Linear Classifier in TensorFlow

Using Low Level API in Eager Execution mode

Load tensorflow

```
!pip3 install -U tensorflow --quiet
```

```
     |████████████████████████████████| 86.3MB 130kB/s
     |████████████████████████████████| 3.8MB 33.7MB/s
     |████████████████████████████████| 450kB 33.4MB/s
     |████████████████████████████████| 81kB 8.2MB/s
ERROR: tensorboard 2.0.1 has requirement grpcio>=1.24.3, but you'll have grpcio 1.15.0 w
ERROR: google-colab 1.0.0 has requirement google-auth~=1.4.0, but you'll have google-aut
```

```
import tensorflow as tf
```

```
#tf.set_random_seed(42)
```

```
#Enable Eager Execution if using tensflow version < 2.0
#From tensorflow v2.0 onwards, Eager Execution will be enabled by default
```

Collect Data

```
from google.colab import drive
drive.mount('/content/drive/')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=9473189

Enter your authorization code:
··········
Mounted at /content/drive/
```

```
import pandas as pd
```

```
data1=pd.read_csv('/content/drive/My Drive/Hackathon/prices.csv')
```

Check all columns in the dataset

```
data1.columns
```

```
Index(['date', 'symbol', 'open', 'close', 'low', 'high', 'volume'], dtype='object')
```

Drop columns `date` and `symbol`

```
data1.drop(columns=['date','symbol'],inplace=True)
```

```
data1.columns
```

```
Index(['open', 'close', 'low', 'high', 'volume'], dtype='object')
```

```
data1.head()
```

	open	close	low	high	volume
0	123.430000	125.839996	122.309998	126.250000	2163600.0
1	125.239998	119.980003	119.940002	125.540001	2386400.0
2	116.379997	114.949997	114.930000	119.739998	2489500.0
3	115.480003	116.620003	113.500000	117.440002	2006300.0
4	117.010002	114.970001	114.089996	117.330002	1408600.0

```
data1.shape
```

```
(851264, 5)
```

```
data1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 851264 entries, 0 to 851263
Data columns (total 5 columns):
open      851264 non-null float64
close     851264 non-null float64
low       851264 non-null float64
high      851264 non-null float64
volume    851264 non-null float64
dtypes: float64(5)
memory usage: 32.5 MB
```

```
data1.describe().transpose()
```

	count	mean	std	min	25%	50%	7
open	851264.0	7.083699e+01	8.369588e+01	0.85	3.384000e+01	5.277000e+01	7.988000e+
close	851264.0	7.085711e+01	8.368969e+01	0.86	3.385000e+01	5.280000e+01	7.989000e+
low	851264.0	7.011841e+01	8.287729e+01	0.83	3.348000e+01	5.223000e+01	7.911000e+
high	851264.0	7.154348e+01	8.446550e+01	0.88	3.419000e+01	5.331000e+01	8.061000e+
volume	851264.0	5.415113e+06	1.249468e+07	0.00	1.221500e+06	2.476250e+06	5.222500e+

Consider only first 1000 rows in the dataset for building feature set and target set

Target 'Volume' has very high values. Divide 'Volume' by 1000,000

```
data=data1.head(1000)
```

```
import numpy as np
```

```
data['volume'] = [np.divide(x,1000000) for x in data['volume']]
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/user_g
  """Entry point for launching an IPython kernel.
```

```
data['volume'][0]
```

```
2.1636
```

```
data1['volume'][0]
```

```
2.1636
```

Divide the data into train and test sets

```
from sklearn.model_selection import train_test_split
```

```
X= np.array(data.drop(columns='volume'))
y= np.array(data['volume'])
```

```
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2,random_state=2324)
```

Convert Training and Test Data to numpy float32 arrays

```
X_train.shape
```

```
(800, 4)
```

```
X_test.shape
```

```
(200, 4)
```

Normalize the data

You can use Normalizer from sklearn.preprocessing

```
from sklearn.preprocessing import Normalizer
```

```
scale = Normalizer()
```

```
X_train_z = scale.fit_transform(X_train)
```

```
X_test_z= scale.fit_transform(X_test)
```

```
X_train_z.shape
```

```
(800, 4)
```

```
X_train_z = X_train_z.astype('float32')
X_test_z = X_test_z.astype('float32')
y_test = y_test.astype('float32')
y_train = y_train.astype('float32')
```

Building the Model in tensorflow

1.Define Weights and Bias, use tf.zeros to initialize weights and Bias

```
#We are initializing weights and Bias with Zero
w = tf.zeros(shape=(4,1))
b = tf.zeros(shape=(1))
```

```
b = tf.zeros(shape=(1))
```

2.Define a function to calculate prediction

```
def prediction(x, w, b):

    xw_matmul = tf.matmul(x, w)
    y = tf.add(xw_matmul, b)

    return y
```

3.Loss (Cost) Function [Mean square error]

```
def loss(y_actual, y_predicted):

    diff = y_actual - y_predicted
    sqr = tf.square(diff)
    avg = tf.reduce_mean(sqr)

    return avg
```

4.Function to train the Model

1. Record all the mathematical steps to calculate Loss
2. Calculate Gradients of Loss w.r.t weights and bias
3. Update Weights and Bias based on gradients and learning rate to minimize loss

```
def train(x, y_actual, w, b, learning_rate=0.01):

    #Record mathematical operations on 'tape' to calculate loss
    with tf.GradientTape() as t:

        t.watch([w,b])

        current_prediction = prediction(x, w, b)
        current_loss = loss(y_actual, current_prediction)

    #Calculate Gradients for Loss with respect to Weights and Bias
    dw, db = t.gradient(current_loss,[w, b])

    #Update Weights and Bias
    w = w - learning_rate*dw
    b = b - learning_rate*db
```

```
    return w, b
```

Train the model for 100 epochs

1. Observe the training loss at every iteration
2. Observe Train loss at every 5th iteration

```
for i in range(100):

    w, b = train(X_train_z, y_train, w, b)
    print('Current Loss on iteration', i, loss(y_train, prediction(X_train_z, w, b)).numpy())
```

```
Current Loss on iteration 0 195.3701
Current Loss on iteration 1 193.45137
Current Loss on iteration 2 191.68303
Current Loss on iteration 3 190.05319
Current Loss on iteration 4 188.5513
Current Loss on iteration 5 187.16687
Current Loss on iteration 6 185.89122
Current Loss on iteration 7 184.71547
Current Loss on iteration 8 183.632
Current Loss on iteration 9 182.63333
Current Loss on iteration 10 181.7131
Current Loss on iteration 11 180.8649
Current Loss on iteration 12 180.08328
Current Loss on iteration 13 179.36282
Current Loss on iteration 14 178.69897
Current Loss on iteration 15 178.08717
Current Loss on iteration 16 177.52328
Current Loss on iteration 17 177.00352
Current Loss on iteration 18 176.52455
Current Loss on iteration 19 176.0832
Current Loss on iteration 20 175.67647
Current Loss on iteration 21 175.30157
Current Loss on iteration 22 174.95602
Current Loss on iteration 23 174.6377
Current Loss on iteration 24 174.34407
Current Loss on iteration 25 174.07378
Current Loss on iteration 26 173.8245
Current Loss on iteration 27 173.5948
Current Loss on iteration 28 173.38312
Current Loss on iteration 29 173.18793
Current Loss on iteration 30 173.00815
Current Loss on iteration 31 172.84247
Current Loss on iteration 32 172.68974
Current Loss on iteration 33 172.549
Current Loss on iteration 34 172.4193
Current Loss on iteration 35 172.2998
Current Loss on iteration 36 172.1896
Current Loss on iteration 37 172.0881
Current Loss on iteration 38 171.99457
Current Loss on iteration 39 171.9083
Current Loss on iteration 40 171.82872
Current Loss on iteration 41 171.75557
Current Loss on iteration 42 171.6881
Current Loss on iteration 43 171.62581
Current Loss on iteration 44 171.56848
Current Loss on iteration 45 171.51573
Current Loss on iteration 46 171.4669
Current Loss on iteration 47 171.42204
Current Loss on iteration 48 171.3807
Current Loss on iteration 49 171.34262
Current Loss on iteration 50 171.30753
Current Loss on iteration 51 171.27513
Current Loss on iteration 52 171.24525
Current Loss on iteration 53 171.21777
Current Loss on iteration 54 171.1924
Current Loss on iteration 55 171.16913
Current Loss on iteration 56 171.14752
```

```
Current Loss on iteration 57 171.12767
Current Loss on iteration 58 171.1094
Current Loss on iteration 59 171.09262
Current Loss on iteration 60 171.07706
Current Loss on iteration 61 171.06279
Current Loss on iteration 62 171.0496
Current Loss on iteration 63 171.03745
Current Loss on iteration 64 171.02623
Current Loss on iteration 65 171.0158
Current Loss on iteration 66 171.00632
Current Loss on iteration 67 170.9977
Current Loss on iteration 68 170.98955
Current Loss on iteration 69 170.98206
Current Loss on iteration 70 170.97522
Current Loss on iteration 71 170.96886
Current Loss on iteration 72 170.9631
Current Loss on iteration 73 170.95767
Current Loss on iteration 74 170.9527
Current Loss on iteration 75 170.94817
Current Loss on iteration 76 170.944
Current Loss on iteration 77 170.94008
Current Loss on iteration 78 170.93648
Current Loss on iteration 79 170.93318
Current Loss on iteration 80 170.93013
Current Loss on iteration 81 170.9274
Current Loss on iteration 82 170.92474
Current Loss on iteration 83 170.92245
Current Loss on iteration 84 170.92026
Current Loss on iteration 85 170.9183
Current Loss on iteration 86 170.91643
Current Loss on iteration 87 170.91463
Current Loss on iteration 88 170.91313
Current Loss on iteration 89 170.9117
Current Loss on iteration 90 170.91032
Current Loss on iteration 91 170.90906
Current Loss on iteration 92 170.90787
Current Loss on iteration 93 170.90677
Current Loss on iteration 94 170.90587
Current Loss on iteration 95 170.90503
Current Loss on iteration 96 170.90425
Current Loss on iteration 97 170.90344
Current Loss on iteration 98 170.90268
Current Loss on iteration 99 170.902
```

Get the shapes and values of W and b

```
#Check Weights and Bias
print('Weights:\n', w.numpy())
print('Bias:\n',b.numpy())
```

```
Weights:
 [[1.2649974]
 [1.26920191
```

Model Prediction on 1st Examples in Test Dataset

```
X_test_z[0]
```

```
array([0.5022779 , 0.49966326, 0.4919523 , 0.50600046], dtype=float32)
```

```
X_test_z.shape
```

```
(200, 4)
```

```
prediction = prediction(X_test_z[0:1], w,b)
```

```
prediction
```

```
<tf.Tensor: id=19633, shape=(1, 1), dtype=float32, numpy=array([[5.066518]], dtype=float
```

```
y_test[0]
```

```
0.4599
```

Classification using tf.Keras

In this exercise, we will build a Deep Neural Network using tf.Keras. We will use Iris Dataset for this ex

Load the given Iris data using pandas (Iris.csv)

```
data2=pd.read_csv('/content/drive/My Drive/Hackathon/Iris.csv')
```

Target set has different categories. So, Label encode them. And convert into one-pandas.

```
data2.head()
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa

```
data2.head()
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

```
from sklearn import preprocessing

# label_encoder object knows how to understand word labels.
label_encoder = preprocessing.LabelEncoder()

# Encode labels in column 'species'.
data2['Species']= label_encoder.fit_transform(data2['Species'])
```

```
data2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 6 columns):
Id                150 non-null int64
SepalLengthCm     150 non-null float64
SepalWidthCm      150 non-null float64
PetalLengthCm     150 non-null float64
PetalWidthCm      150 non-null float64
Species           150 non-null int64
dtypes: float64(4), int64(2)
memory usage: 7.2 KB
```

Splitting the data into feature set and target set

```
X1= np.array(data2.drop(columns=['Species','Id']))
```

```
y1 = np.array(data2['Species'])
```

```
X1_train,X1_test,y1_train,y1_test = train_test_split(X1,y1,test_size=0.2,random_state=2324)
```

```
X1_train_z = scale.fit_transform(X1_train)
```

```
X1_test_z = scale.fit_transform(X1_test)
```

```
y1_train = tf.keras.utils.to_categorical(y1_train, num_classes=3)
y1_test = tf.keras.utils.to_categorical(y1_test, num_classes=3)
```

```
print(y1_train.shape)
print('First 2 examples now are: ', y1_train[0:2])
```

```
(120, 3)
First 2 examples now are:  [[0. 0. 1.]
 [0. 1. 0.]]
```

```
X1_train_z = X1_train_z.astype('float32')
X1_test_z = X1_test_z.astype('float32')
y1_test = y1_test.astype('float32')
y1_train = y1_train.astype('float32')
```

Building Model in tf.keras

Build a Linear Classifier model

1. Use Dense Layer with input shape of 4 (according to the feature set) and number of outputs set to 3
2. Apply Softmax on Dense Layer outputs
3. Use SGD as Optimizer
4. Use categorical_crossentropy as loss function

```
X1_test_z.shape
```

```
(30, 4)
```

```
#Initialize Sequential model
```

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



#Normalize the data
model.add(tf.keras.layers.BatchNormalization())


#Add Dense Layer which provides 10 Outputs after applying softmax
model.add(tf.keras.layers.Dense(3, input_shape=(4,),activation='softmax'))



#Comile the model
model.compile(optimizer='sgd', loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Model Training

```
model.fit(X1_train_z, y1_train,
          validation_data=(X1_test_z, y1_test),
          epochs=100,
          batch_size=X1_train_z.shape[0])
```

```
Train on 120 samples, validate on 30 samples
Epoch 1/100
120/120 [==============================] - 1s 6ms/sample - loss: 1.0165 - accuracy: 0.46
Epoch 2/100
120/120 [==============================] - 0s 98us/sample - loss: 1.0155 - accuracy: 0.4
Epoch 3/100
120/120 [==============================] - 0s 110us/sample - loss: 1.0146 - accuracy: 0.
Epoch 4/100
120/120 [==============================] - 0s 107us/sample - loss: 1.0136 - accuracy: 0.
Epoch 5/100
120/120 [==============================] - 0s 106us/sample - loss: 1.0127 - accuracy: 0.
Epoch 6/100
120/120 [==============================] - 0s 105us/sample - loss: 1.0117 - accuracy: 0.
Epoch 7/100
120/120 [==============================] - 0s 114us/sample - loss: 1.0108 - accuracy: 0.
Epoch 8/100
120/120 [==============================] - 0s 83us/sample - loss: 1.0099 - accuracy: 0.4
Epoch 9/100
120/120 [==============================] - 0s 110us/sample - loss: 1.0089 - accuracy: 0.
Epoch 10/100
120/120 [==============================] - 0s 122us/sample - loss: 1.0080 - accuracy: 0.
Epoch 11/100
120/120 [==============================] - 0s 115us/sample - loss: 1.0071 - accuracy: 0.
Epoch 12/100
120/120 [==============================] - 0s 106us/sample - loss: 1.0061 - accuracy: 0.
Epoch 13/100
120/120 [==============================] - 0s 107us/sample - loss: 1.0052 - accuracy: 0.
Epoch 14/100
120/120 [==============================] - 0s 120us/sample - loss: 1.0043 - accuracy: 0.
Epoch 15/100
120/120 [==============================] - 0s 131us/sample - loss: 1.0034 - accuracy: 0.
Epoch 16/100
120/120 [==============================] - 0s 118us/sample - loss: 1.0025 - accuracy: 0.
Epoch 17/100
120/120 [==============================] - 0s 126us/sample - loss: 1.0016 - accuracy: 0.
Epoch 18/100
120/120 [==============================] - 0s 131us/sample - loss: 1.0007 - accuracy: 0.
Epoch 19/100
120/120 [==============================] - 0s 138us/sample - loss: 0.9998 - accuracy: 0.
Epoch 20/100
120/120 [==============================] - 0s 123us/sample - loss: 0.9988 - accuracy: 0.
Epoch 21/100
120/120 [==============================] - 0s 126us/sample - loss: 0.9979 - accuracy: 0.
Epoch 22/100
120/120 [==============================] - 0s 124us/sample - loss: 0.9970 - accuracy: 0.
Epoch 23/100
120/120 [==============================] - 0s 110us/sample - loss: 0.9961 - accuracy: 0.
Epoch 24/100
120/120 [==============================] - 0s 123us/sample - loss: 0.9953 - accuracy: 0.
Epoch 25/100
120/120 [==============================] - 0s 132us/sample - loss: 0.9944 - accuracy: 0.
Epoch 26/100
120/120 [==============================] - 0s 118us/sample - loss: 0.9935 - accuracy: 0.
Epoch 27/100
120/120 [==============================] - 0s 130us/sample - loss: 0.9926 - accuracy: 0.
Epoch 28/100
120/120 [==============================] - 0s 126us/sample - loss: 0.9917 - accuracy: 0.
```

```
Epoch 29/100
120/120 [==============================] - 0s 133us/sample - loss: 0.9908 - accuracy: 0.
Epoch 30/100
120/120 [==============================] - 0s 128us/sample - loss: 0.9899 - accuracy: 0.
Epoch 31/100
120/120 [==============================] - 0s 190us/sample - loss: 0.9890 - accuracy: 0.
Epoch 32/100
120/120 [==============================] - 0s 160us/sample - loss: 0.9882 - accuracy: 0.
Epoch 33/100
120/120 [==============================] - 0s 136us/sample - loss: 0.9873 - accuracy: 0.
Epoch 34/100
120/120 [==============================] - 0s 122us/sample - loss: 0.9864 - accuracy: 0.
Epoch 35/100
120/120 [==============================] - 0s 136us/sample - loss: 0.9855 - accuracy: 0.
Epoch 36/100
120/120 [==============================] - 0s 142us/sample - loss: 0.9847 - accuracy: 0.
Epoch 37/100
120/120 [==============================] - 0s 128us/sample - loss: 0.9838 - accuracy: 0.
Epoch 38/100
120/120 [==============================] - 0s 134us/sample - loss: 0.9829 - accuracy: 0.
Epoch 39/100
120/120 [==============================] - 0s 134us/sample - loss: 0.9821 - accuracy: 0.
Epoch 40/100
120/120 [==============================] - 0s 141us/sample - loss: 0.9812 - accuracy: 0.
Epoch 41/100
120/120 [==============================] - 0s 135us/sample - loss: 0.9803 - accuracy: 0.
Epoch 42/100
120/120 [==============================] - 0s 107us/sample - loss: 0.9795 - accuracy: 0.
Epoch 43/100
120/120 [==============================] - 0s 132us/sample - loss: 0.9786 - accuracy: 0.
Epoch 44/100
120/120 [==============================] - 0s 132us/sample - loss: 0.9777 - accuracy: 0.
Epoch 45/100
120/120 [==============================] - 0s 137us/sample - loss: 0.9769 - accuracy: 0.
Epoch 46/100
120/120 [==============================] - 0s 88us/sample - loss: 0.9760 - accuracy: 0.5
Epoch 47/100
120/120 [==============================] - 0s 120us/sample - loss: 0.9752 - accuracy: 0.
Epoch 48/100
120/120 [==============================] - 0s 148us/sample - loss: 0.9743 - accuracy: 0.
Epoch 49/100
120/120 [==============================] - 0s 119us/sample - loss: 0.9735 - accuracy: 0.
Epoch 50/100
120/120 [==============================] - 0s 152us/sample - loss: 0.9726 - accuracy: 0.
Epoch 51/100
120/120 [==============================] - 0s 130us/sample - loss: 0.9718 - accuracy: 0.
Epoch 52/100
120/120 [==============================] - 0s 107us/sample - loss: 0.9709 - accuracy: 0.
Epoch 53/100
120/120 [==============================] - 0s 147us/sample - loss: 0.9701 - accuracy: 0.
Epoch 54/100
120/120 [==============================] - 0s 155us/sample - loss: 0.9692 - accuracy: 0.
Epoch 55/100
120/120 [==============================] - 0s 128us/sample - loss: 0.9684 - accuracy: 0.
Epoch 56/100
120/120 [==============================] - 0s 121us/sample - loss: 0.9676 - accuracy: 0.
Epoch 57/100
120/120 [==============================] - 0s 137us/sample - loss: 0.9667 - accuracy: 0.
```