CS550 HW3 Blog

How Neural Networks are trained?

Neural Networks are trained using **back-propagation**, which is basically computing the gradient of the loss function with respect to the weights of the network. It does this efficiently, and makes it feasible to apply gradient descent for training multi-layer networks.

Let's take a toy example to understand this concept.

Eg:

Consider below two functions, y = $\sigma(4 - 2x)$ and z = max (0.1 x, 2x)

Draw a neural network that accepts x as input and gives o as the output, using the below function. y and z. Also calculate the gradients if given w_1 = 2, w_2 = −1, w_3 = 6, x = 2, $\hat{o}$ = −1. Using a learning rate of η = 0.1, write the values of the parameter for the next iteration from the given data.

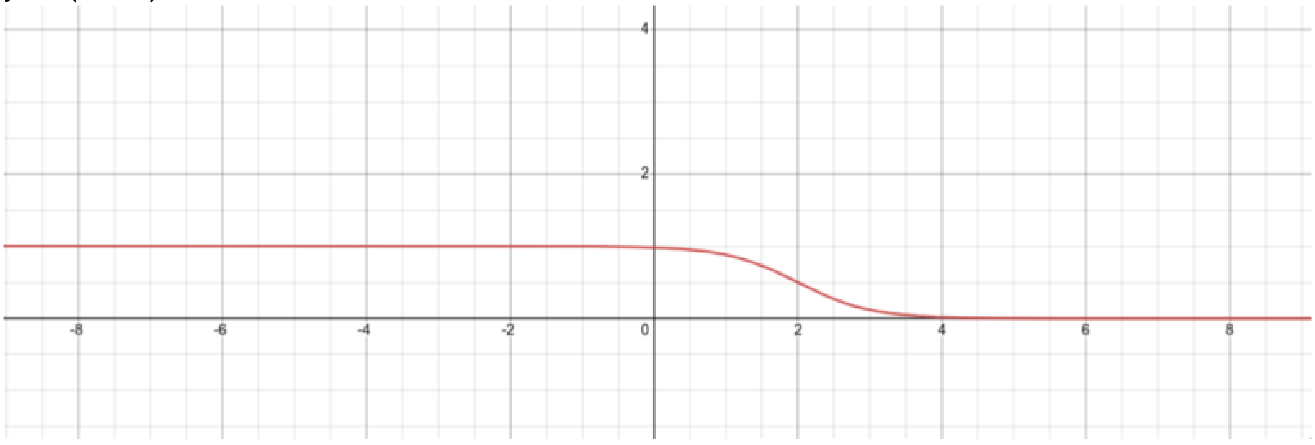
$$o = ELU(w_1 * y + w_2 * z + w_3)$$

$$\text{where, } ELU(x) = \begin{cases} x & \text{if } x \geq 0, \\ (e^x - 1) & \text{if } x < 0, \end{cases}$$

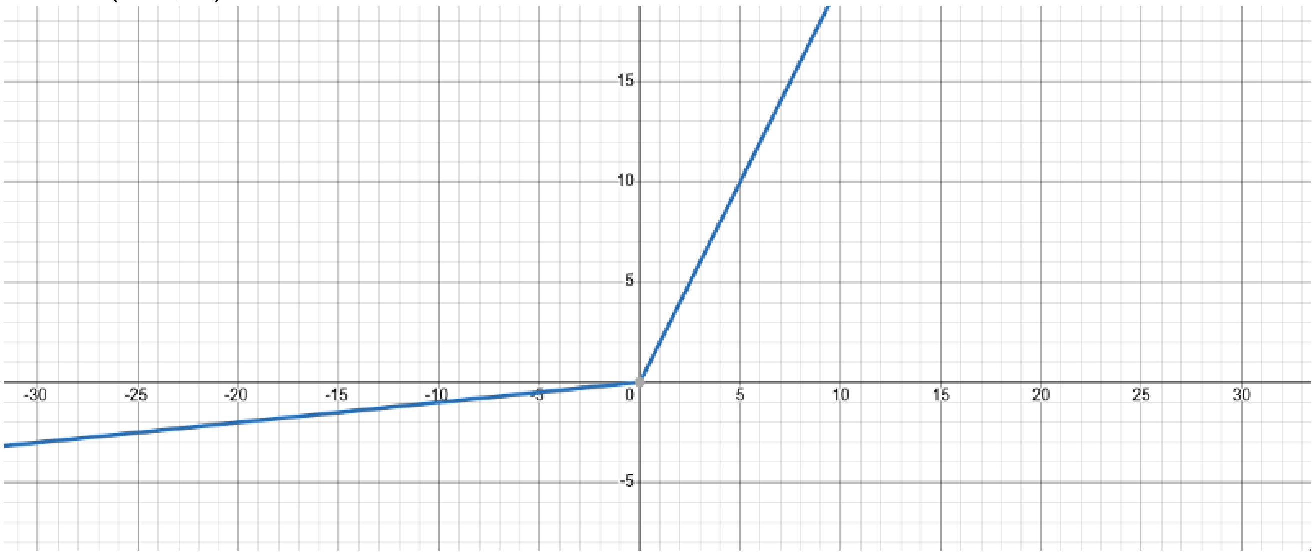
$$\text{Loss} = MSE(o, \hat{o})$$

Let's plot the two functions,

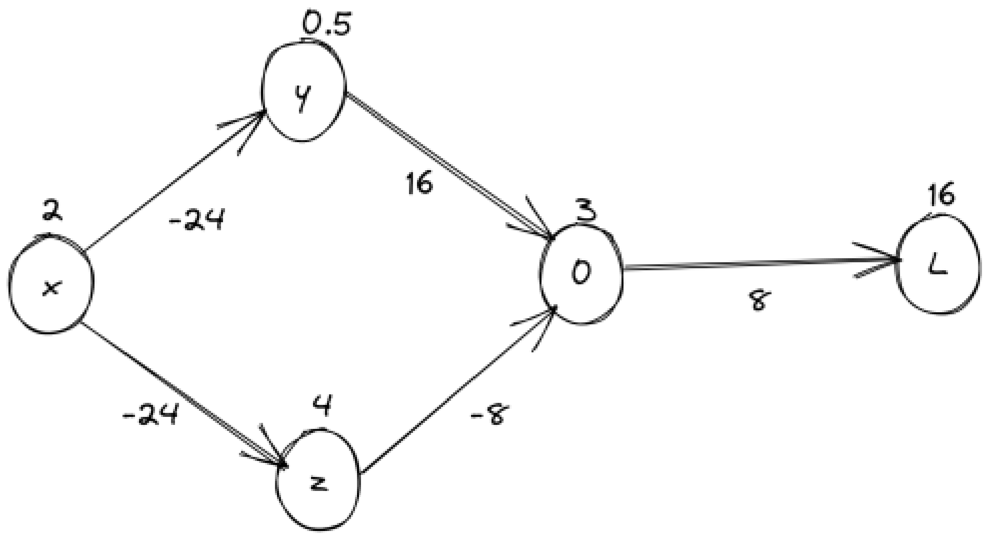
y = $\sigma(4 - 2x)$



z = max (0.1 x, 2x)



Let's draw the neural network,



Computing gradients

$$\begin{aligned}\frac{\partial y}{\partial x} &= 2 \times \tfrac{1}{2} \times \tfrac{-1}{2} = \tfrac{-1}{2} \\ \frac{\partial y}{\partial x} &= \tfrac{-1}{2} \text{ and } \tfrac{\partial z}{\partial x} = 2\end{aligned}$$

Now, $y = \frac{1}{2}$ and $z = 4$

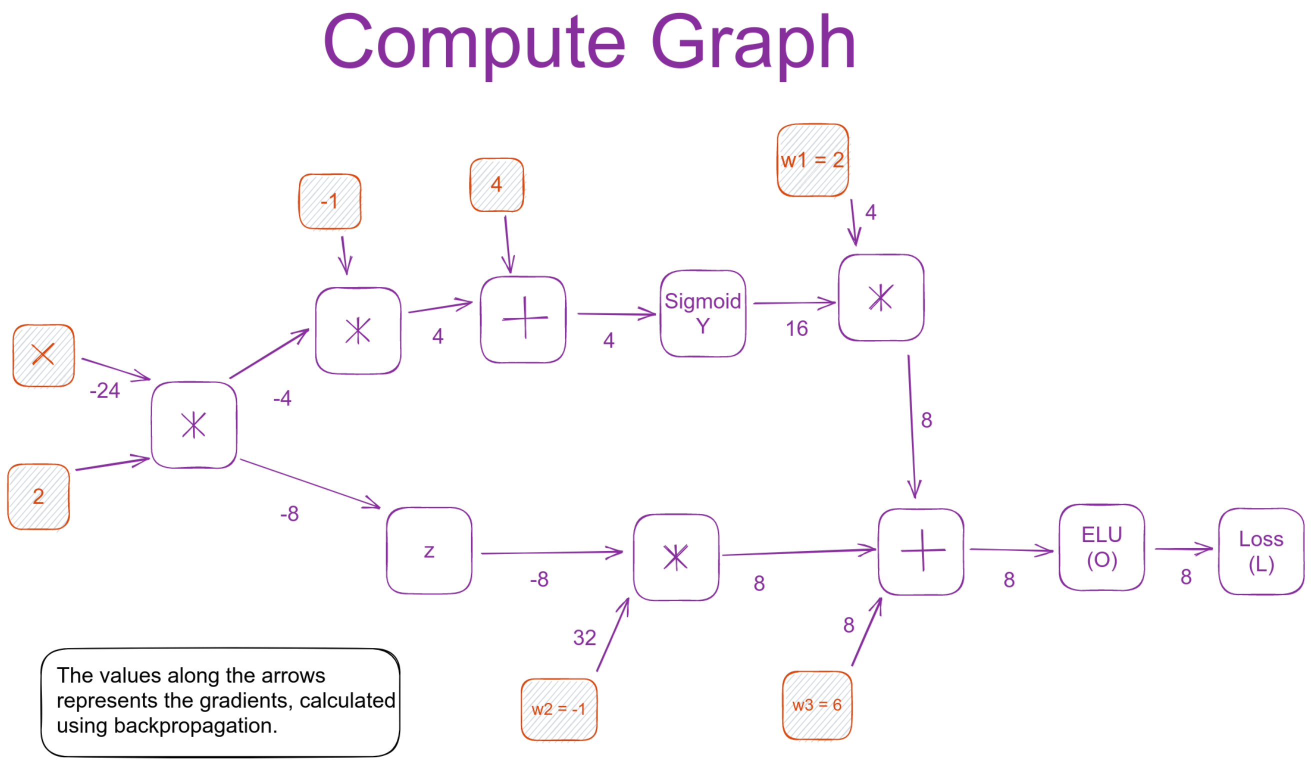
$$w_1 y + w_2 z + w_3 = 1 - 4 + 6 > 0$$
$$\therefore \tfrac{\partial o}{\partial y} = w_1 = 2$$
$$\tfrac{\partial o}{\partial z} = w_2 = -1$$
$$o = 3 \text{ and } \hat{o} = -1$$
$$\tfrac{\partial L}{\partial o} = 2(o - \hat{o}) = 2(3+1) = 8$$

and $L = 16$
Essentially, $L = f(y, z)$ and $y = g(x)$ and $z = h(x)$

$$\frac{\partial L}{\partial x} = (\frac{\partial L}{\partial y} \times \frac{\partial y}{\partial x}) + (\frac{\partial L}{\partial z} \times \frac{\partial z}{\partial x})$$

