Backpropagation

In this assignment, you will implement Backpropagation from scratch. You will then verify the correctness of the your implementation using a "grader" function/cell (provided by us) which will match your implmentation.

The grader fucntion would help you validate the correctness of your code.

Please submit the final Colab notebook in the classroom ONLY after you have verified your code using the grader function/cell.

In [1]:
```
from google.colab import files

files = files.upload()
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving data.pkl to data.pkl
```

Loading data

In [2]:
```
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,)
```

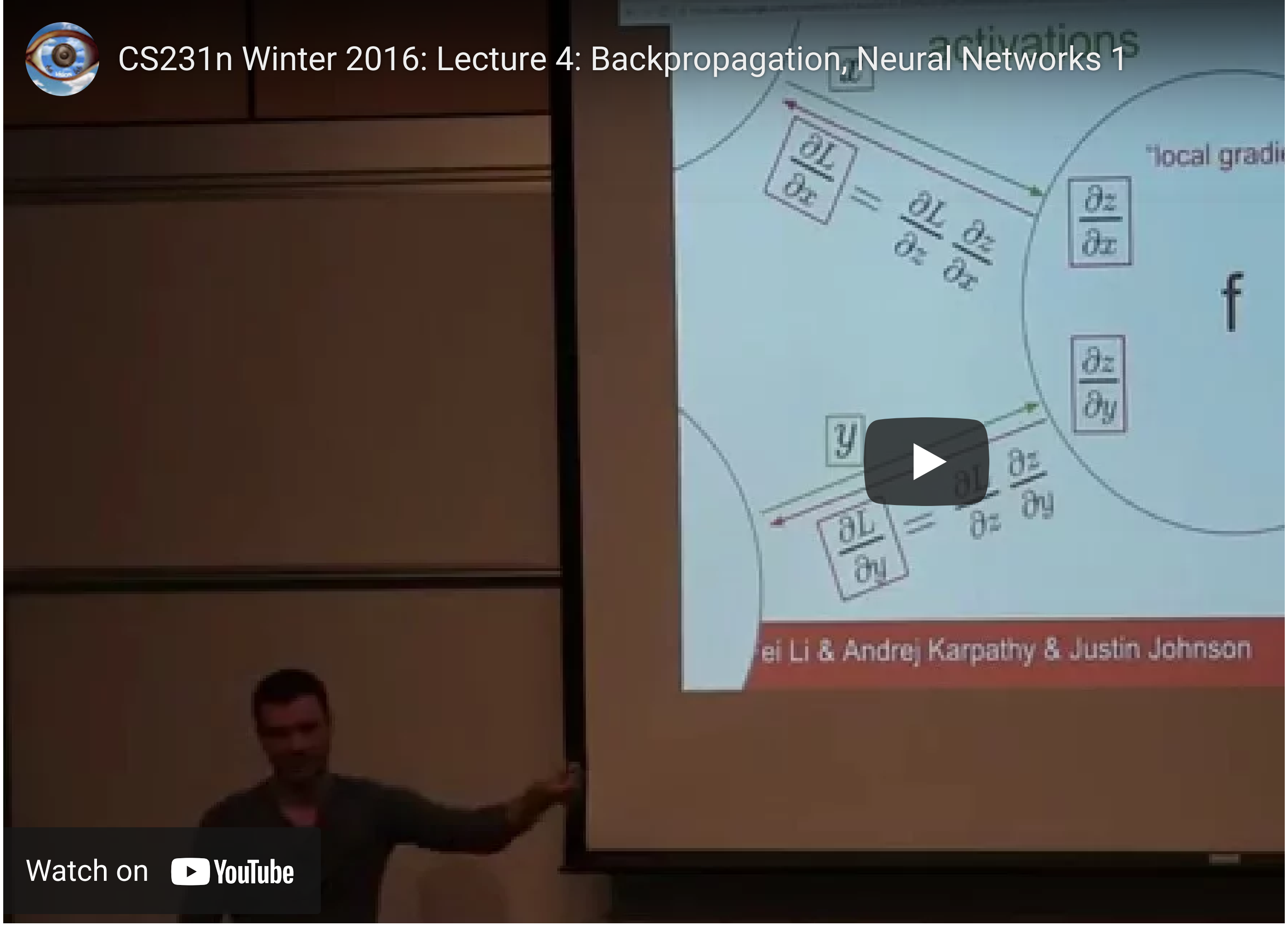
In [3]:
```
type(data)
```

Out[3]:
```
numpy.ndarray
```

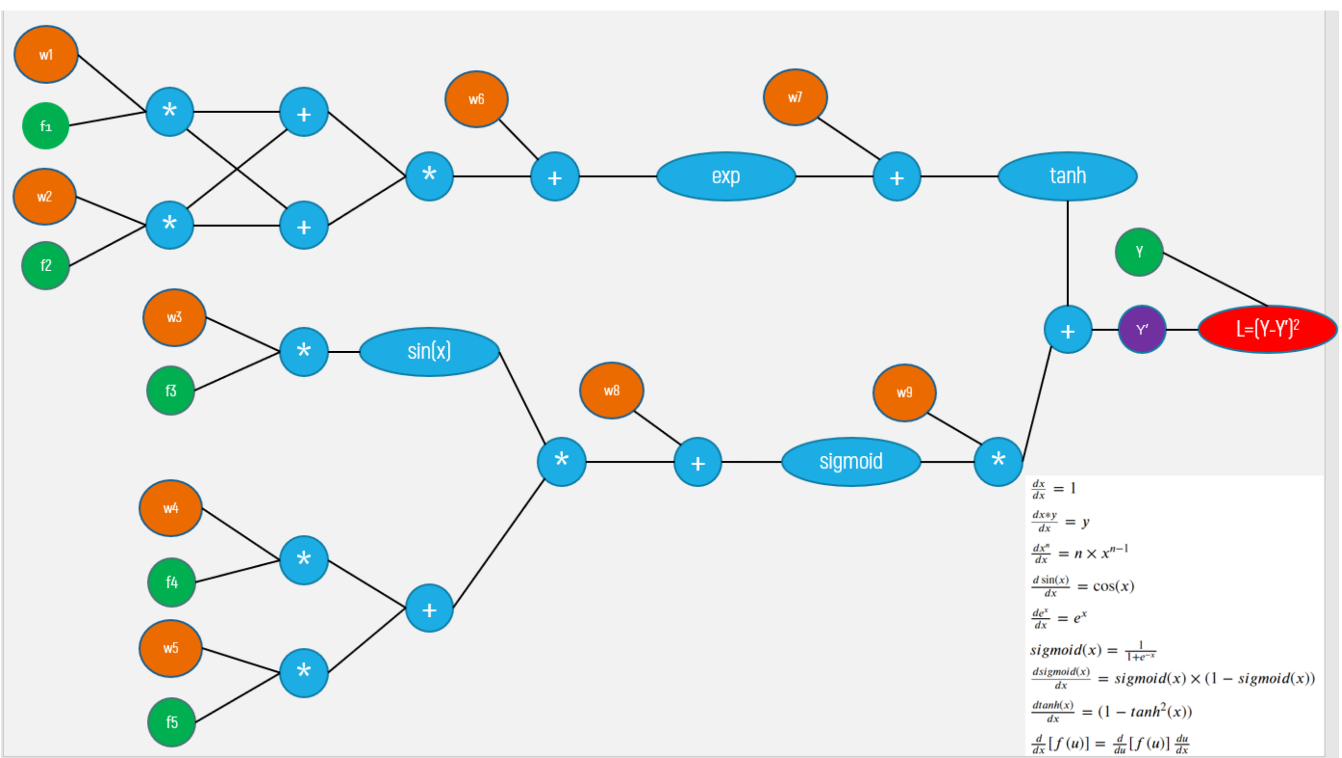
Check this video for better understanding of the computational graphs and back propagation

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

Out[ ]:



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 Forward propagation, Backpropagation and Gradient checking

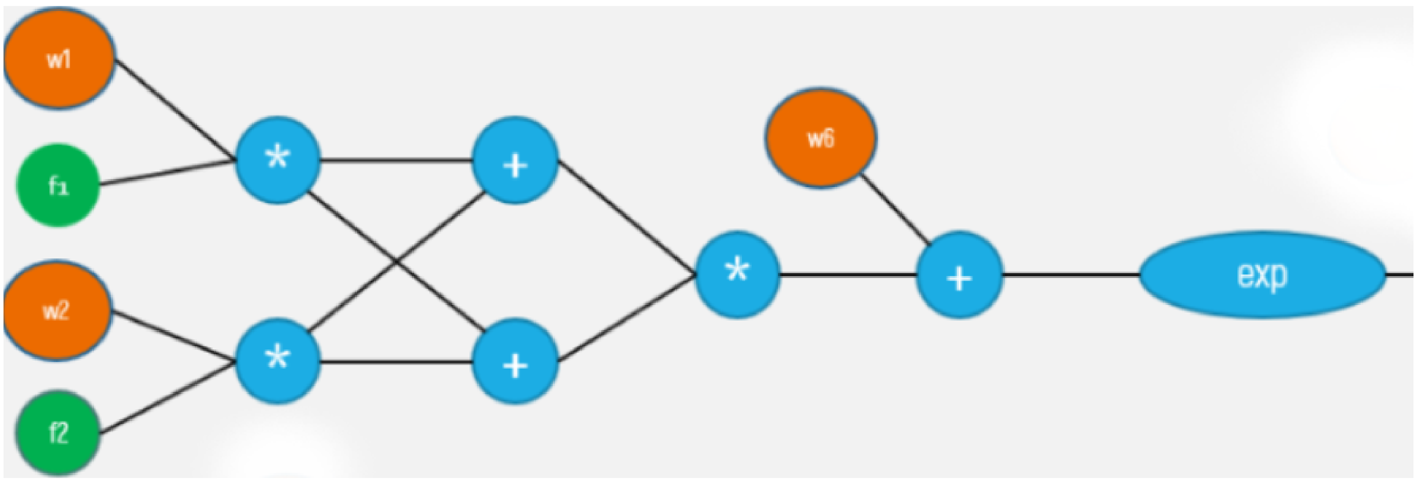
Task 1.1

Forward propagation

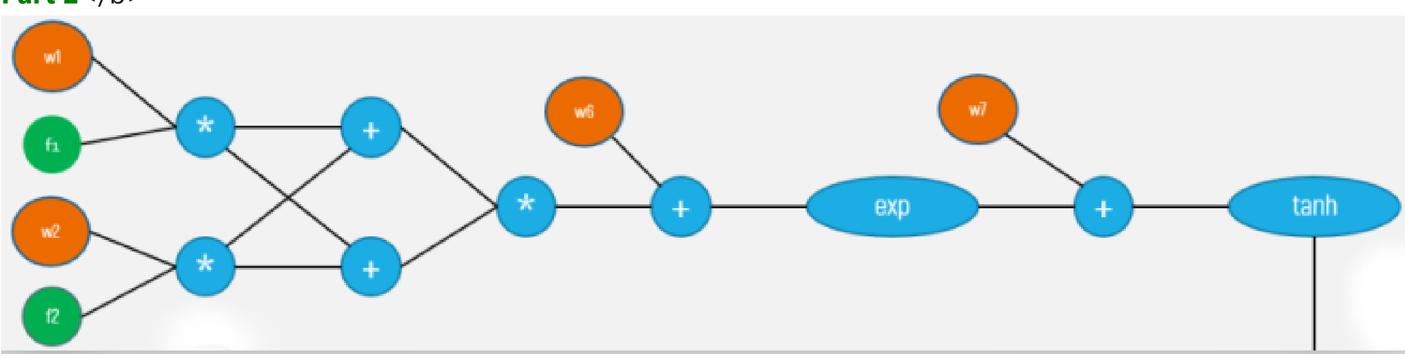
- **Forward propagation(Write your code in def forward_propagation())**

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

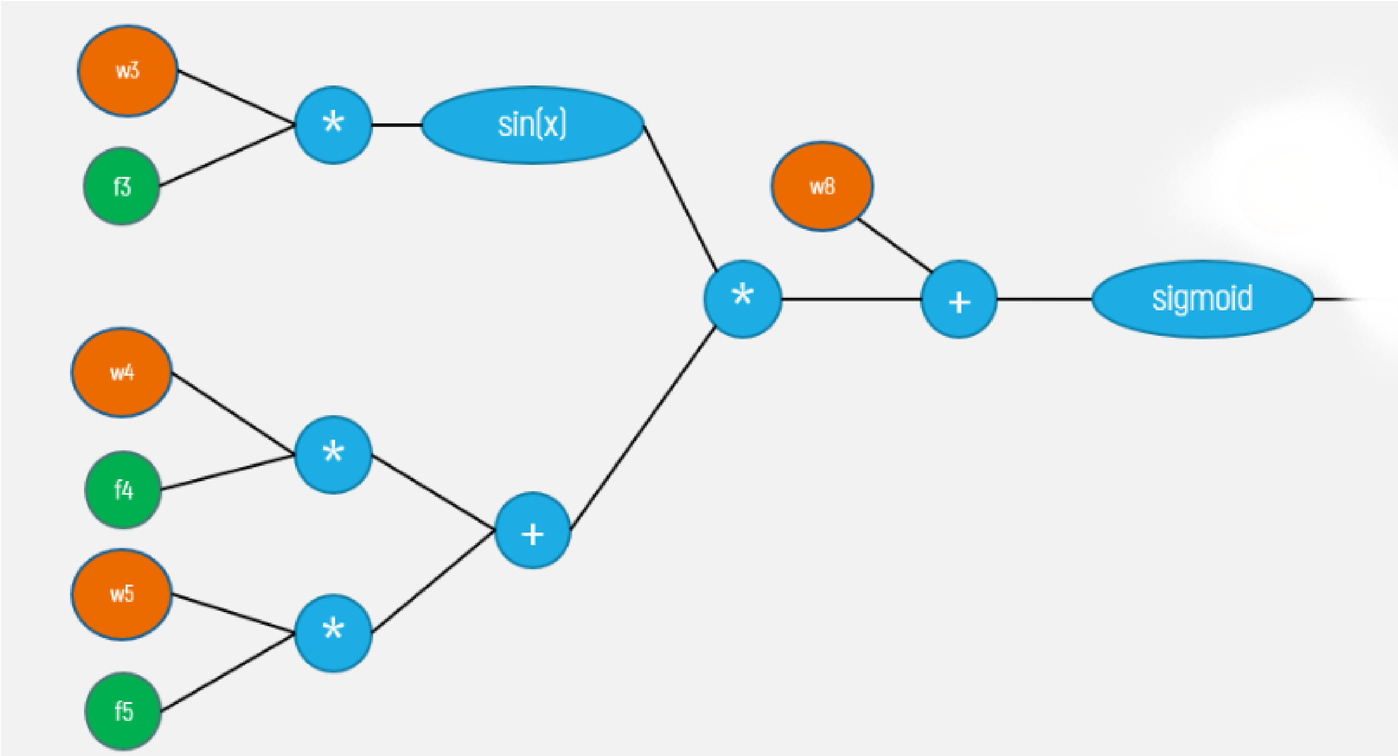
Part 1



Part 2



Part 3



In [4]:
```
def sigmoid(z):
    '''In this function, we will compute the sigmoid(z)'''
    # we can use this function in forward and backward propagation
    # write the code to compute the sigmoid value of z and return that value
    return 1/(1+np.exp(-z))
```

In [5]:
```
def grader_sigmoid(z):
  #if you have written the code correctly then the grader function will output true
  val=sigmoid(z)
  assert(val==0.8807970779778823)
  return True
grader_sigmoid(2)
```

Out[5]:
```
True
```

In [6]:
```
def forward_propagation(x, y, w):
        '''In this function, we will compute the forward propagation '''
        # X: input data point, note that in this assignment you are having 5-d data poin
        # y: output varible
        # W: weight array, its of length 9, W[0] corresponds to w1 in graph, W[1] corres
        # you have to return the following variables
        # exp= part1 (compute the forward propagation until exp and then store the value
        # tanh =part2(compute the forward propagation until tanh and then store the valu
        # sig = part3(compute the forward propagation until sigmoid and then store the v
        # we are computing one of the values for better understanding

        val_1= (w[0]*x[0]+w[1]*x[1]) * (w[0]*x[0]+w[1]*x[1]) + w[5]
        part_1 = np.exp(val_1)

        val_2 = np.sin(w[2]*x[2])*(w[3]*x[3]+w[4]*x[4])+w[7]
        part_2 = sigmoid(val_2)

        val_3 = part_1 + w[6]
        part_3 = np.tanh(val_3)

        y_dash = part_3 + (part_2*w[8])
```

```python
        # after computing part1,part2 and part3 compute the value of y' from the main Co
        # write code to compute the value of L=(y-y')^2 and store it in variable loss
        # compute derivative of L  w.r.to y' and store it in dy_pred
        # Create a dictionary to store all the intermediate values i.e. dy_pred ,loss,ex
        # we will be using the dictionary to find values in backpropagation, you can add

        forward_dict={}
        forward_dict['exp']= part_1
        forward_dict['sigmoid'] = part_2
        forward_dict['tanh'] = part_3
        forward_dict['loss'] = (y-y_dash)**2
        forward_dict['dy_pred'] = (-2)*(y-y_dash)

        return forward_dict
```

In [7]:

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

Task 1.2

Backward propagation

In [8]:

```python
def backward_propagation(x,y,w,forward_dict):
    '''In this function, we will compute the backward propagation '''
    # forward_dict: 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
    # 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


    backward_dict={}
    #store the variables dw1,dw2 etc. in a dict as backward_dict['dw1']= dw1,backward_di
    backward_dict['dw9'] = forward_dict['dy_pred']*forward_dict['sigmoid']
    backward_dict['dw8'] = backward_dict['dw9']*(1-forward_dict['sigmoid'])*w[8]
    backward_dict['dw7'] = forward_dict['dy_pred']*(1-(forward_dict['tanh'])**2)
    backward_dict['dw6'] = backward_dict['dw7']*forward_dict['exp']
    backward_dict['dw5'] = backward_dict['dw8']*np.sin(w[2]*x[2])*x[4]
    backward_dict['dw4'] = backward_dict['dw8']*np.sin(w[2]*x[2])*x[3]
    backward_dict['dw3'] = backward_dict['dw8']*(w[3]*x[3]+w[4]*x[4])*x[2]*np.cos(x[2]*w
    backward_dict['dw2'] = backward_dict['dw6']*2*(w[0]*x[0]+w[1]*x[1])*x[1]
    backward_dict['dw1'] = backward_dict['dw6']*2*(w[0]*x[0]+w[1]*x[1])*x[0]
```

```
    return backward_dict
```

In [9]:

```
def grader_backprop(data):
    dw1=(np.round(data['dw1'],6)==-0.229733)
    dw2=(np.round(data['dw2'],6)==-0.021408)
    dw3=(np.round(data['dw3'],6)==-0.005625)
    dw4=(np.round(data['dw4'],6)==-0.004658)
    dw5=(np.round(data['dw5'],6)==-0.001008)
    dw6=(np.round(data['dw6'],6)==-0.633475)
    dw7=(np.round(data['dw7'],6)==-0.561942)
    dw8=(np.round(data['dw8'],6)==-0.048063)
    dw9=(np.round(data['dw9'],6)==-1.018104)
    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
forward_dict=forward_propagation(X[0],y[0],w)
backward_dict=backward_propagation(X[0],y[0],w,forward_dict)
grader_backprop(backward_dict)
```

Out[9]:

```
True
```

Task 1.3

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
```

In [10]:
```python
def gradient_checking(x,y,w,eps=1e-7):
    # compute the dict value using forward_propagation()
    # compute the actual gradients of W using backword_propagation()
    forward_dict=forward_propagation(x,y,w)
    backward_dict=backward_propagation(x,y,w,forward_dict)

    #we are storing the original gradients for the given datapoints in a list

    original_gradients_list=list(backward_dict.values())
    # make sure that the order is correct i.e. first element in the list corresponds to
    # you can use reverse function if the values are in reverse order
    original_gradients_list = original_gradients_list[::-1]
    approx_gradients_list=[]

    #now we have to write code for approx gradients, here you have to make sure that you
    #write your code here and append the approximate gradient value for each weight in
    for i in range(9):
      w_dash_h = w
      w_dash_l = w
      w_dash_h[i]+= eps
      w_dash_l[i]-= eps
      dict_h = forward_propagation(x,y,w_dash_h)['loss']
      dict_l = forward_propagation(x,y,w_dash_l)['loss']
      approx_gradients_list.append(float((dict_h-dict_l)/(2*eps)))
      #print(dict_l)
      #approx_gradients_list.append()
    #performing gradient check operation
    original_gradients_list=np.array(original_gradients_list)
    approx_gradients_list=np.array(approx_gradients_list)
    gradient_check_value =(original_gradients_list-approx_gradients_list)/(original_grad

    return gradient_check_value
```

In [11]:
```python
def grader_grad_check(value):
    print(value)
    assert(np.all(value <= 10**-3))
    return True

w=[ 0.00271756,  0.01260512,  0.00167639, -0.00207756,  0.00720768,
   0.00114524,  0.00684168,  0.02242521,  0.01296444]

eps=10**-7
value = gradient_checking(X[0],y[0],w,eps)
grader_grad_check(value)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Out[11]:
```
True
```

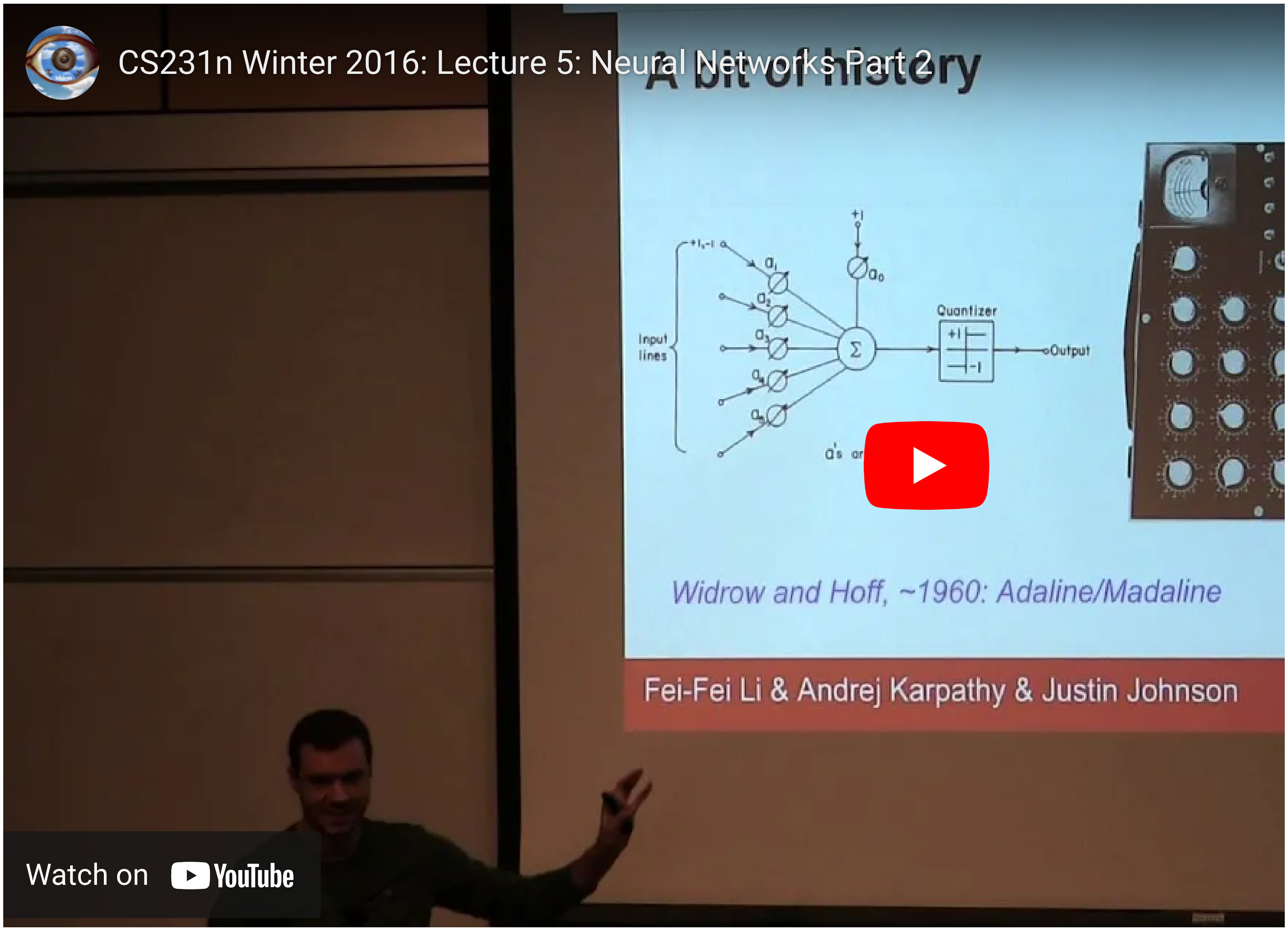
Task 2 : Optimizers

- As a part of this task, you will be implementing 2 optimizers(methods to update weight)
- Use the same computational graph that was mentioned above to do this task
- The weights have been initialized from normal distribution with mean=0 and std=0.01. The initialization of weights is very important otherwiswe you can face vanishing gradient and exploding gradients problem.

Check below video for reference purpose

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

Out[ ]:



Algorithm

```
    for each epoch(1-20):
        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
```

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 .

2.1 Algorithm with Vanilla update of weights

In [12]:
```
#initialzation of weights from the normal distribution
from sklearn.metrics import mean_squared_error
rate=.001
mu, sigma = 0, 0.01
w = np.random.normal(mu, sigma, 9)
```

In [13]:
```
#code for vanilla updates
vanilla_loss = []
for epoch in range(20):
  loss = []
  for point in range(len(data)):
    forward_dict=forward_propagation(X[point], y[point], w)
    loss.append(forward_dict['loss'])
    backward=backward_propagation(X[point], y[point], w,forward_dict)
    for i in range(9):
      w[i]=w[i]-rate*backward['dw'+str(i+1)]


  vanilla_loss.append(sum(loss))
```

2.2 Algorithm with Momentum update of weights

Momentum based Gradient Descent Update Rule

$$v_t = \gamma * v_{t-1} + \eta \nabla w_t$$

$$w_{t+1} = w_t - v_t$$

Here Gamma referes to the momentum coefficient, eta is leaning rate and v_t is moving average of our gradients at timestep t

In [14]:
```
#momentum updates
mom = 0.9
moment_loss = []
m = np.zeros(9)
w = np.random.normal(0, 0.01, 9)
```

```
for epoch in range(20):
  loss = []
  for point in range(len(data)):
    forward_dict=forward_propagation(X[point], y[point], w)
    loss.append(forward_dict['loss'])
    backward=backward_propagation(X[point], y[point], w,forward_dict)
    for i in range(9):
      m[i] =mom*m[i]+rate*backward['dw'+str(i+1)]
      w[i] = w[i] - m[i]

  moment_loss.append(sum(loss))
```

2.3 Algorithm with Adam update of weights

$$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * \nabla w_t$$

$$v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * (\nabla w_t)^2$$

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

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

In [15]:

```
#Adam updates
beta1 = 0.9
beta2 = 0.99
eps = 1e-8
m = np.zeros(9)
v = np.zeros(9)
w = np.random.normal(0, 0.01, 9)
adam_loss = []
for epoch in range(20):
  loss = []
  for point in range(len(data)):
    forward_dict=forward_propagation(X[point], y[point], w)
    loss.append(forward_dict['loss'])
    backward=backward_propagation(X[point], y[point], w,forward_dict)
    for i in range(9):
      m[i] = beta1*m[i]+ (1-beta1)*backward['dw'+str(i+1)]
      v[i] = beta2*v[i]+ (1-beta2)*(backward['dw'+str(i+1)]**2)
      mt = m[i]/(1-beta1)
      vt = v[i]/(1-beta2)
      w[i] = w[i] - (rate/np.sqrt(vt+eps))*mt

  adam_loss.append(sum(loss))
```

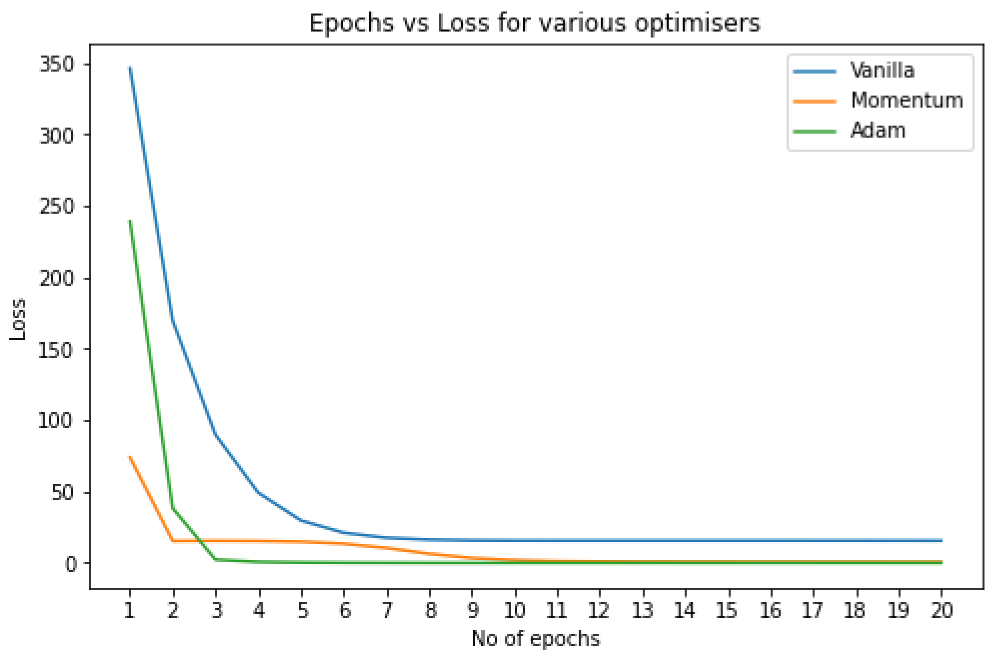
Comparision plot between epochs and loss with different optimizers. Make sure that loss is conerging with increaing epochs

In [18]:

```
#plot the graph between loss vs epochs for all 3 optimizers.
```

```python
import matplotlib.pyplot as plt

epochs = list(range(1,21))
fig1 = plt.figure(figsize = (8,5))
plt.plot(epochs, vanilla_loss, label = 'Vanilla')
plt.plot(epochs, moment_loss, label = 'Momentum')
plt.plot(epochs, adam_loss, label = 'Adam')
plt.xticks(np.arange(min(epochs), max(epochs)+1, 1.0))
plt.legend()
plt.title('Epochs vs Loss for various optimisers')
plt.xlabel('No of epochs')
plt.ylabel('Loss')
plt.show()
```



You can go through the following blog to understand the implementation of other optimizers . **[Gradients update blog](https://cs231n.github.io/neural-networks-3/)**

Observations for each optimizer:

The learning rate of all the optimizers was 0.001.

1. **VANILLA OPTIMIZER:**

- **The initial loss of the vanilla optimizer was the highest.**
- **The convergence happended at the 7th epoch.**

1. **MOMENTUM OPTIMIZER:**

- **The initial loss of the momentum loss was the lowest of all the three.**
- **The value of m in momentum updates used here is 0.9, which is widely considered to be the most optimal value for m.**
- **It converged at the 10th epoch, surprisingly worse than the vanilla optimizer. Since, it is in theory supposed to be converge faster than the vanilla optimizer.**

1. **ADAM OPTIMIZER:**

- **Of all the three update methods the performance of Adam optimiser is the best, as was expected.**

- **It converged at the 3rd epoch itself.**
- **The most widely considered optimial values of 0.9, 0.99 and 1e-8 were used for beta1, beta2 and eps respectively, in Adam updates.**

In [ ]:
```

```