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 x 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_repeats=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: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Please change the shape 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_repeats=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__)
```

```
2.3.0
```

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]:

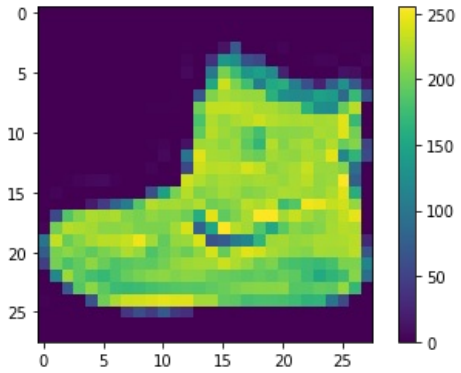
```
#Print the number of labels in the test dataset

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 [11]:

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

test_images = test_images / 255.0
```

In [12]:

```
#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 [13]:

```
#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 [15]:

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

In [16]:

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

```
Epoch 1/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4944 - accuracy: 0.8256
Epoch 2/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.3743 - accuracy: 0.8660
Epoch 3/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3354 - accuracy: 0.8785
Epoch 4/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3122 - accuracy: 0.8868
Epoch 5/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2946 - accuracy: 0.8914
Epoch 6/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2798 - accuracy: 0.8959
Epoch 7/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2690 - accuracy: 0.8993
Epoch 8/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2555 - accuracy: 0.9048
Epoch 9/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2476 - accuracy: 0.9070
Epoch 10/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2401 - accuracy: 0.9097
```

Out[16]:

```
<tensorflow.python.keras.callbacks.History at 0x1d86b334408>
```

In [17]:

```
#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: 0.3318 - accuracy: 0.8828

Test accuracy: 0.8827999830245972
```

In [20]:

```
#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 [22]:

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

In [23]:

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

Out[23]:

```
array([1.3223512e-06, 6.6183128e-11, 4.7320402e-08, 3.7670883e-08,
       3.5262861e-08, 4.0443847e-04, 4.5508304e-07, 5.1502712e-02,
       5.6752953e-07, 9.4809037e-01], dtype=float32)
```

In [24]:

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

Out[24]:

```
9
```

In [25]:

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

Out[25]:

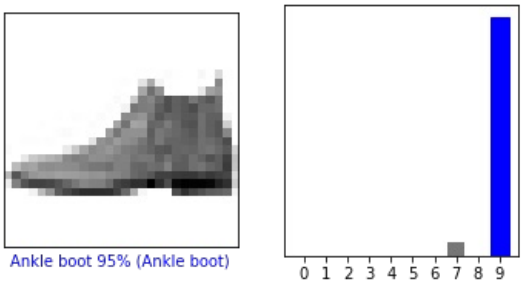
```
9
```

In [27]:

```
#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')
```

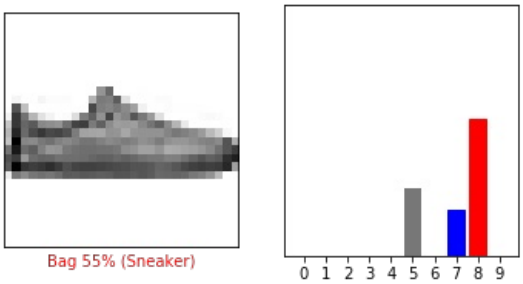
In [28]:

```
#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()
```



