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.

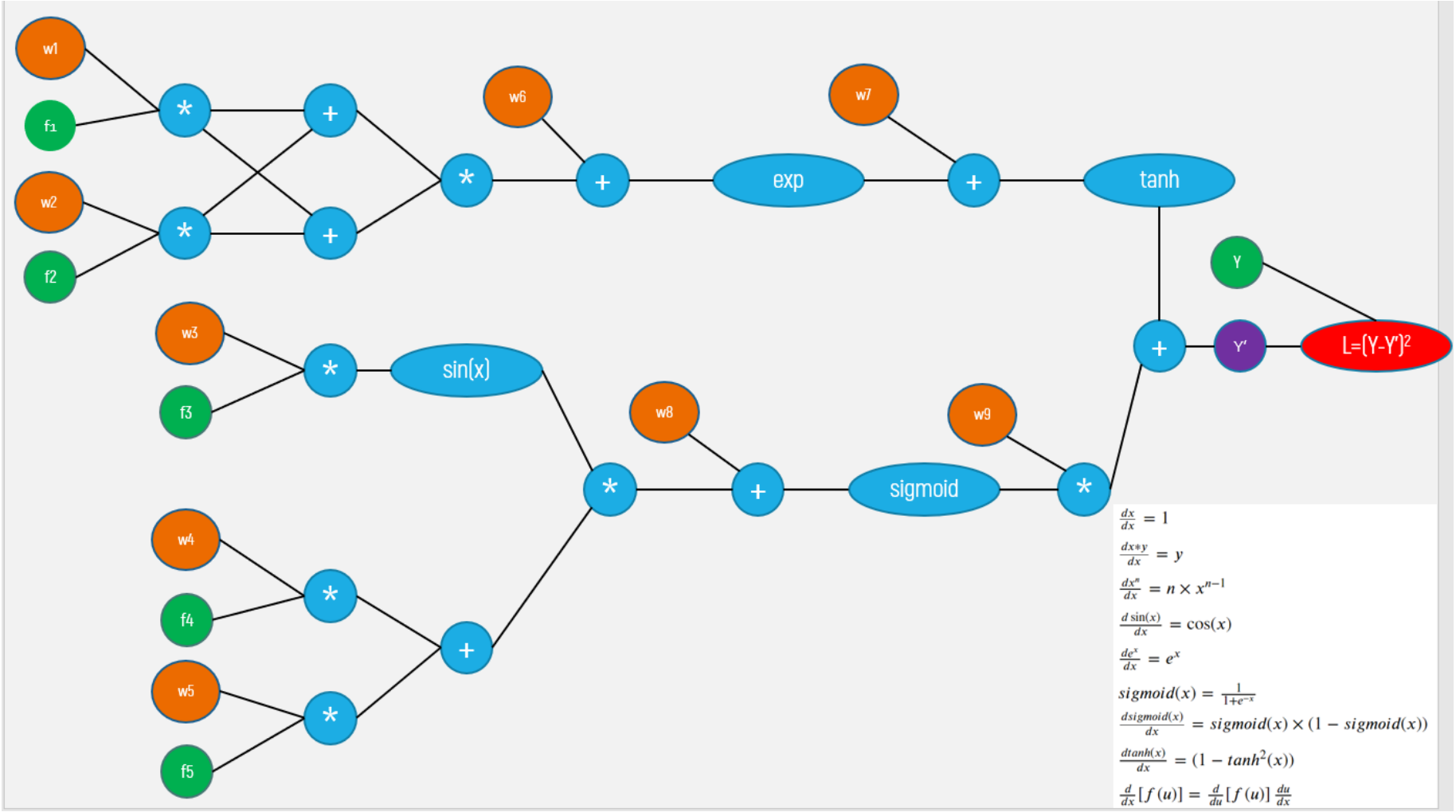
Every Grader function has to return True.

Loading data

In [1]:
```
# import necessary libraries
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)

# check the shape of the dataset
print(data.shape)
X = data[:, :5]
y = data[:, -1]
print(X.shape, y.shape)
```

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

Computational graph



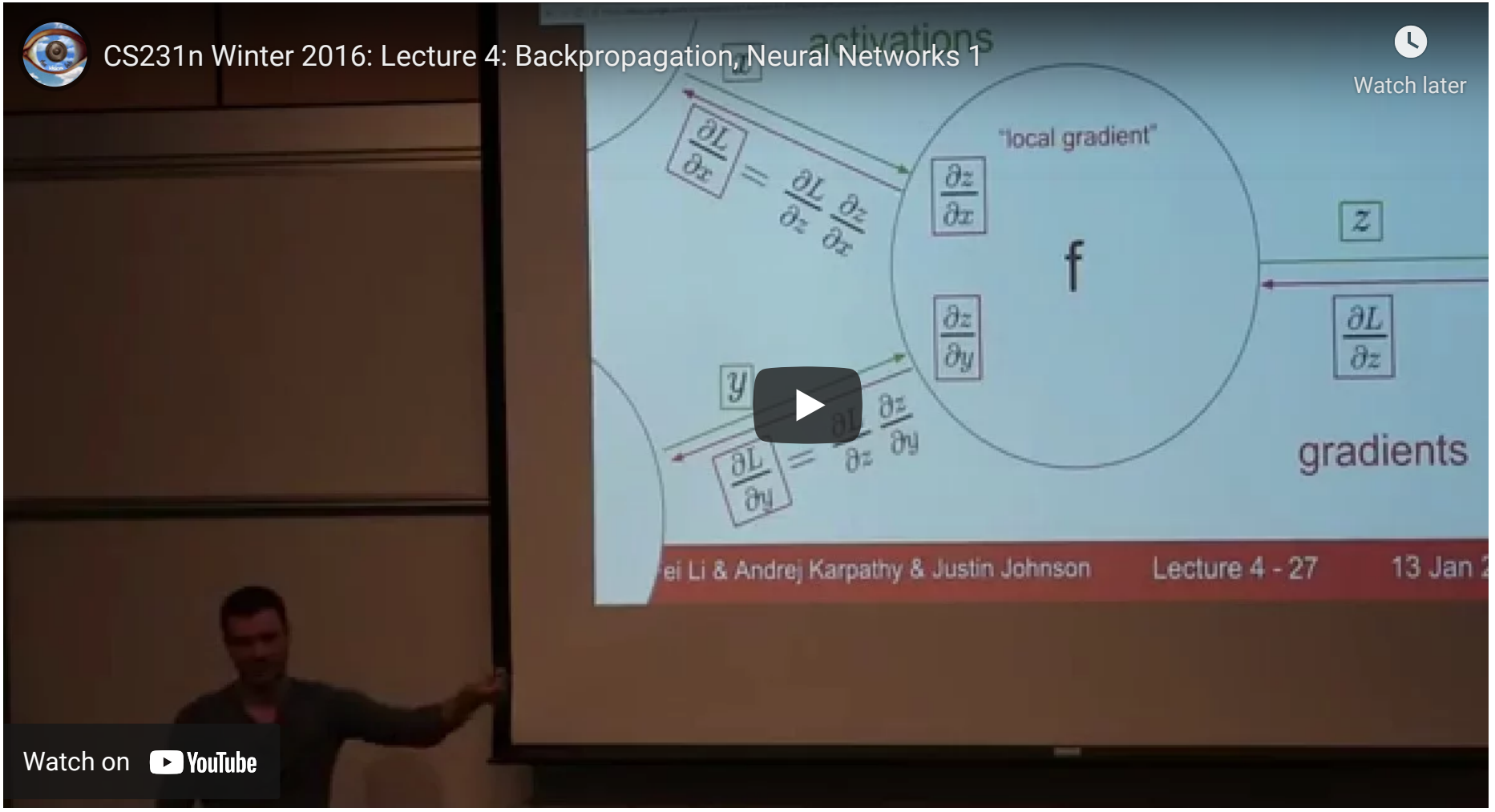
- **If you observe the graph, we are having input features [f1, f2, f3, f4, f5] and 9 weights [w1, w2, w3, w4, w5, w6, w7, w8, w9].**
- **The final output of this graph is a value L which is computed as (Y-Y')^2**

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('i940vYb6noo',width="1000",height="500")
```

Out[2]:

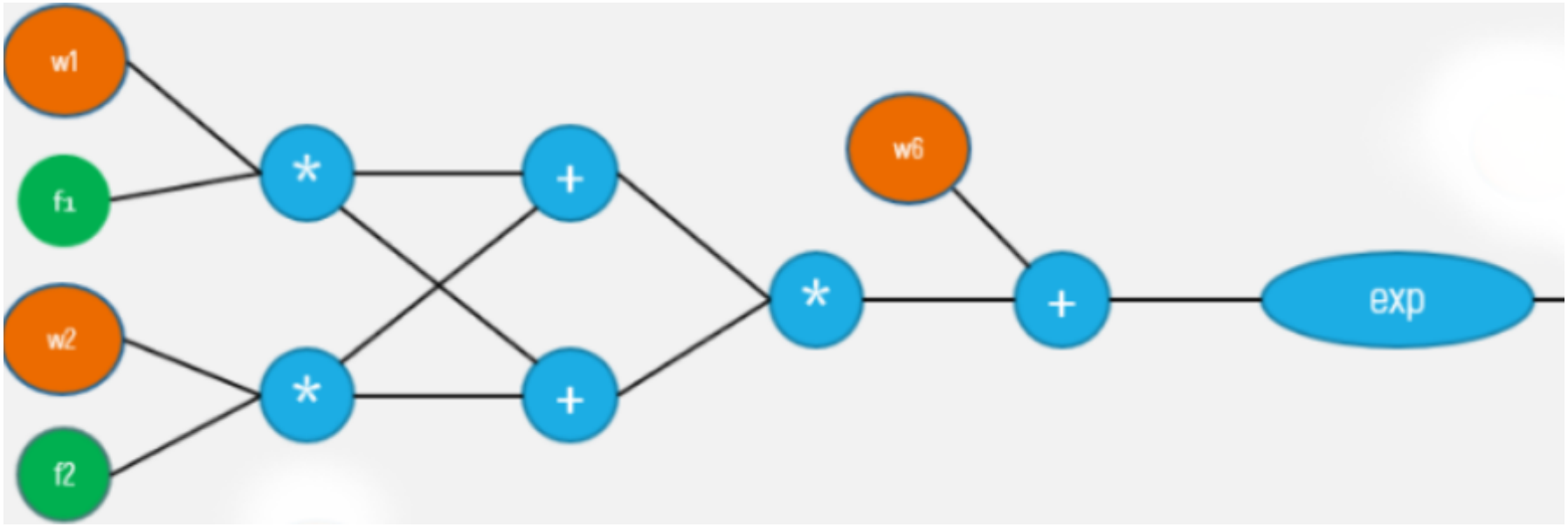


- **Write two functions**

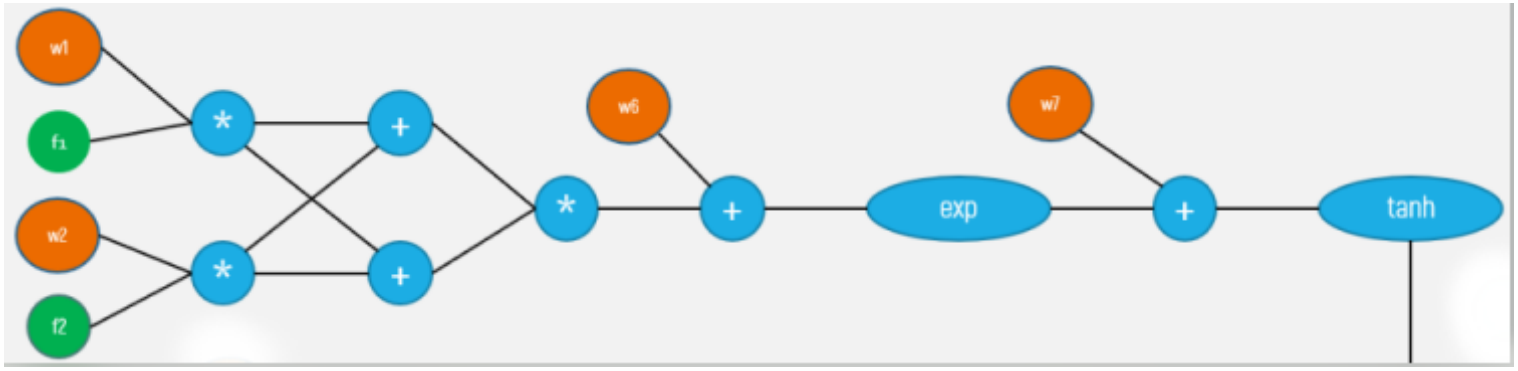
- **Forward propagation(Write your code in def forward_propagation())**

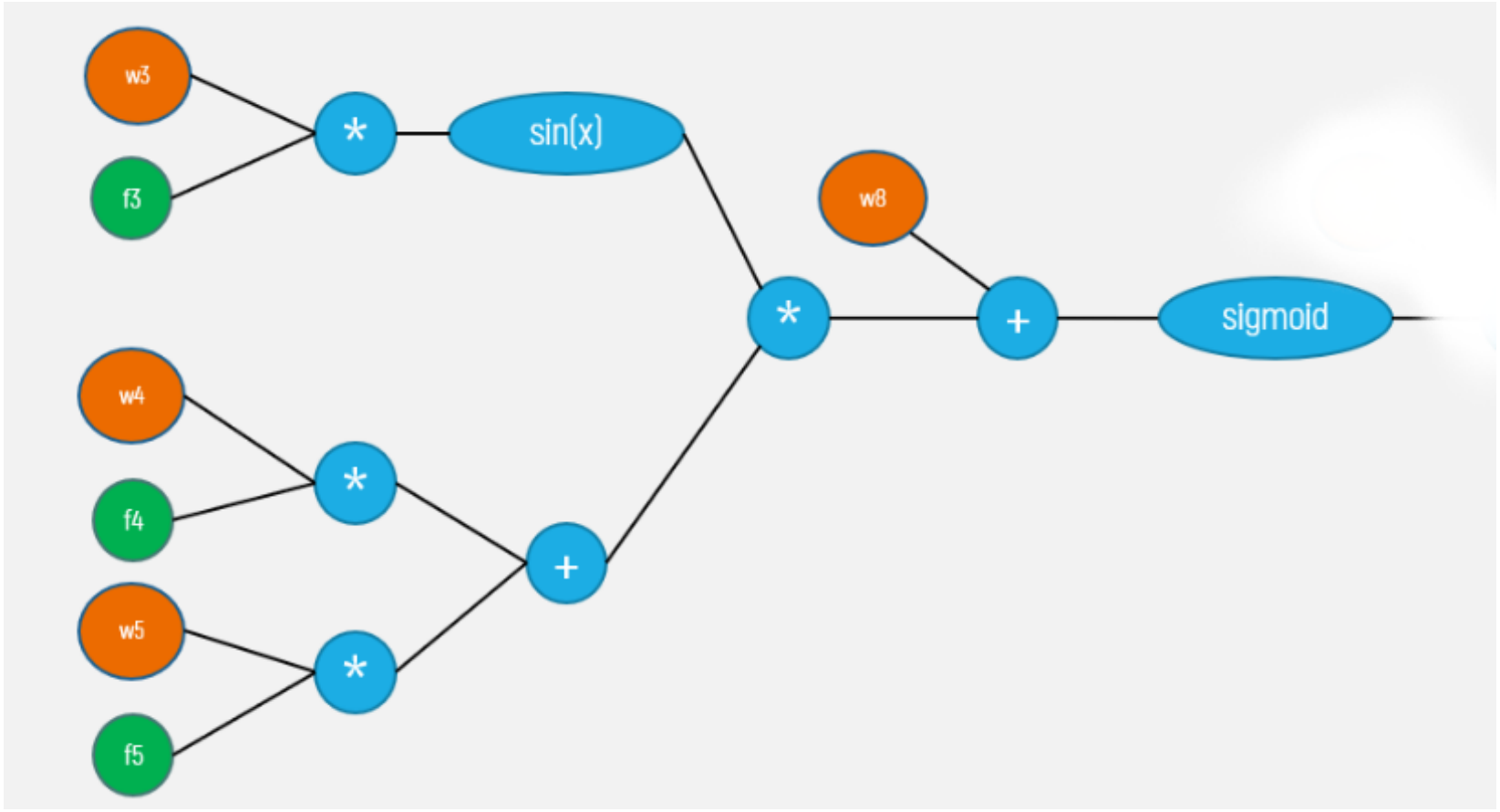
  For easy debugging, we will break the computational graph into 3 parts.

  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 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
```

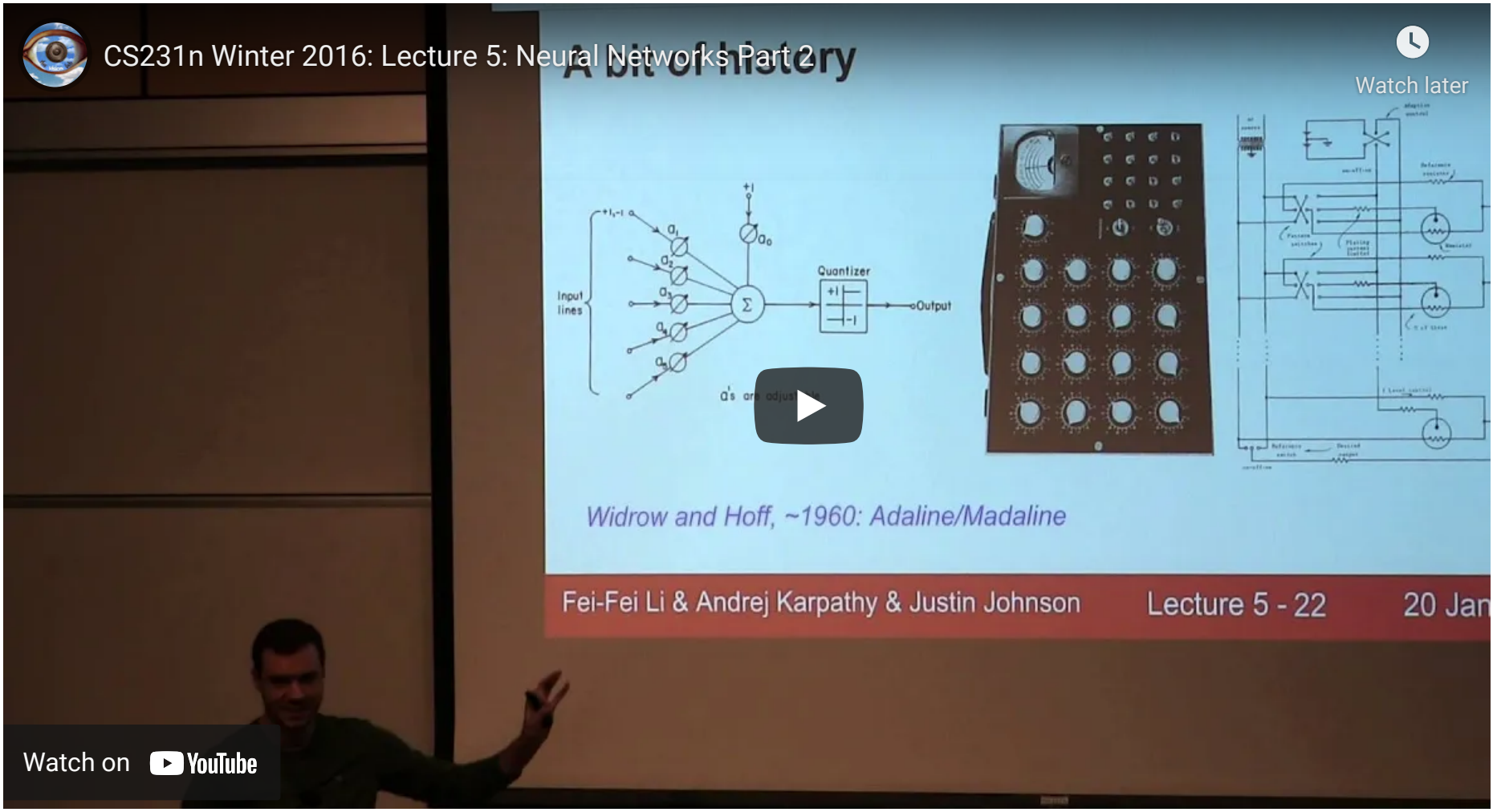
Task 2 : Optimizers

- **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 [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 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 [2]:

```
import math

# In this function we will compute sigmoid function
def sigmoid(x):
    sig = 1 / (1 + math.exp(-x))
    return sig

def forward_propagation(X, y, W):
        '''In this function, we will compute the forward propagation '''
        # X: input data point, note that in this assignment we 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
        # we have to return the following variables
```

```
        # exp= part1 (compute the forward propagation until exp and then store the values in exp)
        expval = math.exp( ( (W[0]*X[0] + W[1]*X[1]) * (W[0]*X[0] + W[1]*X[1]) ) + W[5])

        # tanh =part2(compute the forward propagation until tanh and then store the values in tanh)
        tanhval = math.tanh(expval + W[6])

        # sig = part3(compute the forward propagation until sigmoid and then store the values in sig)
        g1 = math.sin(W[2]*X[2]);g2 = (W[3]*X[3]) + (W[4]*X[4]);g3 = W[7]
        sigmoidval = sigmoid( (g1*g2) + g3 )

        # compute the Y_hat values
        y_hatval = (sigmoidval*W[8]) + tanhval

        # compute derivative of L  w.r.to Y' and store it in dl
        lossval = (y - y_hatval) * (y - y_hatval)
        dl = (-2 * (y - y_hatval))

        # Create a dictionary to store all the intermediate values
        variables = {'loss' : lossval ,'exp' : expval,'tanh' : tanhval,'sigmoid' : sigmoidval ,'dy_pr' : dl }

        return variables # hence we return the variables dictionary

# Lets make a sanity check
W = np.ones(9)*0.1
print('Outputs of forward propagation layer :- \n')
forward_propagation(X[0],y[0],W)
```

```
Outputs of forward propagation layer :-
```

Out[2]:
```
{'loss': 0.9298048963072919,
 'exp': 1.1272967040973583,
 'tanh': 0.8417934192562146,
 'sigmoid': 0.5279179387419721,
 'dy_pr': -1.9285278284819143}
```

Grader function - 1

In [3]:
```
def grader_sigmoid(z):
  val=sigmoid(z)
  assert(val==0.8807970779778823)
```

```
  return True
grader_sigmoid(2)
```

Out[3]: True

Grader function - 2

In [4]:
```
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[4]: True

Backward propagation

In [5]:
```
def backward_propagation(X,W,dict_fp):
    '''In this function, we will compute the backward propagation '''

    # dictionary: the outputs of the forward_propagation() function
    # Lets compute the derivatives of Loss with respect to the weights [w1,w2...w9]

    temp = (W[0]*X[0]) + (W[1]*X[1])

    dw0 =  (dict_fp['dy_pr']) *(1)* (1- np.square(dict_fp['tanh'])) * (dict_fp['exp']) * ( 2 * (temp * X[0]) )

    dw1 =  (dict_fp['dy_pr']) *(1)* (1- (dict_fp['tanh']* dict_fp['tanh'])) *(dict_fp['exp']) * ( 2 * (temp * X[1]) )

    g2 = (W[3]*X[3]) + (W[4]*X[4])
    dw2 =  (dict_fp['dy_pr']) *(W[8]) * ( (dict_fp['sigmoid'])*(1-dict_fp['sigmoid']) * g2 * X[2] * math.cos(W[2]*X[2
```

```
    g1 = math.sin(W[2]*X[2])
    dw3 =  (dict_fp['dy_pr']) *(W[8]) * ( (dict_fp['sigmoid'])*(1-dict_fp['sigmoid']) * g1 * X[3] * 1 )
    dw4 =  (dict_fp['dy_pr']) *(W[8]) * ( (dict_fp['sigmoid'])*(1-dict_fp['sigmoid']) * g1 * X[4] * 1 )

    dw5 =  (dict_fp['dy_pr']) *(1)* (1- (dict_fp['tanh']* dict_fp['tanh'])) *(dict_fp['exp'] * 1)

    dw6 =  (dict_fp['dy_pr']) *(1)* (1- (dict_fp['tanh']* dict_fp['tanh'])) * 1

    dw7 =  (dict_fp['dy_pr']) *(W[8]) * ( (dict_fp['sigmoid'])*(1-dict_fp['sigmoid']) * 1 )

    dw8 =  (dict_fp['dy_pr']) * dict_fp['sigmoid']

    dw = {'dw1' : dw0 , 'dw2' : dw1 , 'dw3' : dw2 , 'dw4' : dw3 , 'dw5' : dw4 , 'dw6' : dw5,'dw7' : dw6,'dw8' : dw7,'

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

Grader function - 3

In [6]:

```
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)
print(d1.values())
grader_backprop(d1)
```

```
dict_values([-0.22973323498702003, -0.021407614717752925, -0.005625405580266319, -0.004657941222712423, -0.0010077228
498574246, -0.6334751873437471, -0.561941842854033, -0.04806288407316516, -1.0181044360187037])
```

Out[6]:

```
True
```

Implement gradient checking

In [14]:

```
# Define the function:
def gradient_checking(X,y,W):

    # compute the L value using forward_propagation()
    dict3 = forward_propagation(X[0],y[0],W)
    L = dict3['loss']
    print("The current loss value :- \n",L)
    print('\n')

    # compute the gradients of W using backward_propagation()
    gradients = backward_propagation(X[0],W,dict3)
    print("Actual gradients :- \n",gradients.values())
    print('\n')
    approx_gradients = []

    # Apply the gradient checking formula & verify the outputs
    for i in range(W.shape[0]):
        W = np.ones(9) * 0.1
        eps = 0.0000001
        W[i] = W[i] + eps
        dict1 = forward_propagation(X[0],y[0],W)
        W = np.ones(9) * 0.1
        W[i] = W[i] - eps
        dict2 = forward_propagation(X[0],y[0],W)
        approx_gradients.append( (dict1['loss'] - dict2['loss'])/(2*eps) )
    # Lets check the difference between actual & approx gradient values
    r1 = list(gradients.values()); r2 = approx_gradients;
    print("Approximate gradients :- \n",r1)
    print('\n')
    gradient_check = []
    for j in range(W.shape[0]):
        val = ( (r1[j] - r2[j]) / (r1[j] + r2[j]) )
        gradient_check.append(val)

    # compare the gradient of weights W from backword_propagation() with the aproximation gradients of weights with g
    return gradient_check

# Initialize the W randomly & check the outputs
W = np.ones(9) * 0.1
```

```
gradient_check = gradient_checking(X,y,W)
print("Difference between actual gradients & approximate gradients :- \n",gradient_check)
```

```
The current loss value :-
 0.9298048963072919


Actual gradients :-
 dict_values([-0.22973323498702003, -0.021407614717752925, -0.005625405580266319, -0.004657941222712423, -0.001007722
8498574246, -0.6334751873437471, -0.561941842854033, -0.04806288407316516, -1.0181044360187037])


Approximate gradients :-
 [-0.22973323498702003, -0.021407614717752925, -0.005625405580266319, -0.004657941222712423, -0.0010077228498574246,
-0.6334751873437471, -0.561941842854033, -0.04806288407316516, -1.0181044360187037]


Difference between actual gradients & approximate gradients :-
 [-4.918866799775059e-10, 3.452723092703601e-09, -5.969913798437536e-08, -3.4579583689222766e-09, -2.477633744353281e
-07, -4.083967703487569e-10, 3.124184690307166e-10, -1.6117411221405448e-09, 1.284676527687261e-10]
```

Task 2: Optimizers

Algorithm with Vanilla update of weights

In [68]:

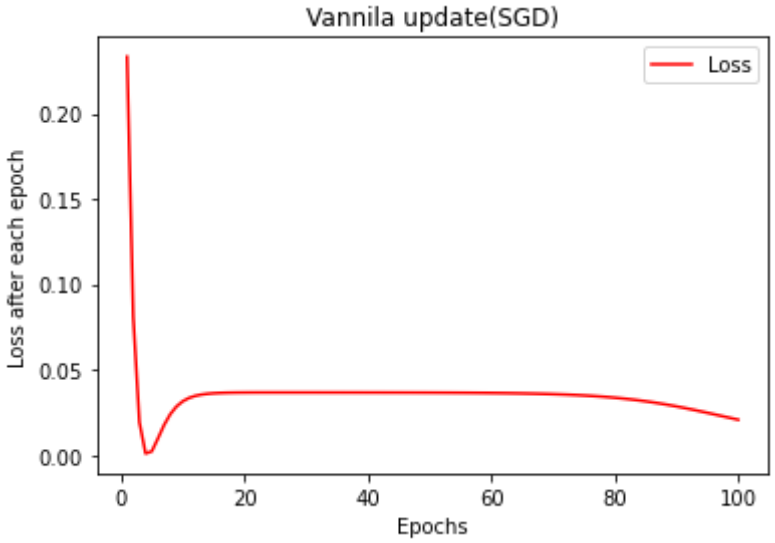
```
# Lets initialize some variables
import scipy.stats as sp
epoch = 100
W = sp.norm.rvs(0,0.01,size=9) # initialize weights from a gaussian distribution
lr = 0.001
Loss1 = []
for e in range(epoch):
    #print("For the Epoch {} :--->".format(e+1))

    for i in range(X.shape[0]):
        fp_dict = forward_propagation(X[i],y[i],W)
        bp_gradients = backward_propagation(X[i],W,fp_dict)
        bp_list = list(bp_gradients.values())
        W_new = []
        for j in range(W.shape[0]):
```

```
            w_val = (W[j]) - (lr * bp_list[j])
            W_new.append(w_val)

        W = np.array(W_new)

    #print('\nAfter Epoch {} -'.format(e+1))
    #print('\nThe updated weights :- \n',W)

    # At the end of each epoch lets compute the loss
    Loss1.append(fp_dict['loss'])

    #print('\n')

print('Final updated weights at the last epoch :\n',W)
print('\nLosses at the end of each epoch :-\n',Loss1)
```

```
Final updated weights at the last epoch :
 [-0.02014602 -0.03341819  0.48532681  0.24142305  0.4423634   0.51528066
  0.37327705  0.337693    1.3680326 ]

Losses at the end of each epoch :-
 [0.23327529626486185, 0.08091341041517738, 0.019173913469753376, 0.0009200948029449951, 0.002152785282434640
3, 0.00970556328138537, 0.01762287635105389, 0.023939281093942444, 0.02843000230725123, 0.031437382509665, 0.0
3338458079800460
4, 0.03462058601951032, 0.03539582735749302, 0.03587848977339332, 0.03617756714685832, 0.03636226998521559, 0.
03647602191387355, 0.03654587152801973, 0.036588593393154095, 0.03661456210371368, 0.03663018278610662, 0.0366
3940500890788, 0.03664466166557312, 0.03664744968984987, 0.03664868864210837, 0.03664894189566681, 0.036648552
967576235, 0.036647729493362105, 0.03664659491455562, 0.036645220259165584, 0.03664364364712254, 0.03664188222
338398, 0.03663994941512108, 0.03663780929599206
5, 0.03663547915423736
5, 0.03663293093831885, 0.03663014199295201, 0.03662708533744884, 0.03662372963880334, 0.0366200389703606
54, 0.03661597240858085, 0.03661148349637635, 0.03660651958624128, 0.03660102106648249, 0.03659492046720809,
0.03658814143804238, 0.036580597586012335, 0.03657219115921976, 0.03656281155948532, 0.03655233366496263, 0.0
3654061594170395
6, 0.03652749832129694
6, 0.03651279982003034, 0.03649631587365792
5, 0.03647781536087195, 0.03645703728823532, 0.03643368710983399
5, 0.0364074326566274, 0.03637789965383426, 0.03634466681020241, 0.03630726047141541
5, 0.03626514884193961, 0.03621773579633293, 0.03616435432357012, 0.03610425967759523
7, 0.03603662234551833, 0.03596052099313086, 0.03587493560715252
6, 0.03577874112597178, 0.03567070193624335, 0.03554946771113516, 0.03541357117537198
6, 0.03526142849836767, 0.03509134313237186, 0.03490151401653475
4, 0.03469004914412527, 0.03445498551766676
4, 0.03419431646925828, 0.03390602717121276, 0.03358813887494975
5, 0.03323876196842941, 0.03285615732078476, 0.03243880459500008
6, 0.03198547529382
16, 0.03149530733632817, 0.03096787705696944, 0.03040326381985686
3, 0.02980210210353406
7, 0.02916561607102871
3, 0.02849563237828959
7, 0.02779456828878389
8, 0.02706539395121376
8, 0.02631156976630159, 0.02553696185943024, 0.02474574051342167
2, 0.02394226775839446, 0.02313098099924961
6, 0.02231627952550007
5, 0.02150242004162400
4, 0.02069342612096897
5]
```

Plot between epochs and loss

In [69]:
```
import matplotlib.pyplot as plt

plt.plot(range(1,epoch+1),Loss1,color = 'red')
plt.xlabel('Epochs')
plt.ylabel('Loss after each epoch')
plt.legend(['Loss'])
plt.title('Vannila update(SGD)')
plt.show()
```



Algorithm with SGD update with momentum of weights

In [70]:
```
# Lets initialize some variables
epoch = 100
W = sp.norm.rvs(0,0.01,size=9) # initialize weights from a gaussian distribution
lr = 0.001
Loss2 = []
val = 0
for e in range(epoch):
    #print("For the Epoch {} :--->".format(e+1))
    for i in range(X.shape[0]):
        fp_dict = forward_propagation(X[i],y[i],W)
        bp_gradients = backward_propagation(X[i],W,fp_dict)
```

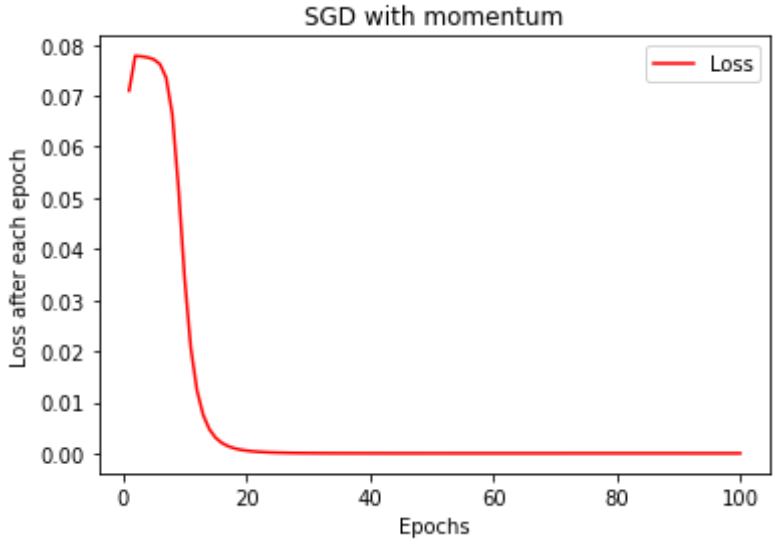
```
        # list of current gradients
        bp_list = list(bp_gradients.values())
        decay = 0.9

        val = (decay * val) + (lr * np.array(bp_list))
        W = W - val

    #print('\nAfter Epoch {} -'.format(e+1))
    #print('\nThe updated weights :- \n',W)

    # At the end of each epoch lets compute the training loss
    Loss2.append(fp_dict['loss'])

    #print('\n')

print('Final updated weights at the last epoch :\n',W)
print('\nLosses at the end of each epoch :-\n',Loss2)
```

```
Final updated weights at the last epoch :
 [-0.03901887  0.01800378 -0.97396998 -0.70232043 -0.7735639   0.44776678
  0.3808836   0.70107094  1.1225011 ]

Losses at the end of each epoch :-
 [0.07098532873361446, 0.07780576685099706, 0.07769268020602423, 0.07749167310374475, 0.07712027840419951, 0.07619091
58539027, 0.07353746110956975, 0.06642205318750372, 0.052034261085315864, 0.034272540397473884, 0.020652200389952198,
0.012345356970226921, 0.007542323474115544, 0.004750992622394131, 0.0030911671041835855, 0.002075120076882372, 0.0014
337661505894115, 0.0010165408444738036, 0.0007373445352693175, 0.0005456723058920902, 0.0004110809674447902, 0.000314
7030005930273, 0.00024452009294307205, 0.00019267334282322665, 0.00015389696049859932, 0.00012458407375932624, 0.0001
0221564620503564, 8.500257269876304e-05, 7.165579516560272e-05, 6.123512076836203e-05, 5.304759044789854e-05, 4.65777
913559825e-05, 4.143923786883751e-05, 3.733994668287428e-05, 3.4057759045816756e-05, 3.1422468676557164e-05, 2.930276
8976441678e-05, 2.7596652912665083e-05, 2.6224310087030856e-05, 2.512284384648282e-05, 2.4242323197943827e-05, 2.3542
81869277153e-05, 2.2992166816072236e-05, 2.256427581216723e-05, 2.2237835345465394e-05, 2.1995328411598428e-05, 2.182
227027072444e-05, 2.1706618540622142e-05, 2.1638312861231397e-05, 2.1608913090116853e-05, 2.161131279865915e-05, 2.16
39510634422398e-05, 2.1688426421925178e-05, 2.175375208156545e-05, 2.1831829839512883e-05, 2.1919551990213898e-05, 2.
2014277814623353e-05, 2.211376426475362e-05, 2.2216107785911197e-05, 2.2319695222637605e-05, 2.2423162192455567e-05,
2.2525357645141606e-05, 2.262531358222353e-05, 2.2722219109700056e-05, 2.2815398151342507e-05, 2.2904290271007656e-0
5, 2.298843414786399e-05, 2.3067453324830366e-05, 2.3141043911243783e-05, 2.320896397100382e-05, 2.3271024367183945e-
05, 2.3327080868016506e-05, 2.3377027346725805e-05, 2.3420789930684882e-05, 2.3458321975253582e-05, 2.348959975399032
2e-05, 2.3514618771196034e-05, 2.3533390614823483e-05, 2.354594027842444e-05, 2.355230388955915e-05, 2.35525267902220
34e-05, 2.3546661921556637e-05, 2.3534768471165952e-05, 2.3516910746371934e-05, 2.349315724168386e-05, 2.346357987236
3037e-05, 2.3428253349866026e-05, 2.3387254677717346e-05, 2.3340662749353146e-05, 2.328855803174236e-05, 2.3231022320
```

```
631547e-05, 2.316813855536186e-05, 2.3099990682645233e-05, 2.302666356006112e-05, 2.2948242891569963e-05, 2.286481518
8227942e-05, 2.2776467748301694e-05, 2.2683288651934538e-05, 2.2585366766169977e-05, 2.2482791756795034e-05]
```

Plot between epochs and loss

In [71]:
```
import matplotlib.pyplot as plt

plt.plot(range(1,epoch+1),Loss2,color = 'red')
plt.xlabel('Epochs')
plt.ylabel('Loss after each epoch')
plt.legend(['Loss'])
plt.title('SGD with momentum')
plt.show()
```



Algorithm with Adam update of weights

In [72]:
```
# Lets initialize some variables
epoch = 100
W = sp.norm.rvs(0,0.01,size=9) # initialize weights from a gaussian distribution
lr = 0.001
Loss3 = []
```

```
# required parameters for Adam optimizer
beta1 = 0.9;beta2 = 0.99
epsilon = 1e-8
v_t = 0 ; s_t = 0 ;

for e in range(epoch):
    #print("For the Epoch {} :--->".format(e+1))
    for i in range(X.shape[0]):

        fp_dict = forward_propagation(X[i],y[i],W)
        bp_gradients = backward_propagation(X[i],W,fp_dict)

        # list of current gradients
        bp_list = list(bp_gradients.values())

        #bias corrections & weight updates
        v_t = (beta1 * v_t) + ( (1-beta1)*np.array(bp_list))
        s_t = (beta2 * s_t) + ( (1-beta2)*(np.array(bp_list)**2))

        v_hat_t = (v_t / (1 - beta1))
        s_hat_t = (s_t / (1 - beta2))

        denom = (np.sqrt(s_hat_t) + epsilon)
        W = W - ( (lr * v_hat_t) / denom )

    #print('\nAfter Epoch {} -'.format(e+1))
    #print('\nThe updated weights :- \n',W)

    # At the end of each epoch lets compute the loss
    Loss3.append(fp_dict['loss'])


print('Final updated weights at the last epoch :\n',W)
print('\nLosses at the end of each epoch :-\n',Loss3)
```

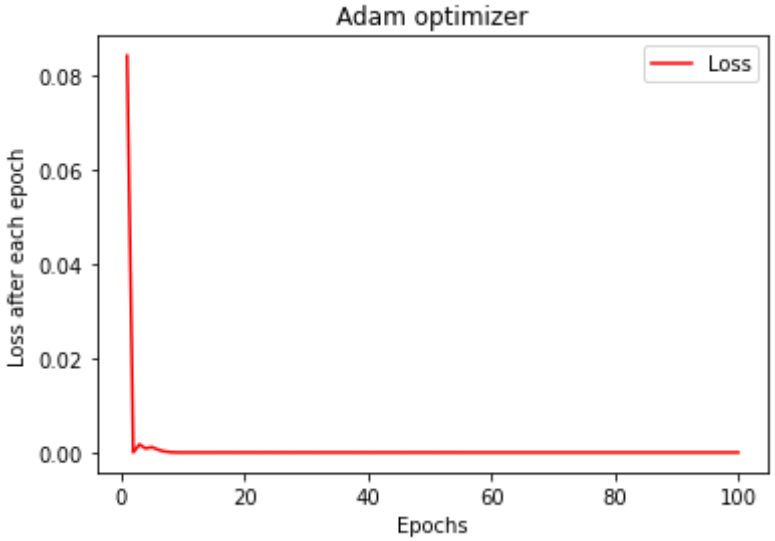
```
Final updated weights at the last epoch :
 [-0.84108393 -0.90495551  0.99854603  0.99651906  0.99731368  0.99874318
  0.90175931  1.00469888  0.99650514]

Losses at the end of each epoch :-
 [0.08429928907602464, 1.532182062396191e-05, 0.0017185177540325778, 0.0008511556215556644, 0.001145188762491921, 0.0
005834164897845877, 0.0002303343524044389, 6.330922684525501e-05, 7.66143530927384e-06, 9.90025330644395e-07, 9.70764
```

```
6009429321e-07, 3.2891361000094044e-07, 8.243279092813693e-08, 6.18448869203407e-07, 4.275590816216602e-08, 3.3935089
41121592e-07, 3.450765563347274e-07, 1.0755170476015207e-07, 4.394259775052705e-07, 3.90118560005372e-07, 1.942536455
00702e-07, 1.2562991692662065e-07, 4.804044351993852e-07, 3.964580868273305e-07, 1.2887462459607324e-07, 4.0994885663
281045e-07, 3.401166303326444e-07, 1.2428569242169644e-07, 4.00821573131276e-07, 2.966271137600498e-07, 1.19448269134
27314e-07, 4.1144090552859333e-07, 2.7676659159269487e-07, 1.7110165182141422e-07, 2.3106063420433786e-07, 1.55359300
7275854e-07, 2.9373226142643876e-07, 1.275812662849508e-07, 2.722979313626693e-07, 2.353598146111116e-07, 1.387432703
3774755e-07, 3.8611475324289405e-07, 2.4965602201910117e-07, 2.0072200464092122e-07, 2.5722170112951197e-07, 1.272125
6278395182e-07, 2.719041274722536e-07, 2.1463686324246111e-07, 1.329884710316946e-07, 1.8508308706809792e-07, 5.75484
4807291114e-08, 2.2838376947488905e-07, 3.2799241661277916e-08, 1.5619907619246824e-07, 1.187770623336713e-07, 5.7204
540547384746e-08, 9.810014821780054e-08, 1.1043076259464602e-07, 4.597478115815716e-08, 9.866883894405832e-08, 6.5250
66959821901e-08, 3.4344510504473597e-08, 6.253978091118317e-08, 2.801186685936014e-08, 6.996364874657236e-08, 5.03485
885249098e-08, 6.402883684787382e-08, 7.701560830904352e-08, 3.6019492980922675e-08, 6.934054657661082e-08, 7.0272453
07244126e-08, 3.480570791666262e-08, 4.99518299681107e-08, 4.84317371560263e-09, 2.7108210059901982e-08, 1.8368368144
272224e-08, 1.1721246676500723e-09, 2.167551339489703e-08, 2.0475394969642703e-09, 2.422343420705046e-08, 4.045640080
752241e-08, 1.528488683155289e-08, 4.4552556653968385e-09, 1.755176025179816e-08, 5.407150468394747e-08, 2.5548816306
52995e-08, 3.1742418699689e-08, 3.796327270031166e-08, 2.711337284535103e-08, 2.051258478048972e-08, 3.10781594432330
7e-08, 1.2973337971386928e-08, 4.216775459391696e-09, 1.0669744583407095e-08, 8.219998028911166e-09, 1.48300985164245
79e-08, 1.2263600604161309e-08, 1.0843108046026901e-08, 2.11429603026097e-08, 3.0858799748176107e-08]
```

Plot between epochs and loss

In [73]:

```
import matplotlib.pyplot as plt

plt.plot(range(1,epoch+1),Loss3,color = 'red')
plt.xlabel('Epochs')
plt.ylabel('Loss after each epoch')
plt.legend(['Loss'])
plt.title('Adam optimizer')
plt.show()
```

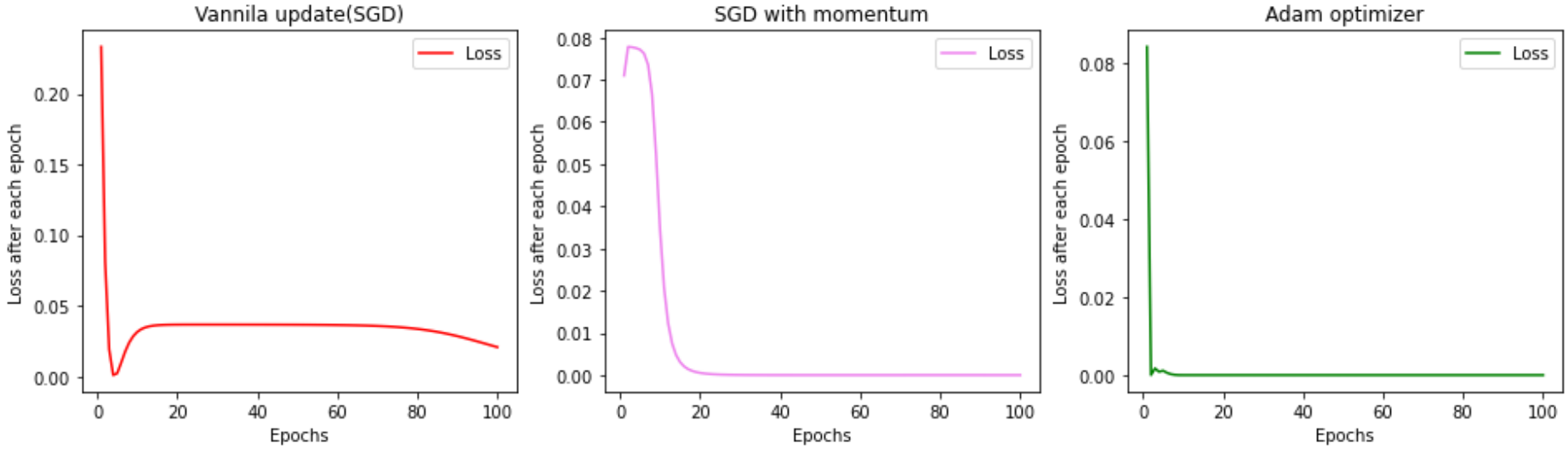


Comparision plot between epochs and loss with different optimizers

In [74]:

```
# lets plot the final
plt.figure(figsize = (16,4))

# PLOT (1) --> Plain SGD
plt.subplot(1,3,1)
plt.plot(range(1,epoch+1),Loss1,color = 'red')
plt.xlabel('Epochs')
plt.ylabel('Loss after each epoch')
plt.legend(['Loss'])
plt.title('Vannila update(SGD)')

# PLOT (2) --> SGD with momentum
plt.subplot(1,3,2)
plt.plot(range(1,epoch+1),Loss2,color = 'violet')
plt.xlabel('Epochs')
plt.ylabel('Loss after each epoch')
plt.legend(['Loss'])
plt.title('SGD with momentum')

# PLOT (3) --> Adam
plt.subplot(1,3,3)
```

```
plt.plot(range(1,epoch+1),Loss3,color = 'green')
plt.xlabel('Epochs')
plt.ylabel('Loss after each epoch')
plt.legend(['Loss'])
plt.title('Adam optimizer')

plt.show()
```



Observation :- Hence we can clearly see that adam optimizer converges faster than both sgd & sgd with momentum optimizers.