ENDG 511 - Lab 1: Introduction to Deep Neural Networks using TensorFlow

This colab notebook provides code and a framework for ***Lab 1***. You can work out your solutions here. The example in this lab was inspired by https://www.tensorflow.org/tutorials/quickstart/beginner, with some modifications and additions.

Goals

In this lab, you will be introduced to the baiscs of TensorFlow and Keras, and you will learn how to use them to create deep learning models. The goals of this lab are:

- Understand the basic workflow for creating models in TensorFlow.
- Understand how to train TensorFlow models.
- Evaluate models in terms of accuracy, size and inference time.
- Understand how different model parameters can impact accuracy, size and inference time

Layout

This lab is split into **two** parts.

- **Part 1**: Run through the full example to gain familiarity with TensorFlow. And complete the exercises.
- **Part 2**: Convlutional Neural Netowrks Using TensorFlow. And complete the exercises.

Part 1: Neural Network Machine Learning Model Using TensorFlow

This part of the lab demonstrates building and training a Neural Network to classify MNIST images. This tutorial uses TensorFlow and it is meant to give you a better understanding of how to build, train and evaluate machine learning models using TensorFlow.

Import TensorFlow and Other Required Modules

```
import tensorflow as tf

from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
import time
import tempfile
```

```
WARNING:tensorflow:From c:\Users\johns\Documents\
EngineeringRepositories\ENDG511\lab-env\Lib\site-packages\keras\src\
losses.py:2976: The name tf.losses.sparse_softmax_cross_entropy is
deprecated. Please use
tf.compat.v1.losses.sparse_softmax_cross_entropy instead.
```

Download and prepare the MNIST dataset

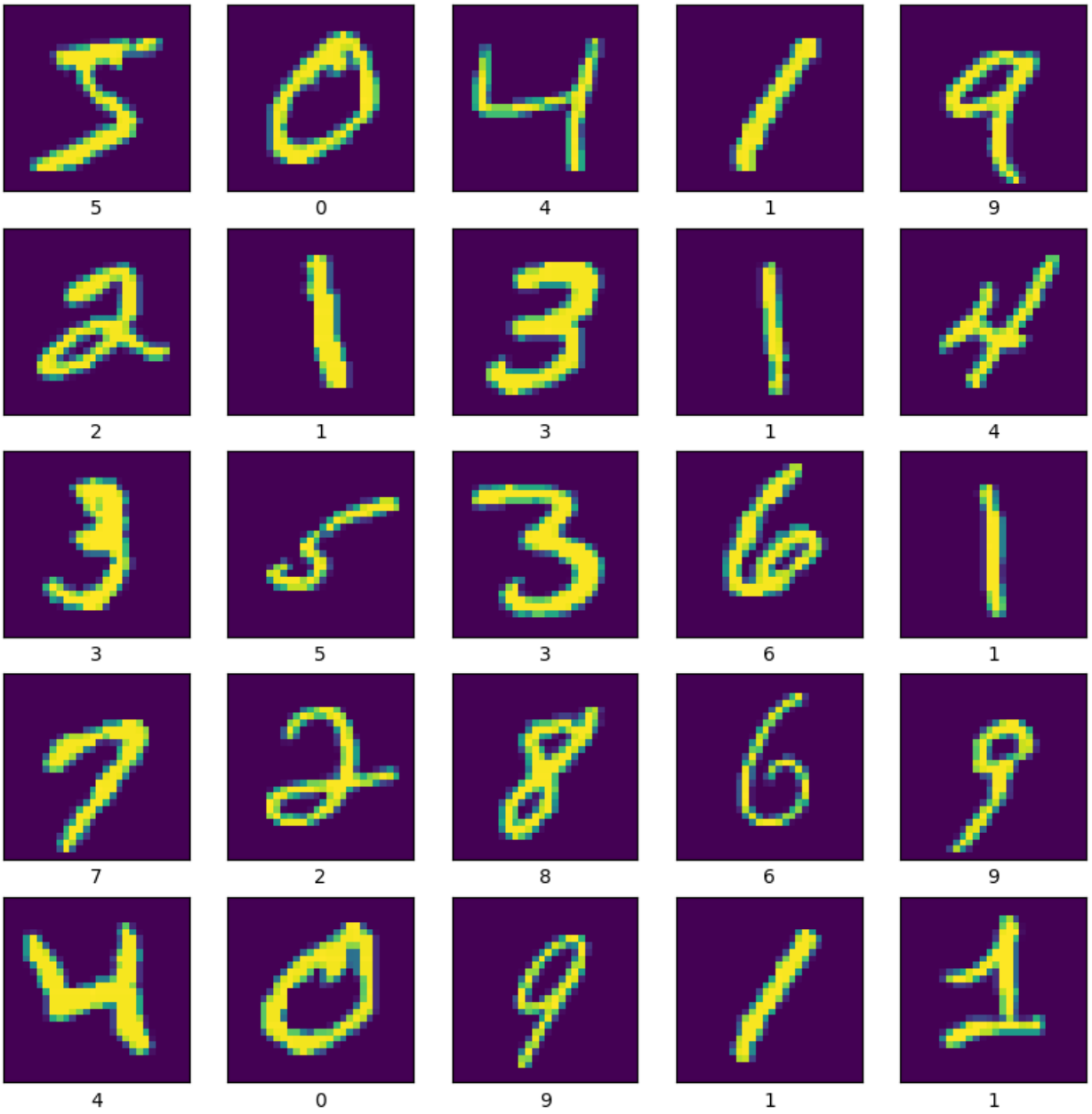
The MNIST dataset is an acronym that stands for the Modified National Institute of Standards and Technology dataset. It is a dataset of 60,000 small square 28×28 pixel grayscale images of handwritten single digits between 0 and 9

```
mnist = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) =
mnist.load_data()

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
```

Verify the data

To verify that the dataset looks correct, let's plot the first 25 images from the training set and display the class name below each image:

```
class_names = ['0', '1', '2', '3', '4',
               '5', '6', '7', '8', '9']

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i])
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```



Build Machine Learning Model

Sequential is useful for stacking layers where each layer has one input tensor and one output tensor. Layers are functions with a known mathematical structure that can be reused and have trainable variables. Most TensorFlow models are composed of layers. This model uses the Flatten, Dense, and Dropout layers.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
```

```
  tf.keras.layers.Dense(10)
])
```

```
WARNING:tensorflow:From c:\Users\johns\Documents\
EngineeringRepositories\ENDG511\lab-env\Lib\site-packages\keras\src\
backend.py:873: The name tf.get_default_graph is deprecated. Please
use tf.compat.v1.get_default_graph instead.
```

Let's display the architecture of our model:

```
model.summary()
```

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

 dense (Dense)               (None, 128)               100480

 dropout (Dropout)           (None, 128)               0

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 101770 (397.54 KB)
Trainable params: 101770 (397.54 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Save and Load Model

Let us save and load the model before training. This allows us to save the randomized weights and biases and make more accurate comparisons. We will use the saved model in the exercise section.

```
# Save your model
model.save('part1_model.h5')

# Recreate the exact same model, including its weights and the
optimizer
new_model = tf.keras.models.load_model('part1_model.h5')

# Show the model architecture
new_model.summary()
```

```
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics
have yet to be built. `model.compile_metrics` will be empty until you
```

```
train or evaluate the model.
WARNING:tensorflow:No training configuration found in the save file,
so the model was *not* compiled. Compile it manually.
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 128)               100480

 dropout (Dropout)           (None, 128)               0

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 101770 (397.54 KB)
Trainable params: 101770 (397.54 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________

c:\Users\johns\Documents\EngineeringRepositories\ENDG511\lab-env\Lib\
site-packages\keras\src\engine\training.py:3103: UserWarning: You are
saving your model as an HDF5 file via `model.save()`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
```

Compile and train the model

When compiling a model, an optimizer, a loss function and metrics need to be defined.

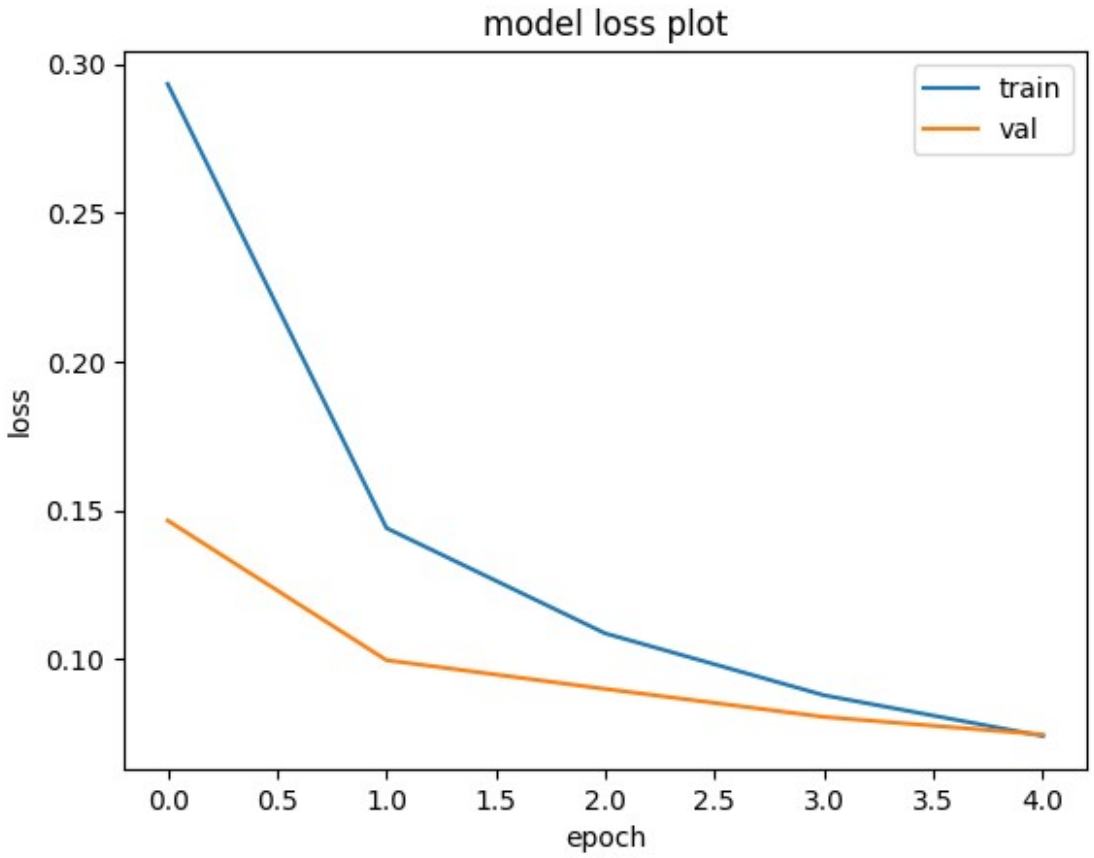
- Optimizer: The Adam optimizer is the most popular optimizer and generally works well for classification and regression problems. It is based of gradient descent algorithms.
- Loss Function: The loss function is the function that computes the distance between the current output of the algorithm and the expected output. These functions need to be chosen depending on the application, the SparseCategoricalCrossentropy for example computes the crossentropy loss between the labels and predictions.
- Metrics: A metric is a function that is used to judge the performance of your model. This could be accuracy, mean squared error, cosine similarity or whatever metric works best for your use case.

