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.

Loading data

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

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
```

```
import pickle
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt

with open('/content/drive/MyDrive/Named_Assignments/Backpropagation/data.pkl',
    data = pickle.load(f)
print(data.shape)
X = data[:, :5]
y = data[:, -1]
print(X.shape, y.shape)
```

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

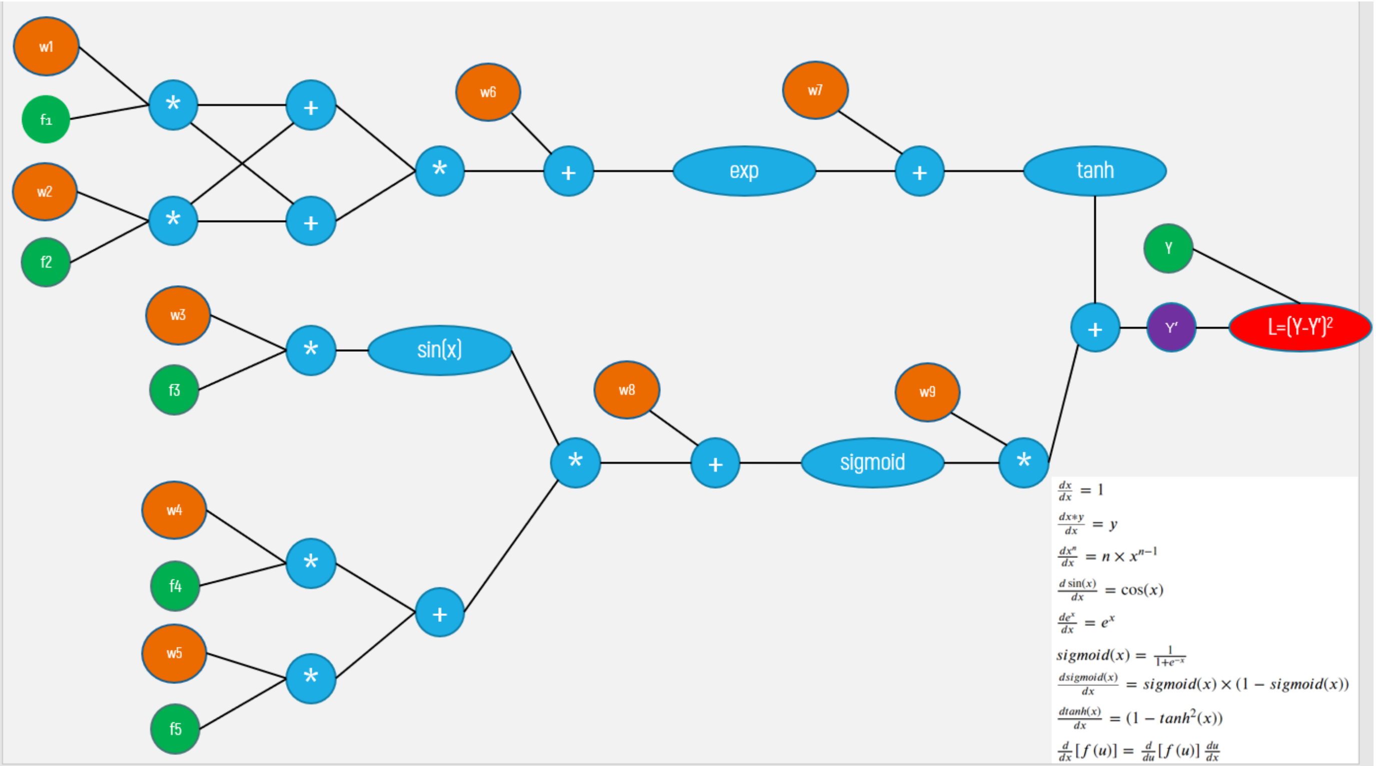
```
print(X[0,:])
print(y[0])
```

```
[-1.2879095  -0.12001342 -1.45900038 -0.66660821 -0.14421743]
1.858849127371369
```

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

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

Computational graph



- **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

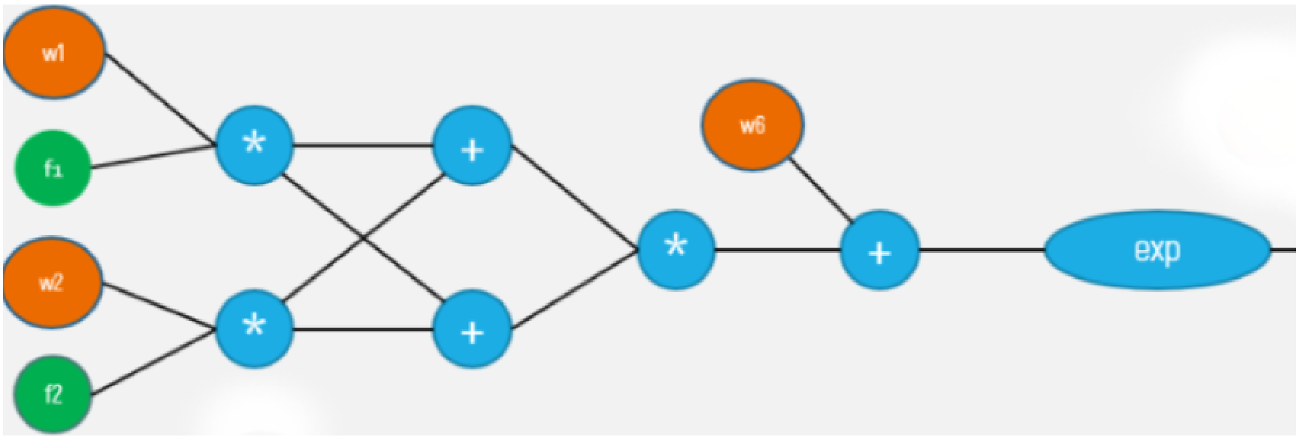
Task 1.1

Forward propagation

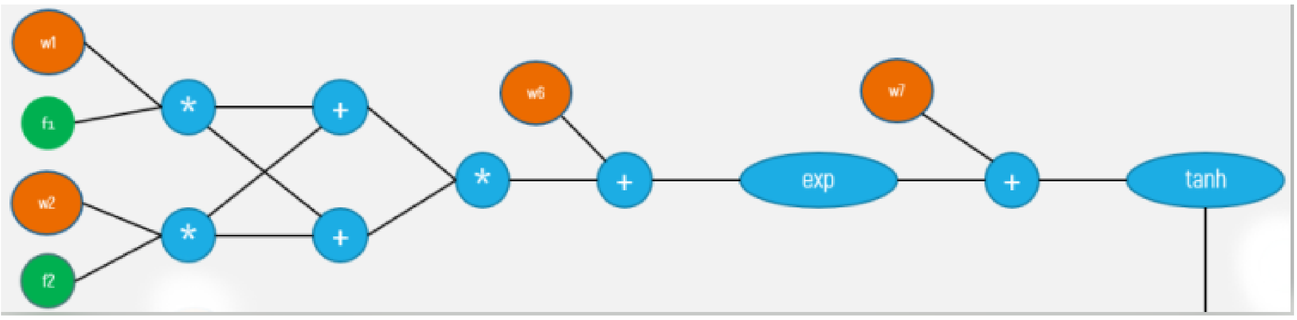
- **Forward propagation**(Write your code in def forward_propagation())

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

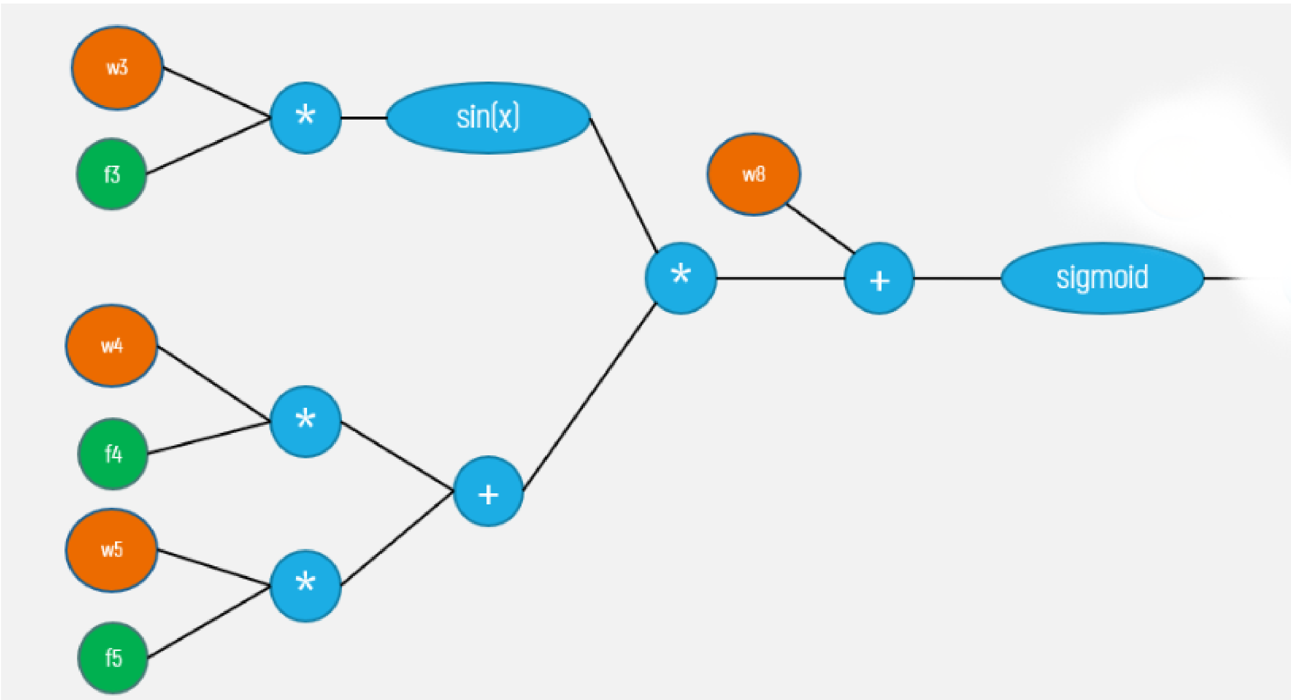
Part 1



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(-1*z)))
```

```
def grader_sigmoid(z):
  #if you have written the code correctly then the grader function will output
  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
        # y: output varible
        # W: weight array, its of length 9, W[0] corresponds to w1 in graph, W|
        # you have to return the following variables
        # exp= part1 (compute the forward propagation until exp and then store
        # tanh =part2(compute the forward propagation until tanh and then store
        # sig = part3(compute the forward propagation until sigmoid and then st
        # 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)

        val_2 = (part_1+w[6])
        part_2 = np.tanh(val_2)

        val_3 = w[7] + (np.sin(w[2]*x[2]) * (w[3]*x[3]+w[4]*x[4]))
        part_3 = sigmoid(val_3)

        y_pred = (part_3*w[8]) + part_2

        loss = (y-y_pred)**2
        dy_pred = (-2)*(y-y_pred)
        # after computing part1,part2 and part3 compute the value of y' from th
        # write code to compute the value of L=(y-y')^2 and store it in variabl
        # 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
        # we will be using the dictionary to find values in backpropagation, yo

        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'] = dy_pred

        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

    dw1 = 2*x[0]*(w[0]*x[0]+w[1]*x[1])*forward_dict['dy_pred']*(1-np.power(forw
    dw2 = 2*x[1]*(w[0]*x[0]+w[1]*x[1])*forward_dict['dy_pred']*(1-np.power(forw
    dw3 = x[2]*w[8]*np.cos(w[2]*x[2])*(w[3]*x[3]+w[4]*x[4])*forward_dict['dy_p
    dw4 = x[3]*np.sin(w[2]*x[2])*forward_dict['dy_pred']*forward_dict['sigmoid
    dw5 = x[4]*np.sin(w[2]*x[2])*forward_dict['dy_pred']*forward_dict['sigmoid
    dw6 = forward_dict['dy_pred']*(1-np.power(forward_dict['tanh'],2))*forward_
    dw7 = forward_dict['dy_pred']*(1-np.power(forward_dict['tanh'],2))
    dw8 = w[8]*forward_dict['dy_pred']*forward_dict['sigmoid']*(1-forward_dict[
    dw9 = forward_dict['dy_pred']*forward_dict['sigmoid']

    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

    #store the variables dw1,dw2 etc. in a dict as backward_dict['dw1']= dw1,ba

```

```
    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)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-167-25aea7b57884> in <module>()
     14 forward_dict=forward_propagation(X[0],y[0],w)
     15 backward_dict=backward_propagation(X[0],y[0],w,forward_dict)
---> 16 grader_backprop(backward_dict)

<ipython-input-167-25aea7b57884> in grader_backprop(data)
      9     dw8=(np.round(data['dw8'],6)==-0.048063)
     10     dw9=(np.round(data['dw9'],6)==-1.018104)
---> 11     assert(dw1 and dw2 and dw3 and dw4 and dw5 and dw6 and dw7 and
dw8 and dw9)
     12     return True
     13 w=np.ones(9)*0.1

AssertionError:
```

SEARCH STACK OVERFLOW

I got the small differnce in dw4 and dw5 gradients. Iam attaching the image here [image]

```
print(np.round(list(backward_dict.values()),6))
```

```
[-0.229733 -0.021408 -0.005625 -0.046579 -0.010077 -0.633475 -0.561942
 -0.048063 -1.018104]
```

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

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

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
        # subtract a small value to weight wi, and then find the values of L with
        # 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 apro
  gradient_check formula
    return gradient_check
NOTE: you can do sanity check by checking all the return values of gradient_check
 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 corres
    # 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 sur
    #write your code here and append the approximate gradient value for each w
    for i in range(len(w)):
        w_pos = np.array(w)
        w_neg = np.array(w)
        w_pos[i] = w[i] + eps
        w_neg[i] = w[i] - eps
        Loss_pos = forward_propagation(x,y,w_pos)
        Loss_neg = forward_propagation(x,y,w_neg)
        approx_gradients_list.append((Loss_pos['loss'] - Loss_neg['loss']) / (

    original_gradients_list=np.array(original_gradients_list)
    approx_gradients_list=np.array(approx_gradients_list)
    gradient_check_value = (np.linalg.norm(original_gradients_list - approx_gr

    return gradient_check_value

```

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

w=[ 0.00271756,  0.01260512,  0.00167639, -0.00207756,  0.00720768,
   0.00114524,  0.00684168,  0.02242521,  0.01296444]

eps=10**-7
value= gradient_checking(X[0],y[0],w,eps)
grader_grad_check(value)
```

```
0.00026560337436839175
True
```

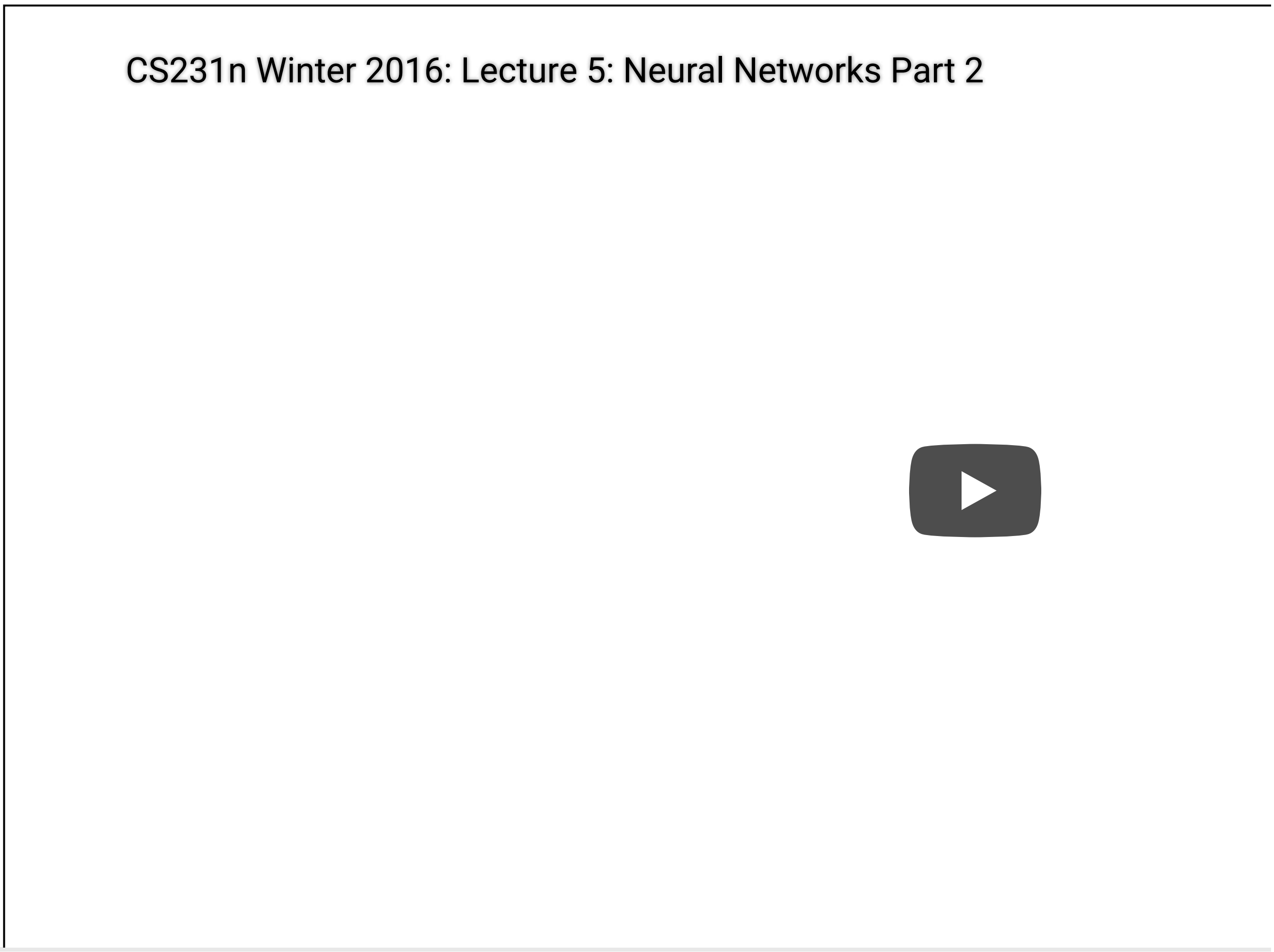
Task 2 : Optimizers

- 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()
        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 .

2.1 Algorithm with Vanilla update of weights

```
no_epochs = 20
mean=0
std=0.01
learning_rate = 0.0075
w = np.random.normal(mean,std, (9))
train_loss_vanilla = []
for i in range(no_epochs):
    for j in range(len(X)):
        x = X[j,:]
        y_actual = y[j]
        forward_dict=forward_propagation(x,y_actual,w)
        backward_dict=backward_propagation(x,y_actual,w,forward_dict)
        dw = np.array(list(backward_dict.values()))
        w = w - learning_rate * dw
    train_loss_vanilla.append(forward_dict['loss'])
```

2.2 Algorithm with Momentum update of weights

Momentum based Gradient Descent Update Rule

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

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

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

```
no_epochs = 20
mean=0
std=0.01
learning_rate = 0.0075
```

```

w = np.random.normal(mean,std, (9))
vt = np.ones(9)
gamma = 0.001
train_loss_momentum = []
for i in range(no_epochs):
    for j in range(len(X)):
        x = X[j,:]
        y_actual = y[j]
        forward_dict=forward_propagation(x,y_actual,w)
        backward_dict=backward_propagation(x,y_actual,w,forward_dict)
        dw = np.array(list(backward_dict.values()))
        vt = gamma*vt + learning_rate* dw
        w = w - vt
    train_loss_momentum.append(forward_dict['loss'])
```

2.3 Algorithm with Adam update of weights

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

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

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

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

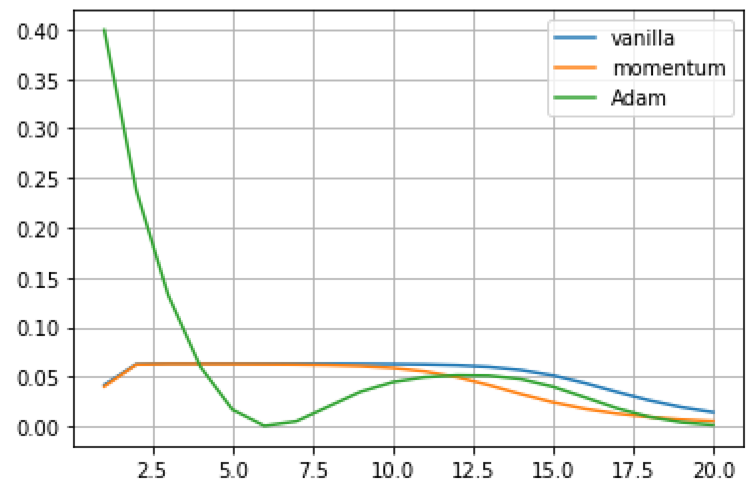
```
no_epochs = 20
mean=0
std=0.01
learning_rate = 0.001
beta1 = 0.9
beta2 = 0.999
eps = 0.00001
w = np.random.normal(mean,std, (9))
vt = np.ones(9)
mt = np.ones(9)
gamma = 0.001
train_loss_adam = []
for i in range(no_epochs):
    for j in range(len(X)):
        x = X[j,:]
```

```
        y_actual = y[j]
        forward_dict=forward_propagation(x,y_actual,w)
        backward_dict=backward_propagation(x,y_actual,w,forward_dict)
        dw = np.array(list(backward_dict.values()))
        mt = beta1*mt + ((1-beta1)*dw)
        vt = beta2*vt + ((1-beta2)*(dw**2))
        mt_hat = mt/(1-(beta1**(j+1)))
        vt_hat = vt/(1-(beta2**(j+1)))
        w += -((learning_rate*mt_hat)/(np.sqrt(vt_hat)+eps))
    train_loss_adam.append(forward_dict['loss'])
```

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.
import matplotlib.pyplot as plt
epochs = np.arange(1,21)
print(epochs)
fig,ax = plt.subplots()
ax.plot(epochs, train_loss_vanilla, label="vanilla")
ax.plot(epochs, train_loss_momentum, label="momentum")
ax.plot(epochs, train_loss_adam, label="Adam")
plt.legend()
plt.grid()
fig.canvas.draw()
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
```



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

0s completed at 12:17