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

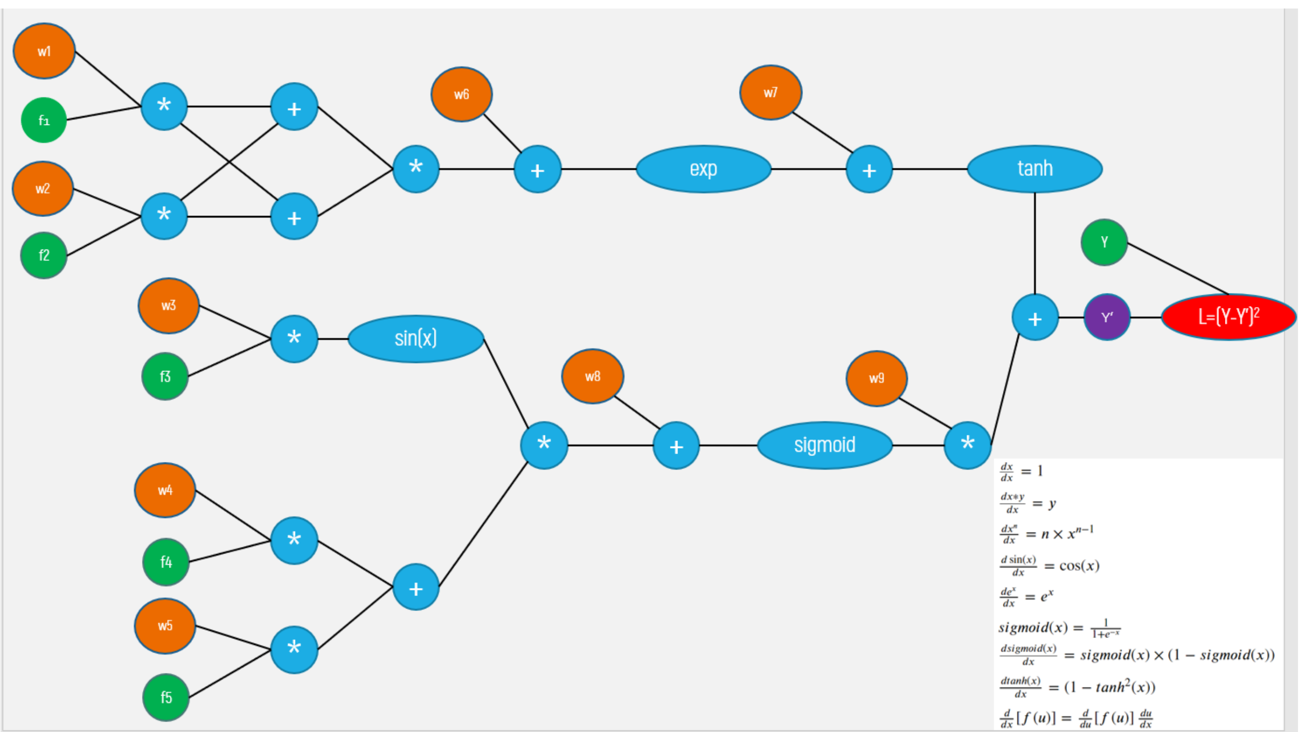
In [8]:
```
import pickle
import numpy as np
from tqdm import tqdm
import matplotlib.pyplot as plt

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,)
```

In [9]:
```
print(x[0])
print(x[0][0])
```

```
[-1.2879095  -0.12001342 -1.45900038 -0.66660821 -0.14421743]
-1.287909498957745
```

Computational graph



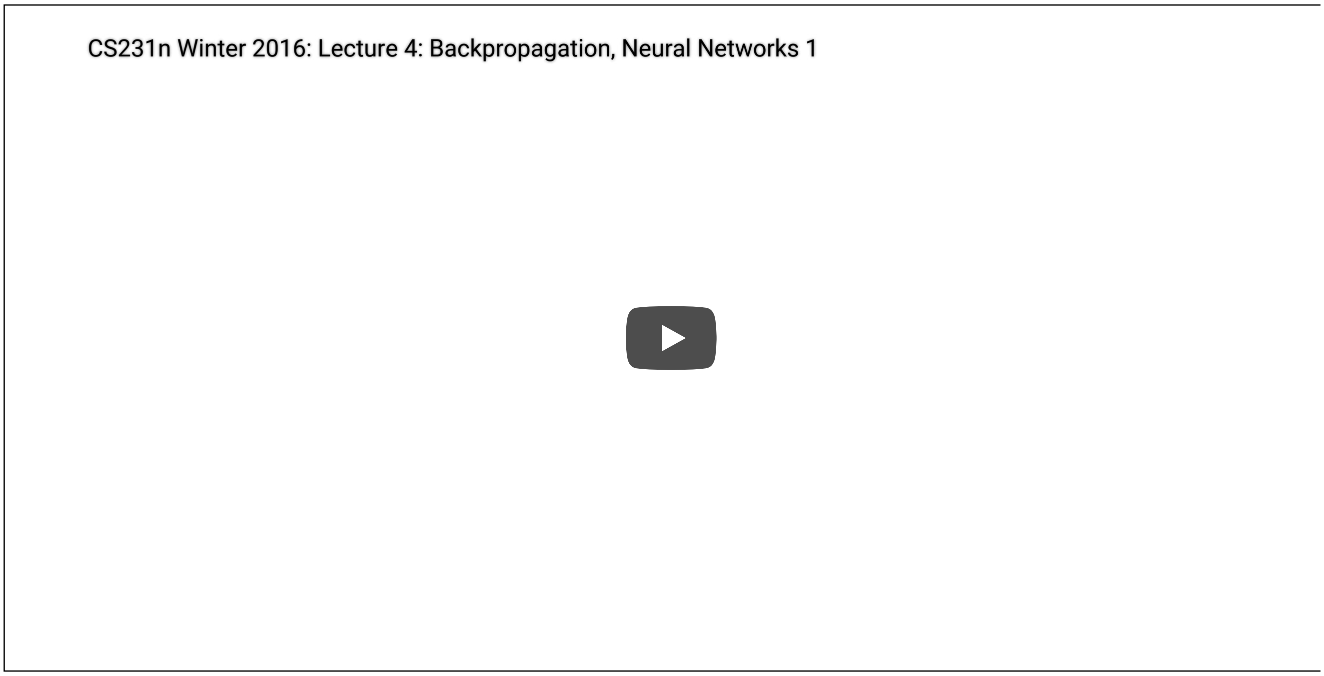
- **If you observe the graph, we are having input features [f1, f2, f3, f4, f5] and 9 weights [w1, w2, w3, w4, w5, w6, w7, w8, w9]**.
- **The final output of this graph is a value L which is computed as (Y-Y')^2**

Task 1: Implementing backpropagation and Gradient checking

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

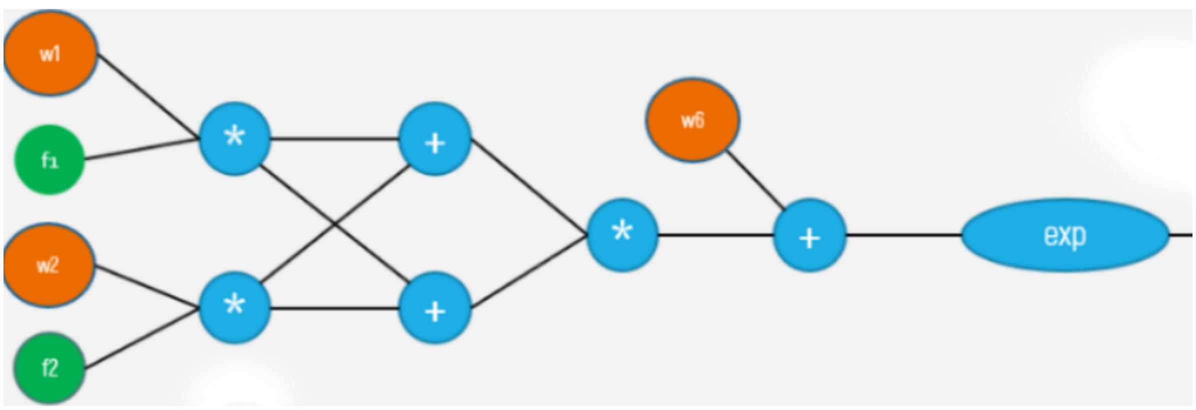
In [10]:
```
from IPython.display import YouTubeVideo
YouTubeVideo('i94OvYb6noo',width="1000",height="500")
```

Out[10]:



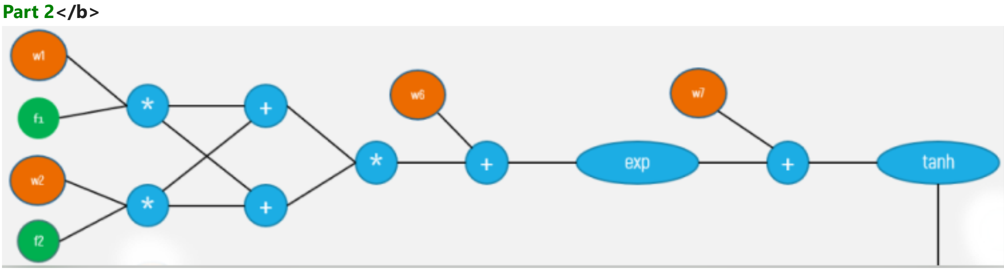
- **Write two functions**
    - **Forward propagation(Write your code in def forward_propagation())**

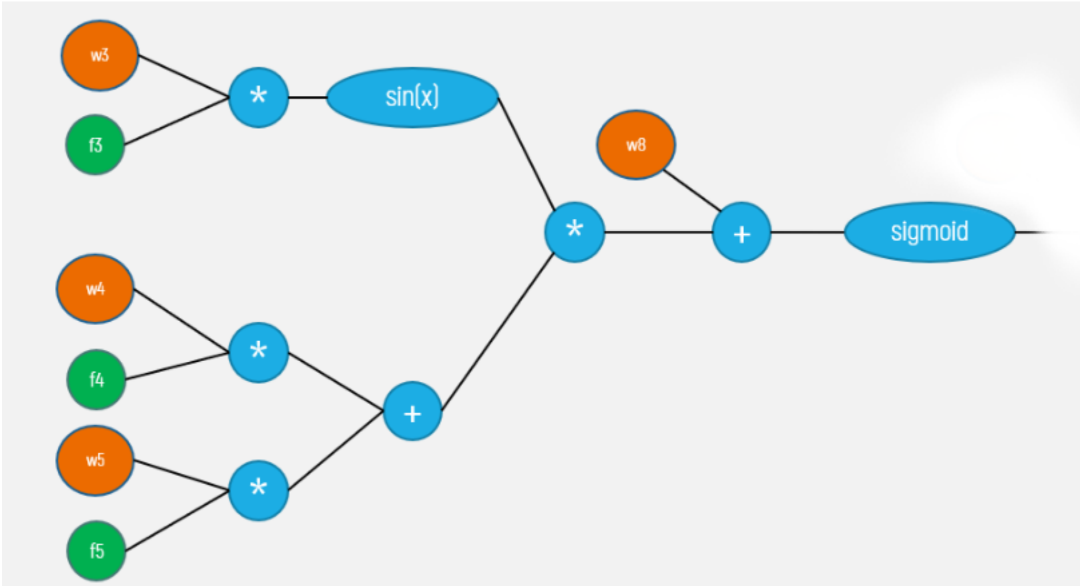
      For easy debugging, we will break the computational graph into 3 parts.



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-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] 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 values in tanh)
 # sig = part3(compute the forward propagation until sigmoid and then store the values 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** $\epsilon = 0.0001$

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

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()</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 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</font>
    approx_gradients.append(approximation gradients of weight wi)<font color='grey'>
# compare the gradient of weights W from backword_propagation() with the aproximation gradients 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
```

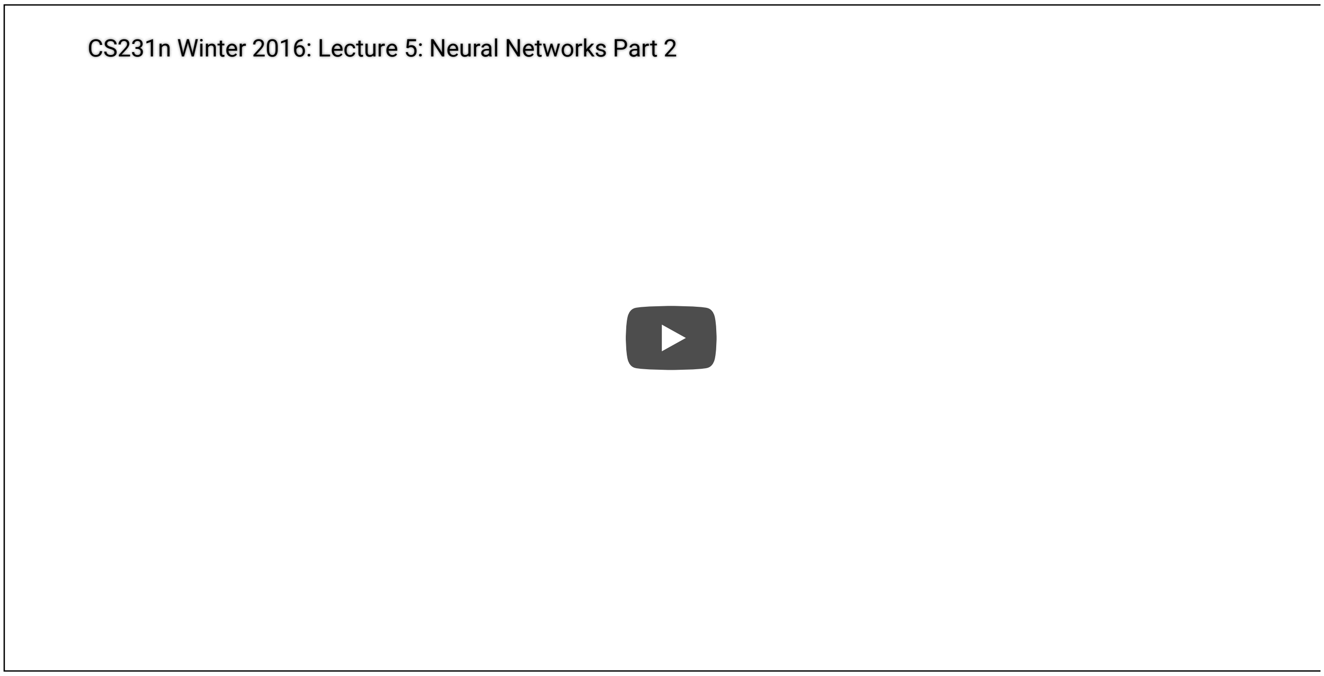
Task 2 : Optimizers

- **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 [11]:
```
from IPython.display import YouTubeVideo
YouTubeVideo('gYpoJMlgyXA',width="1000",height="500")
```

Out[11]:



Algorithm

```
for each epoch(1-100):
    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  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

Forward propagation

In [12]:
```
def sigmoid(z):
  sig = 1/(1 + math.exp(-z))
  return sig


import math
def forward_propagation(x, y, w,i):
  exp = math.exp(pow((w[0]*x[i][0] +w[1]*x[i][1]),2)+w[5])
  tanh = np.tanh(exp+w[6])
  a = (math.sin(w[2]*x[i][2]))*(w[3]*x[i][3]+w[4]*x[i][4]) + w[7]
  sig = sigmoid(a)
  y1 = sig*w[8] + tanh
  L = pow((y[i]-y1),2)
  dl = -2*(y[i]-y1)

  return {'dy_pr':dl,'loss':L,'exp':exp,'tanh':tanh,'sigmoid':sig}
```

Grader function - 1

In [13]:
```
def grader_sigmoid(z):
  val=sigmoid(z)
  assert(val==0.8807970779778823)
  return True

grader_sigmoid(2)
```

Out[13]: True

Grader function - 2

In [14]:
```
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,y,w,0)
grader_forwardprop(d1)
```

Out[14]: True

In [15]:
```
print(d1)
print(d1['loss'])
```

```
{'dy_pr': -1.9285278284819143, 'loss': 0.9298048963072919, 'exp': 1.1272967040973583, 'tanh': 0.8417934192562146, 'sigmoid': 0.5279179387
419721}
0.9298048963072919
```

Backward propagation

In [16]:
```
def backward_propagation(x,w,dict,i):
  z = w[0]*x[i][0] +w[1]*x[i][1]
  t = w[3]*x[i][3]+w[4]*x[i][4]


  dw1 = (dict['dy_pr']*(1-pow(dict['tanh'],2))*dict['exp']*4*z*x[i][0])/2
  dw2 = (dict['dy_pr']*(1-pow(dict['tanh'],2))*dict['exp']*4*z*x[i][1])/2
  dw3 = dict['dy_pr']*w[8]*dict['sigmoid']*(1-dict['sigmoid'])*t*x[i][2]*math.cos(w[2]*x[i][2])
  dw4 = dict['dy_pr']*w[8]*dict['sigmoid']*(1-dict['sigmoid'])*x[i][3]*math.sin(w[2]*x[i][2])
  dw5 = dict['dy_pr']*w[8]*dict['sigmoid']*(1-dict['sigmoid'])*x[i][4]*math.sin(w[2]*x[i][2])
  dw7 = dict['dy_pr']*(1-pow(dict['tanh'],2))
  dw6 = dw7*dict['exp']
  dw8 = dict['dy_pr']*w[8]*dict['sigmoid']*(1-dict['sigmoid'])
  dw9 = dict['dy_pr']*dict['sigmoid']


  return {'dw1':dw1, 'dw2':dw2, 'dw3':dw3, 'dw4':dw4, 'dw5':dw5, 'dw6':dw6, 'dw7':dw7, 'dw8':dw8, 'dw9':dw9}
```

In [17]:
```
w=np.ones(9)*0.1
print(w)
```

```
[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
```

In [18]:
```
w=np.ones(9)*0.1
d1=forward_propagation(x,y,w,0)
a = backward_propagation(x,w,d1,0)
print(a)
```

```
{'dw1': -0.22973323498702, 'dw2': -0.02140761471775293, 'dw3': -0.005625405580266319, 'dw4': -0.004657941222712423, 'dw5': -0.00100772284
98574246, 'dw6': -0.6334751873437471, 'dw7': -0.561941842854033, 'dw8': -0.04806288407316516, 'dw9': -1.0181044360187037}
```

Grader function - 3

In [19]:
```
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,y,w,0)
d1=backward_propagation(x,w,d1,0)
grader_backprop(d1)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-19-18f8c9862680> in <module>()
     14 d1=forward_propagation(x,y,w,0)
     15 d1=backward_propagation(x,w,d1,0)
---> 16 grader_backprop(d1)
```

```
<ipython-input-19-18f8c9862680> in grader_backprop(data)
      9     dw8=(data['dw8']==-0.04806288407316516)
     10     dw9=(data['dw9']==-1.0181044360187037)
---> 11     assert(dw1 and dw2 and dw3 and dw4 and dw5 and dw6 and dw7 and dw8 and dw9)
     12     return True
     13 w=np.ones(9)*0.1

AssertionError:
```

Implement gradient checking

In [20]:
```
def forward_propagation(x, y, w,i):
  exp = math.exp(pow((w[0]*x[i][0] +w[1]*x[i][1]),2)+w[5])
  tanh = np.tanh(exp+w[6])
  a = (math.sin(w[2]*x[i][2]))*(w[3]*x[i][3]+w[4]*x[i][4]) + w[7]
  sig = sigmoid(a)
  y1 = sig*w[8] + tanh
  L = pow((y[i]-y1),2)
  dl = -2*(y[i]-y1)

  return {'dy_pr':dl,'loss':L,'exp':exp,'tanh':tanh,'sigmoid':sig, 'y_pred':y1}
```

In [41]:
```
def gradient_checking(x,y,W,k):
  dict = forward_propagation(x, y, W,k)
  dict1 = backward_propagation(x,W,dict,k)
  dict1 = [i for i in dict1.values()]
  print(dict1)
  approx_gradients = []

  gradient_check = []
  for  i,wi in enumerate(W):
    noise1 = wi + 0.0001
    W[i] = noise1
    di1 = forward_propagation(x, y, W,k)
    noise2 = wi - 0.0001
    W[i] = noise2
    di2 = forward_propagation(x, y, W,k)
    W = np.ones(9)*0.1
    dw_approx = (di1['loss']-di2['loss'])/(2*0.0001)



    approx_gradients.append(dw_approx)
  print(approx_gradients)


  for i in range(len(W)):
    gradient_check.append((np.linalg.norm(dict1[i] - approx_gradients[i]))/(np.linalg.norm(dict1[i])+np.linalg.norm(approx_gradients[i])))

  return gradient_check
```

In [42]:
```
W = np.ones(9)*0.1
gradient_check = gradient_checking(x,y,W,0)
print(gradient_check)
```

```
[-0.22973323498702, -0.02140761471775293, -0.005625405580266319, -0.004657941222712423, -0.0010077228498574246, -0.6334751873437471, -0.5
61941842854033, -0.04806288407316516, -1.0181044360187037]
[-0.22973323022201786, -0.02140761471536301, -0.0056254055608162545, -0.004657941222729889, -0.0010077228507210378, -0.6334751863795729,
-0.5619418463920223, -0.0480628840343611, -1.0181044360180191]
[1.0370728885929153e-08, 5.581934643713204e-11, 1.7287700041112022e-09, 1.87486944153289e-12, 4.2849738752544037e-10, 7.610196933782967e-
10, 3.1480030084674753e-09, 4.0368014625577295e-10, 3.361951351774315e-13]
```

Task 2: Optimizers

Algorithm with Vanilla update of weights

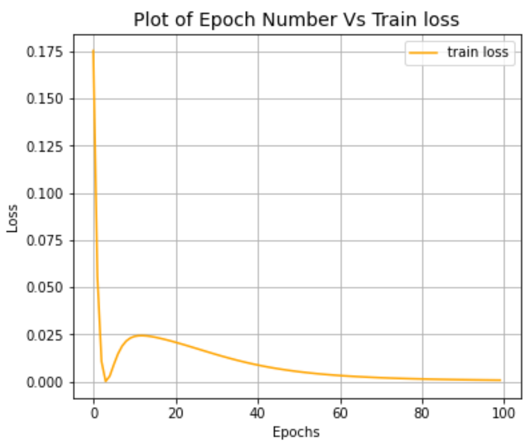
In [43]:
```
import matplotlib.pyplot as plt
def graph(train_loss,num):
  epoch = np.arange(num)
  plt.figure( figsize=(6,5))
  plt.grid()
  plt.plot(epoch,train_loss,color='orange')
  plt.xlabel("Epochs")
  plt.ylabel("Loss")
  plt.title('Plot of Epoch Number Vs Train loss',fontsize = 14)
  plt.legend(['train loss'])
  plt.show()
```

In [44]:
```
def initialize_weights():
  w=np.ones(9)*0.1
  return w

def train(X_train,y_train,epochs,alpha):
  train_loss = list()
  w= initialize_weights()
  for i in range(epochs):
    for j in range(len(X_train)):
```

```
            dict = forward_propagation(X_train, y_train, w,j)
            dw = backward_propagation(X_train,w,dict,j)
            dw = [k for k in dw.values()]
            w = [w[l] - alpha*m for l,m in enumerate(dw)]
        dict = forward_propagation(X_train, y_train, w,j)
        train_loss.append(dict['loss'])


    return train_loss
```

Plot between epochs and loss

In [45]:
```
train_loss1 = train(x,y,100,0.001)
graph(train_loss1,100)
```



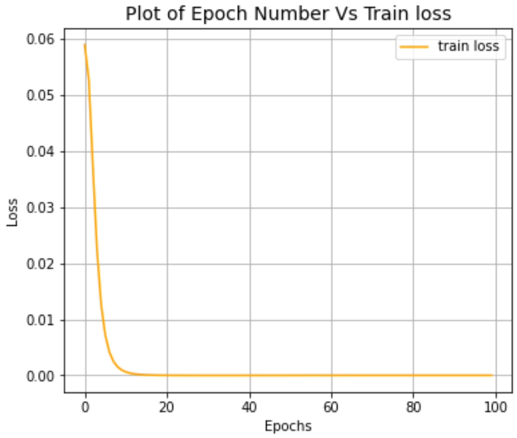
Algorithm with Momentum update of weights

In [82]:
```
def initialize_weights():
    w=np.ones(9)*0.1
    return w

def train(X_train,y_train,epochs,alpha,beta):
    m = np.zeros(9)
    #m = np.ones(9)
    train_loss = list()
    w= initialize_weights()
    for i in range(epochs):
        for j in range(len(X_train)):
            dict = forward_propagation(X_train, y_train, w,j)
            dw = backward_propagation(X_train,w,dict,j)
            dw = np.array([k for k in dw.values()])
            w = [w[l] - alpha*n for l,n in enumerate(m)]
            #w = [w[l] - alpha*m for l,m in enumerate(dw)]
            #w = [w[l] - n for l,n in enumerate(m)]
            m = [ beta*m[o] - (1-beta)*p for o,p in enumerate(dw)]
            #m = m*beta - alpha*dw
        dict = forward_propagation(X_train, y_train, w,j)
        train_loss.append(dict['loss'])


    return train_loss
```

In [91]:
```
def initialize_weights():
    w=np.ones(9)*0.1
    return w

def train(X_train,y_train,epochs,alpha,gamma):
    v = np.zeros(9)
    #m = np.ones(9)
    train_loss = list()
    w= initialize_weights()
    for i in range(epochs):
        for j in range(len(X_train)):
            dict = forward_propagation(X_train, y_train, w,j)
            dw = backward_propagation(X_train,w,dict,j)
            dw = np.array([k for k in dw.values()])
            v = v*gamma + alpha*dw
            w = w - v
        dict = forward_propagation(X_train, y_train, w,j)
        train_loss.append(dict['loss'])


    return train_loss
```

Plot between epochs and loss

In [92]:
```
train_loss2 = train(x,y,100,0.001,0.9)
graph(train_loss2,100)
```

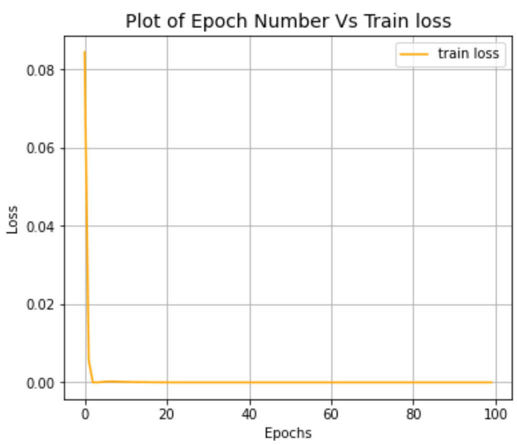


Algorithm with adam update of weights

In [96]:
```
def initialize_weights():
  w=np.ones(9)*0.1
  return w

def train(X_train,y_train,epochs,alpha,beta1,beta2,epsilon):
  m = [0,0,0,0,0,0,0,0,0]
  v = [0,0,0,0,0,0,0,0,0]
  train_loss = list()
  w= initialize_weights()
  power = 1
  for i in range(epochs):
    for j in range(len(X_train)):
      dict = forward_propagation(X_train, y_train, w,j)
      dw = backward_propagation(X_train,w,dict,j)
      dw = [k for k in dw.values()]
      m = [ beta1*g + (1-beta1)*dw[f] for f,g in enumerate(m)]
      m_ = [h/(1-pow(beta1,power)) for h in m]
      v = [ beta2*r + (1-beta2)*pow(dw[q],2) for q,r in enumerate(v)]
      v_ = [s/(1-pow(beta2,power)) for s in v]
      w = [w[l] - alpha*m_[l]*(1/(math.sqrt(v_[l]+epsilon))) for l in range(len(m))]
      power += 1
    dict = forward_propagation(X_train, y_train, w,j)
    train_loss.append(dict['loss'])


  return train_loss
```

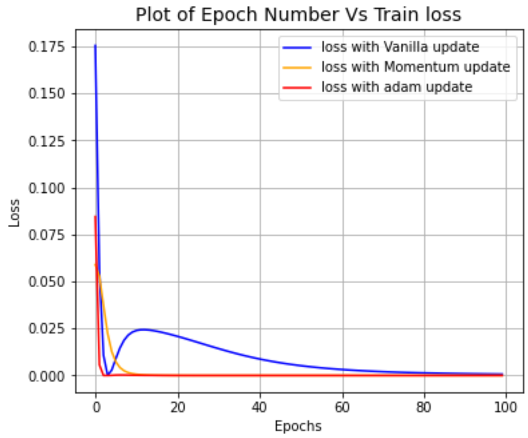
Plot between epochs and loss

In [95]:
```
train_loss3  = train(x,y,100,0.001,0.9,0.999,0.000001)
graph(train_loss3,100)
```



Comparision plot between epochs and loss with different optimizers

In [98]:
```
epoch = np.arange(100)
plt.figure( figsize=(6,5))
plt.grid()
plt.plot(epoch,train_loss1,color='blue')
plt.plot(epoch,train_loss2,color='orange')
plt.plot(epoch,train_loss3,color='red')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title('Plot of Epoch Number Vs Train loss',fontsize = 14)
plt.legend(['loss with Vanilla update','loss with Momentum update','loss with adam update'])
plt.show()
```



Observation

1. **In vanilla update, it is found that loss is more at epoch 1 and droping in next epoch. 2.In Momentum update, loss is comparitively found to be less as compare to vanilla update but converse fast. 3.In adam update , loss is small as compare to both of them and took few epoch extra to converse.**