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
model.compile(optimizer=optimizer,

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=5,
                    validation_data=(test_images, test_labels))
```

```
Epoch 1/5
WARNING:tensorflow:From c:\Users\johns\Documents\
EngineeringRepositories\ENDG511\lab-env\Lib\site-packages\keras\src\
utils\tf_utils.py:492: The name tf.ragged.RaggedTensorValue is
deprecated. Please use tf.compat.v1.ragged.RaggedTensorValue instead.

WARNING:tensorflow:From c:\Users\johns\Documents\
EngineeringRepositories\ENDG511\lab-env\Lib\site-packages\keras\src\
engine\base_layer_utils.py:384: The name
tf.executing_eagerly_outside_functions is deprecated. Please use
tf.compat.v1.executing_eagerly_outside_functions instead.

1875/1875 [==============================] - 7s 3ms/step - loss:
0.2933 - accuracy: 0.9150 - val_loss: 0.1466 - val_accuracy: 0.9570
Epoch 2/5
1875/1875 [==============================] - 6s 3ms/step - loss:
0.1441 - accuracy: 0.9572 - val_loss: 0.0997 - val_accuracy: 0.9700
Epoch 3/5
1875/1875 [==============================] - 6s 3ms/step - loss:
0.1087 - accuracy: 0.9682 - val_loss: 0.0901 - val_accuracy: 0.9719
Epoch 4/5
1875/1875 [==============================] - 6s 3ms/step - loss:
0.0879 - accuracy: 0.9727 - val_loss: 0.0806 - val_accuracy: 0.9748
Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss:
0.0742 - accuracy: 0.9762 - val_loss: 0.0747 - val_accuracy: 0.9766
```

Plot accuracy and loss graphs

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.title('model accuracy plot')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='best')
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss plot')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='best')
plt.show()
```



Evaluate the model

There are many metrics that can be used to evaluate a model. We will focus on three metrics for this lab:

1. Model Loss and Accuracy
2. Model Size

3. Inference time

```
# Evaluate prediction accuracy
test_loss, test_acc = model.evaluate(test_images,  test_labels,
verbose=2)


# Evaluate Model Size
def get_gzipped_model_size(file):
  # Returns size of gzipped model, in bytes.
  import os
  import zipfile

  _, zipped_file = tempfile.mkstemp('.zip')
  with zipfile.ZipFile(zipped_file, 'w',
compression=zipfile.ZIP_DEFLATED) as f:
    f.write(file)

  return os.path.getsize(zipped_file)


# Evaluate Inference Time
startTime = time.time()
prediction = model.predict(test_images)
executionTime = (time.time() - startTime)/len(test_images)

# Print
print('\nModel Accuracy:', test_acc*100, '%')
print("Model Size: %.2f bytes" %
(get_gzipped_model_size('part1_model.h5')))
print("Inference Time is", executionTime, "s")
```

```
313/313 - 1s - loss: 0.0747 - accuracy: 0.9766 - 579ms/epoch -
2ms/step
313/313 [==============================] - 1s 2ms/step

Model Accuracy: 97.65999913215637 %
Model Size: 374881.00 bytes
Inference Time is 8.716819286346435e-05 s
```

Excercies (2.5 points)

Question 1:

Load the untrained model. Change the learning rate to 1.5, recompile and retrain the model and display the accuracy and loss plots. What do you observe? **(0.5 point)**

Loading the untrained model

```
# Recreate the exact same model, including its weights and the
optimizer
```

```
new_model = tf.keras.models.load_model('part1_model.h5')

# Show the model architecture
new_model.summary()
```

```
WARNING:tensorflow:No training configuration found in the save file,
so the model was *not* compiled. Compile it manually.
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 128)               100480

 dropout (Dropout)           (None, 128)               0

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 101770 (397.54 KB)
Trainable params: 101770 (397.54 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Changing the learning rate to 1.5

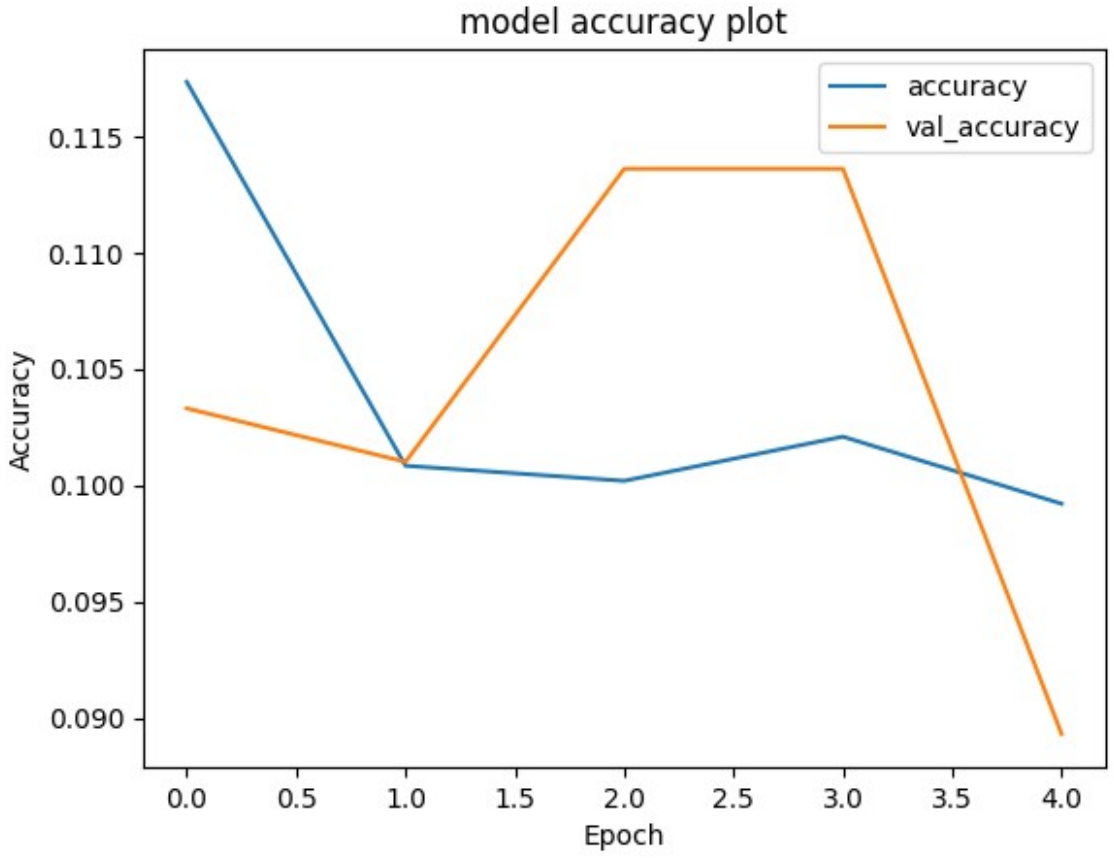
```
optimizer = tf.keras.optimizers.Adam(learning_rate=1.5)
new_model.compile(optimizer=optimizer,

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = new_model.fit(train_images, train_labels, epochs=5,
                    validation_data=(test_images, test_labels))
```

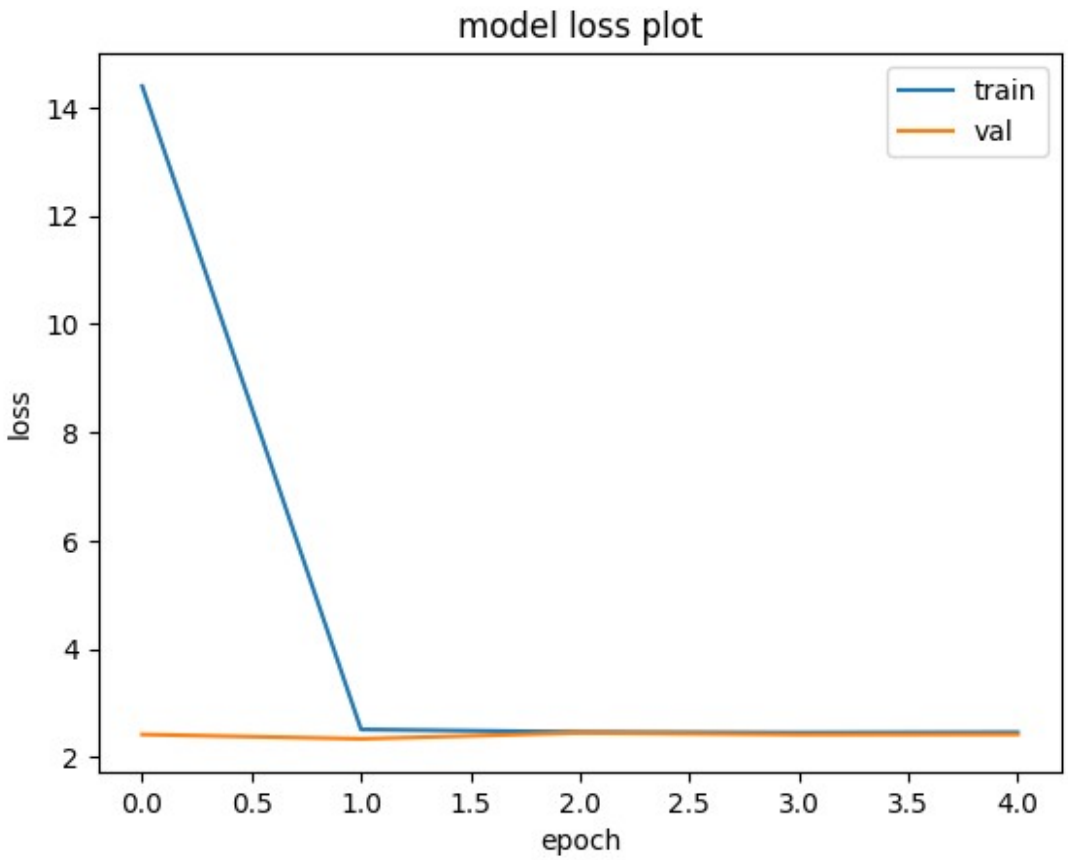
```
Epoch 1/5
1875/1875 [==============================] - 7s 3ms/step - loss:
14.4011 - accuracy: 0.173 - val_loss: 2.4157 - val_accuracy: 0.1033
Epoch 2/5
1875/1875 [==============================] - 6s 3ms/step - loss:
2.5078 - accuracy: 0.1008 - val_loss: 2.3349 - val_accuracy: 0.1010
Epoch 3/5
1875/1875 [==============================] - 5s 3ms/step - loss:
2.4629 - accuracy: 0.1002 - val_loss: 2.4454 - val_accuracy: 0.1136
Epoch 4/5
1875/1875 [==============================] - 6s 3ms/step - loss:
2.4497 - accuracy: 0.1021 - val_loss: 2.4149 - val_accuracy: 0.1136
Epoch 5/5
```

```
1875/1875 [==============================] - 6s 3ms/step - loss:
2.4566 - accuracy: 0.0992 - val_loss: 2.4173 - val_accuracy: 0.0893
```

Generating the plots

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.title('model accuracy plot')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='best')
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss plot')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='best')
plt.show()
```





Observations

With a learning rate of 1.5 it can be observed that the model has a low accuracy towards the last epoch. A learning rate of 0.001 yeilds a training accuracy in the margin ~0.91 to ~0.98 and a validation accuracy ~0.955 to ~0.98 with general increasing trend towards the last epoch. However, a learning rate of 1.5 yeilds a training accuracy with a decreasing trend in the margin ~0.120 to ~0.100 and a validation accuracy in the margin ~0.115 to 0.090 with fluctuations throughout the epochs.

With a learning rate of 0.001 it can be observed that the model training loss has a general decreasing trend within the margins of ~0.30 to ~0.09 and the model validation loss within the margins of ~0.15 to ~0.09. However, with a learning rate of 1.5, it can be observed that the validation loss is somewhat static at ~2.5 whereas the training loss decreases from ~14 to ~2.5

Question 2:

Load the untrained model. Change the learning rate to 0.0001, recompile and retrain the model and display the accuracy and loss plots. What do you observe? **(0.5 point)**

Loading the untrained model

```
# Recreate the exact same model, including its weights and the
optimizer
```

```
new_model = tf.keras.models.load_model('part1_model.h5')

# Show the model architecture
new_model.summary()
```

```
WARNING:tensorflow:No training configuration found in the save file,
so the model was *not* compiled. Compile it manually.
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 128)               100480

 dropout (Dropout)           (None, 128)               0

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 101770 (397.54 KB)
Trainable params: 101770 (397.54 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Changing the learning rate to 0.0001

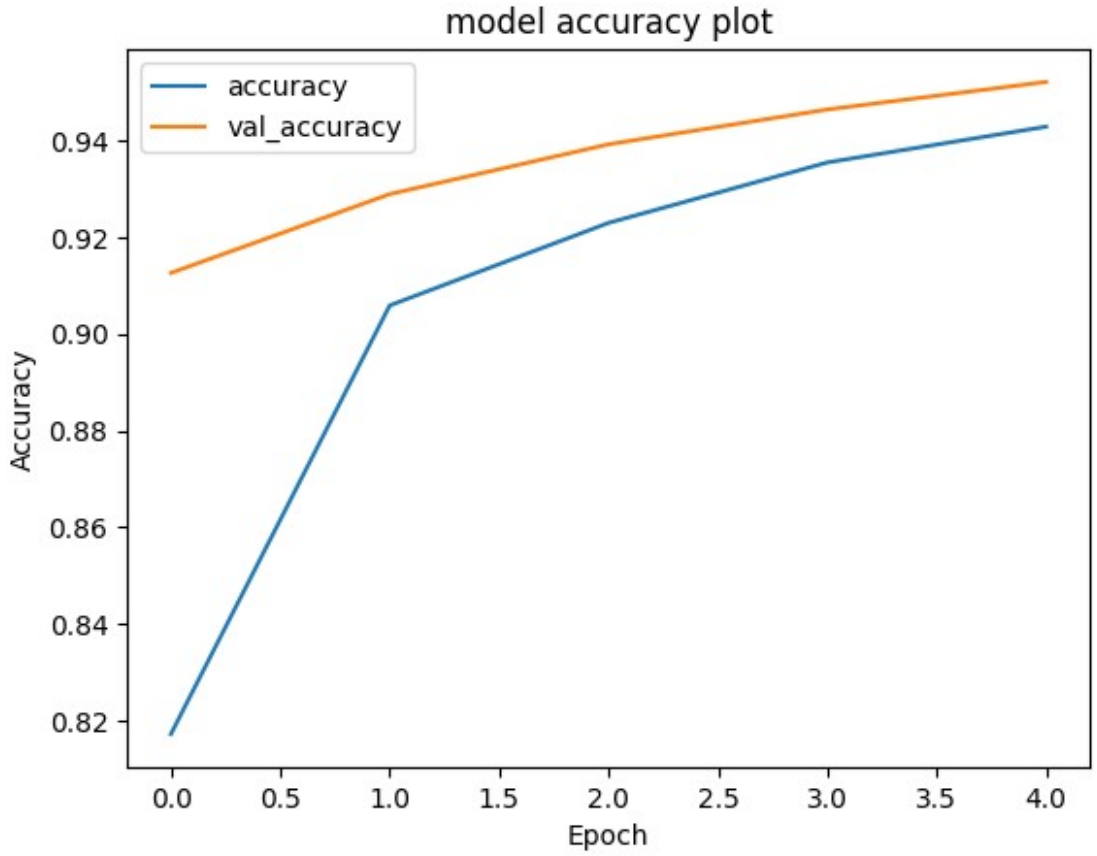
```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001)
new_model.compile(optimizer=optimizer,

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = new_model.fit(train_images, train_labels, epochs=5,
                    validation_data=(test_images, test_labels))
```

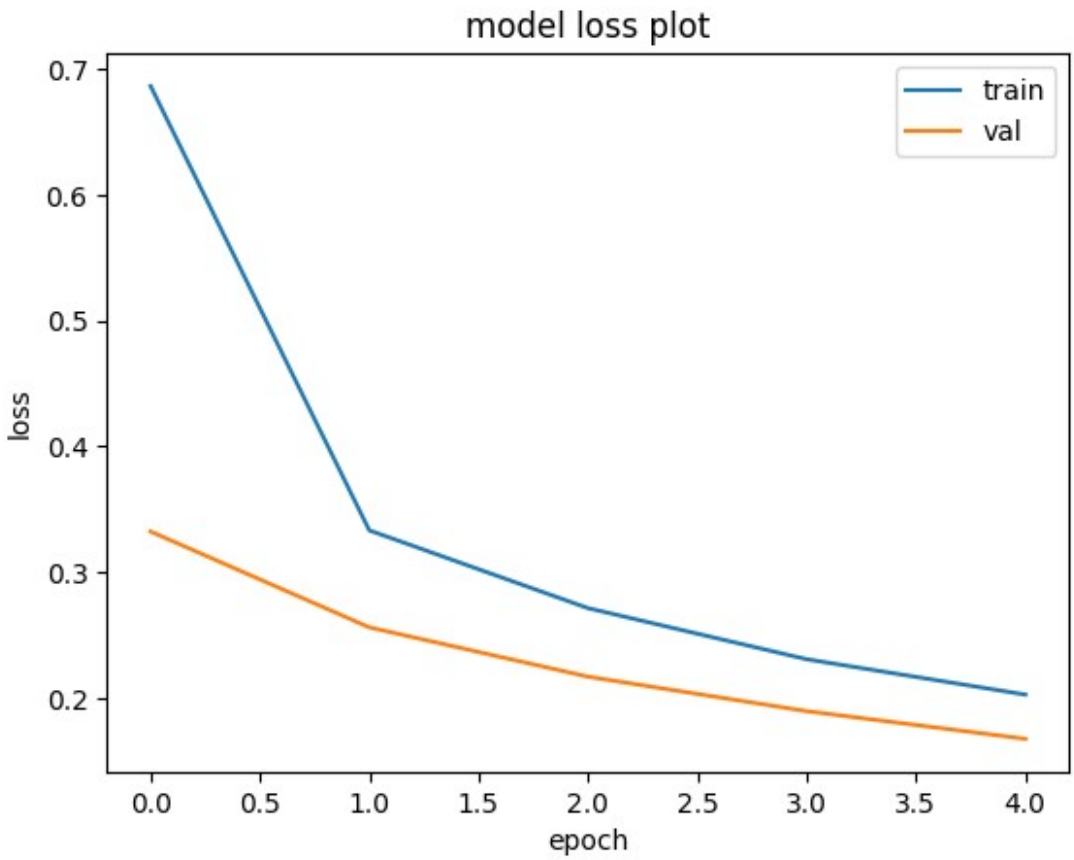
```
Epoch 1/5
1875/1875 [==============================] - 7s 3ms/step - loss:
0.6866 - accuracy: 0.8173 - val_loss: 0.3323 - val_accuracy: 0.9126
Epoch 2/5
1875/1875 [==============================] - 6s 3ms/step - loss:
0.3332 - accuracy: 0.9059 - val_loss: 0.2561 - val_accuracy: 0.9289
Epoch 3/5
1875/1875 [==============================] - 6s 3ms/step - loss:
0.2713 - accuracy: 0.9230 - val_loss: 0.2168 - val_accuracy: 0.9392
Epoch 4/5
1875/1875 [==============================] - 6s 3ms/step - loss:
0.2307 - accuracy: 0.9355 - val_loss: 0.1894 - val_accuracy: 0.9464
Epoch 5/5
```

```
1875/1875 [==============================] - 6s 3ms/step - loss:
0.2026 - accuracy: 0.9428 - val_loss: 0.1674 - val_accuracy: 0.9521
```

Generating the plots

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.title('model accuracy plot')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='best')
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss plot')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='best')
plt.show()
```





Observations

With a learning rate of 0.0001, it can be observed that the model validation accuracy has an increasing trend from ~0.91 to ~0.95 and a model training accuracy with in increasing trend from ~0.81 to ~0.93 throughout the epochs. In both cases, the accuracy for this learning rate is still lower than the accuracy with a learning rate of 0.001 which yeilded an accuracy in the final epoch ~0.97-0.98. A hypothesis to this result is that this learning rate does not provide too much changes in the model weights to reach the optimal weights that allow correct classifications of the images.

There is a general decreasing trend in the loss for both training and validation. Training loss starts high in the first epochs at ~0.68 and ends with a loss of ~0.22 at the last epoch. The validation loss starts at ~0.33 and ends with a loss of ~0.10 at the last epoch.

However, the loss of the 0.001 learning rate used prior was below 0.10 at the last epoch. Perhaps if the number of epochs is increased, this general decrease in the loss could continue to match the loss observed with a learning rate that is 10x larger.

Question 3:

Rebuild the model with 8 neurons in the first Dense layer instead of 128. Then recompile and retrain the model and save it. Finally compare the new size, accuracy and inference time to the original model. **(0.75 point)**

Rebuilding the model with 8 neurons

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(8, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
])
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_1 (Flatten)         (None, 784)               0

 dense_2 (Dense)             (None, 8)                 6280

 dropout_1 (Dropout)         (None, 8)                 0

 dense_3 (Dense)             (None, 10)                90

=================================================================
Total params: 6370 (24.88 KB)
Trainable params: 6370 (24.88 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Recompiling, Retraining, Saving

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
model.compile(optimizer=optimizer,

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=5,
                    validation_data=(test_images, test_labels))

# Save your model
model.save('part1_model_8_neurons.h5')
```

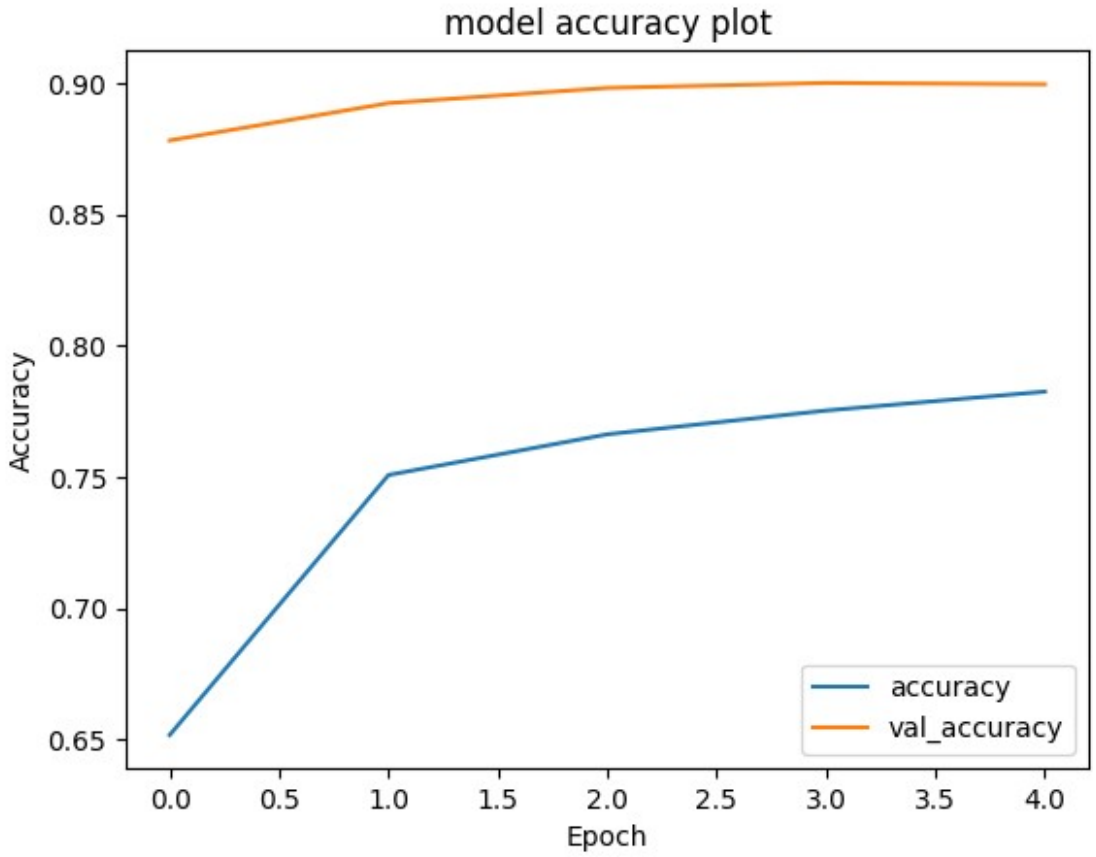
```
Epoch 1/5
1875/1875 [==============================] - 5s 2ms/step - loss:
1.0289 - accuracy: 0.6518 - val_loss: 0.4693 - val_accuracy: 0.8782
Epoch 2/5
1875/1875 [==============================] - 4s 2ms/step - loss:
0.7454 - accuracy: 0.7508 - val_loss: 0.4219 - val_accuracy: 0.8924
Epoch 3/5
1875/1875 [==============================] - 4s 2ms/step - loss:
0.6850 - accuracy: 0.7663 - val_loss: 0.3814 - val_accuracy: 0.8982
```

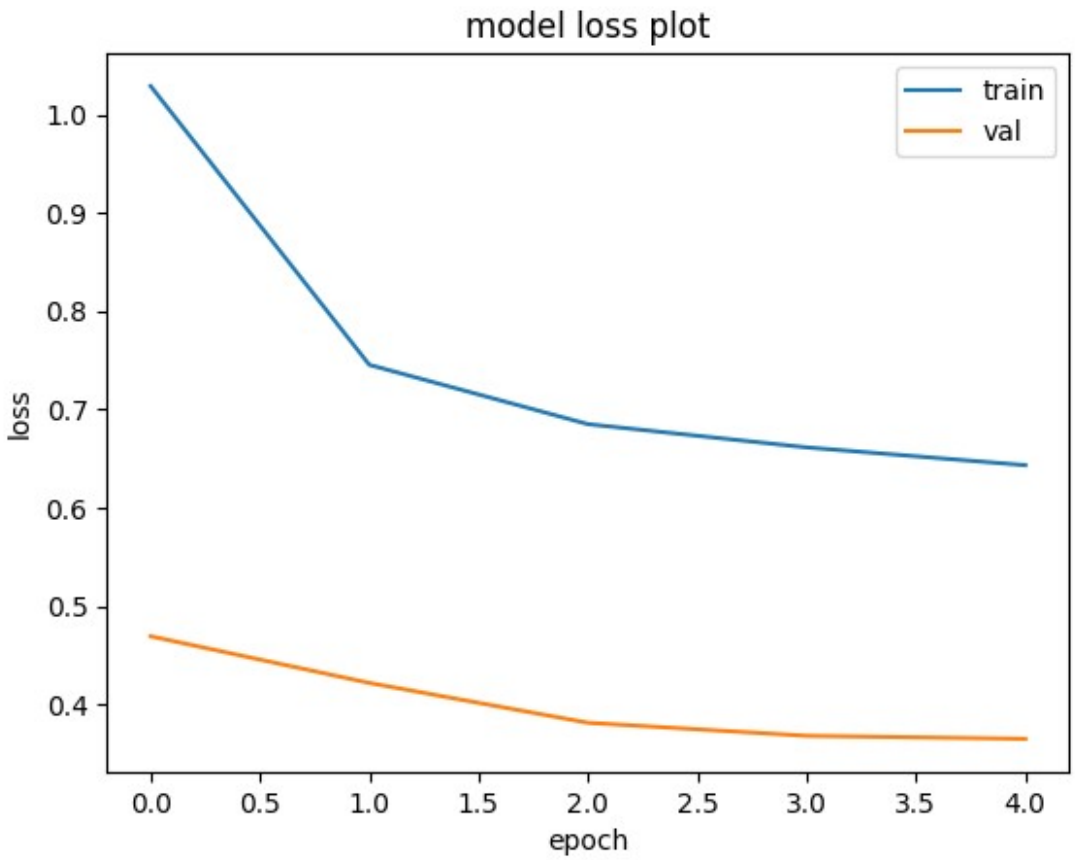
```
Epoch 4/5
1875/1875 [==============================] - 4s 2ms/step - loss:
0.6614 - accuracy: 0.7753 - val_loss: 0.3681 - val_accuracy: 0.9001
Epoch 5/5
1875/1875 [==============================] - 4s 2ms/step - loss:
0.6433 - accuracy: 0.7825 - val_loss: 0.3650 - val_accuracy: 0.8996
```

Model Evaluation

```
# Recreate the exact same model, including its weights and the
optimizer
new_model = tf.keras.models.load_model('part1_model_8_neurons.h5')

plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.title('model accuracy plot')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='best')
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss plot')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='best')
plt.show()

# Evaluate prediction accuracy
test_loss, test_acc = new_model.evaluate(test_images,  test_labels,
verbose=2)


# Evaluate Model Size
def get_gzipped_model_size(file):
  # Returns size of gzipped model, in bytes.
  import os
  import zipfile

  _, zipped_file = tempfile.mkstemp('.zip')
  with zipfile.ZipFile(zipped_file, 'w',
compression=zipfile.ZIP_DEFLATED) as f:
    f.write(file)

  return os.path.getsize(zipped_file)


# Evaluate Inference Time
startTime = time.time()
```

```
prediction = new_model.predict(test_images)
executionTime = (time.time() - startTime)/len(test_images)

# Print
print('\nModel Accuracy:', test_acc*100, '%')
print("Model Size: %.2f bytes" %
(get_gzipped_model_size('part1_model_8_neurons.h5')))
print("Inference Time is", executionTime, "s")
```






```
313/313 - 1s - loss: 0.3650 - accuracy: 0.8996 - 695ms/epoch -
2ms/step
313/313 [==============================] - 1s 1ms/step

Model Accuracy: 89.96000289916992 %
Model Size: 71460.00 bytes
Inference Time is 7.732150554656983e-05 s
```

Observations

The original model had a model accuracy ~97.66%, a model size around 374.88KB, and an inference time ~8.72 seconds. This model trained with 8 neurons, has a model accuracy ~89.96% an ~8% drop in accuracy. However, it is 71.46KB which is 5x smaller and an inference time 77.32us seconds which is 112,000x faster.

Question 4:

Rebuild the model with 2048 neurons in the first Dense layer instead of 128. Then recompile and retrain the model and save it. Finally compare the new size, accuracy and inference time to the original model. **(0.75 point)**

Rebuilding the model with 2048 neurons.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(2048, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10)
])
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_2 (Flatten)         (None, 784)               0

 dense_4 (Dense)             (None, 2048)              1607680

 dropout_2 (Dropout)         (None, 2048)              0

 dense_5 (Dense)             (None, 10)                20490

=================================================================
Total params: 1628170 (6.21 MB)
Trainable params: 1628170 (6.21 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Recompiling, Retraining, Saving

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
model.compile(optimizer=optimizer,

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=5,
                    validation_data=(test_images, test_labels))

# Save your model
model.save('part1_model_2048_neurons.h5')
```

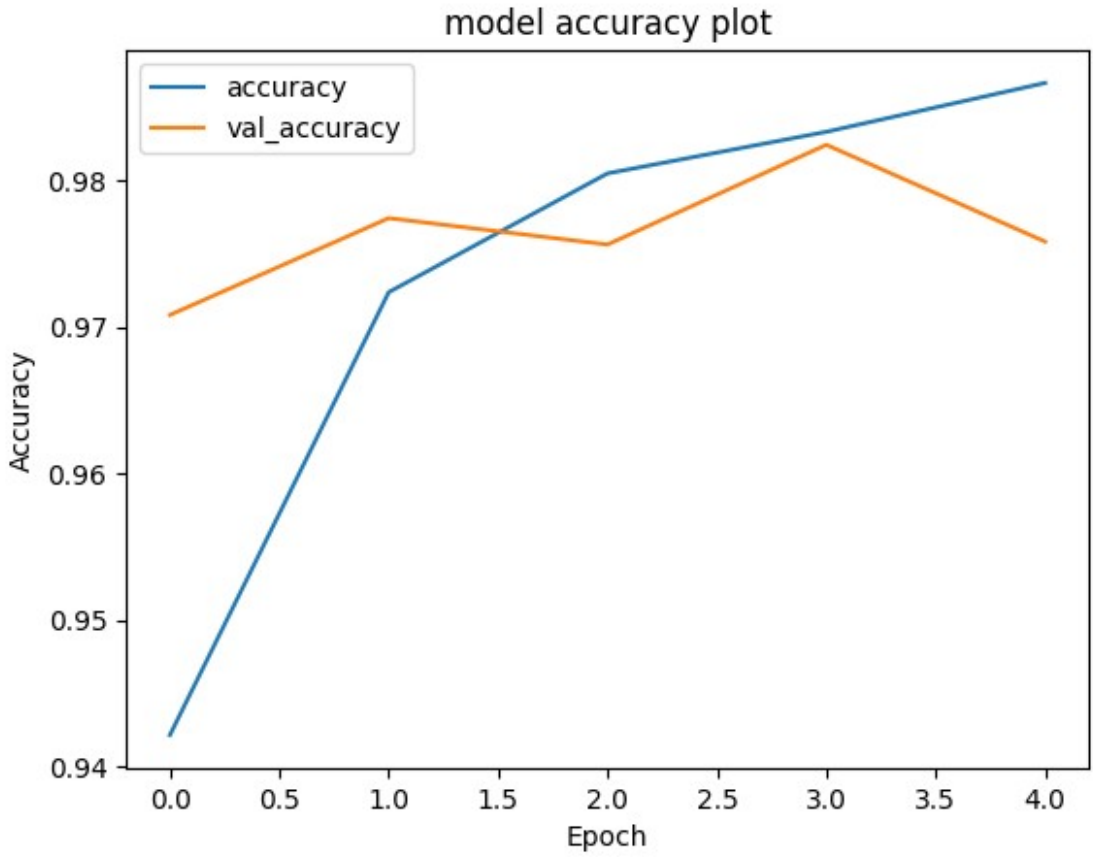
```
Epoch 1/5
1875/1875 [==============================] - 49s 26ms/step - loss:
0.1900 - accuracy: 0.9422 - val_loss: 0.0919 - val_accuracy: 0.9708
Epoch 2/5
1875/1875 [==============================] - 44s 24ms/step - loss:
0.0871 - accuracy: 0.9724 - val_loss: 0.0769 - val_accuracy: 0.9774
Epoch 3/5
1875/1875 [==============================] - 49s 26ms/step - loss:
0.0611 - accuracy: 0.9804 - val_loss: 0.0812 - val_accuracy: 0.9756
```

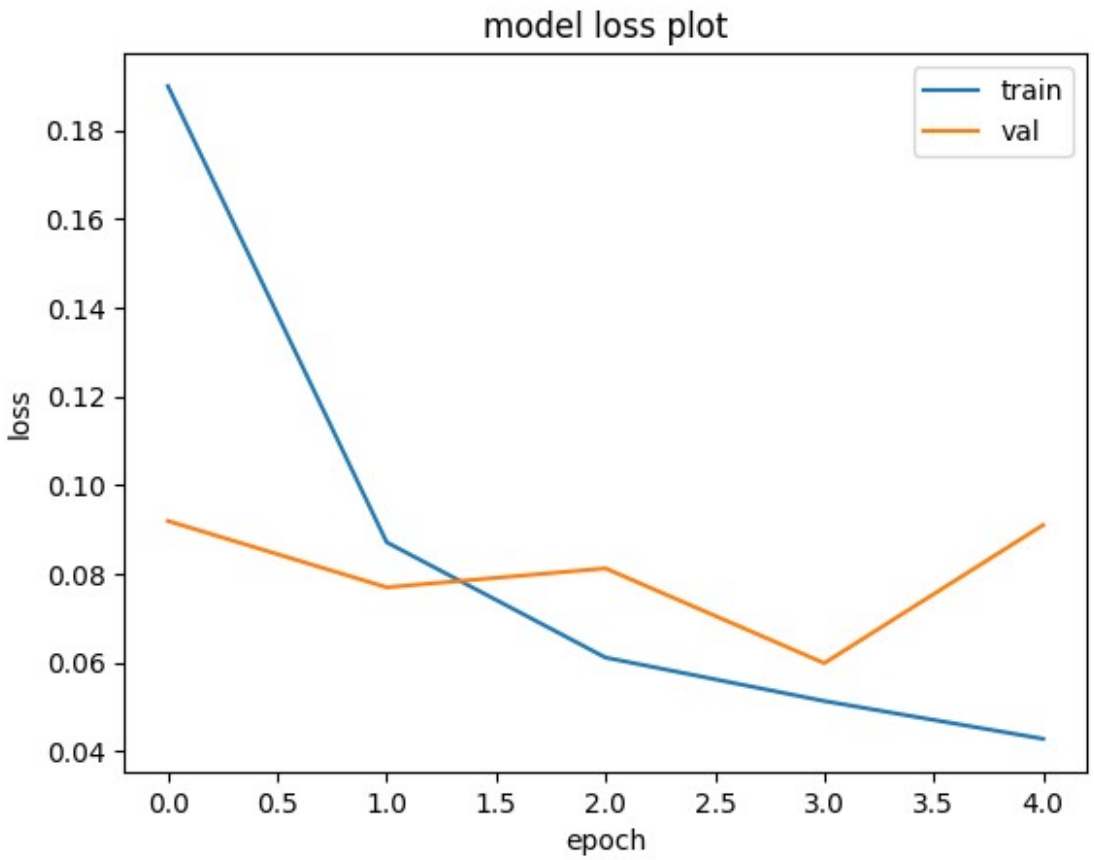
```
Epoch 4/5
1875/1875 [==============================] - 49s 26ms/step - loss:
0.0513 - accuracy: 0.9833 - val_loss: 0.0598 - val_accuracy: 0.9824
Epoch 5/5
1875/1875 [==============================] - 48s 25ms/step - loss:
0.0428 - accuracy: 0.9866 - val_loss: 0.0910 - val_accuracy: 0.9758
```

Model Evaluation

```
# Recreate the exact same model, including its weights and the
optimizer
new_model = tf.keras.models.load_model('part1_model_2048_neurons.h5')

plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.title('model accuracy plot')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='best')
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss plot')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='best')
plt.show()

# Evaluate prediction accuracy
test_loss, test_acc = new_model.evaluate(test_images,  test_labels,
verbose=2)


# Evaluate Model Size
def get_gzipped_model_size(file):
  # Returns size of gzipped model, in bytes.
  import os
  import zipfile

  _, zipped_file = tempfile.mkstemp('.zip')
  with zipfile.ZipFile(zipped_file, 'w',
compression=zipfile.ZIP_DEFLATED) as f:
    f.write(file)

  return os.path.getsize(zipped_file)


# Evaluate Inference Time
startTime = time.time()
```

```
prediction = new_model.predict(test_images)
executionTime = (time.time() - startTime)/len(test_images)

# Print
print('\nModel Accuracy:', test_acc*100, '%')
print("Model Size: %.2f bytes" %
(get_gzipped_model_size('part1_model_2048_neurons.h5')))
print("Inference Time is", executionTime, "s")
```






```
313/313 - 1s - loss: 0.0910 - accuracy: 0.9758 - 1s/epoch - 4ms/step
313/313 [==============================] - 1s 3ms/step

Model Accuracy: 97.57999777793884 %
Model Size: 16938440.00 bytes
Inference Time is 0.00013290605545043945 s
```

Observations

The original model with 128 neurons had an accuracy of ~97.66%, a size of ~374.88KB, and an inference time of ~8.72 seconds. The new model built with 2048 neurons has a ~0.1% decrease in accuracy at ~97.58%. However more neurons means a larger model and there was 45x increase in model size with 2048 neurons at ~16.94MB. Furthermore, the new model is faster running at 0.13ms which is ~62000 times faster.

Part 2: Convolutional Neural Networks Using TensorFlow

Introduction:

Convolutional Neural Networks (CNN) are distinguished by their ability to identify patterns and features in images (e.g. edges, lines, circles or even objects). CNN's are widely adopted in computer vision applications and found success in audio signals and text.

A typical Convolutional neural network is composed mainly of three types of layers:

- **Convolutional layer:** The core building block of a CNN, which consists of Filters or kernels to detect features. Generally, the first convolutional layers detect simple features (e.g. edges) and later layers identify objects.
- **Pooling layer:** Reduces the spatial size of the feature maps so having fewer learnable weights and less computation cost.
- **Fully-connected layer:** Known as the output layer, where the last identified feature maps are flattened and connected to dense layers for the classification task.

Import TensorFlow and Other Required Modules

```
import tensorflow as tf

from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
import time
import tempfile
```

Download and prepare the MNIST dataset

Similar to Part 1, loading the MNIST dataset and performing normalization to improve the training performance and speed.

```
mnist = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) =
mnist.load_data()

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
```

Build Machine Learning Model

In this part, we will build the LeNet-5 CNN model, one of the earliest CNN architectures proposed by Yann LeCun and others for recognizing handwritten and machine-printed characters.

The LeNet-5 architecture cosisits of 2 convolutional layers, 2 pooling layers and fully connected layers as shown in the figure below.

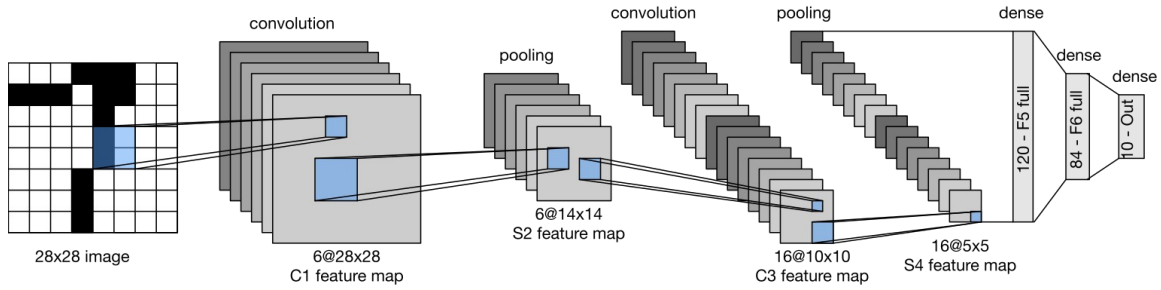


Image Source

```
lenet_5_model = models.Sequential()
# Convolutional layer1: cosits of 6 filters, filter size 5x5 and
stride of 1
lenet_5_model.add(layers.Conv2D(filters = 6, kernel_size =
(5,5),strides=1, padding = 'same', activation = 'relu', input_shape =
(28,28,1)))
#Pooling layer 1
lenet_5_model.add(layers.AveragePooling2D(pool_size = (2,2)))
# Convolutional layer2: cosits of 16 filters, filter size 5x5 and
stride of 1
lenet_5_model.add(layers.Conv2D(filters = 16, kernel_size =
5,strides=1, activation = 'relu'))
#Pooling layer 2
lenet_5_model.add(layers.AveragePooling2D(pool_size = (2,2)))

#Output layer( Fully connected layers)
lenet_5_model.add(layers.Flatten())
lenet_5_model.add(layers.Dense(120, activation='relu'))
lenet_5_model.add(layers.Dense(84, activation='relu'))
lenet_5_model.add(layers.Dense(10, activation='softmax'))

# Save your model
lenet_5_model.save('part2_model.h5')

lenet_5_model.summary()
```

```
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics
have yet to be built. `model.compile_metrics` will be empty until you
train or evaluate the model.
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 6)         156
```

```
 average_pooling2d (Average  (None, 14, 14, 6)         0
 Pooling2D)

 conv2d_1 (Conv2D)           (None, 10, 10, 16)        2416

 average_pooling2d_1 (Avera  (None, 5, 5, 16)          0
 gePooling2D)

 flatten_3 (Flatten)         (None, 400)               0

 dense_6 (Dense)             (None, 120)               48120

 dense_7 (Dense)             (None, 84)                10164

 dense_8 (Dense)             (None, 10)                850

=================================================================
Total params: 61706 (241.04 KB)
Trainable params: 61706 (241.04 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________

c:\Users\johns\Documents\EngineeringRepositories\ENDG511\lab-env\Lib\
site-packages\keras\src\engine\training.py:3103: UserWarning: You are
saving your model as an HDF5 file via `model.save()`. This file format
is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
```

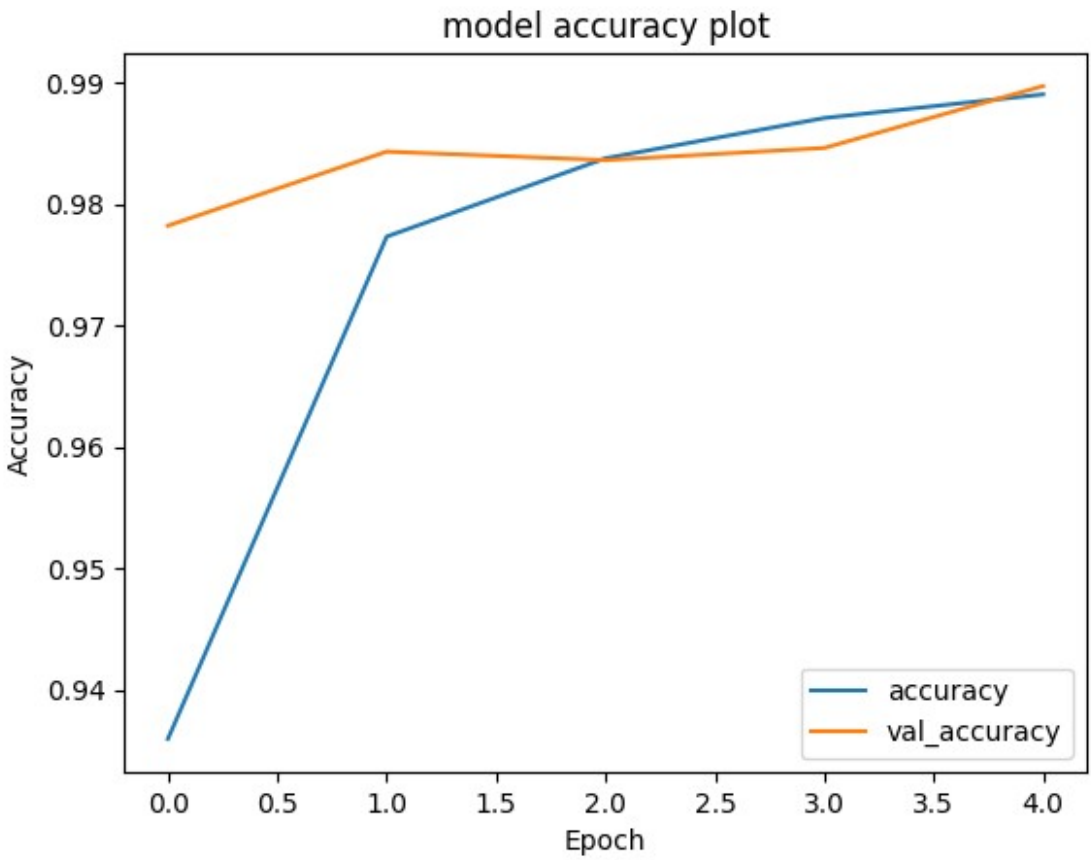
Compile and train the model

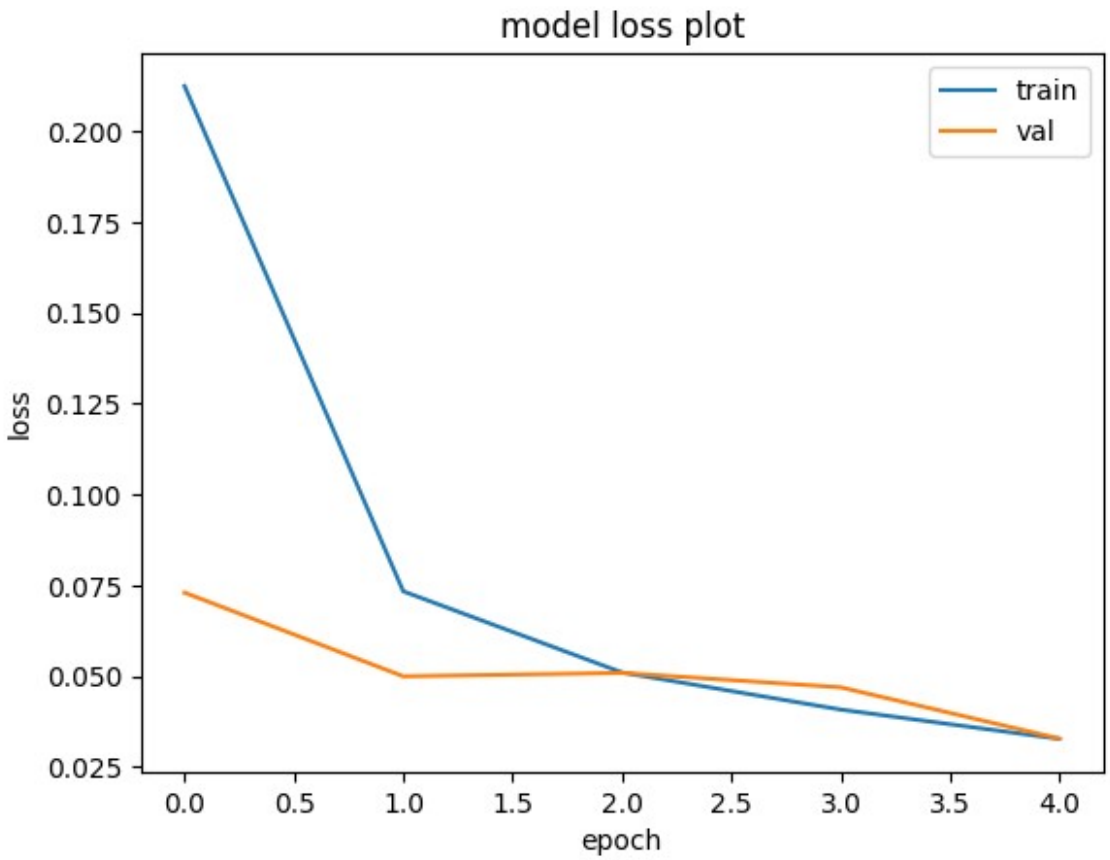
```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
lenet_5_model.compile(optimizer=optimizer,

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = lenet_5_model.fit(train_images, train_labels, epochs=5,
                    validation_data=(test_images, test_labels))

Epoch 1/5

c:\Users\johns\Documents\EngineeringRepositories\ENDG511\lab-env\Lib\
site-packages\keras\src\backend.py:5727: UserWarning:
"`sparse_categorical_crossentropy` received `from_logits=True`, but
the `output` argument was produced by a Softmax activation and thus
does not represent logits. Was this intended?
  output, from_logits = _get_logits(
```

```
1875/1875 [==============================] - 12s 6ms/step - loss:
0.2124 - accuracy: 0.9359 - val_loss: 0.0730 - val_accuracy: 0.9782
Epoch 2/5
1875/1875 [==============================] - 11s 6ms/step - loss:
0.0734 - accuracy: 0.9773 - val_loss: 0.0500 - val_accuracy: 0.9843
Epoch 3/5
1875/1875 [==============================] - 11s 6ms/step - loss:
0.0511 - accuracy: 0.9837 - val_loss: 0.0509 - val_accuracy: 0.9836
Epoch 4/5
1875/1875 [==============================] - 10s 6ms/step - loss:
0.0408 - accuracy: 0.9871 - val_loss: 0.0470 - val_accuracy: 0.9846
Epoch 5/5
1875/1875 [==============================] - 11s 6ms/step - loss:
0.0328 - accuracy: 0.9890 - val_loss: 0.0329 - val_accuracy: 0.9897
```

Plot accuracy and loss graphs

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.title('model accuracy plot')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='best')
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss plot')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='best')
plt.show()
```


model accuracy plot
0.99
0.98
0.97
0.96
0.95
0.94
Accuracy
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Epoch
accuracy
val_accuracy



Evaluate the model

Applying the same evaluation metrics in part 1 to evaluate the trained CNN model:

1. Model Loss and Accuracy
2. Model Size
3. Inference time

```
# Evaluate prediction accuracy
test_loss, test_acc = lenet_5_model.evaluate(test_images,
test_labels, verbose=2)


# Evaluate Model Size
def get_gzipped_model_size(file):
  # Returns size of gzipped model, in bytes.
  import os
  import zipfile

  _, zipped_file = tempfile.mkstemp('.zip')
  with zipfile.ZipFile(zipped_file, 'w',
compression=zipfile.ZIP_DEFLATED) as f:
    f.write(file)
```

```
  return os.path.getsize(zipped_file)


# Evaluate Inference Time
startTime = time.time()
prediction = lenet_5_model.predict(test_images)
executionTime = (time.time() - startTime)/len(test_images)

# Print
print('\nModel Accuracy:', test_acc*100, '%')
print("Model Size: %.2f bytes" %
(get_gzipped_model_size('part2_model.h5')))
print("Inference Time is", executionTime, "s")
```

```
313/313 - 1s - loss: 0.0329 - accuracy: 0.9897 - 885ms/epoch -
3ms/step
313/313 [==============================] - 1s 3ms/step

Model Accuracy: 98.97000193595886 %
Model Size: 228337.00 bytes
Inference Time is 0.00011248953342437744 s
```

Excercises: (2.5 points)

Question 1:

Briefly compare the Fully connected (FC) model in part 1 and the CNN model in part 2 in terms of the model size, accuracy and inference time.**(0.5 point)**

The accuracy in the original FC model with 128 neurons was ~97.66%, a size of ~374.88KB, and an inference time of ~8.72 seconds. With the CNN model, the accuracy is ~98.97% which is ~1.3% better than the FC model. Furthermore, the CNN model is ~0.6x smaller than the FC model and substantially ~77850 times faster than the FC model at 0.11ms.

Question 2:

Modify the two convolutional layers of the LeNet-5 stride value into 2 and discuss the effect on the model size and accuracy. If the performance is affected, explain the reason? **(0.5 point)**

Building the Model

```
lenet_5_model = models.Sequential()
# Convolutional layer1: cosits of 6 filters, filter size 5x5 and
stride of 1
lenet_5_model.add(layers.Conv2D(filters = 6, kernel_size =
(5,5),strides=2, padding = 'same', activation = 'relu', input_shape =
(28,28,1)))
#Pooling layer 1
lenet_5_model.add(layers.AveragePooling2D(pool_size = (2,2)))
```

```
# Convolutional layer2: cosits of 16 filters, filter size 5x5 and
stride of 1
lenet_5_model.add(layers.Conv2D(filters = 16, kernel_size =
5,strides=2, activation = 'relu'))
#Pooling layer 2
lenet_5_model.add(layers.AveragePooling2D(pool_size = (2,2)))

#Output layer( Fully connected layers)
lenet_5_model.add(layers.Flatten())
lenet_5_model.add(layers.Dense(120, activation='relu'))
lenet_5_model.add(layers.Dense(84, activation='relu'))
lenet_5_model.add(layers.Dense(10, activation='softmax'))
lenet_5_model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)           (None, 14, 14, 6)         156

 average_pooling2d_2 (Avera  (None, 7, 7, 6)           0
 gePooling2D)

 conv2d_3 (Conv2D)           (None, 2, 2, 16)          2416

 average_pooling2d_3 (Avera  (None, 1, 1, 16)          0
 gePooling2D)

 flatten_4 (Flatten)         (None, 16)                0

 dense_9 (Dense)             (None, 120)               2040

 dense_10 (Dense)            (None, 84)                10164

 dense_11 (Dense)            (None, 10)                850

=================================================================
Total params: 15626 (61.04 KB)
Trainable params: 15626 (61.04 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Training the model

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
lenet_5_model.compile(optimizer=optimizer,

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

```
history = lenet_5_model.fit(train_images, train_labels, epochs=5,
                    validation_data=(test_images, test_labels))
```

```
Epoch 1/5
1875/1875 [==============================] - 9s 4ms/step - loss:
0.6153 - accuracy: 0.8012 - val_loss: 0.2724 - val_accuracy: 0.9174
Epoch 2/5
1875/1875 [==============================] - 8s 4ms/step - loss:
0.2479 - accuracy: 0.9247 - val_loss: 0.2003 - val_accuracy: 0.9379
Epoch 3/5
1875/1875 [==============================] - 7s 4ms/step - loss:
0.1809 - accuracy: 0.9440 - val_loss: 0.1545 - val_accuracy: 0.9514
Epoch 4/5
1875/1875 [==============================] - 7s 4ms/step - loss:
0.1513 - accuracy: 0.9526 - val_loss: 0.1293 - val_accuracy: 0.9583
Epoch 5/5
1875/1875 [==============================] - 8s 4ms/step - loss:
0.1294 - accuracy: 0.9596 - val_loss: 0.1061 - val_accuracy: 0.9673
```

Evaluating the model

```
# Evaluate prediction accuracy
test_loss, test_acc = lenet_5_model.evaluate(test_images,
test_labels, verbose=2)


# Evaluate Model Size
def get_gzipped_model_size(file):
  # Returns size of gzipped model, in bytes.
  import os
  import zipfile

  _, zipped_file = tempfile.mkstemp('.zip')
  with zipfile.ZipFile(zipped_file, 'w',
compression=zipfile.ZIP_DEFLATED) as f:
    f.write(file)

  return os.path.getsize(zipped_file)


# Evaluate Inference Time
startTime = time.time()
prediction = lenet_5_model.predict(test_images)
executionTime = (time.time() - startTime)/len(test_images)

# Print
print('\nModel Accuracy:', test_acc*100, '%')
print("Model Size: %.2f bytes" %
```

```
(get_gzipped_model_size('part2_model.h5')))
print("Inference Time is", executionTime, "s")
```

```
313/313 - 1s - loss: 0.1061 - accuracy: 0.9673 - 632ms/epoch -
2ms/step
313/313 [==============================] - 1s 3ms/step

Model Accuracy: 96.72999978065491 %
Model Size: 228337.00 bytes
Inference Time is 0.00013354616165161133 s
```

Observation

Changing the stride to 2 in both convolutional layers does not affect the model size as shown. However, the number of parameters is 4x smaller at 15626 in comparison to the model with a stride of 1 at 61706. Stride also affects model accuracy because it controls the spaces the filter moves across the image. A high stride would mean skipped spaces that may result in a lower accuracy because information is lost as a result. This can be shown here that with a stride of 1, the model accuracy is 98.97%. However, with a stride of 2 there is a ~2% drop in accuracy at ~96.73%.

Question 3:

Replace the average-pooling layer with a max-pooling layer for the LeNet-5 CNN model and discuss the effect on the trained model size and accuracy.**(0.5 point)**

Building the model

```
lenet_5_model = models.Sequential()
# Convolutional layer1: cosits of 6 filters, filter size 5x5 and
stride of 1
lenet_5_model.add(layers.Conv2D(filters = 6, kernel_size = (5,5),
strides=1, padding = 'same', activation = 'relu', input_shape =
(28,28,1)))
#Pooling layer 1
lenet_5_model.add(layers.MaxPooling2D(pool_size = (2,2)))
# Convolutional layer2: cosits of 16 filters, filter size 5x5 and
stride of 1
lenet_5_model.add(layers.Conv2D(filters = 16, kernel_size = 5,
strides=1, activation = 'relu'))
#Pooling layer 2
lenet_5_model.add(layers.MaxPooling2D(pool_size = (2,2)))

#Output layer( Fully connected layers)
lenet_5_model.add(layers.Flatten())
lenet_5_model.add(layers.Dense(120, activation='relu'))
lenet_5_model.add(layers.Dense(84, activation='relu'))
lenet_5_model.add(layers.Dense(10, activation='softmax'))
```

```
# Save your model
lenet_5_model.save('part2_model_max_pooling.h5')

lenet_5_model.summary()
```

```
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics
have yet to be built. `model.compile_metrics` will be empty until you
train or evaluate the model.
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_10 (Conv2D)          (None, 28, 28, 6)         156

 max_pooling2d_4 (MaxPoolin  (None, 14, 14, 6)         0
 g2D)

 conv2d_11 (Conv2D)          (None, 10, 10, 16)        2416

 max_pooling2d_5 (MaxPoolin  (None, 5, 5, 16)          0
 g2D)

 flatten_8 (Flatten)         (None, 400)               0

 dense_21 (Dense)            (None, 120)               48120

 dense_22 (Dense)            (None, 84)                10164

 dense_23 (Dense)            (None, 10)                850

=================================================================
Total params: 61706 (241.04 KB)
Trainable params: 61706 (241.04 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Training the model

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
lenet_5_model.compile(optimizer=optimizer,

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = lenet_5_model.fit(train_images, train_labels, epochs=5,
                    validation_data=(test_images, test_labels))
```

```
Epoch 1/5
1875/1875 [==============================] - 12s 6ms/step - loss:
0.1888 - accuracy: 0.9416 - val_loss: 0.1017 - val_accuracy: 0.9654
```

```
Epoch 2/5
1875/1875 [==============================] - 10s 5ms/step - loss:
0.0637 - accuracy: 0.9799 - val_loss: 0.0586 - val_accuracy: 0.9817
Epoch 3/5
1875/1875 [==============================] - 10s 5ms/step - loss:
0.0446 - accuracy: 0.9854 - val_loss: 0.0482 - val_accuracy: 0.9838
Epoch 4/5
1875/1875 [==============================] - 10s 5ms/step - loss:
0.0356 - accuracy: 0.9883 - val_loss: 0.0386 - val_accuracy: 0.9884
Epoch 5/5
1875/1875 [==============================] - 11s 6ms/step - loss:
0.0293 - accuracy: 0.9906 - val_loss: 0.0435 - val_accuracy: 0.9860
```

Evaluating the model

```
# Evaluate prediction accuracy
test_loss, test_acc = lenet_5_model.evaluate(test_images,
test_labels, verbose=2)

# Evaluate Model Size
def get_gzipped_model_size(file):
  # Returns size of gzipped model, in bytes.
  import os
  import zipfile

  _, zipped_file = tempfile.mkstemp('.zip')
  with zipfile.ZipFile(zipped_file, 'w',
compression=zipfile.ZIP_DEFLATED) as f:
    f.write(file)

  return os.path.getsize(zipped_file)


# Evaluate Inference Time
startTime = time.time()
prediction = lenet_5_model.predict(test_images)
executionTime = (time.time() - startTime)/len(test_images)

# Print
print('\nModel Accuracy:', test_acc*100, '%')
print("Model Size: %.2f bytes" %
(get_gzipped_model_size('part2_model_max_pooling.h5')))
print("Inference Time is", executionTime, "s")
```

```
313/313 - 1s - loss: 0.0435 - accuracy: 0.9860 - 1s/epoch - 3ms/step
313/313 [==============================] - 1s 3ms/step

Model Accuracy: 98.60000014305115 %
Model Size: 228387.00 bytes
Inference Time is 0.0001449413776397705 s
```

Observations

Changing the pooling type from average to maximum does not affect overall model accuracy yielding 98.60% and 98.97% with the average pooling. Furthermore, the model sizes are tad different with average pooling at 228337 bytes and max pooling at 228387 bytes. A hypothesis to this is that floating point representations that are stored in the model may not exactly be equal.

Question 4:

IoT devices are limited in storage and computation resources. Therefore lightweight machine-learning or compressed models are ideal for IoT edge devices.

In this exercise, you will modify the LeNet-5 CNN model on the MNIST dataset to reduce its size as much as possible (expecting less than 13k parameters) while maintaining inference accuracy equal to or above 95%. There is no limitation to the applied modifications. You may modify the number of filters, filter size, stride value, fully connected layers etc.**(1 point)**

Building the model

```
lenet_5_model = models.Sequential()
# Convolutional layer1: cosits of 6 filters, filter size 5x5 and
stride of 1
lenet_5_model.add(layers.Conv2D(filters = 6, kernel_size =
(5,5),strides=2, padding = 'same', activation = 'relu', input_shape =
(28,28,1)))
#Pooling layer 1
lenet_5_model.add(layers.AveragePooling2D(pool_size = (2,2)))
# Convolutional layer2: cosits of 16 filters, filter size 5x5 and
stride of 1
lenet_5_model.add(layers.Conv2D(filters = 10, kernel_size =
5,strides=1, activation = 'relu'))
#Pooling layer 2
lenet_5_model.add(layers.AveragePooling2D(pool_size = (2,2)))

#Output layer( Fully connected layers)
lenet_5_model.add(layers.Flatten())
lenet_5_model.add(layers.Dense(109, activation='relu'))
lenet_5_model.add(layers.Dense(84, activation='relu'))
lenet_5_model.add(layers.Dense(10, activation='softmax'))
lenet_5_model.summary()
```

```
Model: "sequential_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_8 (Conv2D)           (None, 14, 14, 6)         156

 average_pooling2d_4 (Avera  (None, 7, 7, 6)           0
 gePooling2D)
```

```
 conv2d_9 (Conv2D)           (None, 3, 3, 10)          1510

 average_pooling2d_5 (Avera  (None, 1, 1, 10)          0
 gePooling2D)

 flatten_7 (Flatten)         (None, 10)                0

 dense_18 (Dense)            (None, 109)               1199

 dense_19 (Dense)            (None, 84)                9240

 dense_20 (Dense)            (None, 10)                850

=================================================================
Total params: 12955 (50.61 KB)
Trainable params: 12955 (50.61 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Training the model

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
lenet_5_model.compile(optimizer=optimizer,

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = lenet_5_model.fit(train_images, train_labels, epochs=5,
                    validation_data=(test_images, test_labels))

Epoch 1/5
1875/1875 [==============================] - 10s 5ms/step - loss:
0.5694 - accuracy: 0.8137 - val_loss: 0.2629 - val_accuracy: 0.9187
Epoch 2/5
1875/1875 [==============================] - 9s 5ms/step - loss:
0.2357 - accuracy: 0.9267 - val_loss: 0.1989 - val_accuracy: 0.9373
Epoch 3/5
1875/1875 [==============================] - 8s 4ms/step - loss:
0.1845 - accuracy: 0.9437 - val_loss: 0.1835 - val_accuracy: 0.9391
Epoch 4/5
1875/1875 [==============================] - 11s 6ms/step - loss:
0.1516 - accuracy: 0.9531 - val_loss: 0.1391 - val_accuracy: 0.9567
Epoch 5/5
1875/1875 [==============================] - 9s 5ms/step - loss:
0.1290 - accuracy: 0.9594 - val_loss: 0.1275 - val_accuracy: 0.9592
```

Evaluating the model

```
# Evaluate prediction accuracy
test_loss, test_acc = lenet_5_model.evaluate(test_images,
test_labels, verbose=2)

# Evaluate Model Size
def get_gzipped_model_size(file):
  # Returns size of gzipped model, in bytes.
  import os
  import zipfile

  _, zipped_file = tempfile.mkstemp('.zip')
  with zipfile.ZipFile(zipped_file, 'w',
compression=zipfile.ZIP_DEFLATED) as f:
    f.write(file)

  return os.path.getsize(zipped_file)


# Evaluate Inference Time
startTime = time.time()
prediction = lenet_5_model.predict(test_images)
executionTime = (time.time() - startTime)/len(test_images)

# Print
print('\nModel Accuracy:', test_acc*100, '%')
print("Model Size: %.2f bytes" %
(get_gzipped_model_size('part2_model.h5')))
print("Inference Time is", executionTime, "s")
```

```
313/313 - 1s - loss: 0.1275 - accuracy: 0.9592 - 650ms/epoch -
2ms/step
313/313 [==============================] - 1s 2ms/step

Model Accuracy: 95.92000246047974 %
Model Size: 228337.00 bytes
Inference Time is 0.00010336132049560547 s
```

Summary

From this lab report, it was observed that a very high learning rate can yeild poor performances much like a very small learning rate can yeild poor performances. It is important to **find the optimal learning rate** that yeilds the best performances for a given number of epochs for training.

Furthermore, *decreasing the number of neurons* in the Dense layer by a factor of 16 substantially *reduced the inference time* of the model by a factor of 112000. Furthermore, it *decreased the*

model's accuracy by ~8%, *decreased the model parameters by factor of 15*, and *decreased the model size by a factor of 5.*

On the other hand, *increasing the number of neurons* in the Dense layer by a factor of 16, substantially *reduced the inference time* of the model by a factor of 62000, *decreased the accuracy by 0.1%, and increased the model's size by a factor of 45.*

Incorporating CNN layers onto the FC neural network reduced the number of parameters of the model by ~30,000 - 40,000. Furthermore, CNN incorporation yeilds a model that is ~77850x faster and ~0.6x smaller, and 1-2% higher in accuracy.