BackPropagation

There will be some functions that start with the word "grader" ex: grader_sigmoid(), grader_forwardprop(), grader_backprop() etc, you should not change those function definition.

Every Grader function has to return True.

Loading data

In [1]:

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

```
Drive already mounted at drive; to attempt to forcibly remount, call drive.mount("drive", force_remount
=True).
```

In [2]:

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

In [3]:

```
with open('/content/drive/MyDrive/19_Backpropagation and 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)
```

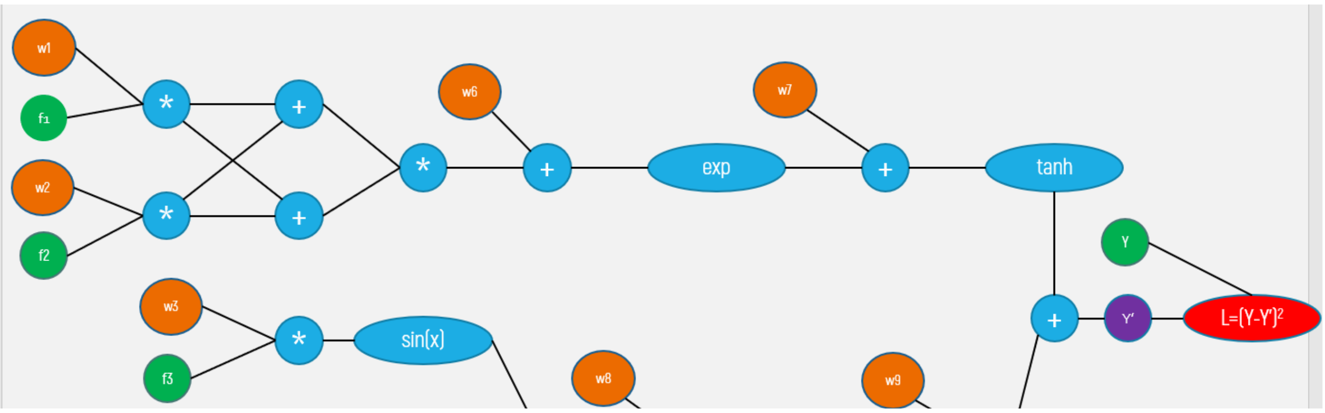
```
(506, 6)
(506, 5) (506,)
```

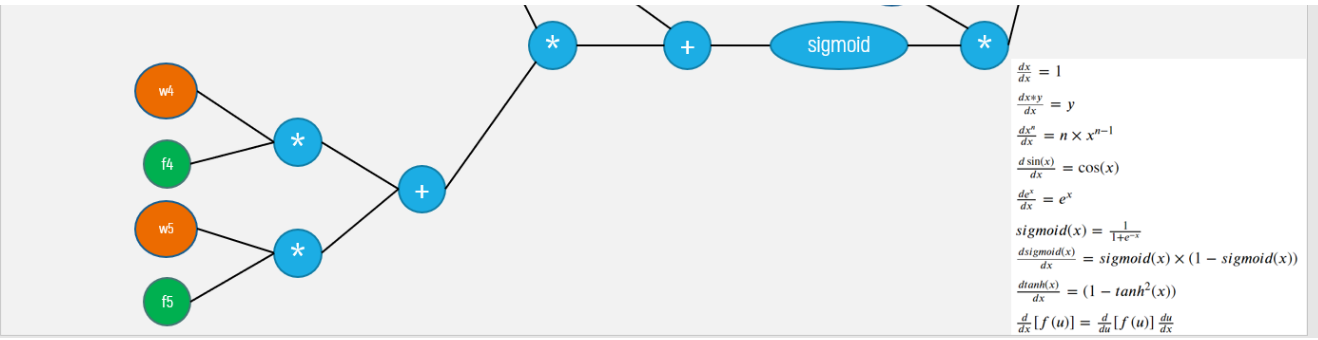
In [4]:

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

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

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 backpropagation and Gradient checking

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

In [5]:

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

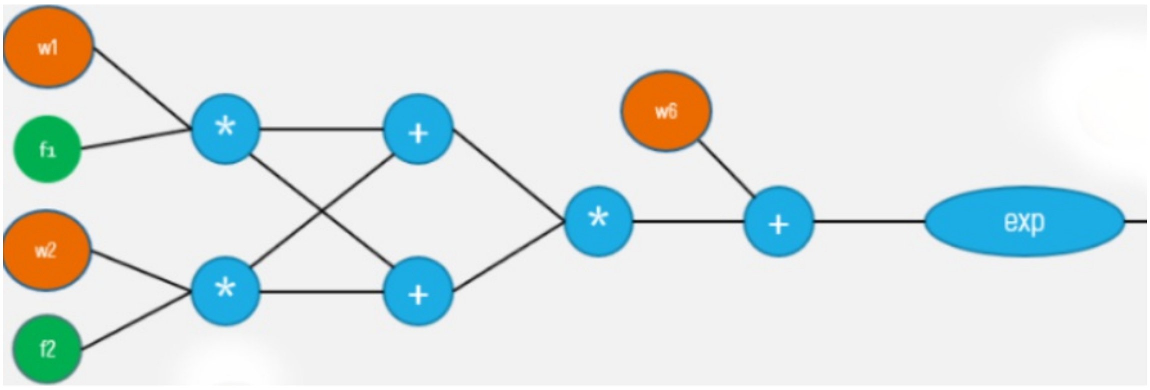
Out[5]:

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

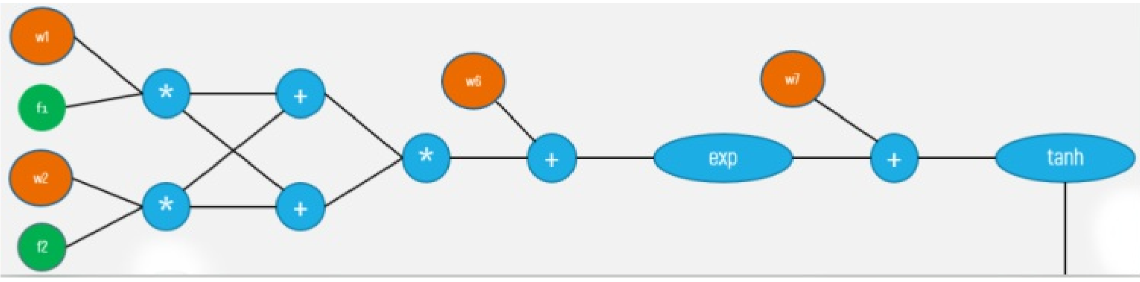
    For easy debugging, we will break the computational graph into 3 parts.

    Part 1

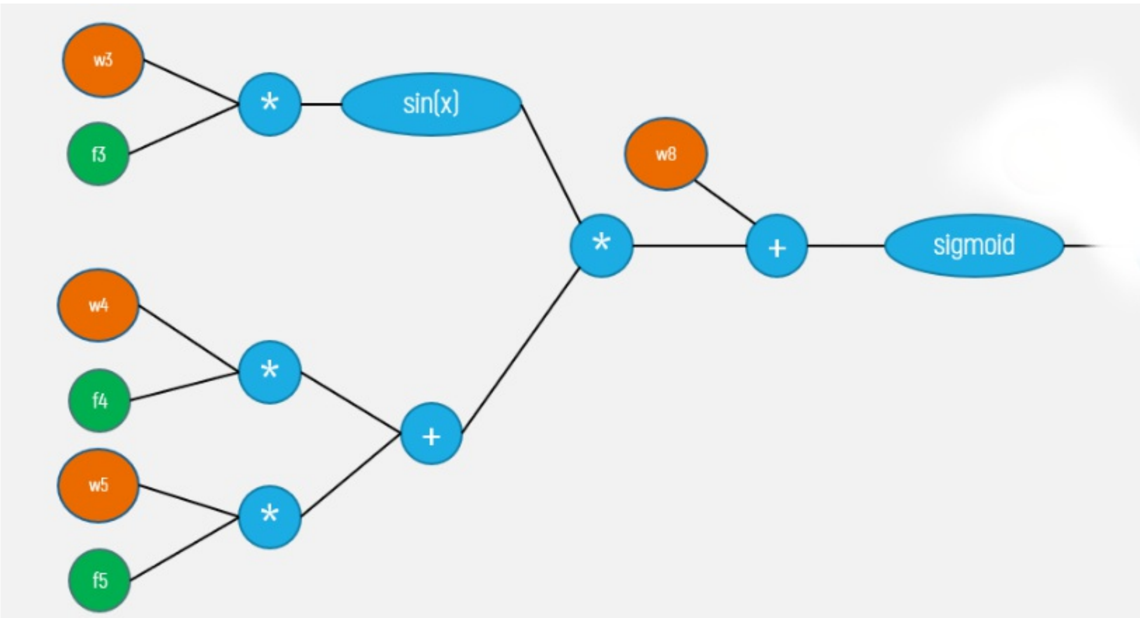
Part 1



Part 2



Part 3




```
def forward_propagation(X, y, W):

 # X: input data point, note that in this assignment you are having 5-d data point
s
 # y: output varible
 # W: weight array, its of length 9, W[0] corresponds to w1 in graph, W[1] corresp
onds to w2 in graph,
         ..., W[8] corresponds to w9 in graph.
 # 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 value
s in tanh)
 # sig = part3(compute the forward propagation until sigmoid and then store the va
lues in sig)
```

```
    # now compute remaining values from computional graph and get y'
    # write code to compute the value of L=(y-y')^2
    # compute derivative of L  w.r.to Y' and store it in dl
    # Create a dictionary to store all the intermediate values
    # store L, exp,tanh,sig,dl variables

    return (dictionary, which you might need to use for back propagation)
```

- **Backward propagation(Write your code in def backward_propagation())**

```
def backward_propagation(L, W,dictionary):

    # 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
    # return dW, dW is a dictionary with gradients of all the weights

    return dW
```

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** $\epsilon = 0.0001$

$$\begin{aligned} dw_1^{approx} &= \frac{f(w1+\epsilon, w2, x1, x2) - f(w1-\epsilon, w2, x1, x2)}{2\epsilon} \\ &= \frac{((1+0.0001)^2.3+2.4) - ((1-0.0001)^2.3+2.4)}{2\epsilon} \\ &= \frac{(1.00020001.3+2.4) - (0.99980001.3+2.4)}{2*0.0001} \\ &= \frac{(11.00060003) - (10.99940003)}{0.0002} \end{aligned}$$

$$= 5.99999999999$$

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()</font>
    approx_gradients = []
    for each wi weight value in W:<font color='grey'>
        # add a small value to weight wi, and then find the values of L with the updated weig
    hts
        # 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</font>
        approx_gradients.append(approximation gradients of weight wi)<font color='grey'>
    # compare the gradient of weights W from backword_propagation() with the aproximation gra
    dients of weights with <br>  gradient_check formula</font>
    return gradient_check</font>
```

```
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
```

Task 2 : Optimizers

- As a part of this task, you will be implementing 3 type of optimizers(methods to update weight)
- Use the same computational graph that was mentioned above to do this task
- Initilze the 9 weights from normal distribution with mean=0 and std=0.01

Check below video and this blog

In [6]:

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

Out[6]:

Algorithm

```
    for each epoch(1-100):
        for each data point in your data:
            using the functions forward_propagation() and backword_propagation() compute the gra
dients of weights
            update the weigts with help of gradients  ex: w1 = w1-learning_rate*dw1
```

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 .

Task 1

Task 1

Forward propagation

In [7]:

```
import math
```

In [8]:

```
def sigmoid(z):
  val = 1/(1+np.exp(-z))
  return val
```

In [9]:

```
def grader_sigmoid(z):
  val=sigmoid(z)
  assert(val==0.8807970779778823)
  return True
grader_sigmoid(2)
```

Out[9]:

```
True
```

In [10]:

```
def forward_propagation(x, y, w):
        '''In this function, we will compute the forward propagation '''
        # # features and weights as per computational graph
        f1 = x[0];f2 = x[1];f3 = x[2];f4=x[3];f5=x[4]
        w1 = w[0];w2=w[1];w3=w[2];w4=w[3];w5=w[4];w6=w[5];w7=w[6];w8=w[7];w9=w[8]
        # dictionary to store the required
        dic = {}
        # exp= part1 (compute the forward propagation until exp and then store the values in exp)
        dic['exp'] = math.exp(np.square(w1*f1+w2*f2)+w6)
        # tanh =part2(compute the forward propagation until tanh and then store the values in tanh)
        dic['tanh'] = np.tanh(dic['exp']+w7)
        # sig = part3(compute the forward propagation until sigmoid and then store the values in sig)
        dic['sigmoid'] = sigmoid(np.sin(w3*f3)*(w4*f4+w5*f5)+w8)
        # now compute remaining values from computional graph and get y'
        y_pre = dic['sigmoid']*w9+dic['tanh']
        # write code to compute the value of L=(y-y')^2
        dic['loss'] = (y-y_pre)**2
        # compute derivative of L  w.r.to Y' and store it in dl
        dic['dy_pr'] = (2.0)*(y_pre-y)        # Create a dictionary to store all the intermediate values

        ## Local gradients....
        ## dexp_w1,dexp_w2,dexp_w6,dtanh_w7,dsigmoid_w4,dsigmoid_w5,dsigmoid_w3,dsigmoid_w8,dtanh_exp,dyout_tanh,dl_yout
        dic['dexp_w1'] = dic['exp']*2*(w1*f1+w2*f2)*f1
        dic['dexp_w2'] = dic['exp']*2*(w1*f1+w2*f2)*f2
        dic['dexp_w6'] = dic['exp']
        dic['dtanh_w7'] = (1-dic['tanh']**2)
        dic['dsigmoid_w4'] = dic['sigmoid']*(1-dic['sigmoid'])*(math.sin(w3*f3))*f4
        dic['dsigmoid_w5'] = dic['sigmoid']*(1-dic['sigmoid'])*(math.sin(w3*f3))*f5
        dic['dsigmoid_w3'] = dic['sigmoid']*(1-dic['sigmoid'])*(w4*f4+w5*f5)*np.cos(w3*f3)*f3
        dic['dsigmoid_w8'] = dic['sigmoid']*(1-dic['sigmoid'])
        dic['dtanh_exp'] =   (1- dic['tanh']**2)
        dic['dypre_tanh']  = 1.0
        dic['dypre_sigmoid'] = w9
        dic['dypre_w9']  = dic['sigmoid']

        return dic
```

Grader function - 1

In [11]:

```
def grader_sigmoid(z):
  val=sigmoid(z)
  assert(val==0.8807970779778823)
  return True
grader_sigmoid(2)
```

Out[11]:

True

Grader function - 2

In [12]:

```
def grader_forwardprop(data):
    dl = (data['dy_pr']==-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)
```

Out[12]:

True

Backward propagation

In [13]:

```
def backward_propagation(x,W,dic):
    '''In this function, we will compute the backward propagation '''
    # features and weights as per computational graph
    f1 = x[0];f2 = x[1];f3 = x[2];f4=x[3];f5=x[4]
    w1 = w[0];w2=w[1];w3=w[2];w4=w[3];w5=w[4];w6=w[5];w7=w[6];w8=w[7];w9=w[8]
    dict = {}

    # computing derivatives using internal derivates multiplication derived in forward propagation.

    dw4  = dic['dy_pr']*dic['dypre_sigmoid']*dic['dsigmoid_w4']
    dw5  = dic['dy_pr']*dic['dypre_sigmoid']*dic['dsigmoid_w5']
    dw6  = dic['dy_pr']*dic['dypre_tanh']*dic['dtanh_exp']*dic['dexp_w6']
    dw7  = dic['dy_pr']*dic['dypre_tanh']*dic['dtanh_w7']
    dw8  = dic['dy_pr']*dic['dypre_sigmoid']*dic['dsigmoid_w8']
    dw9  = dic['dy_pr']*dic['dypre_w9']
    dw1  = dw6*(2*f1)*(w1*f1+w2*f2)
    dw2  = dw6*(2*(w1*f1+w2*f2)*f2)
    dw3  = dw8*((w4*f4+w5*f5)*np.cos(w3*f3)*f3)

    dict['dw1'] = dw1
    dict['dw2'] = dw2
    dict['dw3'] = dw3
    dict['dw4'] = dw4
    dict['dw5'] = dw5
    dict['dw6'] = dw6
    dict['dw7'] = dw7
    dict['dw8'] = dw8
    dict['dw9'] = dw9

    return dict

    # return dW, dW is a dictionary with gradients of all the weights
```

Grader function - 3

In [14]:

```
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[14]:

```
True
```

Implement gradient checking

In [15]:

```
from numpy.linalg import norm
```

In [16]:

```
def gradient_checking(data_point,y, W,epsilon):
    # compute the L value using forward_propagation()
    forward=forward_propagation(data_point,y,W)
    # compute the gradients of W using backword_propagation()
    back_gradients=backward_propagation(data_point,W,forward)
    approx_gradients = []
    for i in range(len(W)):
        W_plus = np.copy(W)
        # adding small value to weight and calculating loss
        W_plus[i] = W_plus[i]+epsilon
        f1 = forward_propagation(data_point,y,W_plus)
        f1_loss = f1['loss']
        W_minus = np.copy(W)
        # subtracting small value from weight and calculating loss
        W_minus[i]  = W_minus[i]-epsilon
        f2 = forward_propagation(data_point,y,W_minus)
        f2_loss = f2['loss']

        # approximation ..
        w_apprx = (f1_loss-f2_loss)/(2*(epsilon))
        approx_gradients.append(w_apprx)

    # compare the gradient of weights W from backword_propagation() with the aproximation gradients of
weights with gradient_check formula
    back_gradients = np.array(list(back_gradients.values()))
    approx_gradients  = np.array(approx_gradients)
    numerator = norm(back_gradients - approx_gradients)
    denominator = norm(back_gradients)+norm(approx_gradients)
    gradient_check = numerator/denominator
    if gradient_check < 1e-7:
      print("gradient is correct")
    else :
      print("gradient is wrong")
    return gradient_check
```

In [17]:

In [17]:

```
W = np.ones(9)*0.1
epsilon = 1e-7
gradient_checking(X[0],y[0], W,epsilon)
```

```
gradient is correct
```

Out[17]:

```
4.1691571909481967e-10
```

Task 2: Optimizers

- 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

references:

1. https://arxiv.org/pdf/1609.04747.pdf
2. https://towardsdatascience.com/10-gradient-descent-optimisation-algorithms-86989510b5e9

In [18]:

```
import matplotlib.pyplot as plt
```

Algorithm with Vanilla update of weights

vanilla gradient descent (aka batch gradient decent) We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters. So that's just one step of gradient descent in one epoch.

In [19]:

```
def sgd_vanilla(X,y,W,epochs,alpha=0.001):
  epochWise_avgLoss = []

  N = len(X)
  for each_epoch in range(epochs):
    # We take the average of the gradients of all the training examples
    # and then use that mean gradient to update our parameters. So that's just one step of gradient des
cent in one epoch.

    # to store the gradients (sumOf w1 for all datapoints,sumOfw1 for all datapoints,...,sumOf w9 for a
ll datapoints) and then do mean
    # and then update 'W'
    grads = [0,0,0,0,0,0,0,0,0]

    for x1,y1 in zip(X,y):
      forward = forward_propagation(x1,y1,W)
      gradients = backward_propagation(x1,y1,forward)
      l = list(gradients.values())
      grads = [i+j for i,j in zip(grads,l)]

    grads = np.array(grads)/N
    W = W - alpha*(grads)
    # once after weights updated calculate loss w.r.t updated weights.
    loss = []
    for x1,y1 in zip(X,y):
      forward = forward_propagation(x1,y1,W)
      loss.append(forward['loss'])
    epochWise_avgLoss.append(np.mean(loss))
  return epochWise_avgLoss
```

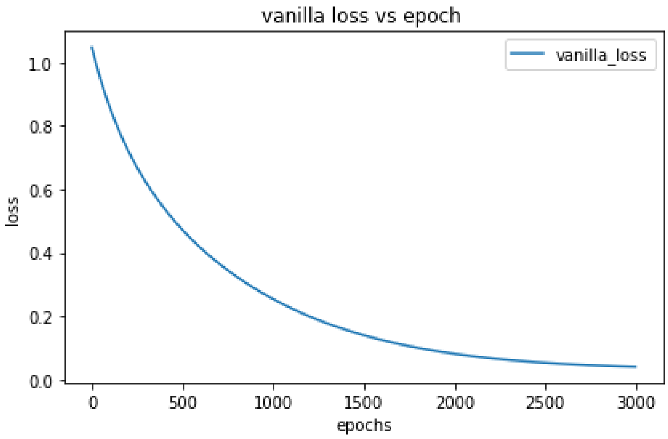
In [20]:

```
W = np.random.normal(loc=0,scale=0.01,size=9)
epochs = 3000
vanilla_loss = sgd_vanilla(X,y,W,epochs)
```

Plot between epochs and loss

In [21]:

```
plt.plot(range(3000),vanilla_loss,label='vanilla_loss')
plt.legend()
plt.xlabel('epochs')
plt.ylabel('loss')
plt.title("vanilla loss vs epoch ")
plt.tight_layout()
```



Algorithm with momentum update of weights

Instead of depending only on the current gradient to update the weight, gradient descent with momentum replaces the current gradient with m ("momentum"), which is an aggregate of gradients.

$$w_{t+1} = w_t - \alpha m_t$$

where,

$$m_t = \beta m_{t-1} + (1-\beta)\frac{\partial L}{\partial w_t}$$

In [22]:

```
def sgd_momentum(X,y,W,epochs,alpha=0.001,beta=0.9,m=0):
  epochWise_avgLoss = []
  for each_epoch in range(epochs):
    epoch_loss = []
    for x1,y1 in zip(X,y):
      forward = forward_propagation(x1,y1,W)
      # loss for every point w.r.t to 'W'  and stored to get avg loss for each epoch
      epoch_loss.append(forward['loss'])
      gradients = backward_propagation(x1,y1,forward)
      grads = np.array(list(gradients.values()))

      #updating weights using momentum:
      m = beta*(m)+(1-beta)*grads
      W = W - alpha*(m)

    epochWise_avgLoss.append(np.mean(epoch_loss))
```

```
    return epochWise_avgLoss
```

In [23]:

```
W = np.random.normal(loc=0,scale=0.01,size=9)
epochs = 10
momentum_loss=sgd_momentum(X,y,W,epochs)
```

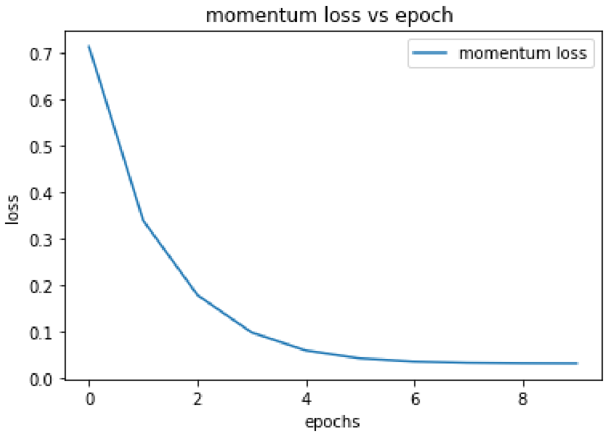
Plot between epochs and loss

In [24]:

```
plt.plot(range(10),momentum_loss,label='momentum loss')
plt.legend()
plt.xlabel('epochs')
plt.ylabel('loss')
plt.title("momentum loss vs epoch ")
```

Out[24]:

```
Text(0.5, 1.0, 'momentum loss vs epoch ')
```



Algorithm with adam update of weights

both gradient and learning rate are learned.

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

where

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

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial L}{\partial w_t}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left[\frac{\partial L}{\partial w_t}\right]^2$$

In [25]:

```
def sgd_adam(X,y,W,epochs,alpha=0.001,beta1=0.9,beta2=0.999,m=0,v=0,epsilon=1e-8):
```

```
def sgd_adam(X,y,W,epochs,alpha=0.001,beta1=0.9,beta2=0.999,m=0,v=0,epsilon=1e-8):
  epochWise_avgLoss = []
  t = 1
  for each_epoch in range(epochs):
    epoch_loss = []
    for x1,y1 in zip(X,y):

      forward = forward_propagation(x1,y1,W)
      epoch_loss.append(forward['loss'])
      gradients = backward_propagation(x1,y1,forward)
      grads = np.array(list(gradients.values()))

      #updating weights using adam:
      m = beta1*m + (1-beta1)*(grads)
      v = beta2*v + (1-beta2)*(grads**2)
      m_hat = m/(1-np.power(beta1,t))
      v_hat = v/(1-np.power(beta2,t))
      t+=1
      W = W - (alpha/(np.sqrt(v_hat)+epsilon))*(m_hat)

    epochWise_avgLoss.append(np.mean(epoch_loss))
  return epochWise_avgLoss
```

In [26]:

```
W = np.random.normal(loc=0,scale=0.01,size=9)
epochs = 10
adam_loss = sgd_adam(X,y,W,epochs)
```

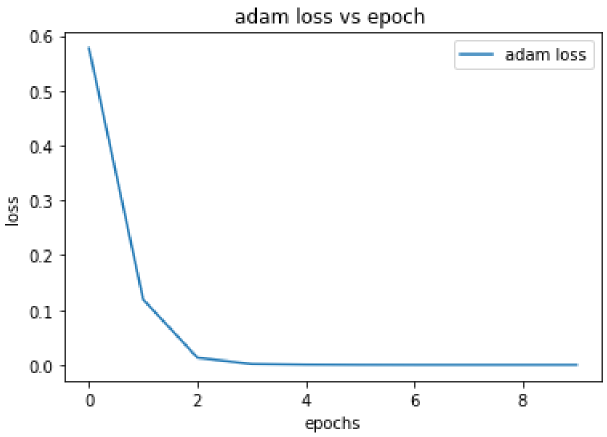
Plot between epochs and loss

In [27]:

```
plt.plot(range(10),adam_loss,label='adam loss')
plt.legend()
plt.xlabel('epochs')
plt.ylabel('loss')
plt.title("adam loss vs epoch ")
```

Out[27]:

```
Text(0.5, 1.0, 'adam loss vs epoch ')
```

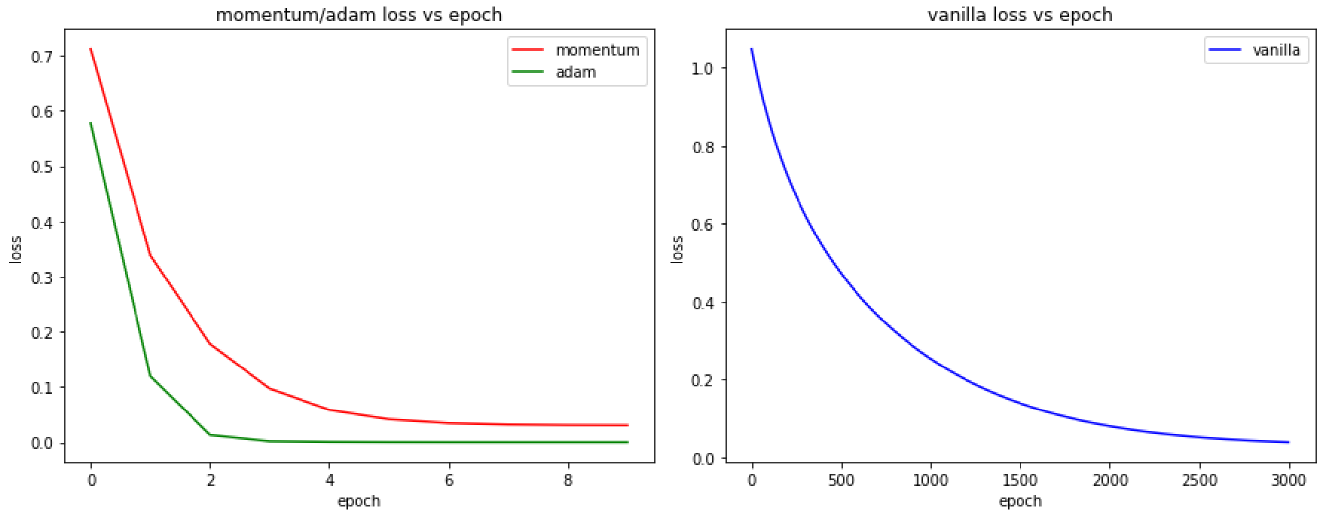


Comparision plot between epochs and loss with different optimizers

In [28]:

```
fig,ax = plt.subplots(1,2,figsize=(12.5,5))
ax[1].plot(range(3000),vanilla_loss,color='blue',label='vanilla')
ax[1].set_xlabel("epoch")
ax[1].set_ylabel("loss")
ax[1].legend()
ax[1].set_title("vanilla loss vs epoch")
```

```
ax[0].plot(range(10),momentum_loss,color='red',label='momentum')
ax[0].plot(range(10),adam_loss,color='green',label='adam')
ax[0].set_xlabel("epoch")
ax[0].set_ylabel("loss")
ax[0].set_title("momentum/adam loss vs epoch")
ax[0].legend()
plt.tight_layout()
```



Observations::

1. adam and momentum converges faster than vanilla gradient descent.
2. adam is slighlty better than momentum sgd.
3. vanilla gradient descent takes more epochs (which inturns more time) to converge.

In [28]:

```

```