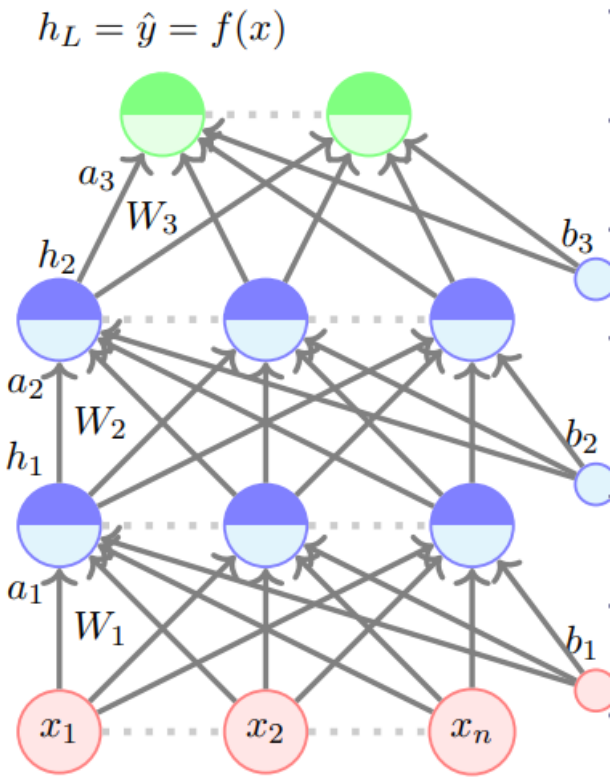
Deep Learning Assaignment-1

1. What is Feedforward Neural Networks?



- The input to the network is an n-dimensional vector
- The network contains L − 1 hidden layers (2, in this case) having n neurons each
- Finally, there is one output layer containing k neurons (say, corresponding to k classes)
- Each neuron in the hidden layer and output layer can be split into two parts : pre-activation and activation (ai and hi are vectors)
- The input layer can be called the 0-th layer and the output layer can be called the (L)-th layer
- $W_i \in R^{n \times n}$ and $b_i \in R^n$ are the weight and bias between layers i − 1 and i ($0 < i < L$)
- $W_L \in R^{n \times k}$ and $b_L \in R^k$ are the weight and bias between the last hidden layer and the output layer (L = 3 in this case)
- The pre-activation at layer i is given by

  $$a_i(x) = b_i + W_i h_{i-1}(x)$$
- The activation at layer i is given by

  $$h_i = g(a_i)$$

  where g is called the activation function (for example, logistic, tanh, linear, etc.)
- The activation at the output layer is given by

  $$f(x) = h_L = O(a_L)$$

  where O is the output activation function (for example, softmax, linear, etc.)

- **Data:** $\{x_i, y_i\}^N_{i=1}$
- **Model:**

$$\hat{y}_i = f(x_i) = O(W_3 g(W_2 g(W_1 x + b_1) + b_2) + b_3)$$

- **Parameters:**

$$\theta = W_1, .., W_L, b_1, b_2, ..., b_L (L = 3)$$

- **Algorithm:** Gradient Descent with Backpropagation (we will see soon)
- **Objective/Loss/Error function:** Say,

$$min \ \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{k} (\hat{y}_{ij} - y_{ij})^2$$

$$In \ general, \ min \ \mathscr{L}(\theta)$$

where L (θ) is some function of the parameters

Learning Parameters of Feedforward Neural Networks (Intuition):

- Recall our gradient descent algorithm
- We can write it more concisely as

Algorithm: gradient_descent()

$t \leftarrow 0$;
$max\_iterations \leftarrow 1000$;
$Initialize \quad \theta_0 = [W_1^0, ..., W_L^0, b_1^0, ..., b_L^0]$;
while $t{+}{+} < max\_iterations$ **do**
  $\theta_{t+1} \leftarrow \theta_t - \eta \nabla \theta_t$;
end

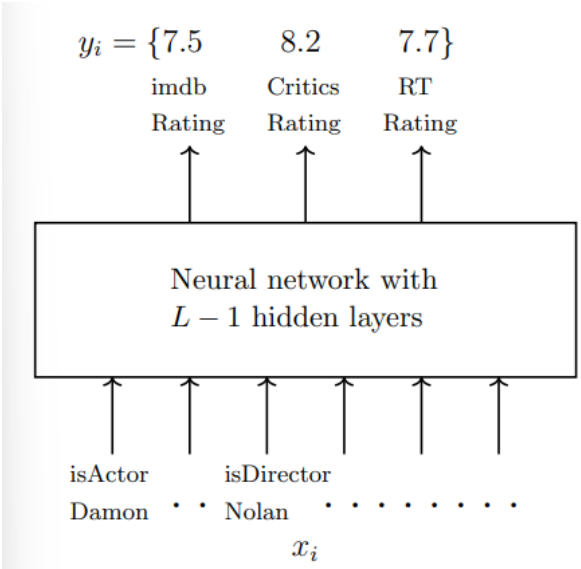
where $\nabla \theta_t = [\frac{\partial \mathscr{L}(\theta)}{\partial W_{1,t}}, .., \frac{\partial \mathscr{L}(\theta)}{\partial W_{L,t}}, \frac{\partial \mathscr{L}(\theta)}{\partial b_{1,t}}, .., \frac{\partial \mathscr{L}(\theta)}{\partial b_{L,t}}]^T$

- Now, in this feedforward neural network, instead of θ = *[w, b]* we have θ = $[W_1, W_2, .., W_L, b_1, b_2, .., b_L]$
- We can still use the same algorithm for learning the parameters of our model
- Except that now our ∇θ looks much more nasty

$$\begin{bmatrix} \frac{\partial \mathscr{L}(\theta)}{\partial W_{111}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{11n}} & \frac{\partial \mathscr{L}(\theta)}{\partial W_{211}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{21n}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{L,11}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{L,1k}} & \frac{\partial \mathscr{L}(\theta)}{\partial W_{L,1k}} & \frac{\partial \mathscr{L}(\theta)}{\partial b_{11}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial b_{L1}} \\ \frac{\partial \mathscr{L}(\theta)}{\partial W_{121}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{12n}} & \frac{\partial \mathscr{L}(\theta)}{\partial W_{221}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{22n}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{L,21}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{L,2k}} & \frac{\partial \mathscr{L}(\theta)}{\partial W_{L,2k}} & \frac{\partial \mathscr{L}(\theta)}{\partial b_{12}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial b_{L2}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\partial \mathscr{L}(\theta)}{\partial W_{1n1}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{1nn}} & \frac{\partial \mathscr{L}(\theta)}{\partial W_{2n1}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{2nn}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{L,n1}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial W_{L,nk}} & \frac{\partial \mathscr{L}(\theta)}{\partial W_{L,nk}} & \frac{\partial \mathscr{L}(\theta)}{\partial b_{1n}} & \cdots & \frac{\partial \mathscr{L}(\theta)}{\partial b_{Lk}} \end{bmatrix}$$

- ∇θ is thus composed of
  $\nabla W_1, \nabla W_2, ...\nabla W_{L-1} \in R^{n \times n}$, $\nabla W_L \in R^{n \times k}$,
  $\nabla b_1, \nabla b_2, ..., \nabla b_{L-1} \in R^n$ and $\nabla b_L \in R^k$

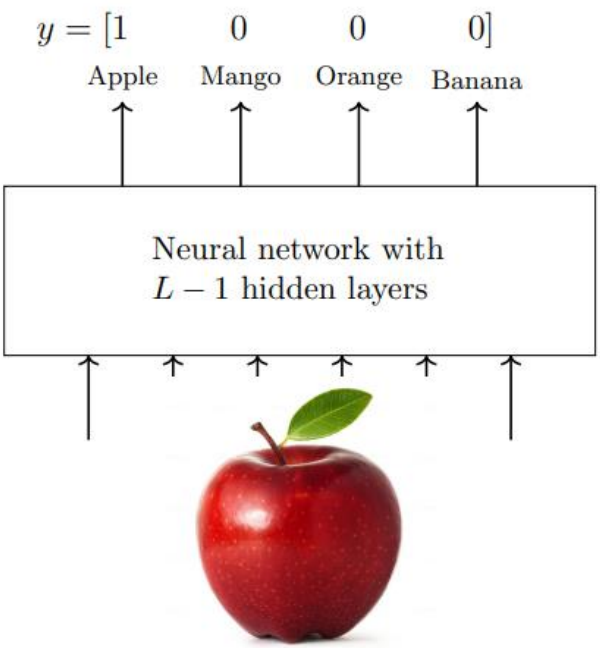
Output Functions and Loss Functions:



- The choice of loss function depends on the problem at hand
- We will illustrate this with the help of two examples
- Consider our movie example again but this time we are interested in predicting ratings
- Here $y_i \in R^3$
- The loss function should capture how much $\hat{y}_i$ deviates from y_i
- If $y_i \in R^n$ then the squared error loss can capture this deviation

$$\mathscr{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{3}(\hat{y}_{ij} - y_{ij})^2$$

- Example:
  - Now let us consider another problem for which a different loss function would be appropriate
  - Suppose we want to classify an image into 1 of k classes
  - Here again we could use the squared error loss to capture the deviation
  - But can you think of a better function?



- Notice that y is a probability distribution
- Therefore we should also ensure that ŷ is a probability distribution
- What choice of the output activation 'O' will ensure this ?

$$a_L = W_L h_{L-1} + b_L$$
$$\hat{y}_j = O(a_L)_j = \frac{e^{a_{L,j}}}{\sum_{i=1}^{k} e^{a_{L,i}}}$$

$O(a_L)j$ is the jth element of ŷ and a_L, j is the jth element of the vector a_L.

- This function is called the softmax function
- Now that we have ensured that both y & ŷ are probability distributions can you think of a function which captures the difference between them?
- Cross-entropy

$$\mathscr{L}(\theta) = -\sum_{c=1}^{k} y_c \log \hat{y}_c$$

- Notice that

$$y_c = 1 \quad \text{if } c = \ell \text{ (the true class label)}$$
$$= 0 \quad \text{otherwise}$$
$$\therefore \ \mathscr{L}(\theta) = -\log \hat{y}_\ell$$

- So, for classification problem (where you have to choose 1 of K classes), we use the following objective function

$$\underset{\theta}{\text{minimize}} \quad \mathscr{L}(\theta) = -\log \hat{y}_\ell$$
$$\text{or} \quad \underset{\theta}{\text{maximize}} \quad -\mathscr{L}(\theta) = \log \hat{y}_\ell$$

- But wait! Is ŷ` a function of $\theta = [W_1, W_2, ., W_L, b_1, b_2, ., b_L]$?
- Yes, it is indeed a function of θ

$$\hat{y}_\ell = [O(W_3 g(W_2 g(W_1 x + b_1) + b_2) + b_3)]_\ell$$

- What does ŷ` encode? It is the probability that x belongs to the ` th class (bring it as close to 1).
- log ŷ` is called the log-likelihood of the data.

	Outputs	
	Real Values	Probabilities
Output Activation	Linear	Softmax
Loss Function	Squared Error	Cross Entropy

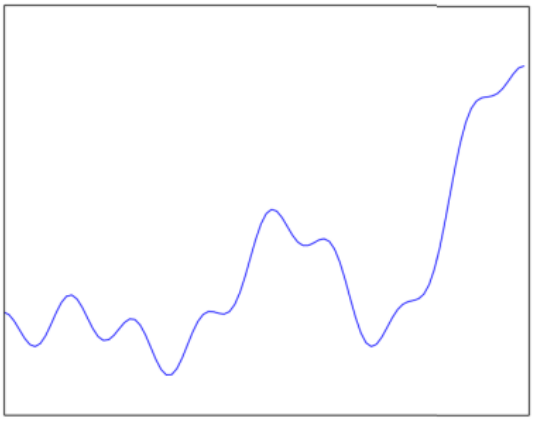
- Of course, there could be other loss functions depending on the problem at hand but the two loss functions that we just saw are encountered very often

