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

In [1]:

```
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,)
```

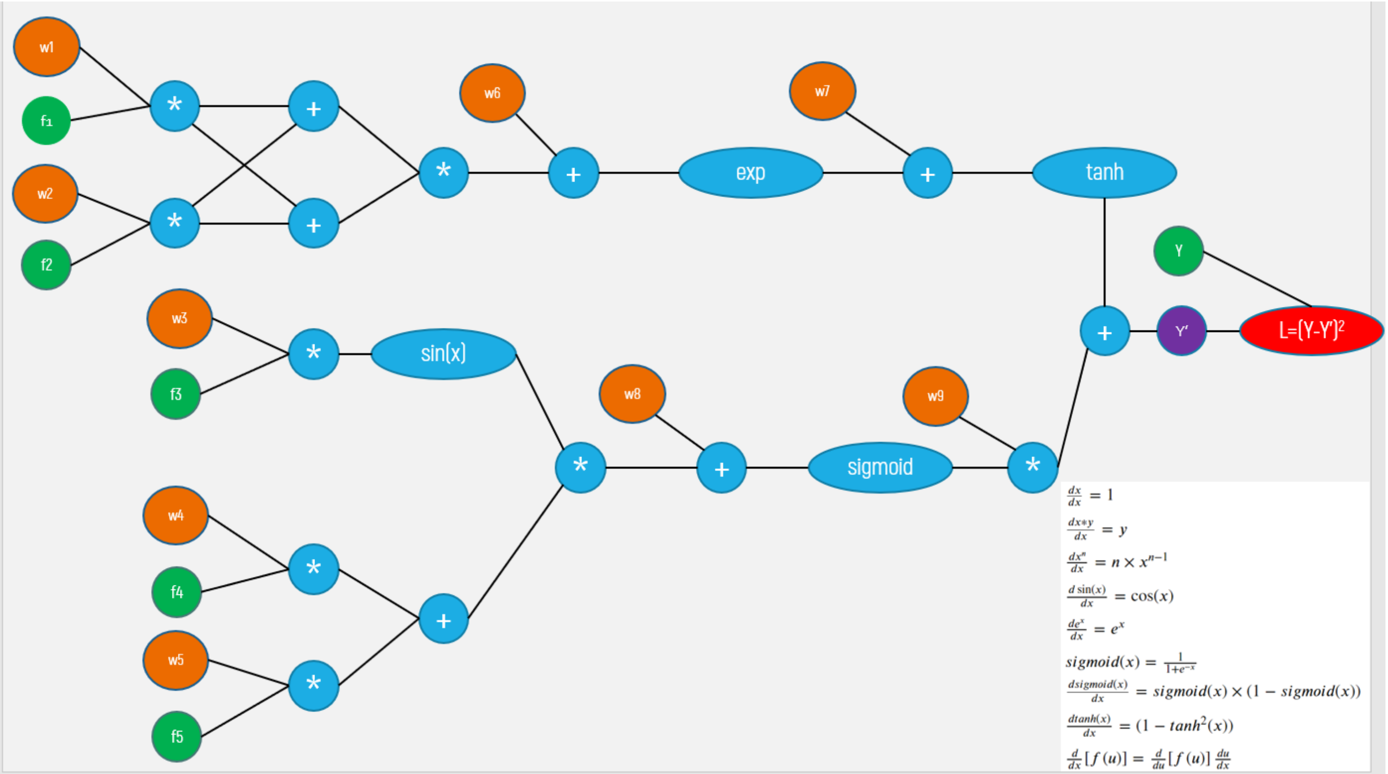
Check this video for better understanding of the computational graphs and back propagation

In [2]:

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

Out[2]:

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

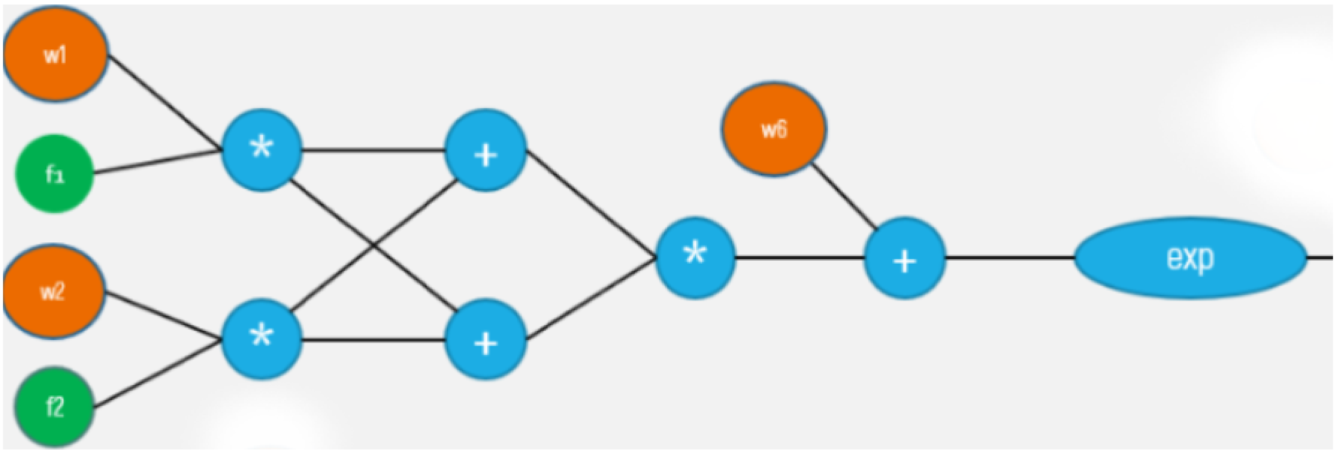
Task 1.1

Forward propagation

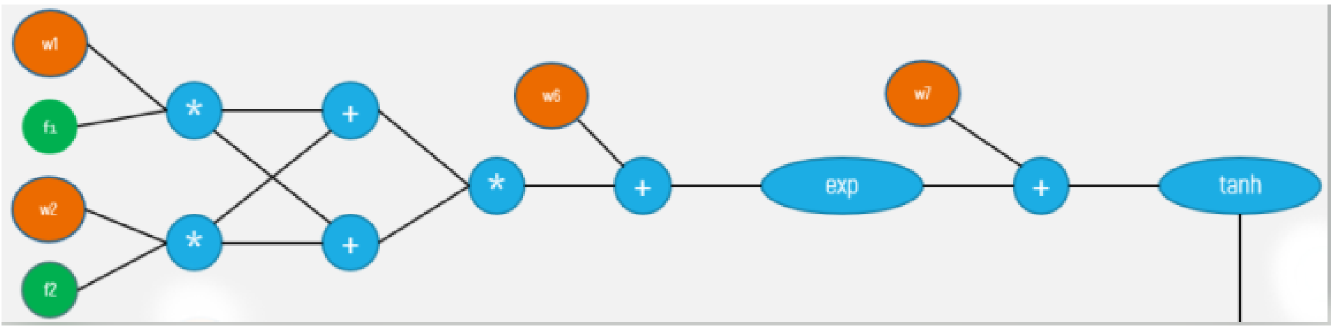
- **Forward propagation**(Write your code in def forward_propagation())

