**DEEP LEARNING**

## Mini project:3

Date:09/07/2020

Presented By:

Vismay Gunda,

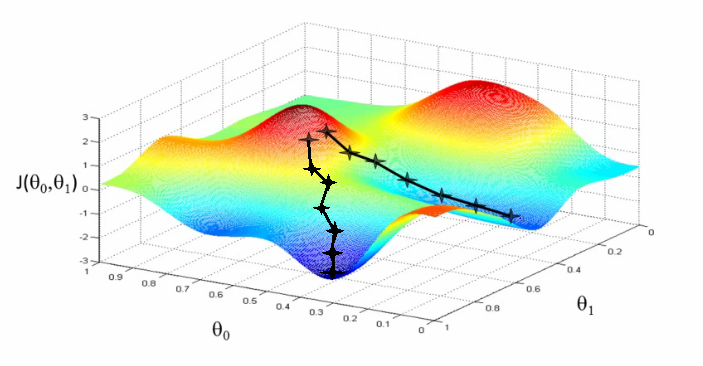
Computer science engineering,

KLS Gogte institute of Technology Belagavi

Title:

**“GRADIENT DESCENT TRAJECTORY USING TENSORFLOW.”**

## **1. Introduction:**



Gradient descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. In machine learning, we use gradient descent to update the parameters of our model.

**2. packages**

**Python code:**

In[1]:

# set tf 1.x for colab

%tensorflow\_version 1.x

Out[1]:

TensorFlow 1.x selected.

In[2]:

! shred -u setup\_google\_colab.py

! wget https://raw.githubusercontent.com/hse-aml/intro-to-dl/master/setup\_google\_colab.py -O setup\_google\_colab.py

import setup\_google\_colab

# please, uncomment the week you're working on

setup\_google\_colab.setup\_week2()

Out[2]:

--2020-07-09 14:35:57-- <https://raw.githubusercontent.com/hse-aml/intro-to-dl/master/setup_google_colab.py> Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.0.133, 151.101.64.133, 151.101.128.133, ... Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.0.133|:443... connected. HTTP request sent, awaiting response... 200 OK Length: 3636 (3.6K) [text/plain] Saving to: ‘setup\_google\_colab.py’ setup\_google\_colab. 100%[===================>] 3.55K --.-KB/s in 0s 2020-07-09 14:35:57 (52.1 MB/s) - ‘setup\_google\_colab.py’ saved [3636/3636] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* inception\_v3\_weights\_tf\_dim\_ordering\_tf\_kernels\_notop.h5 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* cifar-10-batches-py.tar.gz \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* mnist.npz

**Note:**

If you run the notebook locally, you should be able to access TensorBoard on http://127.0.0.1:7007/

In[3]:

import tensorflow as tf

import sys

sys.path.append("../..")

from keras\_utils import reset\_tf\_session

s = reset\_tf\_session()

print("We're using TF", tf.\_\_version\_\_)

Out[3]:

Using TensorFlow backend.

WARNING:tensorflow:From /content/keras\_utils.py:68: The name tf.get\_default\_session is deprecated. Please use tf.compat.v1.get\_default\_session instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow\_backend.py:79: The name tf.reset\_default\_graph is deprecated. Please use tf.compat.v1.reset\_default\_graph instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow\_backend.py:82: The name tf.placeholder is deprecated. Please use tf.compat.v1.placeholder instead.

WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow\_backend.py:84: The name tf.get\_default\_graph is deprecated. Please use tf.compat.v1.get\_default\_graph instead.

WARNING:tensorflow:From /content/keras\_utils.py:75: The name tf.ConfigProto is deprecated. Please use tf.compat.v1.ConfigProto instead.

WARNING:tensorflow:From /content/keras\_utils.py:77: The name tf.InteractiveSession is deprecated. Please use tf.compat.v1.InteractiveSession instead.

We're using TF 1.15.2

**3.Computation of sum of squares of numbers from 0 to N-1**

**Python code:**

In[4]:

# An integer parameter

N = tf.placeholder('int64', name="input\_to\_your\_function")

# A recipe on how to produce the same result

result = tf.reduce\_sum(tf.range(N)\*\*2)

In[5]:

# just a graph definition

result

Out[5]:

<tf.Tensor 'Sum:0' shape=() dtype=int64>

In[6]:

%%time

# actually executing

result.eval({N: 10\*\*5})

Out[6]:

CPU times: user 326 ms, sys: 206 ms, total: 533 ms

Wall time: 5.45 s

333328333350000

In[7]:

# logger for tensorboard

writer = tf.summary.FileWriter("tensorboard\_logs", graph=s.graph)

**4. Working procedures**

1.Define placeholders where you'll send inputs

2.Make a symbolic graph: a recipe for mathematical transformation of those placeholders

3.Compute outputs of your graph with particular values for each placeholder

* + output.eval({placeholder: value})
  + s.run(output, {placeholder: value})

So far there are two main entities: "placeholder" and "transformation" (operation output)

* Both can be numbers, vectors, matrices, tensors, etc.
* Both can be int32/64, floats, booleans (uint8) of various size.
* You can define new transformations as an arbitrary operation on placeholders and other transformations
  + tf.reduce\_sum(tf.arange(N)\*\*2) are 3 sequential transformations of placeholder N
  + There's a tensorflow symbolic version for every numpy function
    - * a+b, a/b, a\*\*b, ... behave just like in numpy
      * np.mean -> tf.reduce\_mean
      * np.arange -> tf.range
      * np.cumsum -> tf.cumsum
      * If you can't find the operation you need, see the [docs](https://www.tensorflow.org/versions/r1.3/api_docs/python).

tf.contrib has many high-level features, may be worth a look.

**Python code:**

In[8]:

**with** tf.name\_scope("Placeholders\_examples"):

*# Default placeholder that can be arbitrary float32*

*# scalar, vertor, matrix, etc.*

arbitrary\_input = tf.placeholder('float32')

*# Input vector of arbitrary length*

input\_vector = tf.placeholder('float32', shape=(**None**,))

*# Input vector that \_must\_ have 10 elements and integer type*

fixed\_vector = tf.placeholder('int32', shape=(10,))

*# Matrix of arbitrary n\_rows and 15 columns*

*# (e.g. a minibatch of your data table)*

input\_matrix = tf.placeholder('float32', shape=(**None**, 15))

*# You can generally use None whenever you don't need a specific shape*

input1 = tf.placeholder('float64', shape=(**None**, 100, **None**))

input2 = tf.placeholder('int32', shape=(**None**, **None**, 3, 224, 224))

*# elementwise multiplication*

double\_the\_vector = input\_vector\*2

*# elementwise cosine*

elementwise\_cosine = tf.cos(input\_vector)

*# difference between squared vector and vector itself plus one*

vector\_squares = input\_vector\*\*2 - input\_vector + 1

In[9]:

my\_vector = tf.placeholder('float32', shape=(**None**,), name="VECTOR\_1")

my\_vector2 = tf.placeholder('float32', shape=(**None**,))

my\_transformation = my\_vector \* my\_vector2 / (tf.sin(my\_vector) + 1)

print(my\_transformation)

Out[9]:

Tensor("truediv:0", shape=(?,), dtype=float32)

In[10]:

dummy = np.arange(5).astype('float32')

print(dummy)

my\_transformation.eval({my\_vector: dummy, my\_vector2: dummy[::-1]})

Out[10]:

[0. 1. 2. 3. 4.]

array([0. , 1.6291324, 2.0950115, 2.6289961, 0. ],

dtype=float32)

In[11]:

writer.add\_graph(my\_transformation.graph) writer.flush()

**Note:**

1.TensorBoard allows writing scalars, images, audio, histogram.

