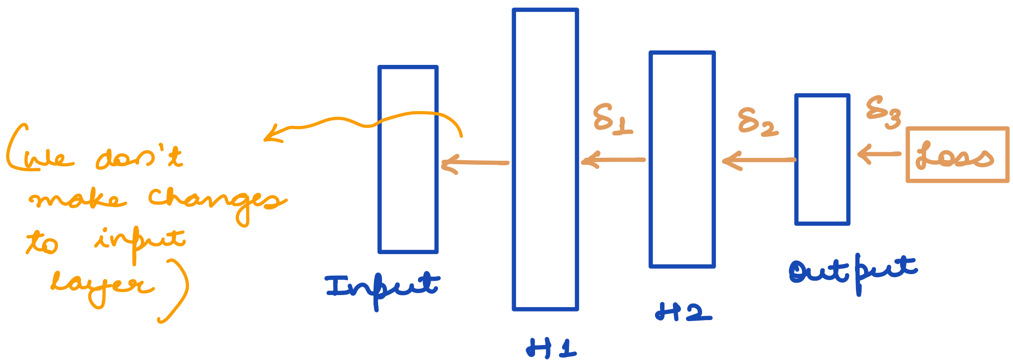
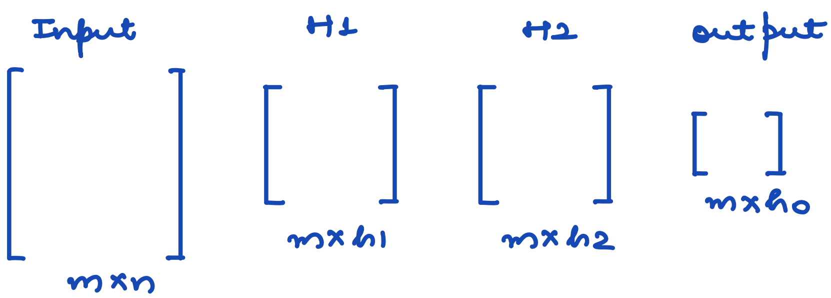
Lets say you have a network like this:



At output layer you have loss function, we will compute what error should be propagated back at every step.
We have to compute δ_1, δ_2 and δ_3.

Our input is going to be matrix which has m rows and n features. Lets say H1 layer has h_1 units and H_2 has h_2 units.
H1 layer will produce a matrix of shape $m \times h1$
H2 layer will produce a matrix of shape $m \times h2$
Output layer will produce a matrix of shape $m \times ho$

ho = no. of classes



$w1$	$w2$	$w3$
$n \times h_1$	$h1 \times h2$	$h_2 \times ho$

We will compute ∂w_1, ∂w_2 and ∂w_3.
These will have same shapes as w_1, w_2 and w_3.

```python
class NeuralNetwork :

    def __init__(self, input_size, layers, output_size) :

        np.random.seed(0)

        model = {} # Dictionary

        # First Layer
        model['W1'] = np.random.randn(input_size, layers[0])
        model['b1'] = np.zeros((1, layers[0]))

        # Second Layer
        model['W2'] = np.random.randn(layers[0], layers[1])
        model['b2'] = np.zeros((1, layers[1]))

        # Third Layer
        model['W3'] = np.random.randn(layers[1], output_size)
        model['b3'] = np.zeros((1, layers[2]))

        self.model = model

    def forward(self, x) :

        W1,W2,W3 = self.model['W1'], self.model['W2'], self.model['W3']
        b1,b2,b3 = self.model['b1'], self.model['b2'], self.model['b3']

        z1 = np.dot(x,W1) + b1
        a1 = np.tanh(z1)

        z2 = np.dot(a1,W2) + b2
        a2 = np.tanh(z2)

        z3 = np.dot(a2,W3) + b3
        y_ = softmax(z3)

        self.activation_outputs = (a1, a2, y_)
        return y_

    def backward(self, x, y, learning_rate=0.001) :

        W1,W2,W3 = self.model['W1'], self.model['W2'], self.model['W3']
        b1,b2,b3 = self.model['b1'], self.model['b2'], self.model['b3']
        m = x.shape[0]

        a1, a2, y_ = self.activation_outputs

        delta3 = y_ - y
        dw3 = np.dot(a2.T, delta3)
        db3 = np.sum(delta3, axis=0)/float(m)

        delta2 = (1-np.square(a2)) * np.dot(delta3,W3.T)
        dw2 = np.dot(a1.T, delta2)
        db2 = np.sum(delta2,axis=0)/float(m)

        delta1 = (1-np.square(a1)) * np.dot(delta2,W2.T)
        dw1 = np.dot(X.T, delta1)
        db1 = np.sum(delta1, axis=0)/float(m)

        # Update the Model Parameters using Gradient Descent
        self.model["W1"] -= learning_rate * dw1
        self.model["b1"] -= learning_rate * db1

        self.model["W2"] -= learning_rate * dw2
        self.model["b2"] -= learning_rate * db2

        self.model["W3"] -= learning_rate * dw3
        self.model["b3"] -= learning_rate * db3

    def predict(self, x) :
        y_out = self.forward(x)
        return np.argmax(y_out, axis=1)

    def summary(self) :
        W1,W2,W3 = self.model['W1'], self.model['W2'], self.model['W3']
        a1,a2,y_ = self.activation_outputs

        print("W1 ", W1.shape)
        print("A1 ", a1.shape)

        print("W2 ", W2.shape)
        print("A2 ", a2.shape)

        print("W3 ", W3.shape)
        print("Y_ ", y_.shape)
```

output_size

output coming from 1st hidden layer.

output coming from 2nd hidden layer.

output coming from output layer.

in forward propagation, when data flows through every layer, store the output in form of tuple and this is named as activation output.

y_ will have shape of m×C

We need to take average

a2 is simply tanh(z)

probability

$$\begin{bmatrix} 0.5 & 0.3 & 0.2 \\ & & \\ & & \end{bmatrix}$$

m × C

examples classes

For each example you will get with what probability it belongs to each class.

y_out is of (m, no)

$$\begin{array}{c} 1 \\ 2 \\ \vdots \\ m \end{array}\begin{bmatrix} - & - & - \\ & & \\ & & \end{bmatrix}$$

For 1st example, probabilities of each class.

Take argmax over this axis you will get class with highest probability

One hot vectors

probability

```
def loss(y_oht, p) :
    l = -np.mean(y_oht * np.log(p))
```

Loss function is going to be categorical cross entropy.

$$-\sum_{i=1}^{m} \sum_{n=1}^{c} y_{i,c} \log \hat{y}_{i,c}$$