Introduction to TensorFlow

TensorFlow is a deep learning library developed by Google. It was originally developed by a team within Google called the Google Brain team for their internal use and was subsequently open sourced in 2015. After being in Alpha/Beta release for a long time, the final version of TensorFlow 2.0 was released on September 30, 2019. The focus of TF2.0 was to make the development of deep learning applications easier.

Exercise 1: Implementing a Mathematical Equation

In this exercise, we will solve the following mathematical equation using TensorFlow:

$$Z = X^2 Y + Y + 2$$

Mathematical equation to be solved using TensorFlow

We will use TensorFlow to solve it, as follows:

```
X=3
Y=4
```

While there are multiple ways of doing this, we will only explore one of the ways in this exercise. Follow these steps to complete this exercise:

1. Open a Jupyter Notebook and import the TensorFlow library using the following command:
   ```
   import tensorflow as tf
   ```
2. Now, let's solve the equation. For that, you will need to create two variables, **`X`** and **`Y`**, and initialize them to the given values of **`3`** and **`4`**, respectively:
   ```
   X=tf.Variable(3)
   ```

```
Y=tf.Variable(4)
```

3. In our equation, the value of **2** isn't changing, so we'll store it as a constant by typing the following code:
   ```
   C=tf.constant(2)
   ```
4. Define the function that will solve our equation:
   ```
   def myfunc(x,y,c):
       Z=x*x*y+y+c
       return Z
   ```
5. Call the function by passing **X**, **Y**, and **C** as parameters. We'll be storing the output of this function in a variable called **result**:
   ```
   result=myfunc(X,Y,C)
   ```
6. Print the result using the **tf.print()** function:
   ```
   tf.print(result)
   ```
   The output will be as follows:
   ```
   42
   ```

In this exercise, we learned how to define and use a function. You will notice that it is not a lot different from normal Python code.

Exercise 2: Matrix Multiplication Using TensorFlow

In this exercise, we will use the **tf.matmul()** method to multiply two matrices using **tensorflow**. Follow these steps to complete this exercise:

1. Import the **tensorflow** library and create two variables, **X** and **Y**, as matrices. **X** is a 2 x 3 matrix and **Y** is a 3 x 2 matrix:
   ```
   import tensorflow as tf
   X=tf.Variable([[1,2,3],[4,5,6]])
   Y=tf.Variable([[7,8],[9,10],[11,12]])
   ```
2. Print and display the values of **X** and **Y** to make sure the matrices are created correctly. We'll start by printing the value of **X**:
   ```
   tf.print(X)
   ```

The output will be as follows:

```
[[1 2 3]
 [4 5 6]]
```

Now, let's print the value of **`Y`**:

```
tf.print(Y)
```

The output will be as follows:

```
[[7 8]
 [9 10]
 [11 12]]
```

3. Perform matrix multiplication by calling the TensorFlow **`tf.matmul()`** function:

```
c1=tf.matmul(X,Y)
```

To display the result, print the value of **`c1`**:

```
tf.print(c1)
```

The output will be as follows:

```
[[58 64]
 [139 154]]
```

4. Let's perform matrix multiplication by changing the order of the matrices:

```
c2=tf.matmul(Y,X)
```

To display the result, let's print the value of **`c2`**:

```
tf.print(c2)
```

The resulting output will be as follows.

```
[[39 54 69]
 [49 68 87]
 [59 82 105]]
```

Note that the results are different since we changed the order.

In this exercise, we learned how to create matrices in TensorFlow and how to perform matrix multiplication. This will come in handy when we create our own neural networks.

Exercise 3: Reshaping Matrices Using the reshape() Function in TensorFlow

In this exercise, we will reshape a **`[5,4]`** matrix into the shape of **`[5,4,1]`** using the **`reshape()`** function. This exercise will help us understand how **`reshape()`** can be used to change the rank of a tensor. Follow these steps to complete this exercise:

1. Import **`tensorflow`** and create the matrix we want to reshape:
   ```
   import tensorflow as tf
   A=tf.Variable([[1,2,3,4], \
                  [5,6,7,8], \
                  [9,10,11,12], \
                  [13,14,15,16], \
                  [17,18,19,20]])
   ```
2. First, we'll print the variable **`A`** to check whether it is created correctly, using the following command:
   ```
   tf.print(A)
   ```
   The output will be as follows:
   ```
   [[1 2 3 4]
    [5 6 7 8]
    [9 10 11 12]
    [13 14 15 16]
    [17 18 19 20]]
   ```
3. Let's print the shape of **`A`**, just to be sure:
   ```
   A.shape
   ```
   The output will be as follows:
   ```
   TensorShape([5, 4])
   ```
   Currently, it has a rank of 2. We'll be using the **`reshape()`** function to change its rank to 3.
4. Now, we will reshape **`A`** to the shape [5,4,1] using the following command. We've thrown in the **`print`** command just to see what the output looks like:
   ```
   tf.print(tf.reshape(A,[5,4,1]))
   ```
   We'll get the following output:
   ```
   [[[1]
     [2]
     [3]
   ```

```
  [4]]
 [[5]
  [6]
  [7]
  [8]]
 [[9]
  [10]
  [11]
  [12]]
 [[13]
  [14]
  [15]
  [16]]
 [[17]
  [18]
  [19]
  [20]]]
```

That worked as expected.

5. Let's see the new shape of **A**:

```
A.shape
```

The output will be as follows:

```
TensorShape([5, 4])
```

We can see that **A** still has the same shape. Remember that we discussed that in order to save the new shape, we need to assign it to itself. Let's do that in the next step.

6. Here, we'll assign the new shape to **A**:

```
A = tf.reshape(A,[5,4,1])
```

7. Let's check the new shape of **A** once again:

```
A.shape
```

We will see the following output:

```
TensorShape([5, 4, 1])
```

With that, we have not just reshaped the matrix but also changed its rank from 2 to 3. In the next step, let's print out the contents of **A** just to be sure.

8. Let's see what **A** contains now:

```
tf.print(A)
```

The output, as expected, will be as follows:

```
[[[1]
  [2]
  [3]
  [4]]
 [[5]
  [6]
  [7]
  [8]]
 [[9]
  [10]
  [11]
  [12]]
 [[13]
  [14]
  [15]
  [16]]
 [[17]
  [18]
  [19]
  [20]]]
```

In this exercise, we saw how to use the **`reshape()`** function. Using **`reshape()`**, we can change the rank and shape of tensors. We also learned that reshaping a matrix changes the shape of the matrix without changing the order of the elements within the matrix.

Exercise 4: Implementing the argmax() Function

In this exercise, we are going to use the **`argmax`** function to find the position of the maximum value in a given matrix along axes 0 and 1. Follow these steps to complete this exercise:

1. Import **`tensorflow`** and create the following matrix:
   ```
   import tensorflow as tf
   X=tf.Variable([[91,12,15], [11,88,21],[90, 87,75]])
   ```
2. Let's print **`X`** and see what the matrix looks like:
   ```
   tf.print(X)
   ```
   The output will be as follows:

```
[[91 12 15]
 [11 88 21]
 [90 87 75]]
```

3. Print the shape of **`X`**:

   ```
   X.shape
   ```

   The output will be as follows:

   ```
   TensorShape([3, 3])
   ```

4. Now, let's use **`argmax`** to find the positions of the maximum values while keeping **`axis`** as **`0`**:

   ```
   tf.print(tf.argmax(X,axis=0))
   ```

   The output will be as follows:

   ```
   [0 1 2]
   ```

   Referring to the matrix in *Step 2*, we can see that, moving across the columns, the index of the maximum value (91) in the first column is 0. Similarly, the index of the maximum value along the second column (88) is 1. And finally, the maximum value across the third column (75) has index 2. Hence, we have the aforementioned output.

5. Now, let's change the **`axis`** to **`1`**:

   ```
   tf.print(tf.argmax(X,axis=1))
   ```

   The output will be as follows:

   ```
   [0 1 0]
   ```

Again, referring to the matrix in *Step 2*, if we move along the rows, the maximum value along the first row is 91, which is at index 0. Similarly, the maximum value along the second row is 88, which is at index 1. Finally, the third row is at index 0 again, with a maximum value of 75.

In this exercise, we learned how to use the **`argmax`** function to find the position of the maximum value along a given axis of a tensor.

Exercise 5: Using an Optimizer for a Simple Linear Regression

In this exercise, we are going to see how to use an optimizer to train a simple linear regression model. We will start off by assuming arbitrary values for the parameters (**w** and **b**) in a linear equation **w*x + b**. Using the optimizer, we will observe how the values of the parameters change to get to the right parameter values, thus mapping the relationship between the input values (**x**) and output (**y**). Using the optimized parameter values, we will predict the output (**y**) for some given input values (**x**). After completing this exercise, we will see that the linear output, which is predicted by the optimized parameters, is very close to the real values of the output values. Follow these steps to complete this exercise:

1. Import **tensorflow**, create the variables, and initialize them to 0. Here, our assumed values are zero for both these parameters:
   ```
   import tensorflow as tf
   w=tf.Variable(0.0)
   b=tf.Variable(0.0)
   ```
2. Define a function for the linear regression model. We learned how to create functions in TensorFlow earlier:
   ```
   def regression(x):
       model=w*x+b
       return model
   ```
3. Prepare the data in the form of features (**x**) and labels (**y**):
   ```
   x=[1,2,3,4]
   y=[0,-1,-2,-3]
   ```
4. Define the **loss** function. In this case, this is the absolute value of the difference between the predicted value and the label:
   ```
   loss=lambda:abs(regression(x)-y)
   ```
5. Create an **Adam** optimizer instance with a learning rate of **.01**. The learning rate defines at what rate the optimizer should change the assumed parameters. We will discuss the learning rate in subsequent chapters:
   ```
   optimizer=tf.optimizers.Adam(.01)
   ```

6. Train the model by running the optimizer for 1,000 iterations to minimize the loss:

```
for i in range(1000):
    optimizer.minimize(loss,[w,b])
```

7. Print the trained values of the **w** and **b** parameters:

```
tf.print(w,b)
```

The output will be as follows:

```
-1.00371706 0.999803364
```

We can see that the values of the **w** and **b** parameters have been changed from their original values of 0, which were assumed. This is what is done during the optimizing process. These updated parameter values will be used for predicting the values of **Y**.

Note

The optimization process is stochastic in nature (having a random probability distribution), and you might get values for **w** *and* **b** *that are different to the value that was printed here.*

8. Use the trained model to predict the output by passing in the **x** values. The model predicts the values, which are very close to the label values (**y**), which means the model was trained to a high level of accuracy:

```
tf.print(regression([1,2,3,4]))
```

The output of the preceding command will be as follows:

```
[-0.00391370058 -1.00763083 -2.01134801 -3.01506495]
```

In this exercise, we saw how to use an optimizer to train a simple linear regression model. During this exercise, we saw how the initially assumed values of the parameters were updated to get the true values. Using the true values of the parameters, we were able to get the predictions close to the actual values. Understanding how to apply the optimizer will help you with training neural network models.

Regression and Classification Models

In these exercises, you will learn how to build regression models using TensorFlow. You will build models with TensorFlow utilizing Keras layers, which are a simple approach to model building that offer a high-level API for building and training models.

Exercise 6: Creating an ANN with TensorFlow

In this exercise, you will create your first sequential ANN in TensorFlow. You will have an input layer, a hidden layer with four units and a ReLU activation function, and an output layer with one unit. Then, you will create some simulation data by generating random numbers and passing it through the model, using the model's **`predict`** method to simulate a prediction for each data example.

Perform the following steps to complete the exercise:

1. Import the TensorFlow library:
   ```
   import tensorflow as tf
   ```
2. Initialize a Keras model of the sequential class:
   ```
   model = tf.keras.Sequential()
   ```
3. Add an input layer to the model using the model's **`add`** method, and add the **`input_shape`** argument with size **`(8,)`** to represent input data with eight features:
   ```
   model.add(tf.keras.layers.InputLayer(input_shape=(8,)
   , \
                                        name='Input_laye
   r'))
   ```
4. Add two layers of the **`Dense`** class to the model. The first will represent your hidden layer with four units and a ReLU

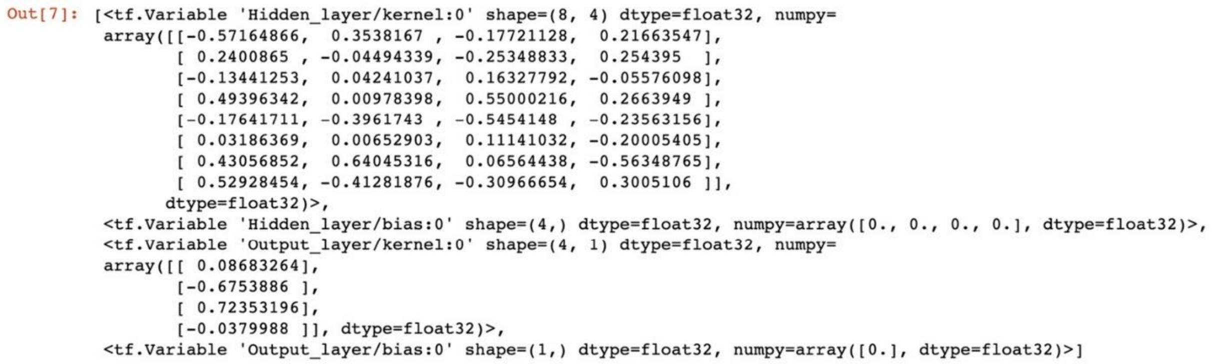
activation function, and the second will represent your output layer with one unit:

```
model.add(tf.keras.layers.Dense(4, activation='relu', \
                                  name='First_hidden_la
yer'))
model.add(tf.keras.layers.Dense(1,
name='Output_layer'))
```

5. View the weights by calling the **`variables`** attribute of the model:

```
model.variables
```

You should get the following output:



```
Out[7]: [<tf.Variable 'Hidden_layer/kernel:0' shape=(8, 4) dtype=float32, numpy=
         array([[-0.57164866,  0.3538167 , -0.17721128,  0.21663547],
                [ 0.2400865 , -0.04494339, -0.25348833,  0.254395  ],
                [-0.13441253,  0.04241037,  0.16327792, -0.05576098],
                [ 0.49396342,  0.00978398,  0.55000216,  0.2663949 ],
                [-0.17641711, -0.3961743 , -0.5454148 , -0.23563156],
                [ 0.03186369,  0.00652903,  0.11141032, -0.20005405],
                [ 0.43056852,  0.64045316,  0.06564438, -0.56348765],
                [ 0.52928454, -0.41281876, -0.30966654,  0.3005106 ]],
               dtype=float32)>,
         <tf.Variable 'Hidden_layer/bias:0' shape=(4,) dtype=float32, numpy=array([0., 0., 0., 0.], dtype=float32)>,
         <tf.Variable 'Output_layer/kernel:0' shape=(4, 1) dtype=float32, numpy=
         array([[ 0.08683264],
                [-0.6753886 ],
                [ 0.72353196],
                [-0.0379988 ]], dtype=float32)>,
         <tf.Variable 'Output_layer/bias:0' shape=(1,) dtype=float32, numpy=array([0.], dtype=float32)>]
```

The variables of the ANN

This output shows all the variables that compose the model; they include the values for all weights and biases in each layer.

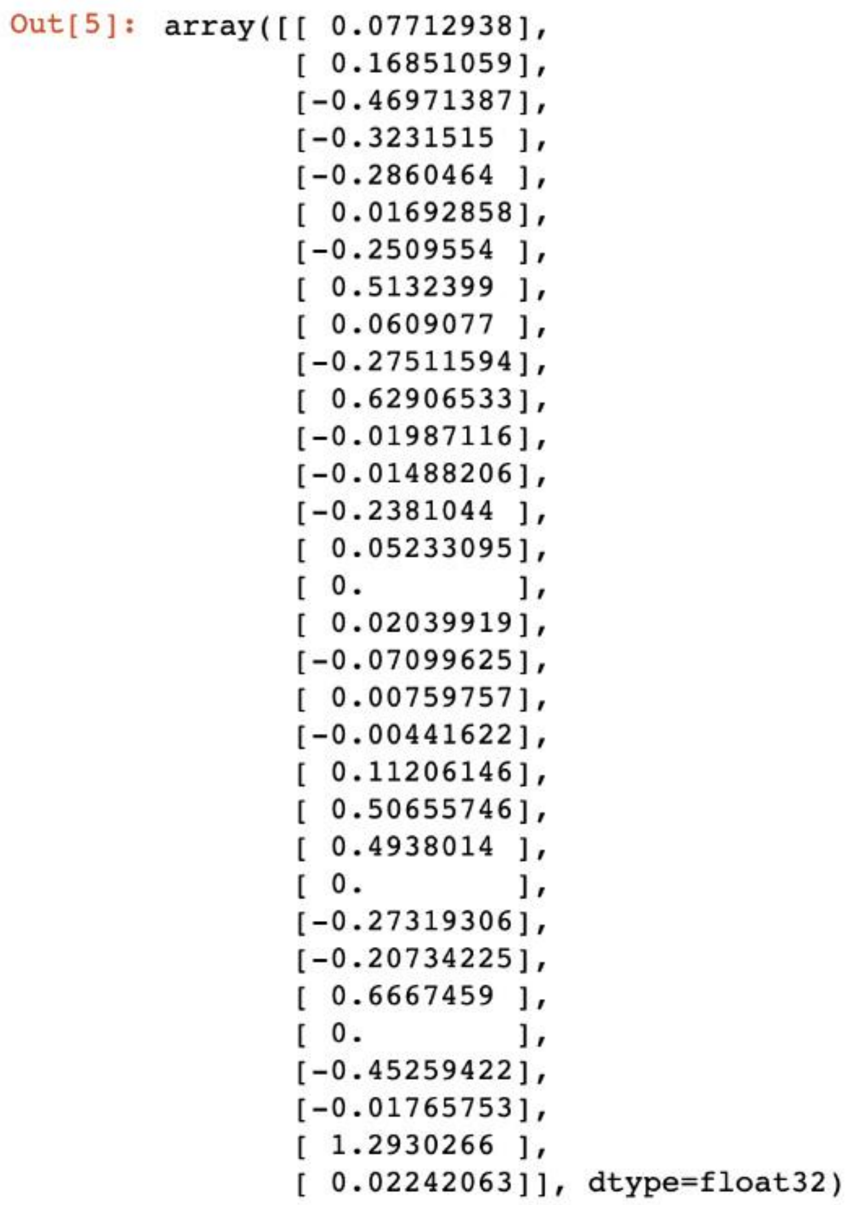
6. Create a tensor of size **`32x8`**, which represents a tensor with 32 records and 8 features:

```
data = tf.random.normal((32,8))
```

7. Call the **`predict`** method of the model and pass in the sample data:

```
model.predict(data)
prediction
```

You should get the following result:



```
Out[5]: array([[ 0.07712938],
               [ 0.16851059],
               [-0.46971387],
               [-0.3231515 ],
               [-0.2860464 ],
               [ 0.01692858],
               [-0.2509554 ],
               [ 0.5132399 ],
               [ 0.0609077 ],
               [-0.27511594],
               [ 0.62906533],
               [-0.01987116],
               [-0.01488206],
               [-0.2381044 ],
               [ 0.05233095],
               [ 0.        ],
               [ 0.02039919],
               [-0.07099625],
               [ 0.00759757],
               [-0.00441622],
               [ 0.11206146],
               [ 0.50655746],
               [ 0.4938014 ],
               [ 0.        ],
               [-0.27319306],
               [-0.20734225],
               [ 0.6667459 ],
               [ 0.        ],
               [-0.45259422],
               [-0.01765753],
               [ 1.2930266 ],
               [ 0.02242063]], dtype=float32)
```

The output of the ANN after random inputs have been applied

Calling the **`predict()`** method on the sample data will propagate the data through the network. In each layer, there will be a matrix multiplication of the data with the weights, and the bias will be added before the data is passed as input data to the next layer. This process continues until the final output layer.

In this exercise, you created a sequential model with multiple layers. You initialized a model, added an input layer to accept data with eight features, added a hidden layer with four units, and added an output layer with one unit. Before fitting a model to training data, you must first compile the model with an optimizer and choose a loss function to minimize the value it computes by updating weights in the training process.

Exercise 7: Creating a Linear Regression Model as an ANN with TensorFlow

In this exercise, you will create a linear regression model as an ANN using TensorFlow. The dataset, **`Bias_correction_ucl.csv`**, describes the bias correction of air temperature forecasts of Seoul, South Korea. The fields represent temperature measurements of the given date, the weather station at which the metrics were measured, model forecasts of weather-related metrics such as humidity, and projections for the temperature the following day. You are required to predict the next maximum and minimum temperature given measurements of the prior timepoints and attributes of the weather station.

Perform the following steps to complete this exercise:

1. Import the TensorFlow and pandas libraries:
```
import tensorflow as tf
import pandas as pd
```
2. Load in the dataset using the pandas **`read_csv`** function:
```
df = pd.read_csv('Bias_correction_ucl.csv')
```
3. Drop the **`date`** column and drop any rows that have null values since your model requires numerical values only:
```
df.drop('Date', inplace=True, axis=1)
df.dropna(inplace=True)
```
4. Create target and feature datasets. The target dataset will contain the columns named **`Next_Tmax`** and **`Next_Tmin`**,

while the feature dataset will contain all columns except those named **`Next_Tmax`** and **`Next_Tmin`**:

```
target = df[['Next_Tmax', 'Next_Tmin']]
features = df.drop(['Next_Tmax', 'Next_Tmin'],
axis=1)
```

5. Rescale the feature dataset:

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
feature_array = scaler.fit_transform(features)
features = pd.DataFrame(feature_array,
columns=features.columns)
```

6. Initialize a Keras model of the **`Sequential`** class:

```
model = tf.keras.Sequential()
```

7. Add an input layer to the model using the model's **`add`** method, and set **`input_shape`** to be the number of columns in the feature dataset:

```
model.add(tf.keras.layers.InputLayer\
         (input_shape=(features.shape[1],), \
                        name='Input_layer'))
```

8. Add the output layer of the **`Dense`** class to the model with a size of **`2`**, representing the two target variables:

```
model.add(tf.keras.layers.Dense(2,
name='Output_layer'))
```

9. Compile the model with an RMSprop optimizer and a mean squared error loss:

```
model.compile(tf.optimizers.RMSprop(0.001),
loss='mse')
```

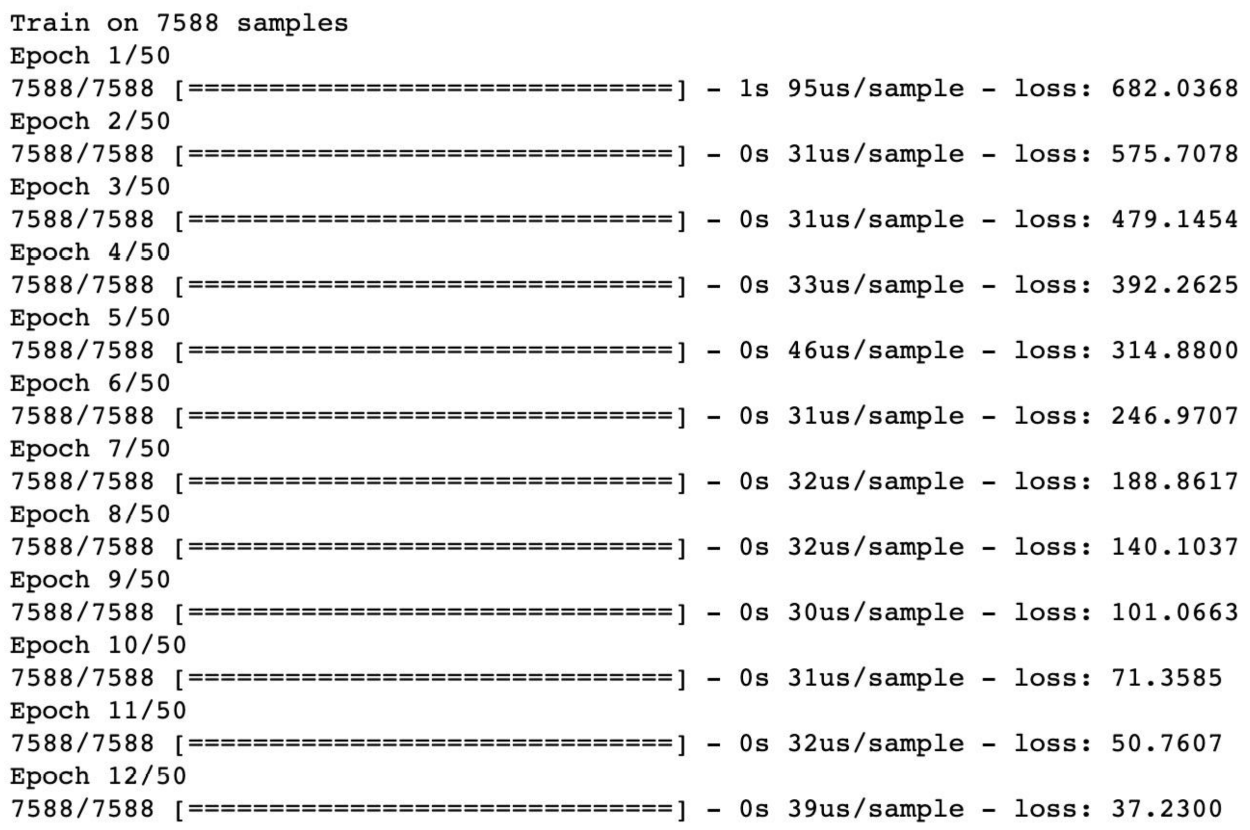
10. Add a callback for TensorBoard:

```
tensorboard_callback = tf.keras.callbacks\
                         .TensorBoard(log_dir="./logs
")
```

11. Fit the model to the training data:

```
model.fit(x=features.to_numpy(),
y=target.to_numpy(),\
          epochs=50,
callbacks=[tensorboard_callback])
```

You should get the following output:



```
Train on 7588 samples
Epoch 1/50
7588/7588 [==============================] - 1s 95us/sample - loss: 682.0368
Epoch 2/50
7588/7588 [==============================] - 0s 31us/sample - loss: 575.7078
Epoch 3/50
7588/7588 [==============================] - 0s 31us/sample - loss: 479.1454
Epoch 4/50
7588/7588 [==============================] - 0s 33us/sample - loss: 392.2625
Epoch 5/50
7588/7588 [==============================] - 0s 46us/sample - loss: 314.8800
Epoch 6/50
7588/7588 [==============================] - 0s 31us/sample - loss: 246.9707
Epoch 7/50
7588/7588 [==============================] - 0s 32us/sample - loss: 188.8617
Epoch 8/50
7588/7588 [==============================] - 0s 32us/sample - loss: 140.1037
Epoch 9/50
7588/7588 [==============================] - 0s 30us/sample - loss: 101.0663
Epoch 10/50
7588/7588 [==============================] - 0s 31us/sample - loss: 71.3585
Epoch 11/50
7588/7588 [==============================] - 0s 32us/sample - loss: 50.7607
Epoch 12/50
7588/7588 [==============================] - 0s 39us/sample - loss: 37.2300
```

The output of the fitting process showing the epoch, train time per sample, and loss after each epoch

12. Evaluate the model on the training data:

```
loss = model.evaluate(features.to_numpy(),
target.to_numpy())
print('loss:', loss)
```

This results in the following output:

```
loss: 3.5468221449764012
```

In this exercise, you have learned how to create, train, and evaluate an ANN with TensorFlow by using Keras layers. You recreated the linear regression algorithm by creating an ANN with an input layer and an output layer that has one unit for each output. Here, there were two outputs representing the maximum and minimum values of the temperature; thus, the output layer has two units.

In the next exercise, you will create and train ANNs that have multiple hidden layers.

Exercise 8: Creating a Multi-Layer ANN with TensorFlow

In this exercise, you will create a multi-layer ANN using TensorFlow. This model will have four hidden layers. You will add multiple layers to the model and activation functions to the output of the layers. The first hidden layer will have `16` units, the second will have `8` units, and the third will have `4` units. The output layer will have `2` units. You will utilize the same dataset as in *Exercise 6*, *Creating a Linear Regression Model as an ANN with TensorFlow*, which describes the bias correction of air temperature forecasts for Seoul, South Korea. The exercise aims to predict the next maximum and minimum temperature given measurements of the prior timepoints and attributes of the weather station.

Perform the following steps to complete this exercise:

1. Import the TensorFlow and pandas libraries:
```
import tensorflow as tf
import pandas as pd
```
2. Load in the dataset using the pandas `read_csv` function:
```
df = pd.read_csv('Bias_correction_ucl.csv')
```
3. Drop the `Date` column and drop any rows that have null values:
```
df.drop('Date', inplace=True, axis=1)
df.dropna(inplace=True)
```
4. Create target and feature datasets:
```
target = df[['Next_Tmax', 'Next_Tmin']]
features = df.drop(['Next_Tmax', 'Next_Tmin'],
axis=1)
```
5. Rescale the feature dataset:
```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
```

```
feature_array = scaler.fit_transform(features)
features = pd.DataFrame(feature_array,
columns=features.columns)
```

6. Initialize a Keras model of the **`Sequential`** class:

```
model = tf.keras.Sequential()
```

7. Add an input layer to the model using the model's **`add`** method, and set **`input_shape`** to the number of columns in the feature dataset:

```
model.add(tf.keras.layers.InputLayer\
                             (input_shape=(features.shape
[1],), \
                              name='Input_layer'))
```

8. Add three hidden layers and an output layer of the **`Dense`** class to the model. The first hidden layer will have **`16`** units, the second will have **`8`** units, and the third will have **`4`** units. Label the layers appropriately. The output layer will have two units to match the target variable that has two columns:

```
model.add(tf.keras.layers.Dense(16,
name='Dense_layer_1'))
model.add(tf.keras.layers.Dense(8,
name='Dense_layer_2'))
model.add(tf.keras.layers.Dense(4,
name='Dense_layer_3'))
model.add(tf.keras.layers.Dense(2,
name='Output_layer'))
```

9. Compile the model with an RMSprop optimizer and mean squared error loss:

```
model.compile(tf.optimizers.RMSprop(0.001),
loss='mse')
```

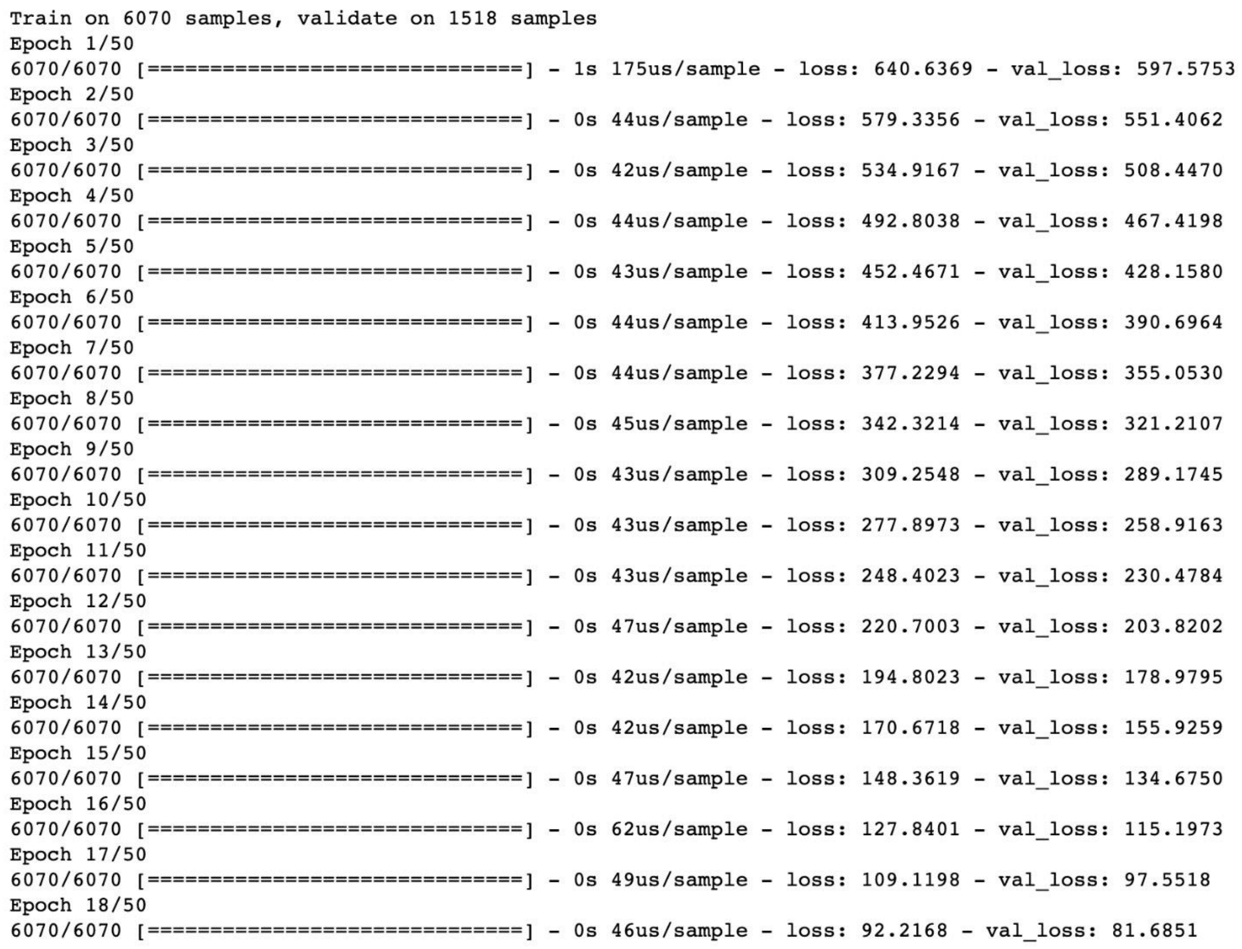
10. Add a callback for TensorBoard:

```
tensorboard_callback = tf.keras.callbacks\
                             .TensorBoard(log_dir="./logs
")
```

11. Fit the model to the training data for **`50`** epochs and add a validation split equal to 20%:

```
model.fit(x=features.to_numpy(),
y=target.to_numpy(),\
          epochs=50, callbacks=[tensorboard_callback]
, \
          validation_split=0.2)
```

You should get the following output:



```
Train on 6070 samples, validate on 1518 samples
Epoch 1/50
6070/6070 [==============================] - 1s 175us/sample - loss: 640.6369 - val_loss: 597.5753
Epoch 2/50
6070/6070 [==============================] - 0s 44us/sample - loss: 579.3356 - val_loss: 551.4062
Epoch 3/50
6070/6070 [==============================] - 0s 42us/sample - loss: 534.9167 - val_loss: 508.4470
Epoch 4/50
6070/6070 [==============================] - 0s 44us/sample - loss: 492.8038 - val_loss: 467.4198
Epoch 5/50
6070/6070 [==============================] - 0s 43us/sample - loss: 452.4671 - val_loss: 428.1580
Epoch 6/50
6070/6070 [==============================] - 0s 44us/sample - loss: 413.9526 - val_loss: 390.6964
Epoch 7/50
6070/6070 [==============================] - 0s 44us/sample - loss: 377.2294 - val_loss: 355.0530
Epoch 8/50
6070/6070 [==============================] - 0s 45us/sample - loss: 342.3214 - val_loss: 321.2107
Epoch 9/50
6070/6070 [==============================] - 0s 43us/sample - loss: 309.2548 - val_loss: 289.1745
Epoch 10/50
6070/6070 [==============================] - 0s 43us/sample - loss: 277.8973 - val_loss: 258.9163
Epoch 11/50
6070/6070 [==============================] - 0s 43us/sample - loss: 248.4023 - val_loss: 230.4784
Epoch 12/50
6070/6070 [==============================] - 0s 47us/sample - loss: 220.7003 - val_loss: 203.8202
Epoch 13/50
6070/6070 [==============================] - 0s 42us/sample - loss: 194.8023 - val_loss: 178.9795
Epoch 14/50
6070/6070 [==============================] - 0s 42us/sample - loss: 170.6718 - val_loss: 155.9259
Epoch 15/50
6070/6070 [==============================] - 0s 47us/sample - loss: 148.3619 - val_loss: 134.6750
Epoch 16/50
6070/6070 [==============================] - 0s 62us/sample - loss: 127.8401 - val_loss: 115.1973
Epoch 17/50
6070/6070 [==============================] - 0s 49us/sample - loss: 109.1198 - val_loss: 97.5518
Epoch 18/50
6070/6070 [==============================] - 0s 46us/sample - loss: 92.2168 - val_loss: 81.6851
```

The output of the fitting process showing the epoch, training time per sample, and loss after each epoch

12. Evaluate the model on the training data:

```
loss = model.evaluate(features.to_numpy(),
target.to_numpy())
print('loss:', loss)
```

This will display the following result:

```
loss: 1.664448248190068
```

In this exercise, you have created an ANN with multiple hidden layers. The loss you obtained was lower than that achieved using linear regression, which demonstrates the power of ANNs. With some tuning to the hyperparameters (such as varying the number of layers, the number of units within each layer, adding activation functions, and changing the loss and optimizer), the loss could be even lower.

Classification Models

The goal of classification models is to classify data into distinct classes. Classification models can classify data into more than two distinct classes (known as **multi-class classification**) or classify data with multiple positive labels (known as **multi-label classification**).

Exercise 9: Creating a Logistic Regression Model as an ANN with TensorFlow

In this exercise, you will create a logistic regression model as an ANN using TensorFlow. The dataset, **`qsar_androgen_receptor.csv`**, is used to develop classification models for the discrimination of binder/non-binder molecules given various attributes of the molecules. Here, the molecule attributes represent the features of your dataset, and their binding properties represent the target variable, in which a positive value represents a binding molecule, and a negative value represents a non-binding molecule. You will create a logistic regression model to predict the binding properties of the molecule given attributes of the molecule provided in the dataset.

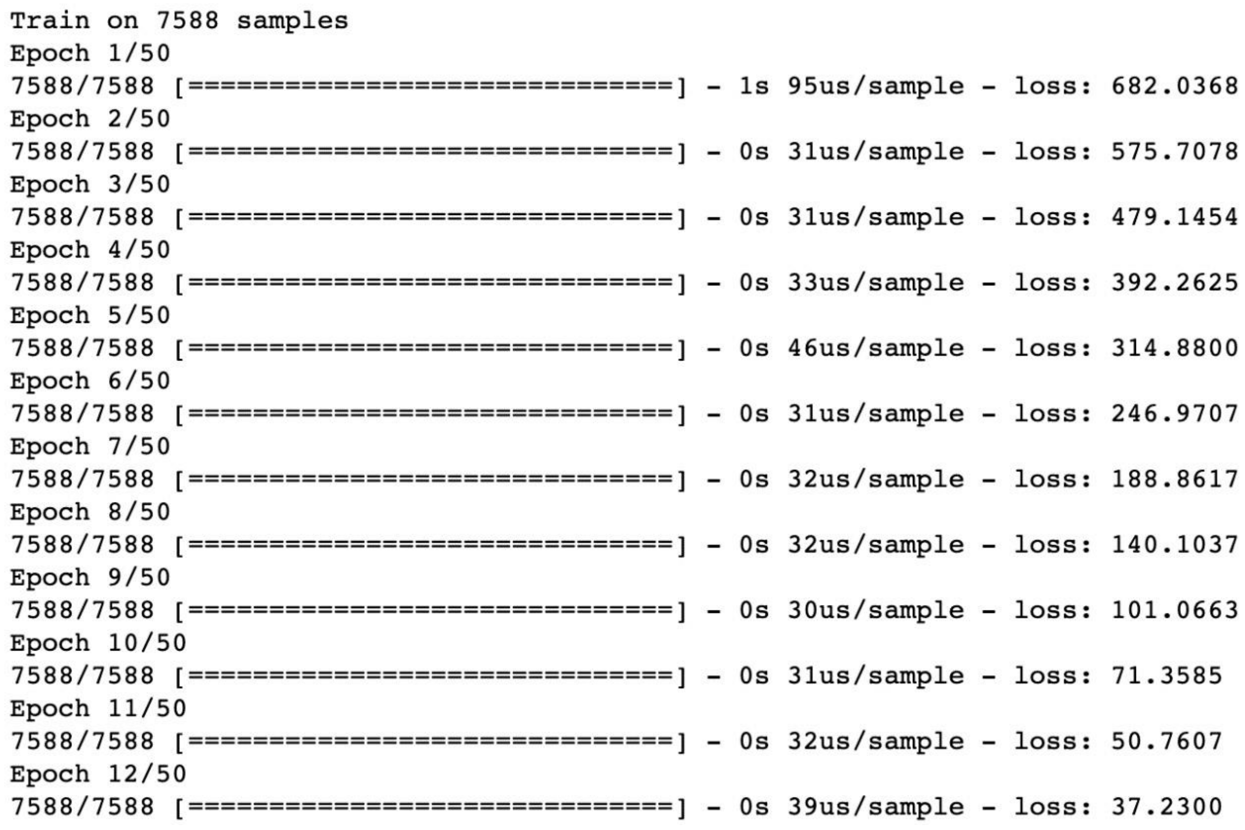
Perform the following steps to complete this exercise:

1. Import the TensorFlow and pandas libraries:
   ```
   import tensorflow as tf
   import pandas as pd
   ```

2. Load in the dataset using the pandas **read_csv** function:
   ```
   df = pd.read_csv('qsar_androgen_receptor.csv', \
                    sep=';')
   ```
3. Drop any rows that have null values:
   ```
   df.dropna(inplace=True)
   ```
4. Create target and feature datasets:
   ```
   target = df['positive'].apply(lambda x: 1 if
   x=='positive' else 0)
   features = df.drop('positive', axis=1)
   ```
5. Initialize a Keras model of the **Sequential** class:
   ```
   model = tf.keras.Sequential()
   ```
6. Add an input layer to the model using the model's **add** method and set **input_shape** to be the number of columns in the feature dataset:
   ```
   model.add(tf.keras.layers.InputLayer\
            (input_shape=(features.shape[1],), \
                          name='Input_layer'))
   ```
7. Add the output layer of the **Dense** class to the model with a size of **1**, representing the target variable:
   ```
   model.add(tf.keras.layers.Dense(1,
   name='Output_layer', \
                                   activation='sigmoid')
   )
   ```
8. Compile the model with an RMSprop optimizer and binary cross-entropy for the loss, and compute the accuracy:
   ```
   model.compile(tf.optimizers.RMSprop(0.0001), \
                 loss='binary_crossentropy',
   metrics=['accuracy'])
   ```
9. Create a TensorBoard callback:
   ```
   tensorboard_callback =
   tf.keras.callbacks.TensorBoard\
                          (log_dir="./logs")
   ```
10. Fit the model to the training data for **50** epochs, adding the TensorBoard callback with a validation split of 20%:
   ```
   model.fit(x=features.to_numpy(), y=target.to_numpy(),
   \
   ```

```
          epochs=50, callbacks=[tensorboard_callback]
, \
          validation_split=0.2)
```

Your output should be similar to the following figure:



```
Train on 7588 samples
Epoch 1/50
7588/7588 [==============================] - 1s 95us/sample - loss: 682.0368
Epoch 2/50
7588/7588 [==============================] - 0s 31us/sample - loss: 575.7078
Epoch 3/50
7588/7588 [==============================] - 0s 31us/sample - loss: 479.1454
Epoch 4/50
7588/7588 [==============================] - 0s 33us/sample - loss: 392.2625
Epoch 5/50
7588/7588 [==============================] - 0s 46us/sample - loss: 314.8800
Epoch 6/50
7588/7588 [==============================] - 0s 31us/sample - loss: 246.9707
Epoch 7/50
7588/7588 [==============================] - 0s 32us/sample - loss: 188.8617
Epoch 8/50
7588/7588 [==============================] - 0s 32us/sample - loss: 140.1037
Epoch 9/50
7588/7588 [==============================] - 0s 30us/sample - loss: 101.0663
Epoch 10/50
7588/7588 [==============================] - 0s 31us/sample - loss: 71.3585
Epoch 11/50
7588/7588 [==============================] - 0s 32us/sample - loss: 50.7607
Epoch 12/50
7588/7588 [==============================] - 0s 39us/sample - loss: 37.2300
```

The output of the fitting process showing the epoch, training time per sample, and loss after each epoch

11. Evaluate the model on the training data:

```
loss, accuracy = model.evaluate(features.to_numpy(),
\
                                   target.to_numpy())
print(f'loss: {loss}, accuracy: {accuracy}')
```

You should get output something like the following:

```
loss: 0.2781583094794838, accuracy:
0.9110320210456848
```

In this exercise, you have learned how to build a classification model to discriminate between the binding properties of various molecules based on their other molecular attributes. The classification model was equivalent to a logistic regression model since it had only one layer and was preceded by a sigmoid activation function. With only one layer, there is a weight for each input feature and a single value for the bias. The sigmoid activation function transforms the output of the layer into a value between `0` and `1`, which is then rounded to represent your two classes. `0.5` and above represents one class, the molecule with binding properties, and below `0.5` represents the other class, molecules with non-binding properties.