Backpropagation

In this assignment, you will implement Backpropagation from scratch. You will then verify the correctness of the your implementation using a "grader" function/cell (provided by us) which will match your implmentation.

The grader fucntion would help you validate the correctness of your code.

Please submit the final Colab notebook in the classroom ONLY after you have verified your code using the grader function/cell.

```
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sympy import *
from statistics import mean
import pickle
from tqdm import tqdm
```

Loading data

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

```
     Mounted at /content/drive
```

```
with open('drive/My Drive/Backpropagation-Gradient-Checking/data.pkl', 'rb') as f:
    data = pickle.load(f)
print(data.shape)
X = data[:, :5]
y = data[:, -1]
print(X.shape, y.shape)
```

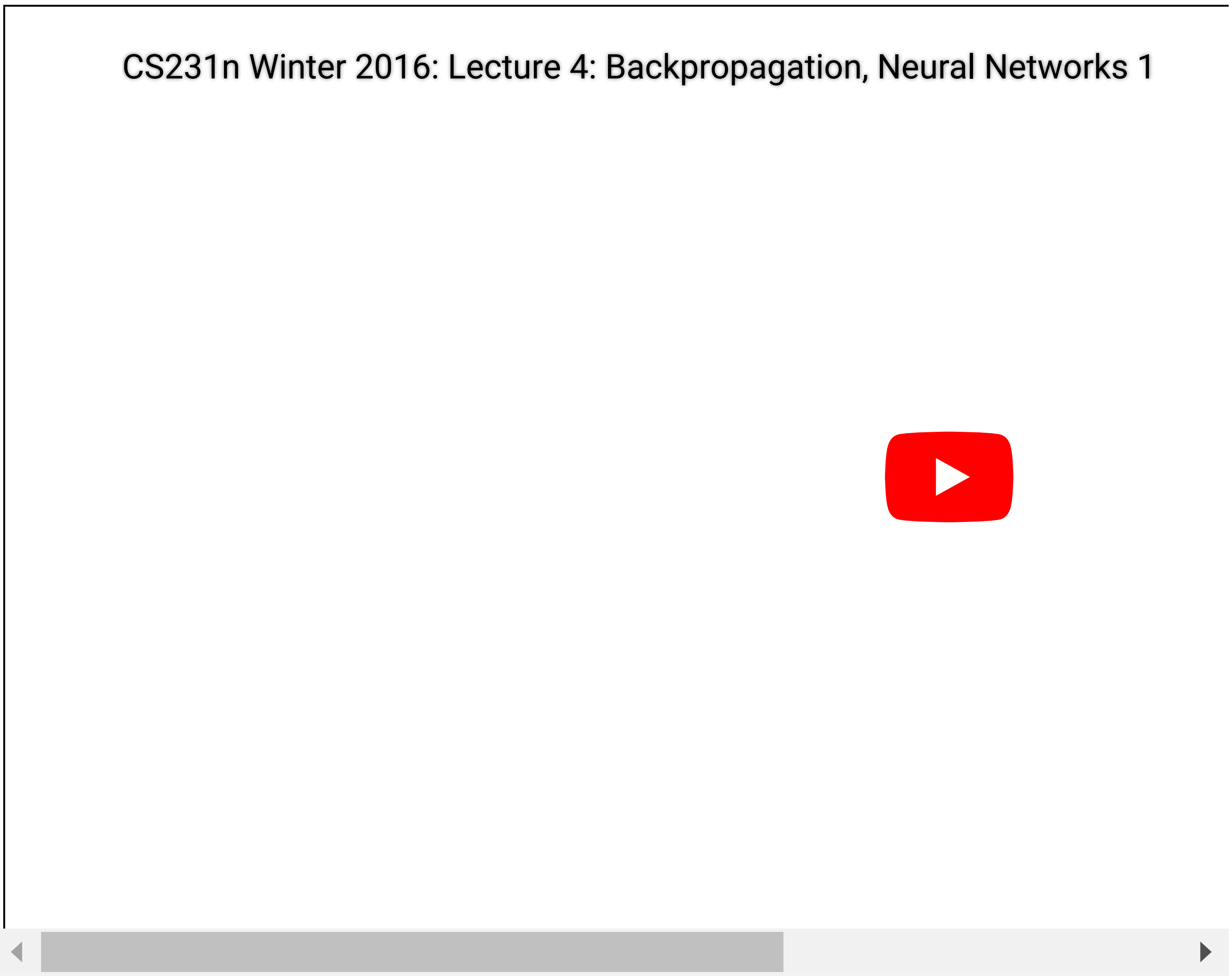
```
     (506, 6)
     (506, 5) (506,)
```

```
w = []
mean, std = 0, 0.1 # mean and standard deviation
w = np.random.normal(mean, std, 9)
w
```

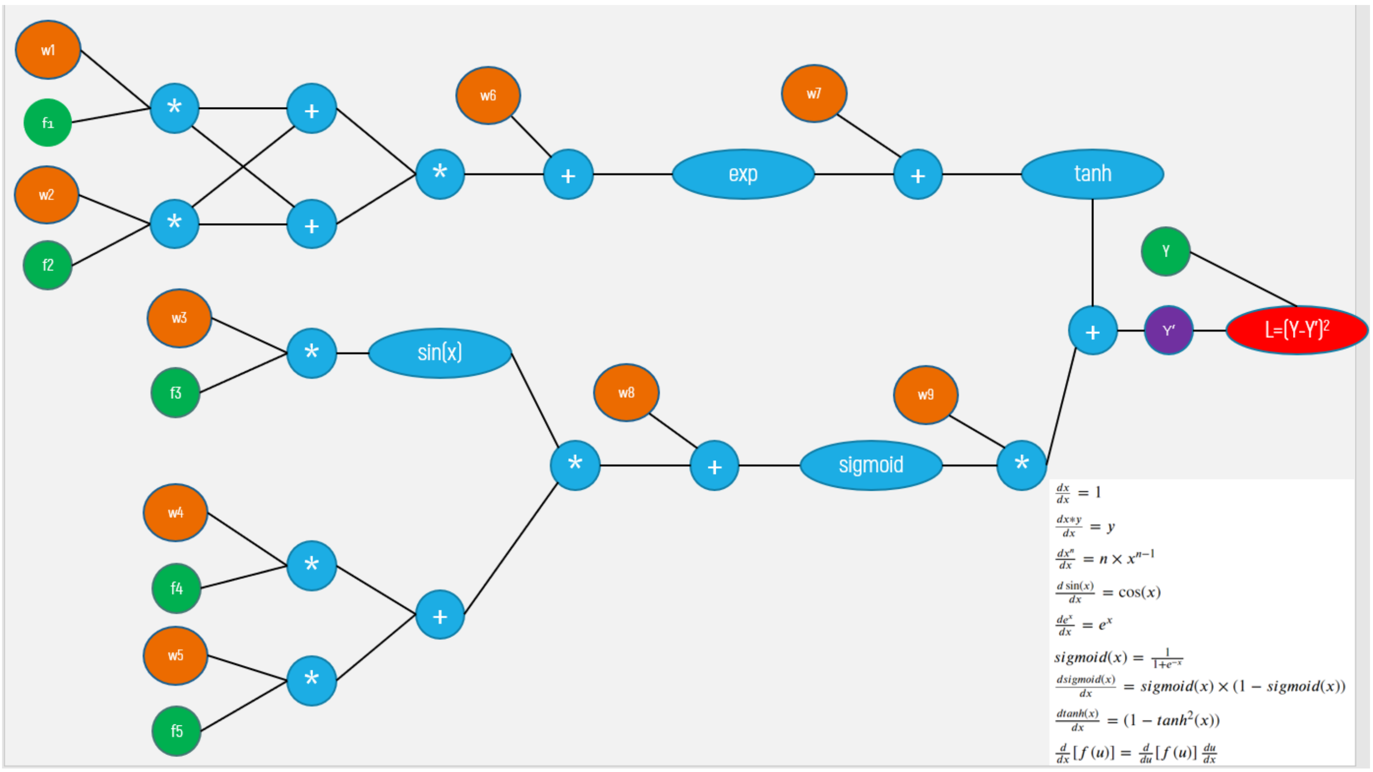
```
    array([ 0.02259681, -0.03064837, -0.1105882 ,  0.04404644, -0.20535659,
           -0.03229631,  0.07434674, -0.13962758,  0.03486219])
```

Check this video for better understanding of the computational graphs and back propagation

```
from IPython.display import YouTubeVideo
YouTubeVideo('i94OvYb6noo',width="1000",height="500")
```


CS231n Winter 2016: Lecture 4: Backpropagation, Neural Networks 1

Computational graph



$$\frac{dx}{dx} = 1$$
$$\frac{dx*y}{dx} = y$$
$$\frac{dx^n}{dx} = n \times x^{n-1}$$
$$\frac{d\sin(x)}{dx} = \cos(x)$$
$$\frac{de^x}{dx} = e^x$$
$$sigmoid(x) = \frac{1}{1+e^{-x}}$$
$$\frac{dsigmoid(x)}{dx} = sigmoid(x) \times (1 - sigmoid(x))$$
$$\frac{dtanh(x)}{dx} = (1 - tanh^2(x))$$
$$\frac{d}{dx}[f(u)] = \frac{d}{du}[f(u)]\frac{du}{dx}$$

- **If you observe the graph, we are having input features [f1, f2, f3, f4, f5] and 9 weights [w1, w2, w3, w4, w5, w6, w7, w8, w9]**.

- **The final output of this graph is a value L which is computed as (Y-Y')^2**

Task 1: Implementing Forward propagation, Backpropagation and Gradient checking

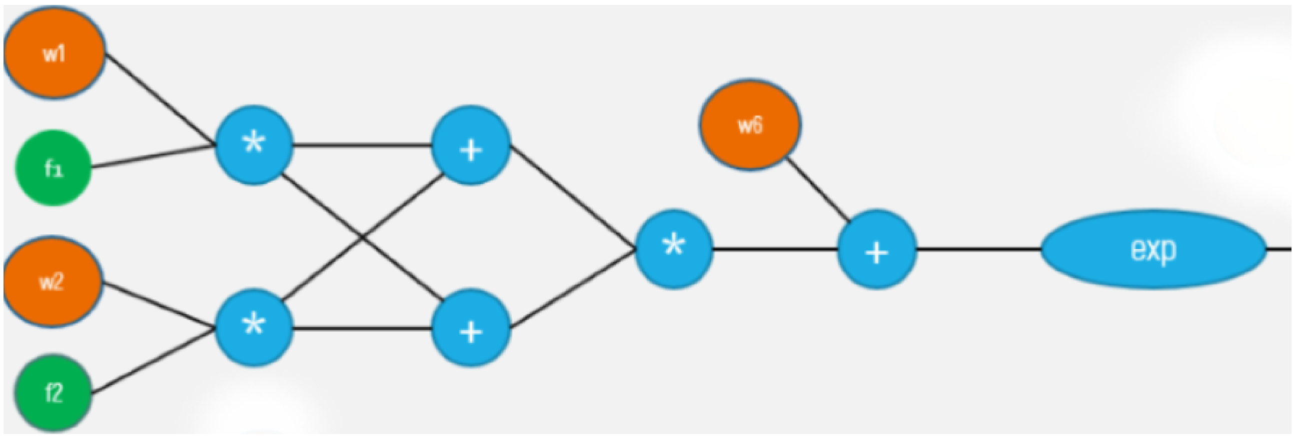
Task 1.1

Forward propagation

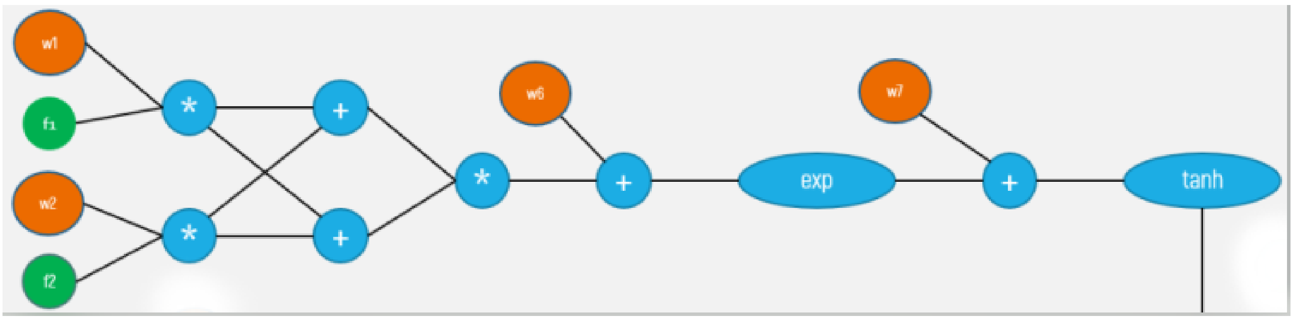
- **Forward propagation**(Write your code in def forward_propagation())

  For easy debugging, we will break the computational graph into 3 parts.

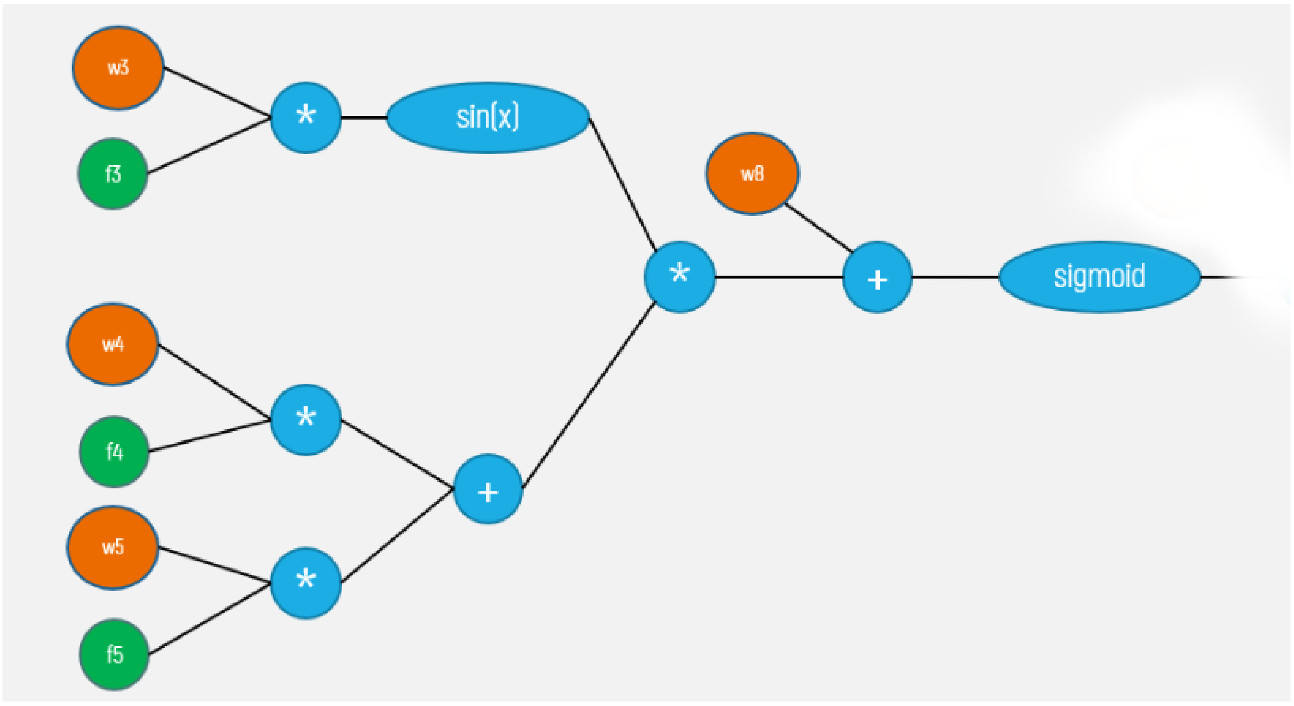
Part 1



Part 2



Part 3



```
def sigmoid(z):
    '''In this function, we will compute the sigmoid(z)'''
    # we can use this function in forward and backward propagation
    # write the code to compute the sigmoid value of z and return that value
    return (1/(1 + np.exp(-z)))
```

```
def grader_sigmoid(z):
  #if you have written the code correctly then the grader function will output true
  val=sigmoid(z)
  assert(val==0.8807970779778823)
  return True
grader_sigmoid(2)

     True
```

```
def forward_propagation(x, y, w):
        '''In this function, we will compute the forward propagation '''
        # X: input data point, note that in this assignment you are having 5-d data points
        # y: output varible
        # W: weight array, its of length 9, W[0] corresponds to w1 in graph, W[1] correspo
        # you have to return the following variables
        # exp= part1 (compute the forward propagation until exp and then store the values
        # tanh =part2(compute the forward propagation until tanh and then store the values
        # sig = part3(compute the forward propagation until sigmoid and then store the val
        # we are computing one of the values for better understanding

        val_1= (w[0]*x[0]+w[1]*x[1]) * (w[0]*x[0]+w[1]*x[1]) + w[5]
        part_1 = np.exp(val_1)
        part_2 = np.tanh(part_1 + w[6])
        sin_value = np.sin(x[2]*w[2])
        cos_value = np.cos(x[2] * w[2])
        product_of_data_point_weight_3_4 = x[3]*w[3] + x[4]*w[4]
        val_2 = sin_value * product_of_data_point_weight_3_4
        val_3 = val_2 + w[7]
        part_3 = sigmoid(val_3)
        val_4 = part_3 * w[8]
        y_pred = val_4 + part_2
        loss = (y - y_pred)**2
        der_loss = -2*(y-y_pred)

        # after computing part1,part2 and part3 compute the value of y' from the main Comp
        # write code to compute the value of L=(y-y')^2 and store it in variable loss
        # compute derivative of L  w.r.to y' and store it in dy_pred
        # Create a dictionary to store all the intermediate values i.e. dy_pred ,loss,exp,
        # we will be using the dictionary to find values in backpropagation, you can add o

        forward_dict={}
        forward_dict['exp']= part_1
        forward_dict['sigmoid'] = part_3
        forward_dict['tanh'] = part_2
        forward_dict['loss'] = loss
        forward_dict['dy_pred'] = der_loss
        forward_dict['val_1'] = val_1
        forward_dict['sin_value'] = sin_value
        forward_dict['cos_value'] = cos_value
        forward_dict['product_of_data_point_weight_3_4'] = product_of_data_point_weight_3_
        forward_dict['val_2'] = val_2
        forward_dict['val_3'] = val_3
        forward_dict['val_4'] = val_4
        return forward_dict
```

```
def grader_forwardprop(data):
    dl = (data['dy_pred']==-1.9285278284819143)
    loss=(data['loss']==0.9298048963072919)
    part1=(data['exp']==1.1272967040973583)
    part2=(data['tanh']==0.8417934192562146)
    part3=(data['sigmoid']==0.5279179387419721)
    assert(dl and loss and part1 and part2 and part3)
    return True
w=np.ones(9)*0.1
d1=forward_propagation(X[0],y[0],w)
grader_forwardprop(d1)
```

```
    True
```

Task 1.2

Backward propagation

```
def backward_propagation(x,y,w,forward_dict):
    '''In this function, we will compute the backward propagation '''
    # forward_dict: the outputs of the forward_propagation() function
    # write code to compute the gradients of each weight [w1,w2,w3,...,w9]
    # Hint: you can use dict type to store the required variables
    # dw1 = # in dw1 compute derivative of L w.r.to w1
    # dw2 = # in dw2 compute derivative of L w.r.to w2
    # dw3 = # in dw3 compute derivative of L w.r.to w3
    # dw4 = # in dw4 compute derivative of L w.r.to w4
    # dw5 = # in dw5 compute derivative of L w.r.to w5
    # dw6 = # in dw6 compute derivative of L w.r.to w6
    # dw7 = # in dw7 compute derivative of L w.r.to w7
    # dw8 = # in dw8 compute derivative of L w.r.to w8
    # dw9 = # in dw9 compute derivative of L w.r.to w9
    L = forward_dict['loss']
    derv_L = forward_dict['dy_pred']
    tan = forward_dict['tanh']
    part1 = forward_dict['exp']
    sigma = forward_dict['sigmoid']
    sin_val =forward_dict['sin_value']
    cos_val =forward_dict['cos_value']

    dw1 = derv_L * ((1 - (tan)**2) * x[0] * part1) * 2 * ((w[0] * x[0]) + (w[1] * x[1]))
    dw2 = derv_L * ((1 - (tan)**2) * x[1] * part1) * 2 * ((w[0] * x[0]) + (w[1] * x[1]))
    dw3 = derv_L * ((sigma) * (1 - sigma)) * x[2] * w[8] * ((w[3] * x[3]) + (w[4] * x[4]))
    dw4 = derv_L * ((sigma) * (1 - sigma)) * x[3] * w[8] * sin_val
    dw5 = derv_L * ((sigma) * (1 - sigma)) * x[4] * w[8] * sin_val
    dw6 = derv_L * (1 - (tan)**2) * part1
    dw7 = derv_L * ((1 - (tan)**2))
    dw8 = derv_L * ((sigma) * (1 - sigma)) * w[8]
    dw9 = derv_L * sigma

    backward_dict={}
    #store the variables dw1,dw2 etc. in a dict as backward_dict['dw1']= dw1,backward_dict
    backward_dict['dw1']= dw1
    backward_dict['dw2']= dw2
    backward_dict['dw3']= dw3
    backward_dict['dw4']= dw4
    backward_dict['dw5']= dw5
    backward_dict['dw6']= dw6
    backward_dict['dw7']= dw7
    backward_dict['dw8']= dw8
    backward_dict['dw9']= dw9

    return backward_dict



def grader_backprop(data):
    dw1=(np.round(data['dw1'],6)==-0.229733)
    dw2=(np.round(data['dw2'],6)==-0.021408)
    dw3=(np.round(data['dw3'],6)==-0.005625)
    dw4=(np.round(data['dw4'],6)==-0.004658)
    dw5=(np.round(data['dw5'],6)==-0.001008)
    dw6=(np.round(data['dw6'],6)==-0.633475)
    dw7=(np.round(data['dw7'],6)==-0.561942)
    dw8=(np.round(data['dw8'],6)==-0.048063)
    dw9=(np.round(data['dw9'],6)==-1.018104)
    assert(dw1 and dw2 and dw3 and dw4 and dw5 and dw6 and dw7 and dw8 and dw9)
    return True
w=np.ones(9)*0.1
forward_dict=forward_propagation(X[0],y[0],w)
backward_dict=backward_propagation(X[0],y[0],w,forward_dict)
grader_backprop(backward_dict)
```

```
    True
```

Task 1.3

Gradient clipping

Check this blog link for more details on Gradient clipping

we know that the derivative of any function is

$$\lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$$

- The definition above can be used as a numerical approximation of the derivative. Taking an epsilon small enough, the calculated approximation will have an error in the range of epsilon squared.
- In other words, if epsilon is 0.001, the approximation will be off by 0.00001.

Therefore, we can use this to approximate the gradient, and in turn make sure that backpropagation is implemented properly. This forms the basis of **gradient checking!**

Gradient checking example

lets understand the concept with a simple example: $f(w1, w2, x1, x2) = w_1^2 . x_1 + w_2 . x_2$

from the above function , lets assume $w_1 = 1$, $w_2 = 2$, $x_1 = 3$, $x_2 = 4$ the gradient of f w.r.t w_1 is

$$\begin{aligned} \frac{df}{dw_1} = dw_1 &= 2.w_1 . x_1 \\ &= 2.1.3 \\ &= 6 \end{aligned}$$

let calculate the aproximate gradient of w_1 as mentinoned in the above formula and considering $\epsilon = 0.0001$

$$\begin{aligned} dw_1^{approx} &= \frac{f(w1+\epsilon, w2, x1, x2) - f(w1-\epsilon, w2, x1, x2)}{2\epsilon} \\ &= \frac{((1+0.0001)^2 .3 + 2.4) - ((1-0.0001)^2 .3 + 2.4)}{2\epsilon} \\ &= \frac{(1.00020001.3 + 2.4) - (0.99980001.3 + 2.4)}{2 * 0.0001} \\ &= \frac{(11.00060003) - (10.99940003)}{0.0002} \\ &= 5.99999999999 \end{aligned}$$

Then, we apply the following formula for gradient check: *gradient_check* = $\frac{\|(dW - dW^{approx})\|_2}{\|(dW)\|_2 + \|(dW^{approx})\|_2}$

The equation above is basically the Euclidean distance normalized by the sum of the norm of the vectors. We use normalization in case that one of the vectors is very small. As a value for epsilon, we usually opt for 1e-7. Therefore, if gradient check return a value less than 1e-7, then it means that backpropagation was implemented correctly. Otherwise, there is potentially a mistake in your implementation. If the value exceeds 1e-3, then you are sure that the code is not correct.

in our example: *gradient_check* $= \frac{(6 - 5.999999999994898)}{(6 + 5.999999999994898)} = 4.2514140356330737e^{-13}$

you can mathamatically derive the same thing like this

$$\begin{aligned} dw_1^{approx} &= \frac{f(w1+\epsilon,w2,x1,x2)-f(w1-\epsilon,w2,x1,x2)}{2\epsilon} \\ &= \frac{((w_1+\epsilon)^2.x_1+w_2.x_2)-((w_1-\epsilon)^2.x_1+w_2.x_2)}{2\epsilon} \\ &= \frac{4.\epsilon.w_1.x_1}{2\epsilon} \\ &= 2.w_1.x_1 \end{aligned}$$

Implement Gradient checking

(Write your code in def gradient_checking())

Algorithm

```
W = initilize_randomly
def gradient_checking(data_point, W):
    # compute the L value using forward_propagation()
    # compute the gradients of W using backword_propagation()
    approx_gradients = []
    for each wi weight value in W:
        # add a small value to weight wi, and then find the values of L with the updated
        # subtract a small value to weight wi, and then find the values of L with the up
        # compute the approximation gradients of weight wi
        approx_gradients.append(approximation gradients of weight wi)
    # compare the gradient of weights W from backword_propagation() with the aproximatio
  gradient_check formula
    return gradient_check
NOTE: you can do sanity check by checking all the return values of gradient_checking(),
 they have to be zero. if not you have bug in your code
```

```
def gradient_checking(x,y,w,eps):
    # compute the dict value using forward_propagation()
    # compute the actual gradients of W using backword_propagation()
    forward_dict=forward_propagation(x,y,w)
    backward_dict=backward_propagation(x,y,w,forward_dict)

    #we are storing the original gradients for the given datapoints in a list

    original_gradients_list=list(backward_dict.values())
    # make sure that the order is correct i.e. first element in the list corresponds to  d
    # you can use reverse function if the values are in reverse order

    approx_gradients_list=[]
    #now we have to write code for approx gradients, here you have to make sure that you u
    #write your code here and append the approximate gradient value for each weight in  ap

    for i in range(len(w)):

        w[i] += eps                                # adding small value epsilon to each wi
        fp_plus = forward_propagation(x,y,w)
        L_plus = fp_plus['loss']

        w[i] -= 2*eps
        fp_minus = forward_propagation(x,y,w)
        L_minus = fp_minus['loss']

        # compute the approximation gradients of weight wi
        approx = (L_plus - L_minus)/(2*eps)
        approx_gradients_list.append(approx)

    #performing gradient check operation
    original_gradients_list=np.array(original_gradients_list)
    approx_gradients_list=np.array(approx_gradients_list)
    numerator = [x1-x2 for (x1,x2) in zip(original_gradients_list,approx_gradients_list)]
    numerator = np.linalg.norm(numerator)
    denominator = np.linalg.norm(original_gradients_list) + np.linalg.norm(approx_gradient
    gradient_check_value = numerator/denominator
    #gradient_check_value =(original_gradients_list-approx_gradients_list)/(original_gradi

    return original_gradients_list,approx_gradients_list,gradient_check_value
```

```
eps=10**-7
w=[ 0.00271756,  0.01260512,  0.00167639, -0.00207756,  0.00720768,
   0.00114524,  0.00684168,  0.02242521,  0.01296444]
original_gradients_list,approx_gradients_list,gradient_check_value = gradient_checking(X[0

gradient_check_value
```

```
     5.309771069360733e-08
```

```
def grader_grad_check(value):
    print(value)
    assert(np.all(value <= 10**-3))
    return True

grader_grad_check(gradient_check_value)
```

```
     5.309771069360733e-08
     True
```

```
approx_gradients_list
```

```
     array([-1.16630516e-02, -1.08678955e-03,  3.55382390e-06, -1.14908083e-05,
              -2.48689958e-06, -9.03275801e-01, -9.02219428e-01, -7.04759251e-03,
              -1.09954656e+00])
```

```
original_gradients_list
```

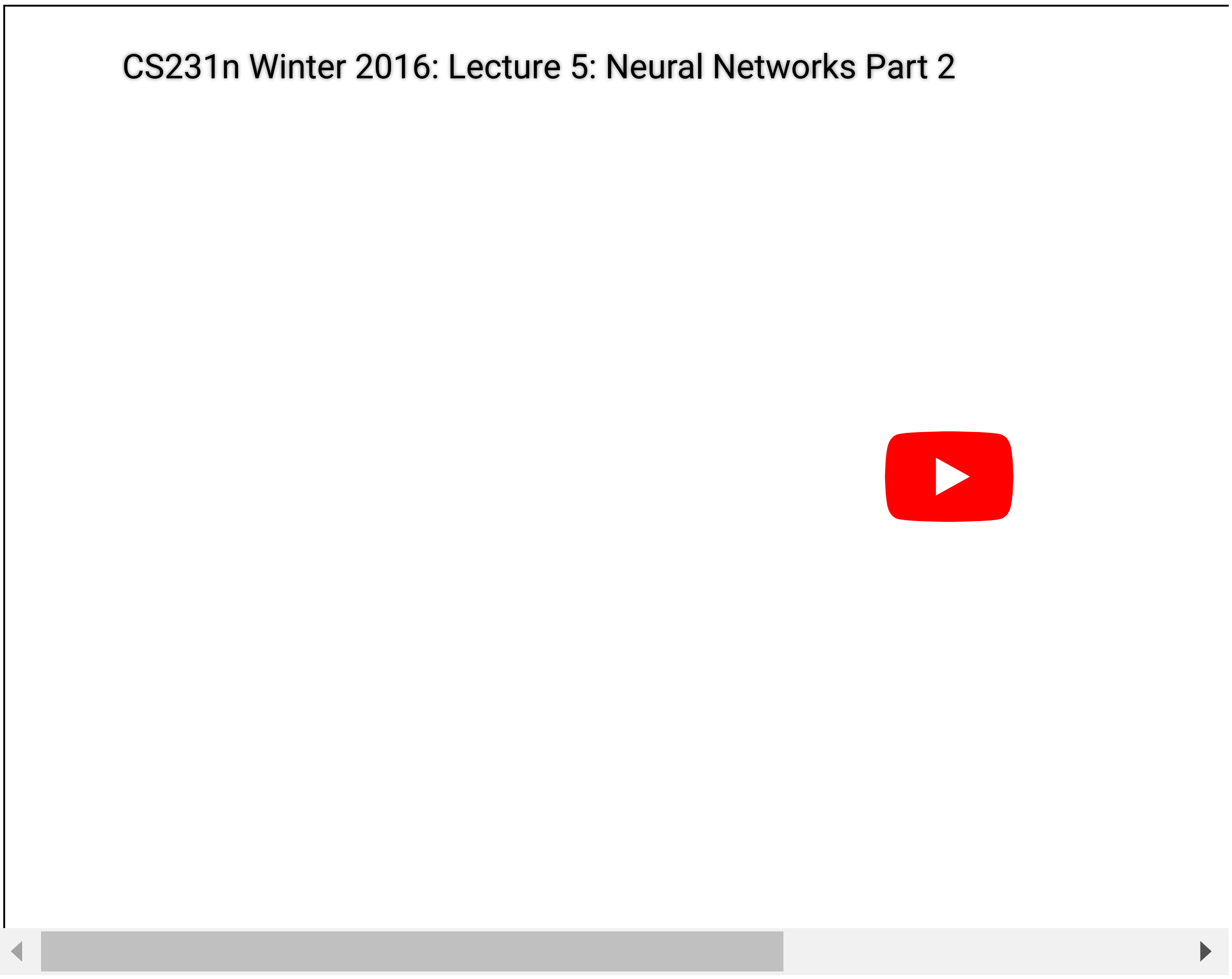
```
     array([-1.16630512e-02, -1.08681753e-03,  3.55201075e-06, -1.14905650e-05,
              -2.48592766e-06, -9.03275801e-01, -9.02219254e-01, -7.04759100e-03,
              -1.09954653e+00])
```

Task 2 : Optimizers

- As a part of this task, you will be implementing 2 optimizers(methods to update weight)
- Use the same computational graph that was mentioned above to do this task
- The weights have been initialized from normal distribution with mean=0 and std=0.01. The initialization of weights is very important otherwiswe you can face vanishing gradient and exploding gradients problem.

Check below video for reference purpose

```
from IPython.display import YouTubeVideo
YouTubeVideo('gYpoJMlgyXA',width="1000",height="500")
```



Algorithm

```
for each epoch(1-20):
    for each data point in your data:
        using the functions forward_propagation() and backword_propagation() compute
        update the weigts with help of gradients
```

Implement below tasks

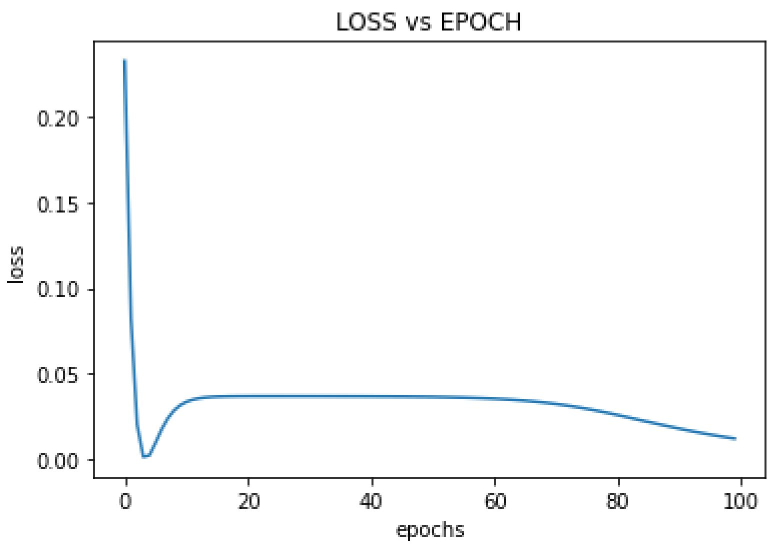
- **Task 2.1**: you will be implementing the above algorithm with **Vanilla update** of weights
- **Task 2.2**: you will be implementing the above algorithm with **Momentum update** of weights
- **Task 2.3**: you will be implementing the above algorithm with **Adam update** of weights

Note : If you get any assertion error while running grader functions, please print the variables in grader functions and check which variable is returning False .Recheck your logic for that variable .

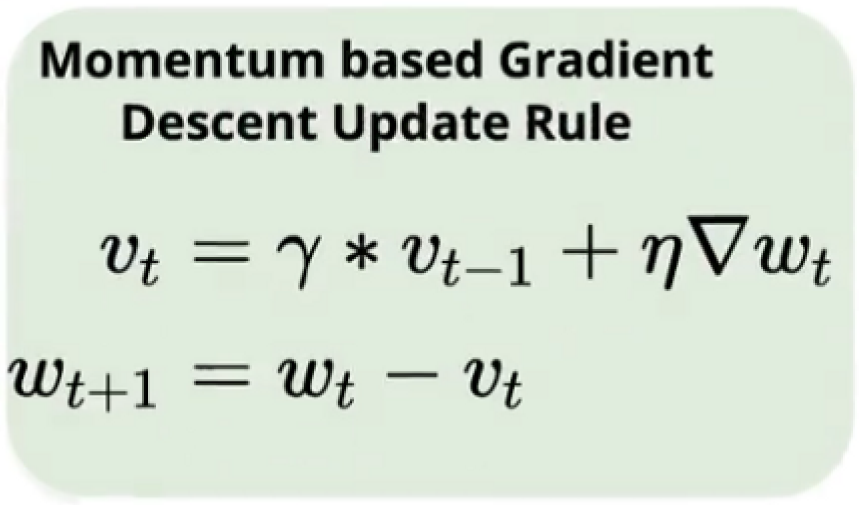
Reference: https://towardsdatascience.com/10-gradient-descent-optimisation-algorithms-86989510b5e9

2.1 Algorithm with Vanilla update of weights

```
mean=0
std=0.01
learning_rate=0.001
w = np.random.normal(mean, std, 9) # weight intialization
loss = []

for epoch in range(100):
    for j in range(len(X)):
        fp=forward_propagation(X[j],y[j],w)        # forward prop
        bp=backward_propagation(X[j],y[j],w,fp)        # backward prop to get dw values
        bp_list = list(bp.values())                # list of all dW values
        for k in range(len(bp_list)):
            w[k] = w[k] - (learning_rate * bp_list[k])      # SGD update equation
    loss.append(fp['loss'])

import matplotlib.pyplot as plt
epoch = 100
plt.xlabel('epochs')
plt.ylabel('loss')
plt.title('LOSS vs EPOCH')
plt.plot(loss)
plt.show()
```



2.2 Algorithm with Momentum update of weights



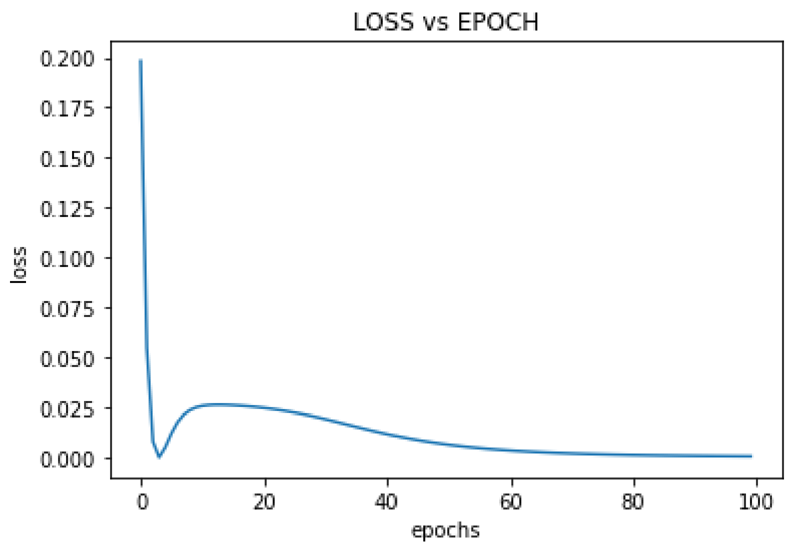
Momentum based Gradient Descent Update Rule

$$v_t = \gamma * v_{t-1} + \eta \nabla w_t$$

$$w_{t+1} = w_t - v_t$$

```
β = 0.9
mean=0
std=0.01
learning_rate=0.001
w = np.random.normal(mean, std, 9) # weight intialization
momentum_loss = []
epochs=100
```

```
momentum=0.3
update_prev = 0

for epoch in range(epochs):
    for j in range(len(X)):
        fp=forward_propagation(X[j],y[j],w)
        bp=backward_propagation(X[j],y[j],w,fp)
        bp_list = list(bp.values())
        for k in range(len(bp_list)):
            update = learning_rate * bp_list[k] + momentum * update_prev
            w[k] = w[k] - update
            update_prev = update

    momentum_loss.append(fp['loss'])


plt.xlabel('epochs')
plt.ylabel('loss')
plt.title('LOSS vs EPOCH')
plt.plot(momentum_loss)
plt.show()
```



Here Gamma referes to the momentum coefficient, eta is leaning rate and v_t is moving average of our gradients at timestep t

Double-click (or enter) to edit

2.3 Algorithm with Adam update of weights

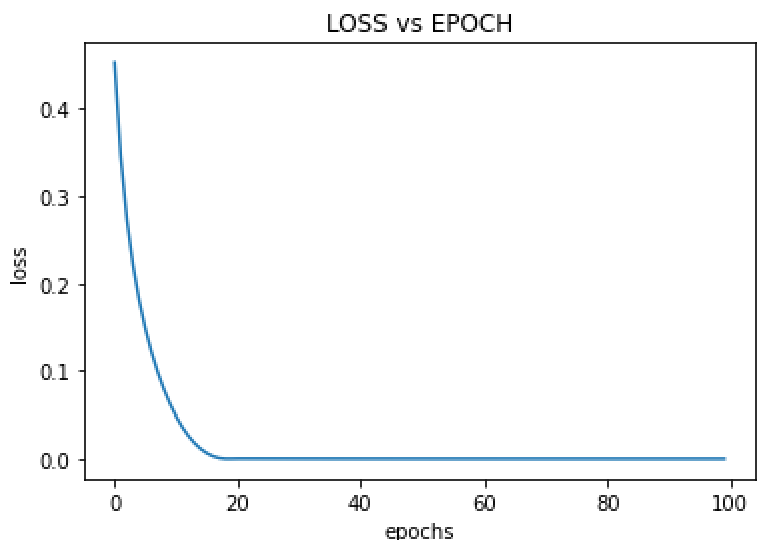
$$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * \nabla w_t$$

$$v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * (\nabla w_t)^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

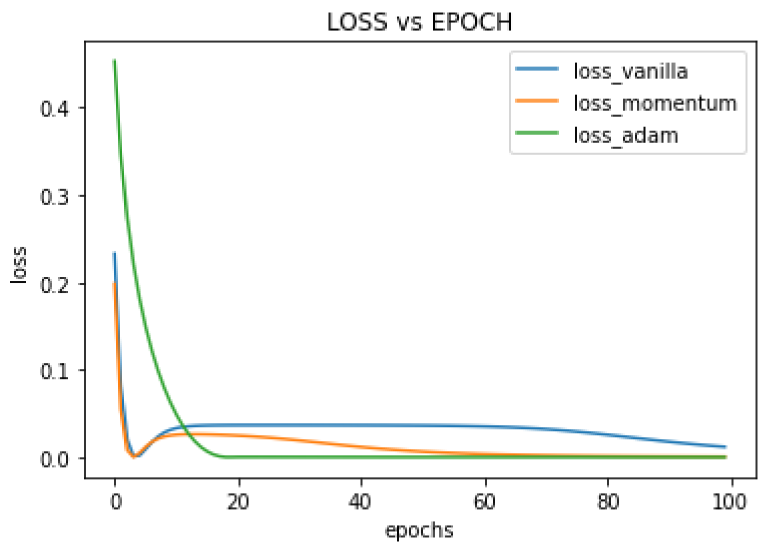
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} * \hat{m}_t$$

```
alpha = 0.001
β1 = 0.9
β2 = 0.999
epsilon = 10**-8
epochs=100
loss_adam=[]
w =[]
w = np.random.normal(mean, std, 9)
mt=0
vt=0

for epoch in range(epochs):
    for j in range(len(X)):
        fp=forward_propagation(X[j],y[j],w)
        bp=backward_propagation(X[j],y[j],w,fp)
        bp_list = list(bp.values())
        for k in range(len(bp_list)):
            mt = β1 * mt + (1 - β1) * bp_list[k]
            vt = β2 * vt + (1 - β2) * bp_list[k]**2
            mhat = mt / (1.0 - β1 **(epoch+1))
            vhat = vt / (1.0 - β2 **(epoch+1))

            w[k] = w[k] - alpha * mhat/(np.sqrt(vhat) + epsilon)       ## update equation

    loss_adam.append(fp['loss'])



plt.xlabel('epochs')
plt.ylabel('loss')
plt.title('LOSS vs EPOCH')
plt.plot(loss_adam)
plt.show()
```



Comparision plot between epochs and loss with different optimizers. Make sure that loss is conerging with increaing epochs

```
#plot the graph between loss vs epochs for all 3 optimizers.

plt.xlabel('epochs')
plt.ylabel('loss')
plt.title('LOSS vs EPOCH')
plt.plot(loss)
plt.plot(momentum_loss)
plt.plot(loss_adam)
plt.legend(['loss_vanilla','loss_momentum','loss_adam'])
plt.show()
```



You can go through the following blog to understand the implementation of other optimizers .
[Gradients update blog]