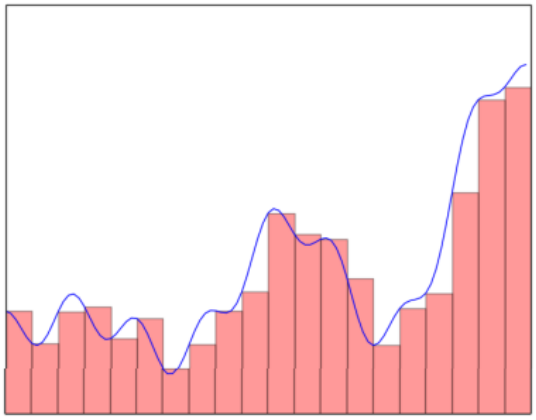
- For the rest of this lecture we will focus on the case where the output activation is a softmax function and the loss function is cross entropy

2.Write about Representation Power of Feedforward Neural Networks?

Representation power is related to ability of a neural network to assign proper labels to a particular instance and create well defined accurate decision boundaries for that class. In this article we will explore a visual approach for learning more about approximating behavior of a neural network which is in direct relation to representation power of neural network.

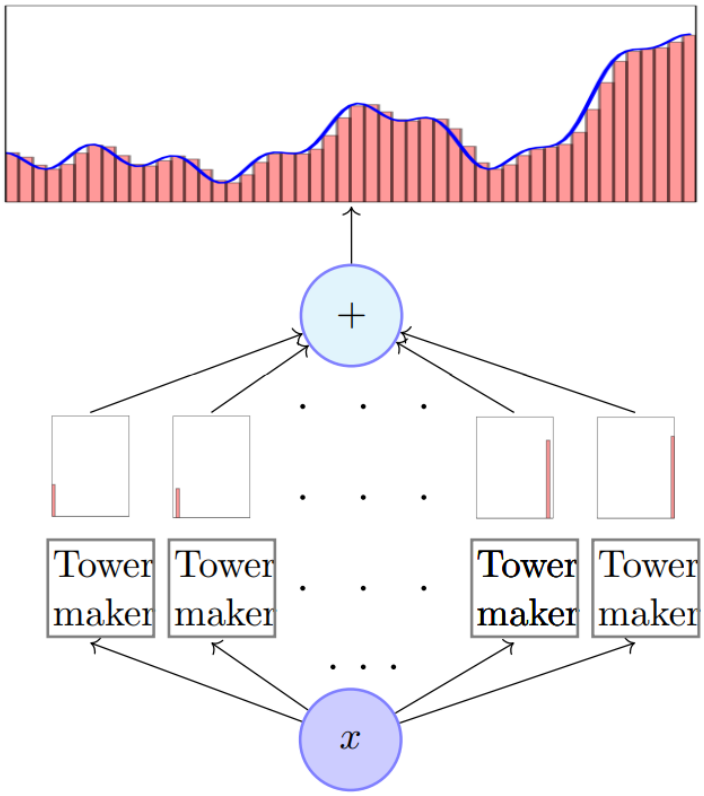
- Representation power of a multilayer network of sigmoid neurons
- A multilayer network of neurons with a single hidden layer can be used to approximate any continuous function to any desired precision
- In other words, there is a guarantee that for any function $f(x) : R^n \rightarrow R^m$, we can always find a neural network (with 1 hidden layer containing enough neurons) whose output $g(x)$ satisfies $|g(x)-f(x)| <$!!
- **Proof:** We will see an illustrative proof of this...



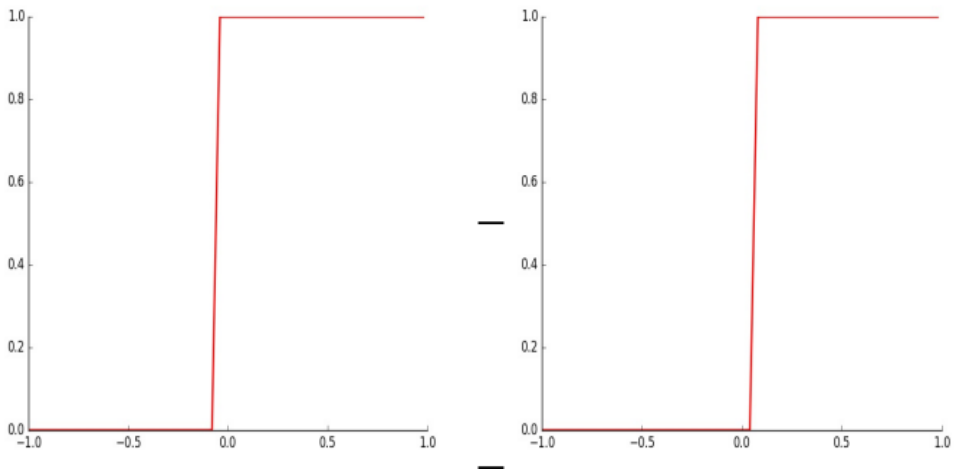


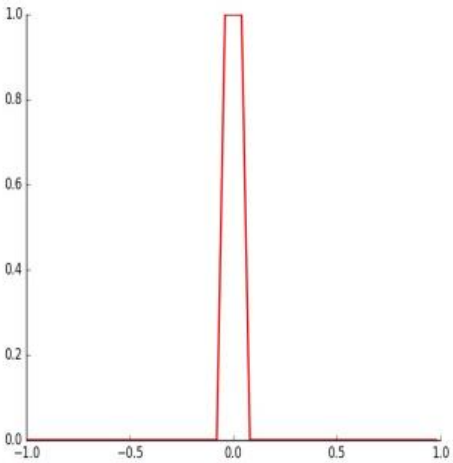
- We are interested in knowing whether a network of neurons can be used to represent an arbitrary function (like the one shown in the figure)
- We observe that such an arbitrary function can be approximated by several "tower" functions
- More the number of such "tower" functions, better the approximation

- To be more precise, we can approximate any arbitrary function by a sum of such "tower" functions



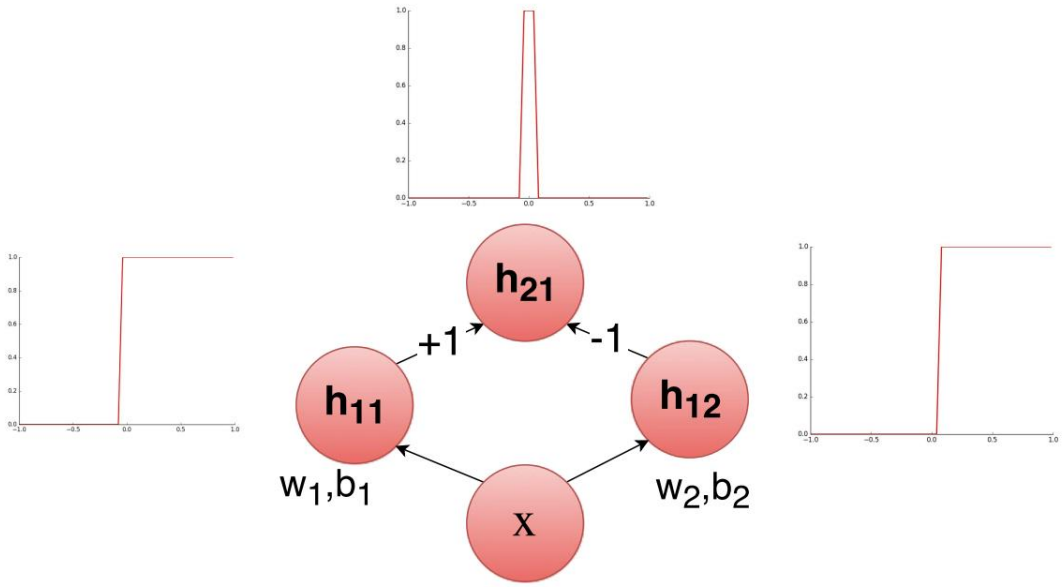
- We make a few observations
- All these "tower" functions are similar and only differ in their heights and positions on the x-axis
- Suppose there is a black box which takes the original input (x) and constructs these tower functions
- We can then have a simple network which can just add them up to approximate the function
- Our job now is to figure out what is inside this blackbox

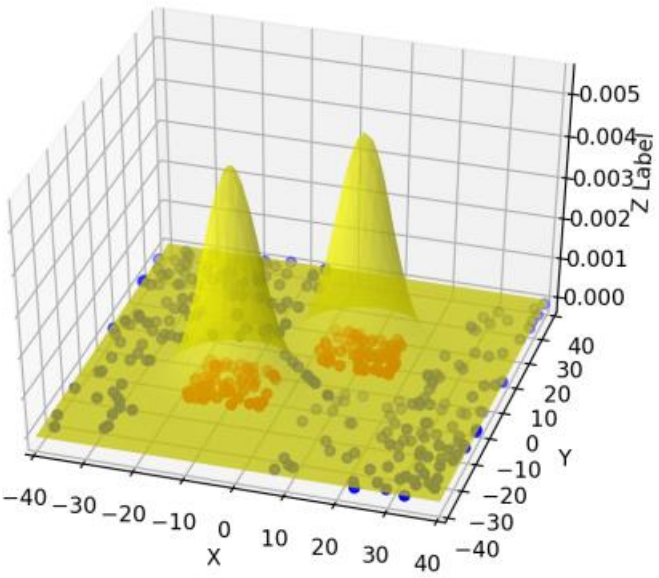




- If we take the logistic function and set w to a very high value we will recover the step function
- Let us see what happens as we change the value of w
- Further we can adjust the value of b to control the position on the x-axis at which the function transitions from 0 to 1
- Now let us see what we get by taking two such sigmoid functions (with different b 0 s) and subtracting one from the other
- Voila! We have our tower function !!

Can we come up with a neural network to represent this operation of subtracting one sigmoid function from another ?

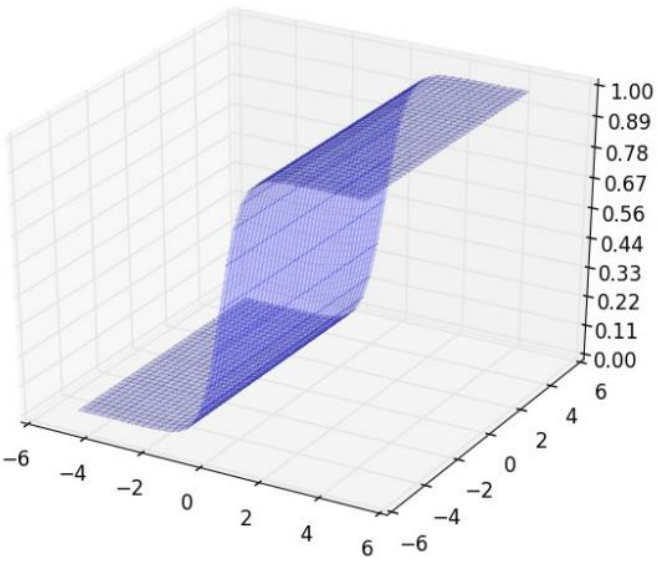




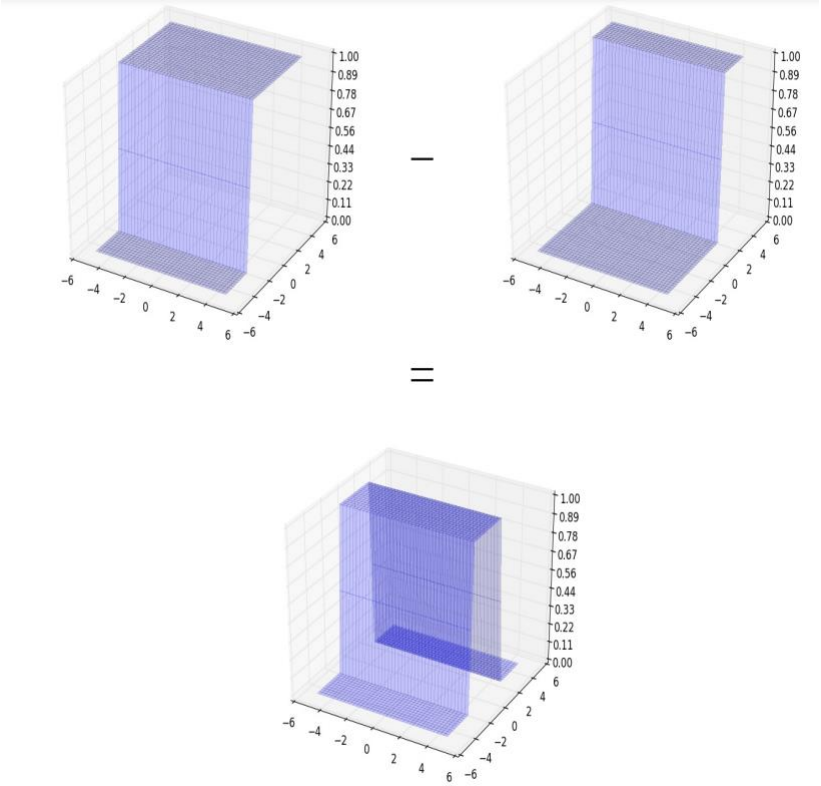
- What if we have more than one input?
- Suppose we are trying to take a decision about whether we will find oil at a particular location on the ocean bed(Yes/No)
- Further, suppose we base our decision on two factors: Salinity (x1) and Pressure (x2)
- We are given some data and it seems that y(oil|no-oil) is a complex function of x1 and x2
- We want a neural network to approximate this function

$$y = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + b)}}$$

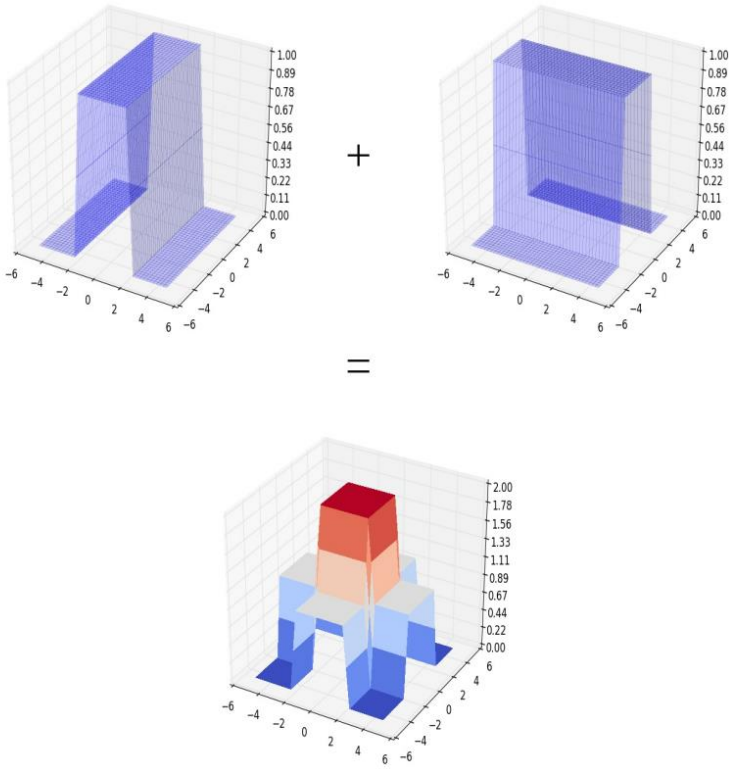
- This is what a 2-dimensional sigmoid looks like
- We need to figure out how to get a tower in this case
- First, let us set w2 to 0 and see if we can get a two dimensional step function



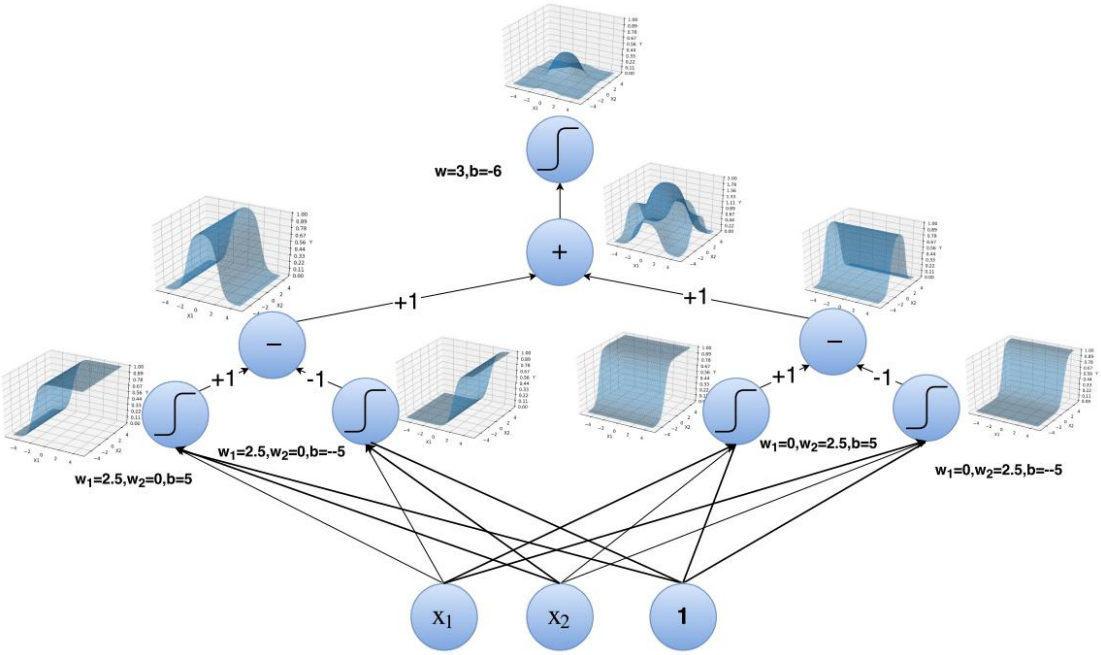
In our understanding we need to make such 3-D closed towers in a 3-D coordinate systems. If we carry on the similar approach mentioned above of subtraction of two sigmoids with different biases in 3-D space. We will get the following equivalent curve.



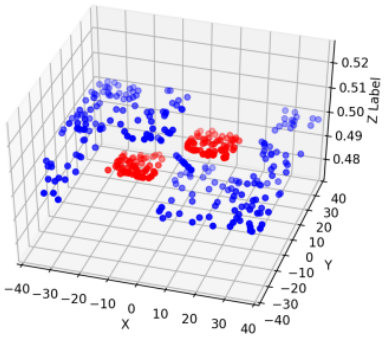
But, we can see that if we take another horizontally perpendicular tower to current resultant curve above. We can get the closed tower upon superimposing these two horizontally perpendicular open towers.

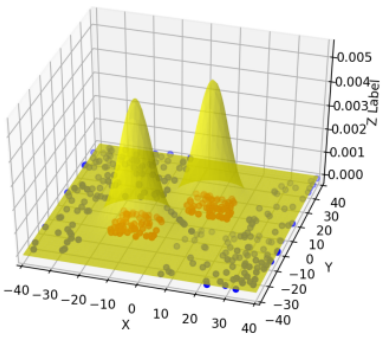


Can we come up with a neural network to represent this entire procedure of constructing a 3 dimensional tower ?



The illustrative proof that we just saw tells us that we can have a neural network with two hidden layers which can approximate the above function by a sum of towers. Which means we can have a neural network which can exactly separate such distributions like mentioned in above case study. There is no theoretical limitation upon the accuracy of neural networks.





We are interested in separating the blue points from the red points. With single sigmoidal neuron there are clearly errors that will exist. But, with two hidden layers we can approximate the above function by a sum of towers. We can have a neural network which can exactly separate the blue points from the red points !!