2. From above exercise it is cleared that

a) Tensorflow is based on computation graphs

b) A graph consists of placeholders and transformations

# **5.Loss function: Mean Squared Error**

Loss function must be a part of the graph as well, so that we can do backpropagation.

**Python code:**

In[12]:

with tf.name\_scope("MSE"):

y\_true = tf.placeholder("float32", shape=(None,), name="y\_true")

y\_predicted = tf.placeholder("float32", shape=(None,), name="y\_predicted")

# Implement MSE(y\_true, y\_predicted), use tf.reduce\_mean(...)

mse = tf.reduce\_mean(tf.square(y\_predicted - y\_true))

def compute\_mse(vector1, vector2):

return mse.eval({y\_true: vector1, y\_predicted: vector2})

In[13]:

writer.add\_graph(mse.graph)

writer.flush()

In[14]:

# Rigorous local testing of MSE implementation

import sklearn.metrics

for n in [1, 5, 10, 10\*\*3]:

elems = [np.arange(n), np.arange(n, 0, -1), np.zeros(n),

np.ones(n), np.random.random(n), np.random.randint(100, size=n)]

for el in elems:

for el\_2 in elems:

true\_mse = np.array(sklearn.metrics.mean\_squared\_error(el, el\_2))

my\_mse = compute\_mse(el, el\_2)

if not np.allclose(true\_mse, my\_mse):

print('mse(%s,%s)' % (el, el\_2))

print("should be: %f, but your function returned %f" % (true\_mse, my\_mse))

raise ValueError('Wrong result')

# **6.Variables**

Placeholder and transformation values are not stored in the graph once the execution is finished. This isn't too comfortable if you want your model to have parameters (e.g. network weights) that are always present, but can change their value over time.

Tensorflow solves this with tf.Variable objects.

* You can assign variable a value at any time in your graph
* Unlike placeholders, there's no need to explicitly pass values to variables when s.run(...)-ing
* You can use variables the same way you use transformations

**Python code:**

In[15]:

# Creating a shared variable

shared\_vector\_1 = tf.Variable(initial\_value=np.ones(5),

name="example\_variable")

In[16]:

# Initialize variable(s) with initial values

s.run(tf.global\_variables\_initializer())

# Evaluating the shared variable

print("Initial value", s.run(shared\_vector\_1))

Out[16]:

Initial value [1. 1. 1. 1. 1.]

In[17]:

# Setting a new value

s.run(shared\_vector\_1.assign(np.arange(5)))

# Getting that new value

print("New value", s.run(shared\_vector\_1))

Out[17]:

New value [0. 1. 2. 3. 4.]

# **7 tf.gradients**

Tensorflow can compute derivatives and gradients automatically using the computation graph

* True to its name it can manage matrix derivatives
* Gradients are computed as a product of elementary derivatives via the chain rule:

∂f(g(x))/∂x=(∂f(g(x))/∂g(x))⋅(∂g(x)/∂x)

**Python code:**

In[18]:

my\_scalar = tf.placeholder('float32')

scalar\_squared = my\_scalar\*\*2

# A derivative of scalar\_squared by my\_scalar

derivative = tf.gradients(scalar\_squared, [my\_scalar, ])

derivative

Out[18]:

[<tf.Tensor 'gradients/pow\_1\_grad/Reshape:0' shape=<unknown> dtype=float32>]

In[19]:

import matplotlib.pyplot as plt

%matplotlib inline

x = np.linspace(-3, 3)

x\_squared, x\_squared\_der = s.run([scalar\_squared, derivative[0]],

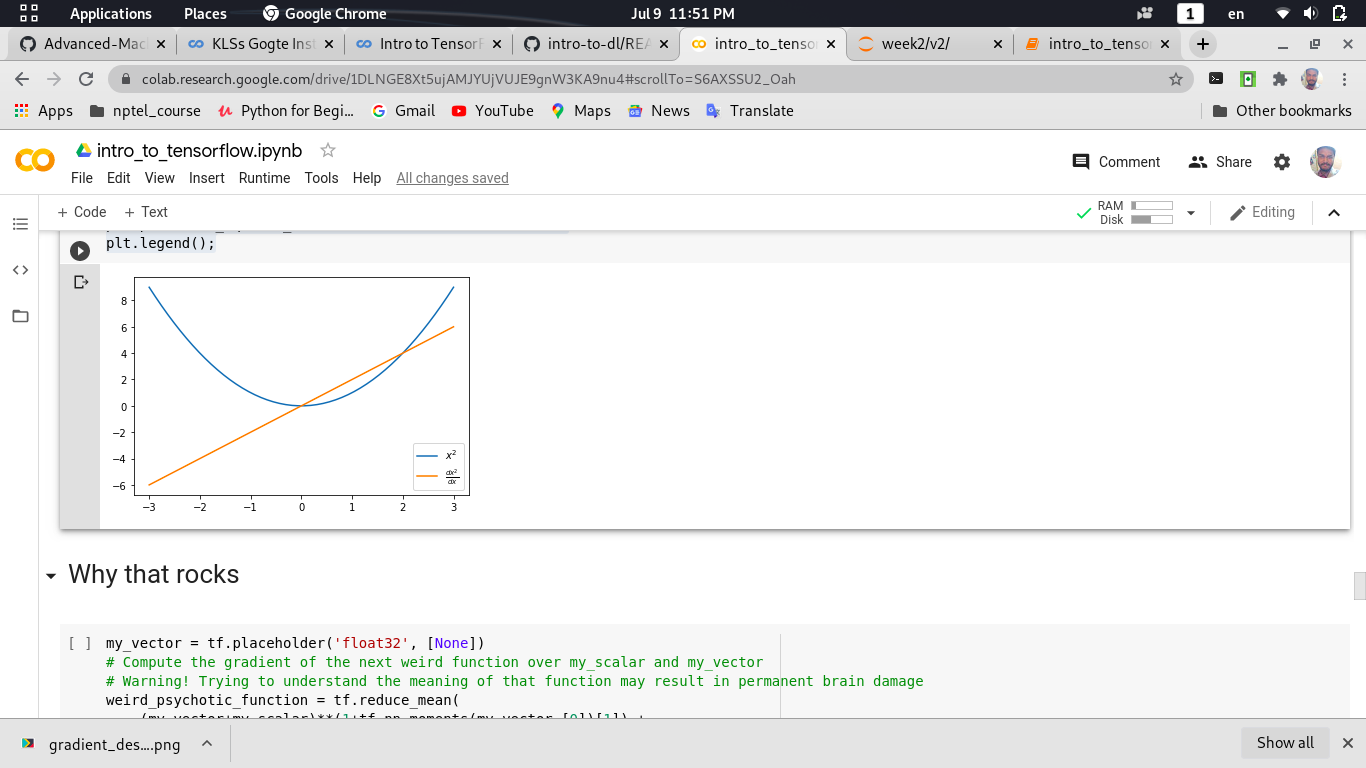
{my\_scalar:x})

plt.plot(x, x\_squared,label="$x^2$")

plt.plot(x, x\_squared\_der, label=r"$\frac{dx^2}{dx}$")

plt.legend();

Out[19]:



In[20]:

my\_vector = tf.placeholder('float32', [None])

# Compute the gradient of the next weird function over my\_scalar and my\_vector

# Warning! Trying to understand the meaning of that function may result in permanent brain damage

weird\_psychotic\_function = tf.reduce\_mean(

(my\_vector+my\_scalar)\*\*(1+tf.nn.moments(my\_vector,[0])[1]) +

1./ tf.atan(my\_scalar))/(my\_scalar\*\*2 + 1) + 0.01\*tf.sin(

2\*my\_scalar\*\*1.5)\*(tf.reduce\_sum(my\_vector)\* my\_scalar\*\*2

)\*tf.exp((my\_scalar-4)\*\*2)/(

1+tf.exp((my\_scalar-4)\*\*2))\*(1.-(tf.exp(-(my\_scalar-4)\*\*2)

)/(1+tf.exp(-(my\_scalar-4)\*\*2)))\*\*2

der\_by\_scalar = tf.gradients(weird\_psychotic\_function, my\_scalar)

der\_by\_vector = tf.gradients(weird\_psychotic\_function, my\_vector)

In[21]:

# Plotting the derivative

scalar\_space = np.linspace(1, 7, 100)

y = [s.run(weird\_psychotic\_function, {my\_scalar:x, my\_vector:[1, 2, 3]})

for x in scalar\_space]

plt.plot(scalar\_space, y, label='function')

y\_der\_by\_scalar = [s.run(der\_by\_scalar,

{my\_scalar:x, my\_vector:[1, 2, 3]})

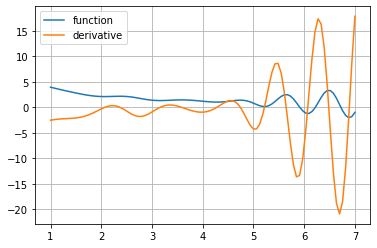
for x in scalar\_space]

plt.plot(scalar\_space, y\_der\_by\_scalar, label='derivative')

plt.grid()

plt.legend();

Out[21]:



# **8. Optimisers**

While we can perform gradient descent by hand with automatic gradients from above, tensorflow also has some optimization methods that can be implemented Using momentum & rmsprop

**Python code:**

In[22]:

y\_guess = tf.Variable(np.zeros(2, dtype='float32'))

y\_true = tf.range(1, 3, dtype='float32')

loss = tf.reduce\_mean((y\_guess - y\_true + 0.5\*tf.random\_normal([2]))\*\*2)

step = tf.train.MomentumOptimizer(0.03, 0.5).minimize(loss, var\_list=y\_guess)

In[23]:

from matplotlib import animation, rc

import matplotlib\_utils

from IPython.display import HTML, display\_html

# nice figure settings

fig, ax = plt.subplots()

y\_true\_value = s.run(y\_true)

level\_x = np.arange(0, 2, 0.02)

level\_y = np.arange(0, 3, 0.02)

X, Y = np.meshgrid(level\_x, level\_y)

Z = (X - y\_true\_value[0])\*\*2 + (Y - y\_true\_value[1])\*\*2

ax.set\_xlim(-0.02, 2)

ax.set\_ylim(-0.02, 3)

s.run(tf.global\_variables\_initializer())

ax.scatter(\*s.run(y\_true), c='red')

contour = ax.contour(X, Y, Z, 10)

ax.clabel(contour, inline=1, fontsize=10)

line, = ax.plot([], [], lw=2)

# start animation with empty trajectory

def init():

line.set\_data([], [])

return (line,)

trajectory = [s.run(y\_guess)]

# one animation step (make one GD step)

def animate(i):

s.run(step)

trajectory.append(s.run(y\_guess))

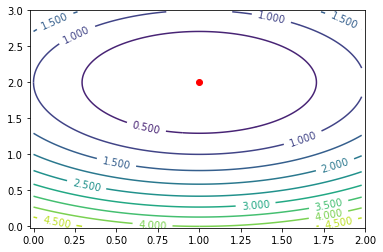
line.set\_data(\*zip(\*trajectory))

return (line,)

anim = animation.FuncAnimation(fig, animate, init\_func=init,

frames=100, interval=20, blit=True)

Out[23]:



In[24]:

try:

display\_html(HTML(anim.to\_html5\_video()))

except (RuntimeError, KeyError):

# In case the build-in renderers are unaviable, fall back to

# a custom one, that doesn't require external libraries

anim.save(None, writer=matplotlib\_utils.SimpleMovieWriter(0.001))

Out[24]: