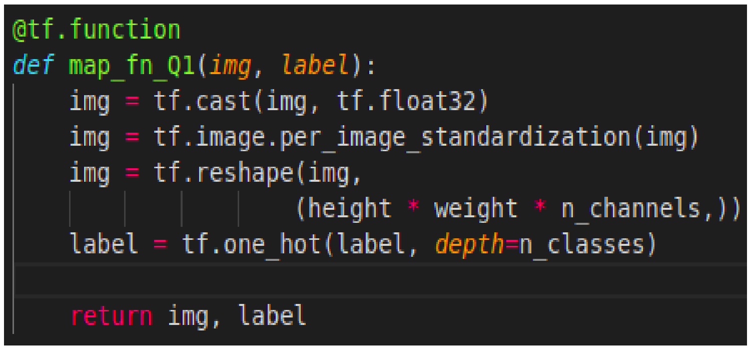
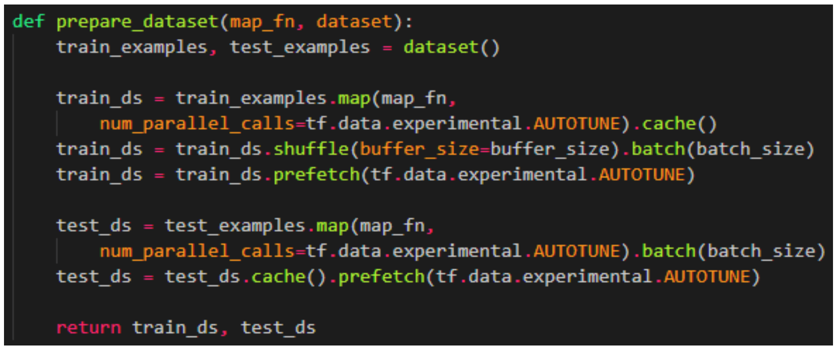
1. Two-layer fully-connected network with single batch (GD) backpropagation.

(a) Data preprocessing

CIFAR10 consists of 60,000 RGB images with 32x32 size. Dataset contains an equal number of samples per class, so the dataset is well balanced. CIFAR10 dataset downloaded and loaded using tensorflow_datasets which is capable of advanced preprocessing techniques to Extract, transform & Load data with much less time.



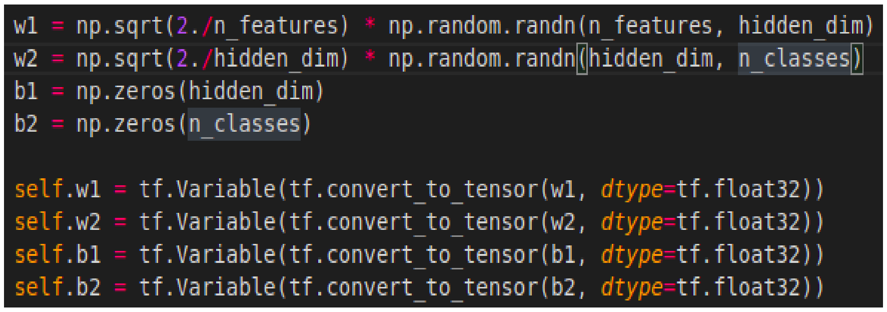
```
@tf.function
def map_fn_Q1(img, label):
    img = tf.cast(img, tf.float32)
    img = tf.image.per_image_standardization(img)
    img = tf.reshape(img,
                     (height * weight * n_channels,))
    label = tf.one_hot(label, depth=n_classes)

    return img, label
```



```
def prepare_dataset(map_fn, dataset):
    train_examples, test_examples = dataset()

    train_ds = train_examples.map(map_fn,
        num_parallel_calls=tf.data.experimental.AUTOTUNE).cache()
    train_ds = train_ds.shuffle(buffer_size=buffer_size).batch(batch_size)
    train_ds = train_ds.prefetch(tf.data.experimental.AUTOTUNE)

    test_ds = test_examples.map(map_fn,
        num_parallel_calls=tf.data.experimental.AUTOTUNE).batch(batch_size)
    test_ds = test_ds.cache().prefetch(tf.data.experimental.AUTOTUNE)

    return train_ds, test_ds
```

tf.function constructs a callable that executes a TensorFlow graph.using **map_fn_Q1** it will apply **standard normalization** of 3D images, **label one hot encoding** and other operations on each batch of images. **Prepare_dataset** function, use map_fn_Q1 and other steps to obtain the final data. caching data used to improve performance of the model. Since single batch backpropagation is the optimizer, batch size is equal to size of the entire dataset.

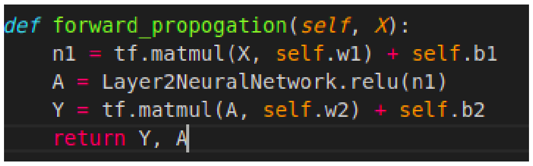
(b) weight initialization



```
w1 = np.sqrt(2./n_features) * np.random.randn(n_features, hidden_dim)
w2 = np.sqrt(2./hidden_dim) * np.random.randn(hidden_dim, n_classes)
b1 = np.zeros(hidden_dim)
b2 = np.zeros(n_classes)

self.w1 = tf.Variable(tf.convert_to_tensor(w1, dtype=tf.float32))
self.w2 = tf.Variable(tf.convert_to_tensor(w2, dtype=tf.float32))
self.b1 = tf.Variable(tf.convert_to_tensor(b1, dtype=tf.float32))
self.b2 = tf.Variable(tf.convert_to_tensor(b2, dtype=tf.float32))
```

Since the activation mostly used in the model is **ReLU** for weights initialized using **MSRA.** To initialize weights it uses the tf.Variables API, **because tensorflow only allows to create trainable parameters using tf.Variables because tensorflow 2.0 does not use placeholders.** By tensorflow 2.0 all the computations are done using EagerExecution. Since graphs create dynamically. No need for creating sessions and being able to perform computations directly.

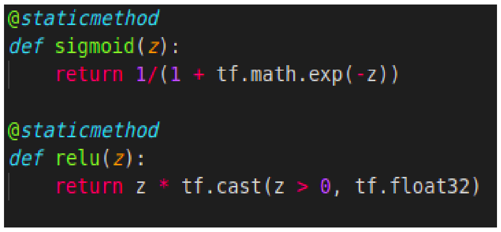
(c) Forward Propagation



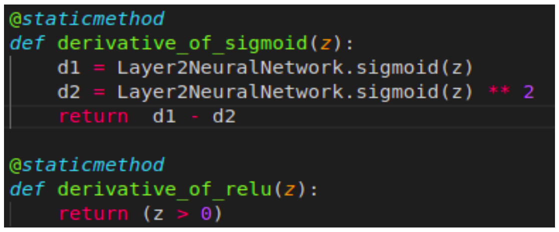
```
def forward_propogation(self, X):
    n1 = tf.matmul(X, self.w1) + self.b1
    A = Layer2NeuralNetwork.relu(n1)
    Y = tf.matmul(A, self.w2) + self.b2
    return Y, A
```