Now, $\frac{\partial L}{\partial y}$ = $\frac{\partial L}{\partial o} \times \frac{\partial o}{\partial y}$ and $\frac{\partial L}{\partial z} = \frac{\partial L}{\partial o} \times \frac{\partial o}{\partial z}$
Also, we get
$\frac{\partial L}{\partial y} = 8 \times 2 = 16$ and $\frac{\partial L}{\partial z} = 8 \times -1 = -8$
Finally, $\frac{\partial L}{\partial x} = -16 - 8 = -24$
$\frac{\partial L}{\partial w_1} = 8y$
$\frac{\partial L}{\partial w_2} = 8z$
$\frac{\partial L}{\partial w_3} = 8$
The updated values of weights are :
$w_1 n = w_1 - \eta \frac{\partial L}{\partial w_1} = 2 - 0.4 = 1.6$
$w_2 n = w_2 - \eta \frac{\partial L}{\partial w_2} = -1 - 3.2 = -4.2$
$w_3 n = w_3 - \eta \frac{\partial L}{\partial w_3} = 6 - 0.8 = 5.2$

And the compute graph for the above neural network.

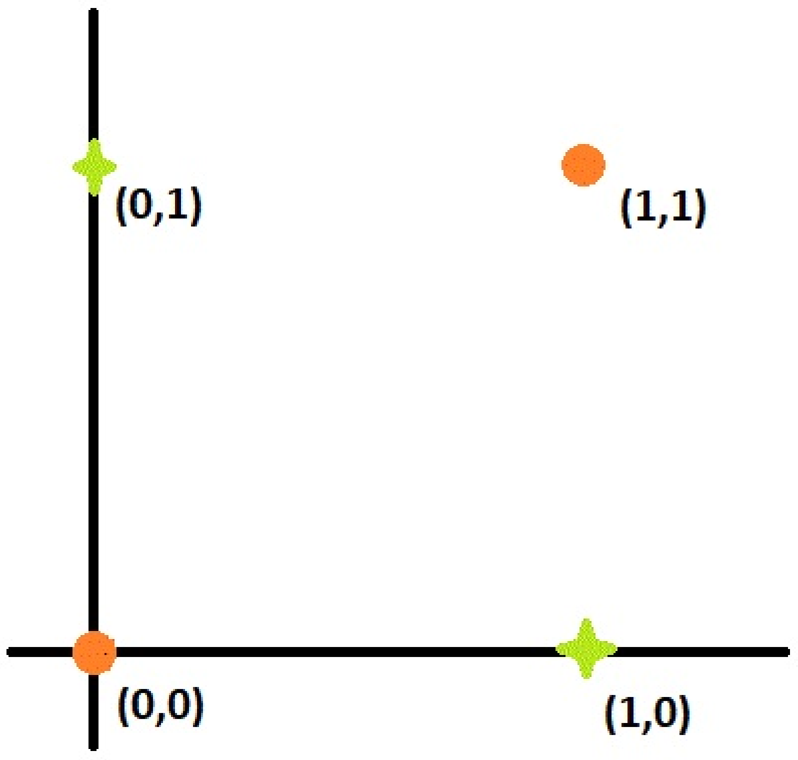

Compute Graph
X
2
-24
-4
-8
-1
4
4
4
Sigmoid Y
16
w1 = 2
4
8
z
-8
32
w2 = -1
8
8
w3 = 6
ELU (O)
8
Loss (L)
8
The values along the arrows represents the gradients, calculated using backpropagation.

How Neural networks compute Exclusive-OR of inputs?

This problem we can think as a binary classification problem where the outputs of xor of inputs,which are 0,1 are considered as two classes.

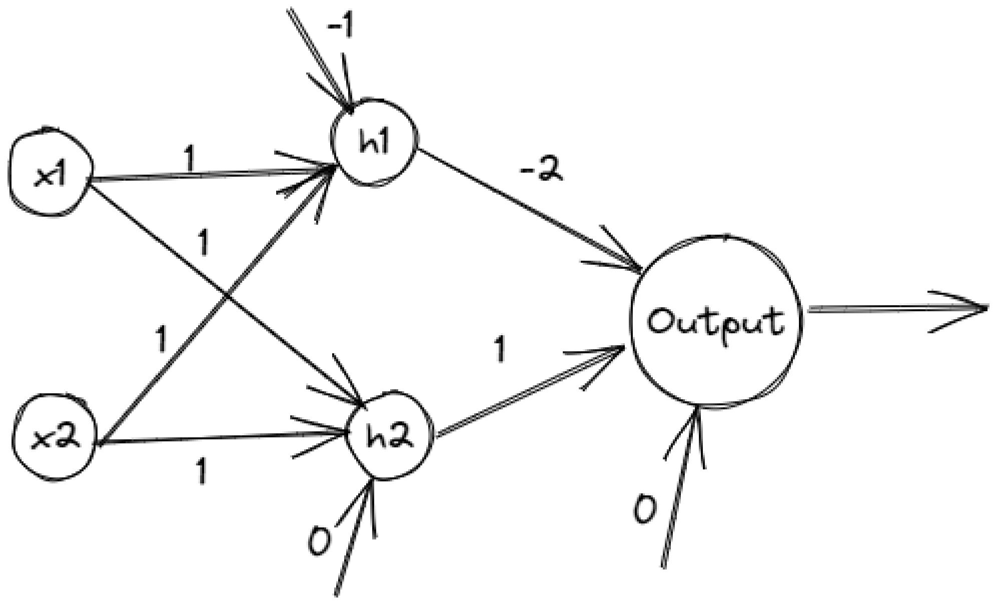
Let's consider a two input xor problem

A single perceptron neuron can only classify linearly seperable data ,hence it can not solve two input xor problem.This is because when we plot the input combinations for two classes,these classes are not linearly seperable.let's how we can plot,



In the above plot orange colour circles represents class 0 and green colour stars represents class 1 and this is clearly visible that these two classes are not linearly seperable because a single straight line can not seperate the two classes perfectly.But with multilayer perceptron we can solve this problem.Let's see how we can solve,

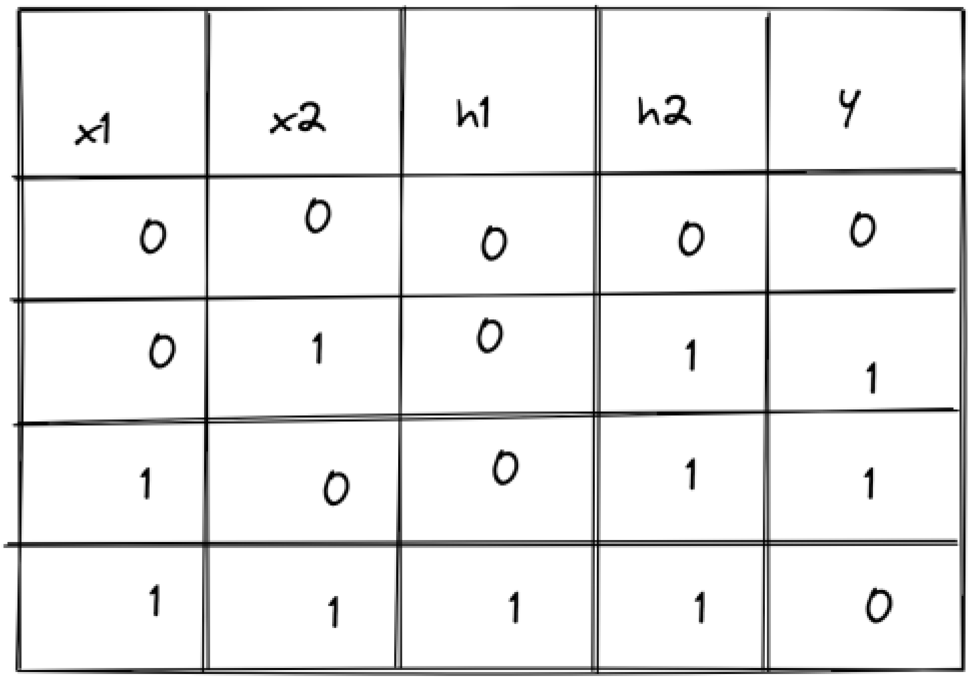
With two inputs the neural network is



In the above network the activation function for hidden layer neurons and output neuron is step function.The output of step function is 1 if the input is greater than zero and the output is zero if the input is less than or equal to zero

$$Step(x) = \begin{cases} 1 \text{ if } x > 0 \\ 0, otherwise \end{cases}$$

The below table shows the working of above neural network

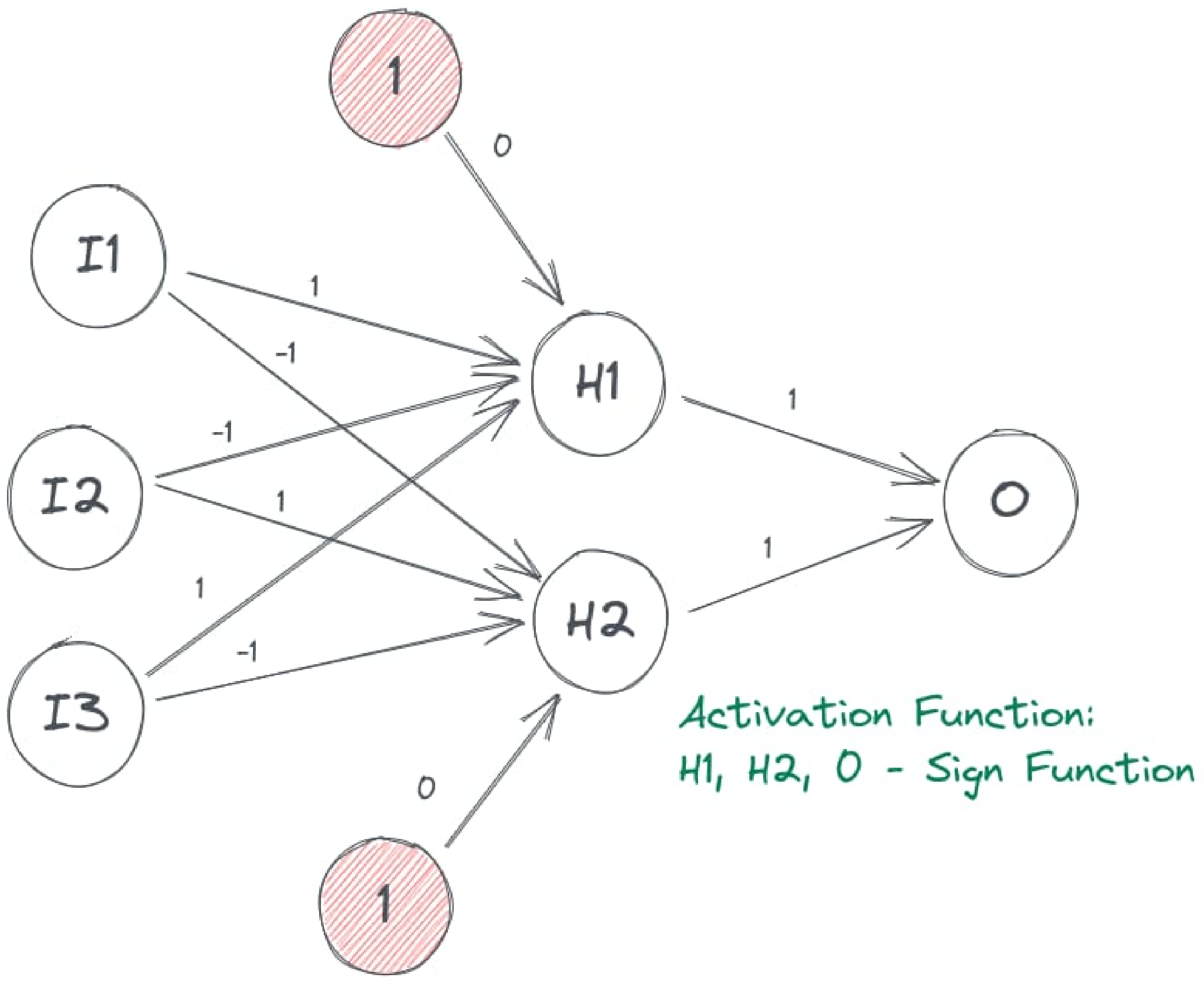


x1	x2	h1	h2	y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

$x1$ - input1
$x2$ - input2
$h1$ - 1st hidden layer neuron1
$h2$ - 1st hidden layer neuron2
y - output

Now consider a three input xor problem

With three inputs the neural network is

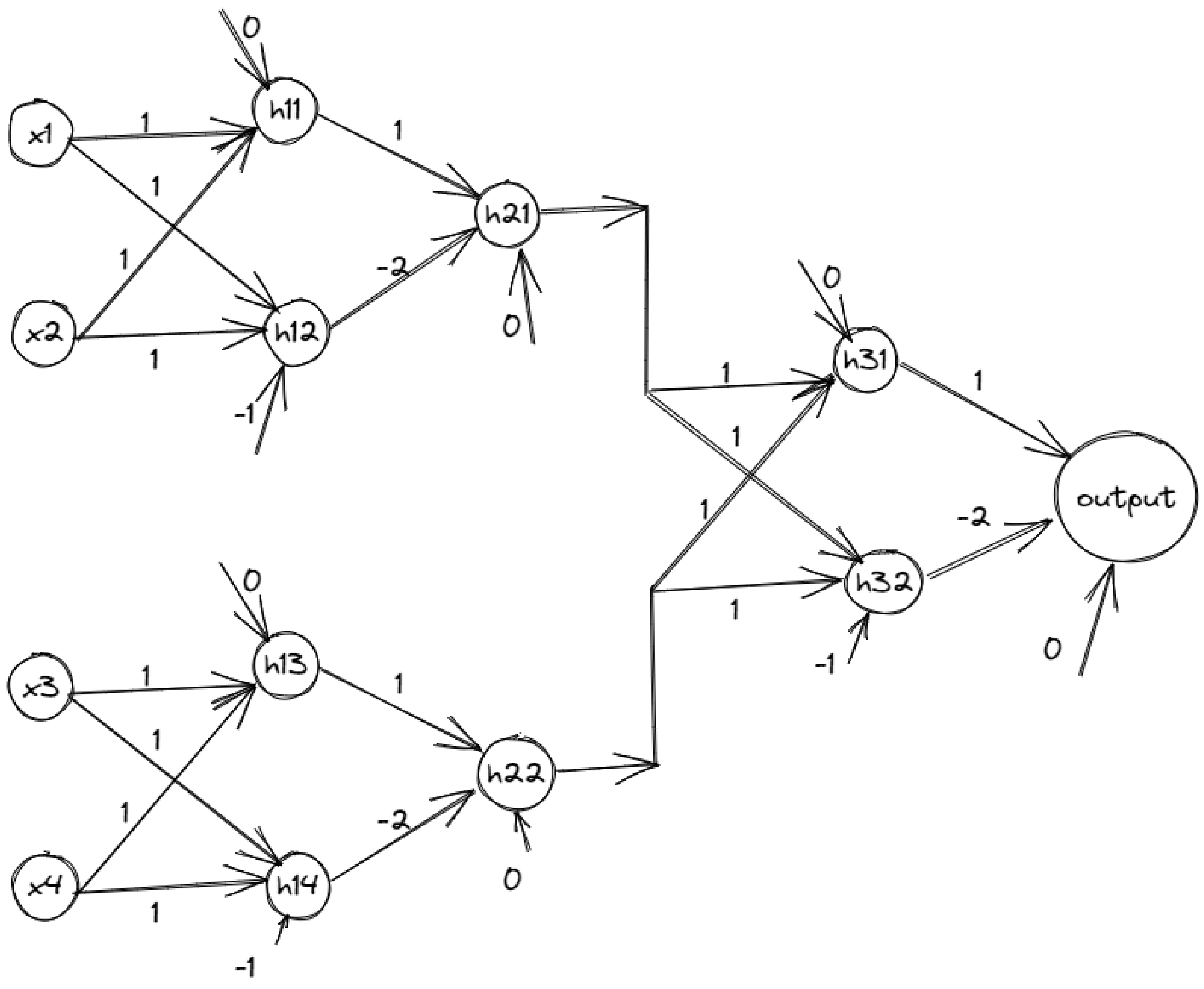


The above neural network identifies the odd number of ones in the input and gives output as 1.If number of ones in the input is even it gives 0 as outut.

eg: consider input combination as 1,1,1.For this input the output of h1 will be 1 and h2 will be 0 and the final output is 1

To generalize for n inputs we can take help of the neural network that we used for two inputs case.
For n inputs we will take two inputs at a time and apply same logic that we applied for two input case and we will proceed further.
To complete the problem, we need $((2 * log_2^n) - 1)$ hidden layers by taking two_two inputs from n inputs.And the no of neurons in a hidden layers are $n, n/2, n/2, n/4, n/4, n/8, n/8.\dots\dots$

Eg: consider 4 inputs, the neural network will be,



The no of hidden layers in the above network is 3.The no of neurons in hidden layers are 4,2,2

How to back propagate through neural network if the output layer is softmax and loss function is cross entropy?

To understand this let's solve the below problem.

Eg:

SOFTMAX + CROSS-ENTROPY LOSS

For classification, it is a common practice to use Softmax activation in the final layer followed by cross entropy loss. In this question, you have to compute the gradient of the combined operator.
Suppose the input to the combined operator is a vector y.
let ,$q = \mu(y) and \mu(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$
$l = H(p, q) = -\sum_{i=1}^{n} p_i log q_i$ where p represents true probability vector
You have to compute the gradient of l w.r.t. y, ∇_l^y in terms of only p and q.

Solution

Let x_i is the input to softmax layer then output of softmax layer is ,

$$q = \mu(y) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

cross entropy loss is,

$$l = H(p,q) = -\sum_{i=1}^{n} p_i log q_i$$

$$l = -\sum_{i=1}^{n} p_i \left(\log(e^{y_i}) - \log(\sum_{j=1}^{n} e^{y_j})\right)$$

$$l = -\sum_{i=1}^{n} \left(p_i y_i - p_i \log\left(\sum_{j=1}^{n} e^{y_j}\right)\right)$$

$$l = -\sum_{i=1}^{n} p_i y_i + \sum_{i=1}^{n} \left(p_i \log \sum_{j=1}^{n} e^{y_j}\right)$$

$$l = -\sum_{i=1}^{n} p_i y_i + \left(\log \sum_{j=1}^{n} e^{y_j}\right) \sum_{i=1}^{n} p_i$$

$$l = -\sum_{i=1}^{n} p_i y_i + \left(\log \sum_{j=1}^{n} e^{y_j}\right) \; as \sum_{i=1}^{n} p_i = 1$$

Now ,

$$\frac{\delta l}{\delta y_k} = -p_k + \frac{e^{y_k}}{\sum_{j=1}^{n} e^{y_j}}$$

$$\frac{\delta l}{\delta y_k} = -p_k + q_k$$

This tells us that the gradient of cross entropy loss function is simply subtraction of true probability vector and predicted probability vector
Therefore, we end up with the neat result:
$\nabla_y l = q - p$

How image dimension will change in CNN?

when CNN filters are applied on an image, the original image is reshaped based on the formula,

$$\begin{gathered} n_{out} = \lfloor \frac{n_{in} + 2p - k}{s} \rfloor + 1 \\ \text{where,} \end{gathered}$$

$$\begin{aligned} n_{in} &= \quad \text{number of input features} \\ n_{out} &= \quad \text{number of output features} \\ k &= \quad \text{convolution kernel size} \\ p &= \quad \text{convolution padding size} \\ s &= \quad \text{convolution stride size} \end{aligned}$$

Let's, try to understand it using a question,

Eg:

Suppose we have images of size 512×512 pixels and we use a filter that is 3×3. Calculate the output image dimension for this layer of a CNN?

Given Image Dimension = 512×512
Given Filter Dimension = 3×3
n_{in}= 512, k = 3,
p = 0, s = 1 (*Default*)

$n_{out} = \frac{512+0-3}{1} + 1$ = 510
Output Dimension of the image = 510×510

Calculate how much zero padding is necessary to produce an output of size equal to the input?

with Zero padding the output obtained after applying filter is

$$\begin{aligned} n_{out} = & \quad \lfloor \tfrac{n_{in}+2p-k}{s} \rfloor + 1 \\ as, & \quad n_{out} == n_{in}, \\ n_{in} = & \quad \lfloor \tfrac{n_{in}+2p-k}{s} \rfloor + 1 \\ 512 = & \quad \lfloor \tfrac{512+2p-3}{1} \rfloor + 1 \\ 512-1 = & \quad 512+2p-3 \\ 2p = & \quad 2 \\ p = & \quad 1 \end{aligned}$$

To get output dimension same as input the padding required is 1.

Suppose we don’t do any zero padding. If the output of one layer is the input to the next layer, after how many layers will there be no output at all?

Starting from a 512×512 dimension image, if we apply a 3×3 filter we will get 510×510 dimension image if we again apply it become 508×508.

Following the pattern,
512×512 -> $510] \times 510$ -> 508×508 --> 2×2

if we apply a filter 3×3 on a 2×2 image we will get no output. Hence to get no output the number of layers needed will be $512/2 = 256$.

Alternative Solution
This we can solved using sum of arithmetic progression formula:
For the Series: $510, 508, \dots .2$

$$\begin{aligned} a_n = & \quad a + (n-1)d \\ 2 = & \quad 510 + (n-1) * (-2) \\ n = & \quad 255 \end{aligned}$$

So, the number of layers required for producing no output = $255 + 1 = 256$ layers. But after 255th layer we should not apply filter of size 3×3 as input image size for this layer is 2×2

How convolutional neural networks learns different features from the image?

Have you ever think of how a CNN extracts features from input image?

CNNs uses filters to do so.The filter weights are learned by the CNNs through back propagation.
Let's look at the below example to get a brief Idea about what should be the weights of filter to extract a particulat feature from image.

Eg:

Assume that inputs are single bits 0 (white)
and 1 (black). Consider a 3 × 3 filter, whose weights are wij , for 0 ≤ i ≤ 2 and
0 ≤ j ≤ 2, and whose bias is b. Suggest wieghts and bias so that the output of
this filter would detect the following simple features:
(a) A vertical boundary, where the left column is 0, and the other two columns
are 1.

(b) A diagonal boundary, where only the triangle of three pixels in the upper right corner are 1.
(c) a corner, in which the 2 × 2 square in the lower right is 0 and the other pixels are 1.

Solution

---

a)

Vertical boundary detection

Filter $= \begin{bmatrix} -1 & 1 & 1 \\ -1 & 1 & 1 \\ -1 & 1 & 1 \end{bmatrix}$ and bias $b = -6$
Activation function is step function

Diagonal boundary detection

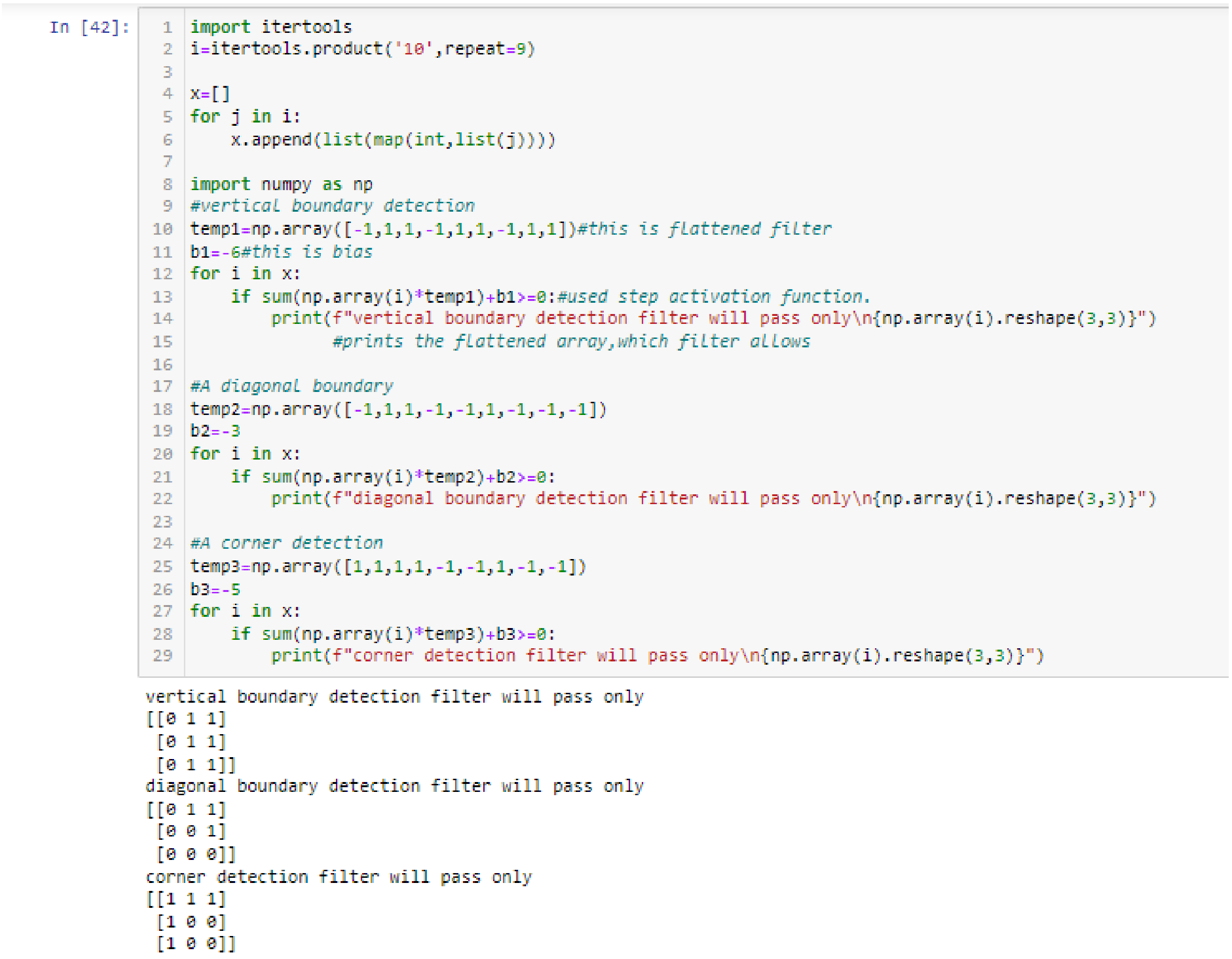
Filter $= \begin{bmatrix} -1 & 1 & 1 \\ -1 & -1 & 1 \\ -1 & -1 & -1 \end{bmatrix}$, and bias $b = -3$
Activation function is step function

c)

Corner detection

Filter $= \begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & -1 \\ 1 & -1 & -1 \end{bmatrix}$, and bias $b = -5$
Activation function is step function

The following python program proves the correct working of filters to produce desired output.



```
In [42]:
import itertools
i=itertools.product('10',repeat=9)

x=[]
for j in i:
    x.append(list(map(int,list(j))))

import numpy as np
#vertical boundary detection
temp1=np.array([-1,1,1,-1,1,1,-1,1,1])#this is flattened filter
b1=-6#this is bias
for i in x:
    if sum(np.array(i)*temp1)+b1>=0:#used step activation function.
        print(f"vertical boundary detection filter will pass only\n{np.array(i).reshape(3,3)}")
              #prints the flattened array,which filter allows

#A diagonal boundary
temp2=np.array([-1,1,1,-1,-1,1,-1,-1,-1])
b2=-3
for i in x:
    if sum(np.array(i)*temp2)+b2>=0:
        print(f"diagonal boundary detection filter will pass only\n{np.array(i).reshape(3,3)}")

#A corner detection
temp3=np.array([1,1,1,1,-1,-1,1,-1,-1])
b3=-5
for i in x:
    if sum(np.array(i)*temp3)+b3>=0:
        print(f"corner detection filter will pass only\n{np.array(i).reshape(3,3)}")
```

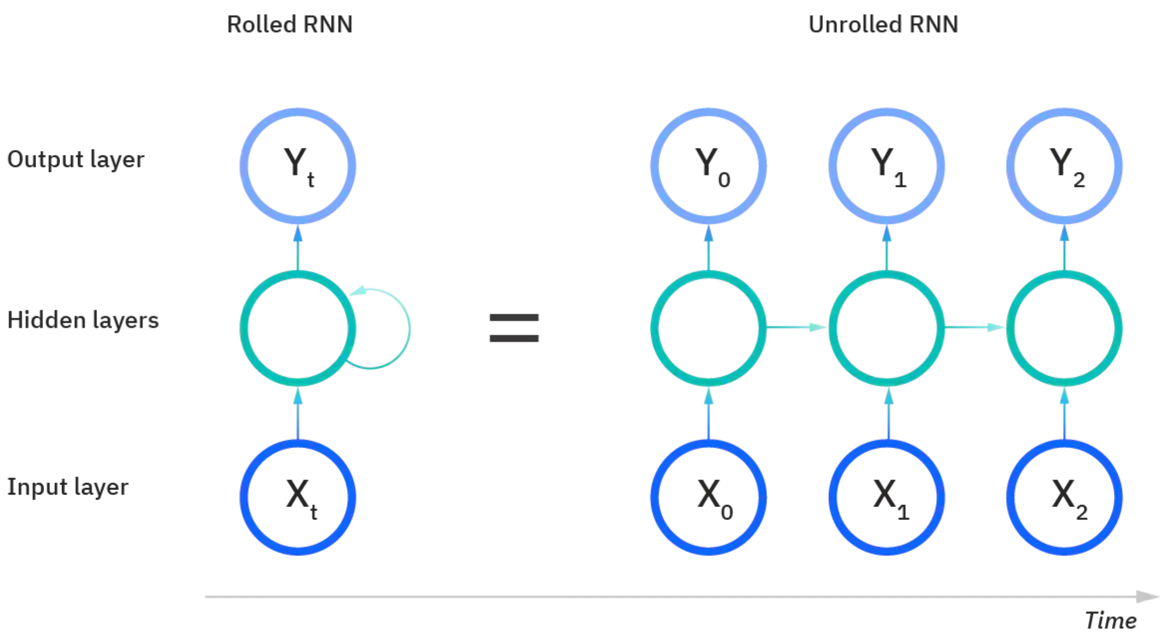
```
vertical boundary detection filter will pass only
[[0 1 1]
 [0 1 1]
 [0 1 1]]
diagonal boundary detection filter will pass only
[[0 1 1]
 [0 0 1]
 [0 0 0]]
corner detection filter will pass only
[[1 1 1]
 [1 0 0]
 [1 0 0]]
```

Recurrent neural networks

Do you know what recurrent neural networks are?

A recurrent neural network (RNN) is a type of artificial neural network which uses sequential data or time series data.These deep learning algorithms are commonly used for ordinal or temporal problems,such as language translation, natural language processing (nlp), speech recognition, and image captioning; they are incorporated into popular applications such as Siri, voice search, and Google Translate.

Let's have a look at how recurrent neural networks are represented



Recurrent neural networks have weight sharing property.To understand this
let's solve a simple problem of finding the weights of RNN to get desired output.

Eg:

In this example, you are asked to design the input weights for one or more nodes of the hidden state of an RNN. The input is a sequence of bits, 0 or 1 only. Note that you can use other nodes to help with the node requested. Also note that you can apply a transformation to the output of the node so a “yes” answer has one value and a “no” answer has another.
a. A node to signal when the input is 1 and the previous input is 0.
b. A node to signal when the last three inputs have all been 1.
c. A node to signal when the input is the same as the previous input.

Solution

a.

With one previous state:
output should be 1 if the present input is 1 and previous input is 0
y_t = VS_t ,bias is taken as zero
y_t = $V(ui_t + ws_{t-1})$,bias is taken as zero
y_t = $uVi_t + Vwui_{t-1} + Vb$,where $s_{t-1} = ui_{t-1} + b$
now take u=v=2,w=-1,b=-1
Cross checking with different input combinations

for (0,0):

where first entry in the tuple represents previous input value
and second entry represents present input value.
y_t= 0 + 0 - 1 = -1
considering the activation function as step function the output will be 0

FOR (0,1):

y_t= 2 + 0 - 1 = 1
considering the activation function as step function the output will be 1

FOR (1,0):

y_t=0-2-1= -3
considering the activation function as step function the output will be 0

FOR (1,1):

y_t=1-1-1=-1
considering the activation function as step function the output will be 0.
In all the possible cases the output, with assumed weights ,is 1 only for (0,1)
where the present input is 1 and previous input is 0.
Consider 'yes ' for output 1 and "no" for output 0.

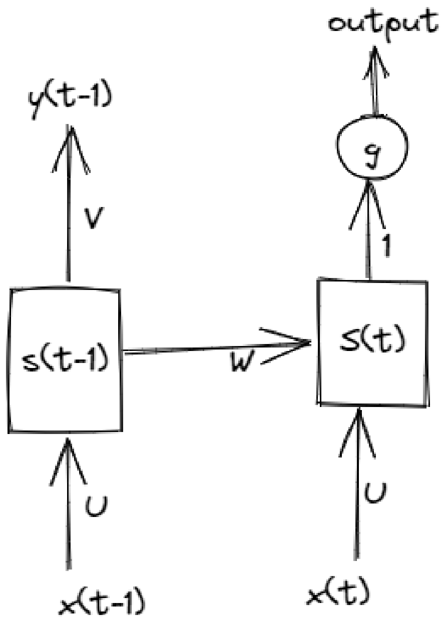
b.

with three previous states:
Y_t=$V(S_t) + b$
Y_t=$V(Ui_t + WS_{t-1})$
by expanding this equation for three previous states we can write
Y_t=$UVi_t + UVWi_{t-1} + UVW^2i_{t-2} + UVW^3i_{t-3} + b$
where the bias in all the intermediate steps is treated as zero.
Consider U=V=1 and W=2,b=-13
with all the three previous inputs 1 and present input is 0, Y_t=1
with all the three previous inputs 1 and present input is 1, Y_t=2
with activation function as step function the output for above two cases wil be 1.
for all other possible cases the Y_t<=0
So with activation function as step function the output for these cases wil be 0.

c.

with one previous input state:
The question looks like XOR problem ,to solve this we can take help of hidden state.

Reccurent nerual network



Note: function g is even detector
It will give 1 as output only if it's input is even

Assume u=v=w=1,
$s(t) = ws(t-1) + ux(t) + b_1$
$s(t-1) = ux(t-1) + b_2$
hence $s(t) = x(t-1) + x(t) + b_1 + b_2$
and $output = g(s(t))$
let $b_1 = b_2 = 1$

now check for different combinations of input:

For (0,0):

output=g(2)=1 as g is even detector

For (0,1):

output=g(3)=0 as g is even detector

For (1,0):

output=g(3)=0 as g is even detector

For (1,1):

output=g(4)=1 as g is even detector

> Note:if bias is not written in the formula,then assume that the bias is taken as zero.