Data Mining HW6

Shardul Dabhane(sdabhane) B-565

April 22nd, 2022

1. (40 points) Decision trees and ensemble approaches.

• Use sklearn's breast cancer data set (from sklearn.datasets import load breast cancer)

• Try the bagging and adaboost approaches using the decision tree as the base predictor. Experiment different parameters (e.g., number of base predictors). You may use BaggingClassifier and AdaBoostClassifier in sklearn.ensemble for this problem.

• Document what you have tried and report your results

In [1]:

```
#Import required modules
import pandas as pd
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RepeatedKFold
from sklearn.model_selection import train_test_split
```

In [2]:

```
#We will now load the breast cancer dataset and convert it into a dataframe to perform operations
breast_cancer_data = load_breast_cancer()
df = pd.DataFrame(breast_cancer_data.data, columns=breast_cancer_data.feature_names)
df['target'] = pd.Series(breast_cancer_data.target)
df.head()
```

Out[2]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	...	worst texture	worst perimeter	wo a
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	0.07871	...	17.33	184.60	201
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	0.05667	...	23.41	158.80	195
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	0.05999	...	25.53	152.50	170
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	0.09744	...	26.50	98.87	56
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	0.05883	...	16.67	152.20	157

5 rows × 31 columns

In [3]:

```
# Separate the target and the feature attribute
features = df.columns[0:-1]
target = 'target'
X = df[features]
y = df[[target]]

# Use train test split to separate the training and testing data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15,random_state=42, stratify=y)
```

In [4]:

```
#Set the base predictor to Decision Tree

dt=DecisionTreeClassifier()
```

In [5]:

```
#We will find the best parameters for bagging by passing a range of parameters
#in the GridSearchCV function and using RepeatedKFold Cross Validation
finding_best_parameter_bagging_classifier = BaggingClassifier(base_estimator=dt)
parameters = {
'bootstrap_features': [True,False],
'max_features': [5,10,15],
'max_samples': [80,100,120],
'n_estimators': [50,75,100],
}
#Use GridSearchCV to get the best parameters that will give the best accuracy
search = GridSearchCV(finding_best_parameter_bagging_classifier, parameters, cv=RepeatedKFold(n_splits=10,n_repea
ts=3,random_state=42))

#Fit that model on the dataset
best_model = search.fit(X_train,y_train.values.ravel())

#Print the best accuracy and the parameters
print('The best accuracy score from this model is:')
print(best_model.best_score_)
print('The best hyperparameter configuration is:')
print(best_model.best_params_)
```

```
The best accuracy score from this model is:
0.9572845804988662
The best hyperparameter configuration is:
{'bootstrap_features': True, 'max_features': 5, 'max_samples': 120, 'n_estimators': 50}
```

In [6]:

```
#Call the bagging classifier function with the best parameters
bagging_classifier = BaggingClassifier(base_estimator=dt,max_features=5,max_samples=120,n_estimators=50,bootstrap
=True,random_state=42)

#Fit that model on the dataset
bagging_classifier.fit(X_train, y_train)

#Print the coefficient of determination R^2 of the predictions on training and testing dataset
#A high score indicates that the model has good accuracy
print("The score on training data is",bagging_classifier.score(X_train, y_train))
print("The score on testing data is",bagging_classifier.score(X_test, y_test))
```

```
The score on training data is 0.9834368530020704
The score on testing data is 0.9418604651162791

/N/u/sdabhane/Carbonate/.conda/envs/CV/lib/python3.8/site-packages/sklearn/utils/validation.py:73: D
ataConversionWarning: A column-vector y was passed when a 1d array was expected. Please change the s
hape of y to (n_samples, ), for example using ravel().
  return f(**kwargs)
```

In [7]:

```
#We will find the best parameters for adaboost by passing a range of parameters
#in the GridSearchCV function and using RepeatedKFold Cross Validation

finding_best_parameter_adaboost_classifier = AdaBoostClassifier(base_estimator=dt)

#Set the parameters
parameters = {
'learning_rate': [0.0005,0.005,0.05,0.5],
'n_estimators': [25, 75, 100]
}

#Use GridSearchCV to get the best parameters that will give the best accuracy
search = GridSearchCV(finding_best_parameter_adaboost_classifier, parameters, cv=RepeatedKFold(n_splits=10,n_repe
ats=3,random_state=42))

#Fit that model on the dataset
best_model = search.fit(X_train,y_train.values.ravel())

#Print the best accuracy and the parameters
print('The best accuracy score from this model is:')
print(best_model.best_score_)
print('The best hyperparameter configuration is:')
print(best_model.best_params_)
```

```
The best accuracy score from this model is:
0.9248157596371881
The best hyperparameter configuration is:
{'learning_rate': 0.005, 'n_estimators': 100}
```

In [8]:

```
#Call the AdaBoostClassifier with the best parameters set from the previous cell
adaboost_classifier = AdaBoostClassifier(base_estimator=dt,learning_rate=0.005,n_estimators=100,random_state=42)

#Fit that model on the dataset
adaboost_classifier.fit(X_train, y_train)

#Print the coefficient of determination R^2 of the predictions on training and testing dataset:
#A high score indicates that the model has good accuracy
print("The score on training data is",adaboost_classifier.score(X_train, y_train))
print("The score on testing data is",adaboost_classifier.score(X_test, y_test))
```

```
The score on training data is 1.0
The score on testing data is 0.9302325581395349

/N/u/sdabhane/Carbonate/.conda/envs/CV/lib/python3.8/site-packages/sklearn/utils/validation.py:73: D
ataConversionWarning: A column-vector y was passed when a 1d array was expected. Please change the s
hape of y to (n_samples, ), for example using ravel().
  return f(**kwargs)
```

As we can observe here:

1. The best performance of BaggingClassifier with Decision Tree as the base predictor is when the following parameters are set:

bootstrap_features=True, max_features=5, max_samples=120, n_estimators=50

The BaggingClassifier gives us an accuracy of **0.9572845804988662** for the above parameters

1. The best performance of AdaBoostClassifier with Decision Tree as the base predictor is when the following parameters are set:

learning_rate=0.005,n_estimators=100

The AdaBoostClassifier gives us an accuracy of **0.9248157596371881** for the above parameters

1. Based on the given best configurations for each ensemble method, BaggingClassifier performs better compared to AdaBoostClassifier on the breast cancer dataset as it gives better accuracy.

References used in this code:

[1]. https://stackoverflow.com/questions/48769682/how-do-i-convert-data-from-a-scikit-learn-bunch-object-to-a-pandas-dataframe (https://stackoverflow.com/questions/48769682/how-do-i-convert-data-from-a-scikit-learn-bunch-object-to-a-pandas-dataframe)

[2]. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html)

[3]. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html)

[4]. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RepeatedKFold.html (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RepeatedKFold.html)

[5]. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html)

[6]. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html (https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html)

[7]. https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html (https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html)

[8]. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html)

[9]. Discussed Question 1 and Question 3 with Mohit Dalvi.

In [ ]:

```
```

2. (20 points) Tensorflow vs pytorch on Google Trend. Write a brief summary including four highlights about what you have learned
Answer:

The Google Trends page for the comparison between the search terms “Tensorflow” and “pytorch” gives us some interesting insights into the interest in these search terms over the past 12 months. To denote “pytorch”, the page uses the colour blue, while the page uses the colour red to denote “Tensorflow”.

As seen in the graph of the month(time) vs interest, the interest in the term “pytorch” was much more popular as a search term compared to “Tensorflow”. The interest in “Tensorflow” was never above the interest in “pytorch” for even a single month. “pytorch” also achieved peak popularity in the week of Nov 7-13, 2021. Unlike “pytorch”, “Tensorflow never achieved peak popularity. It’s highest value in interest was the week of Nov 28-Dec 4, 2021, which was 69. Both the terms had the same popularity in the week of Sep 26-Oct 2, 2021. Both the search terms have a topsy-turvy graph. They have constantly shown alternating rise and fall in interest.

We can also see the interest by subregion in the United States on the Google Trend page for both search terms together. From the graph, we can see that 24 states have more interest in “pytorch” compared to “Tensorflow”. These states are highlighted in blue. 21 states have more interest in “Tensorflow” compared to “pytorch”. These states are highlighted in red. The states of Virginia and Texas have both shown equal interest in the search terms. The states of Vermont, West Virginia, North Dakota, Maine and Wyoming are grey because there was not enough interest or data about the interest in either of the terms to be shown on the map. The state of Pennsylvania has the most interest in “pytorch” compared to “Tensorflow” i.e. 68% to 32%. Massachusetts, California, New Mexico, and Oregon round out the top 5 for most interest in “pytorch”, in comparison to “Tensorflow”. On the other hand, the states of Nebraska, Montana, Idaho, Maine, and South Dakota show 100% interest in “Tensorflow” compared to “pytorch”. We can also see the interest by Metro and by city. Columbus, GA has the highest interest in “pytorch” with value 100. Lincoln & Hastings-Kearney,NE has the highest interest in “Tensorflow” with value 100 and 0 value for “pytorch”. The city of Hanover has the highest interest in “pytorch” with 100 interest value and 0 for “Tensorflow”. “Redwood City” has the highest interest in “Tensorflow” with 45 interest value. So we can see that interest by city is low for “Tensorflow”, with the highest interest value for it being less than the one for “pytorch” at the same place.

There is also a graph showing interest by subregion for both “pytorch” and “Tensorflow” individually. For “pytorch”, Massachusetts has the highest value of interest at 100. California is a close second with 97. We can also see the related queries for the terms as well. For “pytorch”, the top 5 rising related queries are “pytorch m1”, “nn.relu”, “torch.norm”, “l1 loss” and “google collab”. Most of these terms are modules and imports related to pytorch. The topmost related query to “pytorch” is “python”. The other top related queries are queries like “torch”, “pytorch tensor”, “pytorch tensor”, “install pytorch” and “github pytorch”.

In the interest by subregion graph for “Tensorflow”, California has the highest interest value of 100. Since Silicon Valley is based in California, many companies work on both “pytorch” and “Tensorflow” and hence the state has high interest in both terms. The state of Washington hasn interest value of 97 for “Tensorflow”. Seattle is a city in Washington state which has many software companies working with “Tensorflow”. The top 5 rising related queries for “Tensorflow” are “miniforge”, “tensorflow m1”, “xnnpack tensorflow”, “google tensorflow certification” and “tf.reduce_sum”. The word “m1” is there in both “pytorch” and “Tensorflow”. This is because “m1” is one of the models of the Apple Mac laptop. The topmost related query for “Tensorflow” is “tensorflow python”, with 100 value, indicating that this search term is associated the most with “Tensorflow”. The rest of the top 5 top related queries include “python”, “keras”, “tensorflow keras” and “install tensorflow”.

Overall, the average interest in “pytorch” is 70 while the average interest for “Tensorflow” is 54. The interest in “pytorch” is much more compared to “Tensorflow” as it is a more widely used framework. The areas where the IT industry has a bigger presence has more interest for these search terms. We can also download all of this data in the form of CSV files. We can also share the whole webpage on social media sites like Twitter and Facebook. We can also choose to give feedback for this page and include screenshots along with our feedback.

1. (40 points) Using ANN and CNN.

• Read about this tutorial on Tensorflow and examine the provided code for classifying images of clothing (keras.datasets.fashion mnist) using an ANN.

• How many neurons does the hidden layer have in the given ANN?

• Try different numbers of neurons and report how the results change. Also try dropout (with different values) and report its impacts on the performance of the model. You may run the code in google colab and experiment with different settings there.

• Try to implement a CNN based on the given ANN, by adding two convolution layers before the fully connected hidden layer and test different settings (e.g., number/size of filters). Summarize the experiments you have tried and results you get.

• Here are some hints about using CNN:

import Conv2D

from keras.layers import Conv2D

create a model

model = keras.Sequential()

add a convolution layer with 32 filters of 3 × 3

model.add(keras.layers.Conv2D(filters=32, kernel size=(3, 3), · · ·)

Include additional parameters, input shape = (28, 28, 1), data format="channels last")

in Conv2D().

• Finally, because the images used in this example (fashion mnist) are in gray scales (of 28 × 28 pixels), the image data needs to be reformatted to be used as input to the convolution layer, e.g., train images = tf.reshape(train images, shape=[-1, 28, 28, 1])

In [1]:

```
#We will implement the tutorial first and see the results

# TensorFlow and tf.keras
import tensorflow as tf

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt

print(tf.__version__)
```

```
2.8.0
```

In [2]:

```
#Load the dataset and split into training and testing part
fashion_mnist = tf.keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

In [3]:

```
#Set the class names

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

In [4]:

```
#Print the shape of training images dataset
train_images.shape
```

Out[4]:

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

In [5]:

```
#Print the number of labels in the training dataset
len(train_labels)
```

Out[5]:

```
60000
```

In [6]:

```
#Print the labels of the training dataset
train_labels
```

Out[6]:

```
array([9, 0, 0, ..., 3, 0, 5], dtype=uint8)
```

In [7]:

```
#Print the shape of test images dataset
test_images.shape
```

Out[7]:

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

In [8]:

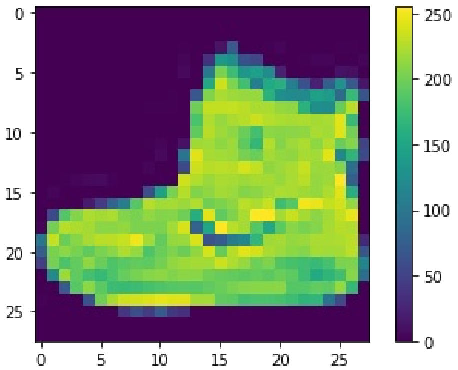
```
#Print the number of labels in the test dataset

len(test_labels)
```

Out[8]:

```
10000
```

In [9]:

```
#Perform preprocessing of the dataset
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```



In [10]:

```
#Scale the values of pixels to between 0 and 1
train_images = train_images / 255.0

test_images = test_images / 255.0
```

In [11]:

```
#Display first 25 images of the dataset to check if the dataset is ready for training
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], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

In [12]:

```
#We will now set up the layers for the ANN.

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])
```

In [13]:

```
#Compile the model
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

In [14]:

```
#Train the model
model.fit(train_images, train_labels, epochs=10)
```

```
Epoch 1/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4984 - accuracy: 0.8263
Epoch 2/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3805 - accuracy: 0.8629
Epoch 3/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3409 - accuracy: 0.8750
Epoch 4/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3160 - accuracy: 0.8839
Epoch 5/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.2966 - accuracy: 0.8904
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2822 - accuracy: 0.8949
Epoch 7/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.2691 - accuracy: 0.8997
Epoch 8/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.2569 - accuracy: 0.9041
Epoch 9/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.2501 - accuracy: 0.9055
Epoch 10/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.2405 - accuracy: 0.9102
```

Out[14]:

```
<keras.callbacks.History at 0x7f96419b8d00>
```

In [15]:

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

print('\nTest accuracy:', test_acc)
```

```
313/313 - 0s - loss: 0.3347 - accuracy: 0.8829 - 496ms/epoch - 2ms/step

Test accuracy: 0.8828999996185303
```

In [16]:

```
#Make predictions. Before that, convert logit values to probabilities using softmax for easier calculations
probability_model = tf.keras.Sequential([model,
                                         tf.keras.layers.Softmax()])
```

In [17]:

```
#Make predictions
predictions = probability_model.predict(test_images)
```

In [18]:

```
#Print the array of confidence for each class for a single image
predictions[0]
```

Out[18]:

```
array([2.1698716e-08, 4.4075559e-12, 5.8617888e-10, 1.2514323e-09,
       1.4771760e-10, 2.6503426e-04, 3.5768293e-09, 4.1003162e-03,
       3.0115501e-09, 9.9563462e-01], dtype=float32)
```

In [19]:

```
#Find the class with the highest confidence
np.argmax(predictions[0])
```

Out[19]:

```
9
```

In [20]:

```
#Verify if given result is correct or not
test_labels[0]
```

Out[20]:

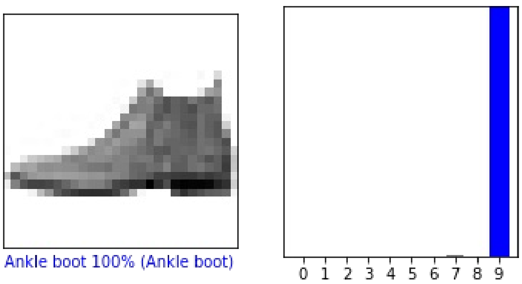
```
9
```

In [21]:

```python
#Plot the results of predictions for first 10 images
def plot_image(i, predictions_array, true_label, img):
    true_label, img = true_label[i], img[i]
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])

    plt.imshow(img, cmap=plt.cm.binary)

    predicted_label = np.argmax(predictions_array)
    if predicted_label == true_label:
        color = 'blue'
    else:
        color = 'red'

    plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                100*np.max(predictions_array),
                                class_names[true_label]),
                                color=color)

def plot_value_array(i, predictions_array, true_label):
    true_label = true_label[i]
    plt.grid(False)
    plt.xticks(range(10))
    plt.yticks([])
    thisplot = plt.bar(range(10), predictions_array, color="#777777")
    plt.ylim([0, 1])
    predicted_label = np.argmax(predictions_array)
    thisplot[predicted_label].set_color('red')
    thisplot[true_label].set_color('blue')
```

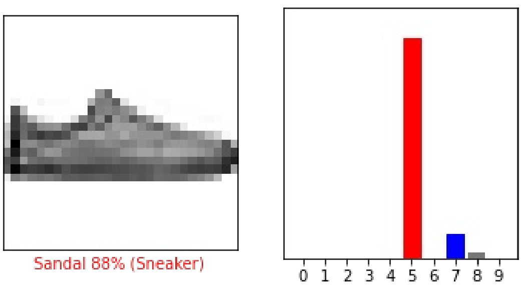
In [22]:

```python
#Verify results
i = 0
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i],  test_labels)
plt.show()
```



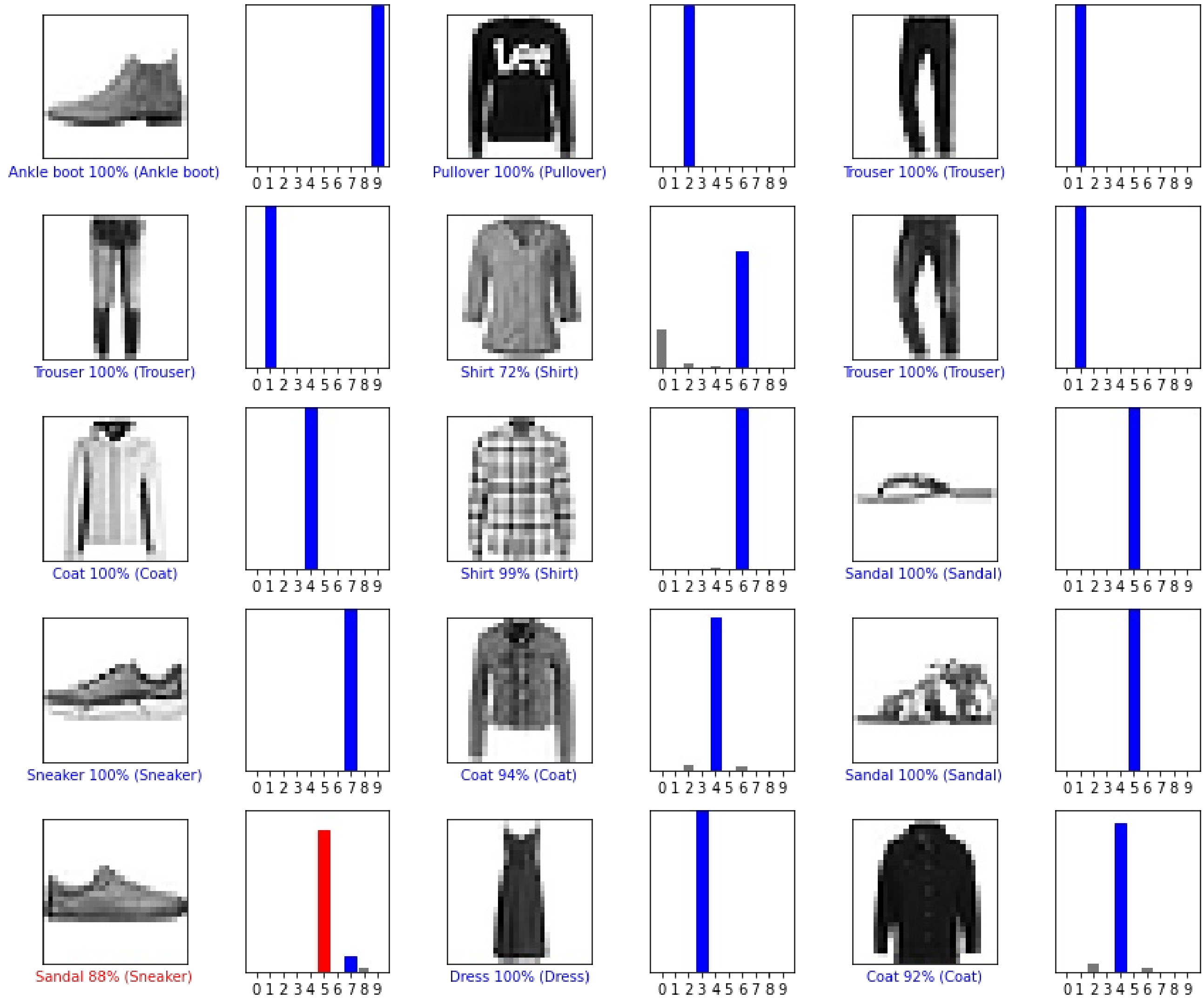
In [23]:

```python
i = 12
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i],  test_labels)
plt.show()
```



In [24]:

```
# Plot the first X test images, their predicted labels, and the true labels.
# Color correct predictions in blue and incorrect predictions in red.
num_rows = 5
num_cols = 3
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plot_image(i, predictions[i], test_labels, test_images)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, predictions[i], test_labels)
plt.tight_layout()
plt.show()
```



In [25]:

```
#Use the trained model
# Grab an image from the test dataset.
img = test_images[1]

print(img.shape)
```

```
(28, 28)
```

In [26]:

```
# Add the image to a batch where it's the only member.
img = (np.expand_dims(img,0))

print(img.shape)
```

```
(1, 28, 28)
```

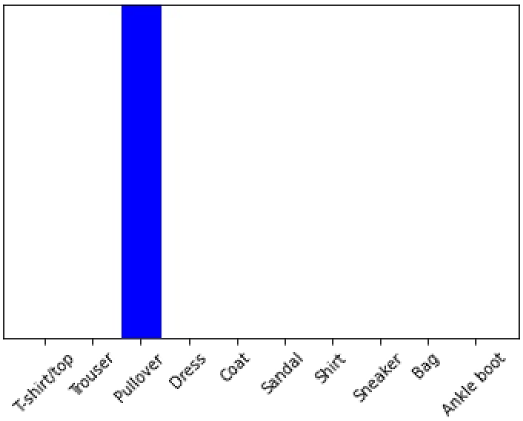
In [27]:

```
#Predict label for this image
predictions_single = probability_model.predict(img)

print(predictions_single)
```

```
[[4.6811168e-05 2.9821896e-14 9.9794811e-01 1.7063621e-10 5.5254676e-04
  3.8682642e-12 1.4525260e-03 1.1965530e-16 6.9279527e-11 1.1757728e-13]]
```

In [28]:

```
plot_value_array(1, predictions_single[0], test_labels)
_ = plt.xticks(range(10), class_names, rotation=45)
plt.show()
```



In [29]:

```
np.argmax(predictions_single[0])
```

Out[29]:

2

Observations:

1. The number of neurons in the hidden layer have for the given ANN is 128.

2. The results of experiments with trying different number of neurons and dropout rates are in subsequent notebooks

3. For the self designed CNN, will do the following experiments: change number of filters and size of filters.

In [ ]:

Implementation of notebook with different number of neurons in the hidden layer

In [1]:

```
# TensorFlow and tf.keras
import tensorflow as tf

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt

print(tf.__version__)
```

```
2.8.0
```

In [2]:

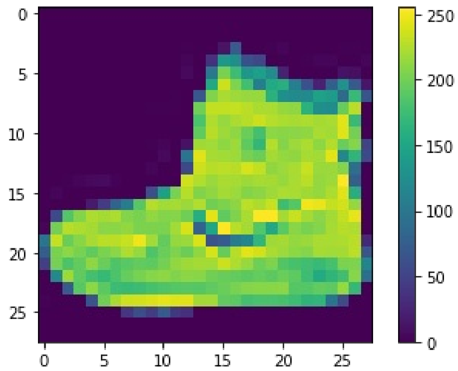
```
#Load the dataset and split into training and testing part
fashion_mnist = tf.keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

In [3]:

```
#Set the class names

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

In [4]:

```
#Perform preprocessing of the dataset
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```



In [5]:

```
#Scale the values of pixels to between 0 and 1
train_images = train_images / 255.0

test_images = test_images / 255.0
```

In [6]:

```
#Display first 25 images of the dataset to check if the dataset is ready for training
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], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

In [7]:

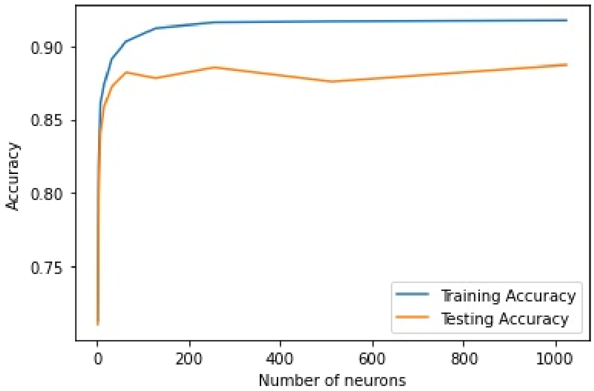
```
#We will now set up the layers for the ANN. We will different number of neurons and plot the accuracy for trainin
g and testing

training_accuracies=[]
testing_accuracies=[]
number_of_neurons = [2,4,8,16,32,64,128,256,512,1024]

#Set number of neurons from the list of number of neurons
for neuron in number_of_neurons:
    #Make the model by adding layers
    model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(neuron, activation='relu'),
    tf.keras.layers.Dense(10)
])


    #Compile the model
    model.compile(optimizer='adam',loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),metrics=[
'accuracy'])

    #Train the model
    final_model=model.fit(train_images, train_labels, epochs=10)

    #Evaluate accuarcy of model
    test_loss, test_accuracy = model.evaluate(test_images,  test_labels, verbose=2)

    training_accuracies.append(final_model.history['accuracy'][-1])
    testing_accuracies.append(test_accuracy)

plt.figure()
plt.plot(number_of_neurons, training_accuracies,label="Training Accuracy")
plt.plot(number_of_neurons, testing_accuracies,label="Testing Accuracy")
plt.xlabel("Number of neurons")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```

```
Epoch 1/10
1875/1875 [==============================] - 3s 1ms/step - loss: 1.5091 - accuracy: 0.4135
Epoch 2/10
1875/1875 [==============================] - 2s 1ms/step - loss: 1.1241 - accuracy: 0.5595
Epoch 3/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.9585 - accuracy: 0.6151
Epoch 4/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.8873 - accuracy: 0.6478
Epoch 5/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.8140 - accuracy: 0.6973
Epoch 6/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.7922 - accuracy: 0.7041
Epoch 7/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.7811 - accuracy: 0.7071
Epoch 8/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.7728 - accuracy: 0.7093
Epoch 9/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.7682 - accuracy: 0.7115
Epoch 10/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.7624 - accuracy: 0.7132
313/313 - 0s - loss: 0.7903 - accuracy: 0.7104 - 460ms/epoch - 1ms/step
Epoch 1/10
1875/1875 [==============================] - 3s 1ms/step - loss: 1.0434 - accuracy: 0.6625
Epoch 2/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.6825 - accuracy: 0.7814
Epoch 3/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.6151 - accuracy: 0.7941
Epoch 4/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.5816 - accuracy: 0.8024
Epoch 5/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.5626 - accuracy: 0.8066
Epoch 6/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.5503 - accuracy: 0.8100
Epoch 7/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.5392 - accuracy: 0.8123
Epoch 8/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.5332 - accuracy: 0.8156
Epoch 9/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.5282 - accuracy: 0.8174
Epoch 10/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.5238 - accuracy: 0.8187
313/313 - 0s - loss: 0.5642 - accuracy: 0.8022 - 414ms/epoch - 1ms/step
Epoch 1/10
```

```
1875/1875 [==============================] - 3s 1ms/step - loss: 0.7212 - accuracy: 0.7595
Epoch 2/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.4724 - accuracy: 0.8379
Epoch 3/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.4477 - accuracy: 0.8463
Epoch 4/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.4347 - accuracy: 0.8486
Epoch 5/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.4238 - accuracy: 0.8532
Epoch 6/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.4163 - accuracy: 0.8550
Epoch 7/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.4116 - accuracy: 0.8569
Epoch 8/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.4067 - accuracy: 0.8573
Epoch 9/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.4017 - accuracy: 0.8605
Epoch 10/10
1875/1875 [==============================] - 2s 1ms/step - loss: 0.3987 - accuracy: 0.8610
313/313 - 0s - loss: 0.4607 - accuracy: 0.8406 - 396ms/epoch - 1ms/step
Epoch 1/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.5826 - accuracy: 0.8024
Epoch 2/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.4398 - accuracy: 0.8451
Epoch 3/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.4137 - accuracy: 0.8542
Epoch 4/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3945 - accuracy: 0.8607
Epoch 5/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3867 - accuracy: 0.8621
Epoch 6/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3757 - accuracy: 0.8663
Epoch 7/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3686 - accuracy: 0.8696
Epoch 8/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3639 - accuracy: 0.8692
Epoch 9/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3553 - accuracy: 0.8727
Epoch 10/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3521 - accuracy: 0.8739
313/313 - 0s - loss: 0.4057 - accuracy: 0.8587 - 446ms/epoch - 1ms/step
Epoch 1/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.5564 - accuracy: 0.8129
Epoch 2/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.4197 - accuracy: 0.8540
Epoch 3/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3877 - accuracy: 0.8630
Epoch 4/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3645 - accuracy: 0.8706
Epoch 5/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3484 - accuracy: 0.8758
Epoch 6/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3345 - accuracy: 0.8801
Epoch 7/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3241 - accuracy: 0.8838
Epoch 8/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3155 - accuracy: 0.8854
Epoch 9/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3096 - accuracy: 0.8870
Epoch 10/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3026 - accuracy: 0.8911
313/313 - 0s - loss: 0.3673 - accuracy: 0.8722 - 430ms/epoch - 1ms/step
Epoch 1/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.5220 - accuracy: 0.8188
Epoch 2/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3921 - accuracy: 0.8624
Epoch 3/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3542 - accuracy: 0.8732
Epoch 4/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3315 - accuracy: 0.8808
Epoch 5/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3149 - accuracy: 0.8864
Epoch 6/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.2990 - accuracy: 0.8906
Epoch 7/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.2870 - accuracy: 0.8947
Epoch 8/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.2776 - accuracy: 0.8974
Epoch 9/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.2690 - accuracy: 0.9012
Epoch 10/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.2629 - accuracy: 0.9032
313/313 - 0s - loss: 0.3350 - accuracy: 0.8821 - 438ms/epoch - 1ms/step
```

```
Epoch 1/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4976 - accuracy: 0.8262
Epoch 2/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3728 - accuracy: 0.8660
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3360 - accuracy: 0.8758
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3110 - accuracy: 0.8858
Epoch 5/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2923 - accuracy: 0.8919
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2779 - accuracy: 0.8973
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2657 - accuracy: 0.9019
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2548 - accuracy: 0.9047
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2439 - accuracy: 0.9093
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2350 - accuracy: 0.9122
313/313 - 0s - loss: 0.3317 - accuracy: 0.8781 - 437ms/epoch - 1ms/step
Epoch 1/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4843 - accuracy: 0.8281
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3647 - accuracy: 0.8671
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3252 - accuracy: 0.8801
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2999 - accuracy: 0.8899
Epoch 5/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2826 - accuracy: 0.8950
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2680 - accuracy: 0.9006
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2561 - accuracy: 0.9040
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2442 - accuracy: 0.9088
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2334 - accuracy: 0.9131
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2247 - accuracy: 0.9160
313/313 - 0s - loss: 0.3380 - accuracy: 0.8856 - 464ms/epoch - 1ms/step
Epoch 1/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4718 - accuracy: 0.8312
Epoch 2/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3587 - accuracy: 0.8687
Epoch 3/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3189 - accuracy: 0.8823
Epoch 4/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2968 - accuracy: 0.8899
Epoch 5/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2792 - accuracy: 0.8956
Epoch 6/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2642 - accuracy: 0.9015
Epoch 7/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2517 - accuracy: 0.9067
Epoch 8/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2413 - accuracy: 0.9093
Epoch 9/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2303 - accuracy: 0.9143
Epoch 10/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2206 - accuracy: 0.9167
313/313 - 1s - loss: 0.3429 - accuracy: 0.8757 - 576ms/epoch - 2ms/step
Epoch 1/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4702 - accuracy: 0.8303
Epoch 2/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3560 - accuracy: 0.8694
Epoch 3/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3211 - accuracy: 0.8815
Epoch 4/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2959 - accuracy: 0.8906
Epoch 5/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2777 - accuracy: 0.8956
Epoch 6/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2636 - accuracy: 0.9015
Epoch 7/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2476 - accuracy: 0.9074
Epoch 8/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2397 - accuracy: 0.9108
Epoch 9/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2279 - accuracy: 0.9145
Epoch 10/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2213 - accuracy: 0.9174
```

```
313/313 - 1s - loss: 0.3321 - accuracy: 0.8872 - 657ms/epoch - 2ms/step
```



As we can see from the above graph, with the increase in the number of neurons in the hidden layer, the training and testing accuracy increases.

In [ ]:

Implementation of notebook with different dropout rates in the hidden layer

In [1]:

```
#We will implement the tutorial first and see the results

# TensorFlow and tf.keras
import tensorflow as tf

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt

print(tf.__version__)
```

2.8.0

In [2]:

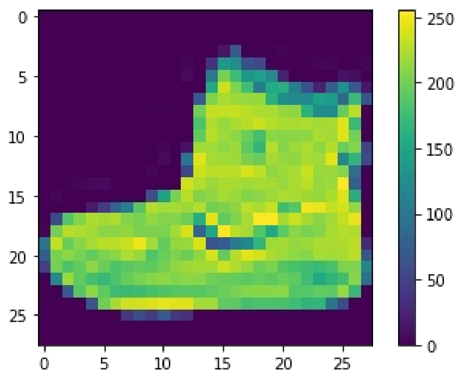
```
#Load the dataset and split into training and testing part
fashion_mnist = tf.keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

In [3]:

```
#Set the class names

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

In [4]:

```
#Perform preprocessing of the dataset
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```



In [5]:

```
#Scale the values of pixels to between 0 and 1
train_images = train_images / 255.0

test_images = test_images / 255.0
```

In [6]:

```
#Display first 25 images of the dataset to check if the dataset is ready for training
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], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

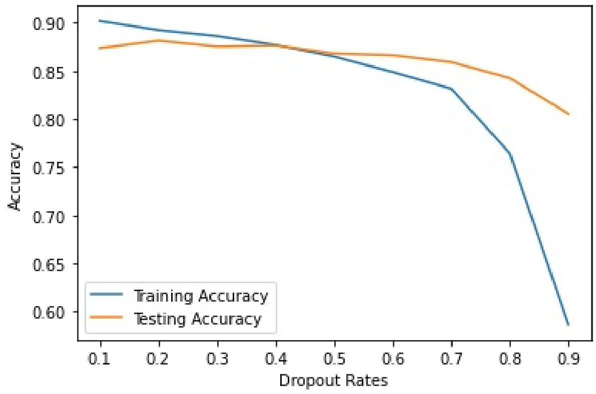
In [7]:

```
#We will now set up the layers for the ANN. We will different number of neurons and plot the accuracy for trainin
g and testing

training_accuracies=[]
testing_accuracies=[]
dropout_rates=[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]

#Set the dropout rate from the list of dropout rates
for each_rate in dropout_rates:
    #Make the model by adding layers
    model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(rate=each_rate),
    tf.keras.layers.Dense(10)
])


    #Compile the model
    model.compile(optimizer='adam',loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),metrics=[
'accuracy'])

    #Train the model
    final_model=model.fit(train_images, train_labels, epochs=10)

    #Evaluate accuarcy of model
    test_loss, test_accuracy = model.evaluate(test_images,  test_labels, verbose=2)

    training_accuracies.append(final_model.history['accuracy'][-1])
    testing_accuracies.append(test_accuracy)

plt.figure()
plt.plot(dropout_rates, training_accuracies,label="Training Accuracy")
plt.plot(dropout_rates, testing_accuracies,label="Testing Accuracy")
plt.xlabel("Dropout Rates")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```

```
Epoch 1/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.5129 - accuracy: 0.8179
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3850 - accuracy: 0.8587
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3502 - accuracy: 0.8729
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3293 - accuracy: 0.8801
Epoch 5/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3106 - accuracy: 0.8853
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3004 - accuracy: 0.8882
Epoch 7/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.2891 - accuracy: 0.8922
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2783 - accuracy: 0.8961
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2689 - accuracy: 0.9005
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2621 - accuracy: 0.9020
313/313 - 1s - loss: 0.3436 - accuracy: 0.8737 - 514ms/epoch - 2ms/step
Epoch 1/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.5291 - accuracy: 0.8138
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4032 - accuracy: 0.8547
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3654 - accuracy: 0.8656
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3440 - accuracy: 0.8747
Epoch 5/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3295 - accuracy: 0.8786
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3194 - accuracy: 0.8819
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3070 - accuracy: 0.8860
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2976 - accuracy: 0.8895
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2924 - accuracy: 0.8915
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.2847 - accuracy: 0.8925
313/313 - 0s - loss: 0.3359 - accuracy: 0.8815 - 475ms/epoch - 2ms/step
```

```
Epoch 1/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.5577 - accuracy: 0.8026
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4190 - accuracy: 0.8479
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3855 - accuracy: 0.8596
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3662 - accuracy: 0.8668
Epoch 5/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3512 - accuracy: 0.8726
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3405 - accuracy: 0.8741
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3297 - accuracy: 0.8778
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3203 - accuracy: 0.8813
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3132 - accuracy: 0.8835
Epoch 10/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3067 - accuracy: 0.8863
313/313 - 0s - loss: 0.3572 - accuracy: 0.8755 - 489ms/epoch - 2ms/step
Epoch 1/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.5819 - accuracy: 0.7916
Epoch 2/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.4408 - accuracy: 0.8406
Epoch 3/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.4098 - accuracy: 0.8506
Epoch 4/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3891 - accuracy: 0.8582
Epoch 5/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3764 - accuracy: 0.8632
Epoch 6/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3610 - accuracy: 0.8674
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3600 - accuracy: 0.8669
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3482 - accuracy: 0.8715
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3407 - accuracy: 0.8740
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3333 - accuracy: 0.8771
313/313 - 0s - loss: 0.3527 - accuracy: 0.8766 - 455ms/epoch - 1ms/step
Epoch 1/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.6135 - accuracy: 0.7819
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4702 - accuracy: 0.8300
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4364 - accuracy: 0.8414
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4235 - accuracy: 0.8450
Epoch 5/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4059 - accuracy: 0.8523
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3925 - accuracy: 0.8548
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3864 - accuracy: 0.8593
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3814 - accuracy: 0.8592
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3717 - accuracy: 0.8644
Epoch 10/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.3661 - accuracy: 0.8650
313/313 - 0s - loss: 0.3619 - accuracy: 0.8681 - 479ms/epoch - 2ms/step
Epoch 1/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6713 - accuracy: 0.7616
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.5179 - accuracy: 0.8132
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4858 - accuracy: 0.8263
Epoch 4/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.4649 - accuracy: 0.8334
Epoch 5/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4527 - accuracy: 0.8357
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4416 - accuracy: 0.8403
Epoch 7/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.4319 - accuracy: 0.8448
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4219 - accuracy: 0.8464
Epoch 9/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.4192 - accuracy: 0.8479
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4098 - accuracy: 0.8489
```

```
313/313 - 0s - loss: 0.3772 - accuracy: 0.8664 - 455ms/epoch - 1ms/step
Epoch 1/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.7499 - accuracy: 0.7329
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.5750 - accuracy: 0.7933
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.5387 - accuracy: 0.8041
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.5215 - accuracy: 0.8097
Epoch 5/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.5037 - accuracy: 0.8145
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4970 - accuracy: 0.8191
Epoch 7/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.4860 - accuracy: 0.8205
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4777 - accuracy: 0.8250
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4717 - accuracy: 0.8288
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4661 - accuracy: 0.8316
313/313 - 0s - loss: 0.3953 - accuracy: 0.8593 - 453ms/epoch - 1ms/step
Epoch 1/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.9403 - accuracy: 0.6513
Epoch 2/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.7413 - accuracy: 0.7187
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.7019 - accuracy: 0.7319
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6801 - accuracy: 0.7452
Epoch 5/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.6650 - accuracy: 0.7505
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6520 - accuracy: 0.7531
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6432 - accuracy: 0.7566
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6365 - accuracy: 0.7579
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6258 - accuracy: 0.7630
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6159 - accuracy: 0.7644
313/313 - 0s - loss: 0.4446 - accuracy: 0.8426 - 437ms/epoch - 1ms/step
Epoch 1/10
1875/1875 [==============================] - 3s 2ms/step - loss: 1.3989 - accuracy: 0.4496
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 1.1724 - accuracy: 0.5201
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 1.1174 - accuracy: 0.5380
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 1.0914 - accuracy: 0.5493
Epoch 5/10
1875/1875 [==============================] - 3s 2ms/step - loss: 1.0608 - accuracy: 0.5645
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 1.0505 - accuracy: 0.5712
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 1.0364 - accuracy: 0.5749
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 1.0351 - accuracy: 0.5804
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 1.0292 - accuracy: 0.5842
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 1.0205 - accuracy: 0.5862
313/313 - 0s - loss: 0.5760 - accuracy: 0.8055 - 463ms/epoch - 1ms/step
```



As we can see from the above graph, with the increase in the dropout rate, the training and testing accuracy decreases.

In [ ]:

```

```

We will run code with CNN here with number of filters=32 and filter size=(3,3)

In [1]:

```
# TensorFlow and tf.keras
import tensorflow as tf

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt

print(tf.__version__)
```

```
2.8.0
```

In [2]:

```
#Load the dataset and split into training and testing part
fashion_mnist = tf.keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

In [3]:

```
#Set the class names

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

In [4]:

```
#Print the shape of training images dataset
train_images.shape
```

Out[4]:

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

In [5]:

```
#Print the number of labels in the training dataset
len(train_labels)
```

Out[5]:

```
60000
```

In [6]:

```
#Print the labels of the training dataset
train_labels
```

Out[6]:

```
array([9, 0, 0, ..., 3, 0, 5], dtype=uint8)
```

In [7]:

```
#Print the shape of test images dataset
test_images.shape
```

Out[7]:

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

In [8]:

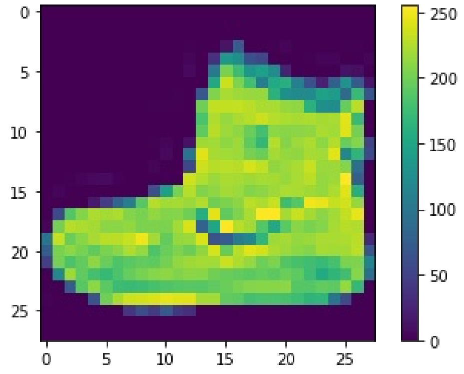
```
#Print the number of labels in the test dataset

len(test_labels)
```

Out[8]:

```
10000
```

In [9]:

```
#Perform preprocessing of the dataset
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```



In [10]:

```
#Scale the values of pixels to between 0 and 1
train_images = train_images / 255.0

test_images = test_images / 255.0
```

In [11]:

```
#Display first 25 images of the dataset to check if the dataset is ready for training
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], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

In [12]:

```
#We will now set up the layers for the CNN. Set filters=32 with given requirement
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), input_shape = (28, 28, 1),activation="relu",data_format="channels_last"),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Conv2D(32, (3, 3), activation="relu",data_format="channels_last"),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])

train_images = tf.reshape(train_images, shape=[-1, 28, 28, 1])

model.summary()
model.compile(optimizer='adam', loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True), metrics=['a
ccuracy'])
model.fit(train_images, train_labels, epochs=10)
```

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

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

 conv2d_1 (Conv2D)           (None, 11, 11, 32)        9248

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

 flatten (Flatten)           (None, 800)               0

 dense (Dense)               (None, 128)               102528

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 113,386
Trainable params: 113,386
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
1875/1875 [==============================] - 13s 6ms/step - loss: 0.4709 - accuracy: 0.8276
Epoch 2/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.3180 - accuracy: 0.8845
Epoch 3/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.2728 - accuracy: 0.8985
Epoch 4/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.2424 - accuracy: 0.9103
Epoch 5/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.2162 - accuracy: 0.9198
Epoch 6/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.1966 - accuracy: 0.9262
Epoch 7/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.1776 - accuracy: 0.9330
Epoch 8/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.1616 - accuracy: 0.9399
Epoch 9/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.1471 - accuracy: 0.9449
Epoch 10/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.1350 - accuracy: 0.9489
```

Out[12]:

```
<keras.callbacks.History at 0x7f9708510880>
```

In [13]:

```
test_loss, test_acc = model.evaluate(tf.reshape(test_images,shape=[-1,28,28,1]),test_labels,verbose=2)
print('\nTest accuracy:', test_acc)
```

```
313/313 - 1s - loss: 0.2843 - accuracy: 0.9087 - 826ms/epoch - 3ms/step

Test accuracy: 0.9086999893188477
```

In [ ]:

We will run code with CNN here with number of filters=64 and filter size=(3,3)

In [1]:

```
# TensorFlow and tf.keras
import tensorflow as tf

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt

print(tf.__version__)
```

```
2.8.0
```

In [2]:

```
#Load the dataset and split into training and testing part
fashion_mnist = tf.keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

In [3]:

```
#Set the class names

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

In [4]:

```
#Print the shape of training images dataset
train_images.shape
```

Out[4]:

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

In [5]:

```
#Print the number of labels in the training dataset
len(train_labels)
```

Out[5]:

```
60000
```

In [6]:

```
#Print the labels of the training dataset
train_labels
```

Out[6]:

```
array([9, 0, 0, ..., 3, 0, 5], dtype=uint8)
```

In [7]:

```
#Print the shape of test images dataset
test_images.shape
```

Out[7]:

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

In [8]:

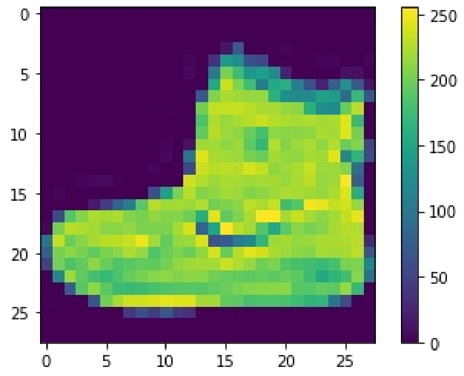
```
#Print the number of labels in the test dataset

len(test_labels)
```

Out[8]:

```
10000
```

In [9]:

```
#Perform preprocessing of the dataset
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```



In [10]:

```
#Scale the values of pixels to between 0 and 1
train_images = train_images / 255.0

test_images = test_images / 255.0
```

In [11]:

```
#Display first 25 images of the dataset to check if the dataset is ready for training
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], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

In [12]:

```
#We will now set up the layers for the CNN. Set filters=64 with given requirement
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, (3, 3), input_shape = (28, 28, 1),activation="relu",data_format="channels_last"),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Conv2D(64, (3, 3), activation="relu",data_format="channels_last"),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])

train_images = tf.reshape(train_images, shape=[-1, 28, 28, 1])

model.summary()
model.compile(optimizer='adam', loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True), metrics=['a
ccuracy'])
model.fit(train_images, train_labels, epochs=10)
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 64)        640

 max_pooling2d (MaxPooling2D  (None, 13, 13, 64)       0
 )

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        36928

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 flatten (Flatten)           (None, 1600)              0

 dense (Dense)               (None, 128)               204928

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 243,786
Trainable params: 243,786
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
1875/1875 [==============================] - 20s 10ms/step - loss: 0.4449 - accuracy: 0.8371
Epoch 2/10
1875/1875 [==============================] - 19s 10ms/step - loss: 0.2980 - accuracy: 0.8915
Epoch 3/10
1875/1875 [==============================] - 19s 10ms/step - loss: 0.2535 - accuracy: 0.9069
Epoch 4/10
1875/1875 [==============================] - 19s 10ms/step - loss: 0.2191 - accuracy: 0.9185
Epoch 5/10
1875/1875 [==============================] - 19s 10ms/step - loss: 0.1923 - accuracy: 0.9269
Epoch 6/10
1875/1875 [==============================] - 19s 10ms/step - loss: 0.1689 - accuracy: 0.9357
Epoch 7/10
1875/1875 [==============================] - 20s 10ms/step - loss: 0.1493 - accuracy: 0.9437
Epoch 8/10
1875/1875 [==============================] - 20s 11ms/step - loss: 0.1309 - accuracy: 0.9508
Epoch 9/10
1875/1875 [==============================] - 20s 11ms/step - loss: 0.1159 - accuracy: 0.9565
Epoch 10/10
1875/1875 [==============================] - 20s 10ms/step - loss: 0.1016 - accuracy: 0.9617
```

Out[12]:

```
<keras.callbacks.History at 0x7f512f6d3970>
```

In [13]:

```
test_loss, test_acc = model.evaluate(tf.reshape(test_images,shape=[-1,28,28,1]),test_labels,verbose=2)
print('\nTest accuracy:', test_acc)
```

```
313/313 - 1s - loss: 0.3191 - accuracy: 0.9094 - 1s/epoch - 4ms/step

Test accuracy: 0.9093999862670898
```

We will run code with CNN here with number of filters=32 and filter size=(5,5)

In [1]:

```
# TensorFlow and tf.keras
import tensorflow as tf

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt

print(tf.__version__)
```

```
2.8.0
```

In [2]:

```
#Load the dataset and split into training and testing part
fashion_mnist = tf.keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

In [3]:

```
#Set the class names

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

In [4]:

```
#Print the shape of training images dataset
train_images.shape
```

Out[4]:

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

In [5]:

```
#Print the number of labels in the training dataset
len(train_labels)
```

Out[5]:

```
60000
```

In [6]:

```
#Print the labels of the training dataset
train_labels
```

Out[6]:

```
array([9, 0, 0, ..., 3, 0, 5], dtype=uint8)
```

In [7]:

```
#Print the shape of test images dataset
test_images.shape
```

Out[7]:

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

In [8]:

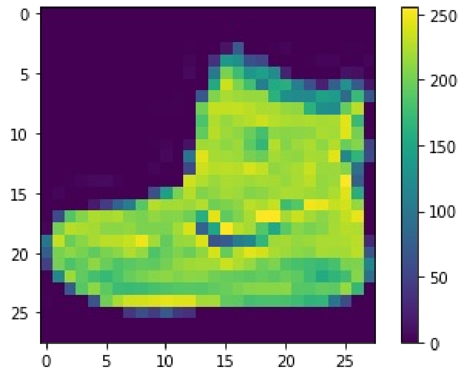
```
#Print the number of labels in the test dataset

len(test_labels)
```

Out[8]:

```
10000
```

In [9]:

```
#Perform preprocessing of the dataset
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```



In [10]:

```
#Scale the values of pixels to between 0 and 1
train_images = train_images / 255.0

test_images = test_images / 255.0
```

In [11]:

```
#Display first 25 images of the dataset to check if the dataset is ready for training
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], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

In [12]:

```
#We will now set up the layers for the CNN. Set filters=32 and filter size=(5,5) with given requirement
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (5, 5), input_shape = (28, 28, 1),activation="relu",data_format="channels_last"),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Conv2D(32, (5, 5), activation="relu",data_format="channels_last"),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])

train_images = tf.reshape(train_images, shape=[-1, 28, 28, 1])

model.summary()
model.compile(optimizer='adam', loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True), metrics=['a
ccuracy'])
model.fit(train_images, train_labels, epochs=10)
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d (Conv2D)             (None, 24, 24, 32)        832       
                                                                 
 max_pooling2d (MaxPooling2D  (None, 12, 12, 32)       0         
 )                                                               
                                                                 
 conv2d_1 (Conv2D)           (None, 8, 8, 32)          25632     
                                                                 
 max_pooling2d_1 (MaxPooling  (None, 4, 4, 32)         0         
 2D)                                                             
                                                                 
 flatten (Flatten)           (None, 512)               0         
                                                                 
 dense (Dense)               (None, 128)               65664     
                                                                 
 dense_1 (Dense)             (None, 10)                1290      
                                                                 
=================================================================
Total params: 93,418
Trainable params: 93,418
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
1875/1875 [==============================] - 13s 6ms/step - loss: 0.4857 - accuracy: 0.8240
Epoch 2/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.3161 - accuracy: 0.8856
Epoch 3/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.2734 - accuracy: 0.9000
Epoch 4/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.2445 - accuracy: 0.9097
Epoch 5/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.2235 - accuracy: 0.9182
Epoch 6/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.2056 - accuracy: 0.9239
Epoch 7/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.1881 - accuracy: 0.9288
Epoch 8/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.1737 - accuracy: 0.9355
Epoch 9/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.1610 - accuracy: 0.9391
Epoch 10/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.1487 - accuracy: 0.9446
```

Out[12]:

```
<keras.callbacks.History at 0x7fbc3dc9bac0>
```

In [13]:

```
test_loss, test_acc = model.evaluate(tf.reshape(test_images,shape=[-1,28,28,1]),test_labels,verbose=2)
print('\nTest accuracy:', test_acc)
```

```
313/313 - 1s - loss: 0.2764 - accuracy: 0.9107 - 892ms/epoch - 3ms/step

Test accuracy: 0.9107000231742859
```

In [ ]:

We will run code with CNN here with number of filters=64 and filter size=(5,5)

In [1]:

```
# TensorFlow and tf.keras
import tensorflow as tf

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt

print(tf.__version__)
```

```
2.8.0
```

In [2]:

```
#Load the dataset and split into training and testing part
fashion_mnist = tf.keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()
```

In [3]:

```
#Set the class names

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

In [4]:

```
#Print the shape of training images dataset
train_images.shape
```

Out[4]:

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

In [5]:

```
#Print the number of labels in the training dataset
len(train_labels)
```

Out[5]:

```
60000
```

In [6]:

```
#Print the labels of the training dataset
train_labels
```

Out[6]:

```
array([9, 0, 0, ..., 3, 0, 5], dtype=uint8)
```

In [7]:

```
#Print the shape of test images dataset
test_images.shape
```

Out[7]:

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

In [8]:

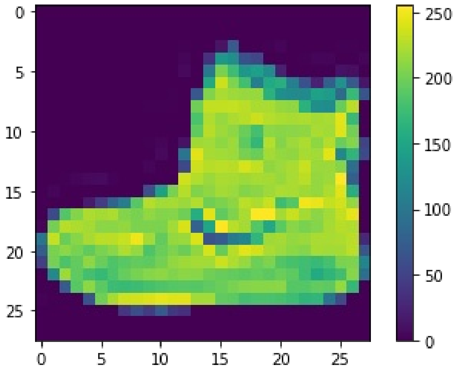
```
#Print the number of labels in the test dataset

len(test_labels)
```

Out[8]:

```
10000
```

In [9]:

```
#Perform preprocessing of the dataset
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```



In [10]:

```
#Scale the values of pixels to between 0 and 1
train_images = train_images / 255.0

test_images = test_images / 255.0
```

In [11]:

```
#Display first 25 images of the dataset to check if the dataset is ready for training
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], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

In [12]:

```
#We will now set up the layers for the CNN. Set filters=64 and filter size=(5,5) with given requirement
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, (5, 5), input_shape = (28, 28, 1),activation="relu",data_format="channels_last"),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Conv2D(64, (5, 5), activation="relu",data_format="channels_last"),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])

train_images = tf.reshape(train_images, shape=[-1, 28, 28, 1])

model.summary()
model.compile(optimizer='adam', loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True), metrics=['a
ccuracy'])
model.fit(train_images, train_labels, epochs=10)
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 24, 24, 64)        1664

 max_pooling2d (MaxPooling2D  (None, 12, 12, 64)       0
 )

 conv2d_1 (Conv2D)           (None, 8, 8, 64)          102464

 max_pooling2d_1 (MaxPooling  (None, 4, 4, 64)         0
 2D)

 flatten (Flatten)           (None, 1024)              0

 dense (Dense)               (None, 128)               131200

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 236,618
Trainable params: 236,618
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
1875/1875 [==============================] - 24s 12ms/step - loss: 0.4454 - accuracy: 0.8398
Epoch 2/10
1875/1875 [==============================] - 21s 11ms/step - loss: 0.2957 - accuracy: 0.8923
Epoch 3/10
1875/1875 [==============================] - 23s 12ms/step - loss: 0.2518 - accuracy: 0.9069
Epoch 4/10
1875/1875 [==============================] - 21s 11ms/step - loss: 0.2214 - accuracy: 0.9179
Epoch 5/10
1875/1875 [==============================] - 20s 11ms/step - loss: 0.1965 - accuracy: 0.9256
Epoch 6/10
1875/1875 [==============================] - 20s 11ms/step - loss: 0.1755 - accuracy: 0.9344
Epoch 7/10
1875/1875 [==============================] - 20s 11ms/step - loss: 0.1560 - accuracy: 0.9410
Epoch 8/10
1875/1875 [==============================] - 24s 13ms/step - loss: 0.1400 - accuracy: 0.9466
Epoch 9/10
1875/1875 [==============================] - 23s 12ms/step - loss: 0.1257 - accuracy: 0.9524
Epoch 10/10
1875/1875 [==============================] - 20s 11ms/step - loss: 0.1143 - accuracy: 0.9567
```

Out[12]:

```
<keras.callbacks.History at 0x7f751be06970>
```

In [13]:

```
test_loss, test_acc = model.evaluate(tf.reshape(test_images,shape=[-1,28,28,1]),test_labels,verbose=2)
print('\nTest accuracy:', test_acc)
```

```
313/313 - 1s - loss: 0.2860 - accuracy: 0.9145 - 1s/epoch - 4ms/step

Test accuracy: 0.9144999980926514
```

Observations:

1. The test accuracy when we set filters=32 for a filter size of (3,3) is **0.9086999893188477**.
2. The test accuracy when we set filters=64 for a filter size of (3,3) is **0.9093999862670898**
3. The test accuracy when we set filters=32 for a filter size of (5,5) is **0.9107000231742859**.
4. The test accuracy when we set filters=64 for a filter size of (5,5) is **0.9144999980926514**

The results of the experiments conclude that:

1. When we increase the filter size in the convolution layers, the accuracy of the model increased slightly.
2. When we increase the number of filters in the convolution layers, the accuracy of the model increased slightly.

In [ ]:

```
```