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.

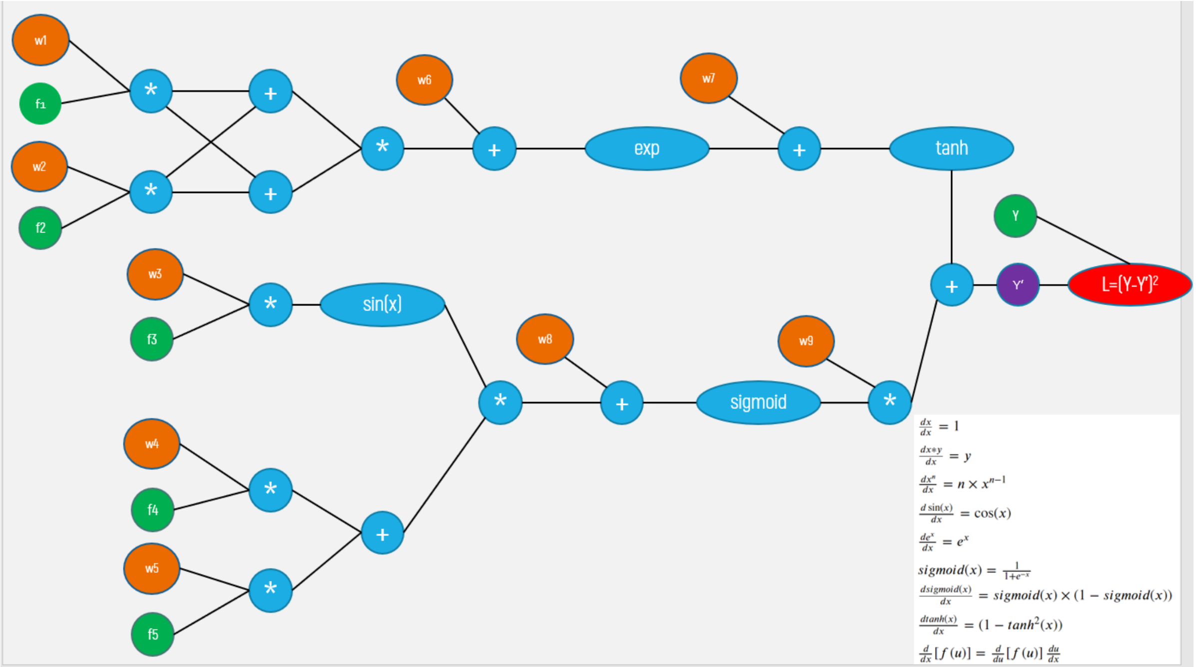
Every Grader function has to return True.

Loading data

In [1]:
```
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,)
```

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].**

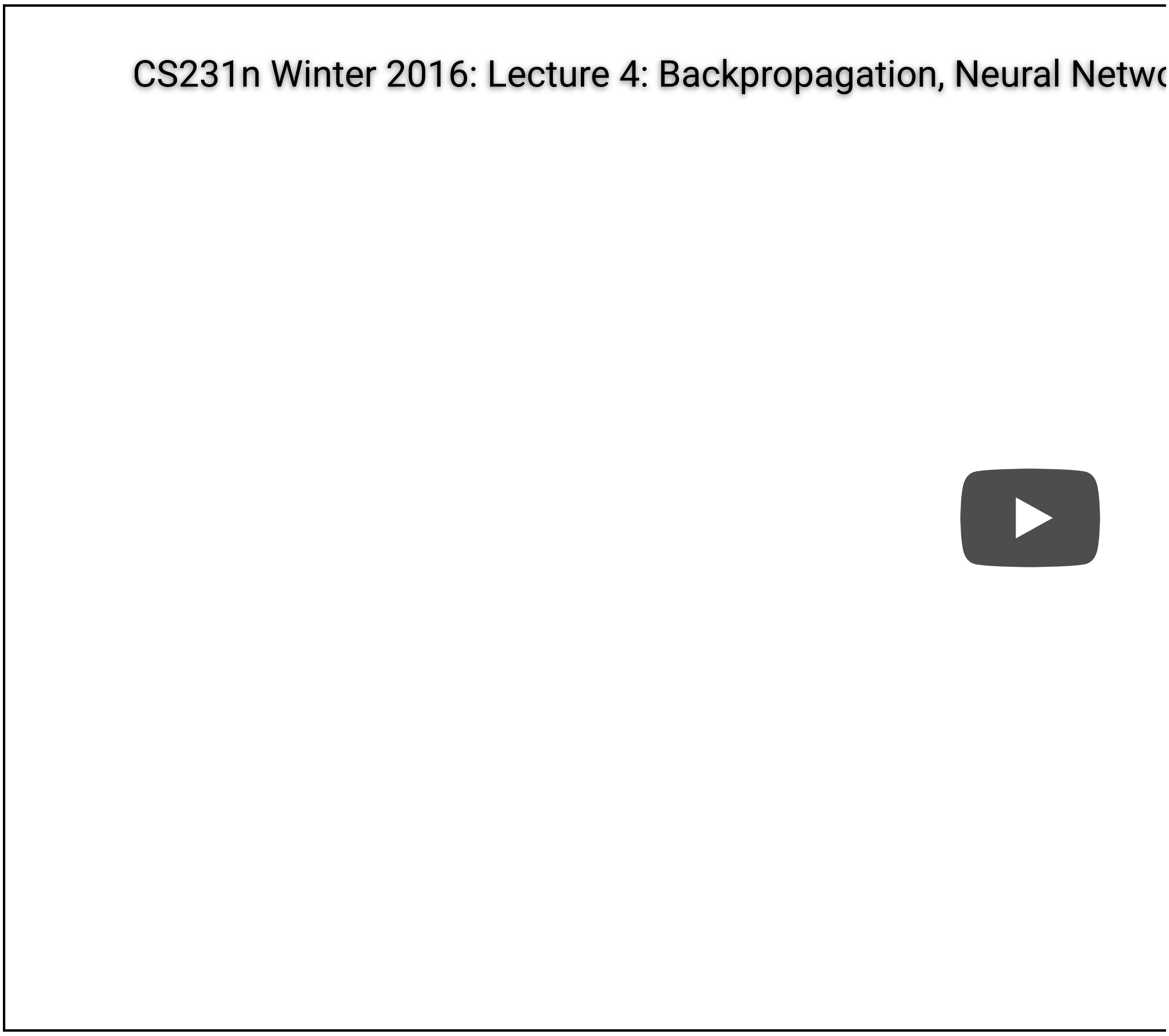
- **The final output of this graph is a value L which is computed as (Y-Y')^2**

Task 1: Implementing backpropagation and Gradient checking

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

In [2]:
```
from IPython.display import YouTubeVideo
YouTubeVideo('i94OvYb6noo',width="1000",height="500")
```

Out[2]:

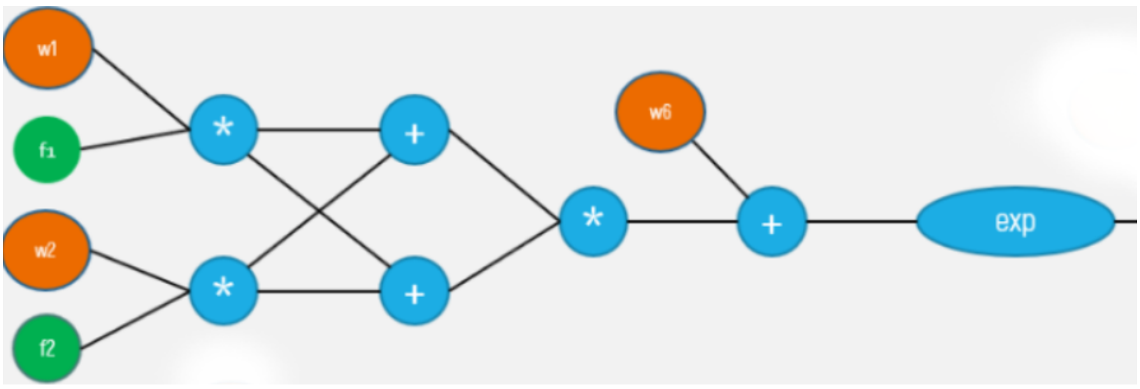


- **Write two functions**
  - **Forward propagation**

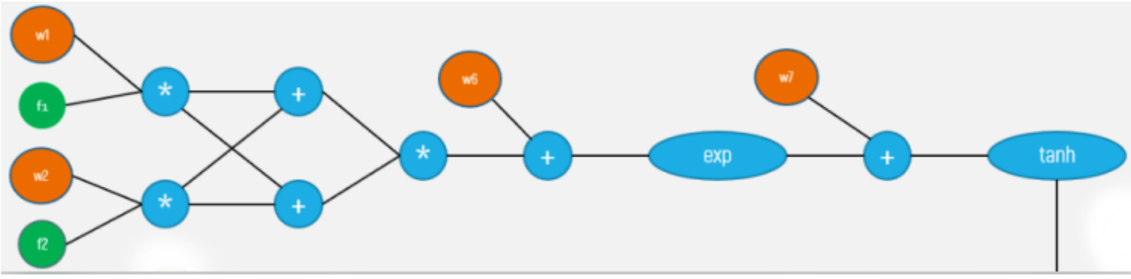
  (Write your code in def forward_propagation())

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

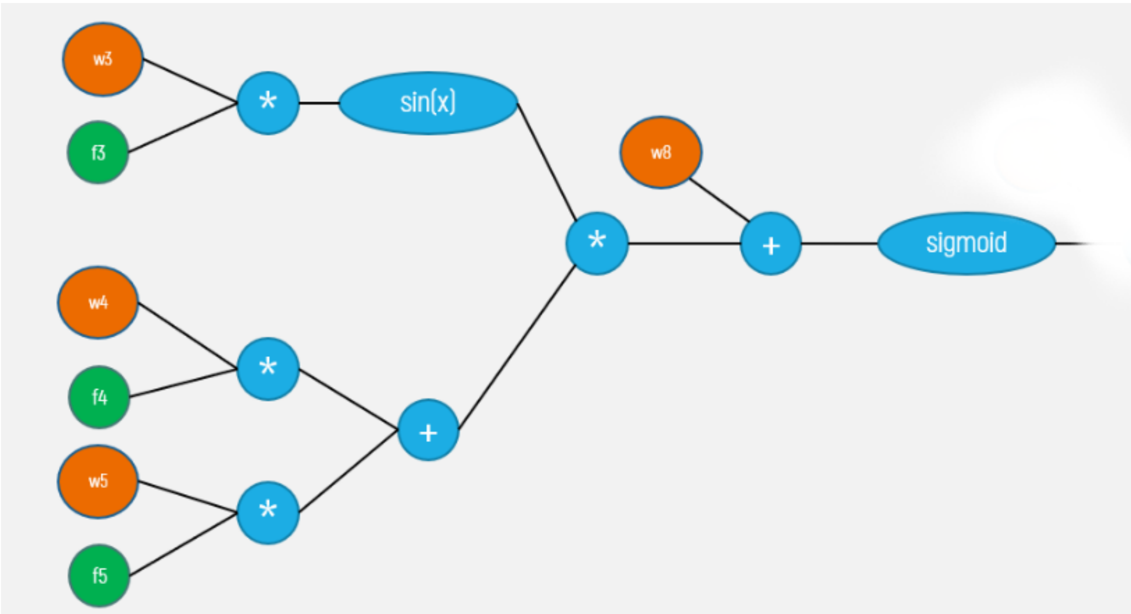
  Part 1



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 tan
h and then store the values in tanh)
 # sig = part3(compute the forward propagation until sig
moid and then store the values in sig)
 # now compute remaining values from computional graph a
nd 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 val
ues
 # store L, exp,tanh,sig,dl variables

 return (dictionary, which you might need to use for bac
k 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 [w
1,w2,w3,...,w9]
 # Hint: you can use dict type to store the required var
iables
 # return dW, dW is a dictionary with gradients of all t
he weights

 return dW
```

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 updated weights
        # subtract a small value to weight wi, and then fin
d 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_propa
gation() with the aproximation gradients of weights with
  gradient_check formula
    return gradient_check
NOTE: you can do sanity check by checking all the return va
lues of gradient_checking(),
 they have to be zero. if not you have bug in your code
```

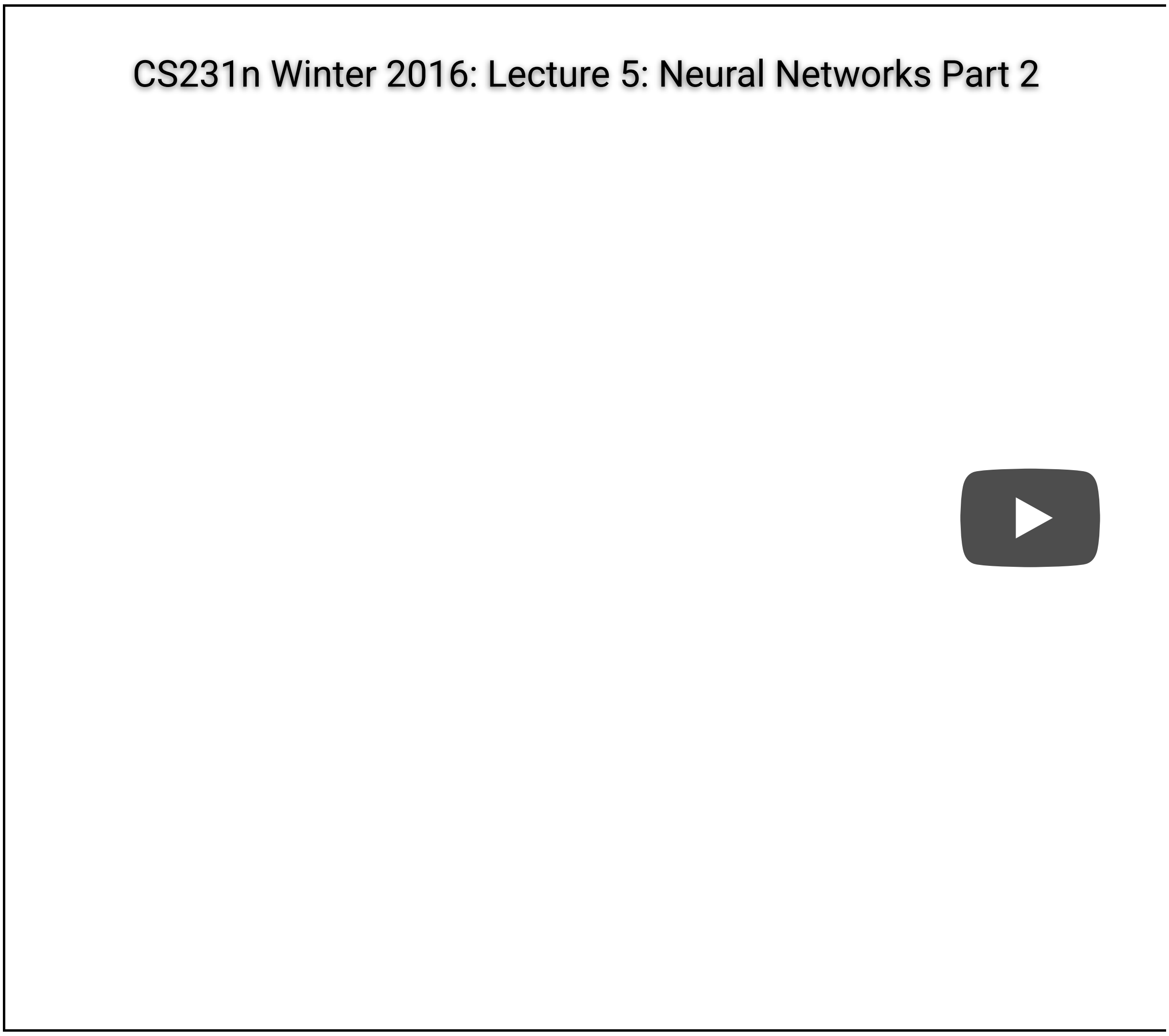
Task 2 : Optimizers

- 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 (https://cs231n.github.io/neural-networks-3/)](https://cs231n.github.io/neural-networks-3/) blog

In [3]:
```
from IPython.display import YouTubeVideo
YouTubeVideo('gYpoJMlgyXA',width="1000",height="500")
```

Out[3]:



Algorithm

```
    for each epoch(1-100):
        for each data point in your data:
            using the functions forward_propagation() and b
ackword_propagation() compute the gradients of weights
            update the weigts with help of gradients  ex: w
1 = 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

In [4]:
```
import math
from numpy import linalg as LA
from math import sqrt
```

Forward propagation

In [5]:
```
def sigmoid(z):
    '''In this function, we will compute the sigmoid(z)'''
    # we can use this function in forward and backward propagation
    sig = 1/(1 + math.exp(-z))
    return sig

def forward_propagation(x, y, w):
        '''In this function, we will compute the forward propagatio
        # X: input data point, note that in this assignment you are
        # y: output varible
        # W: weight array, its of length 9, W[0] corresponds to w1
        # you have to return the following variables
        # exp= part1 (compute the forward propagation until exp and
        # tanh =part2(compute the forward propagation until tanh an
        # sig = part3(compute the forward propagation until sigmoid
        # now compute remaining values from computional graph and g
        # 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 variables

        d = dict()
        exp = math.exp(((w[0]*x[0] + w[1]*x[1])**2) + w[5])
        tanh = math.tanh(exp + w[6])
        sig = sigmoid(((math.sin(w[2]*x[2])) * ((w[3]*x[3])+(w[4]*x
        y_dash = tanh + (sig*w[8])
        L = (y - y_dash)**2
        dL = -2 * (y - y_dash)
        d['exp'] = exp
        d['tanh'] = tanh
        d['sigmoid'] = sig
        d['y_dash'] = y_dash
        d['loss'] = L
        d['dy_pr'] = dL
        return d

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

Grader function - 1

In [6]:
```
def grader_sigmoid(z):
  val=sigmoid(z)
  assert(val==0.8807970779778823)
  return True
grader_sigmoid(2)
```

Out[6]: True

Grader function - 2

In [7]:
```python
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[7]:
```
True
```

Backward propagation

In [8]:
```
def backward_propagation(L,W,d):
    '''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,
    # 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
    d1 = dict()
    dL = d['dy_pr']
    dw7 = dL * (1 - (d['tanh'])**2)
    dw6 = dw7 * d['exp']
    dw9 = d['dy_pr'] * d['sigmoid']
    dw8 = dL * W[8] * d['sigmoid'] * (1 - d['sigmoid'])
    dw5 = dw8 * L[4] * math.sin(W[2]*L[2])
    dw4 = dw8 * L[3] * math.sin(W[2]*L[2])
    dw3 = dw8 * math.cos(W[2]*L[2]) * ((W[3]*L[3])+(W[4]*L[4])) * L
    dw2 = dw6 * 2 * ((W[0]*L[0]) + (W[1]*L[1])) * L[1]
    dw1 = dw6 * 2 * ((W[0]*L[0]) + (W[1]*L[1])) * L[0]
    d1['dw1'] = dw1
    d1['dw2'] = dw2
    d1['dw3'] = dw3
    d1['dw4'] = dw4
    d1['dw5'] = dw5
    d1['dw6'] = dw6
    d1['dw7'] = dw7
    d1['dw8'] = dw8
    d1['dw9'] = dw9
    return d1
    # return dW, dW is a dictionary with gradients of all the weigh
```

Grader function - 3

In [9]:
```
def grader_backprop(data):
    ## changed last 2 to 3 digits in dw1 and dw2
    dw1=(data['dw1']==-0.22973323498702)  ## -0.22973323498702003 i
    dw2=(data['dw2']==-0.021407614717752 93)  ## -0.0214076147177529
    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
    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

In [10]:
```
d1
```

Out[10]:
```
{'dw1': -0.22973323498702,
 'dw2': -0.021407614717752 93,
 'dw3': -0.005625405580266319,
 'dw4': -0.004657941222712423,
 'dw5': -0.0010077228498574246,
 'dw6': -0.6334751873437471,
 'dw7': -0.561941842854033,
 'dw8': -0.04806288407316516,
 'dw9': -1.0181044360187037}
```

In [11]:
```
weights = np.array(list(d1.values()))
weights[0]
```

Out[11]: -0.22973323498702

Implement gradient checking

In [12]:
```
W = np.ones(9)*0.1
def gradient_checking(X, Y, W):
    # compute the L value using forward_propagation()
    d1 = forward_propagation(X,Y,W)

    # compute the gradients of W using backword_propagation()
    d2 = backward_propagation(X,W,d1)

    approx_gradients = []
    for i in range(len(W)):
        # add a small value to weight wi, and then find the values
        temp = W[i]
        W[i] = temp + 0.0001
        loss1 = forward_propagation(X,Y,W)
        # subtract a small value to weight wi, and then find the va
        W[i] = temp - 0.0001
        loss2 = forward_propagation(X,Y,W)
        # compute the approximation gradients of weight wi
        approx_grad = (loss1['loss'] - loss2['loss'])/0.0002
        W[i] = temp
        approx_gradients.append(approx_grad)
    # compare the gradient of weights W from backword_propagation()

    gradients = np.array(list(d2.values()))

    num = LA.norm(gradients - approx_gradients)
    den = LA.norm(gradients) + LA.norm(approx_gradients)

    gradient_check = num/den

    return gradient_check
```

In [13]:
```
gradient_check = []
for i in range(len(X)):
    check = gradient_checking(X[i], y[i], W)
    gradient_check.append(check)
```

In [14]:
```
gradient_check[0]
```

Out[14]: 2.2352076630799546e-09

Task 2: Optimizers

###Algorithm with Vanilla update of weights

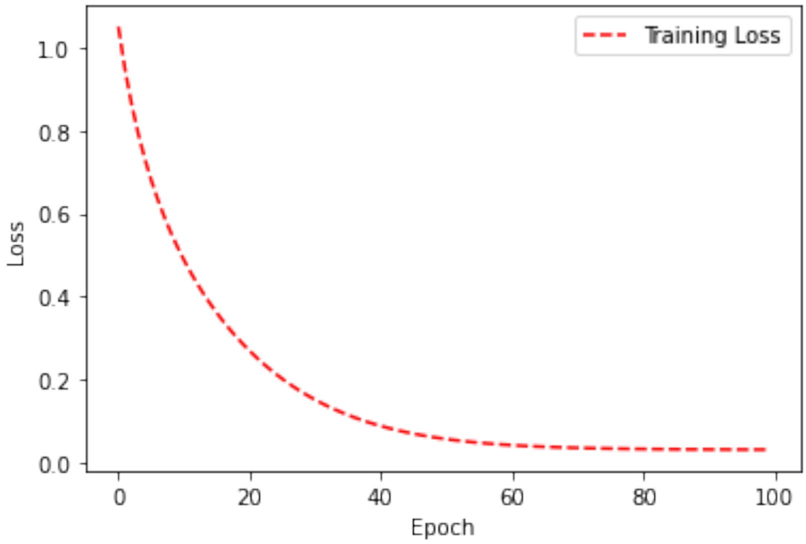
for each epoch(1-100): for each data point in your data: using the functions forward_propagation() and backword_propagation() compute the gradients of weights update the weigts with help of gradients ex: w1 = w1-learning_rate*dw1

In [15]:
```
mu, sigma = 0, 0.1
W = np.random.normal(mu, sigma, size=9)
```

In [16]:
```
learning_rate = 0.0001
train_loss = []
W1 = W.copy()
n = len(X)
for i in range(100):
    loss = []
    for j in range(n):
        d1 = forward_propagation(X[j],y[j],W1)
        d2 = backward_propagation(X[j],W1,d1)

        loss.append(d1['loss'])
        weights = np.array([b for (a,b) in d2.items()])

        W1 -= learning_rate * weights
    train_loss.append(np.average(loss))
```

Plot between epochs and loss

In [17]:
```
import matplotlib.pyplot as plt
epoch_count = range(0, 100)
plt.plot(epoch_count, train_loss, 'r--')
plt.legend(['Training Loss'])
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show();
```

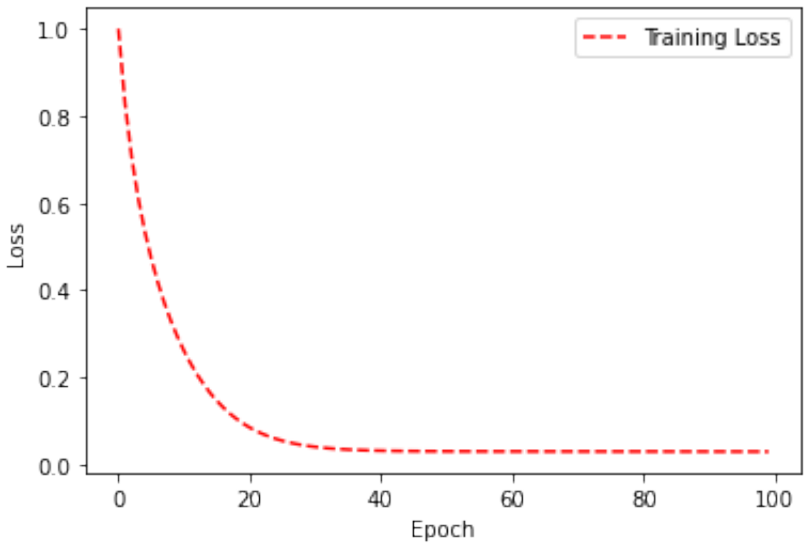


###Algorithm with Momentum update of weights

In [18]:
```
learning_rate = 0.0001
W2 = W.copy()
m = 0.5
v = np.zeros(9)
n = len(X)
train2_loss = []
for i in range(100):
    loss = []
    for j in range(0,n):
        d1 = forward_propagation(X[j],y[j],W2)
        d2 = backward_propagation(X[j],W2,d1)
        loss.append(d1['loss'])

        weights = np.array([b for (a,b) in d2.items()])

        v = (m*v) - (learning_rate*weights)
        W2 += v
    train2_loss.append(np.average(loss))
```

Plot between epochs and loss

In [19]:
```
import matplotlib.pyplot as plt
epoch_count = range(0, 100)
plt.plot(epoch_count, train2_loss, 'r--')
plt.legend(['Training Loss'])
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show();
```



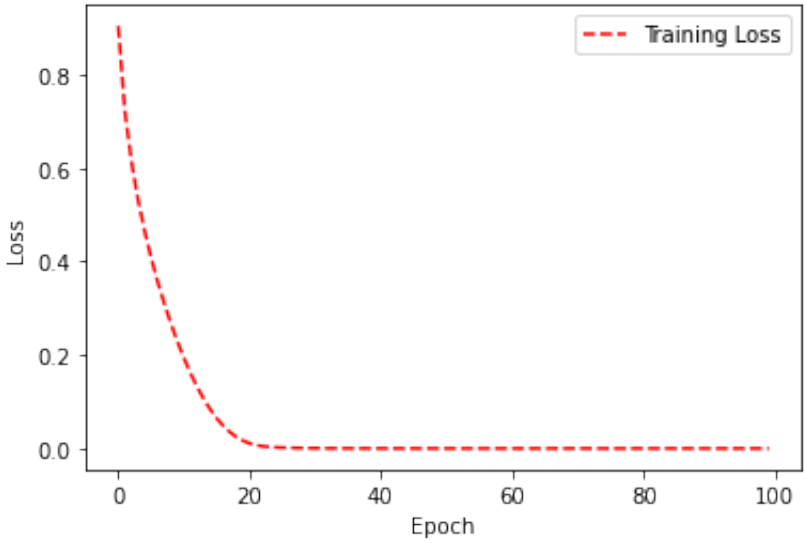
###Algorithm with Adam update of weights

In [20]:
```
learning_rate = 0.0001
W3 = W.copy()
epsilon = 1.0*np.exp(-8)
beta1 = 0.9
beta2 = 0.999
m = v = np.zeros(9)
print(v.shape, m.shape)
n = len(X)
train3_loss = []
for i in range(0,100):
    loss = []
    for j in range(0,n):
        d1 = forward_propagation(X[j],y[j],W3)
        loss.append(d1['loss'])
        d2 = backward_propagation(X[j],W3,d1)

        weights = np.array([b for (a,b) in d2.items()])

        m = (beta1*m) + ((1-beta1)*weights)
        v = (beta2*v) + ((1-beta2)*(weights**2))

        W3 -= (learning_rate * (m / (np.sqrt(v) + epsilon)))
    train3_loss.append(np.average(loss))
```

```
(9,) (9,)
```

Plot between epochs and loss

In [21]:
```
import matplotlib.pyplot as plt
epoch_count = range(0, 100)
plt.plot(epoch_count, train3_loss, 'r--')
plt.legend(['Training Loss'])
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show();
```



Comparision plot between epochs and loss with different optimizers

In [22]:
```
import matplotlib.pyplot as plt
epoch_count = range(0, 100)
plt.plot(epoch_count, train_loss, 'b-')
plt.plot(epoch_count, train2_loss, 'r-')
plt.plot(epoch_count, train3_loss, 'g-')
plt.legend(['SGD Loss', 'SGD + Momentum Loss', 'Adam Loss'])
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Comparison of optimizers with learning rate = 0.0001')
plt.show();
```