Backpropagation

In this notebook, we will implement Backpropagation from scratch for a given complex function...

Loading data ¶

In [92]:

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

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,)
```

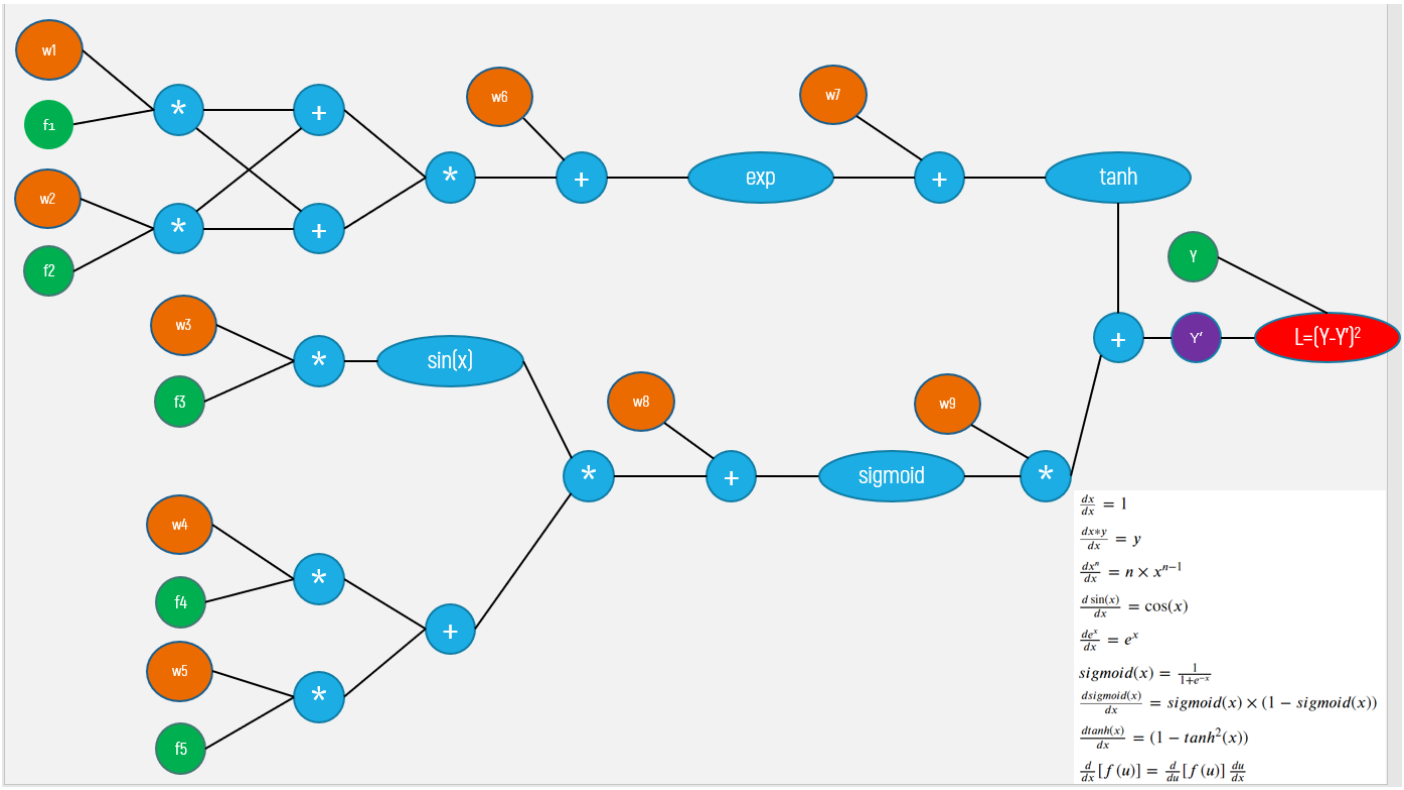
Check this video for better understanding of the computational graphs and back propagation

In [4]:

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

Out[4]:

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

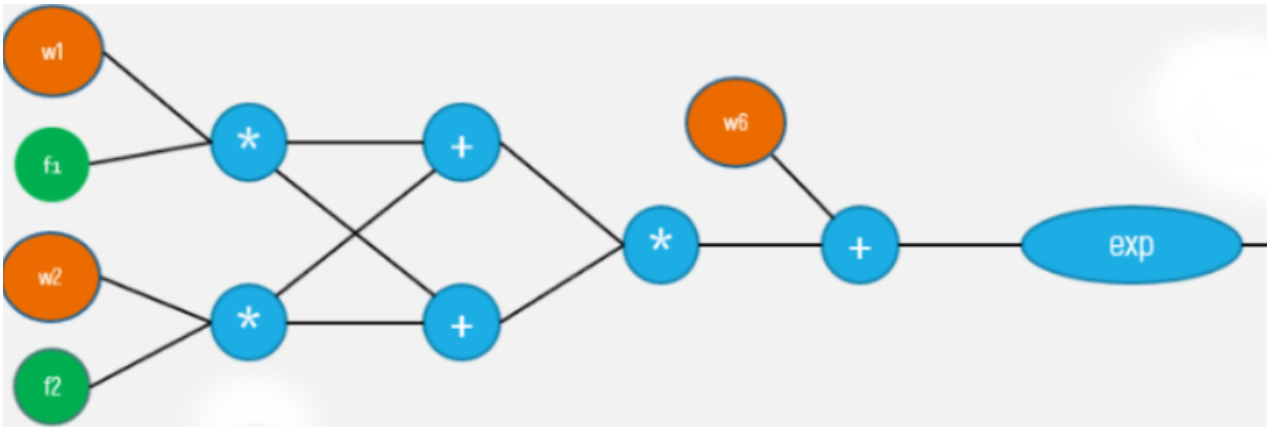
Task 1.1

Forward propagation

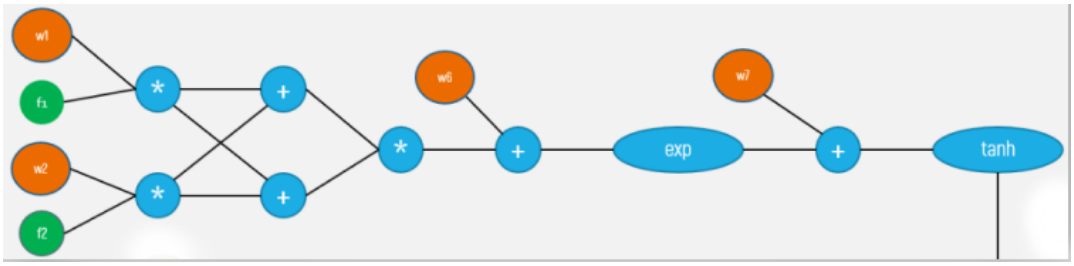
- **Forward propagation**(Write our code in def forward_propagation())

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

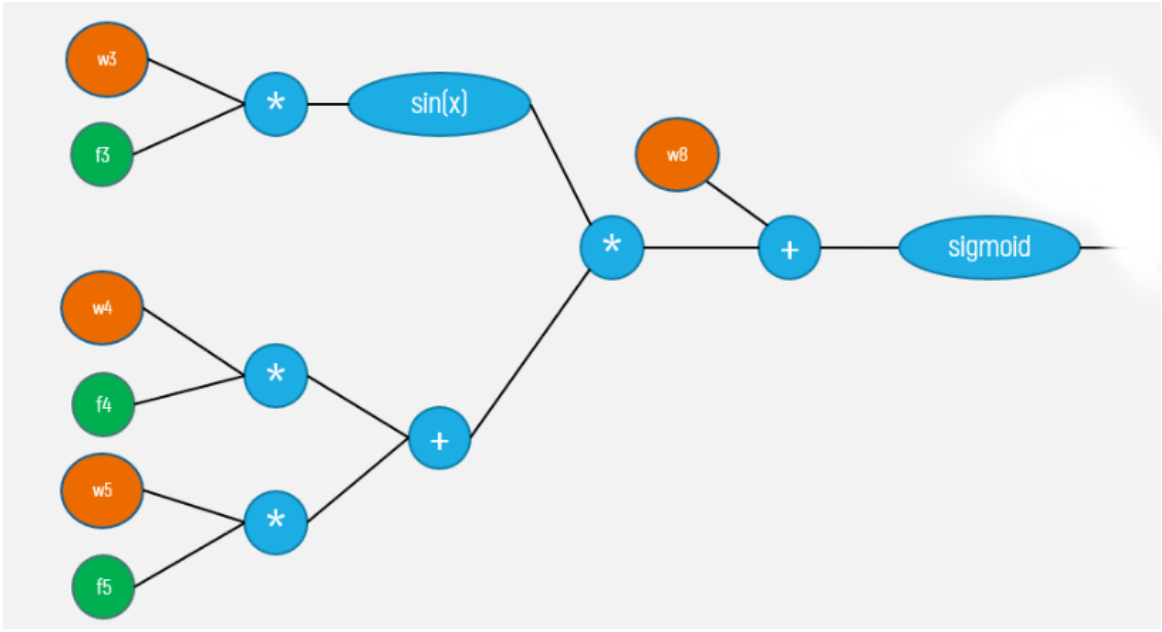
  Part 1



  Part 2



  Part 3



In [180]:

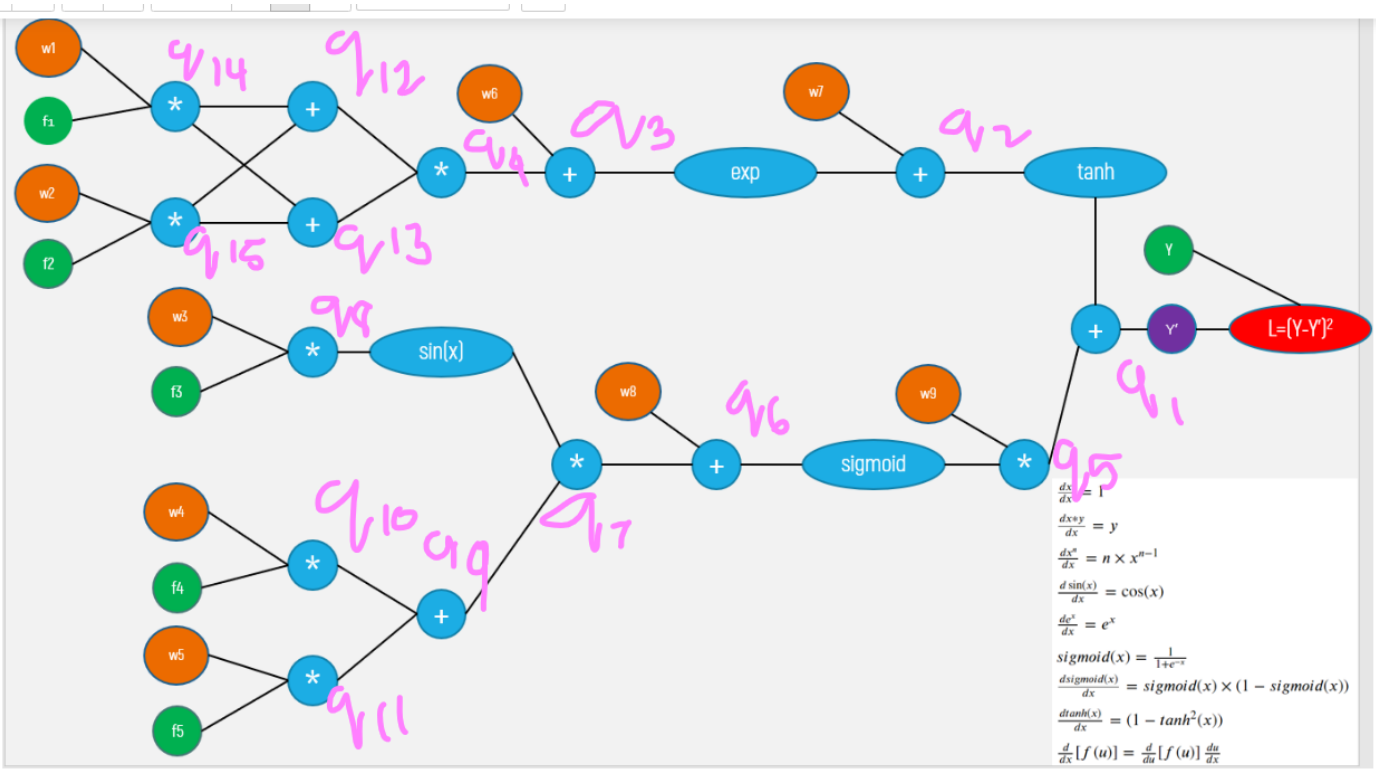
```
def sigmoid(z):
    '''In this function, we will compute the sigmoid(z)'''
    # we can use this function in forward and backward propagation
    # write the code to compute the sigmoid value of z and return that value
    return 1 / (1 + np.exp(-z))
```

In [182]:

```
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] corresponds to w2 in graph,..., W[8] correspond
        # you have to return the following variables
        # exp= part1 (compute the forward propagation until exp and then store the values in exp)
        # tanh =part2(compute the forward propagation until tanh and then store the values in tanh)
        # sig = part3(compute the forward propagation until sigmoid and then store the values in sig)
        # 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)          #exponential

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

        val_2=(np.sin(w[2]*x[2]))*(w[3]*x[3]+w[4]*x[4]))+w[7]
        part_3=sigmoid(val_2)           #sigmoid

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

        # after computing part1,part2 and part3 compute the value of y' from the main Computational graph using required equation
        # 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,tanh,sigmoid
        # we will be using the dictionary to find values in backpropagation, you can add other keys in dictionary as well

        forward_dict={}
        forward_dict['y_pred']= y_
        forward_dict['exp']= part_1
        forward_dict['sigmoid'] =part_3
        forward_dict['tanh'] = part_2
        forward_dict['loss'] =(y-y_)**2
        forward_dict['dy_pred'] = -2*(y-y_)    #derivation of loss with respect to y_pred

        return forward_dict
```

Task 1.2

Backward propagation



In [184]:

```
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
    der={}

    dw1=forward_dict['dy_pred']*(1-(math.pow(forward_dict['tanh'],2)))*forward_dict["exp"]*2*((w[0]*x[0])+(w[1]*x[1]))*x[0]
    dw2=forward_dict['dy_pred']*(1-(math.pow(forward_dict['tanh'],2)))*forward_dict["exp"]*2*((w[0]*x[0])+(w[1]*x[1]))*x[1]
    dw3=forward_dict['dy_pred']*(forward_dict['sigmoid']*(1-forward_dict['sigmoid']))*w[8]*((w[3]*x[3])+(w[4]*x[4]))*math.cos(x[2
    dw4=forward_dict['dy_pred']*(forward_dict['sigmoid']*(1-forward_dict['sigmoid']))*w[8]*math.sin(x[2]*w[2])*x[3]
    dw5=forward_dict['dy_pred']*(forward_dict['sigmoid']*(1-forward_dict['sigmoid']))*w[8]*math.sin(x[2]*w[2])*x[4]
    dw6 =forward_dict['dy_pred']*(1-(math.pow(forward_dict['tanh'],2)))*forward_dict["exp"]
    dw7 = forward_dict['dy_pred']*(1-(math.pow(forward_dict['tanh'],2)))
    dw8 = forward_dict['dy_pred']*(forward_dict['sigmoid']*(1-forward_dict['sigmoid']))*w[8]
    dw9 = forward_dict['dy_pred']*forward_dict['sigmoid']

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

Task 1.3

Gradient clipping

Check this blog link (https://towardsdatascience.com/how-to-debug-a-neural-network-with-gradient-checking-41deec0357a9) 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 our 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 weights
        # subtract a small value to weight wi, and then find the values of L with the updated weights
        # 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 aproximation gradients of weights with
  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

In [186]:

```
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)

    # Store the original gradients for the given data points 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  dw1 ,second element is dw2 etc.
    # 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 update only one weight at a time
    # Write your code here and append the approximate gradient value for each weight in  approx_gradients_list
    original_w = w

    for i in range(len(w)):
        # Make a copy of the original weights for the forward pass
        w_forward = np.copy(w)

        #adding eps
        w_forward[i] += eps
        l1 = forward_propagation(x,y,w_forward)
        loss1 = l1['loss']           #getting loss value from dictionary

        #subtracting eps
        w_forward[i] = w[i] - eps
        l2 = forward_propagation(x,y,w_forward)
        loss2 = l2["loss"]           #getting loss value from dictionary
        aprox = (loss1 - loss2) / (2 * eps)
        approx_gradients_list.append(aprox)

    # Perform 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_gradients_list + approx_gradients_list)

    return gradient_check_value
```

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

In [ ]:

```
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 the gradients of weights
        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

In [188]:

```
rate=0.001
m=np.zeros(9)
mu, std = 0, 0.01 # mean and standard deviation
w = np.random.normal(mu, std, 9) # weight intialization
```

In [189]:

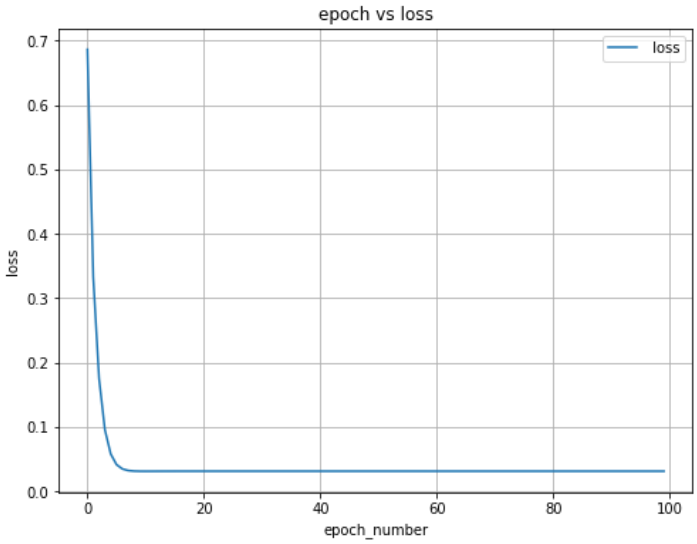
```
loss_vanilla=[]
epoc_vanilla=[]
for epoch in range(100):    #for 100 epoch
    epoc_vanilla.append(epoch)
    y_pred=[]
    for point in range(len(data)):
        forward_dict=forward_propagation(X[point], y[point], w)    #forward propagation
        y_pred.append(forward_dict['y_pred'])
        backward=backward_propagation(X[point],y[point],w,forward_dict)   #backward propagation

        for i in range(0, 9):
            w[i] = w[i] -  rate * backward["dw" + str(i+1)]

    loss=mean_squared_error(y,y_pred)
    loss_vanilla.append(loss)
```

In [190]:

```
#plot loss vs number of epoch
import matplotlib.pyplot as plt
plt.figure(figsize=(8,6))
plt.grid()
plt.plot(epoc_vanilla,loss_vanilla, label=' loss')
plt.title("epoch vs loss")
plt.xlabel("epoch_number")
plt.ylabel("loss")
plt.legend()
```

Out[190]:

```
<matplotlib.legend.Legend at 0x1baceea5910>
```



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

Type *Markdown* and LaTeX: α^2

In [191]:

```
rate=0.001
m=np.zeros(9)
b=0.9
mu, std = 0, 0.01 # mean and standard deviation
w = np.random.normal(mu, std, 9) # weight intialization
```

In [192]:

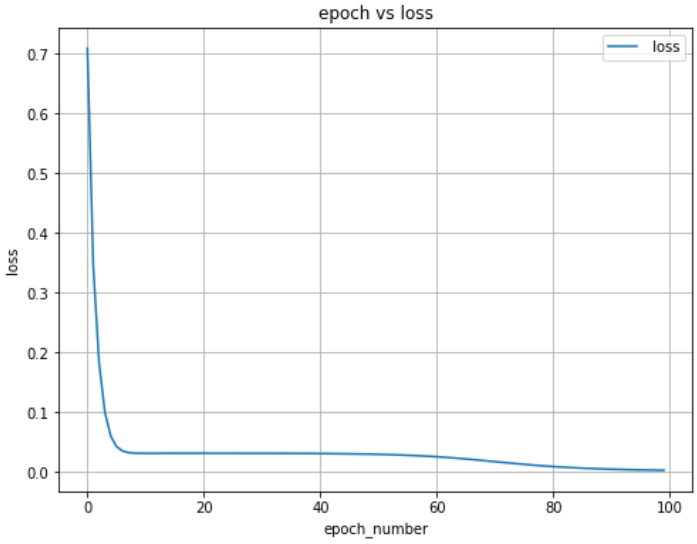
```
loss_momentum=[]
epoc_momentum=[]

for epoch in range(100):    #for 100 epoch
    epoc_momentum.append(epoch)
    y_pred=[]
    for point in range(len(data)):
        forward_dict=forward_propagation(X[point],y[point], w)    #forward propagation
        y_pred.append(forward_dict['y_pred'])
        backward=backward_propagation(X[point],y[point],w,forward_dict)   #backward propagation

        for i in range(0, 9):
            m[i] = b * m[i] + (1 - b) * backward["dw" + str(i+1)]        #momentum based update equation
            w[i] = w[i] - rate*m[i]

    loss=mean_squared_error(y,y_pred)
    loss_momentum.append(loss)
```

In [193]:

```
#plot loss vs number of epoch
import matplotlib.pyplot as plt
plt.figure(figsize=(8,6))
plt.grid()
plt.plot(epoc_momentum,loss_momentum, label=' loss')
plt.title("epoch vs loss")
plt.xlabel("epoch_number")
plt.ylabel("loss")
plt.legend()
```

Out[193]:

```
<matplotlib.legend.Legend at 0x1bacf029ac0>
```



2.3 Algorithm with Adam update of weights

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

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

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

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

In [194]:

```
rate=0.001
m=np.zeros(9)
v=np.zeros(9)
b1=0.9
b2=0.999
z=1e-8
mu, sigma = 0, 0.01 # mean and standard deviation
w = np.random.normal(mu, sigma, 9) # weight intialization
```

In [195]:

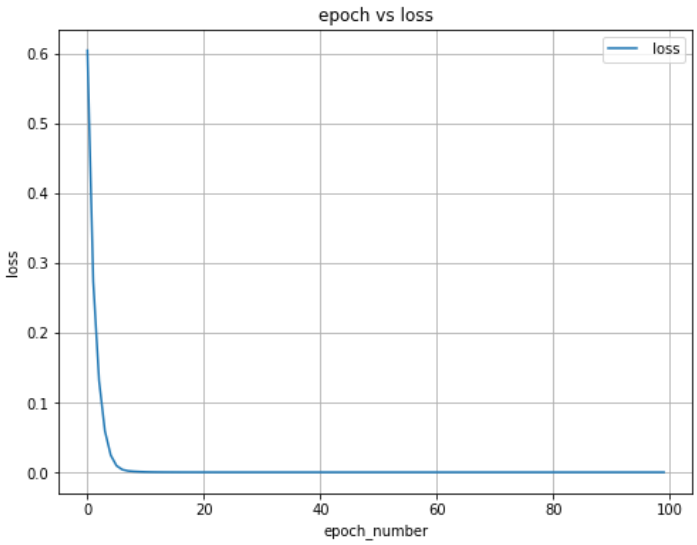
```
loss_adam=[]
epoc_adam=[]

for epoch in range(100):    #for 100 epoch
    epoc_adam.append(epoch)
    y_pred=[]
    for point in range(len(data)):
        forward_dict=forward_propagation(X[point],y[point], w)    #forward propagation
        y_pred.append(forward_dict['y_pred'])
        backward=backward_propagation(X[point],y[point],w,forward_dict)   #backward propagation

        for i in range(0, 9):                                     #adam optimizer
            m[i]=b1*m[i]+(1-b1)*backward["dw" + str(i+1)]
            mt=m[i]/(1-b1)
            v[i]=b2*v[i]+(1-b2)*math.pow(backward["dw" + str(i+1)],2)
            vt=v[i]/(1-b2)
            w[i]=w[i]-(rate/(math.sqrt(vt)+z))*mt

    loss=mean_squared_error(y,y_pred)
    loss_adam.append(loss)
```

In [197]:

```
#plot loss vs number of epoch
import matplotlib.pyplot as plt
plt.figure(figsize=(8,6))
plt.grid()
plt.plot(epoc_adam,loss_adam, label=' loss')
plt.title("epoch vs loss")
plt.xlabel("epoch_number")
plt.ylabel("loss")
plt.legend()
```

Out[197]:

```
<matplotlib.legend.Legend at 0x1bacf114ca0>
```



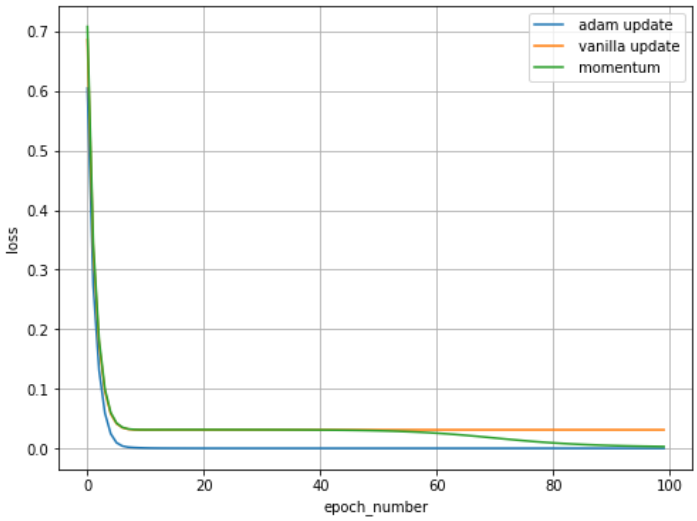
Comparision plot between epochs and loss with different optimizers. Make sure that loss is conerging with increaing epochs

In [198]:

```
#plot the graph between loss vs epochs for all 3 optimizers.
plt.figure(figsize=(8,6))
plt.grid()
plt.plot(epoc_adam,loss_adam, label=' adam update')
plt.plot(epoc_vanilla,loss_vanilla, label=' vanilla update')
plt.plot(epoc_momentum,loss_momentum, label=' momentum')
plt.xlabel("epoch_number")
plt.ylabel("loss")
plt.legend()
```

Out[198]:

```
<matplotlib.legend.Legend at 0x1bad018c4c0>
```



- We can see that using adam optimizer our weight converge to the minimim loss point quickly

In [ ]: