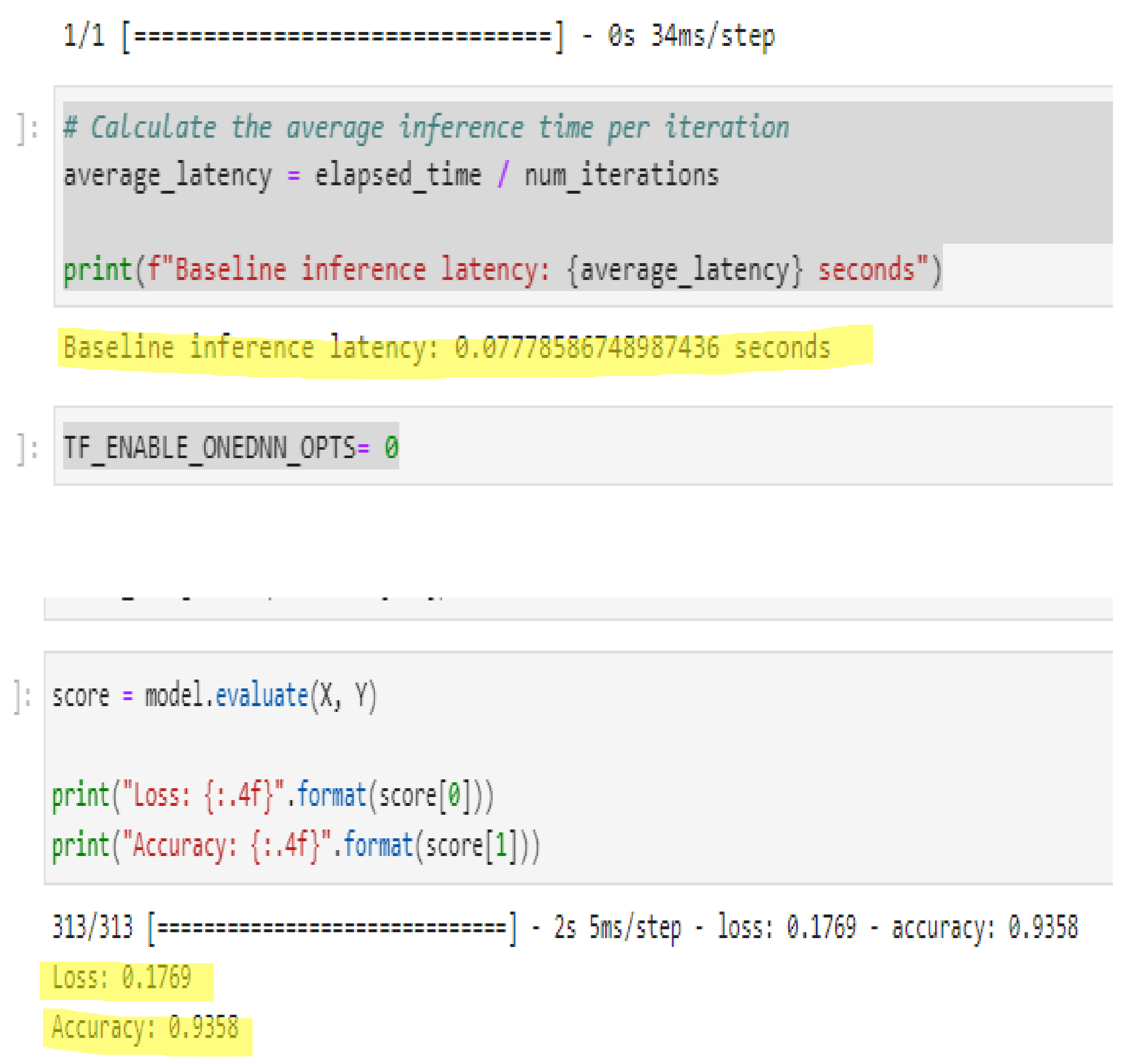
UN-OPTIMIZED ACCURACY, LOSS AND BASELINE INFERENCE LATENCY



```
1/1 [==============================] - 0s 36ms/step

[44]: # Calculate the average inference time per iteration
      average_latency = elapsed_time / num_iterations

      print(f"Baseline inference latency: {average_latency} seconds")

Baseline inference latency: 0.08357548214495182 seconds

[45]: score = model.evaluate(x_test, y_test)

      print('Loss: {:.4f}'.format(score[0]))
      print('Accuracy: {:.4f}'.format(score[1]))

188/188 [==============================] - 6s 33ms/step - loss: 0.1911 - accuracy: 0.9300
Loss: 0.1911
Accuracy: 0.9300

[46]: plt.plot(history.history['loss'])
```

Model 1 : MAIN_Fashionmnistcnn_AshutoshJha

OPTIMIZED ACCURACY, LOSS AND BASELINE INFERENCE LATENCY



```
1/1 [==============================] - 0s 34ms/step
```

```
]: # Calculate the average inference time per iteration
   average_latency = elapsed_time / num_iterations

   print(f"Baseline inference latency: {average_latency} seconds")
```

```
Baseline inference latency: 0.07778586748987436 seconds
```

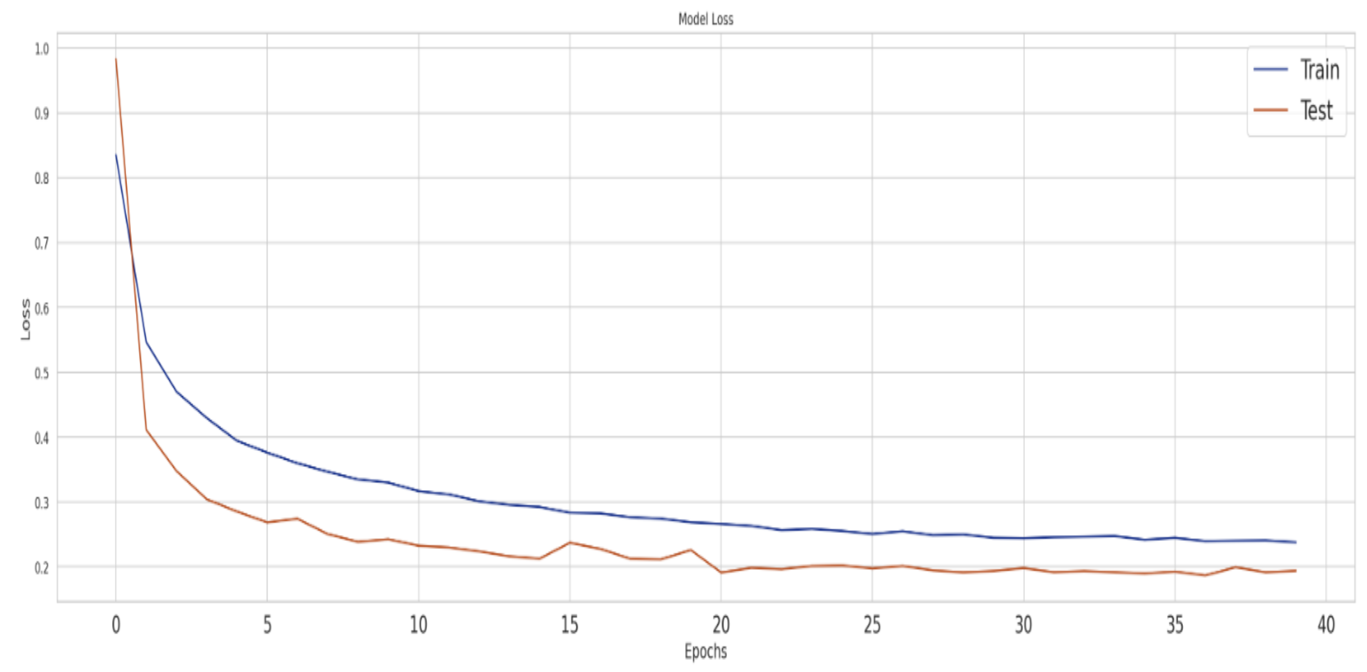
```
]: TF_ENABLE_ONEDNN_OPTS= 0
```

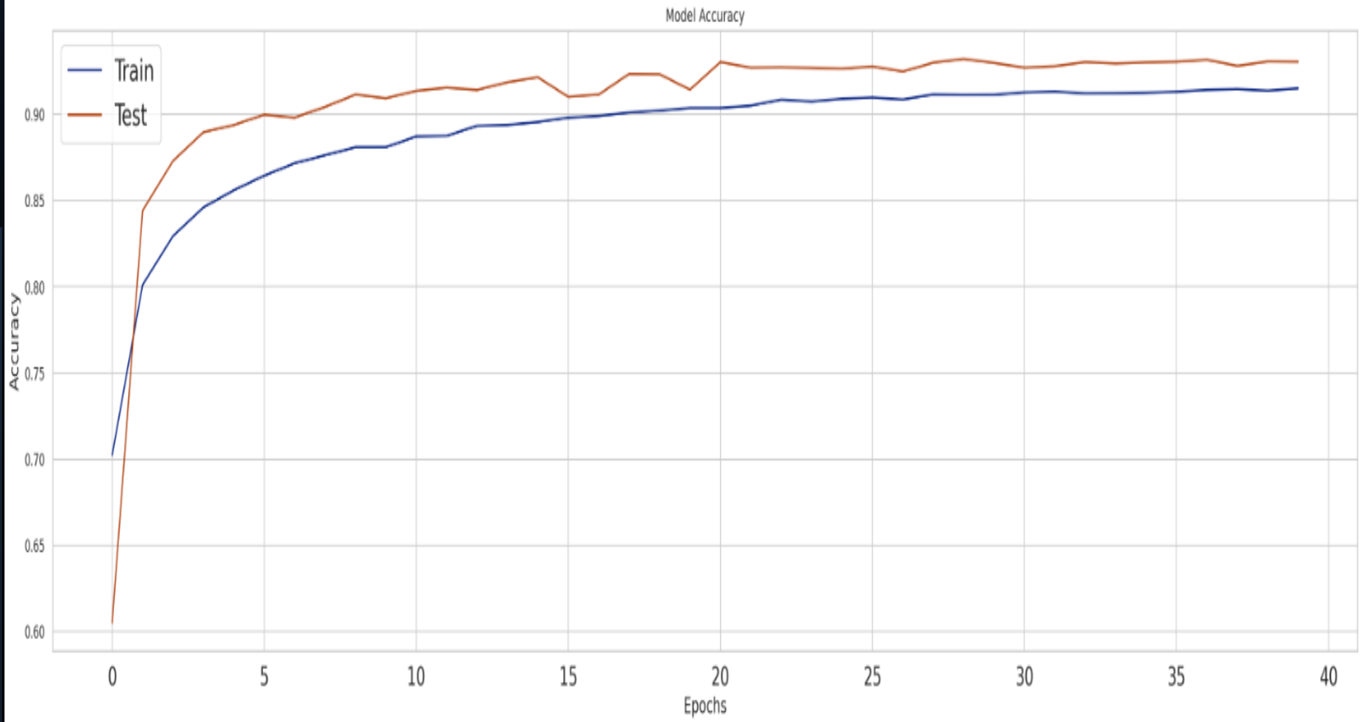
```
]: score = model.evaluate(X, Y)

   print("Loss: {:.4f}".format(score[0]))
   print("Accuracy: {:.4f}".format(score[1]))
```

```
313/313 [==============================] - 2s 5ms/step - loss: 0.1769 - accuracy: 0.9358
Loss: 0.1769
Accuracy: 0.9358
```

