

## ▼ 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 implementation.

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 files
```

```
uploaded = files.upload()
```

data.pkl

- **data.pkl**(n/a) - 24449 bytes, last modified: 10/24/2019 - 100% done  
Saving data.pkl to data.pkl

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

```
with open('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,)
```

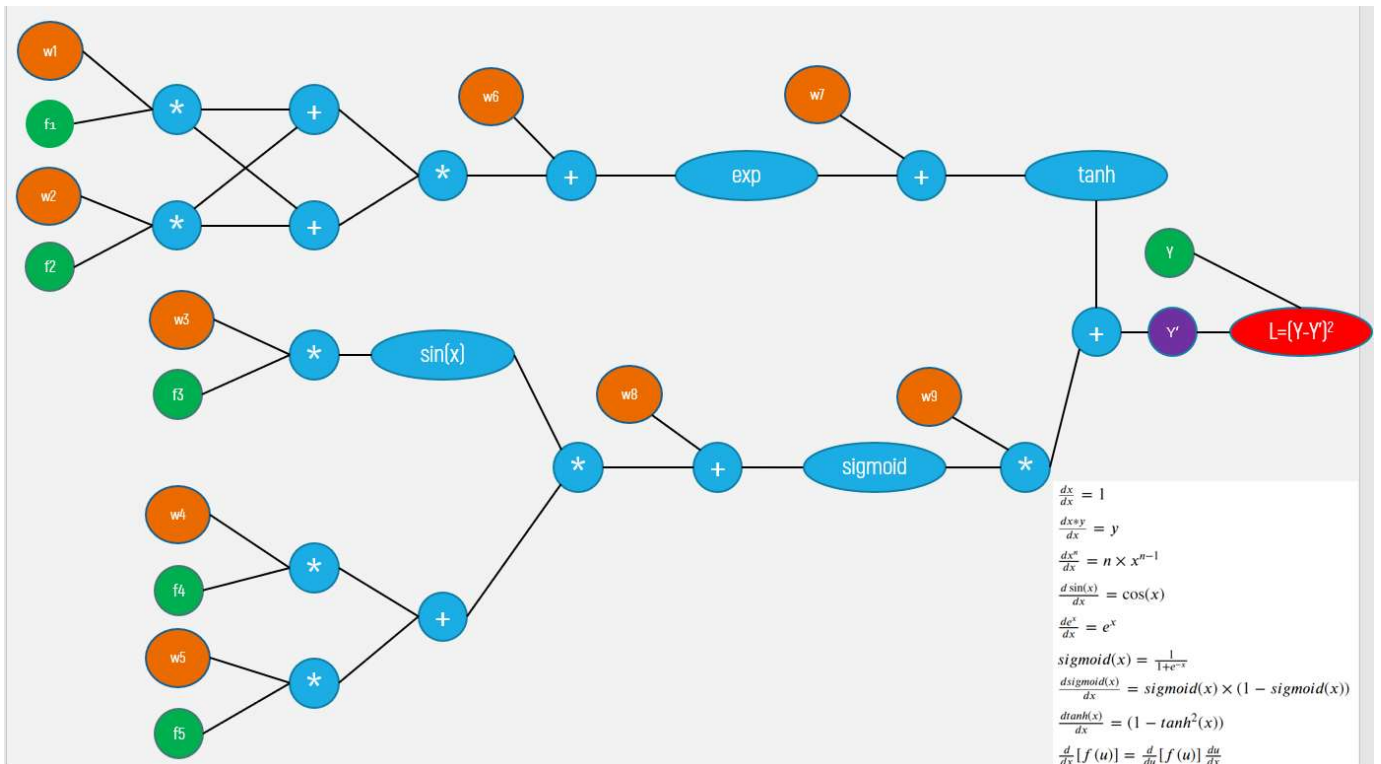
**[Check this video for better understanding of the computational graphs and back propagation](#)**

```
from IPython.display import YouTubeVideo
YouTubeVideo('i940vYb6noo',width="1000",height="500")
```

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



## ▼ 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

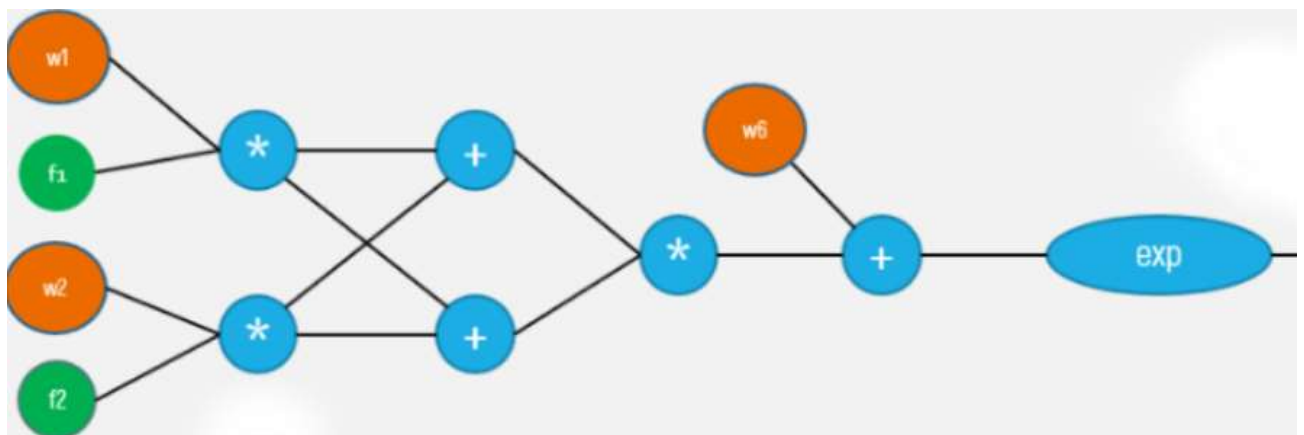
### Task 1.1

#### Forward propagation

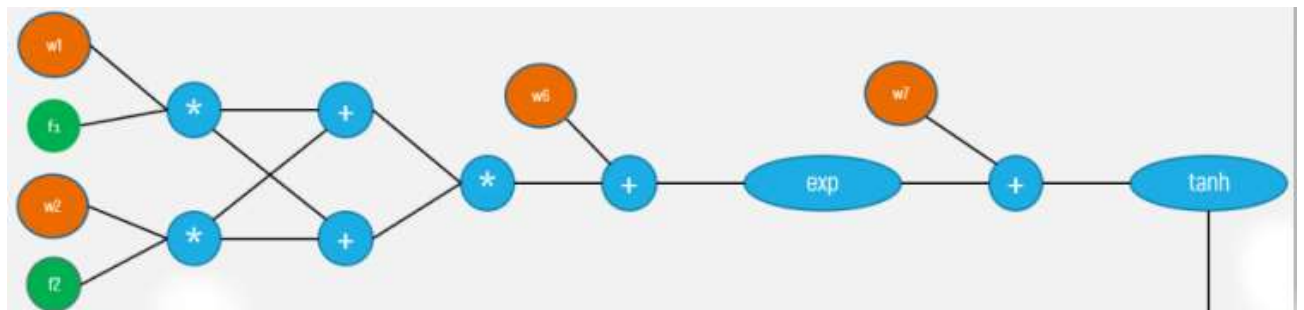
- **Forward propagation**(Write your code in `def forward_propagation()`)

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

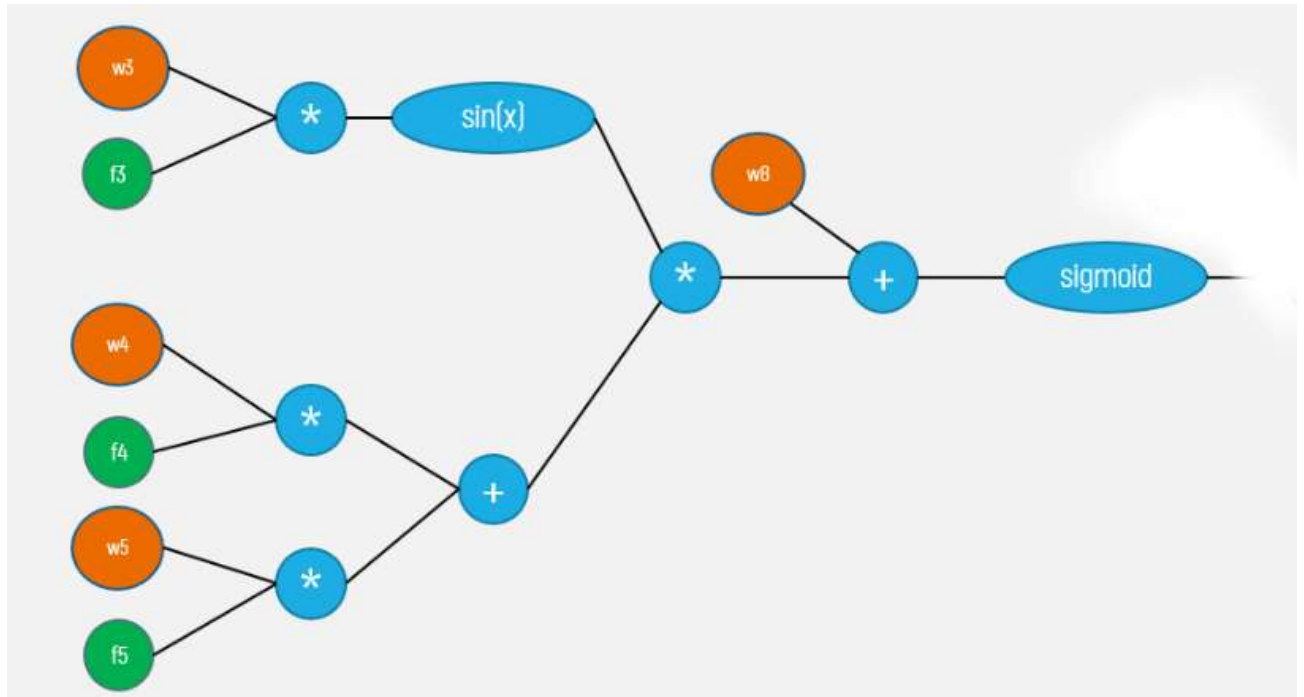
##### Part 1



##### Part 2



### Part 3



```
import math
```

```
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
    sigmoid_val = 1/(1+math.exp(-z))
    return sigmoid_val
```

```
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 variable
    # 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)
```

```

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

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

# 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

y_pred = part_3*w[8]+part_2
loss_val = (y-y_pred)**2

derivative_val = 2*(y_pred-y)

forward_dict={}
forward_dict['exp']= part_1
forward_dict['sigmoid'] = part_3
forward_dict['tanh'] = part_2
forward_dict['loss'] = loss_val
forward_dict['dy_pred'] = derivative_val

return forward_dict

```

```

def grader_forwardprop(data):
    d1 = (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(d1 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
#store the variables dw1,dw2 etc. in a dict as backward_dict['dw1']= dw1,backward_dict
```

```
dw1 = (forward_dict['dy_pred']*np.exp((w[0]*x[0]+w[1]*x[1])**2+w[5]))*2*(w[0]*x[0]+w[1]
dw2 = (forward_dict['dy_pred']*np.exp((w[0]*x[0]+w[1]*x[1])**2+w[5]))*2*(w[0]*x[0]+w[1]
dw3 = forward_dict['dy_pred']*w[8]*(np.exp((w[3]*x[3]+w[4]*x[4])*np.sin(w[2]*x[2]))+w[7]
dw4 = forward_dict['dy_pred']*w[8]*(np.exp((w[3]*x[3]+w[4]*x[4])*np.sin(w[2]*x[2]))+w[7]
dw5 = forward_dict['dy_pred']*w[8]*(np.exp((w[3]*x[3]+w[4]*x[4])*np.sin(w[2]*x[2]))+w[7]
dw6 = forward_dict['dy_pred']*(1-(np.tanh(np.exp((w[0]*x[0]+w[1]*x[1])**2+w[5]))+w[6]))
dw7 = forward_dict['dy_pred']*(1-(np.tanh(np.exp((w[0]*x[0]+w[1]*x[1])**2+w[5]))+w[6]))
dw8 = forward_dict['dy_pred']*w[8]*(np.exp((w[3]*x[3]+w[4]*x[4])*np.sin(w[2]*x[2]))+w[7]
dw9 = forward_dict['dy_pred']*(1/(1+(np.exp(-1*((w[3]*x[3]+w[4]*x[4])*np.sin(w[2]*x[2]
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

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

- 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:  $f(w_1, w_2, x_1, x_2) = w_1^2 \cdot x_1 + w_2 \cdot x_2$

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

$$\begin{aligned} \frac{df}{dw_1} = dw_1 &= 2 \cdot w_1 \cdot x_1 \\ &= 2 \cdot 1 \cdot 3 \\ &= 6 \end{aligned}$$

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

$$\begin{aligned} dw_1^{approx} &= \frac{f(w_1 + \epsilon, w_2, x_1, x_2) - f(w_1 - \epsilon, w_2, x_1, x_2)}{2\epsilon} \\ &= \frac{((1 + 0.0001)^2 \cdot 3 + 2 \cdot 4) - ((1 - 0.0001)^2 \cdot 3 + 2 \cdot 4)}{2\epsilon} \\ &= \frac{(1.00020001 \cdot 3 + 2 \cdot 4) - (0.99980001 \cdot 3 + 2 \cdot 4)}{2 \cdot 0.0001} \\ &= \frac{(11.00060003) - (10.99940003)}{0.0002} \\ &= 5.99999999999 \end{aligned}$$

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:  $gradient\_check = \frac{(6 - 5.9999999999994898)}{(6 + 5.9999999999994898)} = 4.2514140356330737e^{-13}$

you can mathamatically derive the same thing like this

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

## ▼ 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**



```

w = np.random.rand(9)

def gradient_checking(x,y,w,eps):
    # compute the dict value using forward_propagation()
    # compute the actual gradients of W using backward_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,wi in enumerate(w):
        eps = 0.0001
        add=wi+eps
        w_add = w.copy()
        w_add[i] = add
        forward_dict_add=forward_propagation(x,y,w_add)
        sub=wi-eps
        w_sub = w.copy()
        w_sub[i] = sub
        forward_dict_sub=forward_propagation(x,y,w_sub)
        approx_grad = (forward_dict_add['loss']-forward_dict_sub['loss'])/(2*eps)
        approx_gradients_list.append(approx_grad)

    #performing gradient check operation
    original_gradients_list=np.array(original_gradients_list)
    approx_gradients_list=np.array(approx_gradients_list)
    gradient_check_value =(original_gradients_list-approx_gradients_list)/(original_gradie

    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)

[ 7.50037812e-09 -9.01698285e-10 -2.27799429e-08 -7.36355085e-09
```

```
3.51241761e-07 1.22687786e-09 -2.71917712e-09 5.83760696e-10  
-6.12086145e-13]  
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 otherwise you can face vanishing gradient and exploding gradients problem.

**Check below video for reference purpose**

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

### Algorithm

```
for each epoch(1-20):  
    for each data point in your data:  
        using the functions forward_propagation() and backward_propagation() compute  
        update the weights 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

```
w = np.random.normal(0,0.01,9)
learning_rate = 0.001

def pred():
    y_pred_lst = []
    for xi in list(X):
        val_1= (w[0]*xi[0]+w[1]*xi[1]) * (w[0]*xi[0]+w[1]*xi[1]) + w[5]
        part_1 = np.exp(val_1)
        val_2 = part_1+w[6]
        part_2 = np.tanh(val_2)
        val_3 = (np.sin(w[2]*xi[2]))*(w[3]*xi[3]+w[4]*xi[4])+w[7]
        part_3 = sigmoid(val_3)
        y_pred = part_3*w[8]+part_2
        y_pred_lst.append(y_pred)
    return y_pred_lst

from sklearn.metrics import mean_squared_error

Vanilla_loss_lst = []
for epochs in tqdm(range(1,20)):
    y_predict_lst = []
    for xi,yi in list(zip(X,y)):
        forward_dict=forward_propagation(xi,yi,w)
        backward_dict=backward_propagation(xi,yi,w,forward_dict)
        for i, grad in enumerate(backward_dict.values()):
            w[i] = w[i]-learning_rate*grad
    y_predict_lst = pred()
    loss_val = mean_squared_error(y.tolist(),y_predict_lst)
    Vanilla_loss_lst.append(loss_val)
print(Vanilla_loss_lst)
```

100%|██████████| 19/19 [00:01<00:00, 15.00it/s][0.45872249672522203, 0.23870945978416]



## ▼ 2.2 Algorithm with Momentum update of weights

## Momentum based Gradient Descent Update Rule

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

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

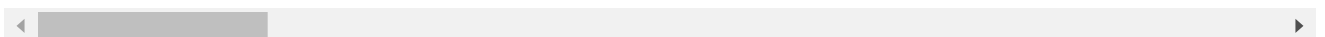
Here Gamma refers to the momentum coefficient, eta is leaning rate and v\_t is moving average of our gradients at timestep t

Double-click (or enter) to edit

```
momentum_loss_lst = []
gamma = 0.001

for epochs in tqdm(range(1,20)):
    y_predict_lst = []
    for xi,yi in list(zip(X,y)):
        forward_dict=forward_propagation(xi,yi,w)
        backward_dict=backward_propagation(xi,yi,w,forward_dict)
        for i, grad in enumerate(backward_dict.values()):
            w[i] = w[i]-(sum(backward_dict.values())/len(backward_dict.values()))*gamma+learning_
    y_predict_lst = pred()
    loss_val = mean_squared_error(y.tolist(),y_predict_lst)
    momentum_loss_lst.append(loss_val)
print(momentum_loss_lst)
```

100%|██████████| 19/19 [00:01<00:00, 12.70it/s][0.34650247717147986, 0.13882064997948]



### ▼ 2.3 Algorithm with Adam update of weights

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

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

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

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

```
adam_loss_lst = []
beta1=0.001
beta2=0.01
eps=1e-6

for epochs in tqdm(range(1,20)):
    y_predict_lst = []
    for xi,yi in list(zip(X,y)):
        forward_dict=forward_propagation(xi,yi,w)
        backward_dict=backward_propagation(xi,yi,w,forward_dict)
        for i, grad in enumerate(backward_dict.values()):
            mean_w = beta1*np.mean(w) + (1-beta1)*grad
            var_w = beta2*np.var(w) + (1-beta2)*(grad**2)
            new_w_mean = mean_w/(1-np.power(beta1, epochs))
            new_w_var = var_w/(1-np.power(beta2, epochs))
            w[i] = w[i]-((learning_rate/((new_w_var+eps)**0.5))*new_w_mean)
    y_predict_lst = pred()
    loss_val = mean_squared_error(y.tolist(),y_predict_lst)
    adam_loss_lst.append(loss_val)
print(adam_loss_lst)
```

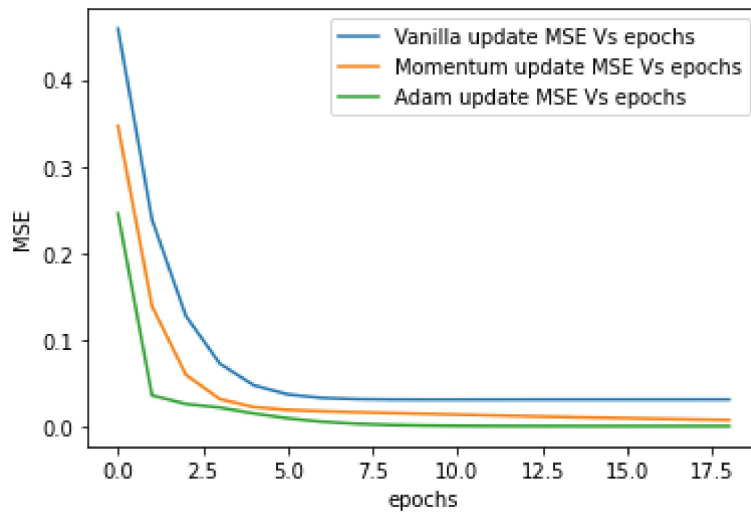
100%|██████████| 19/19 [00:04<00:00, 4.00it/s][0.2456027159079854, 0.03608714456240]

Comparison plot between epochs and loss with different optimizers. Make sure that loss is converging with increasing epochs

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

```
plt.plot(list(range(epochs)),Vanilla loss lst,label="Vanilla update MSE Vs epochs")
```

```
plt.plot(list(range(epochs)),momentum_loss_lst,label="Momentum update MSE Vs epochs")  
plt.plot(list(range(epochs)),adam_loss_lst,label="Adam update MSE Vs epochs")  
plt.xlabel("epochs")  
plt.ylabel("MSE")  
plt.legend()  
plt.show()
```



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