Lab report 11 & 12

Fall 2021

CSE422L Data Analytics Lab

Submitted by: **Ayaz Mehmood**

Registration No.:**18PWCSE1652**

Section: **A**

"On my honor, as student of University of Engineering and Technology, I have neither given nor received unauthorized assistance on this academic work."

Student Signature: _____

Submitted to:

Engr. Mian Ibad Ali Shah

Last date of Submission:

9 March 2022

Department of Computer Systems Engineering

University of Engineering and Technology, Peshawar

TASK 1

Change the target as well as following parameters and analyze the code and output (as a proper lab report):

Learning rate Epochs Optimization function Size of data

Intro to NN file

Changing the Target:

f(x,z) = 2x - 3z + 7

I changed the constant value from 5 to 7 and random generated noise from -2 to 1

Learning rate set:

0.002

Epoch:

3 epoch (nested loop run 3 times for the no of epoch.

Size of data:

2000 data

CODE:

Importing the libraries and generate random input dataset

Import the relevant libraries

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

Generate random input data to train on

In [20]:
```
observations = 2000
xs = np.random.uniform(low=-10, high=10, size=(observations,1))
zs = np.random.uniform(-10, 10, (observations,1))

inputs = np.column_stack((xs,zs))

print (inputs.shape)
```

```
(2000, 2)
```

Generate target:

Generate the targets we will aim at

In [21]:
```
# We want to make a function and see if the algorithm has learned it.
# We add a small random noise to the function i.e. f(x,z) = 2x - 3z + 7 + <small noise> as in real life datasets
noise = np.random.uniform(-2, 1, (observations,1))

targets = 2*xs - 3*zs + 7 + noise

print (targets.shape)
print(noise)
```

```
(2000, 1)
[[-0.63038147]
 [-0.50069333]
 [-1.38068612]
 ...
 [ 0.13201237]
 [-1.59324579]
 [ 0.27336618]]
```

Plotting the training data:

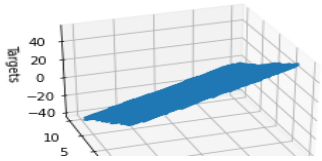
In [42]:
```
targets = targets.reshape(observations,)
xs = xs.reshape(observations,)
zs = zs.reshape(observations,)
fig = plt.figure()

ax = fig.add_subplot(111, projection='3d')

ax.plot(xs, zs, targets)

ax.set_xlabel('xs')
ax.set_ylabel('zs')
ax.set_zlabel('Targets')
ax.view_init(azim=250)
plt.show()
targets = targets.reshape(observations,1)
xs = xs.reshape(observations,1)
zs = zs.reshape(observations,1)

```



Initializing weight and biases:

Initialize variables

In [8]:
```
init_range = 0.1

# Weights are of size k x m, where k is the number of input variables and m is the number of output variables
# In our case, the weights matrix is 2x1 since there are 2 inputs (x and z) and one output (y)
weights = np.random.uniform(low=-init_range, high=init_range, size=(2, 1))

# Biases are of size 1 since there is only 1 output. The bias is a scalar.
biases = np.random.uniform(low=-init_range, high=init_range, size=1)

print (weights)
print (biases)
```

```
[[-0.04976581]
 [-0.01377844]]
[0.0093437]
```

Set a learning rate:

Set a learning rate

In [9]:
```
# Play around with learning rate.
learning_rate = 0.002
```

Training the model

Epoch set is 3:

Train the model

In [45]:
```
for j in range (3):
    for i in range (1000):

        outputs = np.dot(inputs,weights) + biases
        deltas = outputs - targets

        loss = np.sum(deltas ** 2) / 2 / observations



        deltas_scaled = deltas / observations

        # Finally, apply the gradient descent update rules
        # The weights are 2x1, learning rate is 1x1 (scalar), inputs are 1000x2, and deltas_scaled are 1000x1
        # Transpose the inputs so that we get an allowed operation.
        weights = weights - learning_rate * np.dot(inputs.T,deltas_scaled)
        biases = biases - learning_rate * np.sum(deltas_scaled)
    print (loss)
```

```
0.7782300280802876
0.3862873984619478
0.3790533873048634
```

Loss after every epoch is printed below (3 time's nested loop run for the epoch)

Printing weights and bias:

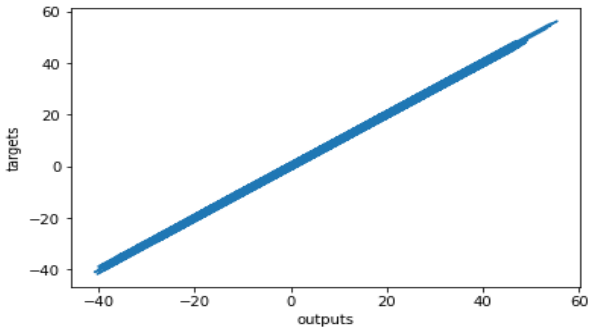
Print weights and biases and see if we have worked correctly.

In [18]:
```
1 print (weights, biases)
```

```
[[ 1.99961543]
 [-2.99776454]] [6.4632317]
```

Plot outputs vs target:

In [19]:
```
1 plt.plot(outputs,targets)
2 plt.xlabel('outputs')
3 plt.ylabel('targets')
4 plt.show()
```



Analysis:

I changed the dataset size to 2000, increase the epoch to 3 and check the cost (total error) which is 0.77 for first epoch and in third epoch it becomes 0.37. I changed the learning rate to 0.002 so the gradient could be low as possible. As we know learning rate is a tuning parameter that determine the step size at each iteration while moving towards minimum cost function. I changed the target too by adding more noise to the output label

Task 2:

Change the target as well as following parameters and analyze the code and output (as a proper lab report):

Learning rate Epochs Optimization function Size of data

Intro to tf file:

Changing the Target:

f(x,z) = 3*xs - 2*zs + 5 + noise

I changed the random generated noise from -3 to 1.

Learning rate set:

0.09

Epoch:

50 epoch (nested loop run 3 times for the no of epoch.

In code I was 100 epoch

Size of data:

500 data

Optimizer:

Adam optimizer

CODE:

Importing the libraries:

Import the relevant libraries

In [16]:
```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow.compat.v1 as tf

tf.disable_v2_behavior()
```

Data generation:

Data generation

We generate data using the exact same logic and code as in the previos lab. The only difference now is that we save it to an npz file. Npz is numpy's file type which allows you to save numpy arrays into a single .npz file.

Nothing to worry about - this is literally saving your NumPy arrays into a file that you can later access!

In [17]:
```
observations = 500

# We will work with two variables as inputs. You can think about them as x1 and x2 in our previous lab.
xs = np.random.uniform(low=-10, high=10, size=(observations,1))
zs = np.random.uniform(-10, 10, (observations,1))

generated_inputs = np.column_stack((xs,zs))

# We add a random small noise to the function i.e. f(x,z) = 2x - 3z + 5 + <small noise>
noise = np.random.uniform(-3, 1, (observations,1))

# Produce the targets according to our f(x,z) = 3*xs - 2*zs + 5 + noise.
# We are basically saying: the weights should be 2 and -3, while the bias is 5.
generated_targets = 3*xs - 2*zs + 5 + noise

# save into an npz file called "TF_intro"
np.savez('TF_intro', inputs=generated_inputs, targets=generated_targets)
```

Shape of the dataset we have created above:

Solving with TensorFlow

In [26]:
```
# The shape of the data we've prepared above.
input_size = 2
output_size = 1
```

Outlining the model and choosing objective function and optimizer method:

In [19]:
```
# Here we define a basic TensorFlow object - the placeholder.
# In the TensorFlow, we feed the data to the model THROUGH the placeholders.

inputs = tf.placeholder(tf.float32, [None, input_size])
targets = tf.placeholder(tf.float32, [None, output_size])

# As before, we define our weights and biases.
# They are the other basic TensorFlow object - a variable.
# We feed data into placeholders and they have a different value for each iteration

weights = tf.Variable(tf.random_uniform([input_size, output_size], minval=-0.1, maxval=0.1))
biases = tf.Variable(tf.random_uniform([output_size], minval=-0.1, maxval=0.1))

# We get the outputs following our linear combination: y = xw + b
outputs = tf.matmul(inputs, weights) + biases
```

Choosing the objective function and the optimization method

In [24]:
```
# Again, we use a loss function. mean_squared_error is the scaled L2-norm (per observation)
mean_loss = tf.losses.mean_squared_error(labels=targets, predictions=outputs) / 2.
# Note that there also exists a function tf.nn.l2_loss.

# Instead of implementing Gradient Descent on our own, in TensorFlow we can simply state
# "Minimize the mean loss by using Adam with a given learning rate"
# Simple!
optimize = tf.train.AdamOptimizer(learning_rate=0.09).minimize(mean_loss)
```

Preparing for execution and initiating variable:

Prepare for execution

In [29]:
```
# So far we've defined the placeholders, variables, the loss function and the optimization method.
# The actual training happens inside sessions.
sess = tf.InteractiveSession()
```

Initializing variables

In [30]:
```
# Before we start training, we need to initialize our variables: the weights and biases.
initializer = tf.global_variables_initializer()

sess.run(initializer)
```

Loading training data and learning:

Loading training data

In [35]:
```
#Load the training data we created above.
training_data = np.load('TF_intro.npz')
```

Learning

In [36]:
```
# As in the previous lab, we train for a set number (100) of iterations over the dataset
for i in range(50):
    # sess.run is the session's function to actually do something, anything.
    # Above, we used it to initialize the variables.
    # Here, we use it to feed the training data to the computational graph, defined by the feed_dict parameter
    # So the line of code means: "Run the optimize and mean_loss operations by filling the placeholder
    # objects with data from the feed_dict parameter".
    _, curr_loss = sess.run([optimize, mean_loss],
        feed_dict={inputs: training_data['inputs'], targets: training_data['targets']})

    print(curr_loss)
```

```
194.50928
181.15617
168.31653
155.99854
```

Plotting the data:

Plotting the data

In [37]:
```
out = sess.run([outputs],
               feed_dict={inputs: training_data['inputs']})
plt.plot(np.squeeze(out), np.squeeze(training_data['targets']))
plt.xlabel('outputs')
plt.ylabel('targets')
plt.show()
```