Model 1 : MAIN_Fashionmnistcnn_AshutoshJha

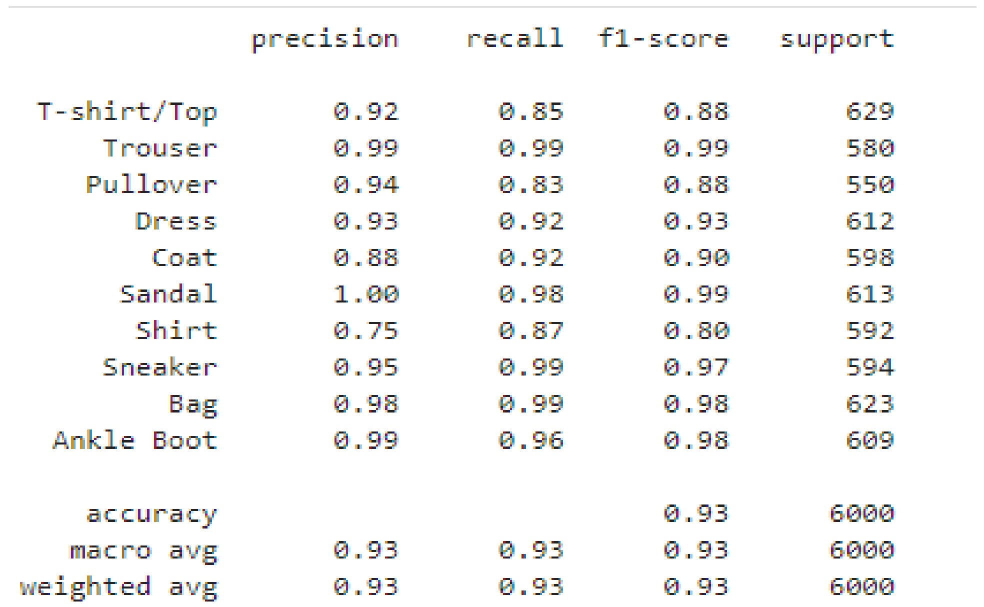
LOSS AND ACCURACY GRAPH



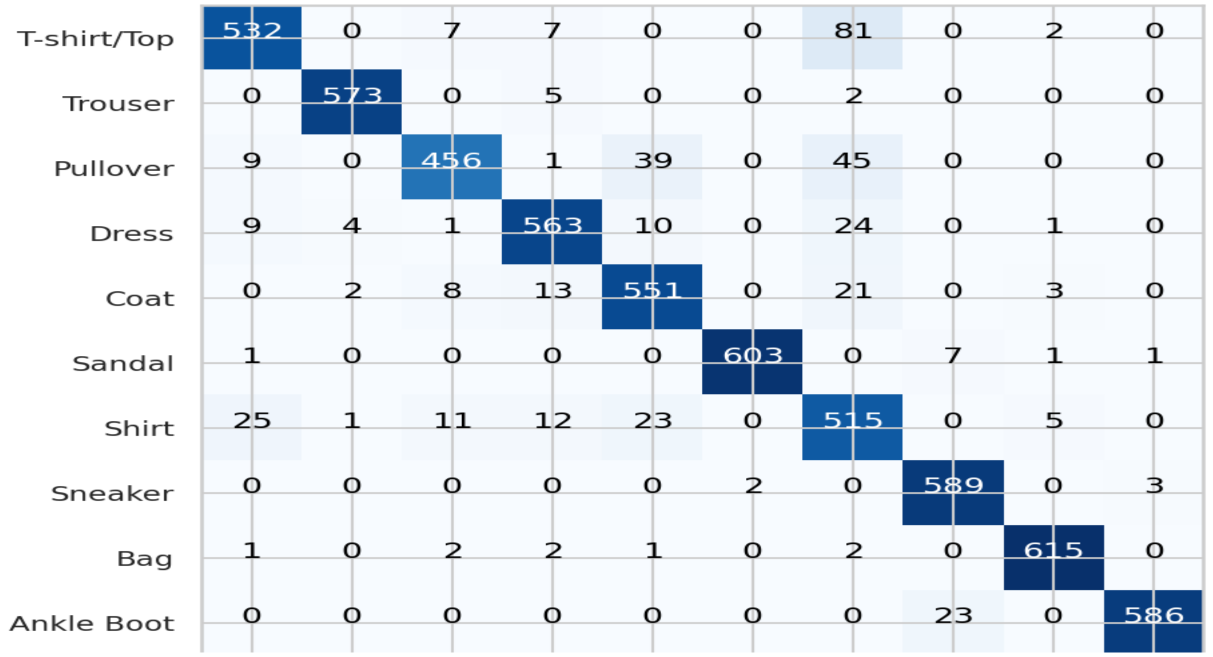


Model 1 : MAIN_Fashionmnistcnn_AshutoshJha

CLASSIFICATION REPORT & CONFUSION MATRIX

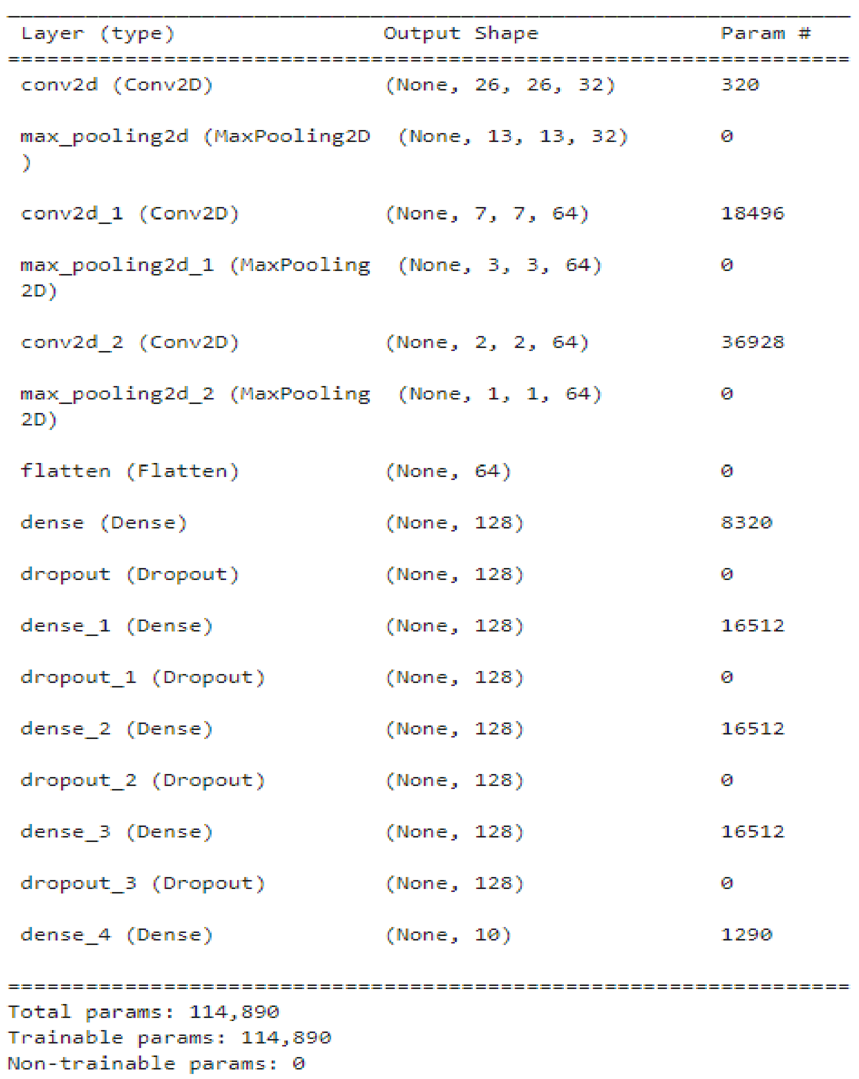


```
              precision    recall  f1-score   support

 T-shirt/Top       0.92      0.85      0.88       629
     Trouser       0.99      0.99      0.99       580
    Pullover       0.94      0.83      0.88       550
       Dress       0.93      0.92      0.93       612
        Coat       0.88      0.92      0.90       598
      Sandal       1.00      0.98      0.99       613
       Shirt       0.75      0.87      0.80       592
     Sneaker       0.95      0.99      0.97       594
         Bag       0.98      0.99      0.98       623
  Ankle Boot       0.99      0.96      0.98       609

    accuracy                           0.93      6000
   macro avg       0.93      0.93      0.93      6000
weighted avg       0.93      0.93      0.93      6000
```



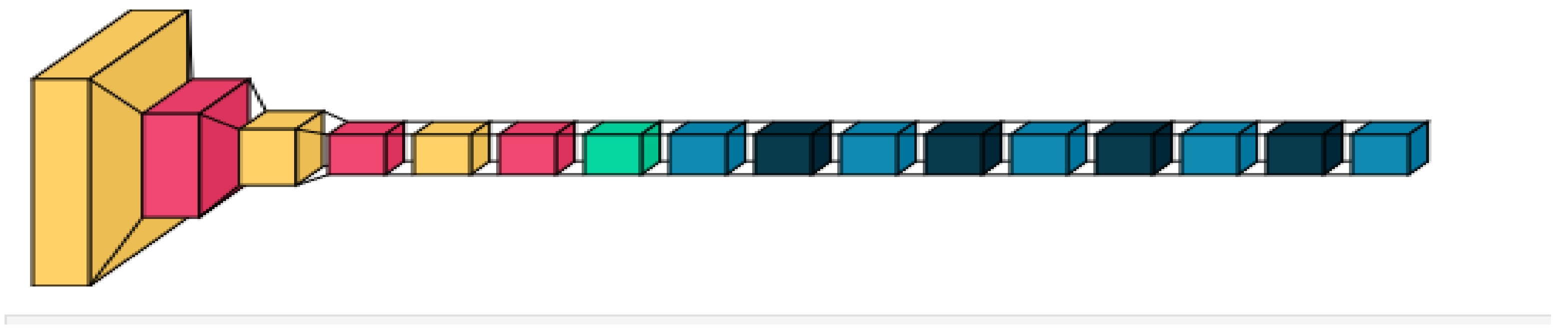
T-shirt/Top	532	0	7	7	0	0	81	0	2	0
Trouser	0	573	0	5	0	0	2	0	0	0
Pullover	9	0	456	1	39	0	45	0	0	0
Dress	9	4	1	563	10	0	24	0	1	0
Coat	0	2	8	13	551	0	21	0	3	0
Sandal	1	0	0	0	0	603	0	7	1	1
Shirt	25	1	11	12	23	0	515	0	5	0
Sneaker	0	0	0	0	0	2	0	589	0	3
Bag	1	0	2	2	1	0	2	0	615	0
Ankle Boot	0	0	0	0	0	0	0	23	0	586

Model 2 : 1stBackup_Ashutoshjha



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 7, 7, 64)          18496

 max_pooling2d_1 (MaxPooling  (None, 3, 3, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 2, 2, 64)          36928

 max_pooling2d_2 (MaxPooling  (None, 1, 1, 64)         0
 2D)

 flatten (Flatten)           (None, 64)                0

 dense (Dense)               (None, 128)               8320

 dropout (Dropout)           (None, 128)               0

 dense_1 (Dense)             (None, 128)               16512

 dropout_1 (Dropout)         (None, 128)               0

 dense_2 (Dense)             (None, 128)               16512

 dropout_2 (Dropout)         (None, 128)               0

 dense_3 (Dense)             (None, 128)               16512

 dropout_3 (Dropout)         (None, 128)               0

 dense_4 (Dense)             (None, 10)                1290

=================================================================
Total params: 114,890
Trainable params: 114,890
Non-trainable params: 0
```

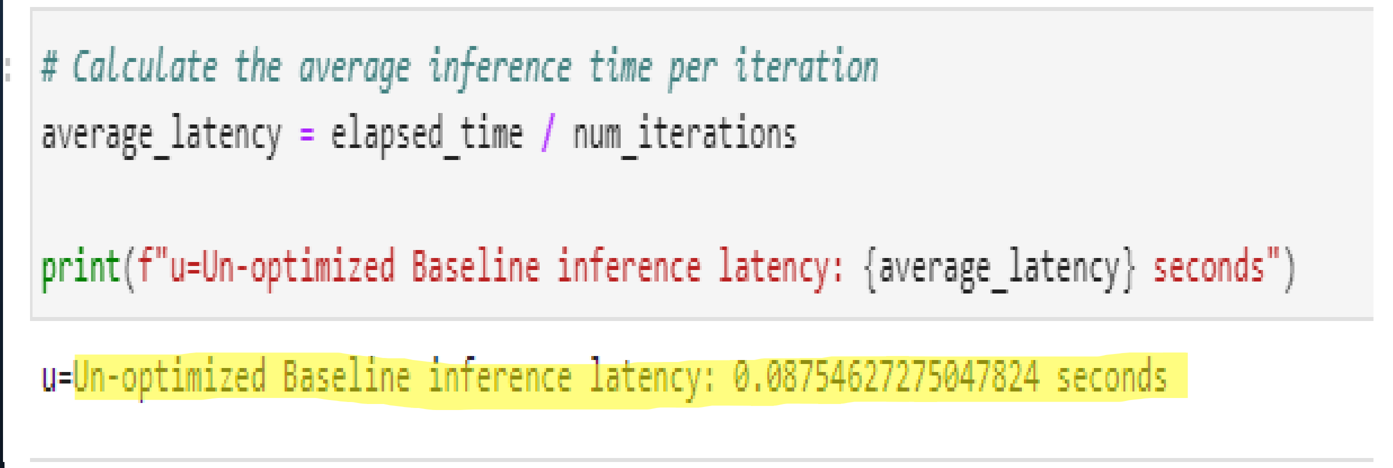
```
visualkeras.layered_view(cnn_model3)
```



This is the Architecture of my Second CNN model, Inference related screenshots for this structure are on the next page

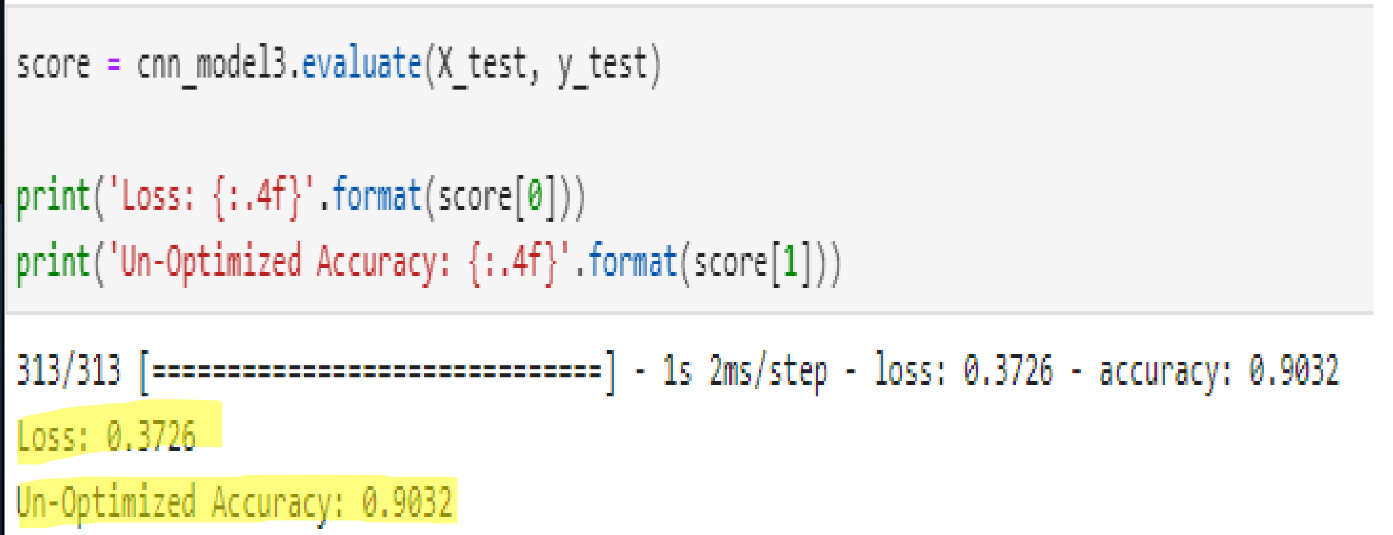
Model 2 : 1stBackup_Ashutoshjha

UN-OPTIMIZED ACCURACY, LOSS AND BASELINE INFERENCE LATENCY



```
# Calculate the average inference time per iteration
average_latency = elapsed_time / num_iterations

print(f"u=Un-optimized Baseline inference latency: {average_latency} seconds")
```

```
u=Un-optimized Baseline inference latency: 0.08754627275047824 seconds
```

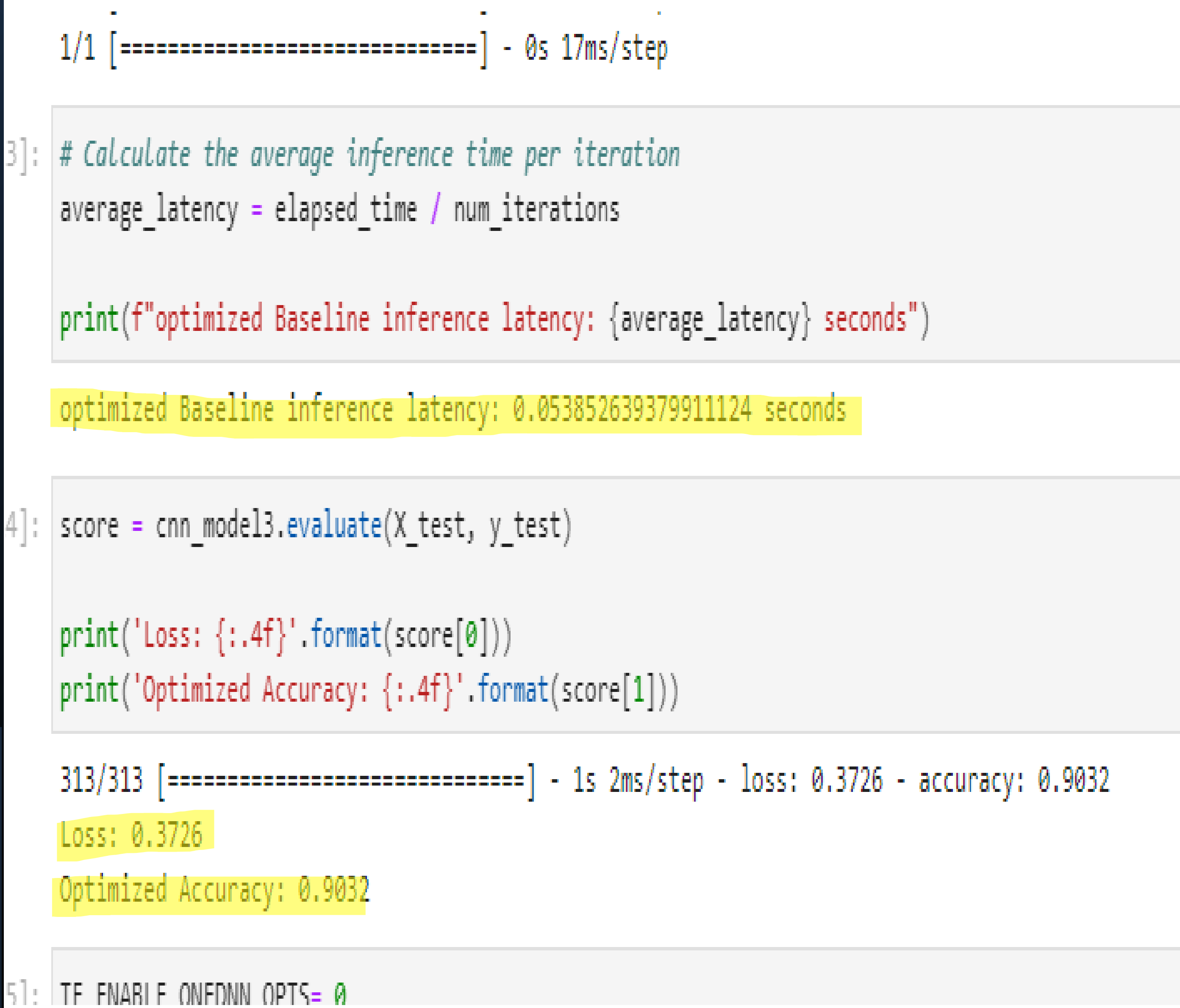


```
score = cnn_model3.evaluate(X_test, y_test)

print('Loss: {:.4f}'.format(score[0]))
print('Un-Optimized Accuracy: {:.4f}'.format(score[1]))
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.3726 - accuracy: 0.9032
Loss: 0.3726
Un-Optimized Accuracy: 0.9032
```

Model 2 : 1stBackup_Ashutoshjha

OPTIMIZED ACCURACY, LOSS AND BASELINE INFERENCE LATENCY



```
1/1 [==============================] - 0s 17ms/step
```

```
3]: # Calculate the average inference time per iteration
average_latency = elapsed_time / num_iterations

print(f"optimized Baseline inference latency: {average_latency} seconds")
```

```
optimized Baseline inference latency: 0.05385263937991124 seconds
```

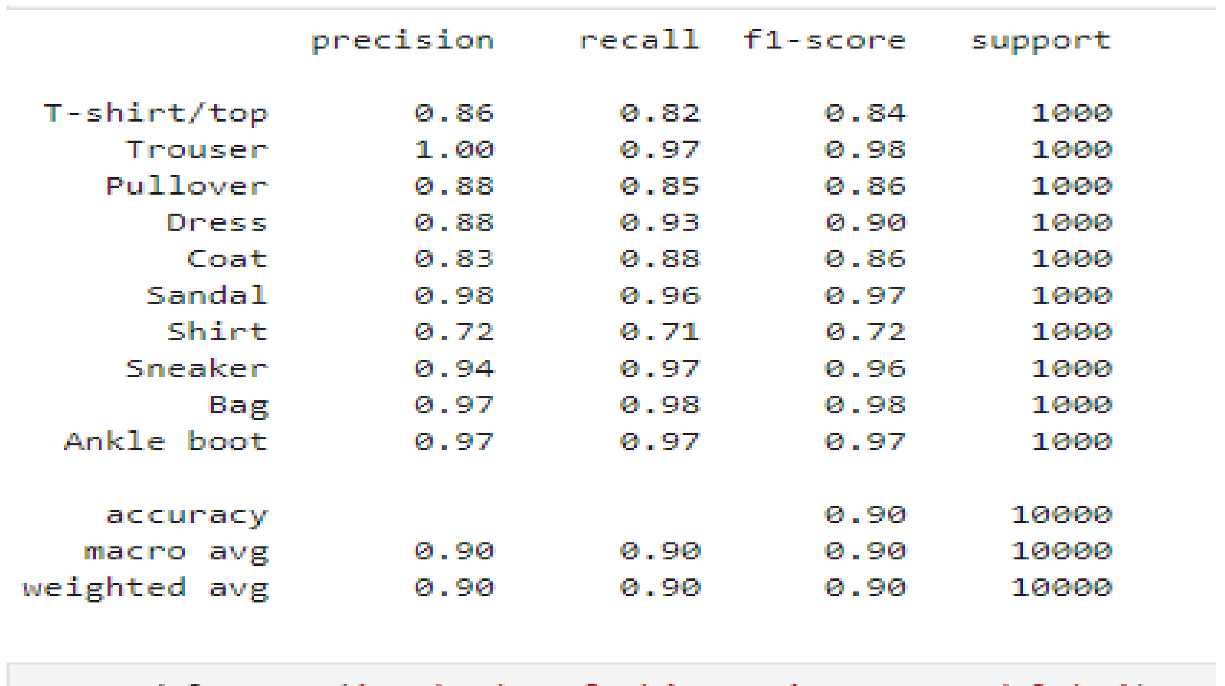
```
4]: score = cnn_model3.evaluate(X_test, y_test)

print('Loss: {:.4f}'.format(score[0]))
print('Optimized Accuracy: {:.4f}'.format(score[1]))
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.3726 - accuracy: 0.9032
Loss: 0.3726
Optimized Accuracy: 0.9032
```

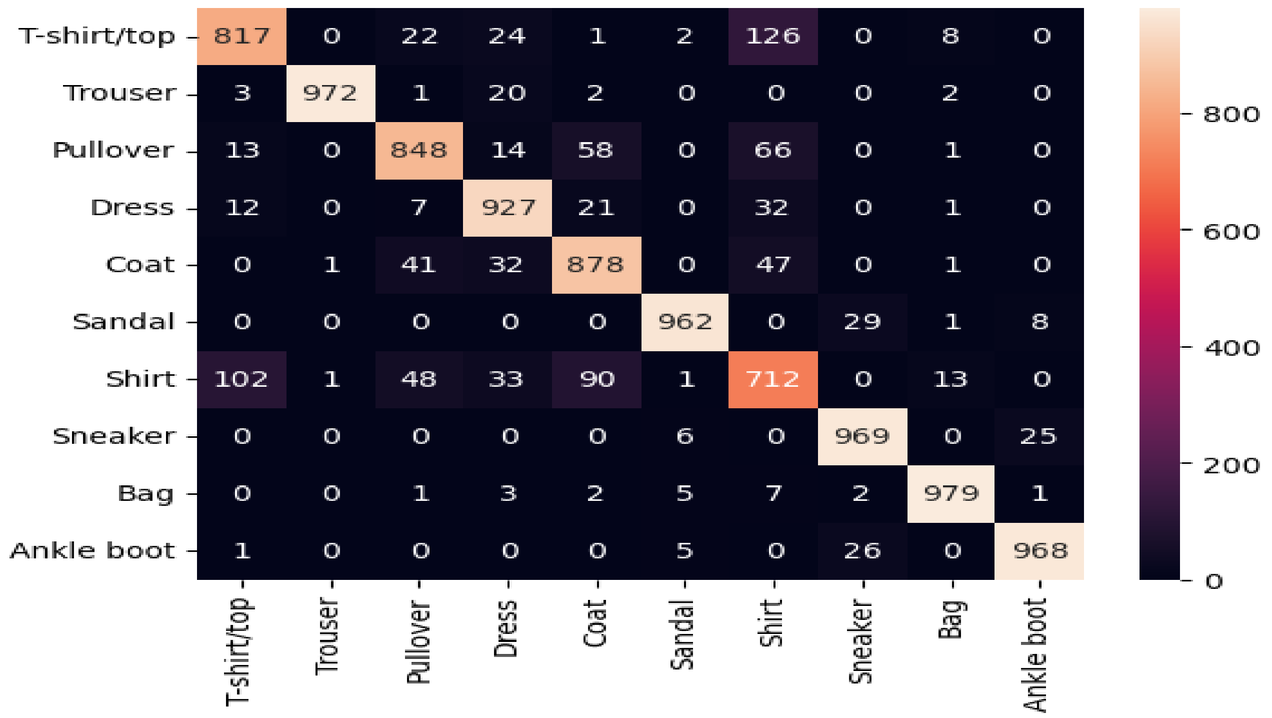
```
5]: TF_ENABLE_ONEDNN_OPTS= 0
```

Model 2 : 1stBackup_Ashutoshjha

CLASSIFICATION REPORT & CONFUSION MATRIX

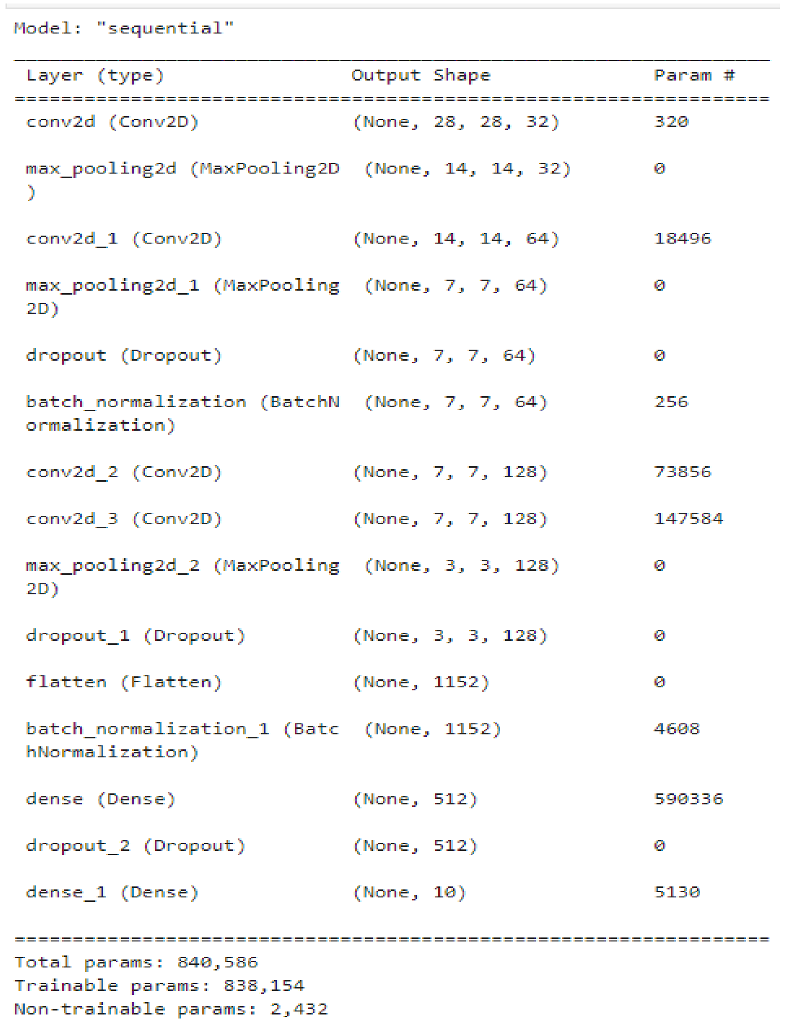


```
              precision    recall  f1-score   support

 T-shirt/top       0.86      0.82      0.84      1000
     Trouser       1.00      0.97      0.98      1000
    Pullover       0.88      0.85      0.86      1000
       Dress       0.88      0.93      0.90      1000
        Coat       0.83      0.88      0.86      1000
      Sandal       0.98      0.96      0.97      1000
       Shirt       0.72      0.71      0.72      1000
     Sneaker       0.94      0.97      0.96      1000
         Bag       0.97      0.98      0.98      1000
  Ankle boot       0.97      0.97      0.97      1000

    accuracy                           0.90     10000
   macro avg       0.90      0.90      0.90     10000
weighted avg       0.90      0.90      0.90     10000
```



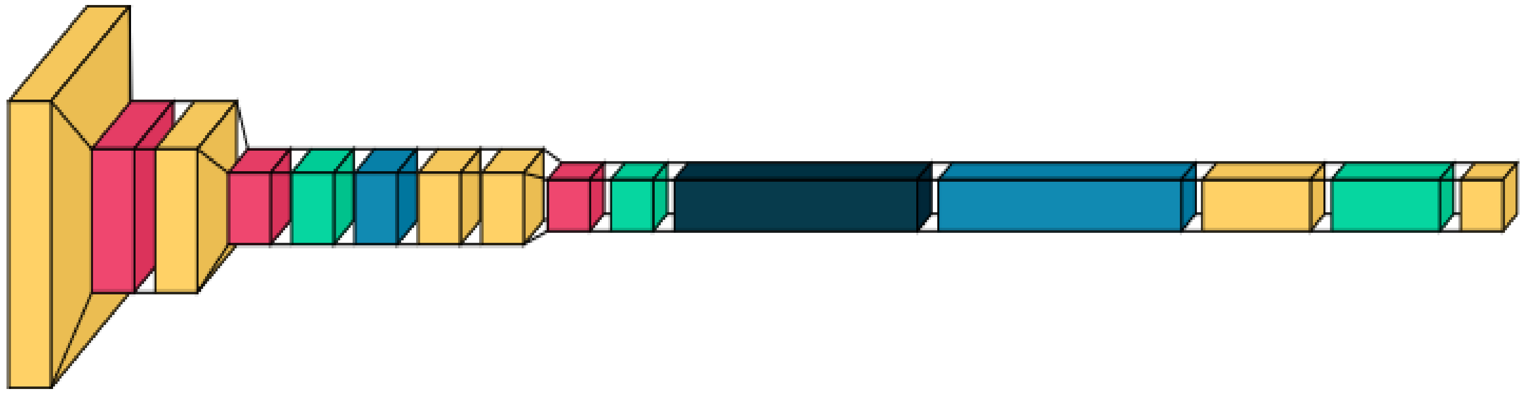
	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
T-shirt/top	817	0	22	24	1	2	126	0	8	0
Trouser	3	972	1	20	2	0	0	0	2	0
Pullover	13	0	848	14	58	0	66	0	1	0
Dress	12	0	7	927	21	0	32	0	1	0
Coat	0	1	41	32	878	0	47	0	1	0
Sandal	0	0	0	0	0	962	0	29	1	8
Shirt	102	1	48	33	90	1	712	0	13	0
Sneaker	0	0	0	0	0	6	0	969	0	25
Bag	0	0	1	3	2	5	7	2	979	1
Ankle boot	1	0	0	0	0	5	0	26	0	968

Model 3 : 2ndBackup_cnnAshutosh



```
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)

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

 batch_normalization (BatchN  (None, 7, 7, 64)         256
 ormalization)

 conv2d_2 (Conv2D)           (None, 7, 7, 128)         73856

 conv2d_3 (Conv2D)           (None, 7, 7, 128)         147584

 max_pooling2d_2 (MaxPooling  (None, 3, 3, 128)        0
 2D)

 dropout_1 (Dropout)         (None, 3, 3, 128)         0

 flatten (Flatten)           (None, 1152)              0

 batch_normalization_1 (Batc  (None, 1152)             4608
 hNormalization)

 dense (Dense)               (None, 512)               590336

 dropout_2 (Dropout)         (None, 512)               0

 dense_1 (Dense)             (None, 10)                5130

=================================================================
Total params: 840,586
Trainable params: 838,154
Non-trainable params: 2,432
```

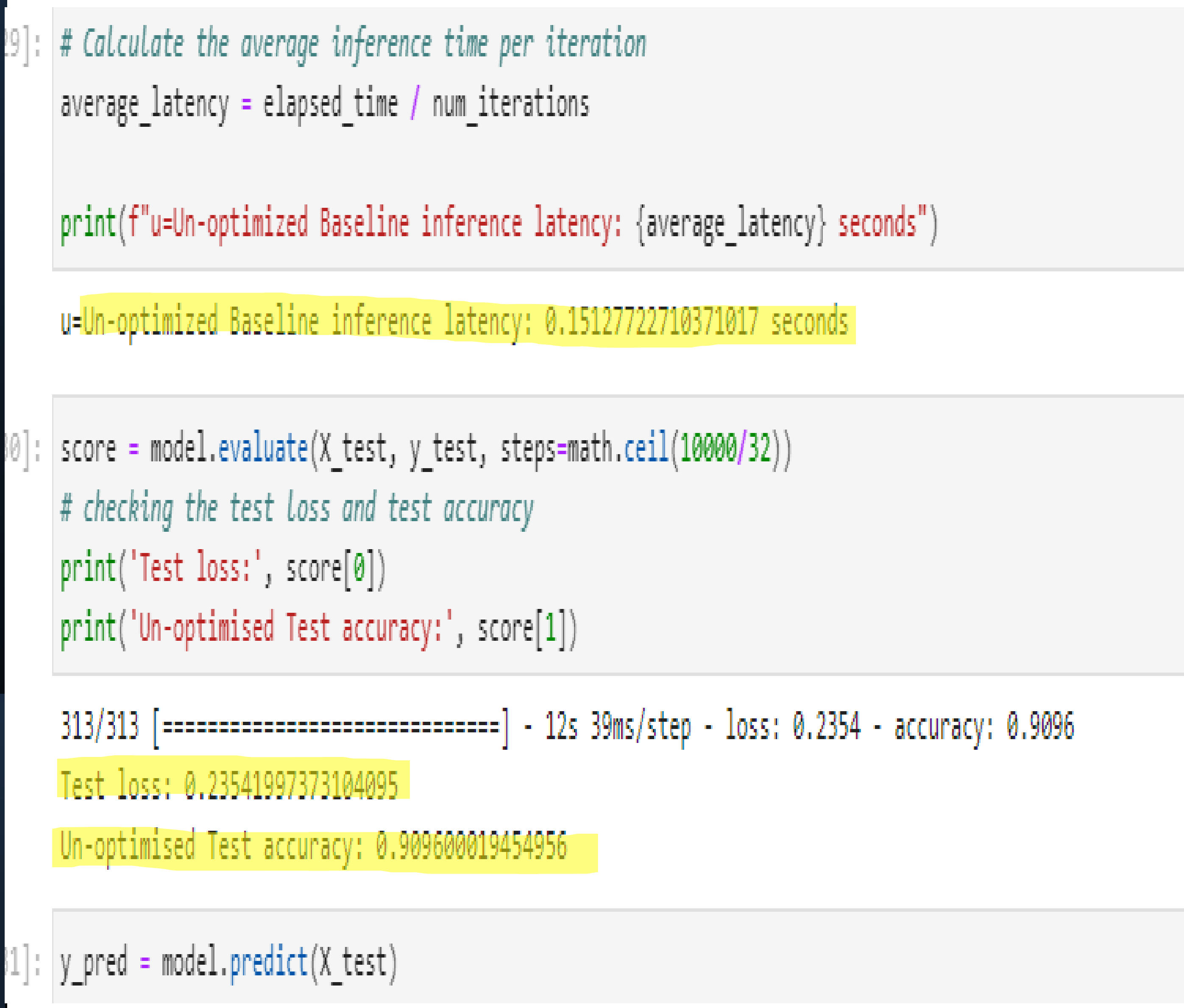
```
visualkeras.layered_view(model)
```



This is the Architecture of my Third CNN model, Inference related screenshots for this structure are on the next page

Model 3 : 2ndBackup_cnnAshutosh

UN-OPTIMIZED ACCURACY AND BASELINE INFERENCE LATENCY



```
# Calculate the average inference time per iteration
average_latency = elapsed_time / num_iterations


print(f"u=Un-optimized Baseline inference latency: {average_latency} seconds")
```