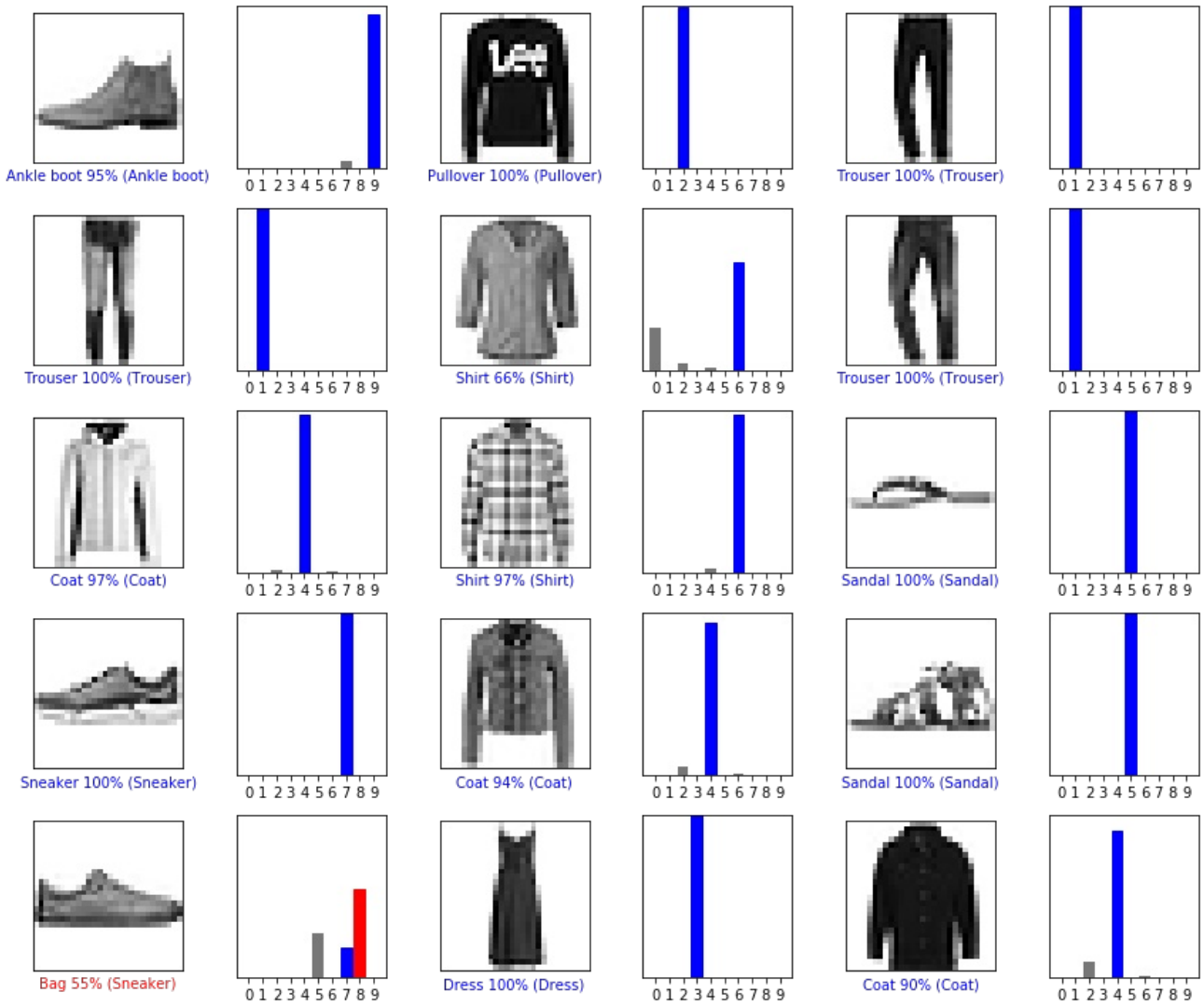
In [29]:

```
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 [30]:

```
# 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 [31]:

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

print(img.shape)
```

```
(28, 28)
```

In [32]:

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

print(img.shape)
```

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

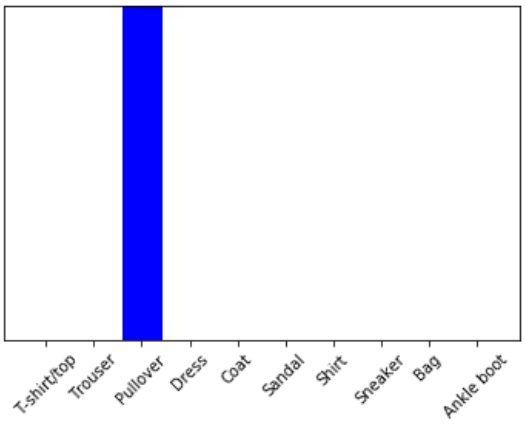
In [34]:

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

print(predictions_single)
```

```
[[1.5685402e-05 1.4652803e-14 9.9762076e-01 9.2181035e-10 1.8769290e-03
  1.3414909e-11 4.8667283e-04 3.9673660e-17 4.6024787e-10 3.0338679e-12]]
```

In [35]:

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



In [36]:

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

Out[36]:

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__)
```

```
2.3.0
```

In [2]:

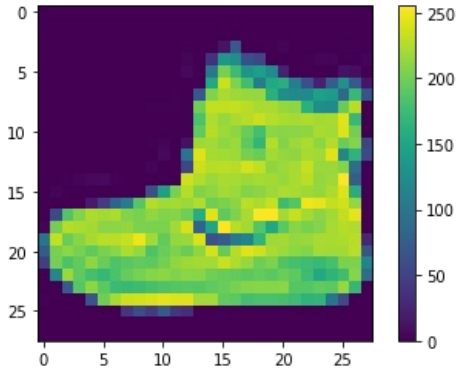
```
#Load the dataset and split into training and testing part
fashion_mnist = tf.keras.datasets.fashion_mnist

(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]:

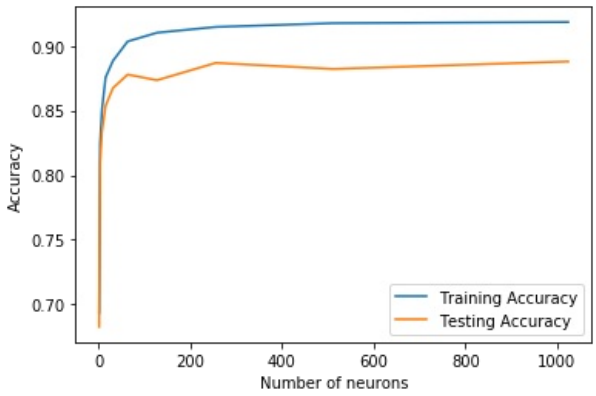
```
#We will now set up the layers for the ANN. We will different number of neurons and plot the accuracy for trainin
g and testing

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 2ms/step - loss: 1.6304 - accuracy: 0.4291
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 1.1997 - accuracy: 0.5922
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 1.0810 - accuracy: 0.6181
Epoch 4/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.0253 - accuracy: 0.6356
Epoch 5/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.9909 - accuracy: 0.6470
Epoch 6/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.9658 - accuracy: 0.6548
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.9454 - accuracy: 0.6635
Epoch 8/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.9306 - accuracy: 0.6737
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.9126 - accuracy: 0.6847
Epoch 10/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.8955 - accuracy: 0.6934
313/313 - 0s - loss: 0.9244 - accuracy: 0.6825
Epoch 1/10
1875/1875 [==============================] - 4s 2ms/step - loss: 1.2029 - accuracy: 0.5741
Epoch 2/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.6700 - accuracy: 0.7683
Epoch 3/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.5761 - accuracy: 0.7990
Epoch 4/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.5496 - accuracy: 0.8076
Epoch 5/10
1875/1875 [==============================] - 3s 1ms/step - loss: 0.5351 - accuracy: 0.8113
Epoch 6/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.5255 - accuracy: 0.8154
Epoch 7/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.5173 - accuracy: 0.8175
Epoch 8/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.5122 - accuracy: 0.8203
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.5062 - accuracy: 0.8231
Epoch 10/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.5013 - accuracy: 0.8242
313/313 - 1s - loss: 0.5418 - accuracy: 0.8096
Epoch 1/10
```

```
1875/1875 [==============================] - 4s 2ms/step - loss: 0.7731 - accuracy: 0.7311
Epoch 2/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.5076 - accuracy: 0.8224
Epoch 3/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4792 - accuracy: 0.8313: 0s - los
s: 0.4786 - accuracy:
Epoch 4/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4634 - accuracy: 0.8363
Epoch 5/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4533 - accuracy: 0.8401
Epoch 6/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.4466 - accuracy: 0.8426
Epoch 7/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4391 - accuracy: 0.8451
Epoch 8/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4337 - accuracy: 0.8470
Epoch 9/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4293 - accuracy: 0.8493
Epoch 10/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.4267 - accuracy: 0.8505
313/313 - 1s - loss: 0.4761 - accuracy: 0.8335
Epoch 1/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.5812 - accuracy: 0.7994
Epoch 2/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4383 - accuracy: 0.8481
Epoch 3/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.4115 - accuracy: 0.8565
Epoch 4/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3965 - accuracy: 0.8621
Epoch 5/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3864 - accuracy: 0.8631
Epoch 6/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3756 - accuracy: 0.8674
Epoch 7/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3676 - accuracy: 0.8701
Epoch 8/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3612 - accuracy: 0.8717
Epoch 9/10
1875/1875 [==============================] - 3s 2ms/step - loss: 0.3540 - accuracy: 0.8740
Epoch 10/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3492 - accuracy: 0.8757
313/313 - 1s - loss: 0.4148 - accuracy: 0.8537
Epoch 1/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.5415 - accuracy: 0.8132
Epoch 2/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.4171 - accuracy: 0.8520
Epoch 3/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3848 - accuracy: 0.8619
Epoch 4/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3674 - accuracy: 0.8693
Epoch 5/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.3520 - accuracy: 0.8732
Epoch 6/10
1875/1875 [==============================] - 10s 6ms/step - loss: 0.3413 - accuracy: 0.8772
Epoch 7/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.3289 - accuracy: 0.8813
Epoch 8/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.3189 - accuracy: 0.8839
Epoch 9/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3127 - accuracy: 0.8875
Epoch 10/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.3051 - accuracy: 0.8885
313/313 - 1s - loss: 0.3661 - accuracy: 0.8674
Epoch 1/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.5104 - accuracy: 0.8206
Epoch 2/10
1875/1875 [==============================] - ETA: 0s - loss: 0.3888 - accuracy: 0.86 - 7s 4ms/step -
loss: 0.3887 - accuracy: 0.8609
Epoch 3/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3527 - accuracy: 0.8716
Epoch 4/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.3269 - accuracy: 0.8809
Epoch 5/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3106 - accuracy: 0.8864
Epoch 6/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2945 - accuracy: 0.8913
Epoch 7/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2841 - accuracy: 0.8958
Epoch 8/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2763 - accuracy: 0.8982
Epoch 9/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2668 - accuracy: 0.9014
Epoch 10/10
```

```
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2577 - accuracy: 0.9035
313/313 - 1s - loss: 0.3441 - accuracy: 0.8779
Epoch 1/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.4998 - accuracy: 0.8239
Epoch 2/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.3768 - accuracy: 0.8647
Epoch 3/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3388 - accuracy: 0.8765
Epoch 4/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3147 - accuracy: 0.8851
Epoch 5/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2945 - accuracy: 0.8921
Epoch 6/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2817 - accuracy: 0.8961
Epoch 7/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.2687 - accuracy: 0.9004
Epoch 8/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2592 - accuracy: 0.9035: 1s - los
s: 0.2
Epoch 9/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2484 - accuracy: 0.9068
Epoch 10/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.2397 - accuracy: 0.9103
313/313 - 1s - loss: 0.3520 - accuracy: 0.8735
Epoch 1/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.4821 - accuracy: 0.8285
Epoch 2/10
1875/1875 [==============================] - 10s 6ms/step - loss: 0.3645 - accuracy: 0.8680
Epoch 3/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.3253 - accuracy: 0.8812
Epoch 4/10
1875/1875 [==============================] - 11s 6ms/step - loss: 0.3022 - accuracy: 0.8884
Epoch 5/10
1875/1875 [==============================] - 13s 7ms/step - loss: 0.2834 - accuracy: 0.8959
Epoch 6/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.2709 - accuracy: 0.8993
Epoch 7/10
1875/1875 [==============================] - 10s 6ms/step - loss: 0.2573 - accuracy: 0.9036
Epoch 8/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.2450 - accuracy: 0.9072
Epoch 9/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.2364 - accuracy: 0.9116 0s - los
s: 0.2
Epoch 10/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.2256 - accuracy: 0.9147
313/313 - 1s - loss: 0.3315 - accuracy: 0.8869
Epoch 1/10
1875/1875 [==============================] - 17s 9ms/step - loss: 0.4739 - accuracy: 0.8309
Epoch 2/10
1875/1875 [==============================] - 16s 9ms/step - loss: 0.3599 - accuracy: 0.8680
Epoch 3/10
1875/1875 [==============================] - 17s 9ms/step - loss: 0.3202 - accuracy: 0.8814
Epoch 4/10
1875/1875 [==============================] - 16s 8ms/step - loss: 0.2970 - accuracy: 0.8900
Epoch 5/10
1875/1875 [==============================] - 15s 8ms/step - loss: 0.2795 - accuracy: 0.8958
Epoch 6/10
1875/1875 [==============================] - 14s 8ms/step - loss: 0.2639 - accuracy: 0.9008
Epoch 7/10
1875/1875 [==============================] - 14s 7ms/step - loss: 0.2527 - accuracy: 0.9054
Epoch 8/10
1875/1875 [==============================] - 15s 8ms/step - loss: 0.2401 - accuracy: 0.9097
Epoch 9/10
1875/1875 [==============================] - 14s 8ms/step - loss: 0.2309 - accuracy: 0.9134
Epoch 10/10
1875/1875 [==============================] - 15s 8ms/step - loss: 0.2195 - accuracy: 0.9177
313/313 - 1s - loss: 0.3583 - accuracy: 0.8822
Epoch 1/10
1875/1875 [==============================] - 26s 14ms/step - loss: 0.4703 - accuracy: 0.8305
Epoch 2/10
1875/1875 [==============================] - 28s 15ms/step - loss: 0.3582 - accuracy: 0.8694
Epoch 3/10
1875/1875 [==============================] - 29s 15ms/step - loss: 0.3207 - accuracy: 0.8808
Epoch 4/10
1875/1875 [==============================] - 29s 15ms/step - loss: 0.2985 - accuracy: 0.8907
Epoch 5/10
1875/1875 [==============================] - 29s 15ms/step - loss: 0.2765 - accuracy: 0.8965
Epoch 6/10
1875/1875 [==============================] - 30s 16ms/step - loss: 0.2635 - accuracy: 0.9026
Epoch 7/10
1875/1875 [==============================] - 28s 15ms/step - loss: 0.2484 - accuracy: 0.9070
Epoch 8/10
1875/1875 [==============================] - 26s 14ms/step - loss: 0.2378 - accuracy: 0.9105
```

```
Epoch 9/10
1875/1875 [==============================] - 24s 13ms/step - loss: 0.2290 - accuracy: 0.91400s - los
s: 0.2291 - accu
Epoch 10/10
1875/1875 [==============================] - 24s 13ms/step - loss: 0.2184 - accuracy: 0.9186
313/313 - 1s - loss: 0.3233 - accuracy: 0.8880
```



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__)
```

```
2.3.0
```

In [2]:

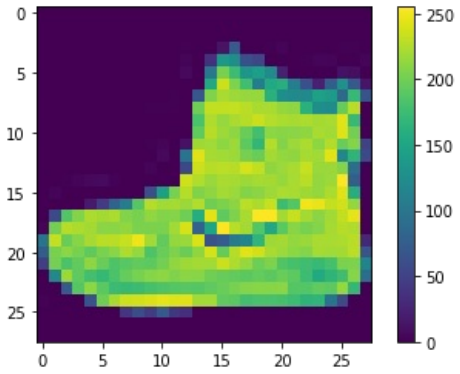
```
#Load the dataset and split into training and testing part
fashion_mnist = tf.keras.datasets.fashion_mnist

(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()
```

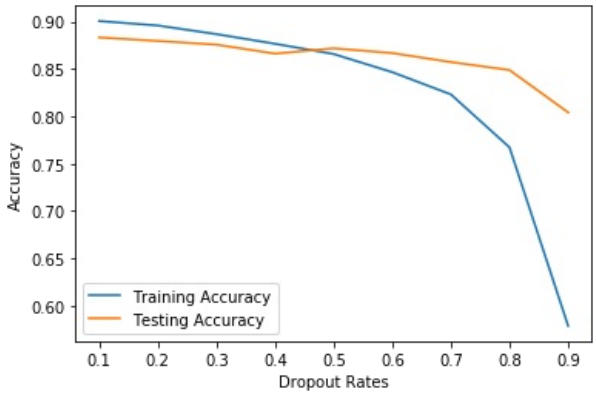
In [8]:

```
#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 [==============================] - 7s 4ms/step - loss: 0.5185 - accuracy: 0.8167
Epoch 2/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3885 - accuracy: 0.8588
Epoch 3/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3537 - accuracy: 0.8708
Epoch 4/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3302 - accuracy: 0.8791
Epoch 5/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3147 - accuracy: 0.8833
Epoch 6/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3029 - accuracy: 0.8878
Epoch 7/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2888 - accuracy: 0.8918
Epoch 8/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2813 - accuracy: 0.8947
Epoch 9/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2720 - accuracy: 0.8979
Epoch 10/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.2642 - accuracy: 0.9005
313/313 - 0s - loss: 0.3299 - accuracy: 0.8832
Epoch 1/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.5329 - accuracy: 0.8118
Epoch 2/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4003 - accuracy: 0.8566
Epoch 3/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3700 - accuracy: 0.8639
Epoch 4/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3457 - accuracy: 0.8729
Epoch 5/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3275 - accuracy: 0.8798
Epoch 6/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.3190 - accuracy: 0.8815
Epoch 7/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3065 - accuracy: 0.8871
Epoch 8/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2985 - accuracy: 0.8892
Epoch 9/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2893 - accuracy: 0.8917
Epoch 10/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.2799 - accuracy: 0.8958
313/313 - 1s - loss: 0.3389 - accuracy: 0.8795
```

```
Epoch 1/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.5502 - accuracy: 0.8058
Epoch 2/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.4152 - accuracy: 0.8521
Epoch 3/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3801 - accuracy: 0.8603
Epoch 4/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3613 - accuracy: 0.8673
Epoch 5/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.3488 - accuracy: 0.8731
Epoch 6/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3402 - accuracy: 0.8760
Epoch 7/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3283 - accuracy: 0.8785
Epoch 8/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3193 - accuracy: 0.8813
Epoch 9/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3117 - accuracy: 0.8835
Epoch 10/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3055 - accuracy: 0.8867
313/313 - 1s - loss: 0.3458 - accuracy: 0.8756
Epoch 1/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.5810 - accuracy: 0.7936
Epoch 2/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4417 - accuracy: 0.8400
Epoch 3/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.4071 - accuracy: 0.8518
Epoch 4/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3911 - accuracy: 0.8571
Epoch 5/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3771 - accuracy: 0.8625
Epoch 6/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3657 - accuracy: 0.8650
Epoch 7/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3603 - accuracy: 0.8673
Epoch 8/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3500 - accuracy: 0.8702
Epoch 9/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3437 - accuracy: 0.8747
Epoch 10/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3378 - accuracy: 0.8765
313/313 - 1s - loss: 0.3737 - accuracy: 0.8661
Epoch 1/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.6187 - accuracy: 0.7813
Epoch 2/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4704 - accuracy: 0.8312
Epoch 3/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.4401 - accuracy: 0.8411
Epoch 4/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4189 - accuracy: 0.8464
Epoch 5/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.4054 - accuracy: 0.8519
Epoch 6/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3917 - accuracy: 0.8565
Epoch 7/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3842 - accuracy: 0.8588
Epoch 8/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3781 - accuracy: 0.8621
Epoch 9/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.3724 - accuracy: 0.8629
Epoch 10/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.3654 - accuracy: 0.8657
313/313 - 1s - loss: 0.3548 - accuracy: 0.8717
Epoch 1/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.6722 - accuracy: 0.7601
Epoch 2/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.5171 - accuracy: 0.8143
Epoch 3/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4843 - accuracy: 0.8246
Epoch 4/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.4670 - accuracy: 0.8292
Epoch 5/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4521 - accuracy: 0.8363
Epoch 6/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.4426 - accuracy: 0.8379
Epoch 7/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4358 - accuracy: 0.8407
Epoch 8/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4289 - accuracy: 0.8434
Epoch 9/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4199 - accuracy: 0.8462
Epoch 10/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.4157 - accuracy: 0.8465
```

```
313/313 - 0s - loss: 0.3762 - accuracy: 0.8667
Epoch 1/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.7646 - accuracy: 0.7261
Epoch 2/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.5877 - accuracy: 0.7864
Epoch 3/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.5577 - accuracy: 0.7980
Epoch 4/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.5353 - accuracy: 0.8049
Epoch 5/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.5218 - accuracy: 0.8102
Epoch 6/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.5096 - accuracy: 0.8135
Epoch 7/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4948 - accuracy: 0.8177
Epoch 8/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.4919 - accuracy: 0.8203
Epoch 9/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4836 - accuracy: 0.8236
Epoch 10/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4838 - accuracy: 0.8231
313/313 - 0s - loss: 0.4031 - accuracy: 0.8572
Epoch 1/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.9451 - accuracy: 0.6486
Epoch 2/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.7384 - accuracy: 0.7197
Epoch 3/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.6936 - accuracy: 0.7403
Epoch 4/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.6714 - accuracy: 0.7463
Epoch 5/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.6601 - accuracy: 0.7521
Epoch 6/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.6470 - accuracy: 0.7580
Epoch 7/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.6404 - accuracy: 0.7594
Epoch 8/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.6253 - accuracy: 0.7614
Epoch 9/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.6261 - accuracy: 0.7622
Epoch 10/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.6113 - accuracy: 0.7671
313/313 - 0s - loss: 0.4397 - accuracy: 0.8488
Epoch 1/10
1875/1875 [==============================] - 5s 3ms/step - loss: 1.4103 - accuracy: 0.4353
Epoch 2/10
1875/1875 [==============================] - 6s 3ms/step - loss: 1.1915 - accuracy: 0.5115
Epoch 3/10
1875/1875 [==============================] - 6s 3ms/step - loss: 1.1499 - accuracy: 0.5253
Epoch 4/10
1875/1875 [==============================] - 6s 3ms/step - loss: 1.1238 - accuracy: 0.5389
Epoch 5/10
1875/1875 [==============================] - 7s 4ms/step - loss: 1.1002 - accuracy: 0.5515
Epoch 6/10
1875/1875 [==============================] - 6s 3ms/step - loss: 1.0911 - accuracy: 0.5590
Epoch 7/10
1875/1875 [==============================] - 7s 4ms/step - loss: 1.0727 - accuracy: 0.5648
Epoch 8/10
1875/1875 [==============================] - 7s 4ms/step - loss: 1.0674 - accuracy: 0.5692
Epoch 9/10
1875/1875 [==============================] - 9s 5ms/step - loss: 1.0634 - accuracy: 0.5734
Epoch 10/10
1875/1875 [==============================] - 6s 3ms/step - loss: 1.0479 - accuracy: 0.5786
313/313 - 1s - loss: 0.5791 - accuracy: 0.8040
```



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__)
```

```
2.3.0
```

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]:

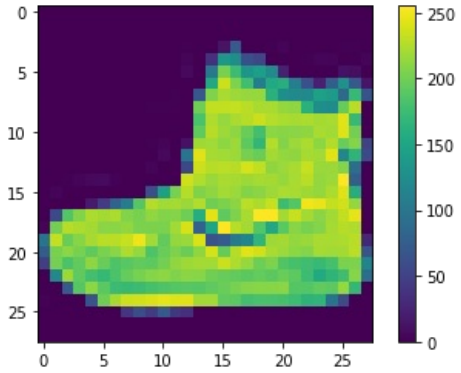
```
#Print the number of labels in the test dataset

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=['accuracy'])
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 (MaxPooling2 (None, 5, 5, 32)          0
_________________________________________________________________
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 [==============================] - 80s 43ms/step - loss: 0.4683 - accuracy: 0.8299
Epoch 2/10
1875/1875 [==============================] - 73s 39ms/step - loss: 0.3152 - accuracy: 0.8852
Epoch 3/10
1875/1875 [==============================] - 58s 31ms/step - loss: 0.2711 - accuracy: 0.9008
Epoch 4/10
1875/1875 [==============================] - 68s 36ms/step - loss: 0.2416 - accuracy: 0.9100
Epoch 5/10
1875/1875 [==============================] - 70s 37ms/step - loss: 0.2168 - accuracy: 0.9191
Epoch 6/10
1875/1875 [==============================] - 70s 38ms/step - loss: 0.1978 - accuracy: 0.9255
Epoch 7/10
1875/1875 [==============================] - 72s 39ms/step - loss: 0.1817 - accuracy: 0.9310
Epoch 8/10
1875/1875 [==============================] - 71s 38ms/step - loss: 0.1652 - accuracy: 0.9376
Epoch 9/10
1875/1875 [==============================] - 72s 39ms/step - loss: 0.1519 - accuracy: 0.9426
Epoch 10/10
1875/1875 [==============================] - 84s 45ms/step - loss: 0.1386 - accuracy: 0.9475
```

Out[12]:

```
<tensorflow.python.keras.callbacks.History at 0x19268d05a88>
```

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 - 3s - loss: 0.2936 - accuracy: 0.9063

Test accuracy: 0.9063000082969666
```

In [ ]:

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

In [2]:

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

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

print(tf.__version__)
```

```
2.3.0
```

In [3]:

```
#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 [4]:

```
#Set the class names

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

In [5]:

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

Out[5]:

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

In [6]:

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

Out[6]:

```
60000
```

In [7]:

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

Out[7]:

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

In [8]:

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

Out[8]:

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

In [9]:

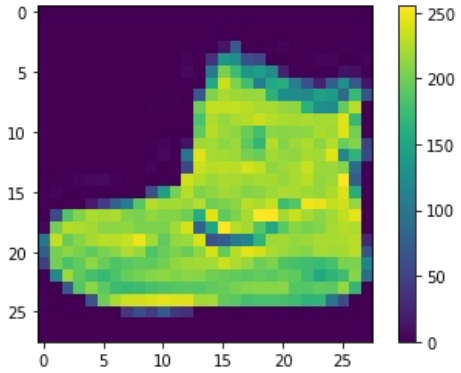
```
#Print the number of labels in the test dataset

len(test_labels)
```

Out[9]:

```
10000
```

In [10]:

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



In [11]:

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

test_images = test_images / 255.0
```

In [12]:

```
#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 [13]:

```
#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=['accuracy'])
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 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
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 [==============================] - 166s 88ms/step - loss: 0.4428 - accuracy: 0.8391
Epoch 2/10
1875/1875 [==============================] - 158s 84ms/step - loss: 0.2970 - accuracy: 0.8915
Epoch 3/10
1875/1875 [==============================] - 144s 77ms/step - loss: 0.2503 - accuracy: 0.9072
Epoch 4/10
1875/1875 [==============================] - 149s 80ms/step - loss: 0.2176 - accuracy: 0.9181
Epoch 5/10
1875/1875 [==============================] - 142s 76ms/step - loss: 0.1923 - accuracy: 0.9284
Epoch 6/10
1875/1875 [==============================] - 137s 73ms/step - loss: 0.1688 - accuracy: 0.9373
Epoch 7/10
1875/1875 [==============================] - 151s 80ms/step - loss: 0.1495 - accuracy: 0.9441
Epoch 8/10
1875/1875 [==============================] - 158s 84ms/step - loss: 0.1321 - accuracy: 0.9506
Epoch 9/10
1875/1875 [==============================] - 140s 75ms/step - loss: 0.1172 - accuracy: 0.9557
Epoch 10/10
1875/1875 [==============================] - 164s 88ms/step - loss: 0.1038 - accuracy: 0.9605
```

Out[13]:

```
<tensorflow.python.keras.callbacks.History at 0x1823233d0c8>
```

In [14]:

```
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 - 6s - loss: 0.3014 - accuracy: 0.9141

Test accuracy: 0.9140999913215637
```

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__)
```

```
2.3.0
```

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]:

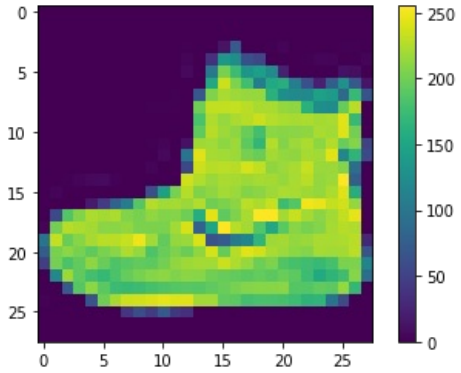
```
#Print the number of labels in the test dataset

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=['accuracy'])
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 (MaxPooling2 (None, 4, 4, 32)          0
_________________________________________________________________
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 [==============================] - 87s 47ms/step - loss: 0.4786 - accuracy: 0.8276
Epoch 2/10
1875/1875 [==============================] - 80s 43ms/step - loss: 0.3189 - accuracy: 0.8841
Epoch 3/10
1875/1875 [==============================] - 96s 51ms/step - loss: 0.2754 - accuracy: 0.8986
Epoch 4/10
1875/1875 [==============================] - 103s 55ms/step - loss: 0.2446 - accuracy: 0.9097
Epoch 5/10
1875/1875 [==============================] - 105s 56ms/step - loss: 0.2217 - accuracy: 0.9178s - los
s: 0.2217 - accuracy:
Epoch 6/10
1875/1875 [==============================] - 92s 49ms/step - loss: 0.2009 - accuracy: 0.9241
Epoch 7/10
1875/1875 [==============================] - 84s 45ms/step - loss: 0.1851 - accuracy: 0.9308
Epoch 8/10
1875/1875 [==============================] - 83s 44ms/step - loss: 0.1672 - accuracy: 0.9372
Epoch 9/10
1875/1875 [==============================] - 81s 43ms/step - loss: 0.1557 - accuracy: 0.94132s - los
s: 0.1556  - ETA: 0s - loss: 0.155
Epoch 10/10
1875/1875 [==============================] - 84s 45ms/step - loss: 0.1434 - accuracy: 0.9463
```

Out[12]:

```
<tensorflow.python.keras.callbacks.History at 0x1fe1e837948>
```

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 - 3s - loss: 0.2803 - accuracy: 0.9084

Test accuracy: 0.9083999991416931
```

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__)
```

```
2.3.0
```

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]:

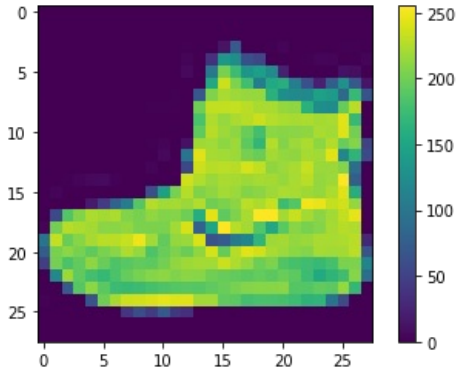
```
#Print the number of labels in the test dataset

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, (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=['accuracy'])
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, 10, 10, 64)        36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 1600)              0
_________________________________________________________________
dense (Dense)                (None, 128)               204928
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 244,810
Trainable params: 244,810
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
1875/1875 [==============================] - 135s 72ms/step - loss: 0.4383 - accuracy: 0.8410
Epoch 2/10
1875/1875 [==============================] - 139s 74ms/step - loss: 0.2883 - accuracy: 0.8941
Epoch 3/10
1875/1875 [==============================] - 132s 70ms/step - loss: 0.2477 - accuracy: 0.9088
Epoch 4/10
1875/1875 [==============================] - 121s 65ms/step - loss: 0.2159 - accuracy: 0.9198
Epoch 5/10
1875/1875 [==============================] - 107s 57ms/step - loss: 0.1918 - accuracy: 0.9286
Epoch 6/10
1875/1875 [==============================] - 106s 56ms/step - loss: 0.1702 - accuracy: 0.9361
Epoch 7/10
1875/1875 [==============================] - 127s 68ms/step - loss: 0.1523 - accuracy: 0.9429
Epoch 8/10
1875/1875 [==============================] - 137s 73ms/step - loss: 0.1373 - accuracy: 0.9481
Epoch 9/10
1875/1875 [==============================] - 136s 73ms/step - loss: 0.1199 - accuracy: 0.9549
Epoch 10/10
1875/1875 [==============================] - 131s 70ms/step - loss: 0.1082 - accuracy: 0.9590
```

Out[12]:

```
<tensorflow.python.keras.callbacks.History at 0x260219ed908>
```

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 - 5s - loss: 0.3114 - accuracy: 0.9121

Test accuracy: 0.9121000170707703
```

Observations:

1. The test accuracy when we set filters=32 for a filter size of (3,3) is **0.9063000082969666**.
2. The test accuracy when we set filters=64 for a filter size of (3,3) is **0.9140999913215637**
3. The test accuracy when we set filters=32 for a filter size of (5,5) is **0.9083999991416931**.
4. The test accuracy when we set filters=64 for a filter size of (5,5) is **0.9121000170707703**

The results of the experiments conclude that:

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

In [ ]: