In this problem of multiclass classification , we are going to build a neural network to classify images of different items of clothing.

```
import tensorflow as tf
from tensorflow.keras.datasets import fashion_mnist

#The data has already been sorted into training and test sets for us.
(train_data, train_labels), (test_data, test_labels) = fashion_mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 [==============================] - 0s 0us/step
```

```
#Showing the first training example
print(f"Training Sample:\n{train_data[0]}\n")
print(f"Training Label:\n{train_labels[0]}\n")
```

```
Training Sample:
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   1   0   0  13  73   0
    0   1   4   0   0   0   0   1   1   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   3   0  36 136 127  62
   54   0   0   0   1   3   4   0   0   3]
 [  0   0   0   0   0   0   0   0   0   0   0   0   6   0 102 204 176 134
  144 123  23   0   0   0   0  12  10   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0 155 236 207 178
  107 156 161 109  64  23  77 130  72  15]
 [  0   0   0   0   0   0   0   0   0   0   0   1   0  69 207 223 218 216
  216 163 127 121 122 146 141  88 172  66]
 [  0   0   0   0   0   0   0   0   0   1   1   1   0 200 232 232 233 229
  223 223 215 213 164 127 123 196 229   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 183 225 216 223 228
  235 227 224 222 224 221 223 245 173   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 193 228 218 213 198
  180 212 210 211 213 223 220 243 202   0]
 [  0   0   0   0   0   0   0   0   0   1   3   0  12 219 220 212 218 192
  169 227 208 218 224 212 226 197 209  52]
 [  0   0   0   0   0   0   0   0   0   0   6   0  99 244 222 220 218 203
  198 221 215 213 222 220 245 119 167  56]
 [  0   0   0   0   0   0   0   0   0   4   0   0  55 236 228 230 228 240
  232 213 218 223 234 217 217 209  92   0]
 [  0   0   1   4   6   7   2   0   0   0   0   0 237 226 217 223 222 219
  222 221 216 223 229 215 218 255  77   0]
 [  0   3   0   0   0   0   0   0   0  62 145 204 228 207 213 221 218 208
  211 218 224 223 219 215 224 244 159   0]
 [  0   0   0   0  18  44  82 107 189 228 220 222 217 226 200 205 211 230
  224 234 176 188 250 248 233 238 215   0]
 [  0  57 187 208 224 221 224 208 204 214 208 209 200 159 245 193 206 223
  255 255 221 234 221 211 220 232 246   0]
 [  3 202 228 224 221 211 211 214 205 205 205 220 240  80 150 255 229 221
  188 154 191 210 204 209 222 228 225   0]
 [ 98 233 198 210 222 229 229 234 249 220 194 215 217 241  65  73 106 117
  168 219 221 215 217 223 223 224 229  29]
 [ 75 204 212 204 193 205 211 225 216 185 197 206 198 213 240 195 227 245
  239 223 218 212 209 222 220 221 230  67]
 [ 48 203 183 194 213 197 185 190 194 192 202 214 219 221 220 236 225 216
  199 206 186 181 177 172 181 205 206 115]
 [  0 122 219 193 179 171 183 196 204 210 213 207 211 210 200 196 194 191
  195 191 198 192 176 156 167 177 210  92]
 [  0   0  74 189 212 191 175 172 175 181 185 188 189 188 193 198 204 209
  210 210 211 188 188 194 192 216 170   0]
 [  2   0   0   0  66 200 222 237 239 242 246 243 244 221 220 193 191 179
  182 182 181 176 166 168  99  58   0   0]
 [  0   0   0   0   0   0   0  40  61  44  72  41  35   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]]
```

Checking the shapes of the test and train data

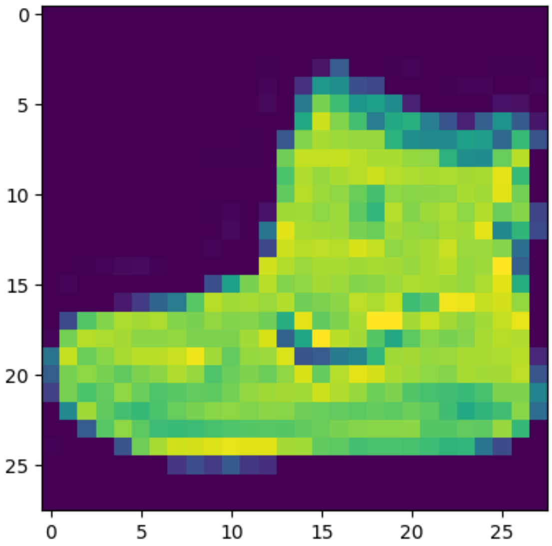
```
train_data.shape, train_labels.shape
```

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

```
test_data.shape, test_labels.shape
```

