Import Library

In [1]:
```
# Data analysis and visualization
import tensorflow as tf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

# Preprocessing and evaluation
from sklearn.model_selection import train_test_split
from sklearn.compose import make_column_transformer
from sklearn.preprocessing import MinMaxScaler
```

Load Data

In [2]:
```
(X_train , y_train), (X_test , y_test) = tf.keras.datasets.boston_housing.load_data(
                                            path = 'boston_housing_npz',
                                            test_split = 0.2,
                                            seed = 42
                                        )
```

Exploratory Data Analysis

Initial Observation

In [3]:
```
# Checking the data shape and type
(X_train.shape, type(X_train)), (X_test.shape, type(X_test)), (y_train.shape, type(y_train)), (y_test.shape, t
```

Out[3]:
```
(((404, 13), numpy.ndarray),
 ((102, 13), numpy.ndarray),
 ((404,), numpy.ndarray),
 ((102,), numpy.ndarray))
```

In [4]:
```
# Converting Data to DataFrame
X_train_df = pd.DataFrame(X_train)
y_train_df = pd.DataFrame(y_train)

# Preview the training data
X_train_df.head(10)
```

Out[4]:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0.09178	0.0	4.05	0.0	0.510	6.416	84.1	2.6463	5.0	296.0	16.6	395.50	9.04
1	0.05644	40.0	6.41	1.0	0.447	6.758	32.9	4.0776	4.0	254.0	17.6	396.90	3.53
2	0.10574	0.0	27.74	0.0	0.609	5.983	98.8	1.8681	4.0	711.0	20.1	390.11	18.07
3	0.09164	0.0	10.81	0.0	0.413	6.065	7.8	5.2873	4.0	305.0	19.2	390.91	5.52
4	5.09017	0.0	18.10	0.0	0.713	6.297	91.8	2.3682	24.0	666.0	20.2	385.09	17.27
5	0.10153	0.0	12.83	0.0	0.437	6.279	74.5	4.0522	5.0	398.0	18.7	373.66	11.97
6	0.31827	0.0	9.90	0.0	0.544	5.914	83.2	3.9986	4.0	304.0	18.4	390.70	18.33
7	0.29090	0.0	21.89	0.0	0.624	6.174	93.6	1.6119	4.0	437.0	21.2	388.08	24.16
8	4.03841	0.0	18.10	0.0	0.532	6.229	90.7	3.0993	24.0	666.0	20.2	395.33	12.87
9	0.22438	0.0	9.69	0.0	0.585	6.027	79.7	2.4982	6.0	391.0	19.2	396.90	14.33

In [5]:
```
# View summary of datasets
X_train_df.info()
```

```
print('_'*40)
y_train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 404 entries, 0 to 403
Data columns (total 13 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   0       404 non-null    float64
 1   1       404 non-null    float64
 2   2       404 non-null    float64
 3   3       404 non-null    float64
 4   4       404 non-null    float64
 5   5       404 non-null    float64
 6   6       404 non-null    float64
 7   7       404 non-null    float64
 8   8       404 non-null    float64
 9   9       404 non-null    float64
 10  10      404 non-null    float64
 11  11      404 non-null    float64
 12  12      404 non-null    float64
dtypes: float64(13)
memory usage: 41.2 KB
________________________________________
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 404 entries, 0 to 403
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   0       404 non-null    float64
dtypes: float64(1)
memory usage: 3.3 KB
```

In [6]:

```
# distribution of numerical feature values across the samples
X_train_df.describe()
```

Out[6]:

	0	1	2	3	4	5	6	7	8	9
count	404.000000	404.000000	404.000000	404.000000	404.000000	404.000000	404.000000	404.000000	404.000000	404.000000
mean	3.789989	11.568069	11.214059	0.069307	0.554524	6.284824	69.119307	3.792258	9.660891	408.960396
std	9.132761	24.269648	6.925462	0.254290	0.116408	0.723759	28.034606	2.142651	8.736073	169.685166
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.137000	1.000000	187.000000
25%	0.081960	0.000000	5.190000	0.000000	0.452000	5.878750	45.475000	2.097050	4.000000	281.000000
50%	0.262660	0.000000	9.690000	0.000000	0.538000	6.210000	77.500000	3.167500	5.000000	330.000000
75%	3.717875	12.500000	18.100000	0.000000	0.624000	6.620500	94.425000	5.118000	24.000000	666.000000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000

Preprocessing

In [7]:

```
# Create column transformer
ct = make_column_transformer(
    (MinMaxScaler(), [0, 1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12])
)

# Normalization and data type change
X_train = ct.fit_transform(X_train).astype('float32')
X_test = ct.transform(X_test).astype('float32')
y_train = y_train.astype('float32')
y_test = y_test.astype('float32')

# Distribution of X_train feature values after normalization
pd.DataFrame(X_train).describe()
```

Out[7]:

	0	1	2	3	4	5	6	7	8	9
count	404.000000	404.000000	404.000000	404.000000	404.000000	404.000000	404.000000	404.000000	404.000000	404.000000
mean	0.042528	0.115681	0.394210	0.348815	0.521905	0.681970	0.241618	0.376560	0.423589	0.625737
std	0.102650	0.242696	0.253866	0.239522	0.138678	0.288719	0.194973	0.379829	0.323827	0.229502
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000850	0.000000	0.173387	0.137860	0.444098	0.438466	0.087361	0.130435	0.179389	0.510638
50%	0.002881	0.000000	0.338343	0.314815	0.507569	0.768280	0.184767	0.173913	0.272901	0.691489
75%	0.041717	0.125000	0.646628	0.491770	0.586223	0.942585	0.362255	1.000000	0.914122	0.808511
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

Model, Predict, Evaluation

In [8]:

```
# Reserve data for validation
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.1, random_state=42)
X_train.shape, X_val.shape, y_train.shape, y_val.shape
```

Out[8]: ((363, 12), (41, 12), (363,), (41,))

Creating the Model and Optimizing the Learning Rate

learning rate = 0.01, batch_size = 32, dense_layers = 2, hidden_units for Dense_1 layer= 10, hidden_units for Dense_2 layer = 100

In [9]:

```
# Set random seed
tf.random.set_seed(42)

# Building the model
model = tf.keras.Sequential([
  tf.keras.layers.Dense(units=10, activation='relu', input_shape=(X_train.shape[1],), name='Dense_1'),
  tf.keras.layers.Dense(units=100, activation='relu', name='Dense_2'),
  tf.keras.layers.Dense(units=1, name='Prediction')
])

# Compiling the model
model.compile(
    loss = tf.keras.losses.mean_squared_error,
    optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.01),
    metrics = ['mse']
)

# Training the model
history = model.fit(
    X_train,
    y_train,
    batch_size=32,
    epochs=50,
    validation_data=(X_val, y_val)
)
```

```
12/12 [==============================] - 0s 3ms/step - loss: 13.7454 - mse: 13.7454 - val_loss: 15.7261 - val_
mse: 15.7261
Epoch 49/50
12/12 [==============================] - 0s 4ms/step - loss: 14.7645 - mse: 14.7645 - val_loss: 9.9898 - val_m
se: 9.9898
Epoch 50/50
12/12 [==============================] - 0s 3ms/step - loss: 15.1951 - mse: 15.1951 - val_loss: 11.3679 - val_
mse: 11.3679
```

Model Evaluation

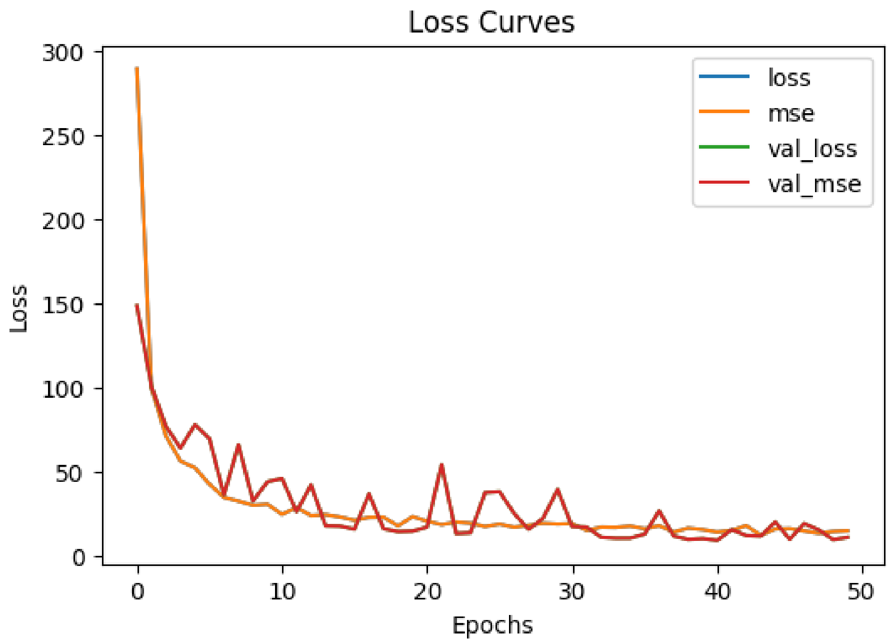
In [10]:
```
# Preview the mean value of training and validation data
y_train.mean(), y_val.mean()
```

Out[10]:
```
(22.235537, 24.89756)
```

In [11]:
```
# Evaluate the model on the test data
print("Evaluation on Test data \n")
loss, mse = model.evaluate(X_test, y_test, batch_size=32)
print(f"\nModel loss on test set: {loss}")
print(f"Model mean squared error on test set: {(mse):.2f}")
```

```
Evaluation on Test data

4/4 [==============================] - 0s 2ms/step - loss: 15.5170 - mse: 15.5170

Model loss on test set: 15.517014503479004
Model mean squared error on test set: 15.52
```

In [12]:
```
# Plot the loss curves
pd.DataFrame(history.history).plot(figsize=(6, 4), xlabel="Epochs", ylabel="Loss", title='Loss Curves')
plt.show()
```



Model Prediction

In [13]:
```
# Make predictions
y_pred = model.predict(X_test)

# View the first prediction
y_pred[0]
```

```
4/4 [==============================] - 0s 1ms/step
```

Out[13]:
```
array([19.093914], dtype=float32)
```

Classifying movie reviews: a binary classification example

Design a neural network to perform two-class classification or *binary classification* , of reviews form IMDB movie reviews dataset, to determine wether the reviews are positive or negative. We will use the Python library Keras to perform the classification

The IMDB Dataset

The IMDB dataset is a set of 50,000 highly polarized reviews from the Internet Movie Database. They are split into 25000 reviews each for training and testing. Each set contains equal number (50%) of positive and negative reviews.

The IMDB dataset comes packaged with Keras. It consists of reviews and their corresponding labels (0 for *negative* and 1 for *positive* review). The reviews are a sequence of words. They come preprocessed as sequence of integers, where each integer stands for a specific word in the dictionary.

The IMDB datset can be loaded directly from Keras and will usually download about 80 MB on your machine.

Import Packages

In [1]:
```
import numpy as np
from keras.datasets import imdb
from keras import models
from keras import layers
from keras import optimizers
from keras import losses
from keras import metrics


import matplotlib.pyplot as plt
%matplotlib inline
```

Loading the Data

In [2]:
```
# Load the data, keeping only 10,000 of the most frequently occuring words
(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words = 10000)
```

In [3]:
```
# Check the first label
train_labels[0]
```

Out[3]: 1

In [4]:
```
# Since we restricted ourselves to the top 10000 frequent words, no word index should exceed
# we'll verify this below

# Here is a list of maximum indexes in every review --- we search the maximum index in this
print(type([max(sequence) for sequence in train_data]))

# Find the maximum of all max indexes
max([max(sequence) for sequence in train_data])
```

```
<class 'list'>
```

Out[4]: 9999

In [5]:
```
# Let's quickly decode a review

# step 1: load the dictionary mappings from word to integer index
word_index = imdb.get_word_index()

# step 2: reverse word index to map integer indexes to their respective words
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])

# Step 3: decode the review, mapping integer indices to words
#
# indices are off by 3 because 0, 1, and 2 are reserverd indices for "padding", "Start of se
decoded_review = ' '.join([reverse_word_index.get(i-3, '?') for i in train_data[0]])

decoded_review
```

Out[5]:
```
"? this film was just brilliant casting location scenery story direction everyone's really
suited the part they played and you could just imagine being there robert ? is an amazing a
ctor and now the same being director ? father came from the same scottish island as myself
so i loved the fact there was a real connection with this film the witty remarks throughout
the film were great it was just brilliant so much that i bought the film as soon as it was
released for ? and would recommend it to everyone to watch and the fly fishing was amazing
really cried at the end it was so sad and you know what they say if you cry at a film it mu
st have been good and this definitely was also ? to the two little boy's that played the ?
of norman and paul they were just brilliant children are often left out of the ? list i thi
nk because the stars that play them all grown up are such a big profile for the whole film
but these children are amazing and should be praised for what they have done don't you thin
k the whole story was so lovely because it was true and was someone's life after all that w
as shared with us all"
```

In [6]:
```
len(reverse_word_index)
```

Out[6]:
```
88584
```

Preparing the data

Vectorize input data

We cannot feed list of integers into our deep neural network. We will need to convert them into tensors.

To prepare our data we will One-hot Encode our lists and turn them into vectors of 0's and 1's. This would blow up all of our sequences into 10,000 dimensional vectors containing 1 at all indices corresponding to integers present in that sequence. This vector will have the element 0 at all indices which are not present in integer sequence.

Simply put, the 10,000 dimensional vector corresponding to each review, will have

- Every index corresponding to a word
- Every index vith value 1, is a word which is present in the review and is denoted by its integer counterpart
- Every index containing 0, is a word not present in the review

We will vectorize our data manually for maximum clarity. This will result in a tensors of shape (25000, 10000).

In [7]:
```
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))    # Creates an all zero matrix of shape
    for i,sequence in enumerate(sequences):
        results[i,sequence] = 1                        # Sets specific indices of results[i]
    return results

# Vectorize training Data
X_train = vectorize_sequences(train_data)

# Vectorize testing Data
X_test = vectorize_sequences(test_data)
```

In [8]:
```
X_train[0]
```
Out[8]:
```
array([0., 1., 1., ..., 0., 0., 0.])
```

In [9]:
```
X_train.shape
```
Out[9]:
```
(25000, 10000)
```

Vectorize labels

In [10]:
```
y_train = np.asarray(train_labels).astype('float32')
y_test  = np.asarray(test_labels).astype('float32')
```

Building the network

Our input data is vectors which needs to be mapped to scaler labels (0s and 1s). This is one of the easiest setups and a simple stack of *fully-connected*, *Dense* layers with *relu* activation perform quite well.

Hidden layers

In this network we will leverage *hidden layers*. we will define our layers as such.

```
Dense(16, activation='relu')
```

The argument being passed to each `Dense` layer, `(16)` is the number of *hidden units* of a layer.

The output from a *Dense* layer with *relu* activation is genrated after a chain of *tensor* operations. This chain of operations is implemented as

```
output = relu(dot(W, input) + b)
```

Where, `W` is the *Weight matrix* and `b` is the bias (tensor).

Having 16 hidden units means that the matrix W will be of the shape (*input_Dimension* , *16*). In this case where the dimension of input vector is 10,000; the shape of Weight matrix will be (10000, 16). If you were to represent this network as graph you would see 16 neurons in this hidden layer.

To put in in laymans terms, there will be 16 balls in this layer.

Each of these balls, or *hidden units* is a dimension in the representation space of the layer. Representaion space is the set of all viable representaions for the data. Every *hidden layer* composed of its *hidden units* aims to learns one specific transformation of the data, or one feature/pattern from the data.

Hidden layers, simply put, are layers of mathematical functions each designed to produce an output specific to an intended result. Hidden layers allow for the function of a neural network to be broken down into specific transformations of the data. Each hidden layer function is specialized to produce a defined output.For example, a hidden layer functions that are used to identify human eyes and ears may be used in conjunction by subsequent layers to identify faces in images. While the functions to identify eyes alone are not enough to independently recognize objects, they can function jointly within a neural network.

Model Architecture

1. For our model we will use
   - two intermediate layers with 16 hidden units each
   - Third layer that will output the scalar sentiment prediction
2. Intermediate layers will use *relu* activation function. *relu* or Rectified linear unit function will zero out the negative values.
3. Sigmoid activation for the final layer or *output layer*. A sigmoid function "*squashes*" arbitary values into the [0,1] range.

There are formal principles that guide our approach in selecting the architectural attributes of a model. These are not

Model defination

In [11]:
```python
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Compiling the model

In this step we will choose an *optimizer*, a *loss function*, and metrics to observe. We will go forward with

- *binary_crossentropy* loss function, commonlu used for Binary Classification
- *rmsprop* optimizer and
- *accuracy* as a measure of performance

We can pass our choices for optimizer, loss function and metrics as *strings* to the `compile` function because `rmsprop` , `binary_crossentropy` and `accuracy` come packaged with Keras.

```python
model.complie(
    optimizer='rmsprop',
    loss = 'binary_crossentropy',
    metrics = ['accuracy']
)
```

One could use a customized loss function or ortimizer by passing the custom *class instance* as argument to the `loss` , `optimizer` or `mertics` fields.

In this example, we will implement our default choices, but, we will do so by passing class instances. This is exactly how we would do it, if we had customized parameters.

In [12]:
```python
model.compile(
    optimizer=optimizers.RMSprop(learning_rate=0.001),
    loss = losses.binary_crossentropy,
    metrics = [metrics.binary_accuracy]
)
```

Setting up Validation

We will set aside a part of our training data for *validation* of the accuracy of the model as it trains. A *validation set* enables us to monitor the progress of our model on previously unseen data as it goes throug epochs during training.

Validation steps help us fine tune the training parameters of the `model.fit` function so as to avoid overfitting and under fitting of data.

In [13]:
```python
# Input for Validation
X_val = X_train[:10000]
partial_X_train = X_train[10000:]

# Labels for validation
y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

Training our model

Initially, we will train our models for 20 epochs in mini-batches of 512 samples. We will aslo pass our *validation set* to the `fit` method.

Calling the `fit` method returns a `History` object. This object contains a member `history` which stores all data

In [14]:
```python
history = model.fit(
    partial_X_train,
    partial_y_train,
    epochs=20,
    batch_size=512,
    validation_data=(X_val, y_val)
)
```

```
Epoch 1/20
30/30 [==============================] - 2s 37ms/step - loss: 0.5156 - binary_accuracy: 0.7
949 - val_loss: 0.3994 - val_binary_accuracy: 0.8627
Epoch 2/20
30/30 [==============================] - 0s 16ms/step - loss: 0.3162 - binary_accuracy: 0.8
993 - val_loss: 0.3088 - val_binary_accuracy: 0.8886
Epoch 3/20
30/30 [==============================] - 0s 16ms/step - loss: 0.2303 - binary_accuracy: 0.9
267 - val_loss: 0.2881 - val_binary_accuracy: 0.8869
Epoch 4/20
30/30 [==============================] - 0s 15ms/step - loss: 0.1831 - binary_accuracy: 0.9
411 - val_loss: 0.2792 - val_binary_accuracy: 0.8897
Epoch 5/20
30/30 [==============================] - 0s 15ms/step - loss: 0.1514 - binary_accuracy: 0.9
501 - val_loss: 0.2767 - val_binary_accuracy: 0.8884
Epoch 6/20
30/30 [==============================] - 0s 15ms/step - loss: 0.1214 - binary_accuracy: 0.9
642 - val_loss: 0.3112 - val_binary_accuracy: 0.8781
Epoch 7/20
30/30 [==============================] - 0s 15ms/step - loss: 0.1041 - binary_accuracy: 0.9
694 - val_loss: 0.3267 - val_binary_accuracy: 0.8804
Epoch 8/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0830 - binary_accuracy: 0.9
779 - val_loss: 0.3235 - val_binary_accuracy: 0.8807
Epoch 9/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0707 - binary_accuracy: 0.9
798 - val_loss: 0.3542 - val_binary_accuracy: 0.8775
Epoch 10/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0553 - binary_accuracy: 0.9
858 - val_loss: 0.3724 - val_binary_accuracy: 0.8774
Epoch 11/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0457 - binary_accuracy: 0.9
889 - val_loss: 0.4186 - val_binary_accuracy: 0.8712
Epoch 12/20
30/30 [==============================] - 0s 15ms/step - loss: 0.0381 - binary_accuracy: 0.9
915 - val_loss: 0.4310 - val_binary_accuracy: 0.8773
Epoch 13/20
30/30 [==============================] - 0s 15ms/step - loss: 0.0305 - binary_accuracy: 0.9
941 - val_loss: 0.4663 - val_binary_accuracy: 0.8751
Epoch 14/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0242 - binary_accuracy: 0.9
953 - val_loss: 0.5045 - val_binary_accuracy: 0.8726
Epoch 15/20
30/30 [==============================] - 0s 15ms/step - loss: 0.0192 - binary_accuracy: 0.9
966 - val_loss: 0.5289 - val_binary_accuracy: 0.8713
Epoch 16/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0162 - binary_accuracy: 0.9
971 - val_loss: 0.5600 - val_binary_accuracy: 0.8704
Epoch 17/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0111 - binary_accuracy: 0.9
987 - val_loss: 0.6111 - val_binary_accuracy: 0.8679
Epoch 18/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0082 - binary_accuracy: 0.9
995 - val_loss: 0.6720 - val_binary_accuracy: 0.8633
Epoch 19/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0106 - binary_accuracy: 0.9
978 - val_loss: 0.6709 - val_binary_accuracy: 0.8653
Epoch 20/20
30/30 [==============================] - 0s 14ms/step - loss: 0.0039 - binary_accuracy: 0.9
999 - val_loss: 0.6942 - val_binary_accuracy: 0.8653
```

At the end of training we have attained a training accuracy of 99.85% and validation accuracy of 86.57%

Now that we have trained our network, we will observe its performance metrics stored in the `History` object.

Calling the `fit` method returns a `History` object. This object has an sttribute `history` which is a dictionary containing four enteries: one per monitored metric.

In [15]:
```
history_dict = history.history
history_dict.keys()
```

Out[15]:
```
dict_keys(['loss', 'binary_accuracy', 'val_loss', 'val_binary_accuracy'])
```

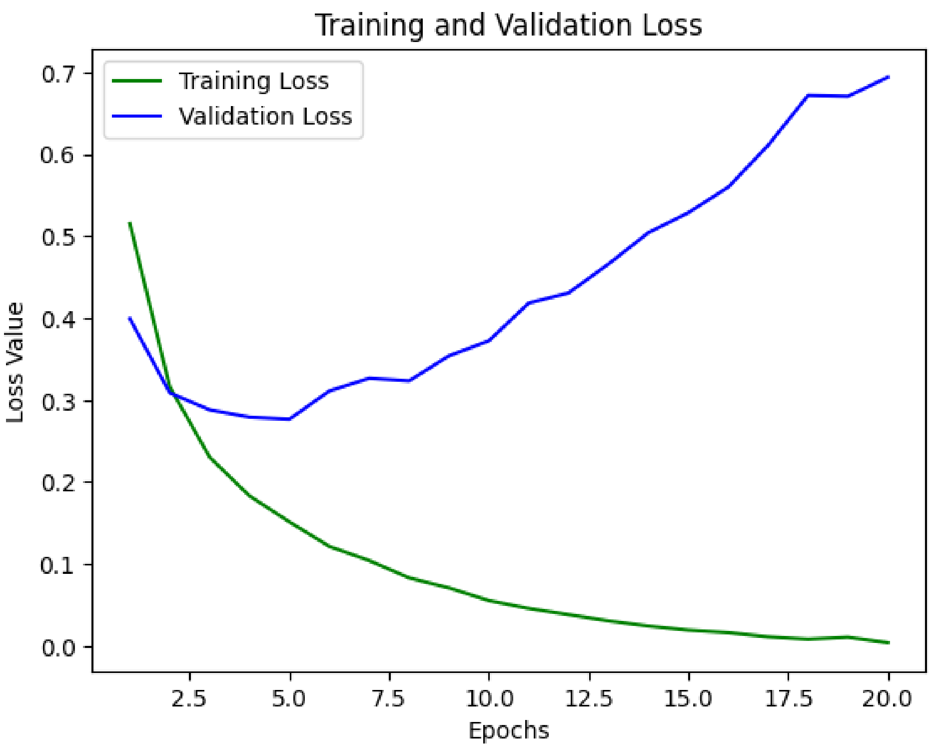
`history_dict` contains values of

- Training loss
- Trainining Accuracy
- Validation Loss
- Validation Accuracy

at the end of each epoch.

Let's use Matplotlib to plot Training and validation losses and Traing and Validation Accuracy side by side.

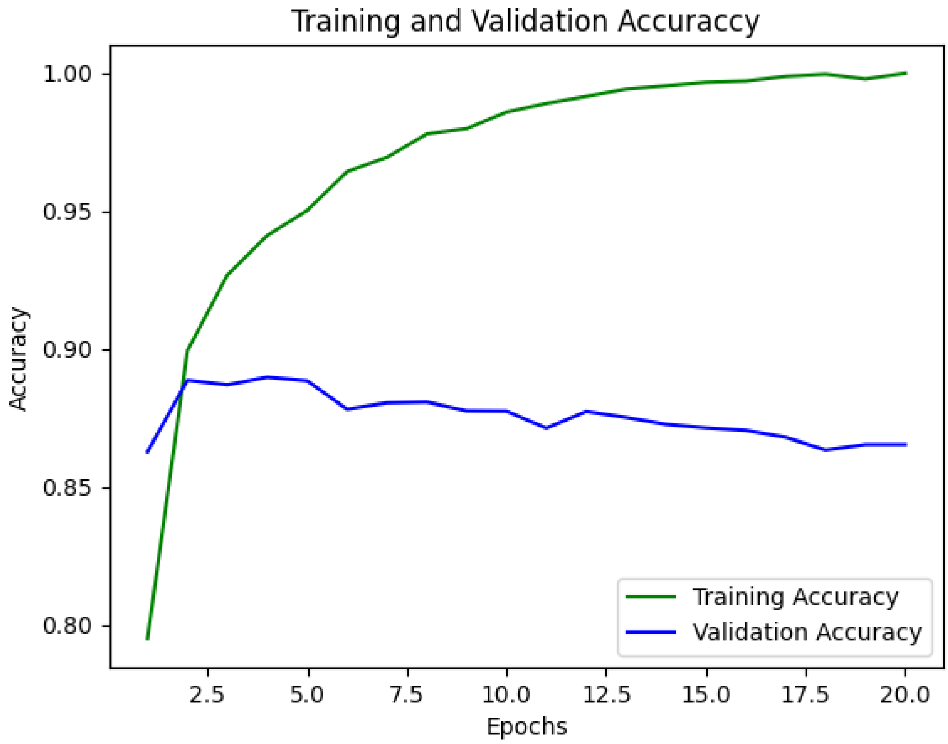
In [16]:
```
# Plotting losses
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss']

epochs = range(1, len(loss_values) + 1)

plt.plot(epochs, loss_values, 'g', label="Training Loss")
plt.plot(epochs, val_loss_values, 'b', label="Validation Loss")

plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss Value')
plt.legend()

plt.show()
```



Loading [MathJax]/extensions/Safe.js

In [17]:

```python
# Training and Validation Accuracy

acc_values = history_dict['binary_accuracy']
val_acc_values = history_dict['val_binary_accuracy']

epochs = range(1, len(loss_values) + 1)

plt.plot(epochs, acc_values, 'g', label="Training Accuracy")
plt.plot(epochs, val_acc_values, 'b', label="Validation Accuracy")

plt.title('Training and Validation Accuraccy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```



We observe that *minimum validation loss* and *maximum validation Accuracy* is achived at around 3-5 epochs. After that we observe 2 trends:

- increase in validation loss and decrease in training loss
- decrease in validation accuracy and increase in training accuracy

This implies that the model is getting better at classifying the sentiment of the training data, but making consistently worse predictions when it encounters new, previously unseed data. This is the hallmark of *Overfitting*. After the 5th epoch the model begins to fit too closely to the trainning data.

To address overfitting, we will reduce the number of epochs to somewhere between 3 and 5. These results may vary depending on your machine and due to the very nature of the random assignment of weights that may vary from model to mode.

In our case we will stop training after 3 epochs.

Retraining our model

In [18]:
```
model.fit(
    partial_X_train,
    partial_y_train,
    epochs=3,
    batch_size=512,
    validation_data=(X_val, y_val)
)
```

```
Epoch 1/3
30/30 [==============================] - 2s 50ms/step - loss: 0.0053 - binary_accuracy: 0.9
996 - val_loss: 0.7299 - val_binary_accuracy: 0.8648
Epoch 2/3
30/30 [==============================] - 0s 15ms/step - loss: 0.0040 - binary_accuracy: 0.9
993 - val_loss: 0.7694 - val_binary_accuracy: 0.8659
Epoch 3/3
30/30 [==============================] - 0s 15ms/step - loss: 0.0019 - binary_accuracy: 0.9
999 - val_loss: 0.8005 - val_binary_accuracy: 0.8638
```

Out[18]:
```
<keras.callbacks.History at 0x7f1e00ba11e0>
```

In the end we achive a *training accuracy* of 99% and a *validation accuray* of 86%

Model Evaluation

In [19]:
```
# Making Predictions for testing data
np.set_printoptions(suppress=True)
result = model.predict(X_test)
```

```
782/782 [==============================] - 1s 1ms/step
```

In [20]:
```
result
```

Out[20]:
```
array([[0.00426335],
       [0.9999999 ],
       [0.99748594],
       ...,
       [0.000199  ],
       [0.02019035],
       [0.5236855 ]], dtype=float32)
```

In [21]:
```
y_pred = np.zeros(len(result))
for i, score in enumerate(result):
    y_pred[i] = np.round(score)
```

In [22]:
```
mae = metrics.mean_absolute_error(y_pred, y_test)
mae
```

Out[22]:
```
<tf.Tensor: shape=(), dtype=float32, numpy=0.15012>
```

Basic classification: Classify images of clothing

In [1]:
```
# TensorFlow and tf.keras
import tensorflow as tf

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

print(tf.__version__)
```

```
2.11.0
```

Import the Fashion MNIST dataset

This guide uses the [Fashion MNIST (https://github.com/zalandoresearch/fashion-mnist)](https://github.com/zalandoresearch/fashion-mnist) dataset which contains 70,000 grayscale images in 10 categories. The images show individual articles of clothing at low resolution (28 by 28 pixels), as seen here:

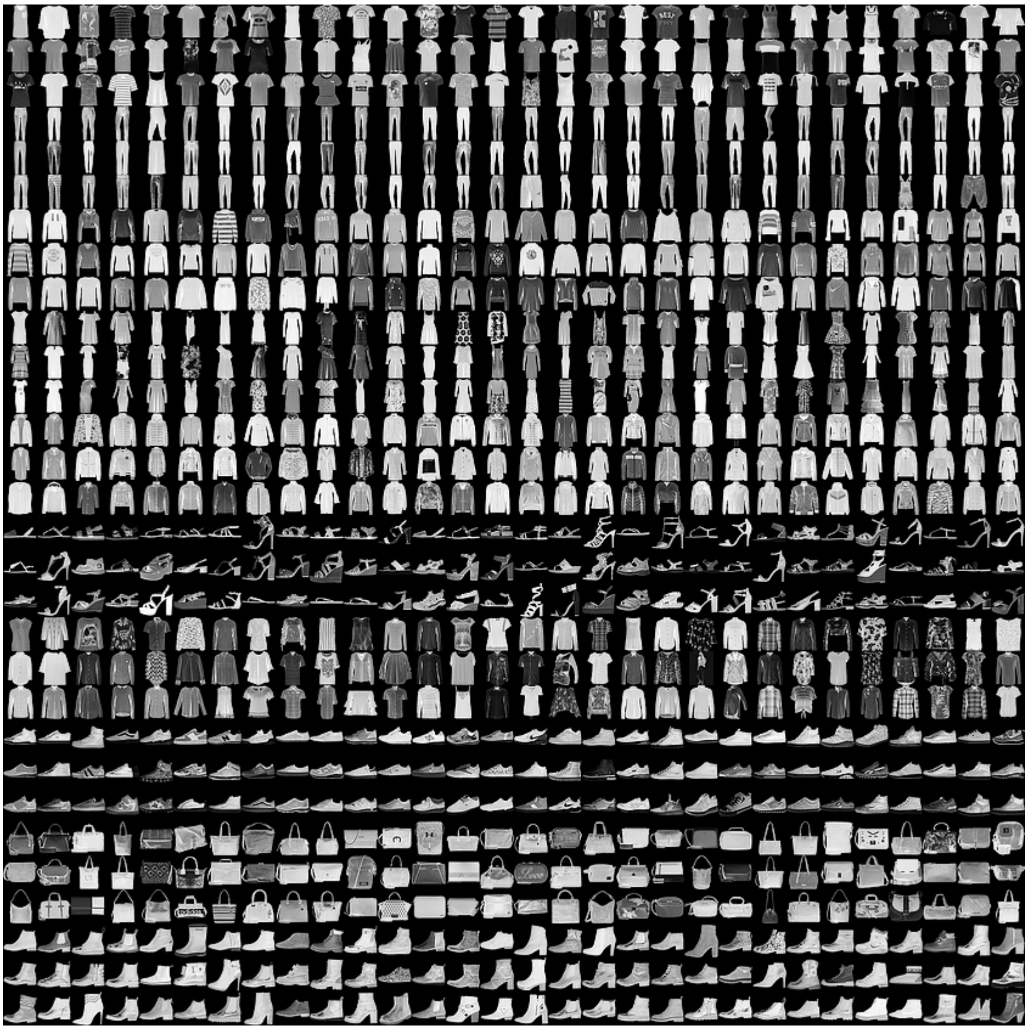


Figure 1. [Fashion-MNIST samples (https://github.com/zalandoresearch/fashion-mnist)](https://github.com/zalandoresearch/fashion-mnist) (by Zalando, MIT License).

Fashion MNIST is intended as a drop-in replacement for the classic [MNIST (http://yann.lecun.com/exdb/mnist/)](http://yann.lecun.com/exdb/mnist/) dataset—often used as the "Hello, World" of machine learning programs for computer vision. The MNIST dataset contains images of handwritten digits (0, 1, 2, etc.) in a format identical to that of the articles of clothing you'll use here.

This guide uses Fashion MNIST for variety, and because it's a slightly more challenging problem than regular MNIST. Both datasets are relatively small and are used to verify that an algorithm works as expected. They're good starting points to test and debug code.

Here, 60,000 images are used to train the network and 10,000 images to evaluate how accurately the network learned to

In [2]:

```
fashion_mnist = tf.keras.datasets.fashion_mnist

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

Loading the dataset returns four NumPy arrays:

- The `train_images` and `train_labels` arrays are the *training set*—the data the model uses to learn.
- The model is tested against the *test set*, the `test_images` , and `test_labels` arrays.

The images are 28x28 NumPy arrays, with pixel values ranging from 0 to 255. The *labels* are an array of integers, ranging from 0 to 9. These correspond to the *class* of clothing the image represents:

Label	**Class**
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Each image is mapped to a single label. Since the *class names* are not included with the dataset, store them here to use later when plotting the images:

In [3]:

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

Explore the data

Let's explore the format of the dataset before training the model. The following shows there are 60,000 images in the training set, with each image represented as 28 x 28 pixels:

In [4]:

```
train_images.shape
```

Out[4]:

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

Likewise, there are 60,000 labels in the training set:

In [5]:

```
len(train_labels)
```

Out[5]:

```
60000
```

Each label is an integer between 0 and 9:

In [6]:

```
train_labels
```

Out[6]:

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

There are 10,000 images in the test set. Again, each image is represented as 28 x 28 pixels:

In [7]:
```
test_images.shape
```

Out[7]:
```
(10000, 28, 28)
```

And the test set contains 10,000 images labels:

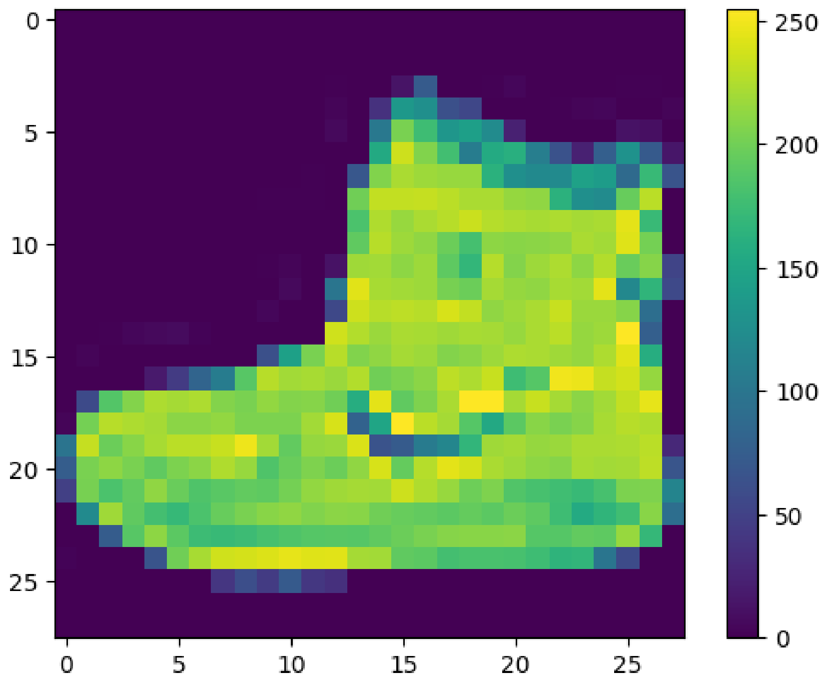
In [8]:
```
len(test_labels)
```

Out[8]:
```
10000
```

Preprocess the data

The data must be preprocessed before training the network. If you inspect the first image in the training set, you will see that the pixel values fall in the range of 0 to 255:

In [9]:
```
plt.figure()
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
plt.show()
```



Scale these values to a range of 0 to 1 before feeding them to the neural network model. To do so, divide the values by 255. It's important that the *training set* and the *testing set* be preprocessed in the same way:

In [10]:
```
train_images = train_images / 255.0

test_images = test_images / 255.0
```

To verify that the data is in the correct format and that you're ready to build and train the network, let's display the first 25 images from the *training set* and display the class name below each image.

In [11]:
```python
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()
```

Build the model

Building the neural network requires configuring the layers of the model, then compiling the model.

Set up the layers

The basic building block of a neural network is the [*layer* (https://www.tensorflow.org/api_docs/python/tf/keras/layers)](https://www.tensorflow.org/api_docs/python/tf/keras/layers). Layers extract representations from the data fed into them. Hopefully, these representations are meaningful for the problem at hand.

Most of deep learning consists of chaining together simple layers. Most layers, such as `tf.keras.layers.Dense`, have parameters that are learned during training.

In [12]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])
```

The first layer in this network, `tf.keras.layers.Flatten`, transforms the format of the images from a two-dimensional array (of 28 by 28 pixels) to a one-dimensional array (of 28 * 28 = 784 pixels). Think of this layer as unstacking rows of pixels in the image and lining them up. This layer has no parameters to learn; it only reformats the data.

After the pixels are flattened, the network consists of a sequence of two `tf.keras.layers.Dense` layers. These are densely connected, or fully connected, neural layers. The first `Dense` layer has 128 nodes (or neurons). The second (and last) layer returns a logits array with length of 10. Each node contains a score that indicates the current image belongs to one of the 10 classes.

Compile the model

Before the model is ready for training, it needs a few more settings. These are added during the model's [*compile* (https://www.tensorflow.org/api_docs/python/tf/keras/Model#compile)](https://www.tensorflow.org/api_docs/python/tf/keras/Model#compile) step:

- [*Loss function* (https://www.tensorflow.org/api_docs/python/tf/keras/losses)](https://www.tensorflow.org/api_docs/python/tf/keras/losses) —This measures how accurate the model is during training. You want to minimize this function to "steer" the model in the right direction.
- [*Optimizer* (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers)](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers) —This is how the model is updated based on the data it sees and its loss function.
- [*Metrics* (https://www.tensorflow.org/api_docs/python/tf/keras/metrics)](https://www.tensorflow.org/api_docs/python/tf/keras/metrics) —Used to monitor the training and testing steps. The following example uses *accuracy*, the fraction of the images that are correctly classified.

In [13]:
```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

Train the model

Training the neural network model requires the following steps:

1. Feed the training data to the model. In this example, the training data is in the `train_images` and `train_labels` arrays.
2. The model learns to associate images and labels.
3. You ask the model to make predictions about a test set—in this example, the `test_images` array.
4. Verify that the predictions match the labels from the `test_labels` array.

Feed the model

To start training, call the [`model.fit` (https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit)](https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit) method—so called because it "fits" the model to the training data:

```
1875/1875 [==============================] - 4s 2ms/step - loss: 0.1624 - accuracy: 0.938
8
Epoch 25/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0.1599 - accuracy: 0.940
2
Epoch 26/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0.1558 - accuracy: 0.941
9
Epoch 27/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0.1524 - accuracy: 0.943
1
Epoch 28/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0.1472 - accuracy: 0.944
4
Epoch 29/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0.1466 - accuracy: 0.945
2
Epoch 30/30
1875/1875 [==============================] - 4s 2ms/step - loss: 0.1420 - accuracy: 0.946
9
```

Out[14]:
```
<keras.callbacks.History at 0x7f326fff3370>
```

As the model trains, the loss and accuracy metrics are displayed. This model reaches an accuracy of about 0.91 (or 91%) on the training data.

Evaluate accuracy

Next, compare how the model performs on the test dataset:

In [15]:
```
test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)

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

```
313/313 - 0s - loss: 0.4365 - accuracy: 0.8796 - 481ms/epoch - 2ms/step

Test accuracy: 0.8795999884605408
```

It turns out that the accuracy on the test dataset is a little less than the accuracy on the training dataset. This gap between training accuracy and test accuracy represents *overfitting*. Overfitting happens when a machine learning model performs worse on new, previously unseen inputs than it does on the training data. An overfitted model "memorizes" the noise and details in the training dataset to a point where it negatively impacts the performance of the model on the new data. For more information, see the following:

- Demonstrate overfitting (https://www.tensorflow.org/tutorials/keras/overfit_and_underfit#demonstrate_overfitting)
- Strategies to prevent overfitting (https://www.tensorflow.org/tutorials/keras/overfit_and_underfit#strategies_to_prevent_overfitting)

Make predictions

With the model trained, you can use it to make predictions about some images. Attach a softmax layer to convert the model's linear outputs—logits (https://developers.google.com/machine-learning/glossary#logits)—to probabilities, which should be easier to interpret.

In [16]:
```
probability_model = tf.keras.Sequential([model, 
                                         tf.keras.layers.Softmax()])
```

In [17]:
```
predictions = probability_model.predict(test_images)
```

```
313/313 [==============================] - 0s 1ms/step
```

Here, the model has predicted the label for each image in the testing set. Let's take a look at the first prediction:

In [18]:
```
predictions[0]
```

Out[18]:
```
array([3.8219037e-12, 6.0263157e-12, 1.1343045e-19, 5.9180233e-16,
       1.9016950e-22, 4.5273669e-08, 5.1865828e-16, 1.9605557e-05,
       1.2793957e-13, 9.9998027e-01], dtype=float32)
```

A prediction is an array of 10 numbers. They represent the model's "confidence" that the image corresponds to each of the 10 different articles of clothing. You can see which label has the highest confidence value:

In [19]:
```
np.argmax(predictions[0])
```

Out[19]:
```
9
```

So, the model is most confident that this image is an ankle boot, or `class_names[9]` . Examining the test label shows that this classification is correct:

In [20]:
```
test_labels[0]
```

Out[20]:
```
9
```

Graph this to look at the full set of 10 class predictions.

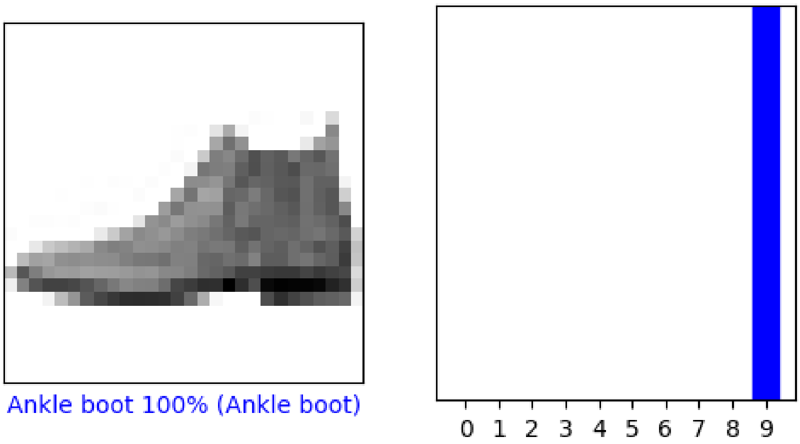
In [21]:
```
def plot_image(i, predictions_array, true_label, img):
  true_label, img = true_label[i], img[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])

  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')
```

Verify predictions

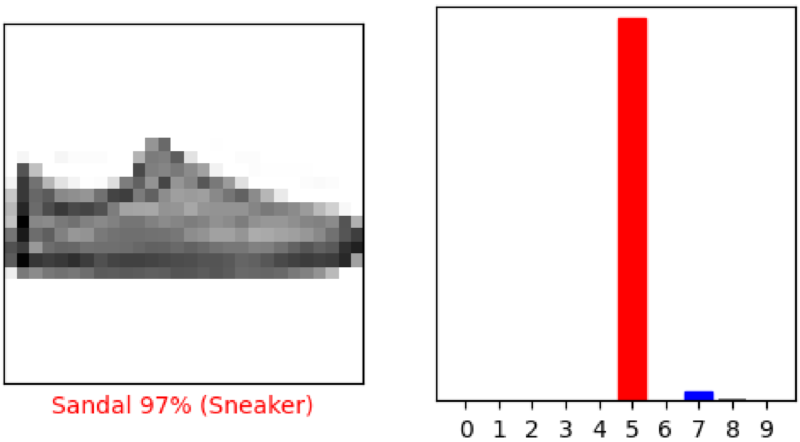
With the model trained, you can use it to make predictions about some images.

Let's look at the 0th image, predictions, and prediction array. Correct prediction labels are blue and incorrect prediction labels are red. The number gives the percentage (out of 100) for the predicted label.

In [22]:
```python
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()
```

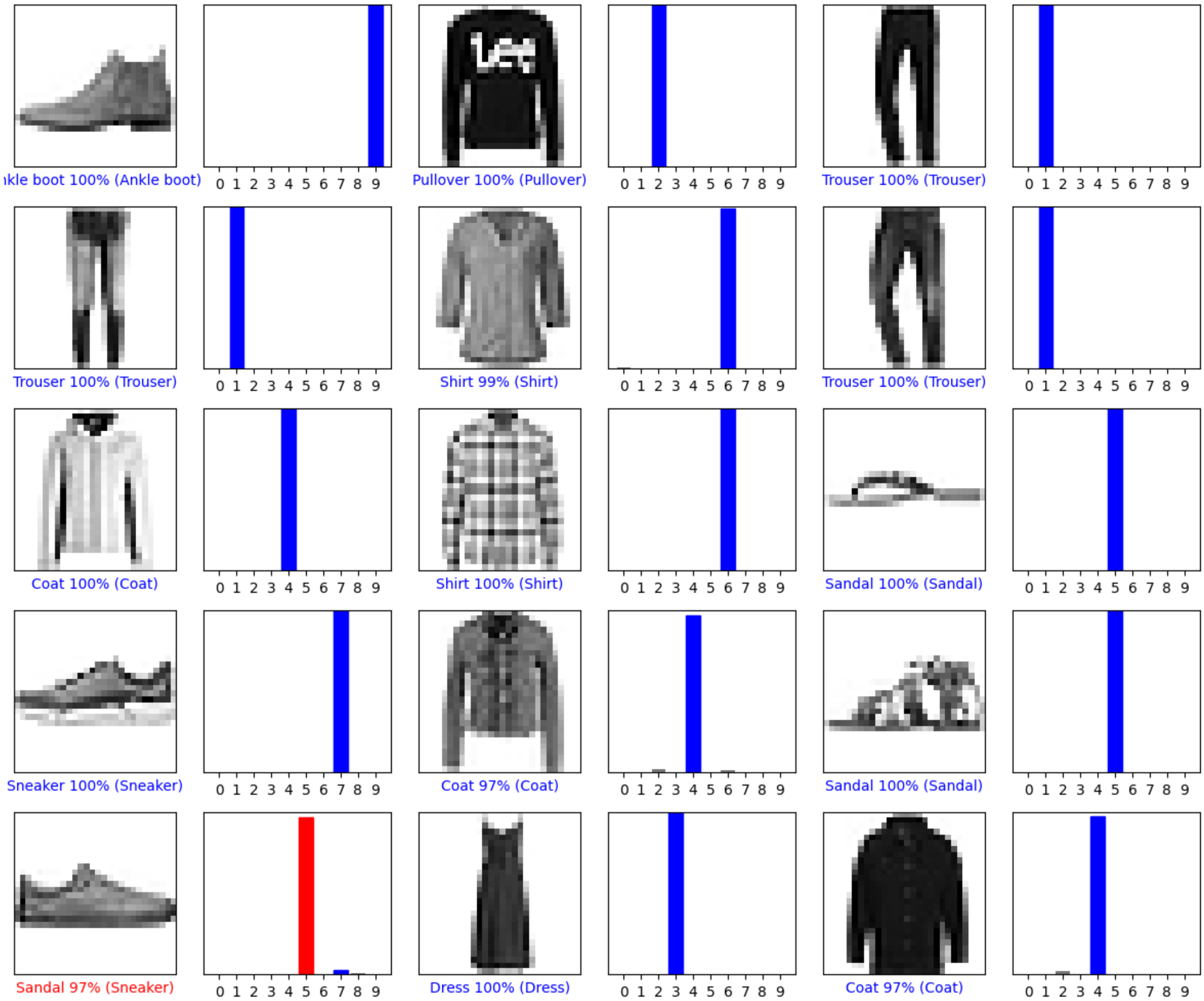


In [23]:
```python
i = 12
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i],  test_labels)
plt.show()
```



Let's plot several images with their predictions. Note that the model can be wrong even when very confident.

In [24]:
```python
# 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()
```



Use the trained model

Finally, use the trained model to make a prediction about a single image.

In [25]:
```python
# Grab an image from the test dataset.
img = test_images[1]

print(img.shape)
```

```
(28, 28)
```

`tf.keras` models are optimized to make predictions on a *batch*, or collection, of examples at once. Accordingly, even though you're using a single image, you need to add it to a list:

In [26]:
```python
# Add the image to a batch where it's the only member.
img = (np.expand_dims(img,0))

print(img.shape)
```

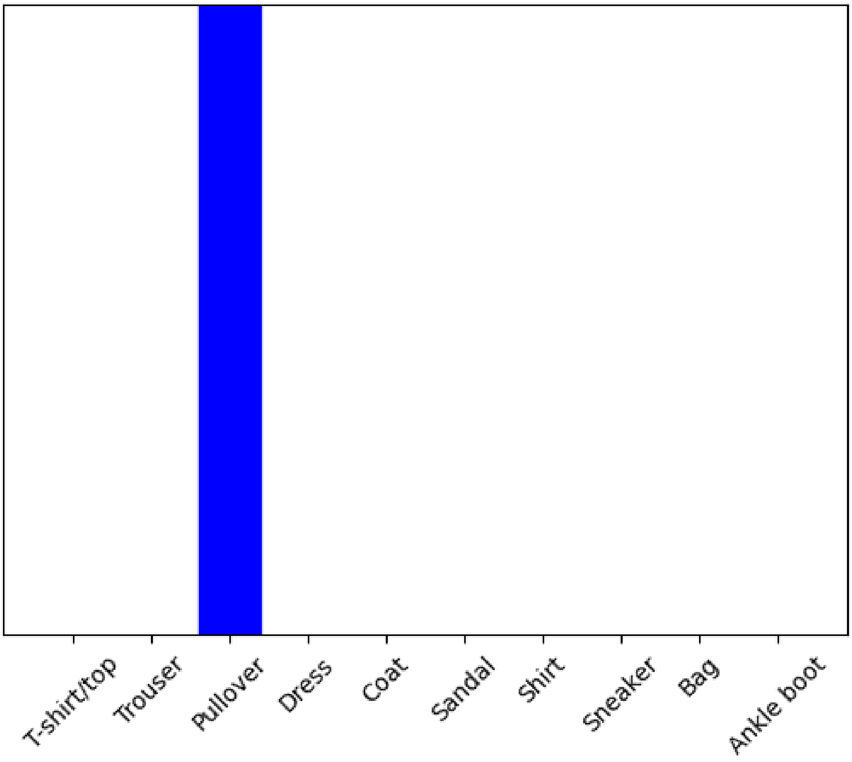
```
(1, 28, 28)
```

Now predict the correct label for this image:

In [27]:
```python
predictions_single = probability_model.predict(img)

print(predictions_single)
```

```
1/1 [==============================] - 0s 19ms/step
[[2.3126481e-06 1.3403139e-21 9.9972457e-01 6.3178419e-14 2.6131392e-04
  7.6748816e-20 1.1800360e-05 6.7637641e-20 1.6529707e-15 6.0258504e-21]]
```

In [28]:
```python
plot_value_array(1, predictions_single[0], test_labels)
_ = plt.xticks(range(10), class_names, rotation=45)
plt.show()
```



`tf.keras.Model.predict` returns a list of lists—one list for each image in the batch of data. Grab the predictions for our (only) image in the batch:

In [29]:
```python
np.argmax(predictions_single[0])
```

Out[29]: 2

And the model predicts a label as expected.