BackPropagation

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

Every Grader function has to return True.

Loading data

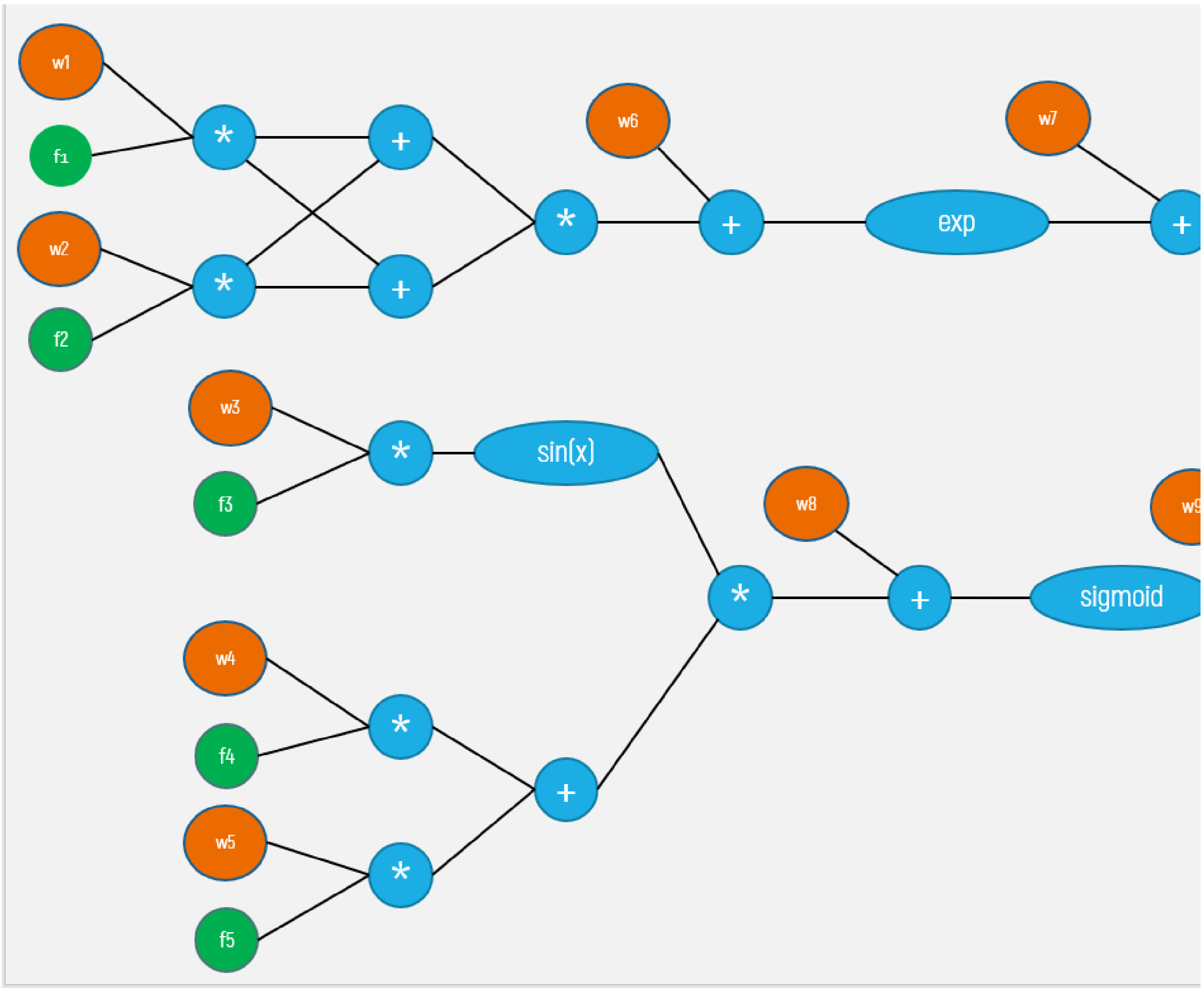
```
from google.colab import files
uploaded = files.upload()
```

```
Choose Files data.pkl
• data.pkl(n/a) - 24449 bytes, last modified: 5/24/2020 - 100% done
Saving data.pkl to data (1).pkl
```

```
import pickle
import numpy as np
import math
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,)
```

Computational graph



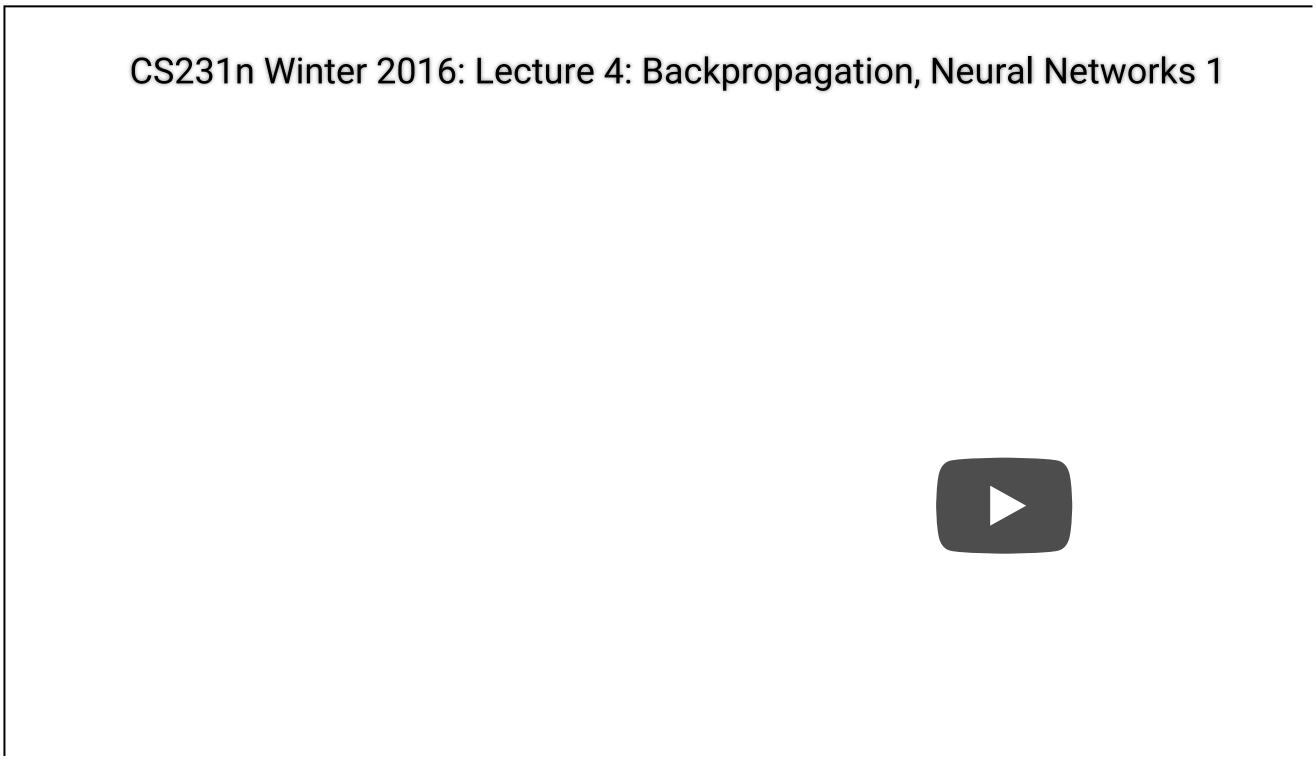
- If you observe the graph, we are having input features [f1, f2, f3, f4, f5] and 9 weights [w1, w

- The final output of this graph is a value L which is computed as (Y-Y')^2

Task 1: Implementing backpropagation and Gradient checkir

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

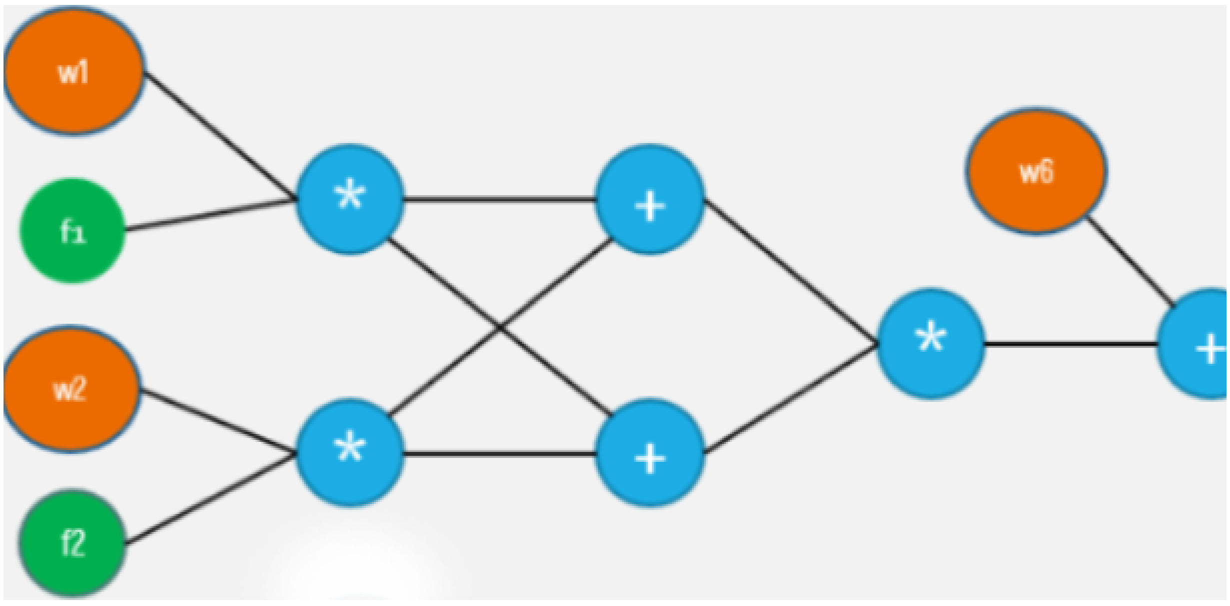
```
from IPython.display import YouTubeVideo
YouTubeVideo('i94OvYb6noo',width="1000",height="500")
```


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

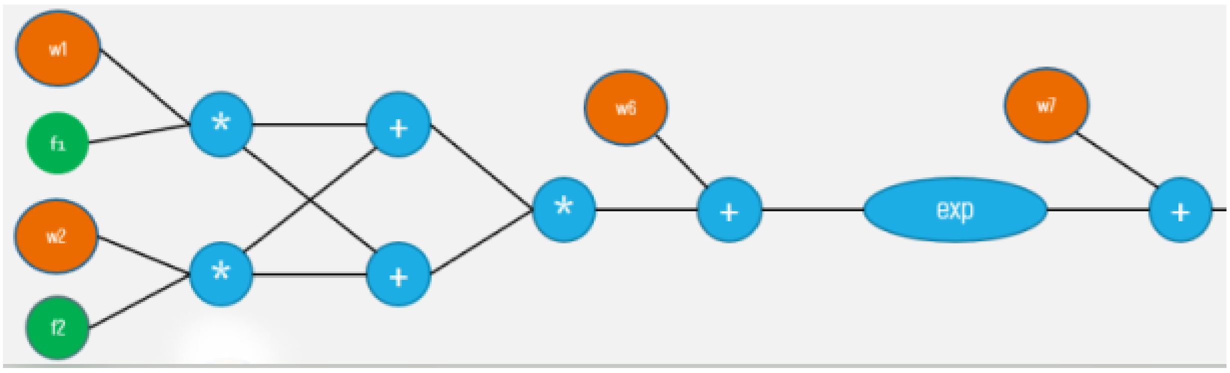
- **Write two functions**
  - **Forward propagation**(Write your code in def forward_propagation())

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

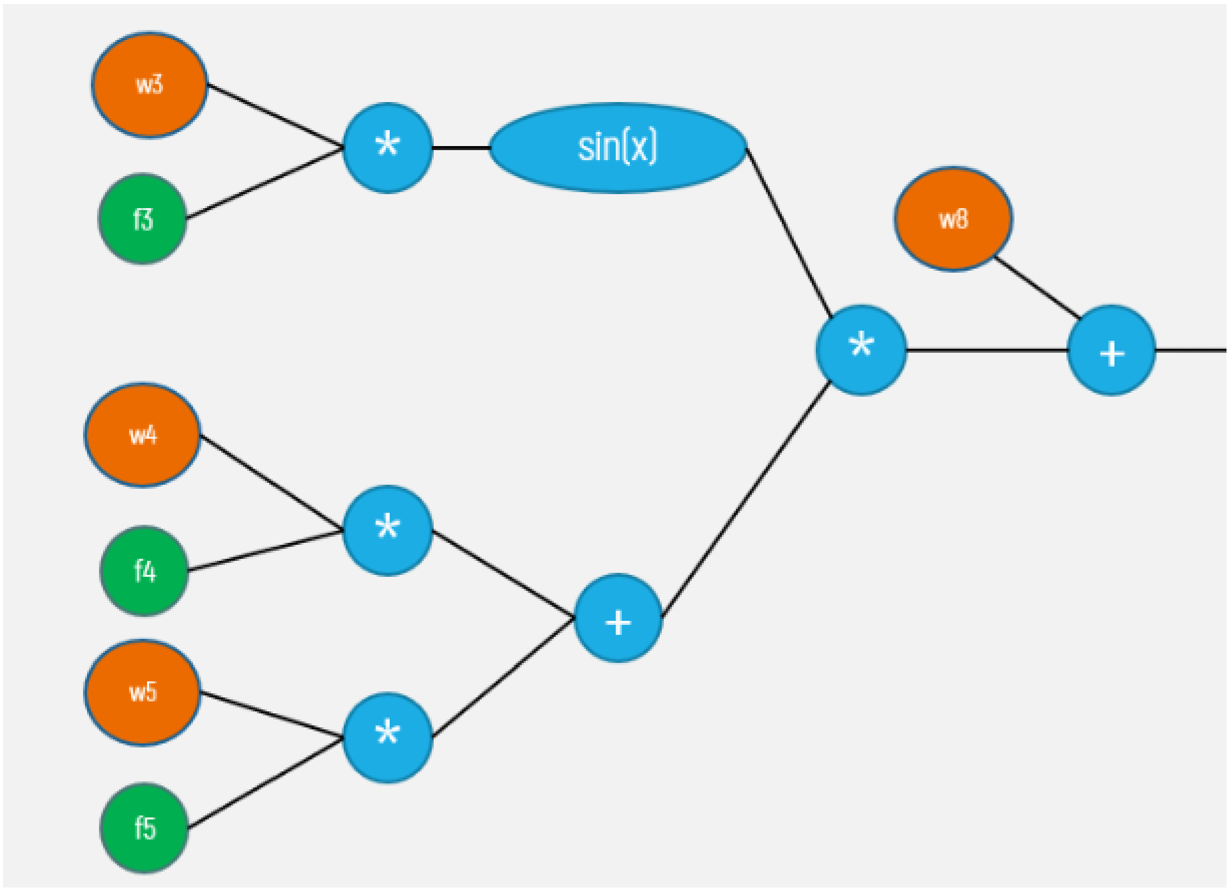
Part 1



Part 2



Part 3



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

  # X: input data point, note that in this assignment you are having 5
  # y: output varible
  # W: weight array, its of length 9, W[0] corresponds to w1 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 stc
  # tanh =part2(compute the forward propagation until tanh and then st
  # sig = part3(compute the forward propagation until sigmoid and ther
  # 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 propagatior
```

- **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,...,w
    # 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 ϵ 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 backpropagatic 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

$$\begin{aligned}\frac{df}{dw_1} = dw_1 &= 2.w_1 . x_1 \\ &= 2.1.3 \\ &= 6\end{aligned}$$

let calculate the aproximate gradient of w_1 as mentinoned in the above formula and considering ϵ

$$\begin{aligned} dw_1^{approx} &= \frac{f(w1+\epsilon,w2,x1,x2)-f(w1-\epsilon,w2,x1,x2)}{2\epsilon} \\ &= \frac{((1+0.0001)^2.3+2.4)-((1-0.0001)^2.3+2.4}{2\epsilon} \\ &= \frac{(1.00020001.3+2.4)-(0.99980001.3+2.4)}{2*0.0001} \\ &= \frac{(11.00060003)-(10.99940003)}{0.0002} \\ &= 5.99999999999 \end{aligned}$$

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 v
one of the vectors is very small. As a value for epsilon, we usually opt for 1e-7. Therefore, if gradie
means that backpropagation was implemented correctly. Otherwise, there is potentially a mistake
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 updated
        # subtract a small value to weight wi, and then find the values of L with the upd
        # 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
  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
```

Task 2 : Optimizers

- As a part of this task, you will be implementing 3 type of optimizers(methods to update weig
- 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

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

CS231n Winter 2016: Lecture 5: Neural Networks Part 2

Algorithm

```
for each epoch(1-100):
    for each data point in your data:
        using the functions forward_propagation() and backword_propagation() compute
        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 g returning False .Recheck your logic for that variable .

Task 1

Forward propagation

```
def sigmoid(z):
    '''In this function, we will compute the sigmoid(z)'''
    value=1/(1+np.exp(-z))
    return value

def forward_propagation(x, y, w):
  dict={}
  exp=np.exp((((w[0]*x[0])+(w[1]*x[1]))*((w[0]*x[0])+(w[1]*x[1])))+w[5])
  tanh=np.tanh(exp+w[6])
  part3=((np.sin(w[2]*x[2]))*((w[3]*x[3])+(w[4]*x[4])))+w[7]
  sig=sigmoid(part3)
  y_hat=tanh+sig*w[8]
  dl=-2*(y-y_hat)
  loss=pow(y-(tanh+sig*w[8]), 2)
  dict["exp"]=exp
  dict['tanh']=tanh
  dict["sigmoid"]=sig
  dict["dl"]=dl
  dict["loss"]=loss
  dict["y_hat"]=y_hat
  return dict

```

Grader function - 1

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

```
True
```

Grader function - 2

```
def grader_forwardprop(data):
    dl = (np.round(data['dl'],4)==-1.9285)
    loss=(np.round(data['loss'],4)==0.9298)
    part1=(np.round(data['exp'],4)==1.1273)
    part2=(np.round(data['tanh'],4)==0.8418)
    part3=(np.round(data['sigmoid'],4)==0.5279)
    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
```

Backward propagation

```
def backward_propagation(x,w,dict):
  der={}
  dw1=dict['dl']*(1-(math.pow(dict['tanh'],2)))*dict["exp"]*2*((w[0]*x[0])+(w[1]*x[1]))
  dw2=dict['dl']*(1-(math.pow(dict['tanh'],2)))*dict["exp"]*2*((w[0]*x[0])+(w[1]*x[1]))
  dw3=dict['dl']*(dict['sigmoid']*(1-dict['sigmoid']))*w[8]*((w[3]*x[3])+(w[4]*x[4]))*m
  dw4=dict['dl']*(dict['sigmoid']*(1-dict['sigmoid']))*w[8]*math.sin(x[2]*w[2])*x[3]
  dw5=dict['dl']*(dict['sigmoid']*(1-dict['sigmoid']))*w[8]*math.sin(x[2]*w[2])*x[4]
  dw6 =dict['dl']*(1-(math.pow(dict['tanh'],2)))*dict["exp"]
  dw7 = dict['dl']*(1-(math.pow(dict['tanh'],2)))
  dw8 = dict['dl']*(dict['sigmoid']*(1-dict['sigmoid']))*w[8]
  dw9 = dict['dl']*dict['sigmoid']
  der['dw1']=dw1
  der['dw2']=dw2
  der['dw3']=dw3
  der['dw4']=dw4
  der['dw5']=dw5
  der['dw6']=dw6
  der['dw7']=dw7
  der['dw8']=dw8
  der['dw9']=dw9
  return der
```

Grader function - 3

```
def grader_backprop(data):
    dw1=(np.round(data['dw1'],4)==-0.2297)
    dw2=(np.round(data['dw2'],4)==-0.0214)
    dw3=(np.round(data['dw3'],4)==-0.0056)
    dw4=(np.round(data['dw4'],4)==-0.0047)
    dw5=(np.round(data['dw5'],4)==-0.001)
    dw6=(np.round(data['dw6'],4)==-0.6335)
    dw7=(np.round(data['dw7'],4)==-0.5619)
    dw8=(np.round(data['dw8'],4)==-0.0481)
    dw9=(np.round(data['dw9'],4)==-1.0181)
    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)
```

```
True
```

Implement gradient checking

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

```
def gradient_checking(data_point,y, w,epsilon=1e-7):
  weight=w
  loss=forward_propagation(data_point,y,w)
  grad=backward_propagation(data_point,w,loss)
  approx_gradients = []
  grad_check=[]
  for i in range(len(w)):
    w=np.ones(9)*0.1
    w[i]=weight[i]+epsilon
    l1=forward_propagation(data_point,y,w)
    loss1=l1['loss']
    w[i]=weight[i]-epsilon
    l2=forward_propagation(data_point,y,w)
    loss2=l2["loss"]
    aprox=(loss1-loss2)/(2*epsilon)
    approx_gradients.append(aprox)
    numerator = np.linalg.norm(grad[(str("dw")+str(i+1))] - aprox)
    denominator = np.linalg.norm(grad[(str("dw")+str(i+1))]) + np.linalg.norm(aprox)
    difference = numerator / denominator
    grad_check.append(difference)
  return grad_check,approx_gradients
```

```
print("ACTUAL GRADIENT USING BACK PROPAGATION")
loss=forward_propagation(X[0],y[0],w)
backward_propagation(X[0],w,loss)
```

```
ACTUAL GRADIENT USING BACK PROPAGATION
{'dw1': -0.22973323498702,
 'dw2': -0.02140761471775293,
 'dw3': -0.005625405580266632,
 'dw4': -0.004657941222712424,
 'dw5': -0.0010077228498574248,
 'dw6': -0.6334751873437471,
 'dw7': -0.561941842854033,
 'dw8': -0.04806288407316517,
 'dw9': -1.0181044360187037}
```

```
diff,aprox_grad=gradient_checking(X[0],y[0], w,epsilon=1e-7)
print("APPROXIMATE GRADIENT AFTER GRADIENT CHECKING")
print(aprox_grad)
print("DIFFERENCE")
print(diff)
```

```
APPROXIMATE GRADIENT AFTER GRADIENT CHECKING
[-0.22973323521302547, -0.021407614569923794, -0.005625406251930087, -0.0046579412549
DIFFERENCE
[4.918867403857781e-10, 3.4527231737366113e-09, 5.96991379072821e-08, 3.4579582758165
```

- we can understand back propagation gradient calculation is working well because difference

Task 2: Optimizers

Algorithm with Vanilla update of weights

```
from sklearn.metrics import mean_squared_error
rate=.001
mu, sigma = 0, 0.01 # mean and standard deviation
w = np.random.normal(mu, sigma, 9) # weight intialization
weight=w
```

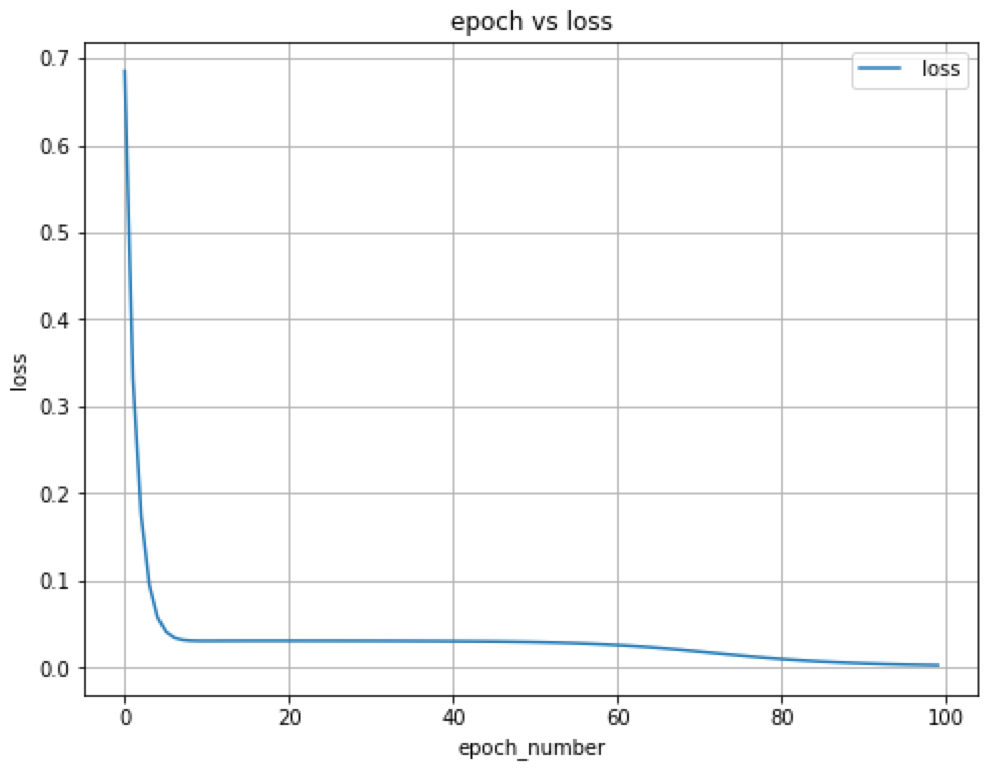
```
loss_value_vanilla=[]
epoc_vanilla=[]
```

```
for epoch in range(100):
  epoc_vanilla.append(epoch)
  y_pred=[]
  for point in range(len(data)):
    forward=forward_propagation(X[point], y[point], w)
    y_pred.append(forward['y_hat'])
    backward=backward_propagation(X[point],w,forward)
    w[0]=w[0]-rate*backward["dw1"]
    w[1]=w[1]-rate*backward["dw2"]
    w[2]=w[2]-rate*backward["dw3"]
    w[3]=w[3]-rate*backward["dw4"]
    w[4]=w[4]-rate*backward["dw5"]
    w[5]=w[5]-rate*backward["dw6"]
    w[6]=w[6]-rate*backward["dw7"]
    w[7]=w[7]-rate*backward["dw8"]
    w[8]=w[8]-rate*backward["dw9"]
  loss=mean_squared_error(y,y_pred)
  loss_value_vanilla.append(loss)
```

Plot between epochs and loss

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

```
<matplotlib.legend.Legend at 0x7ff9cd7a5be0>
```



Algorithm with momentum update of weights

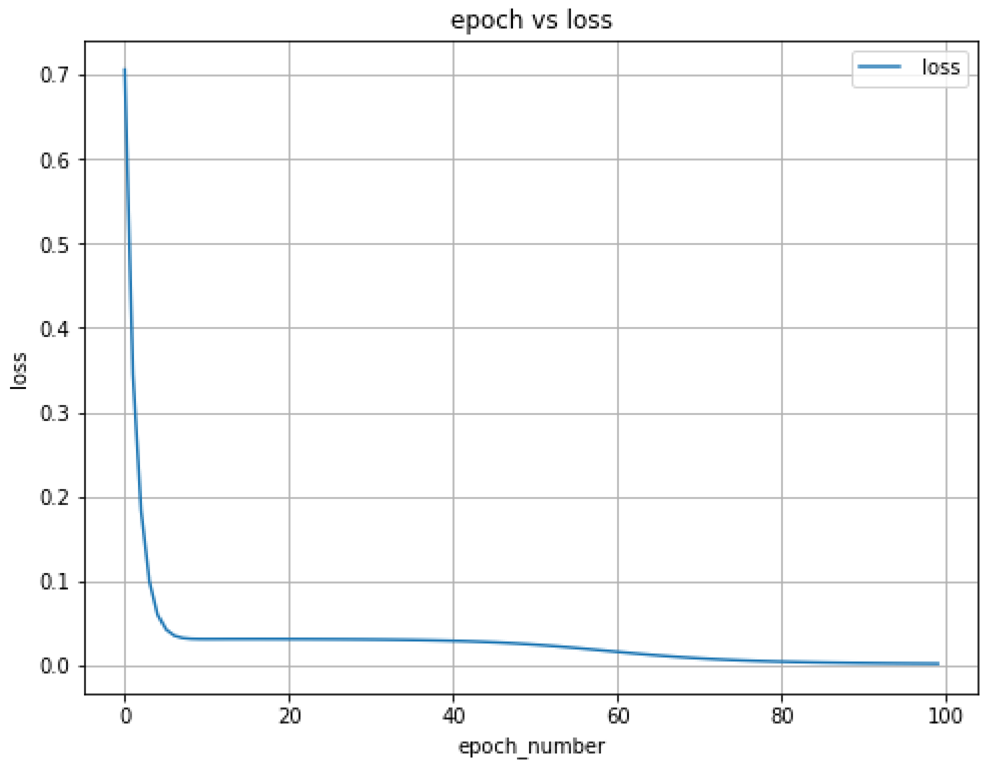
```
rate=.001
m=np.zeros(9)
b=.9
mu, sigma = 0, 0.01 # mean and standard deviation
w = np.random.normal(mu, sigma, 9) # weight intialization
```

```
loss_value_momentum=[]
epoc_momentum=[]
for epoch in range(100):
  epoc_momentum.append(epoch)
  y_pred=[]
  for point in range(len(data)):
    forward=forward_propagation(X[point], y[point], w)
    y_pred.append(forward['y_hat'])
    backward=backward_propagation(X[point],w,forward)
    m[0]=b*m[0]+(1-b)*backward["dw1"]
    w[0]=w[0]-rate*m[0]
    m[1]=b*m[1]+(1-b)*backward["dw2"]
    w[1]=w[1]-rate*m[1]
    m[2]=b*m[2]+(1-b)*backward["dw3"]
    w[2]=w[2]-rate*m[2]
    m[3]=b*m[3]+(1-b)*backward["dw4"]
    w[3]=w[3]-rate*m[3]
    m[4]=b*m[4]+(1-b)*backward["dw5"]
    w[4]=w[4]-rate*m[4]
    m[5]=b*m[5]+(1-b)*backward["dw6"]
    w[5]=w[5]-rate*m[5]
    m[6]=b*m[6]+(1-b)*backward["dw7"]
    w[6]=w[6]-rate*m[6]
    m[7]=b*m[7]+(1-b)*backward["dw8"]
    w[7]=w[7]-rate*m[7]
    m[8]=b*m[8]+(1-b)*backward["dw9"]
    w[8]=w[8]-rate*m[8]
  loss=mean_squared_error(y,y_pred)
  loss_value_momentum.append(loss)

```

Plot between epochs and loss

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

```
<matplotlib.legend.Legend at 0x7ff9cd1db630>
```



Algorithm with Adam update of weights

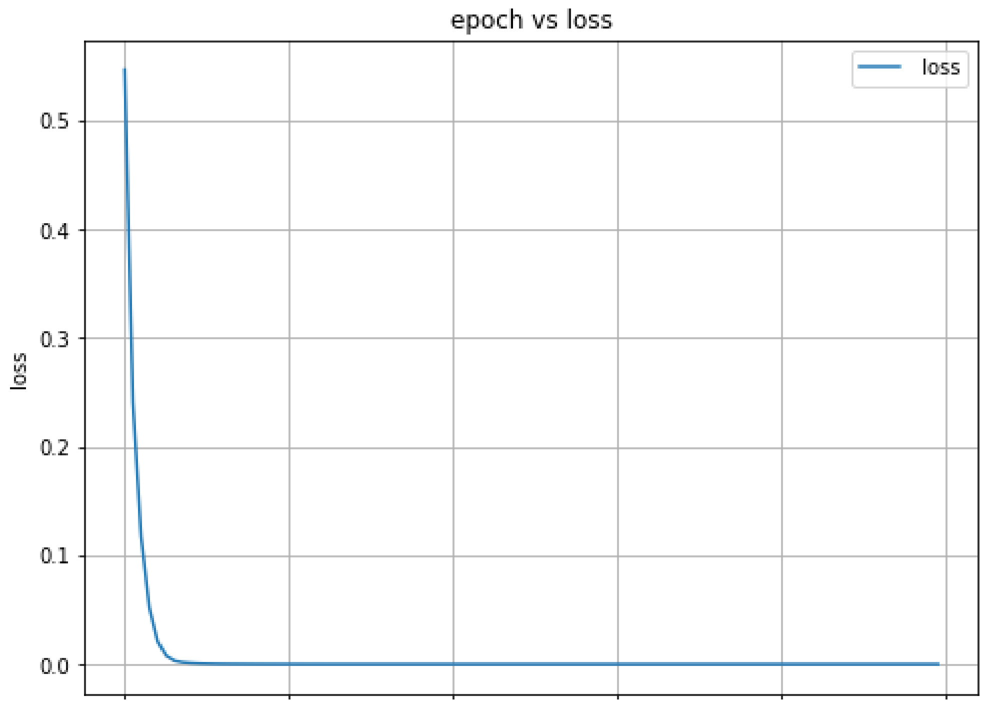
```
rate=.001
m=np.zeros(9)
v=np.zeros(9)
b1=.9
b2=0.999
z=1e-8
mu, sigma = 0, 0.01 # mean and standard deviation
w = np.random.normal(mu, sigma, 9) # weight intialization
```

```
loss_value_adam=[]
epoc_adam=[]
for epoch in range(100):
  epoc_adam.append(epoch)
  y_pred=[]
  for point in range(len(data)):
    forward=forward_propagation(X[point], y[point], w)
    y_pred.append(forward['y_hat'])
    backward=backward_propagation(X[point],w,forward)
    m[0]=b1*m[0]+(1-b1)*backward["dw1"]
    mt=m[0]/(1-b1)
    v[0]=b2*v[0]+(1-b2)*math.pow(backward["dw1"],2)
    vt=v[0]/(1-b2)
    w[0]=w[0]-(rate/(math.sqrt(vt)+z))*mt
    m[1]=b1*m[1]+(1-b1)*backward["dw2"]
    mt=m[1]/(1-b1)
    v[1]=b2*v[1]+(1-b2)*math.pow(backward["dw2"],2)
    vt=v[1]/(1-b2)
    w[1]=w[1]-(rate/(math.sqrt(vt)+z))*mt
    m[2]=b1*m[2]+(1-b1)*backward["dw3"]
    m[2]=b1*m[2]+(1-b1)*backward["dw3"]
    mt=m[2]/(1-b1)
    v[2]=b2*v[2]+(1-b2)*math.pow(backward["dw3"],2)
    vt=v[2]/(1-b2)
    w[2]=w[2]-(rate/(math.sqrt(vt)+z))*mt
    m[3]=b1*m[3]+(1-b1)*backward["dw4"]
    mt=m[3]/(1-b1)
    v[3]=b2*v[3]+(1-b2)*math.pow(backward["dw4"],2)
    vt=v[3]/(1-b2)
    w[3]=w[3]-(rate/(math.sqrt(vt)+z))*mt
    m[4]=b1*m[4]+(1-b1)*backward["dw5"]
    mt=m[4]/(1-b1)
    v[4]=b2*v[4]+(1-b2)*math.pow(backward["dw5"],2)
    vt=v[4]/(1-b2)
    w[4]=w[4]-(rate/(math.sqrt(vt)+z))*mt
    m[5]=b1*m[5]+(1-b1)*backward["dw6"]
    mt=m[5]/(1-b1)
    v[5]=b2*v[5]+(1-b2)*math.pow(backward["dw6"],2)
    vt=v[5]/(1-b2)
    w[5]=w[5]-(rate/(math.sqrt(vt)+z))*mt
    m[6]=b1*m[6]+(1-b1)*backward["dw7"]
    mt=m[6]/(1-b1)
    v[6]=b2*v[6]+(1-b2)*math.pow(backward["dw7"],2)
    vt=v[6]/(1-b2)
    w[6]=w[6]-(rate/(math.sqrt(vt)+z))*mt
    m[7]=b1*m[7]+(1-b1)*backward["dw8"]
    mt=m[7]/(1-b1)
    v[7]=b2*v[7]+(1-b2)*math.pow(backward["dw8"],2)
    vt=v[7]/(1-b2)
    w[7]=w[7]-(rate/(math.sqrt(vt)+z))*mt
    m[8]=b1*m[8]+(1-b1)*backward["dw9"]
    mt=m[8]/(1-b1)
    v[8]=b2*v[8]+(1-b2)*math.pow(backward["dw9"],2)
    vt=v[8]/(1-b2)
    w[8]=w[8]-(rate/(math.sqrt(vt)+z))*mt
  loss=mean_squared_error(y,y_pred)
  loss_value_adam.append(loss)

```

Plot between epochs and loss

```

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

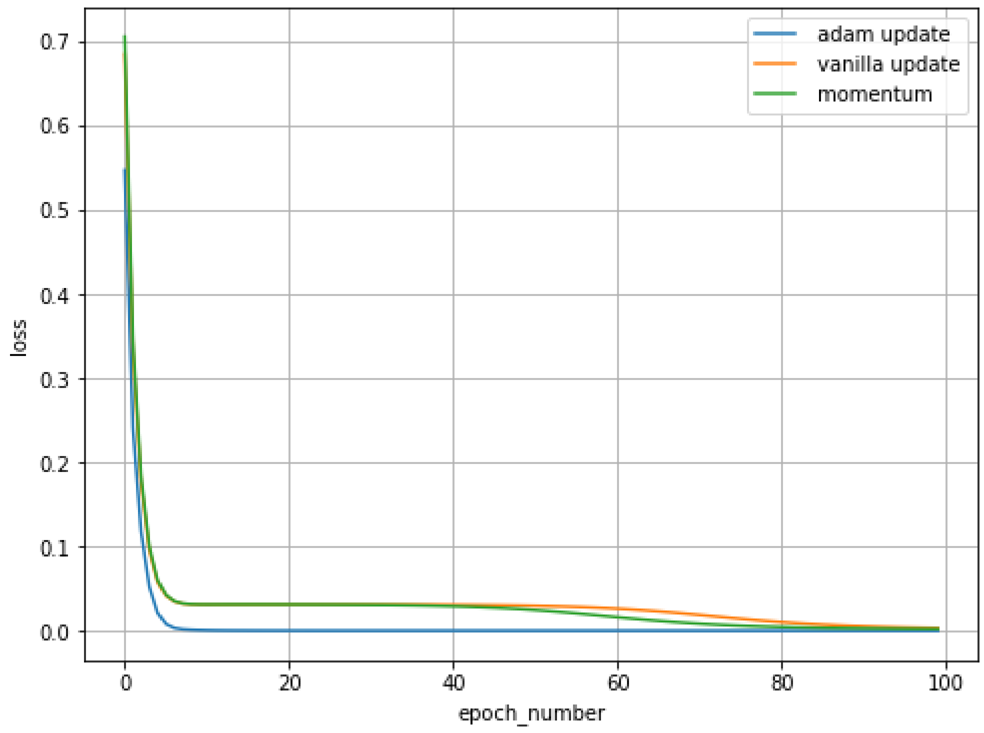
```
<matplotlib.legend.Legend at 0x7ff9ccc15438>
```



Comparision plot between epochs and loss with different optimizers

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.figure(figsize=(8,6))
plt.grid()
plt.plot(epoc_adam,loss_value_adam, label=' adam update')
plt.plot(epoc_vanilla,loss_value_vanilla, label=' vanilla update')
plt.plot(epoc_momentum,loss_value_momentum, label=' momentum')
plt.xlabel("epoch_number")
plt.ylabel("loss")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7ff9ccb0f978>
```



- from the above graph we can understand that adam optmizer convergence faster.