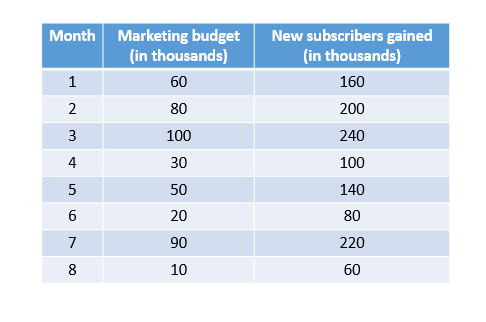
1. Problem is to find the New subscribers gained based on market budget



Since it has only one Feature (I/P 🡪 budget) and label (O/P 🡪 New subscribers) we see the relationship between the amount spent and new subscribers gained is

Subscribers gained = 2 \* Amount Spent + 40

Will solve the problem using TF:

|  |
| --- |
|  |
|  | tf.logging.set\_verbosity(tf.logging.ERROR) |
|  | import numpy as np  import tensorflow as tf |
|  | import matplotlib.pyplot as plt   |  |  | | --- | --- | |  | mar\_budget = np.array([60, 80, 100 , 30, 50, 20, 90, 10], dtype=float) #Feature | |  | subs\_gained = np.array([160, 200, 240, 100, 140, 80, 220, 60], dtype=float) #label | |  |  | |  | for i,c in enumerate(mar\_budget): | |  | print("{} Market budget = {} new subscribers gained".format(c, subs\_gained[i])) | |

Draw a scatter plot to visualize the data:

|  |  |
| --- | --- |
|  | plt.scatter(mar\_budget, subs\_gained) |
|  | plt.xlim(0,120) |
|  | plt.ylim(0,260) |
|  | plt.xlabel('Marketing Budget(in thousand of Dollars)') |
|  | plt.ylabel('Subscribers Gained(in thousand)') |
|  | plt.show() |

Split the data into training and testing Data.

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(mar\_budget,subs\_gained,random\_state=42,

train\_size=0.8, test\_size=0.2)

# **Creating the model:**

**Since the problem is straight forward,** this network will require only a single layer, with a single neuron. So, we will use the simplest possible model we can, a Dense network

Build a layer:

We’ll call the layer layer\_0 and create it by instantiating tf.keras.layers.Dense with the following configuration:

* input\_shape=[1]: This specifies that the input to this layer is a single value. That is, the shape is a one-dimensional array with one member. Since this is the first (and only) layer, the input shape is the input shape of the entire model. The single value is a floating-point number, representing marketing\_budget.
* units=1: This specifies the number of neurons in the layer. The number of neurons defines how many internal variables the layer has to try to learn how to solve the problem. Since this is the final layer, it is also the size of the model’s output — a single float value representing new subscribers gained. (In a multi-layered network, the size and shape of the layer would need to match the `input\_shape` of the next layer.)

layer\_0 = tf.keras.layers.Dense(units=1, input\_shape=[1])

Once layers are defined, they need to be assembled into a model. The Sequential model definition takes a list of layers as arguments specifying the calculation order from the input to the output.

Note: You will often see the layers defined inside the model definition, rather than beforehand as below:

model = tf.keras.Sequential([

tf.keras.layers.Dense(units=1, input\_shape=[1])

])

**Compile the model, with loss and optimizer functions:**

model.compile(loss='mean\_squared\_error',

optimizer=tf.keras.optimizers.Adam(0.1))

* Loss function: A way of measuring how far off predictions are from the desired outcome. (The measured difference is called the “loss”.)
* Optimizer function: A way of adjusting internal values to reduce the loss.

These parameters are used during training (model.fit(), below) to first calculate the loss at each point, and then improve it. The act of calculating the current loss of a model and then improving it is precisely what training is.

During training, the optimizer function is used to calculate adjustments to the model’s internal variables. The goal is to adjust the internal variables until the model (which is a math function) mirrors the actual equation for converting budget\_Spent to New Subs Gained.

1. The loss function ([mean squared error](https://en.wikipedia.org/wiki/Mean_squared_error)) and the optimizer ([Adam](https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/)) used here are standard for simple models like this one, but many others are available. It is not important to know how these specific functions work at this point.
2. One part of the Optimizer you may need to think about when building your models is the learning rate ( 0.1 in the code above). This is the step size taken when adjusting values in the model. If the value is too small, it will take too many iterations to train the model. Too large, and accuracy goes down. Finding a good value often involves some trial and error, but the range is usually within 0.001 (default), and 0.1.

The **epochs** argument specifies how many times this cycle should be run, and the **verbose** argument controls how much output the method produces.

During training, the model takes in marketing budget values, performs a calculation using the current internal variables (called “weights”) and outputs values which are meant to be the `New subs Gained`. Since the weights are initially set randomly, the output will not be close to the correct value. The difference between the actual output and the desired output is calculated using the loss function, and the optimizer function directs how the weights should be adjusted.

This cycle of calculating, comparing, adjusting is controlled by the `fit` method. The first argument is the inputs, the second argument is the desired outputs.

trained\_model = model.fit(X\_train, y\_train, epochs=1000, verbose=False)

print("Finished training the model")

**Display training statistics:**

The `fit` method returns a history object. We can use this object to plot how the loss of our model goes down after each training epoch. A **high loss** means that the value of **new subs gained** the model predicts, are far from the corresponding value of**actual subs gained**.

|  |  |
| --- | --- |
|  | import matplotlib.pyplot as plt |
|  | plt.xlabel('Epoch Number') |
|  | plt.ylabel("Loss Magnitude") |
|  | plt.plot(trained\_model.history['loss']) |

We can see o/p by print(model.predict([80.0]))

print(model.predict([80.0]))

|  |  |
| --- | --- |
|  | y\_pred = model.predict(X\_test) |
|  | print('Actual Values\tPredicted Values') |
|  | print(y\_test,' ',y\_pred.reshape(1,-1)) |

**Verifying the Model accuracy using Performance Metric**

r2\_score(r-squared value)

from sklearn.metrics import r2\_score

r2\_score(y\_test,y\_pred)

**A little Thought experiment**

Just for fun, what if we create a new model with 3 more Dense layers with different units, which therefore also has more variables?

l\_0 = tf.keras.layers.Dense(units=4, input\_shape=[1])

l\_1 = tf.keras.layers.Dense(units=5)

l\_2 = tf.keras.layers.Dense(units=1)

model = tf.keras.Sequential([l\_0, l\_1, l\_2])

model.compile(loss='mean\_squared\_error', optimizer=tf.keras.optimizers.Adam(0.1))

model.fit(X\_train,y\_train, epochs=2000,verbose=False)

print('\n Finished training Model')

|  |  |
| --- | --- |
|  | print(model.predict([80])) |
|  | y\_pred=model.predict(X\_test) |
|  | print(r2\_score(y\_test,y\_pred)) |

As we can see **r2\_score** increased, thus goodness of fit or prediction of our model also increased by adding some more layers.