```
#########################################################################
# This code solves a simple linear regression problem with three
# different approaches:
#   (1) Diffrentiable Programming approach
#   (2) Theoretical approach with matrix multiplications.
#   (3) Probablistic programming approach with PyMC3
# Both the first two approaches are done with TensorFlow.
#
# To run this example:
# (1) Go to  https://colab.research.google.com/notebooks/welcome.ipynb
# (2) File > New Python3 notebook
# (3) Paste the code in a cell (not including this comment block!)
# (4) Run the cell (Reduce training_epochs and pm.sample sample size to
#     shorten the run time)
# You might be asked to log in to your Google account during the process.
#########################################################################
%pylab inline
!pip install arviz
from tensorboardcolab import TensorBoardColab, TensorBoardColabCallback
import pymc3 as pm
import numpy as np
import tensorflow as tf
import tensorflow_probability as tfp
import matplotlib.pyplot as plt
import time

np.random.seed(101)
tf.set_random_seed(101)
tbc = TensorBoardColab()

# Generates 200 normally distributed data points (px,py)
n = 200
true_beta0 = 1
true_beta1 = 2

px = np.linspace(0, 1, n)
mu = true_beta0 + true_beta1 * px
```

```
py = mu + np.random.normal(scale=.5, size=n)

learning_rate = 0.01
training_epochs = 3 # Reduce epoch to reduce run time

#########################################################################
# Computational Solution based on Differentiable Programming

# Inputs
X = tf.placeholder("float")
Y = tf.placeholder("float")

# What are the control Parameters ?
beta1 = tf.Variable(np.random.randn(), dtype = tf.float32, name = "beta1")
beta0 = tf.Variable(np.random.randn(), dtype = tf.float32, name = "beta0")

# How to compute outputs from inputs?
y_pred = tf.add(tf.multiply(X, beta1), beta0)

# How to compute loss function from outputs and expected
#  outputs? Mean Squared Error (MSE)
loss = tf.reduce_sum(tf.pow((y_pred - Y), 2)) / (2*n)

# Optimizer
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

# Global Variables Initializer
init = tf.global_variables_initializer()

# Tensorflow Session
with tf.Session() as sess:
    # `sess.graph` provides access to the graph used in a session
    writer = tf.summary.FileWriter("./Graph", sess.graph)

    # Initializing the Variables
    sess.run(init)
```

```
    start_time_dp = time.time()
    # Iterating through all the epochs
    for epoch in range(training_epochs):
        # Feeding each data point into the optimizer using Feed Dictionary
        for (_x, _y) in zip(px, py):
            sess.run(optimizer, feed_dict = {X : _x, Y : _y})

    writer.close()
    tbc.close()

    # Retrieve values for use outside the Session
    loss_val = sess.run(loss, feed_dict ={X: px, Y: py})
    weight   = sess.run(beta1)
    bias     = sess.run(beta0)

    # Calculate the predictions
    predictions_dp = weight * px + bias

    duration_dp    = time.time() - start_time_dp

    print("DifferentiableProg: ","Loss(MSE) =", loss_val, "Slope =", weight, "Intercept =", bias,
"RunTime =", duration_dp, '\n')
#####################################################################


#####################################################################
# Theoretical Solution based on matrix multiplication

    start_time_theory = time.time()
    ones = np.ones(n)
    X_matrix = np.matrix(np.column_stack((px, ones)))
    Y_matrix = np.transpose(np.matrix(py))

    X_tensor = tf.constant(X_matrix)
    Y_tensor = tf.constant(Y_matrix)
    XtX_tensor = tf.matmul(tf.transpose(X_tensor), X_tensor)
    XtY_tensor = tf.matmul(tf.transpose(X_tensor), Y_tensor)
```

```
    matrixFormSol = tf.matmul(tf.matrix_inverse(XtX_tensor), XtY_tensor)
    soln_opt = sess.run(matrixFormSol).tolist()
    weight_opt, bias_opt = soln_opt

    # Calculating the predictions
    predictions_opt = weight_opt * px + bias_opt
    sess.run(beta1.assign(weight_opt[0]))
    sess.run(beta0.assign(bias_opt[0]))
    loss_opt = sess.run(loss, feed_dict ={X: px, Y: py})
    duration_theory = time.time() - start_time_theory
    print("Theoretical: ","Loss(MSE) =", loss_opt, "Slope =", weight_opt[0], "Intercept =",
bias_opt[0], "RunTime =", duration_theory,'\n')
# end of "with tf.Session() as sess:"
#################################################################

#################################################################
# Probablistic Programming Solution based on PyMC3
start_time_pp = time.time()
with pm.Model() as model:
    # Define prior distributions of parameters
    beta0 = pm.Normal('beta0', 0, sigma=20)
    beta1 = pm.Normal('beta1', 0, sigma=20)
    sigma = pm.HalfCauchy('sigma', beta=10, testval=1.)

    # Define likelihood (sampling distribution)
    jointDist = pm.Normal('Y', mu = beta0 + beta1 * px, sigma=sigma, observed = py)

    # Inference!
    start = pm.find_MAP(model=model)
    step  = pm.Metropolis()
    trace = pm.sample(1000, step=step)

estimate = pm.summary(trace)
predictions_pp = estimate.at['beta0','mean'] + estimate.at['beta1','mean'] * px
duration_pp = time.time() - start_time_pp
```

```
print("Probabilistic: ","Slope =", estimate.at['beta1','mean'], "Intercept =",
estimate.at['beta0','mean'], "RunTime =", duration_pp,'\n')
pm.traceplot(trace)
plt.show()


#################################################################
# Plotting the Results from three different approaches
plt.plot(px, py, 'o', label ='Data points')
plt.plot(px, predictions_dp, label ='DifferentiableProg')
plt.plot(px, predictions_opt, label ='Theoretical')
plt.plot(px, predictions_pp, label ='ProbabilisticProg')
plt.title('Fitted Line')
plt.legend()
plt.show()
```