ASSIGNMENT 2 DEEP LEARNING

CS-878

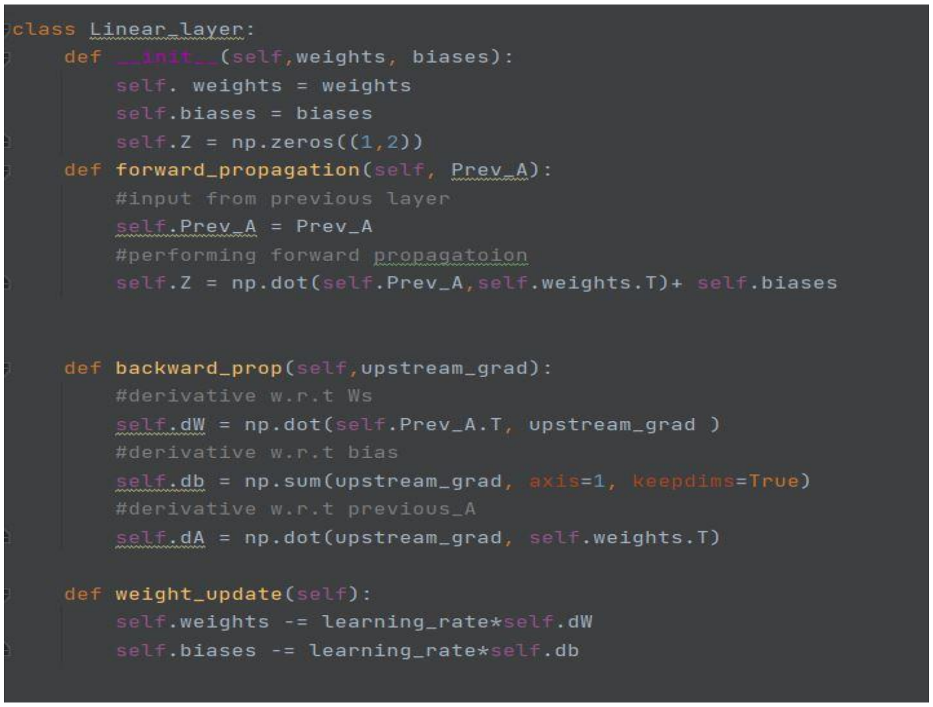
Saeed Ahmad Reg# 362002 MSAI 2k21

Task #1

We were provided a network and were tasked to optimize the weights for required outputs.

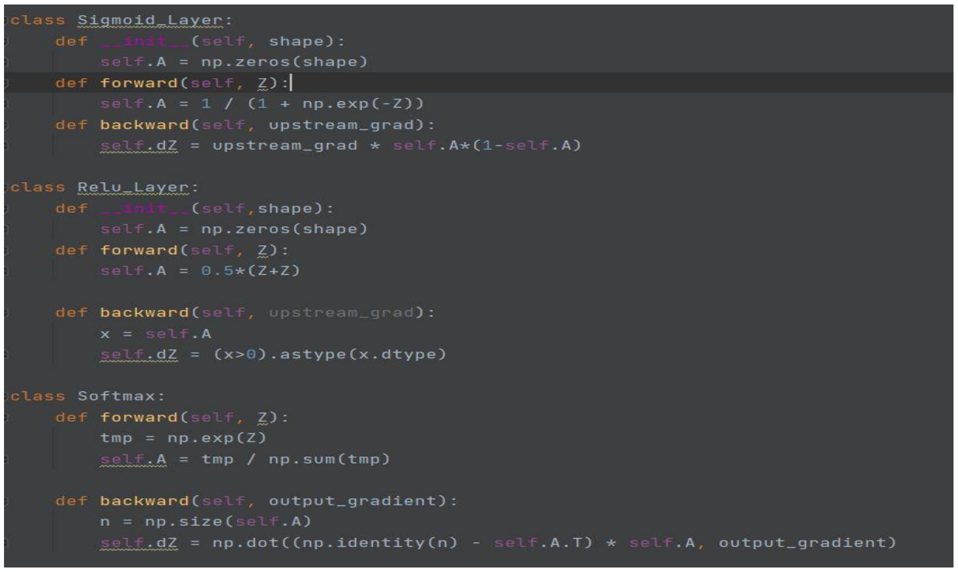
The following are the Network Definitions:

Linear Layer Definition:



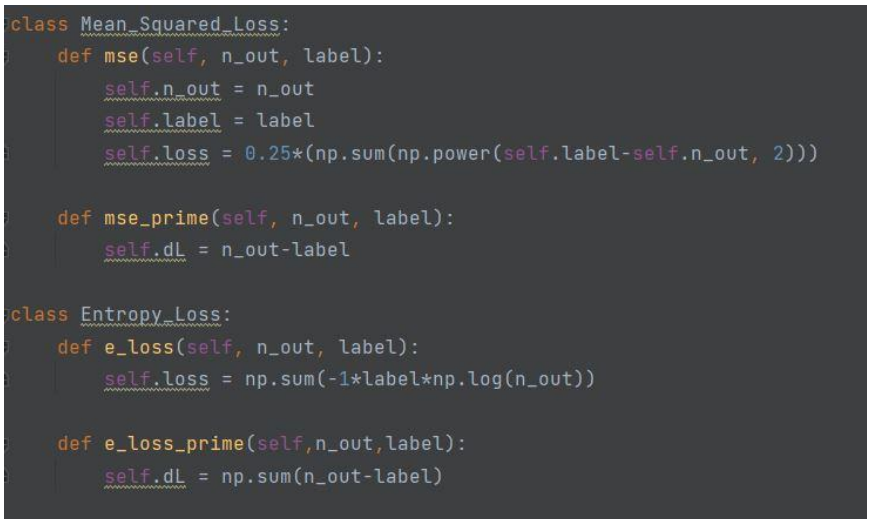
```
class Linear_layer:
    def __init__(self,weights, biases):
        self. weights = weights
        self.biases = biases
        self.Z = np.zeros((1,2))
    def forward_propagation(self, Prev_A):
        #input from previous layer
        self.Prev_A = Prev_A
        #performing forward propagatoion
        self.Z = np.dot(self.Prev_A,self.weights.T)+ self.biases


    def backward_prop(self,upstream_grad):
        #derivative w.r.t Ws
        self.dW = np.dot(self.Prev_A.T, upstream_grad )
        #derivative w.r.t bias
        self.db = np.sum(upstream_grad, axis=1, keepdims=True)
        #derivative w.r.t previous_A
        self.dA = np.dot(upstream_grad, self.weights.T)

    def weight_update(self):
        self.weights -= learning_rate*self.dW
        self.biases -= learning_rate*self.db
```

Sigmoid, Softmax and Relu Definitions:



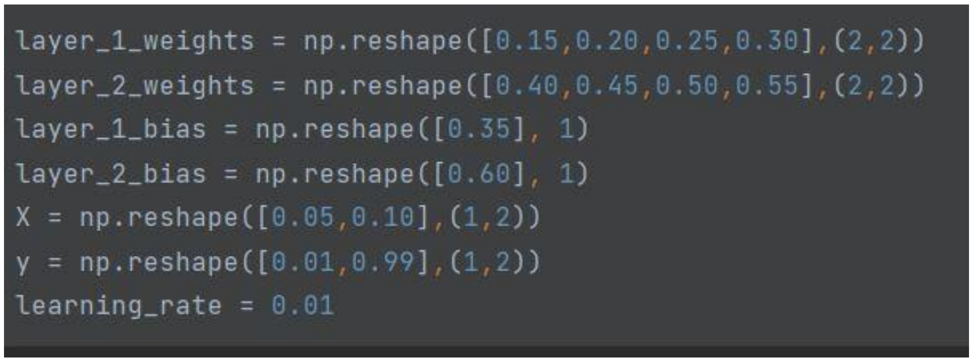
```
class Sigmoid_Layer:
    def __init__(self, shape):
        self.A = np.zeros(shape)
    def forward(self, Z):
        self.A = 1 / (1 + np.exp(-Z))
    def backward(self, upstream_grad):
        self.dZ = upstream_grad * self.A*(1-self.A)

class Relu_Layer:
    def __init__(self,shape):
        self.A = np.zeros(shape)
    def forward(self, Z):
        self.A = 0.5*(Z+Z)

    def backward(self, upstream_grad):
        x = self.A
        self.dZ = (x>0).astype(x.dtype)

class Softmax:
    def forward(self, Z):
        tmp = np.exp(Z)
        self.A = tmp / np.sum(tmp)

    def backward(self, output_gradient):
        n = np.size(self.A)
        self.dZ = np.dot((np.identity(n) - self.A.T) * self.A, output_gradient)
```

Error Function Definitions:



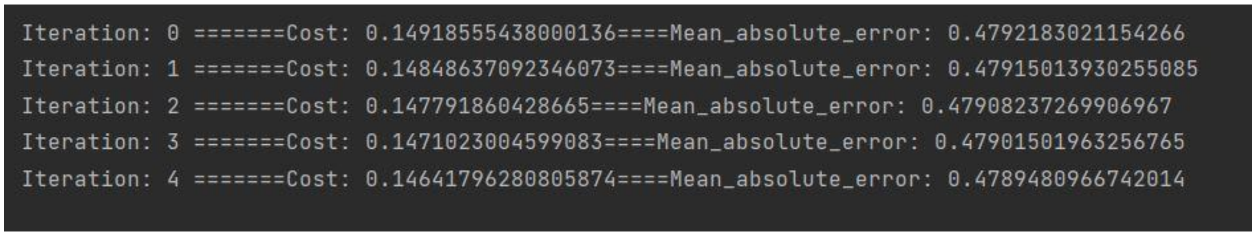
```python
class Mean_Squared_Loss:
    def mse(self, n_out, label):
        self.n_out = n_out
        self.label = label
        self.loss = 0.25*(np.sum(np.power(self.label-self.n_out, 2)))

    def mse_prime(self, n_out, label):
        self.dL = n_out-label

class Entropy_Loss:
    def e_loss(self, n_out, label):
        self.loss = np.sum(-1*label*np.log(n_out))

    def e_loss_prime(self,n_out,label):
        self.dL = np.sum(n_out-label)
```

Data Set-up:



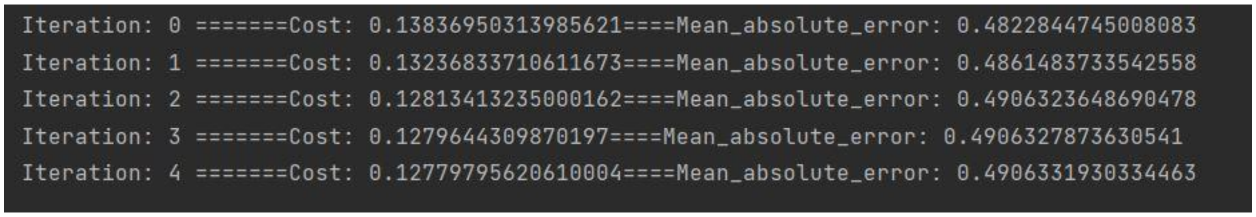
```python
layer_1_weights = np.reshape([0.15,0.20,0.25,0.30],(2,2))
layer_2_weights = np.reshape([0.40,0.45,0.50,0.55],(2,2))
layer_1_bias = np.reshape([0.35], 1)
layer_2_bias = np.reshape([0.60], 1)
X = np.reshape([0.05,0.10],(1,2))
y = np.reshape([0.01,0.99],(1,2))
learning_rate = 0.01
```

Results With Sigmoid Hidden Layer:



```
Iteration: 0 =======Cost: 0.14918555438000136====Mean_absolute_error: 0.4792183021154266
Iteration: 1 =======Cost: 0.14848637092346073====Mean_absolute_error: 0.47915013930255085
Iteration: 2 =======Cost: 0.147791860428665====Mean_absolute_error: 0.47908237269906967
Iteration: 3 =======Cost: 0.147102300459908З====Mean_absolute_error: 0.47901501963256765
Iteration: 4 =======Cost: 0.14641796280805874====Mean_absolute_error: 0.4789480966742014
```

Results With Relu Hidden Layer:



```
Iteration: 0 =======Cost: 0.13836950313985621====Mean_absolute_error: 0.4822844745008083
Iteration: 1 =======Cost: 0.13236833710611673====Mean_absolute_error: 0.4861483733542558
Iteration: 2 =======Cost: 0.12813413235000162====Mean_absolute_error: 0.4906323648690478
Iteration: 3 =======Cost: 0.1279644309870197====Mean_absolute_error: 0.490632787363054l
Iteration: 4 =======Cost: 0.12779795620610004====Mean_absolute_error: 0.4906331930334463
```

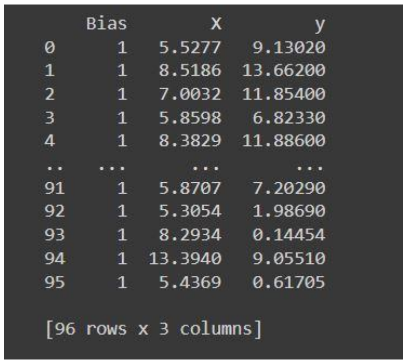
It is evident from the results that the cost and MAE has started to drop slowly with learning rate 0.1.

Discussion:

After doing this task I have come to conclusion that using mean squared divergence for classification tasks isn't suitable because it results in no-convex function, usually the convergence is either really slow or not even possible. This task can best be done using Softmax in output layer and Entropy Loss as divergence.

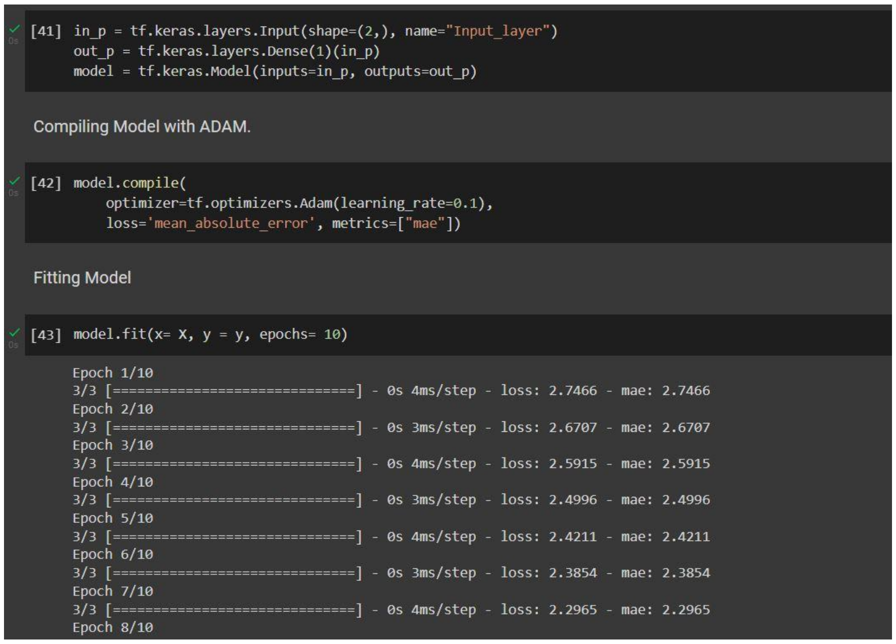
Task # 2:

Data Setup:



```
    Bias        X         y
0      1   5.5277   9.13020
1      1   8.5186  13.66200
2      1   7.0032  11.85400
3      1   5.8598   6.82330
4      1   8.3829  11.88600
..   ...      ...       ...
91     1   5.8707   7.20290
92     1   5.3054   1.98690
93     1   8.2934   0.14454
94     1  13.3940   9.05510
95     1   5.4369   0.61705

[96 rows x 3 columns]
```

Model using Tensorflow's Built-in functions (ADAM, 0.1LR):



```
[41] in_p = tf.keras.layers.Input(shape=(2,), name="Input_layer")
     out_p = tf.keras.layers.Dense(1)(in_p)
     model = tf.keras.Model(inputs=in_p, outputs=out_p)
```

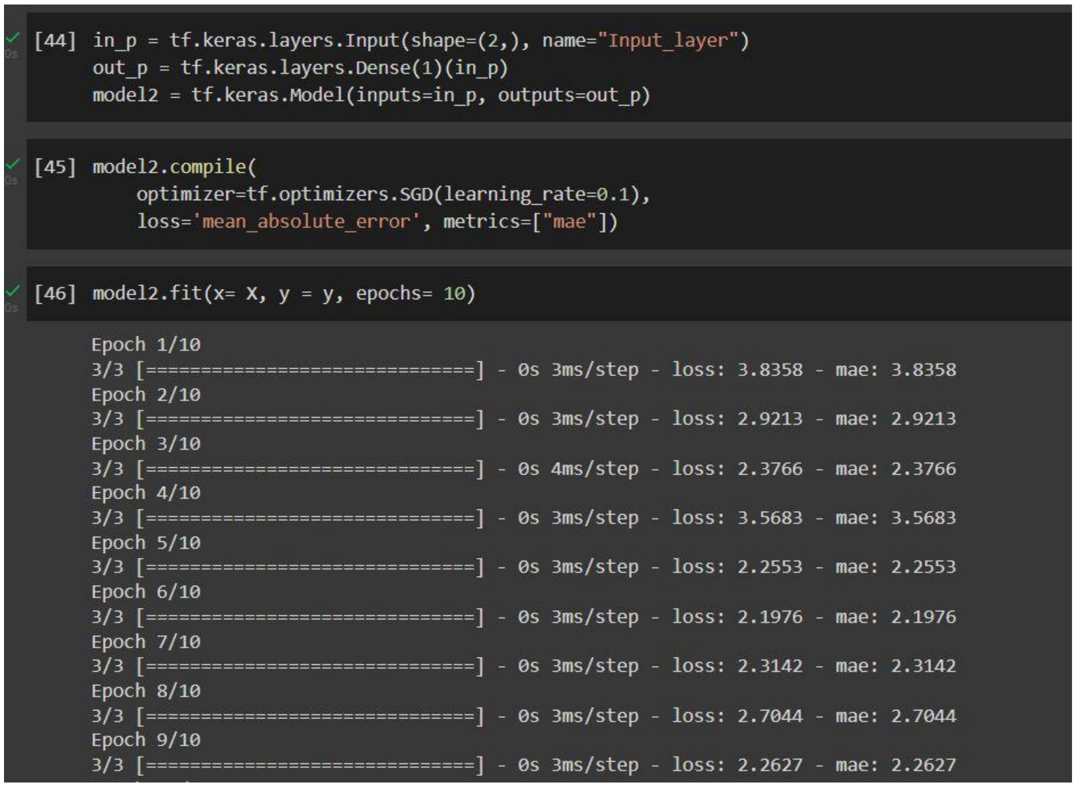
Compiling Model with ADAM.

```
[42] model.compile(
         optimizer=tf.optimizers.Adam(learning_rate=0.1),
         loss='mean_absolute_error', metrics=["mae"])
```

Fitting Model

```
[43] model.fit(x= X, y = y, epochs= 10)

     Epoch 1/10
     3/3 [==============================] - 0s 4ms/step - loss: 2.7466 - mae: 2.7466
     Epoch 2/10
     3/3 [==============================] - 0s 3ms/step - loss: 2.6707 - mae: 2.6707
     Epoch 3/10
     3/3 [==============================] - 0s 4ms/step - loss: 2.5915 - mae: 2.5915
     Epoch 4/10
     3/3 [==============================] - 0s 3ms/step - loss: 2.4996 - mae: 2.4996
     Epoch 5/10
     3/3 [==============================] - 0s 4ms/step - loss: 2.4211 - mae: 2.4211
     Epoch 6/10
     3/3 [==============================] - 0s 3ms/step - loss: 2.3854 - mae: 2.3854
     Epoch 7/10
     3/3 [==============================] - 0s 4ms/step - loss: 2.2965 - mae: 2.2965
     Epoch 8/10
```

Model using Tensorflow's Built-in functions (SGD, 0.1LR):



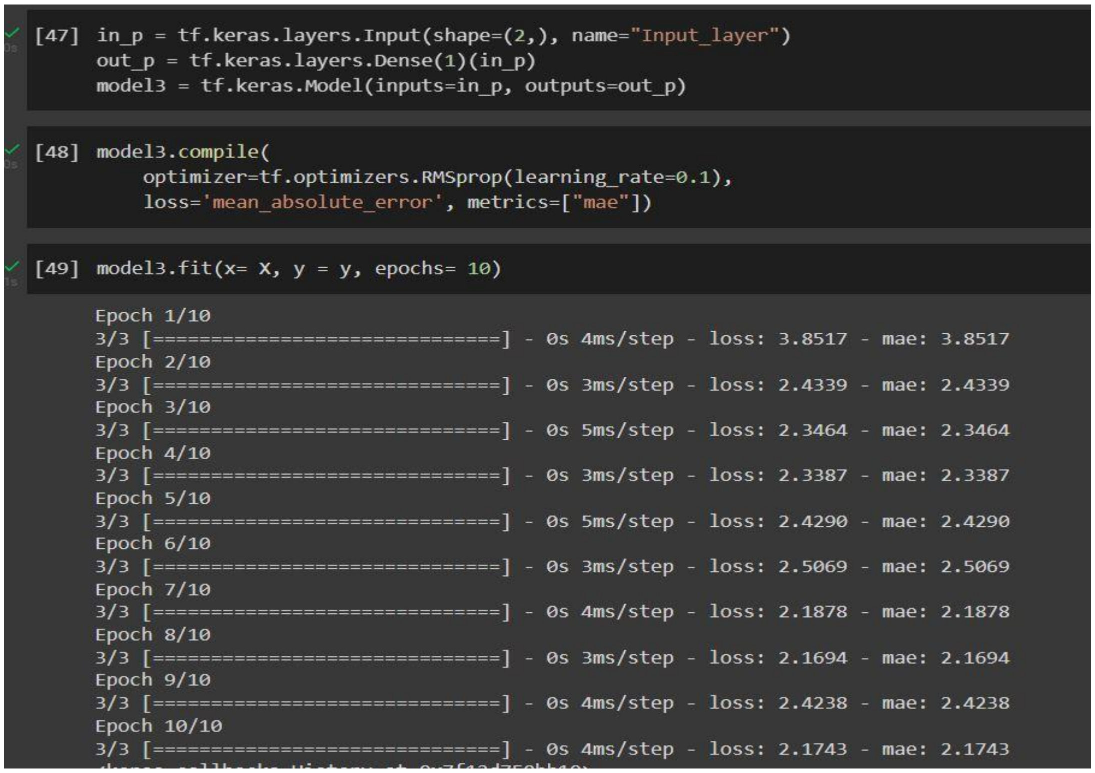
```
[44] in_p = tf.keras.layers.Input(shape=(2,), name="Input_layer")
     out_p = tf.keras.layers.Dense(1)(in_p)
     model2 = tf.keras.Model(inputs=in_p, outputs=out_p)
```

```
[45] model2.compile(
         optimizer=tf.optimizers.SGD(learning_rate=0.1),
         loss='mean_absolute_error', metrics=["mae"])
```

```
[46] model2.fit(x= X, y = y, epochs= 10)
```

```
Epoch 1/10
3/3 [==============================] - 0s 3ms/step - loss: 3.8358 - mae: 3.8358
Epoch 2/10
3/3 [==============================] - 0s 3ms/step - loss: 2.9213 - mae: 2.9213
Epoch 3/10
3/3 [==============================] - 0s 4ms/step - loss: 2.3766 - mae: 2.3766
Epoch 4/10
3/3 [==============================] - 0s 3ms/step - loss: 3.5683 - mae: 3.5683
Epoch 5/10
3/3 [==============================] - 0s 3ms/step - loss: 2.2553 - mae: 2.2553
Epoch 6/10
3/3 [==============================] - 0s 3ms/step - loss: 2.1976 - mae: 2.1976
Epoch 7/10
3/3 [==============================] - 0s 3ms/step - loss: 2.3142 - mae: 2.3142
Epoch 8/10
3/3 [==============================] - 0s 3ms/step - loss: 2.7044 - mae: 2.7044
Epoch 9/10
3/3 [==============================] - 0s 3ms/step - loss: 2.2627 - mae: 2.2627
```

Model using Tensorflow's Built-in functions (RMS Prop, 0.1LR):



```
[47] in_p = tf.keras.layers.Input(shape=(2,), name="Input_layer")
     out_p = tf.keras.layers.Dense(1)(in_p)
     model3 = tf.keras.Model(inputs=in_p, outputs=out_p)
```

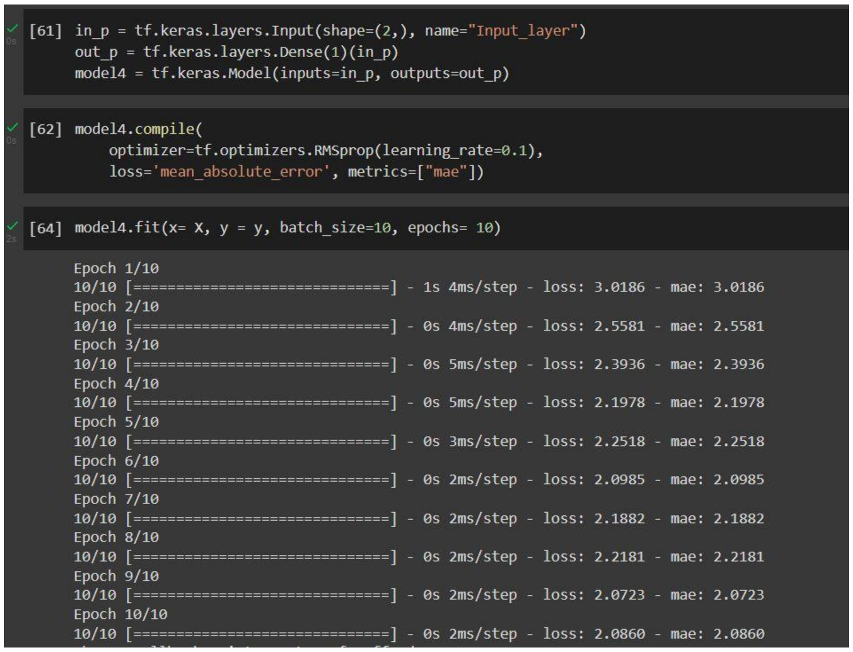
```
[48] model3.compile(
         optimizer=tf.optimizers.RMSprop(learning_rate=0.1),
         loss='mean_absolute_error', metrics=["mae"])
```

```
[49] model3.fit(x= X, y = y, epochs= 10)
```

```
Epoch 1/10
3/3 [==============================] - 0s 4ms/step - loss: 3.8517 - mae: 3.8517
Epoch 2/10
3/3 [==============================] - 0s 3ms/step - loss: 2.4339 - mae: 2.4339
Epoch 3/10
3/3 [==============================] - 0s 5ms/step - loss: 2.3464 - mae: 2.3464
Epoch 4/10
3/3 [==============================] - 0s 3ms/step - loss: 2.3387 - mae: 2.3387
Epoch 5/10
3/3 [==============================] - 0s 5ms/step - loss: 2.4290 - mae: 2.4290
Epoch 6/10
3/3 [==============================] - 0s 3ms/step - loss: 2.5069 - mae: 2.5069
Epoch 7/10
3/3 [==============================] - 0s 4ms/step - loss: 2.1878 - mae: 2.1878
Epoch 8/10
3/3 [==============================] - 0s 3ms/step - loss: 2.1694 - mae: 2.1694
Epoch 9/10
3/3 [==============================] - 0s 4ms/step - loss: 2.4238 - mae: 2.4238
Epoch 10/10
3/3 [==============================] - 0s 4ms/step - loss: 2.1743 - mae: 2.1743
```

It can be Noted that the convergence in ADAM and RMS Prop is smooth while in case of SGD it is a little wayward which is totally intuitive.

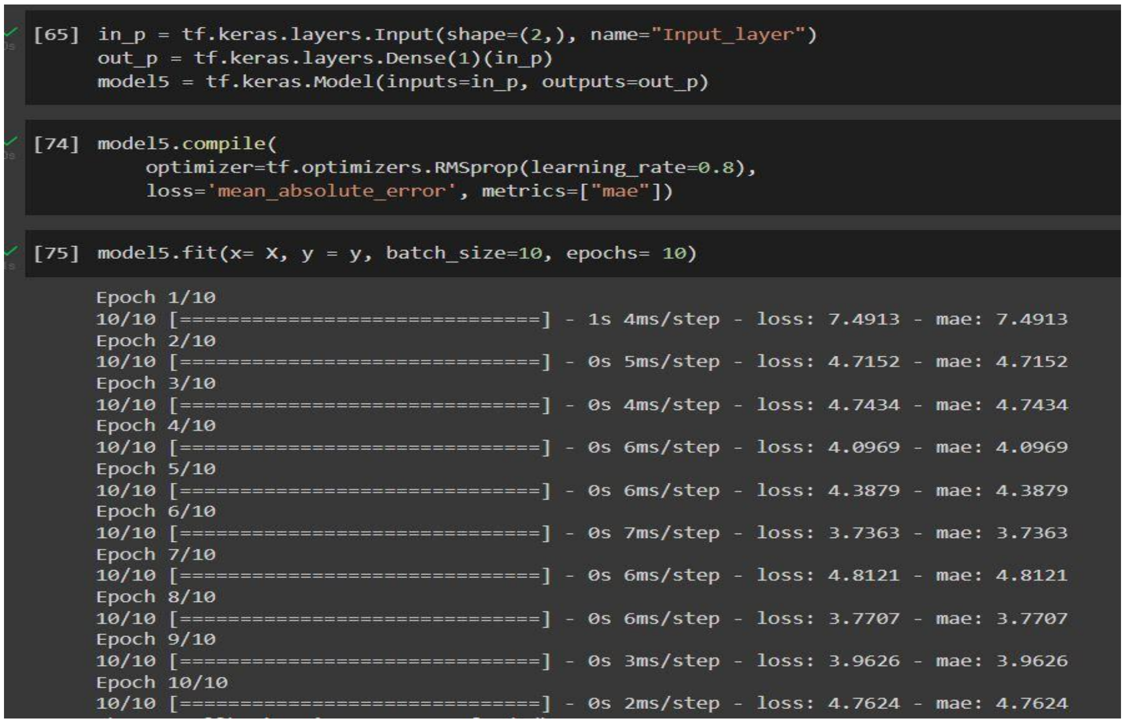
Model using Tensorflow's Built-in functions (RMS Prop, 0.1LR, batch_size = 10):



```
[61] in_p = tf.keras.layers.Input(shape=(2,), name="Input_layer")
     out_p = tf.keras.layers.Dense(1)(in_p)
     model4 = tf.keras.Model(inputs=in_p, outputs=out_p)

[62] model4.compile(
         optimizer=tf.optimizers.RMSprop(learning_rate=0.1),
         loss='mean_absolute_error', metrics=["mae"])

[64] model4.fit(x= X, y = y, batch_size=10, epochs= 10)

     Epoch 1/10
     10/10 [==============================] - 1s 4ms/step - loss: 3.0186 - mae: 3.0186
     Epoch 2/10
     10/10 [==============================] - 0s 4ms/step - loss: 2.5581 - mae: 2.5581
     Epoch 3/10
     10/10 [==============================] - 0s 5ms/step - loss: 2.3936 - mae: 2.3936
     Epoch 4/10
     10/10 [==============================] - 0s 5ms/step - loss: 2.1978 - mae: 2.1978
     Epoch 5/10
     10/10 [==============================] - 0s 3ms/step - loss: 2.2518 - mae: 2.2518
     Epoch 6/10
     10/10 [==============================] - 0s 2ms/step - loss: 2.0985 - mae: 2.0985
     Epoch 7/10
     10/10 [==============================] - 0s 2ms/step - loss: 2.1882 - mae: 2.1882
     Epoch 8/10
     10/10 [==============================] - 0s 2ms/step - loss: 2.2181 - mae: 2.2181
     Epoch 9/10
     10/10 [==============================] - 0s 2ms/step - loss: 2.0723 - mae: 2.0723
     Epoch 10/10
     10/10 [==============================] - 0s 2ms/step - loss: 2.0860 - mae: 2.0860
```

Using mini batches is a really handy tool which can provide a perfect balance between convergence and training times.

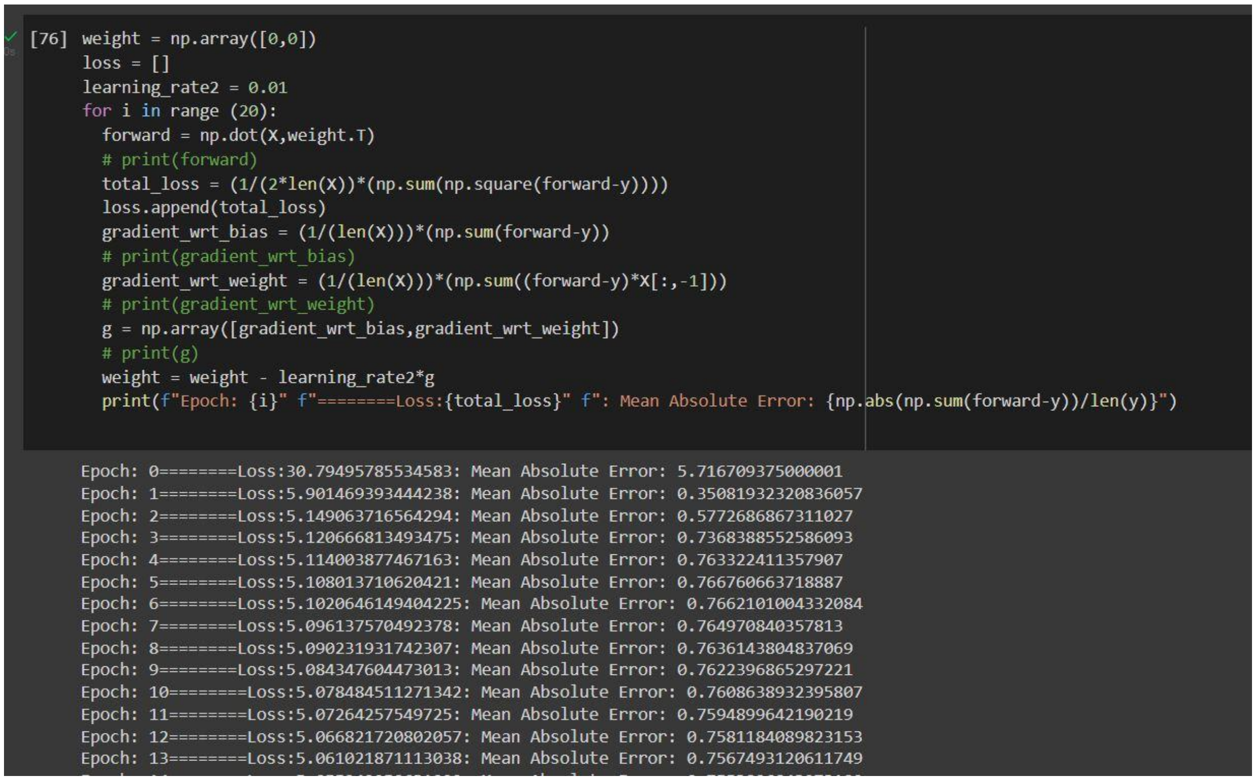
Model using Tensorflow's Built-in functions (RMS Prop, 0.2LR, batch_size = 10):



```
[65] in_p = tf.keras.layers.Input(shape=(2,), name="Input_layer")
     out_p = tf.keras.layers.Dense(1)(in_p)
     model5 = tf.keras.Model(inputs=in_p, outputs=out_p)

[74] model5.compile(
         optimizer=tf.optimizers.RMSprop(learning_rate=0.8),
         loss='mean_absolute_error', metrics=["mae"])

[75] model5.fit(x= X, y = y, batch_size=10, epochs= 10)

     Epoch 1/10
     10/10 [==============================] - 1s 4ms/step - loss: 7.4913 - mae: 7.4913
     Epoch 2/10
     10/10 [==============================] - 0s 5ms/step - loss: 4.7152 - mae: 4.7152
     Epoch 3/10
     10/10 [==============================] - 0s 4ms/step - loss: 4.7434 - mae: 4.7434
     Epoch 4/10
     10/10 [==============================] - 0s 6ms/step - loss: 4.0969 - mae: 4.0969
     Epoch 5/10
     10/10 [==============================] - 0s 6ms/step - loss: 4.3879 - mae: 4.3879
     Epoch 6/10
     10/10 [==============================] - 0s 7ms/step - loss: 3.7363 - mae: 3.7363
     Epoch 7/10
     10/10 [==============================] - 0s 6ms/step - loss: 4.8121 - mae: 4.8121
     Epoch 8/10
     10/10 [==============================] - 0s 6ms/step - loss: 3.7707 - mae: 3.7707
     Epoch 9/10
     10/10 [==============================] - 0s 3ms/step - loss: 3.9626 - mae: 3.9626
     Epoch 10/10
     10/10 [==============================] - 0s 2ms/step - loss: 4.7624 - mae: 4.7624
```

It is to be noted that with fairly big learning rate the loss has started to bounce around which means that the learning rate is bigger than the optimal and lesser than 2x optimal that is why it is still kind of converging and not diverging.

Batch Implementation in Numpy:



```
[76] weight = np.array([0,0])
     loss = []
     learning_rate2 = 0.01
     for i in range (20):
       forward = np.dot(X,weight.T)
       # print(forward)
       total_loss = (1/(2*len(X))*(np.sum(np.square(forward-y))))
       loss.append(total_loss)
       gradient_wrt_bias = (1/(len(X)))*(np.sum(forward-y))
       # print(gradient_wrt_bias)
       gradient_wrt_weight = (1/(len(X)))*(np.sum((forward-y)*X[:,-1]))
       # print(gradient_wrt_weight)
       g = np.array([gradient_wrt_bias,gradient_wrt_weight])
       # print(g)
       weight = weight - learning_rate2*g
       print(f"Epoch: {i}" f"========Loss:{total_loss}" f": Mean Absolute Error: {np.abs(np.sum(forward-y))/len(y)}")
```

```
Epoch: 0========Loss:30.79495785534583: Mean Absolute Error: 5.716709375000001
Epoch: 1========Loss:5.901469393444238: Mean Absolute Error: 0.35081932320836057
Epoch: 2========Loss:5.149063716564294: Mean Absolute Error: 0.5772686867311027
Epoch: 3========Loss:5.120666813493475: Mean Absolute Error: 0.7368388552586093
Epoch: 4========Loss:5.114003877467163: Mean Absolute Error: 0.763322411357907
Epoch: 5========Loss:5.108013710620421: Mean Absolute Error: 0.766760663718887
Epoch: 6========Loss:5.1020646149404225: Mean Absolute Error: 0.7662101004332084
Epoch: 7========Loss:5.096137570492378: Mean Absolute Error: 0.764970840357813
Epoch: 8========Loss:5.090231931742307: Mean Absolute Error: 0.7636143804837069
Epoch: 9========Loss:5.084347604473013: Mean Absolute Error: 0.7622396865297221
Epoch: 10========Loss:5.078484511271342: Mean Absolute Error: 0.7608638932395807
Epoch: 11========Loss:5.07264257549725: Mean Absolute Error: 0.7594899642190219
Epoch: 12========Loss:5.066821720802057: Mean Absolute Error: 0.7581184089823153
Epoch: 13========Loss:5.061021871113038: Mean Absolute Error: 0.7567493120611749
```