```
u=Un-optimized Baseline inference latency: 0.15127722710371017 seconds
```

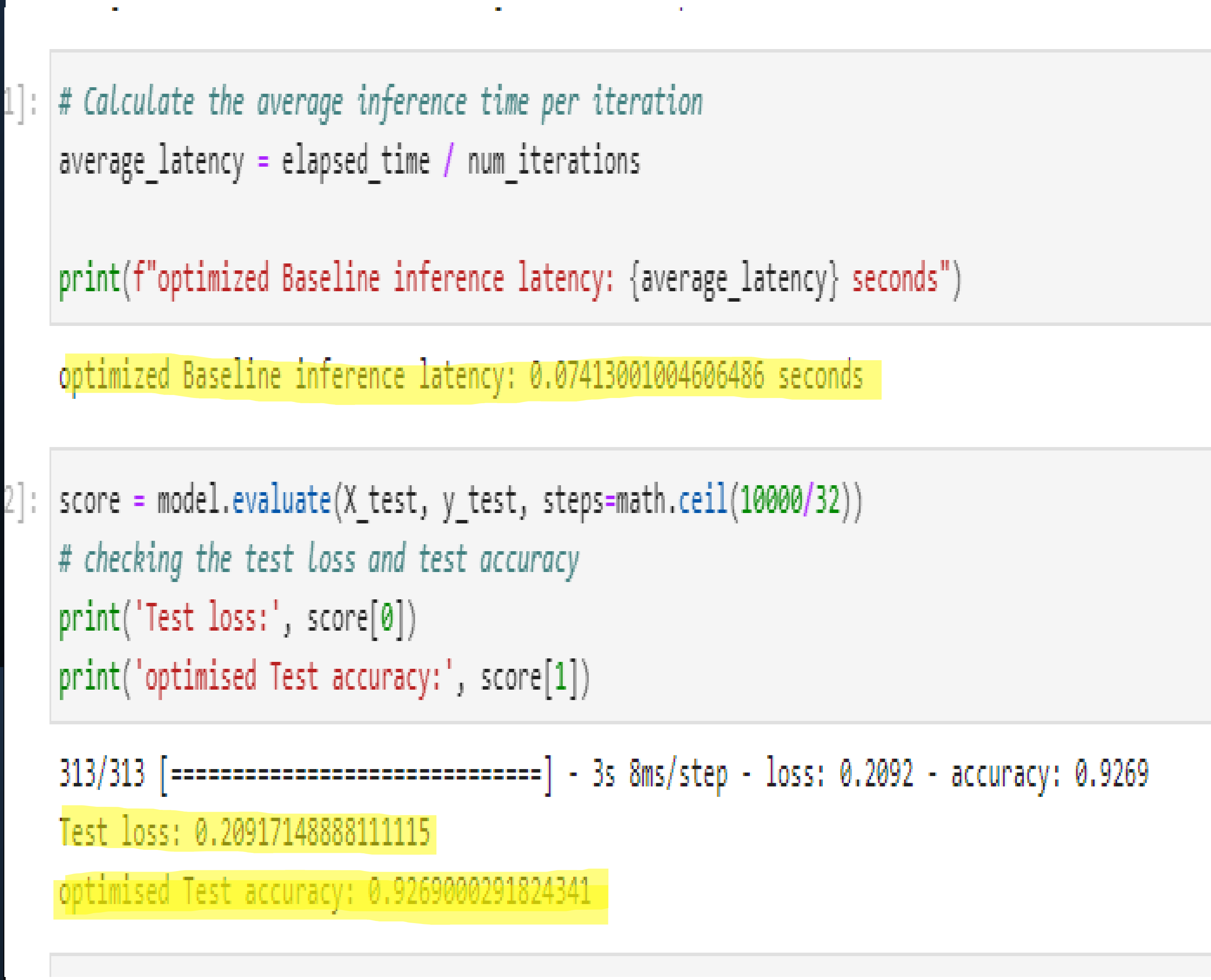
```
score = model.evaluate(X_test, y_test, steps=math.ceil(10000/32))
# checking the test loss and test accuracy
print('Test loss:', score[0])
print('Un-optimised Test accuracy:', score[1])
```

```
313/313 [==============================] - 12s 39ms/step - loss: 0.2354 - accuracy: 0.9096
Test loss: 0.23541997373104095
Un-optimised Test accuracy: 0.909600019454956
```

```
y_pred = model.predict(X_test)
```

Model 3 : 2ndBackup_cnnAshutosh

OPTIMIZED ACCURACY AND BASELINE INFERENCE LATENCY



```
# Calculate the average inference time per iteration
average_latency = elapsed_time / num_iterations


print(f"optimized Baseline inference latency: {average_latency} seconds")
```

```
optimized Baseline inference latency: 0.07413001004606486 seconds
```

```
score = model.evaluate(X_test, y_test, steps=math.ceil(10000/32))
# checking the test loss and test accuracy
print('Test loss:', score[0])
print('optimised Test accuracy:', score[1])
```

```
313/313 [==============================] - 3s 8ms/step - loss: 0.2092 - accuracy: 0.9269
Test loss: 0.20917148888111115
optimised Test accuracy: 0.9269000291824341
```

RESULTS

PARAMETERS	MODEL 1	MODEL 2	MODEL 3
ARCHITECTURE TYPE	CNN model built from scratch	CNN model built from scratch	CNN model built from scratch
TRAINING TIME	10400 seconds (3 hrs approx)	1200 seconds (20 mins approx)	4900 seconds (81 mins approx)
UN-OPTIMIZED ACCURACY	93%	90.32%	90.96%
UN-OPTIMIZED LOSS	19.11%	37.26%	23.5%
UN-OPTIMIZED BASELINE INFERENCE LATENCY	83.5 millisecond	87.5 millisecond	151.2 millisecond
OPTIMIZED ACCURACY	93.58% (reached 93.99 % once)	90.32%	92.69% (reached 93.55% once)
OPTIMIZED LOSS	17.69%	37.26%	20.9%
OPTIMIZED BASELINE INFERENCE LATENCY	77.7 millisecond	53 millisecond	74.1 millisecond
TYPE OF OPTIMIZATION USED	Intel Devcloud integrated oneDNN	Intel Devcloud integrated oneDNN	Intel Devcloud integrated oneDNN

Conclusion

- In conclusion, the Convolutional Neural Network (CNN) model developed for Fashion MNIST dataset has shown promising performance, especially when optimization techniques are applied.

- Initially, without optimization, the CNN model was able to achieve decent accuracy and perform reasonably well in classifying the fashion items. However, by incorporating optimization techniques, the model's performance was significantly enhanced.

- Optimization techniques, such as batch normalization and dropout regularization played a crucial role in improving the model's accuracy and generalization capabilities. Batch normalization helped in normalizing the activations of each layer, reducing the internal covariate shift and improving training speed. Dropout regularization prevented overfitting by randomly dropping a fraction of the neurons during training, thus encouraging the model to learn more robust and generalized features.

- By applying these optimization techniques, the model achieved higher accuracy and performed better in classifying the fashion items in the Fashion MNIST dataset. The optimization techniques helped reduce overfitting and improved the model's ability to handle unseen data.

- Overall, the comparison between the performance of the CNN model with and without optimization clearly demonstrates the effectiveness of optimization techniques in enhancing the model's performance. The optimized model achieved higher accuracy and better generalization, making it more reliable and capable of accurately classifying fashion items in real-world scenarios.

Project Link

Just to be on the safer side I will upload all the required files in github as well as a google drive and I will be pasting the link here for easy access.

Link – https://drive.google.com/drive/folders/16JSGsBdVo1QJcAtrCXD1d-R_Ff3zUP6V?usp=sharing

NOTE: please upload the required dataset to your notebook and paste the exact path in the code to load the dataset to run the model properly.

References

- Youtube.
- Stackoverflow.