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

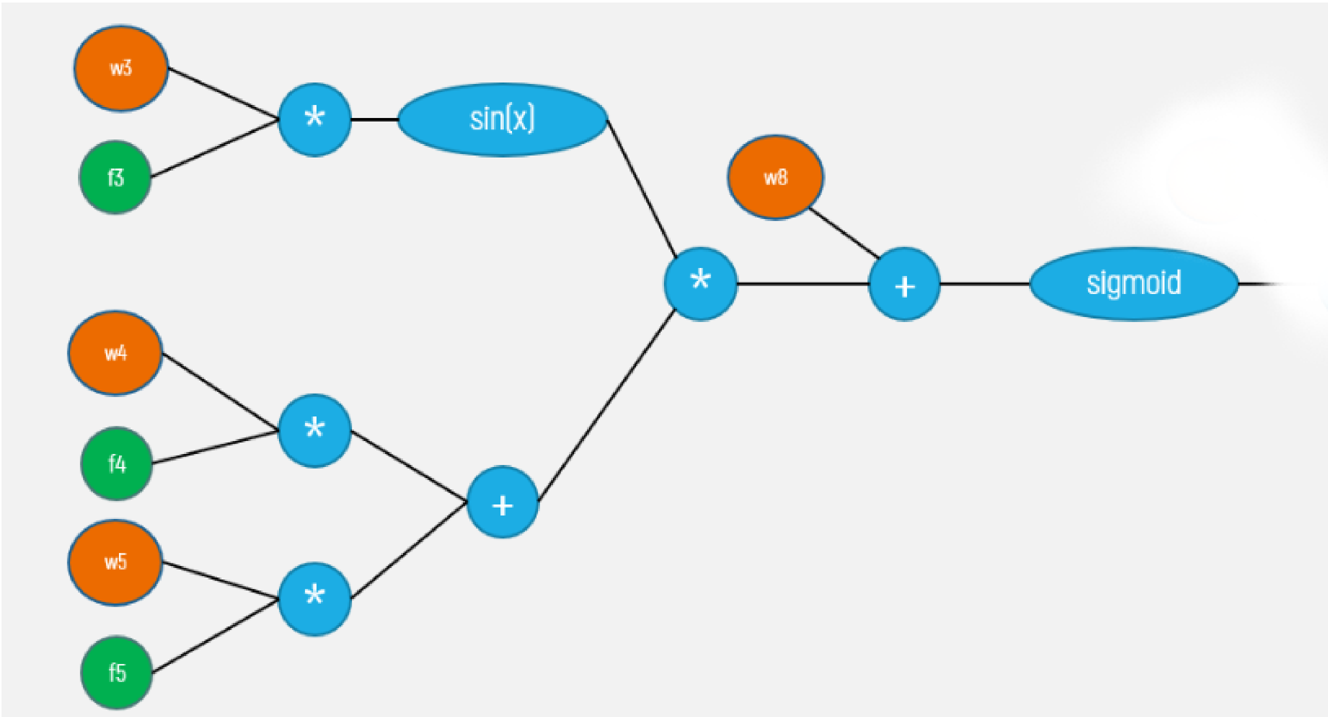
  Part 1



Part 2



Part 3



Task 1 :

In [ ]:

```

```

In [3]:

```
def sigmoid(z):
    '''In this function, we will compute the sigmoid(z)'''
    # we can use this function in forward and backward propagation

    return 1 / (1 + np.exp(-z))
```

In [4]:

```
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)
```

Out[4]:

```
True
```

In [5]:

```
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] correspon
        # you have to return the following variables
        # exp= part1 (compute the forward propagation until exp and then store the values i
        val_1 = np.exp((np.dot(w[0] , x[0]) + np.dot(w[1] , x[1]))**2 + w[5])
        # tanh =part2(compute) the forward propagation until tanh and then store the values
        part_1 = np.tanh(w[6] + val_1)
        # sig = part3(compute the forward propagation until sigmoid and then store the valu
        d = (np.sin(np.dot(x[2] , w[2])) * (np.dot(x[4] , w[4]) + np.dot(x[3] , w[3])))
        sig = sigmoid(d + w[7])
        y_pred = part_1 + np.dot(sig , w[8])
        # now compute remaining values from computional graph and get y'
        # write code to compute the value of L=(y-y')^2
        L = (y - y_pred)**2
        # compute derivative of L  w.r.to Y' and store it in dL
        dL = 2 * (y_pred - y)
        # Create a dictionary to store all the intermediate values

        forward_dict = {}
        # store L, exp,tanh,sig variables
        forward_dict['dL'] = dL
        forward_dict['L'] = L
        forward_dict['val_1'] = val_1
        forward_dict['part_1'] = part_1
        forward_dict['sigmoid'] = sig
        return forward_dict
```

In [6]:

```
def grader_forwardprop(data):
    dl = (data['dL']==-1.9285278284819143)
    loss=(data['L']==0.9298048963072919)
    part1=(data['val_1']==1.1272967040973583)
    part2=(data['part_1']==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)
```

Out[6]:

```
True
```

In [7]:

```
fp = forward_propagation(X[0], y[0], w)
```

Task 1.2

Backward propagation

In [8]:

```
import math
def backward_propagation(L,W,fp):
    '''In this function, we will compute the backward propagation '''
    # L: the loss we calculated for the current point
    # dictionary: 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

    dw9 = fp['dL'] * fp['sigmoid']
    dw8 = fp['dL'] * W[8] * fp['sigmoid'] * (1 - fp['sigmoid'])
    dw7 = fp['dL'] * (1 - (fp['part_1'])**2)
    dw6 = dw7 * fp['val_1']
    dw1 = dw6 * (2 * (np.dot(L[0] , W[0]) + np.dot(L[1] , W[1])) * L[0])
    dw2 = dw6 * (2 * (np.dot(L[0] , W[0]) + np.dot(L[1] , W[1])) * L[1])
    dw5 = dw8 * (math.sin(np.dot(L[2] , W[2]))) * L[4]
    dw4 = dw8 * (math.sin(np.dot(L[2] , W[2]))) * L[3]
    dw3 = dw8 * (np.dot(L[3] , W[3]) + np.dot(L[4] , W[4])) * (L[2] * math.cos(np.dot(L[2]


    dw = {}
    dw['dw1'] = dw1
    dw['dw2'] = dw2
    dw['dw3'] = dw3
    dw['dw4'] = dw4
    dw['dw5'] = dw5
    dw['dw6'] = dw6
    dw['dw7'] = dw7
    dw['dw8'] = dw8
    dw['dw9'] = dw9

    return dw
```

In [9]:

```
def grader_backprop(data):
    dw1=(data['dw1']==-0.22973323498702003)
    dw2=(data['dw2']==-0.021407614717752925)
    dw3=(data['dw3']==-0.005625405580266319)
    dw4=(data['dw4']==-0.004657941222712423)
    dw5=(data['dw5']==-0.0010077228498574246)
    dw6=(data['dw6']==-0.6334751873437471)
    dw7=(data['dw7']==-0.561941842854033)
    dw8=(data['dw8']==-0.04806288407316516)
    dw9=(data['dw9']==-1.0181044360187037)
    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
d1 = forward_propagation(X[0],y[0],w)
d1 = backward_propagation(X[0],w,d1)
grader_backprop(d1)
```

Out[9]:

```
True
```

Task 1.3

In [10]:

```
W = np.ones(9)*0.1
```

Gradient clipping

Check this [blog link (https://towardsdatascience.com/how-to-debug-a-neural-network-with-gradient-checking-41deec0357a9)](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 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 u
pdated 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 aprox
imation gradients of weights with
  gradient_check formula
    return gradient_check
NOTE: you can do sanity check by checking all the return values of gradient_checki
ng(),
 they have to be zero. if not you have bug in your code
```

In [11]:

```
def gradient_checking(x,y,w):
    # compute the dict value using forward_propagation()
    # compute the actual gradients of W using backword_propagation()
    d1=forward_propagation(X[0],y[0],W)
    d1=backward_propagation(X[0],W,d1)

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

    original_gradients_list=[d1['dw1'], d1['dw2'], d1['dw3'], d1['dw4'], d1['dw5'], d1['dw6
    # make sure that the order is correct i.e. first element in the list corresponds to  dw
    # you can use reverse function if the values are in reverse order
    ep = 0.0001
    approx_gradients = []
    #now we have to write code for approx gradients, here you have to make sure that you up
    #write your code here and append the approximate gradient value for each weight in  app
    for i in range(len(W)):

        W[i] = W[i] + ep
        step_up = forward_propagation(X[0],y[0],W)
        upLoss = step_up['L']

        W[i] = W[i] - (2 * ep)
        step_down = forward_propagation(X[0],y[0],W)
        downLoss = step_down['L']

        W[i] = W[i]
        approxGrad = (upLoss - downLoss) / (2 * ep)

        approx_gradients.append(approxGrad)

    #performing gradient check operation

    gradient_check_value =(np.linalg.norm(np.subtract(original_gradients_list, approx_gradi

    return gradient_check_value
gradient_checking(X, y, W)
```

Out[11]:

```
6.740504599246027e-05
```

In [12]:

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

w=[ 0.00324521,  0.0213234,  0.00142312, -0.00102312,  0.00120120,
   0.001141412,  0.00258421,  0.3212311,  0.0121212]

ep=10**-7
value= gradient_checking(X,y,w)
grader_grad_check(value)
```

```
6.739614976897128e-05
```

Out[12]:

```
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

In [13]:

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

Out[13]:

Algorithm

```
for each epoch(1-20):
    for each data point in your data:
        using the functions forward_propagation() and backword_propagation() c
ompute 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 .