Forward propagation maintains the flow of the model architecture. The most basic operations happening in forward propagation is matrix multiplication & layer activation. This model contains 2 layers; one hidden layer and the output layer. **hidden layer used ReLU activation and no activation(logits) used in the output layer.**

(d) activations and their derivatives



```
@staticmethod
def sigmoid(z):
    return 1/(1 + tf.math.exp(-z))

@staticmethod
def relu(z):
    return z * tf.cast(z > 0, tf.float32)
```

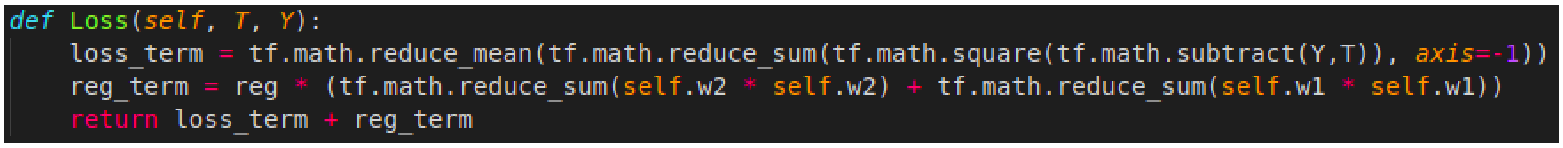


```
@staticmethod
def derivative_of_sigmoid(z):
    d1 = Layer2NeuralNetwork.sigmoid(z)
    d2 = Layer2NeuralNetwork.sigmoid(z) ** 2
    return  d1 - d2

@staticmethod
def derivative_of_relu(z):
    return (z > 0)
```

Activation functions are basically used for mapping non-linearities between inputs & outputs. So it basically helps to catch the hidden patterns in data. The derivatives of activation functions are useful for backpropagation. Here contains about sigmoid & relu functions

(e) Loss Function

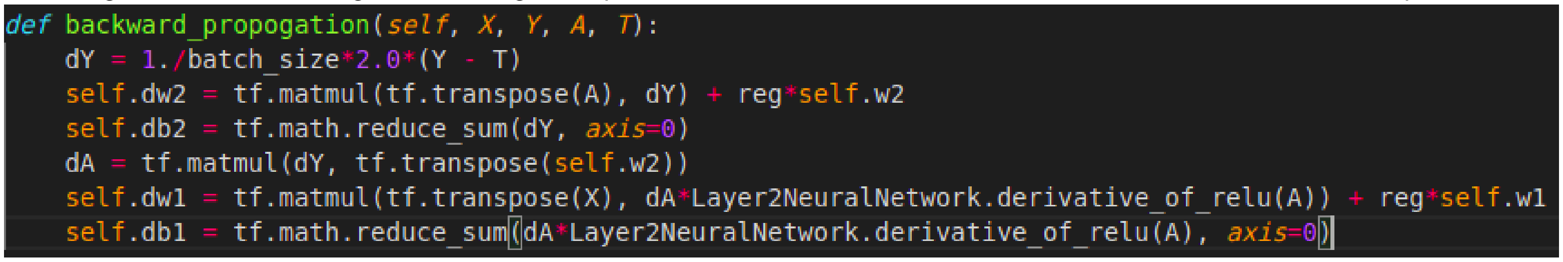
The purpose of the Loss function is to calculate the difference(error) between ground truth labels and the prediction values by the model. These loss functions can be used to evaluate models. Here used **Mean Squared Error** as the loss including regularization terms.



```
def Loss(self, T, Y):
    loss_term = tf.math.reduce_mean(tf.math.reduce_sum(tf.math.square(tf.math.subtract(Y,T)), axis=-1))
    reg_term = reg * (tf.math.reduce_sum(self.w2 * self.w2) + tf.math.reduce_sum(self.w1 * self.w1))
    return loss_term + reg_term
```

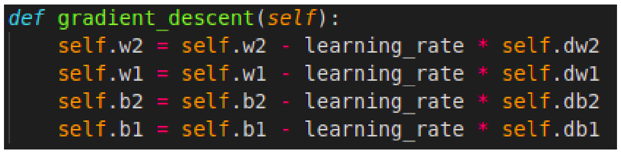
(f) Backward Propagation

For gradient descent optimizer it has to calculate derivative terms to update weights and bias values. To calculate these derivatives it has to **backpropagate the error through each layer.** Using this weight update it helps gradient descent helps to minimize the loss and find global minima. Through the learning rate optimizations it has to decide how much it should learn within one step.



```
def backward_propogation(self, X, Y, A, T):
    dY = 1./batch_size*2.0*(Y - T)
    self.dw2 = tf.matmul(tf.transpose(A), dY) + reg*self.w2
    self.db2 = tf.math.reduce_sum(dY, axis=0)
    dA = tf.matmul(dY, tf.transpose(self.w2))
    self.dw1 = tf.matmul(tf.transpose(X), dA*Layer2NeuralNetwork.derivative_of_relu(A)) + reg*self.w1
    self.db1 = tf.math.reduce_sum(dA*Layer2NeuralNetwork.derivative_of_relu(A), axis=0)
```

(g) Gradient Descent Optimizer

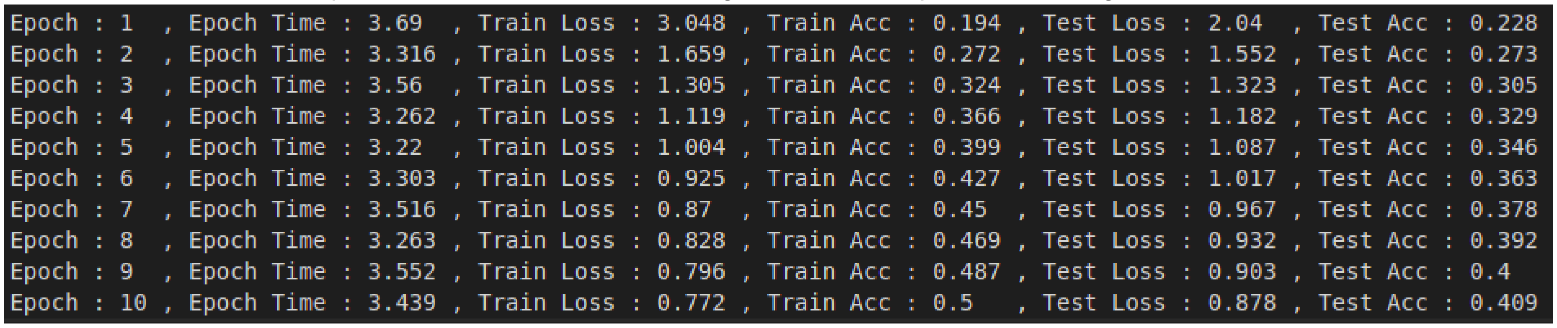


```
def gradient_descent(self):
    self.w2 = self.w2 - learning_rate * self.dw2
    self.w1 = self.w1 - learning_rate * self.dw1
    self.b2 = self.b2 - learning_rate * self.db2
    self.b1 = self.b1 - learning_rate * self.db1
```

Using the gradients of the weights which were calculated from Backpropagation, Gradient descent optimizer used to minimize the loss function. **Objective of the gradient descent is to find Global minima** of the complex loss function whether it may be convex or non-convex.

(h) Model Training

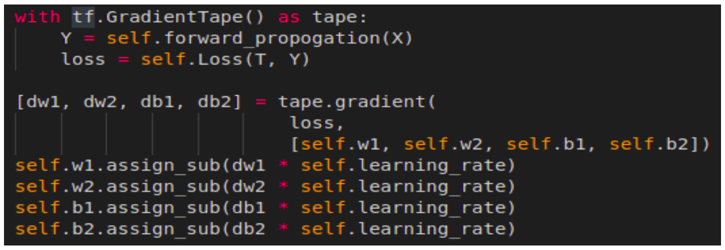
The 2 layer model trained for 100 epochs and the following figure contains the loss and accuracy variations of the both training set and test set for the first 10 epochs. Also it included the training time for each epoch for the single batch.



```
Epoch : 1  , Epoch Time : 3.69  , Train Loss : 3.048 , Train Acc : 0.194 , Test Loss : 2.04  , Test Acc : 0.228
Epoch : 2  , Epoch Time : 3.316 , Train Loss : 1.659 , Train Acc : 0.272 , Test Loss : 1.552 , Test Acc : 0.273
Epoch : 3  , Epoch Time : 3.56  , Train Loss : 1.305 , Train Acc : 0.324 , Test Loss : 1.323 , Test Acc : 0.305
Epoch : 4  , Epoch Time : 3.262 , Train Loss : 1.119 , Train Acc : 0.366 , Test Loss : 1.182 , Test Acc : 0.329
Epoch : 5  , Epoch Time : 3.22  , Train Loss : 1.004 , Train Acc : 0.399 , Test Loss : 1.087 , Test Acc : 0.346
Epoch : 6  , Epoch Time : 3.303 , Train Loss : 0.925 , Train Acc : 0.427 , Test Loss : 1.017 , Test Acc : 0.363
Epoch : 7  , Epoch Time : 3.516 , Train Loss : 0.87  , Train Acc : 0.45  , Test Loss : 0.967 , Test Acc : 0.378
Epoch : 8  , Epoch Time : 3.263 , Train Loss : 0.828 , Train Acc : 0.469 , Test Loss : 0.932 , Test Acc : 0.392
Epoch : 9  , Epoch Time : 3.552 , Train Loss : 0.796 , Train Acc : 0.487 , Test Loss : 0.903 , Test Acc : 0.4
Epoch : 10 , Epoch Time : 3.439 , Train Loss : 0.772 , Train Acc : 0.5   , Test Loss : 0.878 , Test Acc : 0.409
```

2. Two-layer fully-connected network by AutoGrad with GradientTape.

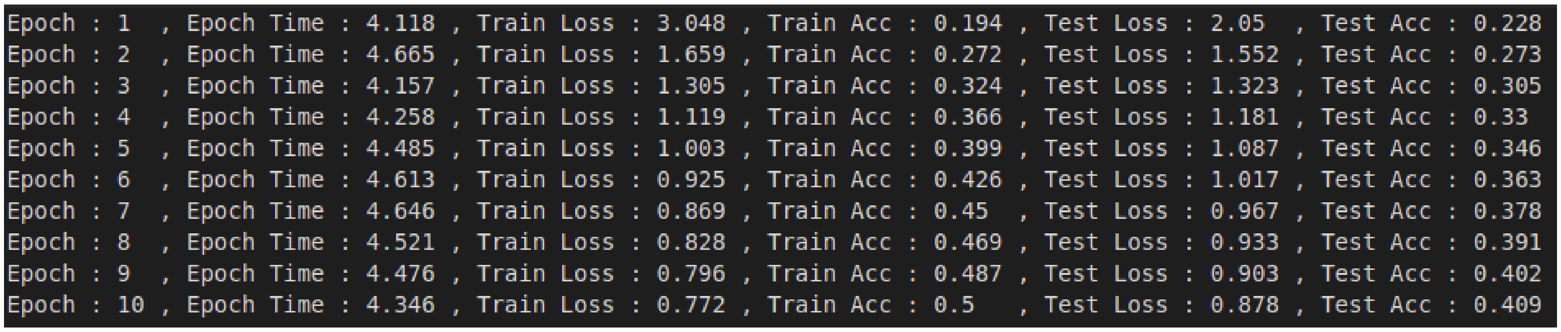
(a) AutoGrad with GradientTape



```
with tf.GradientTape() as tape:
    Y = self.forward_propogation(X)
    loss = self.Loss(T, Y)

[dw1, dw2, db1, db2] = tape.gradient(
                        loss,
                        [self.w1, self.w2, self.b1, self.b2])
self.w1.assign_sub(dw1 * self.learning_rate)
self.w2.assign_sub(dw2 * self.learning_rate)
self.b1.assign_sub(db1 * self.learning_rate)
self.b2.assign_sub(db2 * self.learning_rate)
```

Before version 2.0 Tensorflow wasn't enabled by default eager execution. Which means it has to create sessions & execute the whole process inside that session because graphs are static in earlier versions**. Tensorflow 2.0 enabled default eager execution**. So it can calculate gradients of trainable params dynamically with autograd. After performing forward propagation, calculate loss and apply backpropagation on trainable parameters.

(b) Model Training

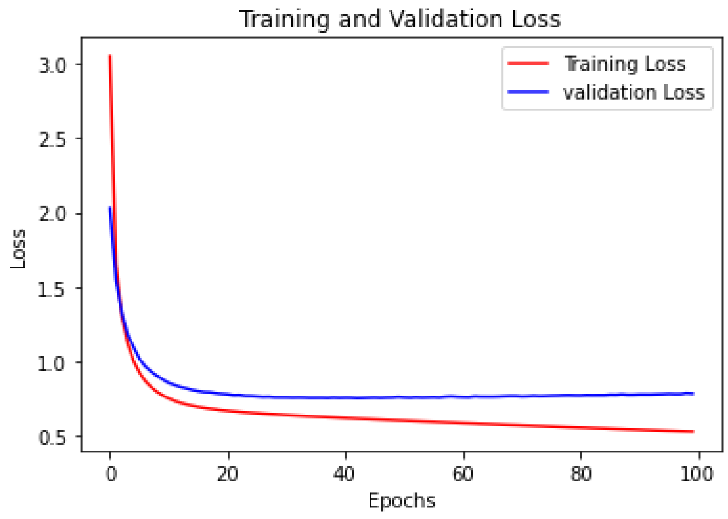


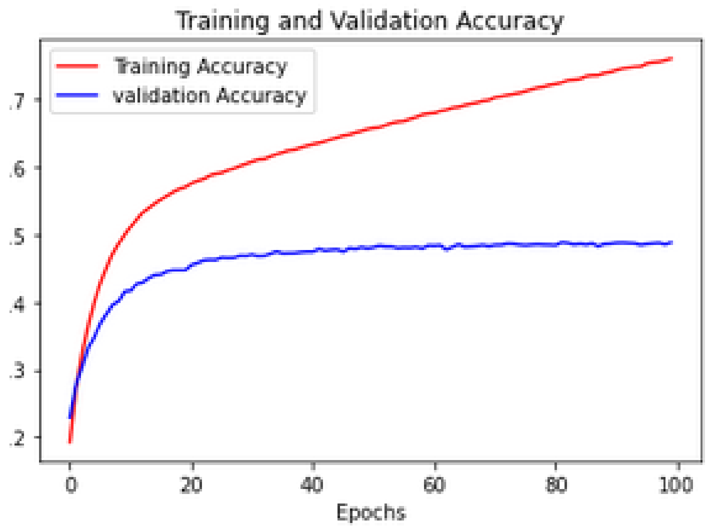
```
Epoch : 1  , Epoch Time : 4.118 , Train Loss : 3.048 , Train Acc : 0.194 , Test Loss : 2.05  , Test Acc : 0.228
Epoch : 2  , Epoch Time : 4.665 , Train Loss : 1.659 , Train Acc : 0.272 , Test Loss : 1.552 , Test Acc : 0.273
Epoch : 3  , Epoch Time : 4.157 , Train Loss : 1.305 , Train Acc : 0.324 , Test Loss : 1.323 , Test Acc : 0.305
Epoch : 4  , Epoch Time : 4.258 , Train Loss : 1.119 , Train Acc : 0.366 , Test Loss : 1.181 , Test Acc : 0.33
Epoch : 5  , Epoch Time : 4.485 , Train Loss : 1.003 , Train Acc : 0.399 , Test Loss : 1.087 , Test Acc : 0.346
Epoch : 6  , Epoch Time : 4.613 , Train Loss : 0.925 , Train Acc : 0.426 , Test Loss : 1.017 , Test Acc : 0.363
Epoch : 7  , Epoch Time : 4.646 , Train Loss : 0.869 , Train Acc : 0.45  , Test Loss : 0.967 , Test Acc : 0.378
Epoch : 8  , Epoch Time : 4.521 , Train Loss : 0.828 , Train Acc : 0.469 , Test Loss : 0.933 , Test Acc : 0.391
Epoch : 9  , Epoch Time : 4.476 , Train Loss : 0.796 , Train Acc : 0.487 , Test Loss : 0.903 , Test Acc : 0.402
Epoch : 10 , Epoch Time : 4.346 , Train Loss : 0.772 , Train Acc : 0.5   , Test Loss : 0.878 , Test Acc : 0.409
```

Comparison of Techniques and Results of Part 1 & Part 2

Parameter	Value
regularization rate	5e-6
epoches	10
Hidden units	1024
Learning rate	1e-3
decay	0.9

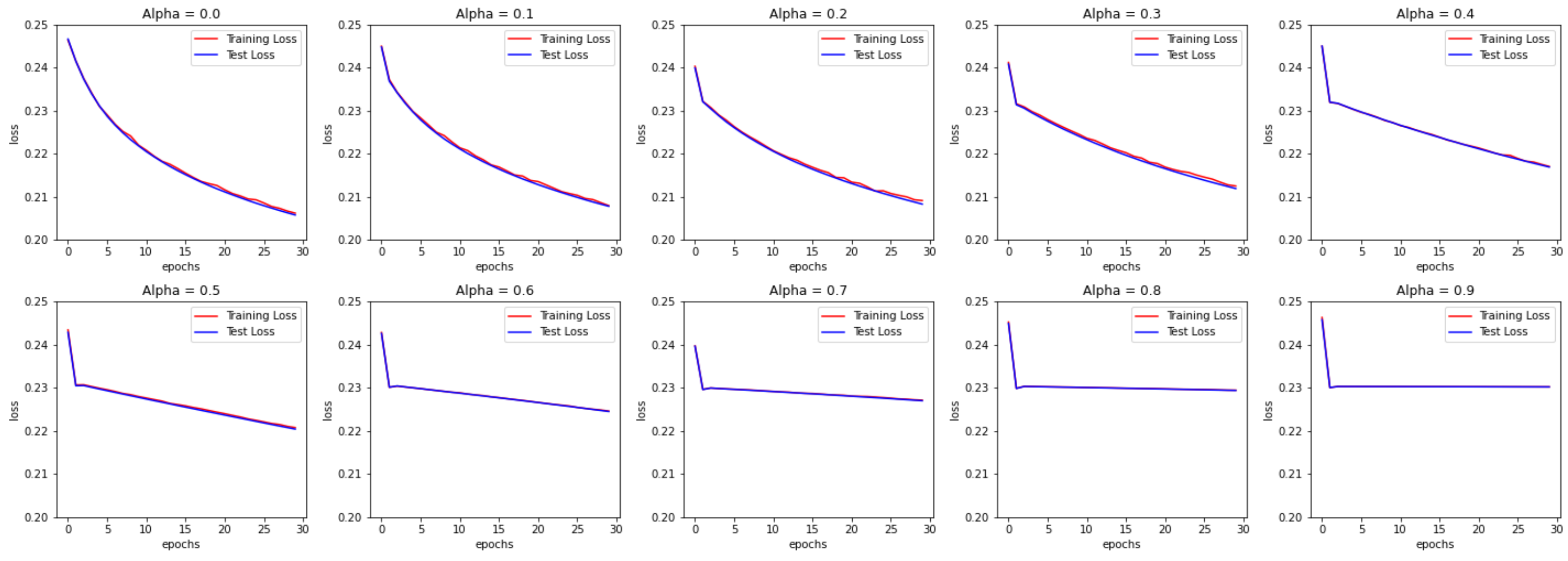
When it is considered forward propagation is implemented using ReLU activations because it has several advantages over sigmoid and tanh. The main problems of sigmoid and tanh are **computational expensive and saturated regions kill gradients.** ReLU uses **max operator, so it's computational efficient and doesn't saturate in positive regions**(linearly grow). When initialized with small random number activations are saturated and local gradients become zero. So the entire downstream tends to zero. So it's better to go with Xavier initialization. But **Xavier initialization assumes activations to be zero centered.** Since ReLU is not zero centered, **activations collapse.** MSRA is better, because activations are nicely scaled for all layers. For the loss function **Mean Squared Error** has been used. To **reduce overfitting of the model L2 regularization has been applied.** The table contains the model hyperparameters used for the task. Since it uses **Batch gradient descent, weight updates become much slower for larger datasets.** Also batch gradient descent has some disadvantages such as complex non convex loss surfaces may not converge into global minima and for large datasets, data vectorization may not fit with the memory. When comparing model training of Part 1and Part 2 iit concludes both models have same loss variation and accuracy variation. But when **comparing epoch time Manual derivation took a bit less than the autograd calculation.** Following plots contain loss and accuracy variations for both models because both models have the same variation.



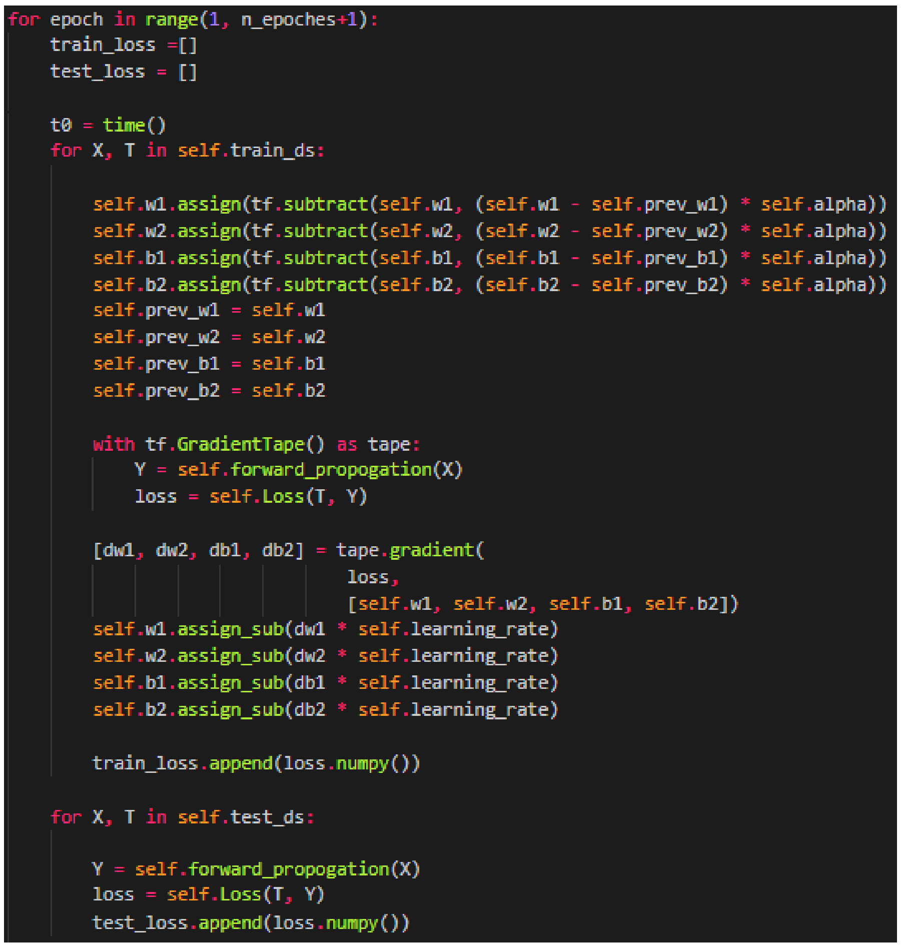


For both models the rate of convergence of the train set is significantly higher than the test set. Similar kinds of things can be seen for accuracy as well. This will clearly lead to model overfitting. **To avoid overfitting it should use mini batch gradient descent.** Because smaller batch sizes can offer an internal regularizing effect. Also to train with smaller batch sizes it has to reduce learning rate to ensure stability of learning.

3. Loss Variation for Custom Weight Update $(\theta_t\alpha = (1-\alpha)\theta_t + \alpha\,\theta_{t-1})$



First it should perform this custom update. After that it should perform gradient descent optimizer. So it should need to backpropagate & find gradients. For that it used Tensorlow Autograd (GradientTape). Similar set of hyper parameters used in above tasks, here used as well.



```python
for epoch in range(1, n_epoches+1):
    train_loss =[]
    test_loss = []

    t0 = time()
    for X, T in self.train_ds:

        self.w1.assign(tf.subtract(self.w1, (self.w1 - self.prev_w1) * self.alpha))
        self.w2.assign(tf.subtract(self.w2, (self.w2 - self.prev_w2) * self.alpha))
        self.b1.assign(tf.subtract(self.b1, (self.b1 - self.prev_b1) * self.alpha))
        self.b2.assign(tf.subtract(self.b2, (self.b2 - self.prev_b2) * self.alpha))
        self.prev_w1 = self.w1
        self.prev_w2 = self.w2
        self.prev_b1 = self.b1
        self.prev_b2 = self.b2

        with tf.GradientTape() as tape:
            Y = self.forward_propogation(X)
            loss = self.Loss(T, Y)

        [dw1, dw2, db1, db2] = tape.gradient(
                                loss,
                                [self.w1, self.w2, self.b1, self.b2])
        self.w1.assign_sub(dw1 * self.learning_rate)
        self.w2.assign_sub(dw2 * self.learning_rate)
        self.b1.assign_sub(db1 * self.learning_rate)
        self.b2.assign_sub(db2 * self.learning_rate)

        train_loss.append(loss.numpy())

    for X, T in self.test_ds:

        Y = self.forward_propogation(X)
        loss = self.Loss(T, Y)
        test_loss.append(loss.numpy())
```

When it analyses the above set of plots for train & test accuracies, it can clearly see that when increasing alpha the loss increases. Also the rate of convergence decreases. These variations can be described by dividing the range of alpha to 3 parts.

- $\alpha = 0$ ($\theta_t\alpha = \theta_t$)

When $\alpha = 0$, it really does nothing. Because this update doesn't give any impact for weights by previous weights . So update only relies on current weights.

- $0 < \alpha < 1$

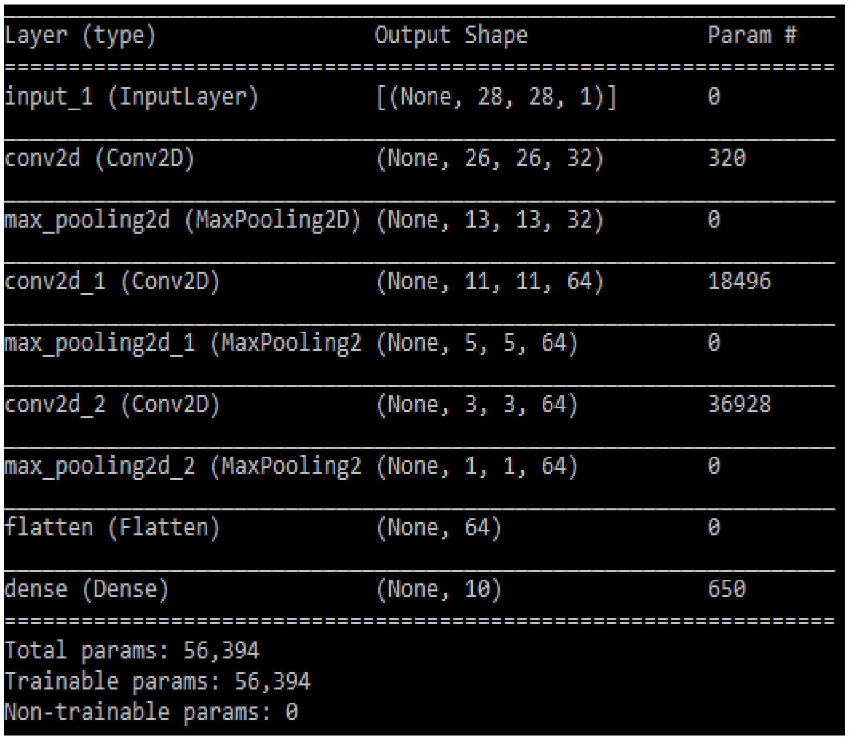
In this particular range, when alpha increases it does more impact for weights by previous weights. So that concludes when the impact of previous weights increases, it slows down the learning process. (may be stop the learning process)

- $\alpha = 1$ ($\theta_t\alpha = \theta_{t-1}$)

Weights only depend on previous weights. In this case it stops the learning process & loss of the model becomes constant.

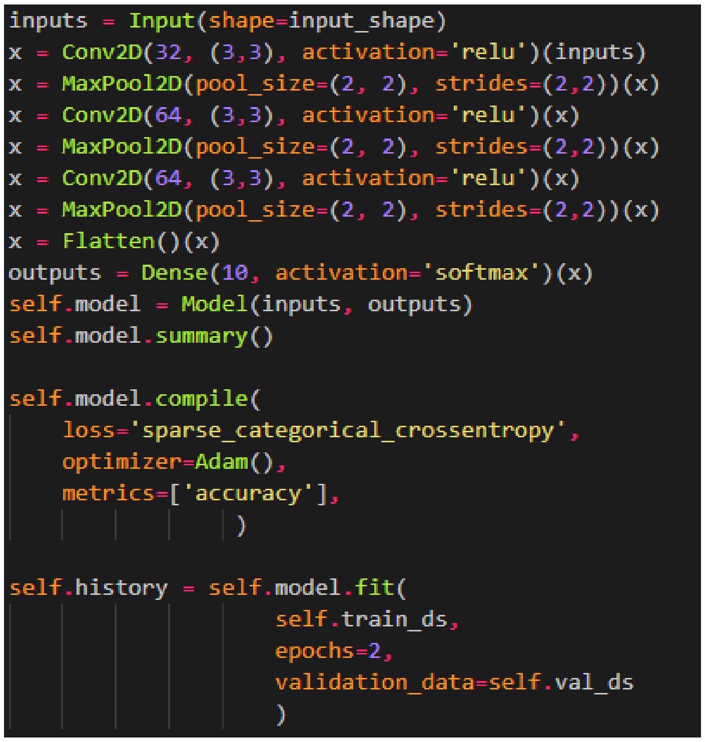
4. Convolutional Neural Network for MNIST

(a) **Model Summary**



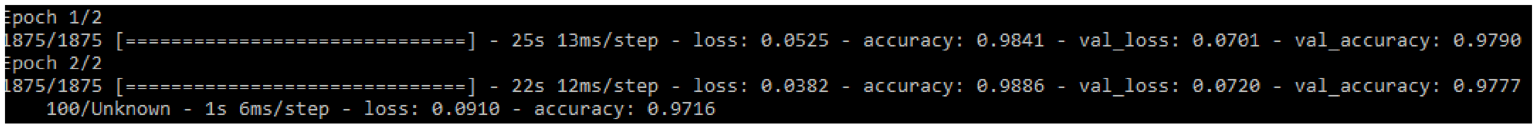
```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 28, 28, 1)]       0

conv2d (Conv2D)              (None, 26, 26, 32)        320

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

conv2d_1 (Conv2D)            (None, 11, 11, 64)        18496

max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0

conv2d_2 (Conv2D)            (None, 3, 3, 64)          36928

max_pooling2d_2 (MaxPooling2 (None, 1, 1, 64)          0

flatten (Flatten)            (None, 64)                0

dense (Dense)                (None, 10)                650
=================================================================
Total params: 56,394
Trainable params: 56,394
Non-trainable params: 0
```

Model Compiling & Training Process

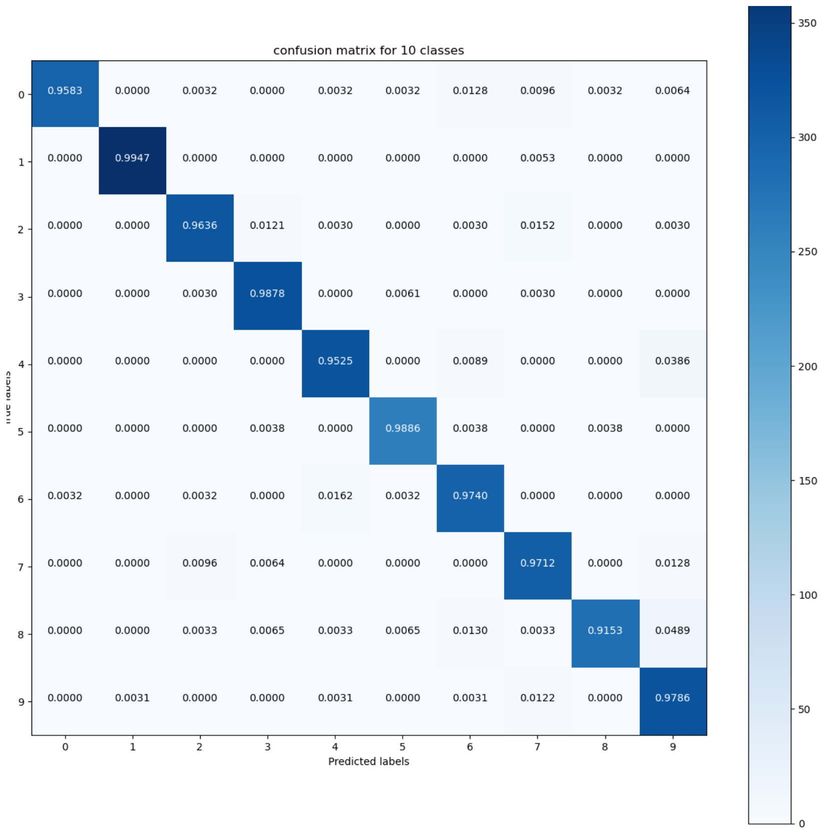


```python
inputs = Input(shape=input_shape)
x = Conv2D(32, (3,3), activation='relu')(inputs)
x = MaxPool2D(pool_size=(2, 2), strides=(2,2))(x)
x = Conv2D(64, (3,3), activation='relu')(x)
x = MaxPool2D(pool_size=(2, 2), strides=(2,2))(x)
x = Conv2D(64, (3,3), activation='relu')(x)
x = MaxPool2D(pool_size=(2, 2), strides=(2,2))(x)
x = Flatten()(x)
outputs = Dense(10, activation='softmax')(x)
self.model = Model(inputs, outputs)
self.model.summary()

self.model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer=Adam(),
    metrics=['accuracy'],
            )

self.history = self.model.fit(
                    self.train_ds,
                    epochs=2,
                    validation_data=self.val_ds
                    )
```

Model Results & Evaluation (Confusion Matrix) Process



```
Epoch 1/2
1875/1875 [==============================] - 25s 13ms/step - loss: 0.0525 - accuracy: 0.9841 - val_loss: 0.0701 - val_accuracy: 0.9790
Epoch 2/2
1875/1875 [==============================] - 22s 12ms/step - loss: 0.0382 - accuracy: 0.9886 - val_loss: 0.0720 - val_accuracy: 0.9777
    100/Unknown - 1s 6ms/step - loss: 0.0910 - accuracy: 0.9716
```



	0	1	2	3	4	5	6	7	8	9
0	0.9583	0.0000	0.0032	0.0000	0.0032	0.0032	0.0128	0.0096	0.0032	0.0064
1	0.0000	0.9947	0.0000	0.0000	0.0000	0.0000	0.0000	0.0053	0.0000	0.0000
2	0.0000	0.0000	0.9636	0.0121	0.0030	0.0000	0.0030	0.0152	0.0000	0.0030
3	0.0000	0.0000	0.0030	0.9878	0.0000	0.0061	0.0000	0.0030	0.0000	0.0000
4	0.0000	0.0000	0.0000	0.0000	0.9525	0.0000	0.0089	0.0000	0.0000	0.0386
5	0.0000	0.0000	0.0000	0.0038	0.0000	0.9886	0.0038	0.0000	0.0038	0.0000
6	0.0032	0.0000	0.0032	0.0000	0.0162	0.0032	0.9740	0.0000	0.0000	0.0000
7	0.0000	0.0000	0.0096	0.0064	0.0000	0.0000	0.0000	0.9712	0.0000	0.0128
8	0.0000	0.0000	0.0033	0.0065	0.0033	0.0065	0.0130	0.0033	0.9153	0.0489
9	0.0000	0.0031	0.0000	0.0000	0.0031	0.0000	0.0031	0.0122	0.0000	0.9786

Since there are more than 2 classes **sparse categorical cross entropy** used as the loss function. Also accuracy used as the evaluation metric.

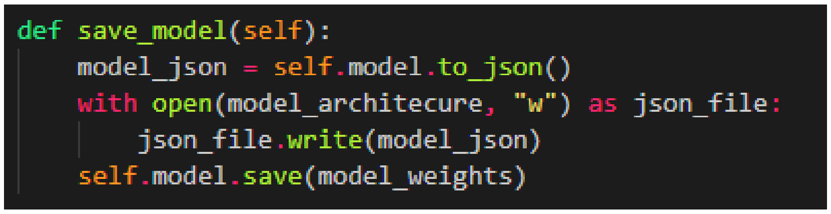
After training for 2 epochs,

Train Accuracy	:	0.9886
Validation Accuracy	:	0.977733
Test Accuracy	:	0.9716

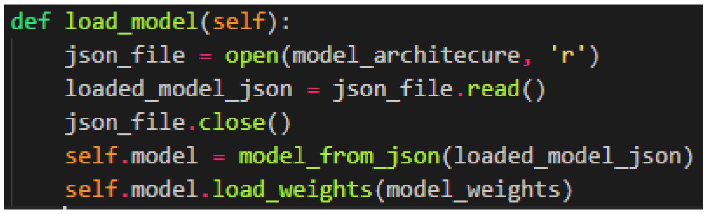
It seems that the model is well fitted with data because there is no clear overfitting or underfitting.

Confusion matrix is an important tool which is used to evaluate the model prediction per each class. Whole ground truth & prediction data can be divided into 4 groups; True Positives, True Negatives, False Positives & False Negatives. By calculating these parameter values for predictions per each class it will generate the matrix. The diagonal of the confusion matrix is corresponding to correct predictions while the rest of the indices related to false positives. Here we can see that the diagonal is much stronger when it compares with other indices. Since that concludes the model performance is at a really good satisfactory level.

(b) Save & Load Model Weights & Architecture



```python
def save_model(self):
    model_json = self.model.to_json()
    with open(model_architecure, "w") as json_file:
        json_file.write(model_json)
    self.model.save(model_weights)
```

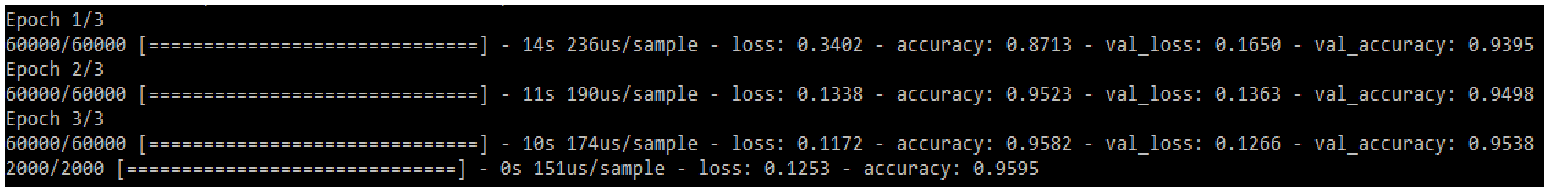


```python
def load_model(self):
    json_file = open(model_architecure, 'r')
    loaded_model_json = json_file.read()
    json_file.close()
    self.model = model_from_json(loaded_model_json)
    self.model.load_weights(model_weights)
```

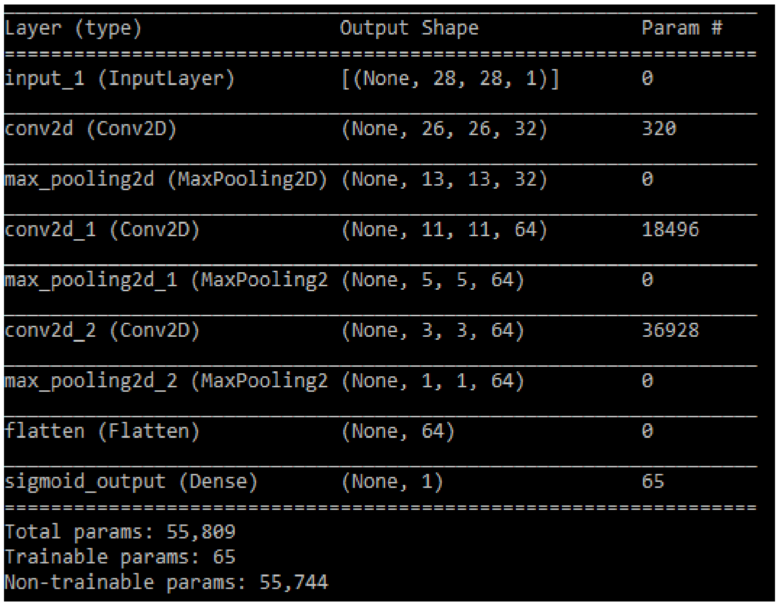
(c) Transfer Learn the Network with two classes

Transfer Learning technique is a method when you freeze some layers, and do additional training to the rest of layers to adjust for your purposes and goals. Deep learning models are end-to-end, so capable of extracting features & classifying them using single model structure. Since both tasks use the same inputs , hierarchical features used in both tasks are similar. So it's ok to change & train only the output sigmoid layer while setting other layer weights are untrainable.

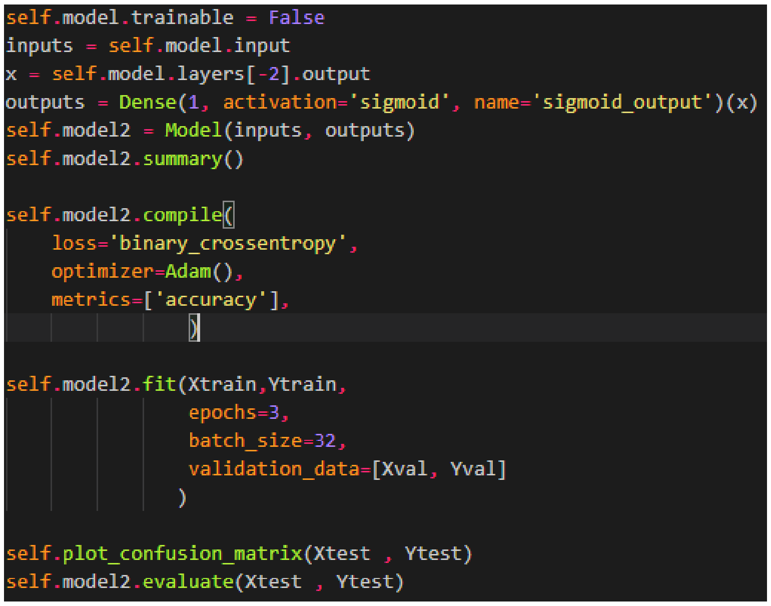
Model Results



```
Epoch 1/3
60000/60000 [==============================] - 14s 236us/sample - loss: 0.3402 - accuracy: 0.8713 - val_loss: 0.1650 - val_accuracy: 0.9395
Epoch 2/3
60000/60000 [==============================] - 11s 190us/sample - loss: 0.1338 - accuracy: 0.9523 - val_loss: 0.1363 - val_accuracy: 0.9498
Epoch 3/3
60000/60000 [==============================] - 10s 174us/sample - loss: 0.1172 - accuracy: 0.9582 - val_loss: 0.1266 - val_accuracy: 0.9538
2000/2000 [==============================] - 0s 151us/sample - loss: 0.1253 - accuracy: 0.9595
```



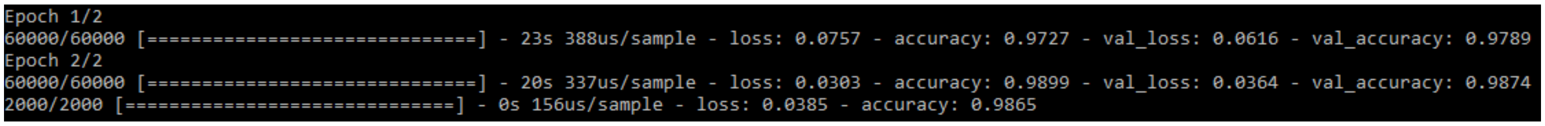
```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 3, 3, 64)          36928
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 1, 1, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 64)                0
_________________________________________________________________
sigmoid_output (Dense)       (None, 1)                 65
=================================================================
Total params: 55,809
Trainable params: 65
Non-trainable params: 55,744
```



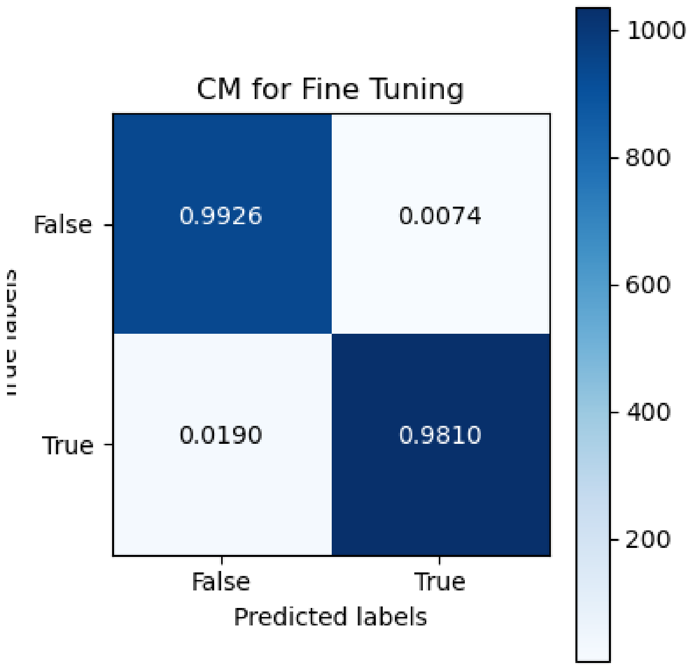
```
self.model.trainable = False
inputs = self.model.input
x = self.model.layers[-2].output
outputs = Dense(1, activation='sigmoid', name='sigmoid_output')(x)
self.model2 = Model(inputs, outputs)
self.model2.summary()

self.model2.compile(
    loss='binary_crossentropy',
    optimizer=Adam(),
    metrics=['accuracy'],
            )

self.model2.fit(Xtrain,Ytrain,
            epochs=3,
            batch_size=32,
            validation_data=[Xval, Yval]
           )

self.plot_confusion_matrix(Xtest , Ytest)
self.model2.evaluate(Xtest , Ytest)
```

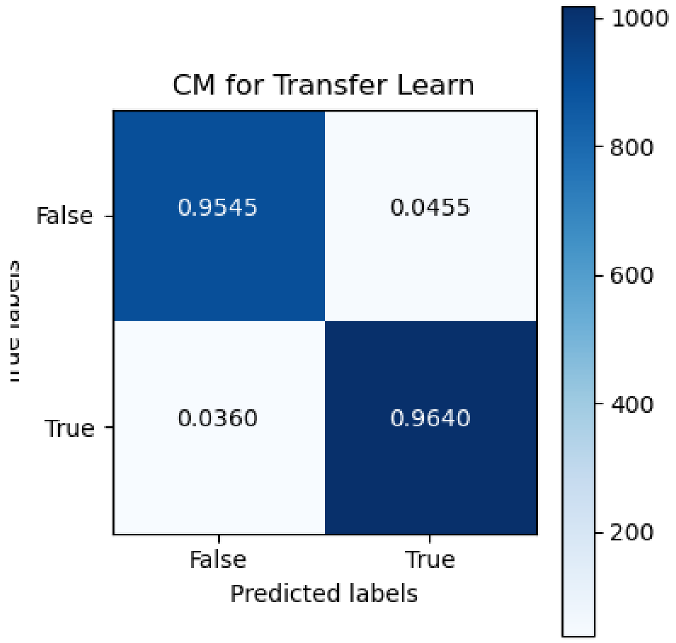
(d) Finetune the Network with two classes

Fine tuning of the network is the process that adjusts hyper-parameters (batch size, learning rate) & retrains the whole network. In more it should decrease the learning rate, because bigger steps in GD, leads to poor model results. This will leads to good results because of retraining, but more training time.



```
Epoch 1/2
60000/60000 [==============================] - 23s 388us/sample - loss: 0.0757 - accuracy: 0.9727 - val_loss: 0.0616 - val_accuracy: 0.9789
Epoch 2/2
60000/60000 [==============================] - 20s 337us/sample - loss: 0.0303 - accuracy: 0.9899 - val_loss: 0.0364 - val_accuracy: 0.9874
2000/2000 [==============================] - 0s 156us/sample - loss: 0.0385 - accuracy: 0.9865
```





Now let's analyze the results of transfer learning & fine tuning techniques. Since there are only 2 classes **binary cross entropy** used as the loss function. Similarly accuracy is used as the evaluation metric.

When it compares results of Transfer Learning & Fine tuning it can see better performance (When compare True Positives & False Positives in both confusion matrices) in Fine tuning because of retraining. But the training time is almost doubled in Fine tuning.