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

```
#Plotting a single sample
import matplotlib.pyplot as plt
plt.imshow(train_data[0]);
```




```
#checking its label
train_labels[0]
```

```
9
```

```
#creating human readable labels of given training data

class_names = ["T-shirt/top", "Trousers", "Pullover", "Dress", "Coat", "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]

len(class_names)
```

```
10
```

```
#Plotting an example image and its label
index = 18
plt.imshow(train_data[index], cmap=plt.cm.binary)
plt.title(class_names[train_labels[index]])
```

```
Text(0.5, 1.0, 'Shirt')
```


```
import random
plt.figure(figsize=(7,7))
for i in range(4):
  ax = plt.subplot(2, 2, i+1)
  random_index = random.choice(range(len(train_data)))
  plt.imshow(train_data[random_index], cmap=plt.cm.binary)
  plt.title(class_names[train_labels[random_index]])
  plt.axis(False)
```

Now lets build the multiclassification model

```
#setting the random seed
tf.random.set_seed(42)

#create the model
model_1 = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28)),
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(4, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])


#compiling the model
model_1.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(),
                optimizer=tf.keras.optimizers.Adam(),
                metrics = ['accuracy'])

#Fitting the model
history = model_1.fit(train_data,
                      train_labels,
                      epochs=10,
                      validation_data=(test_data,test_labels))
```

```
Epoch 1/10
1875/1875 [==============================] - 10s 4ms/step - loss: 2.1452 - accuracy: 0.1860 - val_loss: 1.7850 - val_
Epoch 2/10
1875/1875 [==============================] - 7s 4ms/step - loss: 1.7162 - accuracy: 0.2665 - val_loss: 1.6408 - val_a
Epoch 3/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.6369 - accuracy: 0.2772 - val_loss: 1.5692 - val_a
Epoch 4/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.5362 - accuracy: 0.3207 - val_loss: 1.4878 - val_a
Epoch 5/10
```

```
1875/1875 [==============================] - 5s 2ms/step - loss: 1.4750 - accuracy: 0.3490 - val_loss: 1.4376 - val_a
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 1.4402 - accuracy: 0.3675 - val_loss: 1.4502 - val_a
Epoch 7/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.4186 - accuracy: 0.3839 - val_loss: 1.5016 - val_a
Epoch 8/10
1875/1875 [==============================] - 5s 3ms/step - loss: 1.4182 - accuracy: 0.3844 - val_loss: 1.4111 - val_a
Epoch 9/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.4217 - accuracy: 0.3868 - val_loss: 1.4088 - val_a
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 1.3855 - accuracy: 0.3961 - val_loss: 1.3860 - val_a
```

```
#checkinng the model summary

model_1.summary()
```

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

 dense (Dense)               (None, 4)                 3140

 dense_1 (Dense)             (None, 4)                 20

 dense_2 (Dense)             (None, 10)                50

=================================================================
Total params: 3,210
Trainable params: 3,210
Non-trainable params: 0
_________________________________________________________________
```

Now lets try to improve the accuracy by standardising or normalizing the data (between 0 and 1)

```
#checking the min and max values of the training data
train_data.min(),train_data.max()
```

```
(0, 255)
```

```
#normalizing the training and testing data
norm_train_data = train_data/255.0
norm_test_data = test_data/255.0

#checking our normalized data
norm_train_data.max(), norm_test_data.min()
```

```
(1.0, 0.0)
```

```
#Using this normalised data on the same model we built above

#setting random seed
tf.random.set_seed(42)

#building the model
model_2 = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape = (28,28)),
    tf.keras.layers.Dense(4, activation = "relu"),
    tf.keras.layers.Dense(4, activation = "relu"),
    tf.keras.layers.Dense(10, activation = "softmax")
])

#compiling the model
model_2.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(),
                optimizer=tf.keras.optimizers.Adam(),
                metrics = ['accuracy'])

#fitting the model
norm_history = model_2.fit(norm_train_data,
                           train_labels,
                           epochs=10,
                           validation_data = (norm_test_data, test_labels))
```

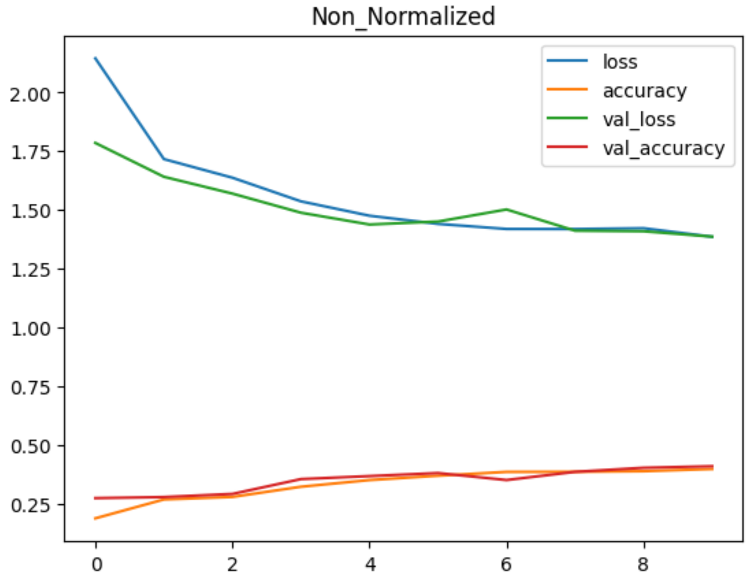
```
Epoch 1/10
1875/1875 [==============================] - 6s 3ms/step - loss: 1.6080 - accuracy: 0.4029 - val_loss: 1.1924 - val_a
Epoch 2/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.0581 - accuracy: 0.6031 - val_loss: 0.9879 - val_a
Epoch 3/10
```

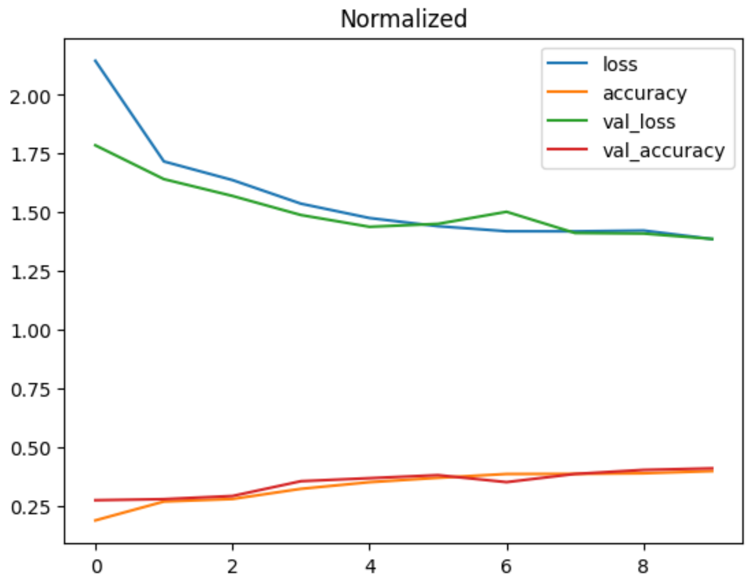
```
1875/1875 [==============================] - 4s 2ms/step - loss: 0.8989 - accuracy: 0.6571 - val_loss: 0.8808 - val_a
Epoch 4/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.8250 - accuracy: 0.6763 - val_loss: 0.8377 - val_a
Epoch 5/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.7917 - accuracy: 0.6867 - val_loss: 0.8135 - val_a
Epoch 6/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.7653 - accuracy: 0.7173 - val_loss: 0.7885 - val_a
Epoch 7/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.7322 - accuracy: 0.7395 - val_loss: 0.7580 - val_a
Epoch 8/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.7015 - accuracy: 0.7476 - val_loss: 0.7343 - val_a
Epoch 9/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6805 - accuracy: 0.7575 - val_loss: 0.7141 - val_a
Epoch 10/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6637 - accuracy: 0.7624 - val_loss: 0.7001 - val_a
```

Plotting the loss curves for normalized data and non-normalized data

```
import pandas as pd

#Plotting the non-normalized data loss curve
pd.DataFrame(history.history).plot(title = "Non_Normalized")

#Plotting the normalized data loss curve
pd.DataFrame(history.history).plot(title = "Normalized")
```

```
<Axes: title={'center': 'Normalized'}>
```





Finding the ideal learning rate using the callback

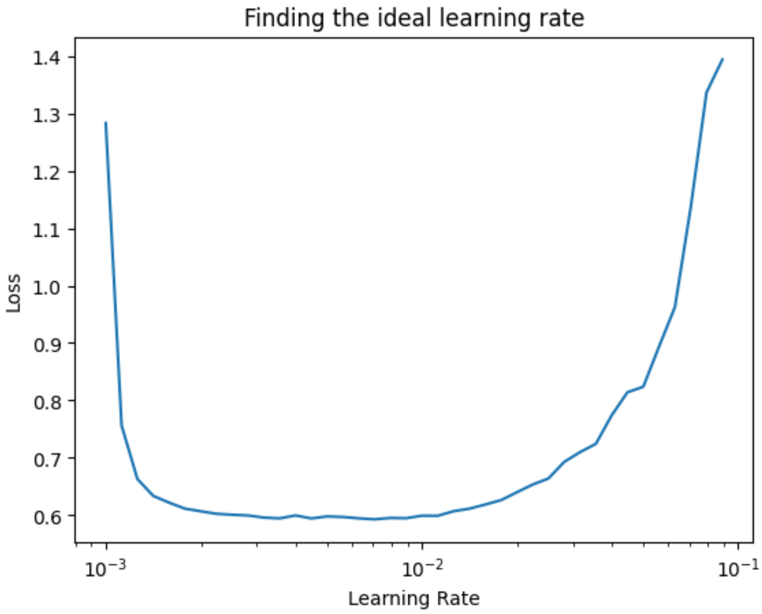
```
#Using this normalised data on the same model we built above but with callback method

#setting random seed
tf.random.set_seed(42)

#building the model
model_3 = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape = (28,28)),
    tf.keras.layers.Dense(4, activation = "relu"),
    tf.keras.layers.Dense(4, activation = "relu"),
    tf.keras.layers.Dense(10, activation = "softmax")
])

#compiling the model
model_3.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(),
                optimizer=tf.keras.optimizers.Adam(),
                metrics = ['accuracy'])

#creaating the learning rate callback
lr_schedular = tf.keras.callbacks.LearningRateScheduler(lambda epoch : 1e-3 * 10**(epoch/20))

#fitting the model
lr_norm_history = model_3.fit(norm_train_data,
                              train_labels,
                              epochs=40,
                              validation_data = (norm_test_data, test_labels),
                              callbacks=[lr_schedular])
```

```
Epoch 12/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5546 - accuracy: 0.7938 - val_loss: 0.6100 - val_a
Epoch 13/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5538 - accuracy: 0.7939 - val_loss: 0.6563 - val_a
Epoch 14/40
1875/1875 [==============================] - 5s 3ms/step - loss: 0.5568 - accuracy: 0.7910 - val_loss: 0.5970 - val_a
Epoch 15/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5566 - accuracy: 0.7927 - val_loss: 0.5727 - val_a
Epoch 16/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5603 - accuracy: 0.7911 - val_loss: 0.5771 - val a
Epoch 17/40
1875/1875 [==============================] - 5s 3ms/step - loss: 0.5584 - accuracy: 0.7926 - val_loss: 0.6058 - v
Epoch 18/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5637 - accuracy: 0.7922 - val_loss: 0.5792 - v
Epoch 19/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5637 - accuracy: 0.7907 - val_loss: 0.5902 - v
Epoch 20/40
1875/1875 [==============================] - 5s 2ms/step - loss: 0.5663 - accuracy: 0.7909 - val_loss: 0.5899 - v
Epoch 21/40
1875/1875 [==============================] - 5s 2ms/step - loss: 0.5722 - accuracy: 0.7885 - val_loss: 0.6066 - val_a
Epoch 22/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5775 - accuracy: 0.7864 - val_loss: 0.5940 - val_a
Epoch 23/40
1875/1875 [==============================] - 5s 3ms/step - loss: 0.5867 - accuracy: 0.7831 - val_loss: 0.6235 - val_a
Epoch 24/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5864 - accuracy: 0.7849 - val_loss: 0.6453 - val_a
Epoch 25/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5939 - accuracy: 0.7829 - val_loss: 0.6242 - val_a
Epoch 26/40
1875/1875 [==============================] - 6s 3ms/step - loss: 0.6108 - accuracy: 0.7779 - val_loss: 0.8094 - val_a
Epoch 27/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6161 - accuracy: 0.7770 - val_loss: 0.6748 - val_a
Epoch 28/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6295 - accuracy: 0.7722 - val_loss: 0.6362 - val_a
Epoch 29/40
```

```
1875/1875 [==============================] - 4s 2ms/step - loss: 1.3824 - accuracy: 0.3855 - val_loss: 1.4553 - val_a
```

```
#plotting the learning rate decay curve
import numpy as np
import matplotlib.pyplot as plt

lrs = 1e-3 * (10**(tf.range(40)/20))
plt.semilogx(lrs, lr_norm_history.history["loss"])
plt.xlabel("Learning Rate")
plt.ylabel("Loss")
plt.title("Finding the ideal learning rate")
```

```
Text(0.5, 1.0, 'Finding the ideal learning rate')
```



Lets now rebuild our model with the ideal learning rate

```
#Using this normalised data on the same model we built above but with callback method and ideal learning rate

#setting random seed
tf.random.set_seed(42)

#building the model
model_4 = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape = (28,28)),
    tf.keras.layers.Dense(4, activation = "relu"),
    tf.keras.layers.Dense(4, activation = "relu"),
    tf.keras.layers.Dense(10, activation = "softmax")
])

#compiling the model
model_4.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(),
                optimizer=tf.keras.optimizers.Adam(lr=0.001),
                metrics = ['accuracy'])

#fitting the model
lr_norm_history = model_4.fit(norm_train_data,
                             train_labels,
                             epochs=40,
                             validation_data = (norm_test_data, test_labels))
```

```
WARNING:absl:`lr` is deprecated in Keras optimizer, please use `learning_rate` or use the legacy optimizer, e.g.,tf.k
Epoch 1/40
1875/1875 [==============================] - 5s 2ms/step - loss: 1.3171 - accuracy: 0.4884 - val_loss: 0.9585 - val_a
Epoch 2/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.8798 - accuracy: 0.6441 - val_loss: 0.8718 - val_a
Epoch 3/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.8215 - accuracy: 0.6564 - val_loss: 0.8271 - val_a
Epoch 4/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.7902 - accuracy: 0.6858 - val_loss: 0.7929 - val_a
Epoch 5/40
1875/1875 [==============================] - 5s 3ms/step - loss: 0.7597 - accuracy: 0.7195 - val_loss: 0.7652 - val_a
Epoch 6/40
1875/1875 [==============================] - 3s 2ms/step - loss: 0.7404 - accuracy: 0.7331 - val_loss: 0.7708 - val_a
Epoch 7/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.7263 - accuracy: 0.7423 - val_loss: 0.7695 - val_a
Epoch 8/40
```

```
1875/1875 [==============================] - 5s 2ms/step - loss: 0.7077 - accuracy: 0.7509 - val_loss: 0.7184 - val_a
Epoch 9/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.6164 - accuracy: 0.7886 - val_loss: 0.6267 - val_a
Epoch 10/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5807 - accuracy: 0.8024 - val_loss: 0.6092 - val_a
Epoch 11/40
1875/1875 [==============================] - 5s 2ms/step - loss: 0.5665 - accuracy: 0.8081 - val_loss: 0.6037 - val_a
Epoch 12/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5559 - accuracy: 0.8114 - val_loss: 0.5957 - val_a
Epoch 13/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5495 - accuracy: 0.8120 - val_loss: 0.5907 - val_a
Epoch 14/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5435 - accuracy: 0.8153 - val_loss: 0.5884 - val_a
Epoch 15/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5389 - accuracy: 0.8170 - val_loss: 0.5807 - val_a
Epoch 16/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5353 - accuracy: 0.8172 - val_loss: 0.5780 - val_a
Epoch 17/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5300 - accuracy: 0.8192 - val_loss: 0.5801 - val_a
Epoch 18/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5285 - accuracy: 0.8188 - val_loss: 0.5752 - val_a
Epoch 19/40
1875/1875 [==============================] - 3s 2ms/step - loss: 0.5255 - accuracy: 0.8208 - val_loss: 0.5760 - val_a
Epoch 20/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5227 - accuracy: 0.8225 - val_loss: 0.5840 - val_a
Epoch 21/40
1875/1875 [==============================] - 5s 3ms/step - loss: 0.5202 - accuracy: 0.8215 - val_loss: 0.5786 - val_a
Epoch 22/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5176 - accuracy: 0.8235 - val_loss: 0.5776 - val_a
Epoch 23/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5165 - accuracy: 0.8232 - val_loss: 0.5770 - val_a
Epoch 24/40
1875/1875 [==============================] - 5s 3ms/step - loss: 0.5133 - accuracy: 0.8250 - val_loss: 0.5666 - val_a
Epoch 25/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5112 - accuracy: 0.8249 - val_loss: 0.5824 - val_a
Epoch 26/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5126 - accuracy: 0.8245 - val_loss: 0.5653 - val_a
Epoch 27/40
1875/1875 [==============================] - 5s 3ms/step - loss: 0.5085 - accuracy: 0.8258 - val_loss: 0.5614 - val_a
Epoch 28/40
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5069 - accuracy: 0.8263 - val_loss: 0.5616 - v
Epoch 29/40
```

Evaulating our multiclass classification model

```
# Note: The following confusion matrix code is a remix of Scikit-Learn's
# plot_confusion_matrix function - https://scikit-learn.org/stable/modules/generated/sklearn.metrics.plot_confusion_matrix
# and Made with ML's introductory notebook - https://github.com/GokuMohandas/MadeWithML/blob/main/notebooks/08_Neural_Netw
import itertools
from sklearn.metrics import confusion_matrix

# Our function needs a different name to sklearn's plot_confusion_matrix
def make_confusion_matrix(y_true, y_pred, classes=None, figsize=(10, 10), text_size=15):
  """Makes a labelled confusion matrix comparing predictions and ground truth labels.

  If classes is passed, confusion matrix will be labelled, if not, integer class values
  will be used.

  Args:
    y_true: Array of truth labels (must be same shape as y_pred).
    y_pred: Array of predicted labels (must be same shape as y_true).
    classes: Array of class labels (e.g. string form). If `None`, integer labels are used.
    figsize: Size of output figure (default=(10, 10)).
    text_size: Size of output figure text (default=15).

  Returns:
    A labelled confusion matrix plot comparing y_true and y_pred.

  Example usage:
    make_confusion_matrix(y_true=test_labels, # ground truth test labels
                          y_pred=y_preds, # predicted labels
                          classes=class_names, # array of class label names
                          figsize=(15, 15),
                          text_size=10)
  """
  # Create the confustion matrix
  cm = confusion_matrix(y_true, y_pred)
  cm_norm = cm.astype("float") / cm.sum(axis=1)[:, np.newaxis] # normalize it
  n_classes = cm.shape[0] # find the number of classes we're dealing with

  # Plot the figure and make it pretty
```

```
  fig, ax = plt.subplots(figsize=figsize)
  cax = ax.matshow(cm, cmap=plt.cm.Blues) # colors will represent how 'correct' a class is, darker == better
  fig.colorbar(cax)

  # Are there a list of classes?
  if classes:
    labels = classes
  else:
    labels = np.arange(cm.shape[0])

  # Label the axes
  ax.set(title="Confusion Matrix",
         xlabel="Predicted label",
         ylabel="True label",
         xticks=np.arange(n_classes), # create enough axis slots for each class
         yticks=np.arange(n_classes),
         xticklabels=labels, # axes will labeled with class names (if they exist) or ints
         yticklabels=labels)

  # Make x-axis labels appear on bottom
  ax.xaxis.set_label_position("bottom")
  ax.xaxis.tick_bottom()

  # Set the threshold for different colors
  threshold = (cm.max() + cm.min()) / 2.

  # Plot the text on each cell
  for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, f"{cm[i, j]} ({cm_norm[i, j]*100:.1f}%)",
             horizontalalignment="center",
             color="white" if cm[i, j] > threshold else "black",
             size=text_size)
```

```
class_names
```

```
['T-shirt/top',
 'Trousers',
 'Pullover',
 'Dress',
 'Coat',
 'Sandal',
 'Shirt',
 'Sneaker',
 'Bag',
 'Ankle boot']
```

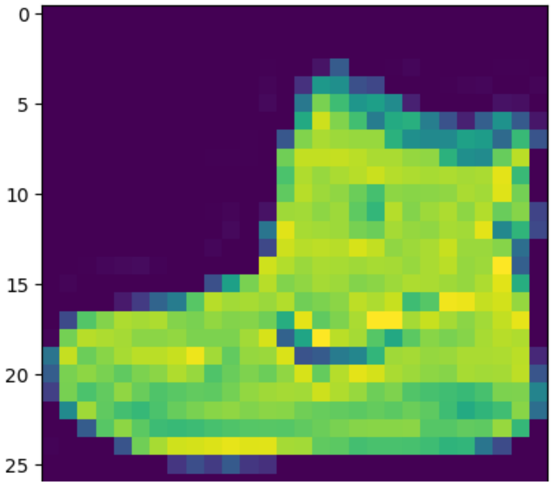
```
test_labels
```

```
array([9, 2, 1, ..., 8, 1, 5], dtype=uint8)
```

```
#Lets make predictions with our model
y_probs = model_4.predict(norm_test_data)

#Viewing the firsr 5 predictions
y_probs[:5]
```

```
313/313 [==============================] - 1s 2ms/step
array([[1.65595619e-14, 1.33894042e-11, 8.38167106e-16, 1.64987485e-25,
        8.12260745e-11, 3.35326016e-01, 4.41123180e-12, 1.15524903e-01,
        2.26949146e-06, 5.49146891e-01],
       [7.65582536e-06, 3.52378776e-17, 9.56999719e-01, 6.71652911e-09,
        1.16431816e-02, 1.29238295e-11, 3.13492306e-02, 5.53316900e-34,
        1.54084987e-07, 7.98325974e-24],
       [2.39907950e-03, 9.78370845e-01, 7.93917934e-06, 1.19544845e-02,
        2.48201486e-05, 2.62084045e-03, 9.18941732e-05, 4.43833135e-03,
        8.30943973e-05, 8.71042357e-06],
       [2.39907950e-03, 9.78370845e-01, 7.93917934e-06, 1.19544845e-02,
        2.48201486e-05, 2.62084045e-03, 9.18941732e-05, 4.43833135e-03,
        8.30943973e-05, 8.71042357e-06],
       [1.35179564e-01, 9.24505457e-06, 5.09983063e-01, 5.59618790e-03,
        1.30348951e-02, 1.49875405e-05, 3.35897774e-01, 5.20340694e-18,
        2.84260197e-04, 1.92658254e-13]], dtype=float32)
```

```
#plotting the ist data object
plt.imshow(train_data[0]);
```




```
#Making its prediction
y_probs[0], tf.argmax(y_probs[0]), class_names[tf.argmax(y_probs[0])]
```

```
(array([1.6559562e-14, 1.3389404e-11, 8.3816711e-16, 1.6498748e-25,
        8.1226074e-11, 3.3532602e-01, 4.4112318e-12, 1.1552490e-01,
        2.2694915e-06, 5.4914689e-01], dtype=float32),
 <tf.Tensor: shape=(), dtype=int64, numpy=9>,
 'Ankle boot')
```

```
#Now lets convert all the prediction probabilities into integers
y_preds = y_probs.argmax(axis=1)

#Viewing the first 1 predictions
y_preds[0], class_names[tf.argmax(y_probs[0])]
```

```
(9, 'Ankle boot')
```

```
test_labels
```

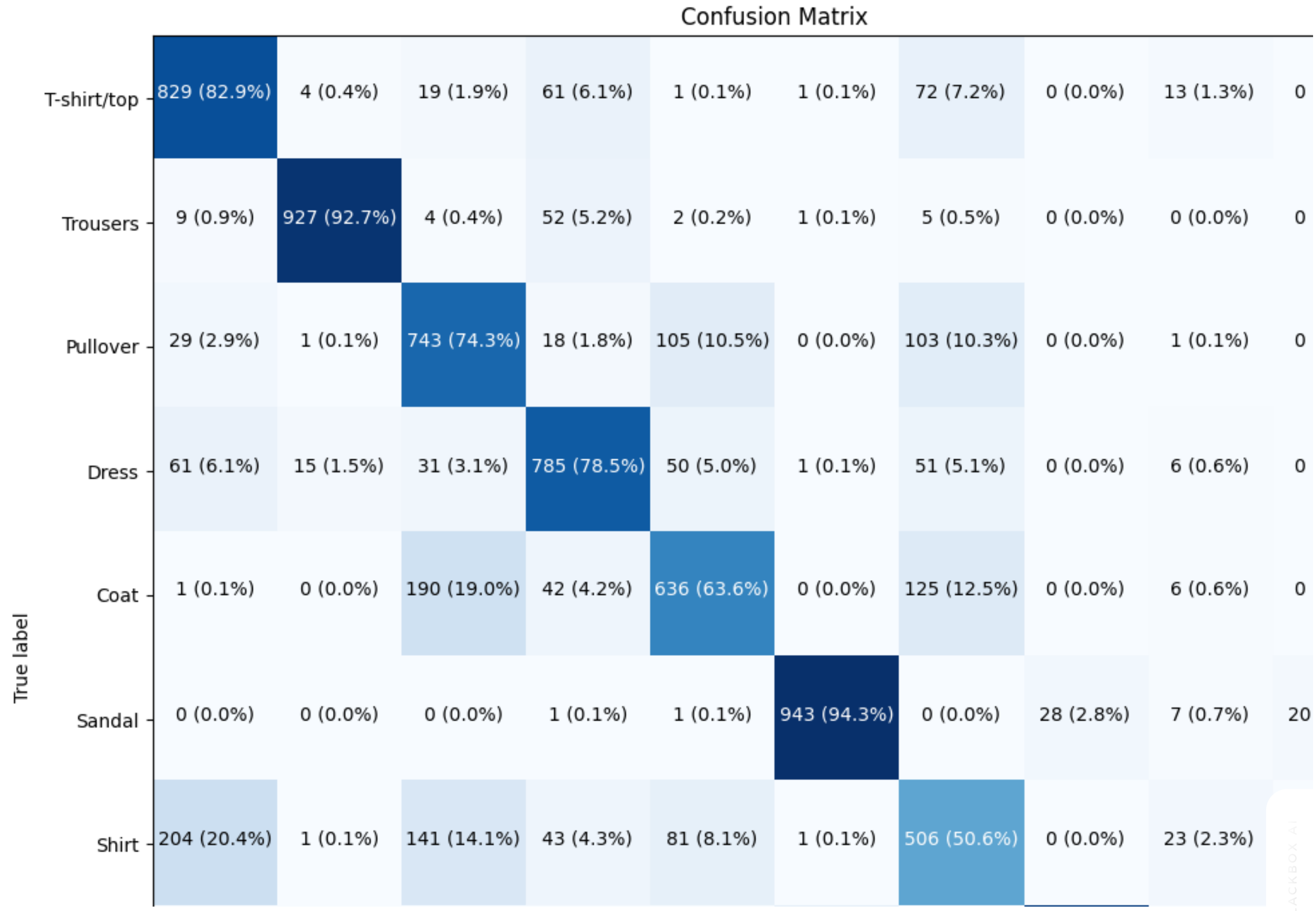
```
array([9, 2, 1, ..., 8, 1, 5], dtype=uint8)
```

Confusion Matrix

```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_true=test_labels,
                 y_pred=y_preds)
```

```
array([[829,   4,  19,  61,   1,   1,  72,   0,  13,   0],
       [  9, 927,   4,  52,   2,   1,   5,   0,   0,   0],
       [ 29,   1, 743,  18, 105,   0, 103,   0,   1,   0],
       [ 61,  15,  31, 785,  50,   1,  51,   0,   6,   0],
       [  1,   0, 190,  42, 636,   0, 125,   0,   6,   0],
       [  0,   0,   0,   1,   1, 943,   0,  28,   7,  20],
       [204,   1, 141,  43,  81,   1, 506,   0,  23,   0],
       [  0,   0,   0,   0,   0,  76,   0, 888,   1,  35],
       [  8,   1,   3,   4,   2,   9,  48,   4, 921,   0],
       [  0,   0,   0,   0,   0,  40,   1,  44,   0, 915]])
```

```
#Lets make a better confusion matrix
make_confusion_matrix(y_true=test_labels,
                      y_pred=y_preds,
                      classes=class_names,
                      figsize=(15,15),
                      text_size = 10)
```



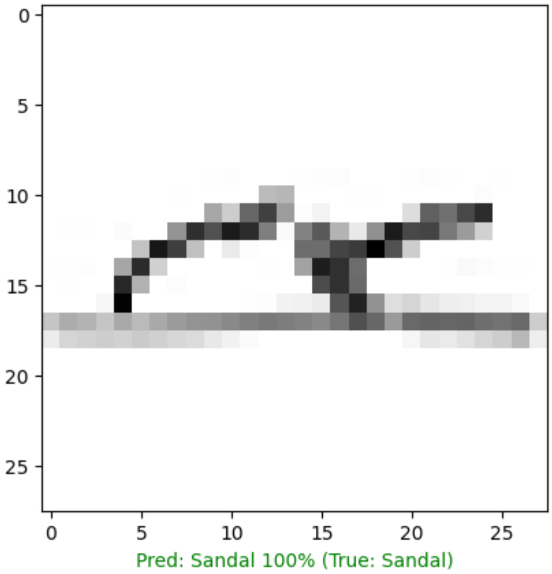
Let's create a function to plot a random image along with its prediction.

```
import random

# Create a function for plotting a random image along with its prediction
def plot_random_image(model, images, true_labels, classes):
  """Picks a random image, plots it and labels it with a predicted and truth label.

  Args:
    model: a trained model (trained on data similar to what's in images).
    images: a set of random images (in tensor form).
    true_labels: array of ground truth labels for images.
    classes: array of class names for images.

  Returns:
    A plot of a random image from `images` with a predicted class label from `model`
    as well as the truth class label from `true_labels`.
  """
  # Setup random integer
  i = random.randint(0, len(images))

  # Create predictions and targets
  target_image = images[i]
  pred_probs = model.predict(target_image.reshape(1, 28, 28)) # have to reshape to get into right size for model
  pred_label = classes[pred_probs.argmax()]
  true_label = classes[true_labels[i]]

  # Plot the target image
  plt.imshow(target_image, cmap=plt.cm.binary)

  # Change the color of the titles depending on if the prediction is right or wrong
  if pred_label == true_label:
    color = "green"
  else:
    color = "red"

  # Add xlabel information (prediction/true label)
```

```
    plt.xlabel("Pred: {} {:2.0f}% (True: {})".format(pred_label,
                                                     100*tf.reduce_max(pred_probs),
                                                     true_label),
                                                     color=color) # set the color to green or red
```

```
# Check out a random image as well as its prediction
plot_random_image(model=model_4,
                  images=test_data,
                  true_labels=test_labels,
                  classes=class_names)
```

```
1/1 [==============================] - 0s 22ms/step
```



Patterns our model is learning

```
# Find the layers of our most recent model
model_4.layers
```

```
[<keras.layers.reshaping.flatten.Flatten at 0x7b90837b0700>,
 <keras.layers.core.dense.Dense at 0x7b9082a0e0b0>,
 <keras.layers.core.dense.Dense at 0x7b9082a0ebc0>,
 <keras.layers.core.dense.Dense at 0x7b9082a0e440>]
```

```
# Extracting a particular layer
model_4.layers[1]
```

```
<keras.layers.core.dense.Dense at 0x7b9082a0e0b0>
```

```
# Getting the patterns of a layer in our network
weights, biases = model_4.layers[1].get_weights()
```

```
# Shape = 1 weight matrix the size of our input data (28x28) per neuron (4)
weights, weights.shape
```

```
(array([[-0.6286177 ,  0.04304771,  0.27682275, -1.1177504 ],
        [-0.50602835,  1.4612656 , -1.3399148 , -0.69901824],
        [-1.6388192 ,  1.4361247 ,  1.1517581 ,  0.20172885],
        ...,
        [-0.19037662, -0.10284518,  0.09389258, -0.10114335],
        [ 0.52314425,  0.522483  ,  0.27723047, -0.4458822 ],
        [-0.16856955, -0.40255272,  0.59475726,  0.01509975]],
       dtype=float32),
 (784, 4))
```

The weights matrix is the same shape as the input data, which in our case is 784 (28x28 pixels). And there's a copy of the weights matrix for each neuron the in the selected layer (our selected layer has 4 neurons).

Each value in the weights matrix corresponds to how a particular value in the input data influences the network's decisions.

Now let's check out the bias vector.

```
# Shape = 1 bias per neuron (we use 4 neurons in the first layer)
biases, biases.shape
```

```
(array([1.6029571, 2.7418947, 4.0676045, 1.9065827], dtype=float32), (4,))
```

Every neuron has a bias vector. Each of these is paired with a weight matrix.

The bias values get initialized as zeroes by default (using the bias_initializer parameter).

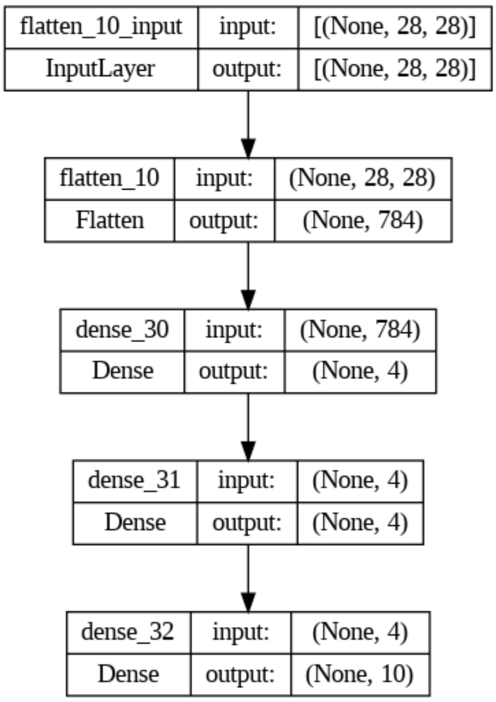
The bias vector dictates how much the patterns within the corresponding weights matrix should influence the next layer.

```
#Now lets calculate the number of paramters in our model
model_4.summary()
```

```
Model: "sequential_10"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_10 (Flatten)        (None, 784)               0

 dense_30 (Dense)            (None, 4)                 3140

 dense_31 (Dense)            (None, 4)                 20

 dense_32 (Dense)            (None, 10)                50

=================================================================
Total params: 3,210
Trainable params: 3,210
Non-trainable params: 0
_________________________________________________________________
```

Starting from the input layer, each subsequent layer's input is the output of the previous layer as shown below

```
from tensorflow.keras.utils import plot_model

# See the inputs and outputs of each layer
plot_model(model_4, show_shapes=True)
```




```
from google.colab.patches import cv2_imshow
img = cv2.imread('/content/pexels-vie-studio-4439457.jpg', cv2.IMREAD_UNCHANGED)
resized_image = cv2.resize(img,(700,300))
cv2_imshow(resized_image)
```

2s completed at 1:19 AM