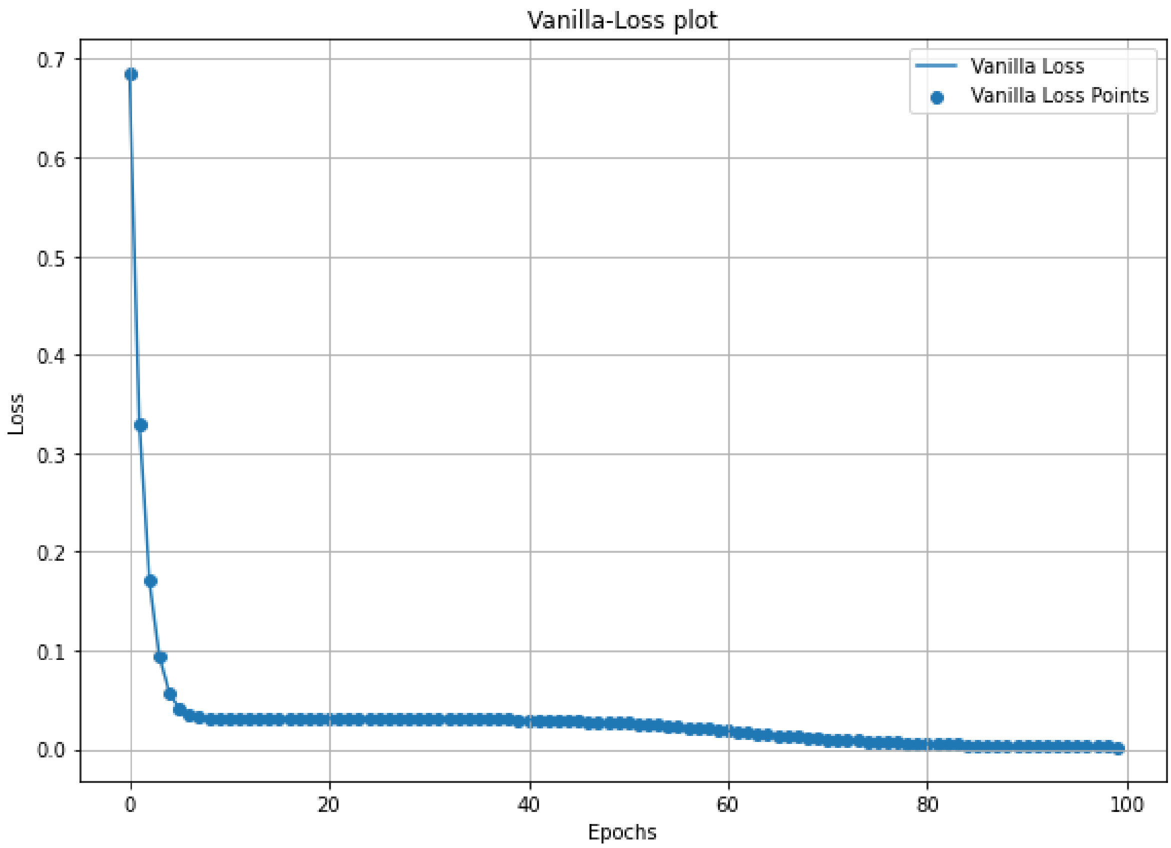
2.1 Algorithm with Vanilla update of weights

In [14]:

```
mu, sigma = 0, 0.01
learning_rate = 0.001
w1 = np.random.normal(mu, sigma, 9)
v_loss = []
for j in range(0,100):
    l1 = []
    for i in range(len(X)):
        fp = forward_propagation(X[i],y[i],w1)
        dw = backward_propagation(X[i],w1,fp)
        grad = np.asarray(list(dw.values()))

        w1 = w1 - learning_rate * grad
        l1.append(fp['L'])
    v_loss.append(np.mean(l1))
```

In [15]:

```
plt.figure(figsize = (10,7))
epoch = np.arange(0,100)
plt.plot(epoch,v_loss, label='Vanilla Loss')
plt.scatter(epoch,v_loss, label = 'Vanilla Loss Points')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Vanilla-Loss plot')
plt.grid()
plt.show()
```



In [ ]:

```

```

In [ ]:

```

```

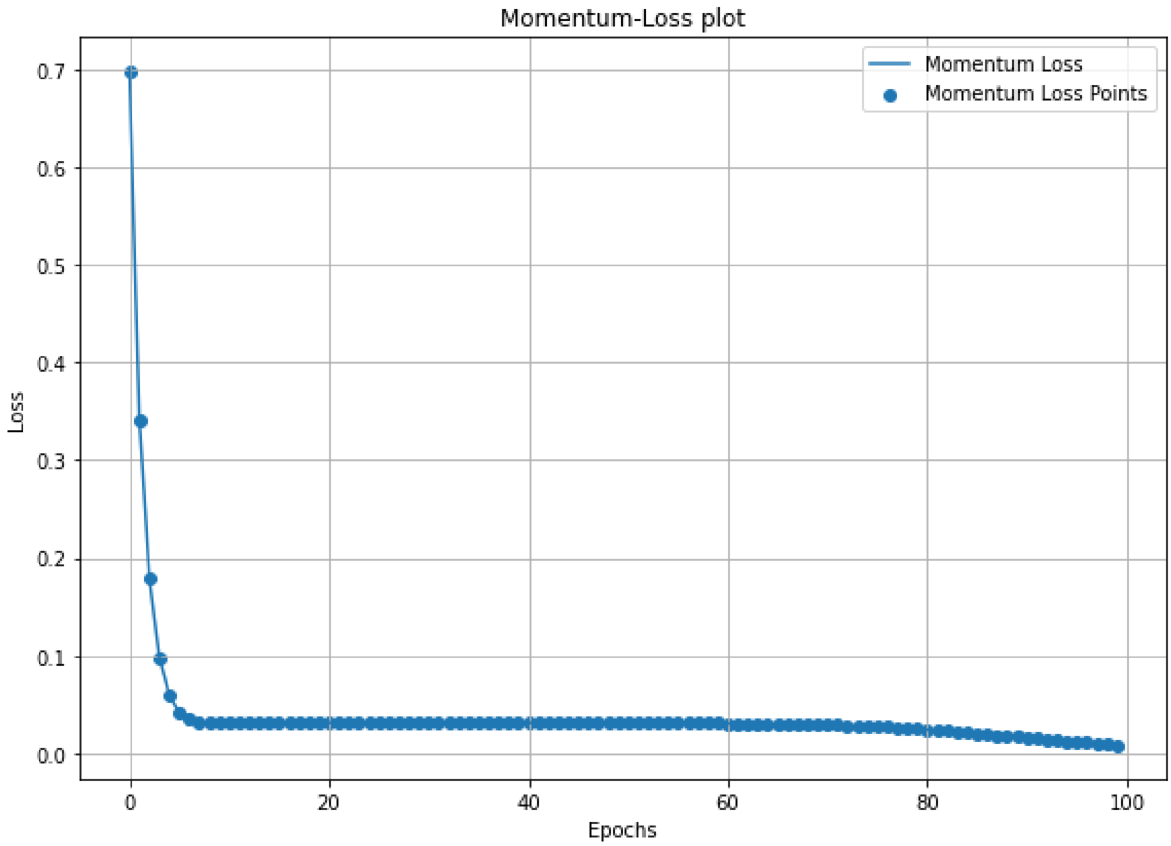
2.2 Algorithm with Momentum update of weights

In [16]:

```
mu, sigma = 0, 0.01
learning_rate = 0.001
w1 = np.random.normal(mu, sigma, 9)
m_loss = []
m = 0
beta = 0.9
for j in range(0,100):
    l1 = []
    for i in range(len(X)):
        fp = forward_propagation(X[i],y[i],w1)
        dw = backward_propagation(X[i],w1,fp)
        grad = np.asarray(list(dw.values()))

        m = (beta*m) + (1 - beta) * grad


        w1 = w1 - learning_rate * m
        l1.append(fp['L'])
    m_loss.append(np.mean(l1))
```

In [17]:

```
plt.figure(figsize = (10,7))
epoch = np.arange(0,100)
plt.plot(epoch,m_loss, label='Momentum Loss')
plt.scatter(epoch,m_loss, label = 'Momentum Loss Points')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Momentum-Loss plot')
plt.grid()
plt.show()
```



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

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 [ ]:

```

```

In [ ]:

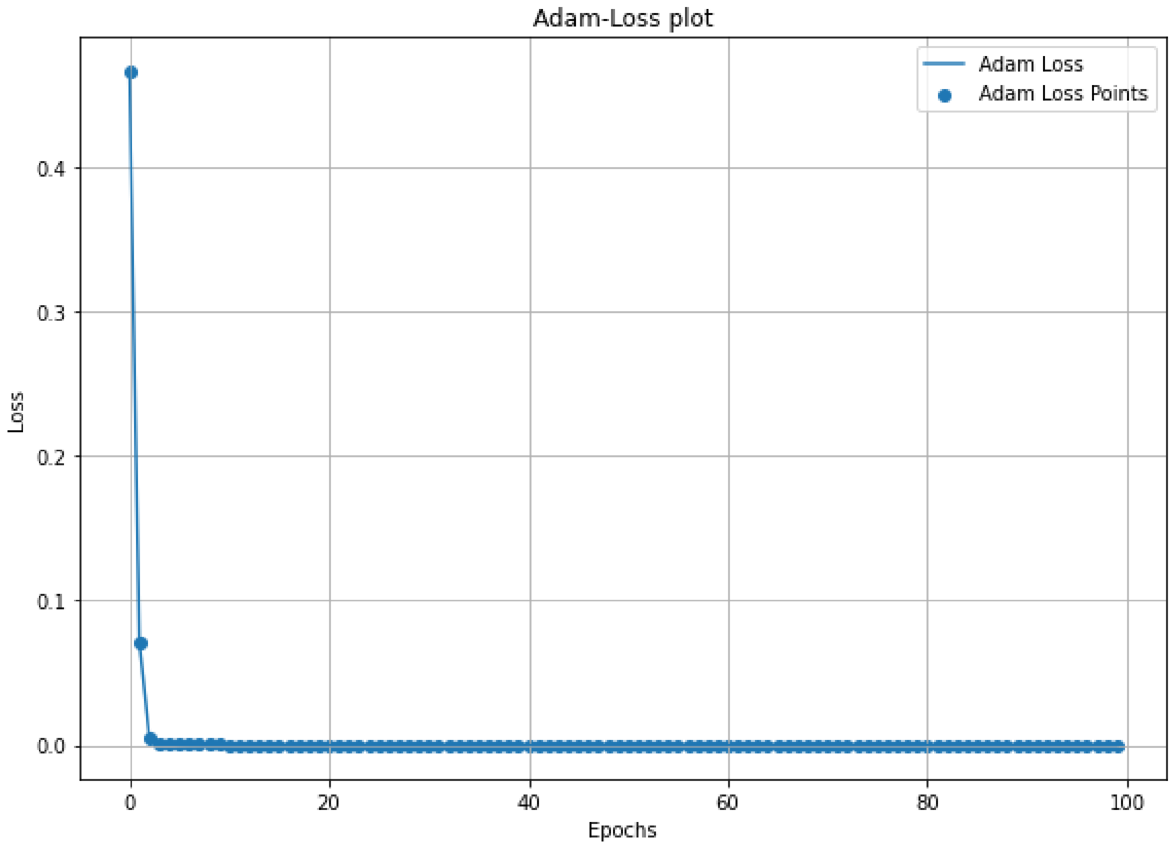
```

```

In [18]:

```
mu, sigma = 0, 0.01
learning_rate = 0.001
w1 = np.random.normal(mu, sigma, 9)
a_loss = []
m = 0
mt = 0
beta1 = 0.9
beta2 = 0.99
ep = 1e-8
v = 0
vt = 0
for j in range(0,100):
    l1 = []
    for i in range(len(X)):
        fp = forward_propagation(X[i],y[i],w1)
        dw = backward_propagation(X[i],w1,fp)
        grad = np.asarray(list(dw.values()))

        m = (beta1 * m) + ((1 - beta1) * grad)
        v = (beta2 * v) + ((1 - beta2) * np.power(grad,2))

        mt = m / (1 - beta1)
        vt = v / (1 - beta2)


        w1 = w1 - ((learning_rate * mt) / (np.sqrt(vt) + ep))
        l1.append(fp['L'])
    a_loss.append(np.mean(l1))
```

In [19]:

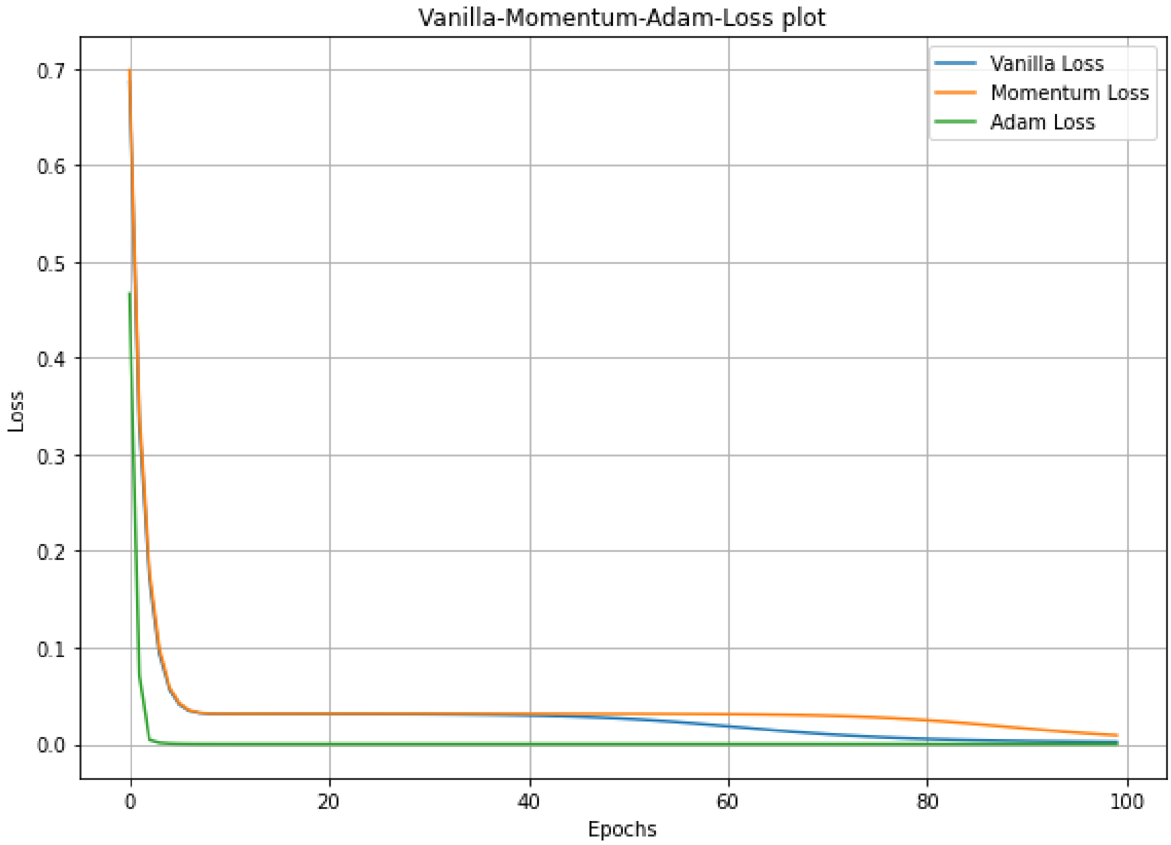
```
plt.figure(figsize = (10,7))
epoch = np.arange(0,100)
plt.plot(epoch,a_loss, label='Adam Loss')
plt.scatter(epoch,a_loss, label = 'Adam Loss Points')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Adam-Loss plot')
plt.grid()
plt.show()
```



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

In [20]:

```
#plot the graph between loss vs epochs for all 3 optimizers.
plt.figure(figsize = (10,7))
epoch = np.arange(0,100)
plt.plot(epoch,v_loss, label='Vanilla Loss')
plt.plot(epoch,m_loss, label='Momentum Loss')
plt.plot(epoch,a_loss, label='Adam Loss')

plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Vanilla-Momentum-Adam-Loss plot')
plt.grid()
plt.show()
```



You can go through the following blog to understand the implementation of other optimizers .
[Gradients update blog (https://cs231n.github.io/neural-networks-3/)](https://cs231n.github.io/neural-networks-3/)

In [ ]:

```

```