CS7015 (Deep Learning) : Lecture 3

Sigmoid Neurons, Gradient Descent, Feedforward Neural Networks, Representation Power of Feedforward Neural Networks

Mitesh M. Khapra

Department of Computer Science and Engineering
Indian Institute of Technology Madras

Acknowledgements

- For Module 3.4, I have borrowed ideas from the videos by Ryan Harris on "visualize backpropagation" (available on youtube)
- For Module 3.5, I have borrowed ideas from this excellent book [a] which is available online
- I am sure I would have been influenced and borrowed ideas from other sources and I apologize if I have failed to acknowledge them

[a] http://neuralnetworksanddeeplearning.com/chap4.html

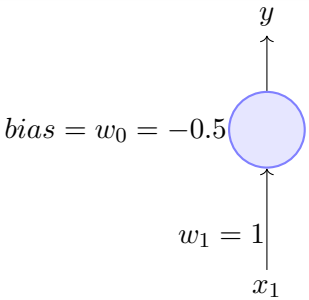
Module 3.1: Sigmoid Neuron

The story ahead ...

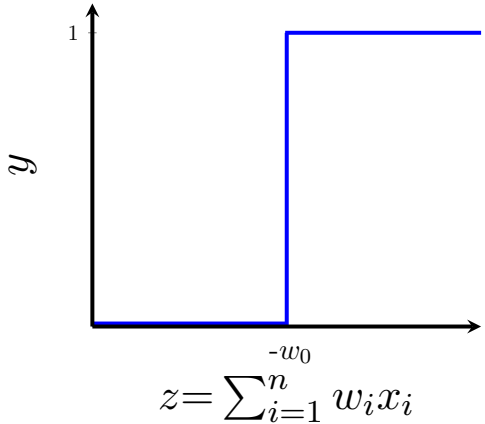
- Enough about boolean functions!
- What about arbitrary functions of the form $y = f(x)$ where $x \in \mathbb{R}^n$ (instead of $\{0,1\}^n$) and $y \in \mathbb{R}$ (instead of $\{0,1\}$) ?
- Can we have a network which can (approximately) represent such functions ?
- Before answering the above question we will have to first graduate from ***perceptrons*** to ***sigmoidal neurons*** ...

Recall

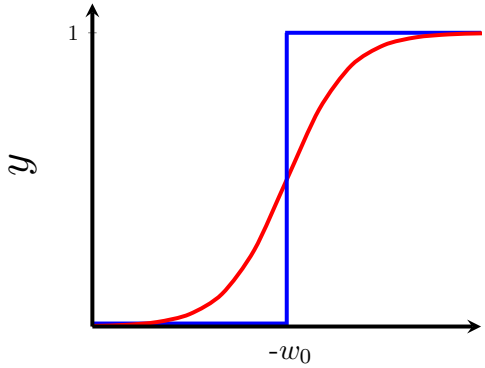
- A perceptron will fire if the weighted sum of its inputs is greater than the threshold ($-w_0$)



- The thresholding logic used by a perceptron is very harsh !
- For example, let us return to our problem of deciding whether we will like or dislike a movie
- Consider that we base our decision only on one input ($x_1 = criticsRating$ which lies between 0 and 1)
- If the threshold is 0.5 ($w_0 = -0.5$) and $w_1 = 1$ then what would be the decision for a movie with $criticsRating = 0.51$? (like)
- What about a movie with $criticsRating = 0.49$? (dislike)
- It seems harsh that we would like a movie with rating 0.51 but not one with a rating of 0.49



- This behavior is not a characteristic of the specific problem we chose or the specific weight and threshold that we chose
- It is a characteristic of the perceptron function itself which behaves like a step function
- There will always be this sudden change in the decision (from 0 to 1) when $\sum_{i=1}^{n} w_i x_i$ crosses the threshold (-w_0)
- For most real world applications we would expect a smoother decision function which gradually changes from 0 to 1

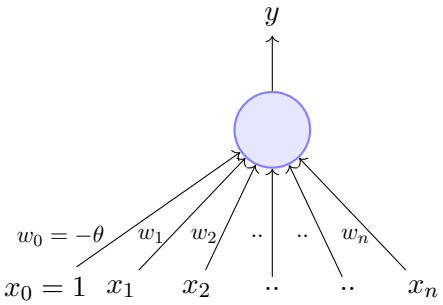


- Introducing sigmoid neurons where the output function is much smoother than the step function
- Here is one form of the sigmoid function called the logistic function

$$y = \frac{1}{1 + e^{-(w_0 + \sum_{i=1}^{n} w_i x_i)}}$$

- We no longer see a sharp transition around the threshold -w_0
- Also the output y is no longer binary but a real value between 0 and 1 which can be interpreted as a probability
- Instead of a like/dislike decision we get the probability of liking the movie

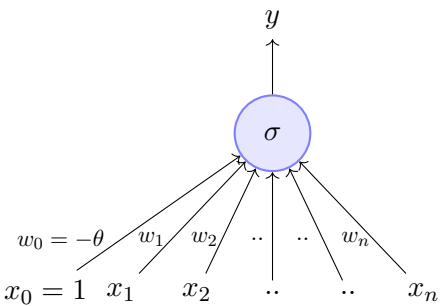
Perceptron



$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$

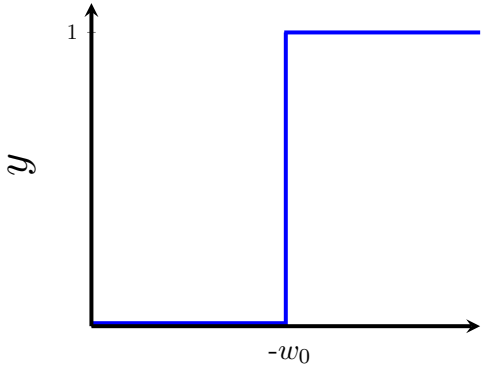
$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

Sigmoid (logistic) Neuron



$$y = \frac{1}{1 + e^{-(\sum_{i=0}^{n} w_i x_i)}}$$

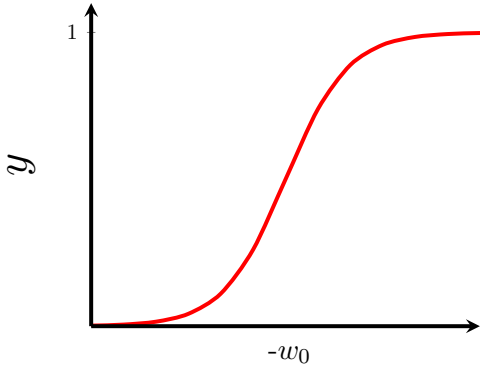
Perceptron



$$z = \sum_{i=1}^{n} w_i x_i$$

Not smooth, not continuous (at $w0$), **not differentiable**

Sigmoid Neuron

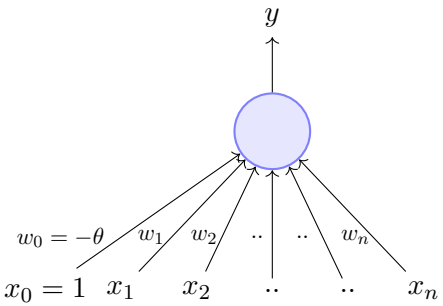


$$z = \sum_{i=1}^{n} w_i x_i$$

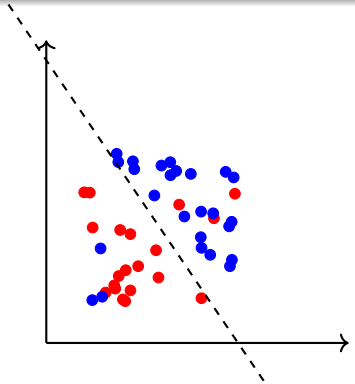
Smooth, continuous, **differentiable**

Module 3.2: A typical Supervised Machine Learning Setup

Sigmoid (logistic) Neuron



- What next ?
- Well, just as we had an algorithm for learning the weights of a perceptron, we also need a way of learning the weights of a sigmoid neuron
- Before we see such an algorithm we will revisit the concept of **error**



- Earlier we mentioned that a single perceptron cannot deal with this data because it is not linearly separable
- What does “cannot deal with” mean?
- What would happen if we use a perceptron model to classify this data ?
- We would probably end up with a line like this ...
- This line doesn’t seem to be too bad
- Sure, it misclassifies 3 blue points and 3 red points but we could live with this error in **most** real world applications
- From now on, we will accept that it is hard to drive the error to 0 in most cases and will instead aim to reach the minimum possible error

This brings us to a typical machine learning setup which has the following components...

- **Data:** $\{x_i, y_i\}_{i=1}^{n}$
- **Model:** Our approximation of the relation between $\mathbf{x}$ and y. For example,

$$\hat{y} = \frac{1}{1 + e^{-(\mathbf{w^T x})}}$$
$$or \quad \hat{y} = \mathbf{w^T x}$$
$$or \quad \hat{y} = \mathbf{x^T W x}$$

  or just about any function

- **Parameters:** In all the above cases, w is a parameter which needs to be learned from the data
- **Learning algorithm:** An algorithm for learning the parameters (w) of the model (for example, perceptron learning algorithm, gradient descent, etc.)
- **Objective/Loss/Error function:** To guide the learning algorithm - the learning algorithm should aim to minimize the loss function

As an illustration, consider our movie example

- **Data:** $\{x_i = movie, y_i = like/dislike\}_{i=1}^{n}$
- **Model:** Our approximation of the relation between $\mathbf{x}$ and y (the probability of liking a movie).

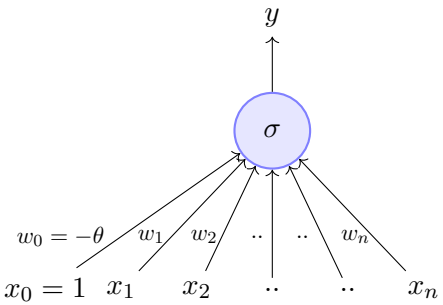
$$\hat{y} = \frac{1}{1 + e^{-(\mathbf{w^T x})}}$$

- **Parameter: w**
- **Learning algorithm:** Gradient Descent [we will see soon]
- **Objective/Loss/Error function:** One possibility is

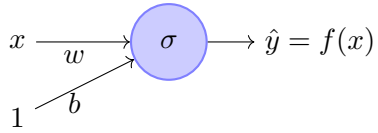
$$\mathscr{L}(\mathbf{w}) = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

The learning algorithm should aim to find a w which minimizes the above function (squared error between y and $\hat{y}$)

Module 3.3: Learning Parameters: (Infeasible) guess work

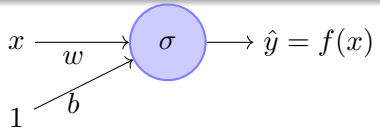


$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

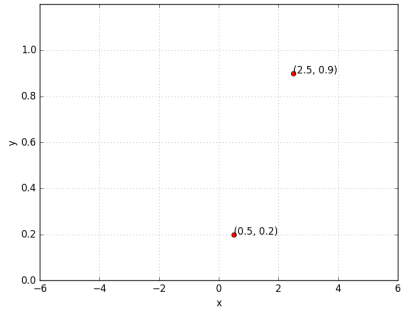


$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

- Keeping this supervised ML setup in mind, we will now focus on this **model** and discuss an **algorithm** for learning the **parameters** of this model from some given **data** using an appropriate **objective function**
- σ stands for the sigmoid function (logistic function in this case)
- For ease of explanation, we will consider a very simplified version of the model having just 1 input
- Further to be consistent with the literature, from now on, we will refer to w_0 as b (bias)
- Lastly, instead of considering the problem of predicting like/dislike, we will assume that we want to predict $criticsRating(y)$ given $imdbRating(x)$ (for no particular reason)



$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$



Input for training

$\{x_i, y_i\}_{i=1}^{N} \rightarrow N$ pairs of (x, y)

Training objective

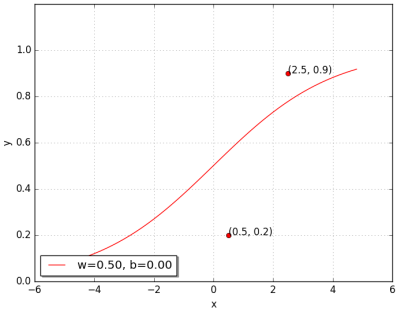
Find w and b such that:

$$\underset{w,b}{\text{minimize}}\, \mathscr{L}(w, b) = \sum_{i=1}^{N} (y_i - f(x_i))^2$$

What does it mean to train the network?

- Suppose we train the network with $(x, y) = (0.5, 0.2)$ and $(2.5, 0.9)$
- At the end of training we expect to find w*, b* such that:
- $f(0.5) \rightarrow 0.2$ and $f(2.5) \rightarrow 0.9$

Let us see this in more detail....



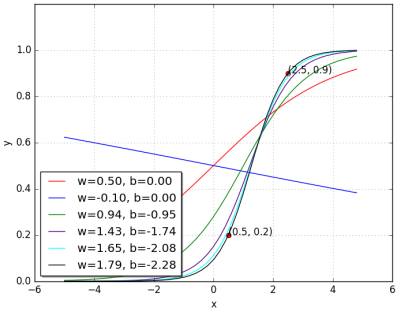
$$\sigma(x) = \frac{1}{1 + e^{-(wx+b)}}$$

- Can we try to find such a w^*, b^* manually
- Let us try a random guess.. (say, $w = 0.5, b = 0$)
- Clearly not good, but how bad is it ?
- Let us revisit $\mathscr{L}(w, b)$ to see how bad it is ...

$$\begin{aligned}
\mathscr{L}(w, b) &= \frac{1}{2} * \sum_{i=1}^{N} (y_i - f(x_i))^2 \\
&= \frac{1}{2} * (y_1 - f(x_1))^2 + (y_2 - f(x_2))^2 \\
&= \frac{1}{2} * (0.9 - f(2.5))^2 + (0.2 - f(0.5))^2 \\
&= 0.073
\end{aligned}$$

We want $\mathscr{L}(w, b)$ to be as close to 0 as possible

Let us try some other values of w, b

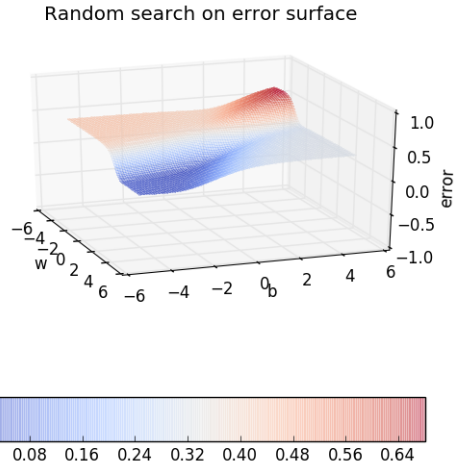


w	b	$\mathscr{L}(w,b)$
0.50	0.00	0.0730
-0.10	0.00	0.1481
0.94	-0.94	0.0214
1.42	-1.73	0.0028
1.65	-2.08	0.0003
1.78	-2.27	0.0000

$$\sigma(x) = \frac{1}{1 + e^{-(wx+b)}}$$

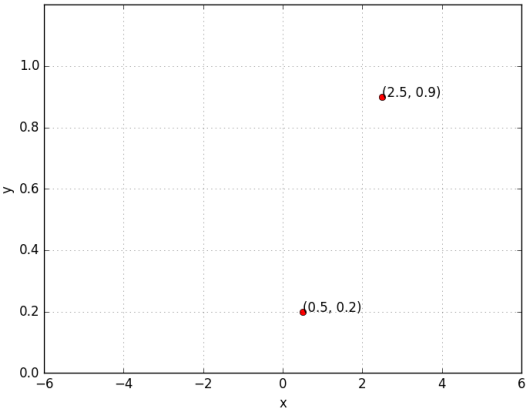
- Oops!! this made things even worse...
- Perhaps it would help to push w and b in the other direction...
- Let us keep going in this direction, *i.e.*, increase w and decrease b
- With some guess work and intuition we were able to find the right values for w and b

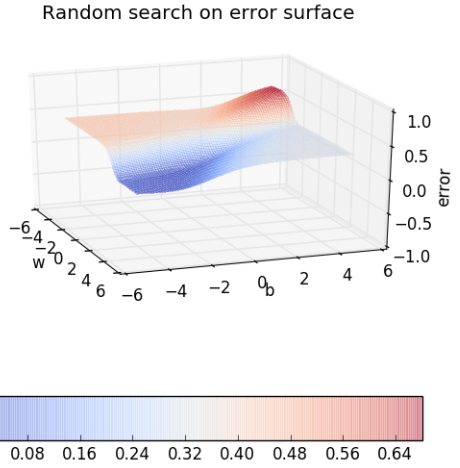
Let us look at something better than our "guess work" algorithm....

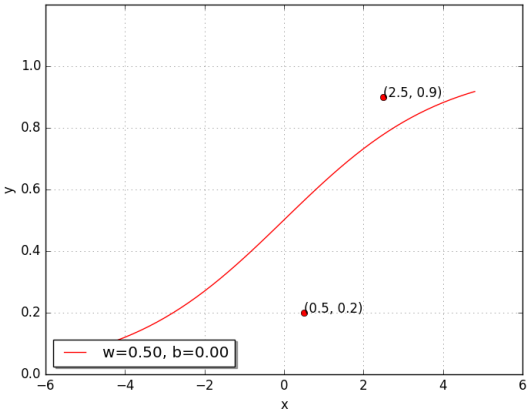


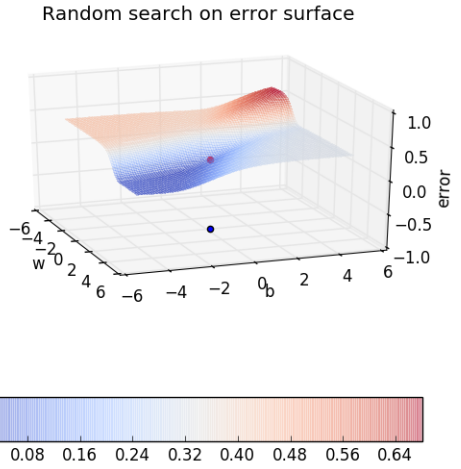
- Since we have only 2 points and 2 parameters (w, b) we can easily plot $\mathscr{L}(w, b)$ for different values of (w, b) and pick the one where $\mathscr{L}(w, b)$ is minimum
- But of course this becomes intractable once you have many more data points and many more parameters !!
- Further, even here we have plotted the error surface only for a small range of (w, b) [from $(-6, 6)$ and not from $(-\inf, \inf)$]

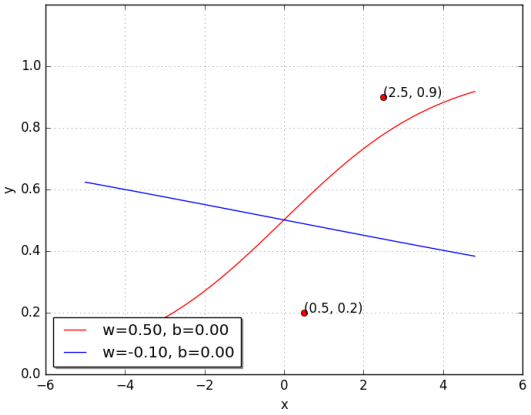
Let us look at the geometric interpretation of our "guess work" algorithm in terms of this error surface

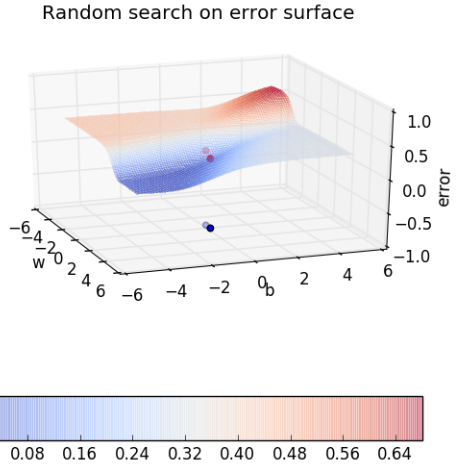

(2.5, 0.9)
(0.5, 0.2)
y
x

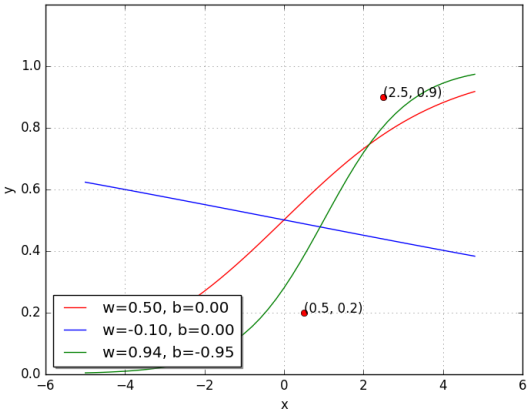

Random search on error surface
error
w
b
0.08 0.16 0.24 0.32 0.40 0.48 0.56 0.64

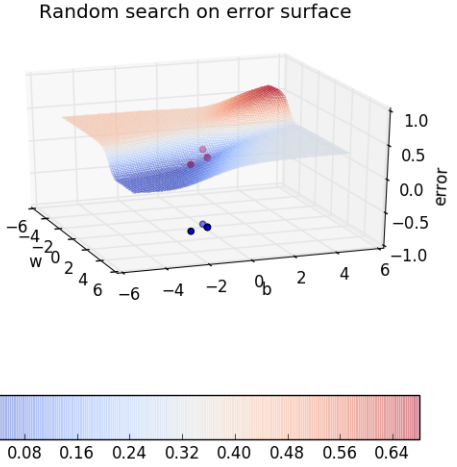

(2.5, 0.9)
(0.5, 0.2)
w=0.50, b=0.00
y
x

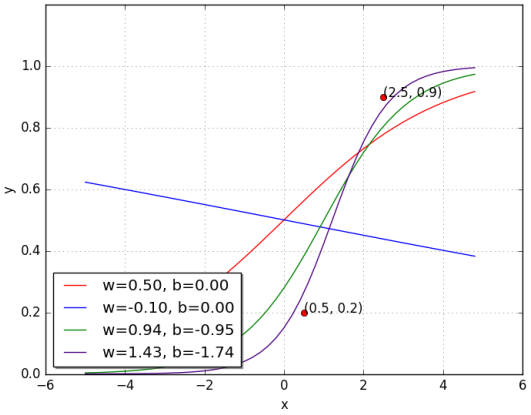

Random search on error surface
error
w
b
0.08 0.16 0.24 0.32 0.40 0.48 0.56 0.64

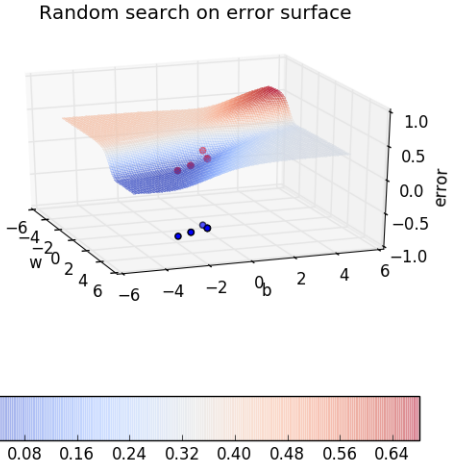

(2.5, 0.9)
(0.5, 0.2)
w=0.50, b=0.00
w=-0.10, b=0.00
x
y

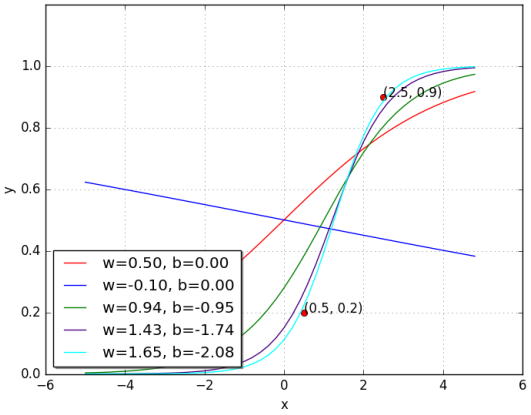

Random search on error surface
w
b
error
0.08 0.16 0.24 0.32 0.40 0.48 0.56 0.64

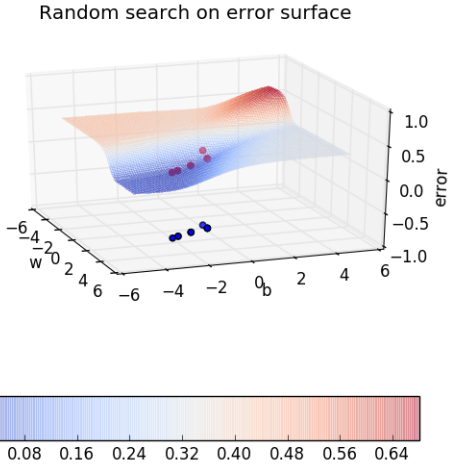

1.0
0.8
0.6
0.4
0.2
0.0
y
−6
−4
−2
0
2
4
6
x
(2.5, 0.9)
(0.5, 0.2)
w=0.50, b=0.00
w=-0.10, b=0.00
w=0.94, b=-0.95

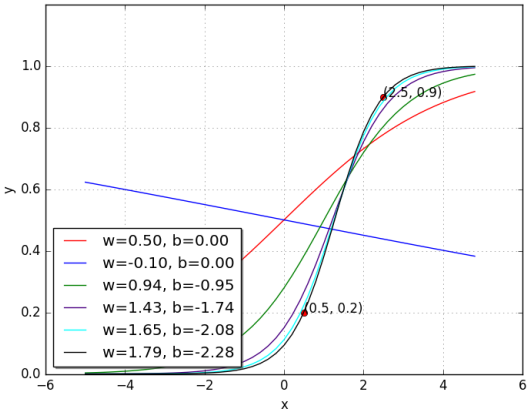

Random search on error surface
error
1.0
0.5
0.0
−0.5
−1.0
w
b
0.08
0.16
0.24
0.32
0.40
0.48
0.56
0.64

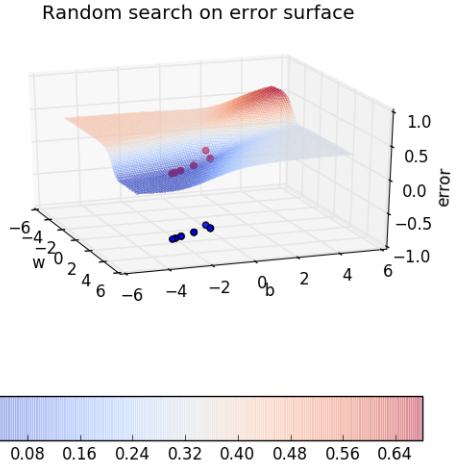

1.0
0.8
0.6
0.4
0.2
0.0
y
−6
−4
−2
0
2
4
6
x
(2.5, 0.9)
(0.5, 0.2)
w=0.50, b=0.00
w=-0.10, b=0.00
w=0.94, b=-0.95
w=1.43, b=-1.74


Random search on error surface
error
1.0
0.5
0.0
−0.5
−1.0
w
b
0.08
0.16
0.24
0.32
0.40
0.48
0.56
0.64


1.0
0.8
0.6
0.4
0.2
0.0
y
−6
−4
−2
0
2
4
6
x
(2.5, 0.9)
(0.5, 0.2)
w=0.50, b=0.00
w=-0.10, b=0.00
w=0.94, b=-0.95
w=1.43, b=-1.74
w=1.65, b=-2.08


Random search on error surface
w
b
error
1.0
0.5
0.0
−0.5
−1.0
0.08
0.16
0.24
0.32
0.40
0.48
0.56
0.64


1.0
0.8
0.6
0.4
0.2
0.0
y
−6
−4
−2
0
2
4
6
x
(2.5, 0.9)
(0.5, 0.2)
w=0.50, b=0.00
w=-0.10, b=0.00
w=0.94, b=-0.95
w=1.43, b=-1.74
w=1.65, b=-2.08
w=1.79, b=-2.28


Random search on error surface
error
1.0
0.5
0.0
−0.5
−1.0
w
b
0.08
0.16
0.24
0.32
0.40
0.48
0.56
0.64

Module 3.4: Learning Parameters : Gradient Descent

Now let us see if there is a more efficient and principled way of doing this

Goal

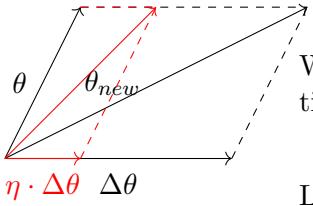
Find a better way of traversing the error surface so that we can reach the minimum value quickly without resorting to brute force search!

vector of parameters, say, randomly initialized

$\theta = [w, b]$

$\Delta\theta = [\Delta w, \Delta b]$

change in the values of w, b



We moved in the direction of $\Delta\theta$

Let us be a bit conservative: move only by a small amount η

$$\theta_{new} = \theta + \eta \cdot \Delta\theta$$

Question: What is the right $\Delta\theta$ to use ?

The answer comes from Taylor series

For ease of notation, let $\Delta\theta = u$, then from Taylor series, we have,

$$\begin{aligned}\mathscr{L}(\theta + \eta u) &= \mathscr{L}(\theta) + \eta * u^T \nabla_\theta \mathscr{L}(\theta) + \frac{\eta^2}{2!} * u^T \nabla^2 \mathscr{L}(\theta) u + \frac{\eta^3}{3!} * ... + \frac{\eta^4}{4!} * ... \\ &= \mathscr{L}(\theta) + \eta * u^T \nabla_\theta \mathscr{L}(\theta) \ [\eta \textit{ is typically small, so } \eta^2, \eta^3, .. \to 0]\end{aligned}$$

Note that the move (ηu) would be favorable only if,

$$\mathscr{L}(\theta + \eta u) - \mathscr{L}(\theta) < 0 \ [\textit{i.e., if the new loss is less than the previous loss}]$$

This implies,

$$u^T \nabla_\theta \mathscr{L}(\theta) < 0$$

Okay, so we have,

$$u^T \nabla_\theta \mathscr{L}(\theta) < 0$$

But, what is the range of $u^T \nabla_\theta \mathscr{L}(\theta)$? Let us see....
Let β be the angle between u and $\nabla_\theta \mathscr{L}(\theta)$, then we know that,

$$-1 \leq cos(\beta) = \frac{u^T \nabla_\theta \mathscr{L}(\theta)}{||u|| * ||\nabla_\theta \mathscr{L}(\theta)||} \leq 1$$

multiply throughout by $k = ||u|| * ||\nabla_\theta \mathscr{L}(\theta)||$

$$-k \leq k * cos(\beta) = u^T \nabla_\theta \mathscr{L}(\theta) \leq k$$

Thus, $\mathscr{L}(\theta + \eta u) - \mathscr{L}(\theta) = u^T \nabla_\theta \mathscr{L}(\theta) = k * cos(\beta)$ will be most negative when $cos(\beta) = -1$ *i.e.*, when β is 180°

Gradient Descent Rule

- The direction u that we intend to move in should be at 180° w.r.t. the gradient
- In other words, move in a direction opposite to the gradient

Parameter Update Equations

$$w_{t+1} = w_t - \eta \nabla w_t$$
$$b_{t+1} = b_t - \eta \nabla b_t$$
$$where, \nabla w_t = \frac{\partial \mathscr{L}(w,b)}{\partial w}_{at\ w=w_t, b=b_t}, \nabla b = \frac{\partial \mathscr{L}(w,b)}{\partial b}_{at\ w=w_t, b=b_t}$$

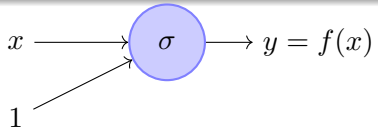
So we now have a more principled way of moving in the w-b plane than our "guess work" algorithm

- Let us create an algorithm from this rule ...

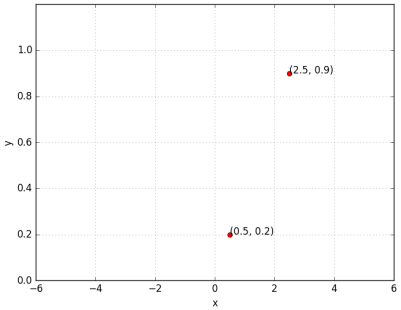
Algorithm: gradient_descent()

$t \leftarrow 0$;
$max\_iterations \leftarrow 1000$;
while $t < max\_iterations$ **do**
    $w_{t+1} \leftarrow w_t - \eta \nabla w_t$;
    $b_{t+1} \leftarrow b_t - \eta \nabla b_t$;
    $t \leftarrow t + 1$;
end

- To see this algorithm in practice let us first derive ∇w and ∇b for our toy neural network



$$f(x) = \frac{1}{1+e^{-(w \cdot x+b)}}$$



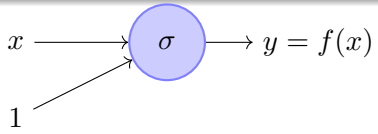
Let's assume there is only 1 point to fit (x, y)

$$\mathscr{L}(w, b) = \frac{1}{2} * (f(x) - y)^2$$

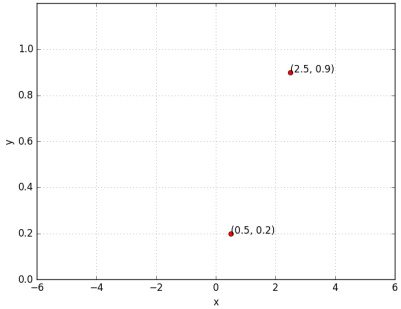
$$\nabla w = \frac{\partial \mathscr{L}(w, b)}{\partial w} = \frac{\partial}{\partial w}[\frac{1}{2} * (f(x) - y)^2]$$

$$
\begin{aligned}
\nabla w &= \frac{\partial}{\partial w}[\frac{1}{2} * (f(x) - y)^2] \\
&= \frac{1}{2} * [2 * (f(x) - y) * \frac{\partial}{\partial w}(f(x) - y)] \\
&= (f(x) - y) * \frac{\partial}{\partial w}(f(x)) \\
&= (f(x) - y) * \frac{\partial}{\partial w}\Big(\frac{1}{1 + e^{-(wx+b)}}\Big) \\
&= (f(x) - y) * f(x) * (1 - f(x)) * x
\end{aligned}
$$

$$
\begin{aligned}
&\frac{\partial}{\partial w}\Big(\frac{1}{1 + e^{-(wx+b)}}\Big) \\
&= \frac{-1}{(1 + e^{-(wx+b)})^2}\frac{\partial}{\partial w}(e^{-(wx+b)})) \\
&= \frac{-1}{(1 + e^{-(wx+b)})^2} * (e^{-(wx+b)})\frac{\partial}{\partial w}(-(wx + b))) \\
&= \frac{-1}{(1 + e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1 + e^{-(wx+b)})} * (-x) \\
&= \frac{1}{(1 + e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1 + e^{-(wx+b)})} * (x) \\
&= f(x) * (1 - f(x)) * x
\end{aligned}
$$



$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$



So if there is only 1 point (x, y), we have,

$$\nabla w = (f(x) - y) * f(x) * (1 - f(x)) * x$$

For two points,

$$\nabla w = \sum_{i=1}^{2} (f(x_i) - y_i) * f(x_i) * (1 - f(x_i)) * x_i$$

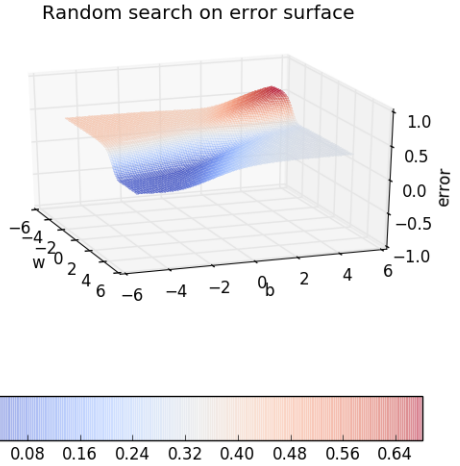
$$\nabla b = \sum_{i=1}^{2} (f(x_i) - y_i) * f(x_i) * (1 - f(x_i))$$

```
X = [0.5, 2.5]
Y = [0.2, 0.9]
```

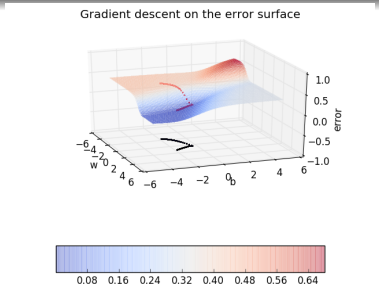
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w,b,x) : #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x + b)))
```

```
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w,b,x) : #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x + b)))

def error (w, b) :
    err = 0.0
    for x,y in zip(X,Y) :
        fx = f(w,b,x)
        err += 0.5 * (fx - y) ** 2
    return err
```

```
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w,b,x) : #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x + b)))

def error (w, b) :
    err = 0.0
    for x,y in zip(X,Y) :
        fx = f(w,b,x)
        err += 0.5 * (fx - y) ** 2
    return err
```



```
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w,b,x) : #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x + b)))

def error (w, b) :
    err = 0.0
    for x,y in zip(X,Y) :
        fx = f(w,b,x)
        err += 0.5 * (fx - y) ** 2
    return err

def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent() :
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```



- Later on in the course we will look at gradient descent in much more detail and discuss its variants
- For the time being it suffices to know that we have an algorithm for learning the parameters of a sigmoid neuron
- So where do we head from here ?

Module 3.5: Representation Power of a Multilayer Network of Sigmoid Neurons

Representation power of a multilayer network of perceptrons

A multilayer network of perceptrons with a single hidden layer can be used to represent any boolean function precisely (no errors)

Representation power of a multilayer network of sigmoid neurons

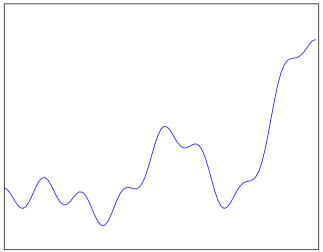
A multilayer network of neurons with a single hidden layer can be used to approximate any continuous function to any desired precision

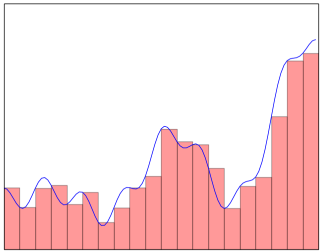
In other words, there is a guarantee that for any function $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$, we can always find a neural network (with 1 hidden layer containing enough neurons) whose output g(x) satisfies $|g(x) - f(x)| < \epsilon$!!

Proof: We will see an illustrative proof of this... [Cybenko, 1989], [Hornik, 1991]

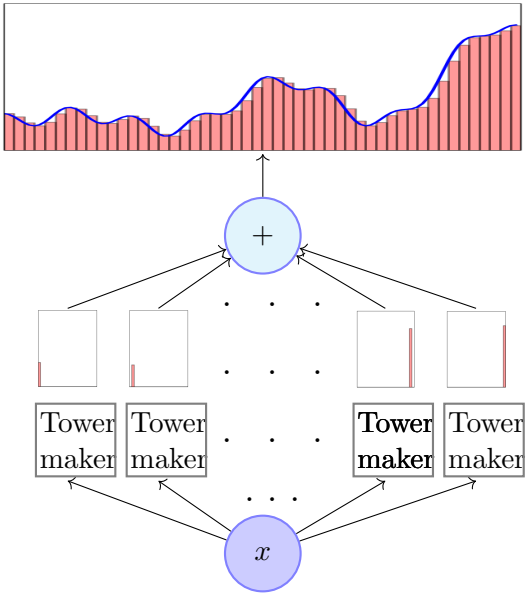
- See this link* for an excellent illustration of this proof
- The discussion in the next few slides is based on the ideas presented at the above link

*http://neuralnetworksanddeeplearning.com/chap4.html



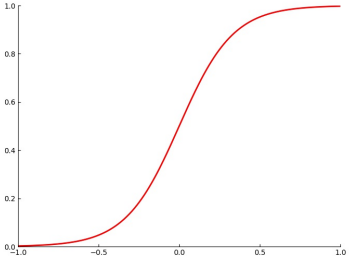


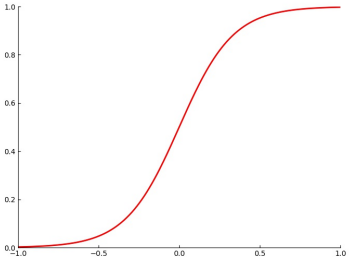
- We are interested in knowing whether a network of neurons can be used to represent an arbitrary function (like the one shown in the figure)
- We observe that such an arbitrary function can be approximated by several "tower" functions
- More the number of such "tower" functions, better the approximation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



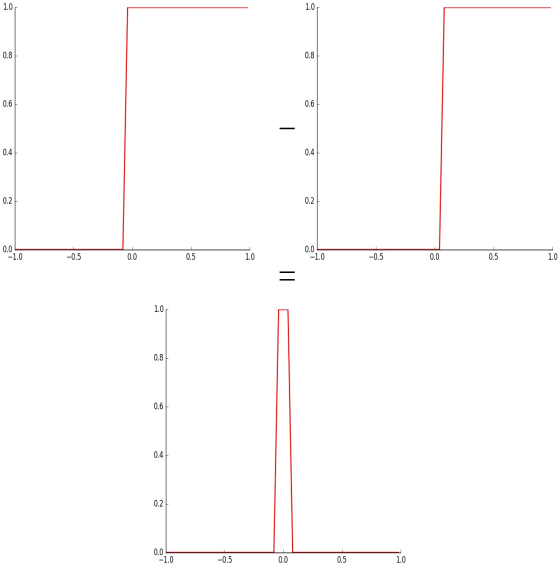
- We make a few observations
- All these "tower" functions are similar and only differ in their heights and positions on the x-axis
- Suppose there is a black box which takes the original input (x) and constructs these tower functions
- We can then have a simple network which can just add them up to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

We will figure this out over the next few slides ...



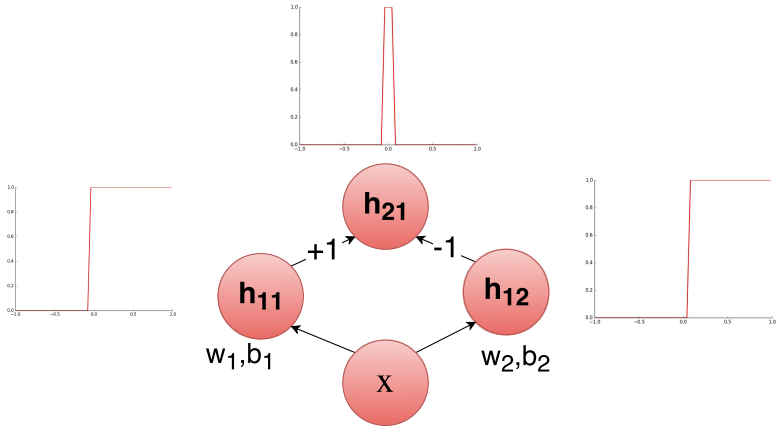


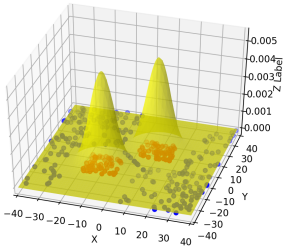
- If we take the logistic function and set w to a very high value we will recover the step function
- Let us see what happens as we change the value of w
- Further we can adjust the value of b to control the position on the x-axis at which the function transitions from 0 to 1



- Now let us see what we get by taking two such sigmoid functions (with different b's) and subtracting one from the other
- Voila! We have our tower function !!

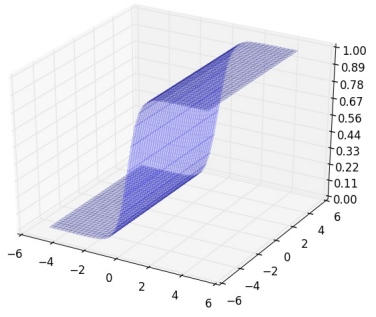
- Can we come up with a neural network to represent this operation of subtracting one sigmoid function from another ?


h21
+1
-1
h11
h12
w1,b1
w2,b2
x



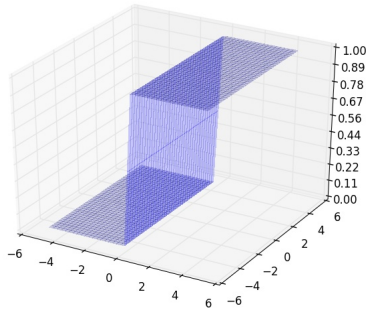
- What if we have more than one input?
- Suppose we are trying to take a decision about whether we will find oil at a particular location on the ocean bed(Yes/No)
- Further, suppose we base our decision on two factors: Salinity (x_1) and Pressure (x_2)
- We are given some data and it seems that $y(\text{oil}|\text{no-oil})$ is a complex function of x_1 and x_2
- We want a neural network to approximate this function

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

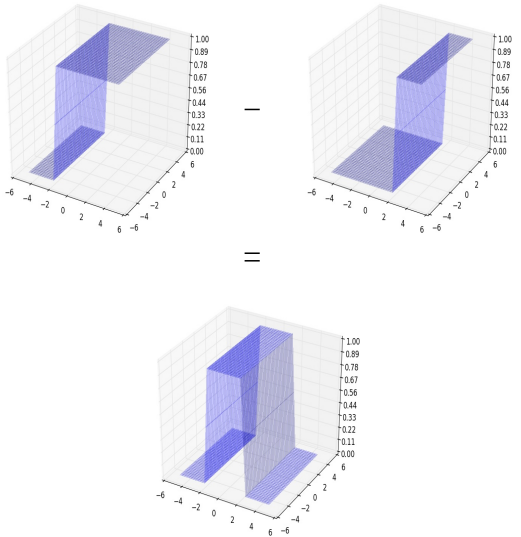


- This is what a 2-dimensional sigmoid looks like
- We need to figure out how to get a tower in this case
- First, let us set w_2 to 0 and see if we can get a two dimensional step function
- What would happen if we change b ?

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



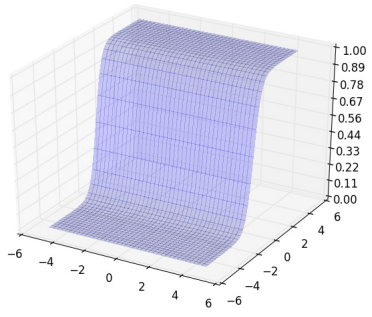
- This is what a 2-dimensional sigmoid looks like
- We need to figure out how to get a tower in this case
- First, let us set w_2 to 0 and see if we can get a two dimensional step function
- What would happen if we change b ?



- What if we take two such step functions (with different b values) and subtract one from the other
- We still don't get a tower (or we get a tower which is open from two sides)

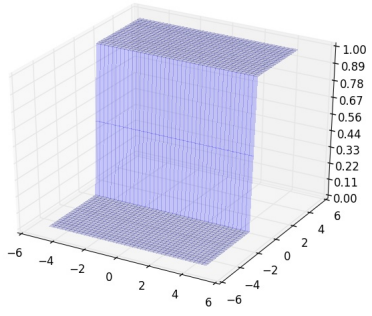
$$y = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + b)}}$$

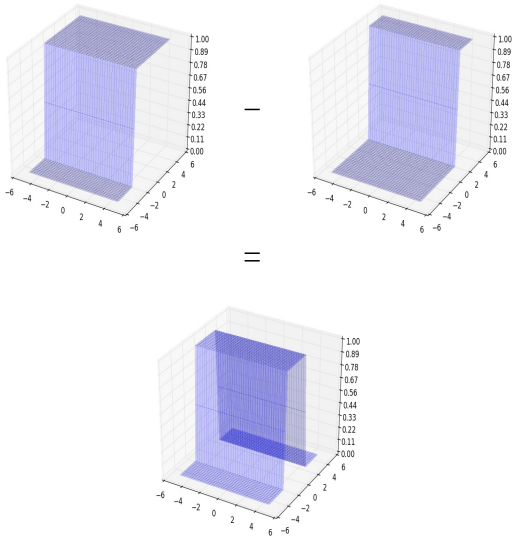
- Now let us set w_1 to 0 and adjust w_2 to get a 2-dimensional step function with a different orientation
- And now we change b



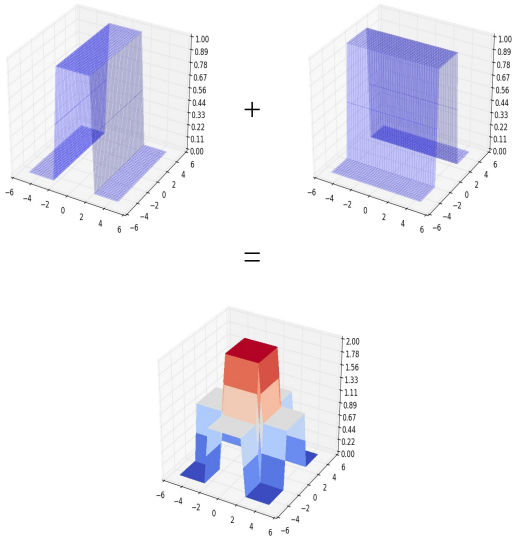
$$y = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + b)}}$$

- Now let us set w_1 to 0 and adjust w_2 to get a 2-dimensional step function with a different orientation
- And now we change b

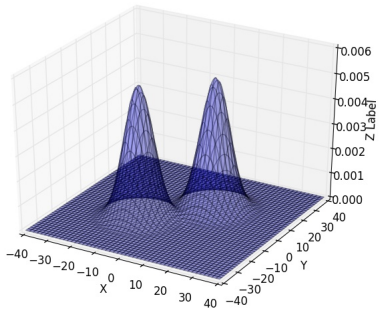




- Again, what if we take two such step functions (with different b values) and subtract one from the other
- We still don't get a tower (or we get a tower which is open from two sides)
- Notice that this open tower has a different orientation from the previous one

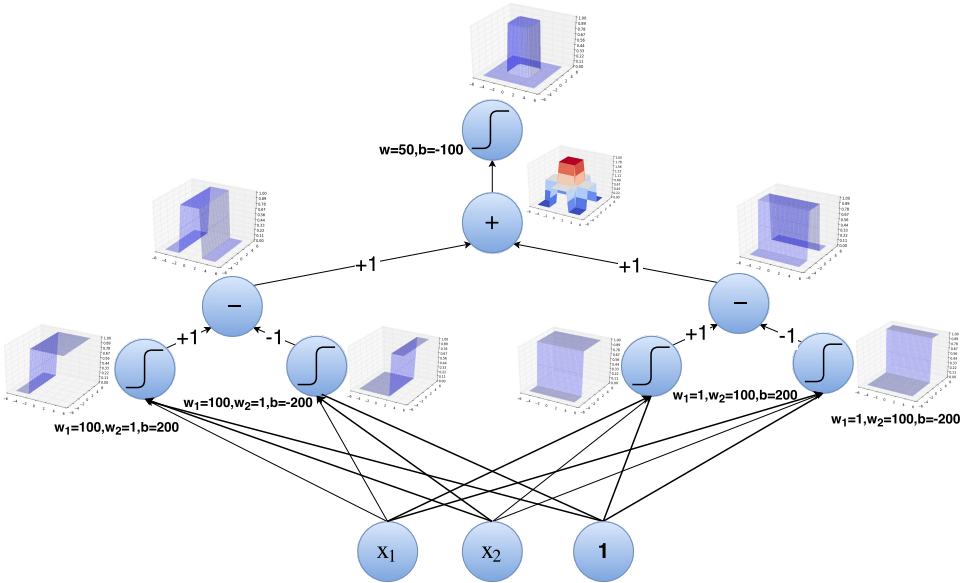


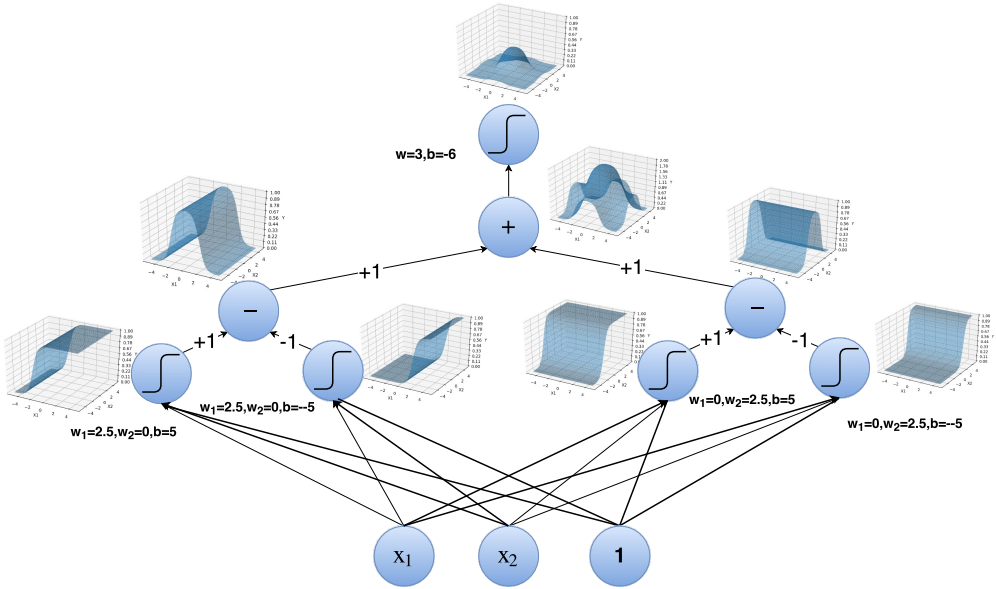
- Now what will we get by adding two such open towers ?
- We get a tower standing on an elevated base
- We can now pass this output through another sigmoid neuron to get the desired tower !
- We can now approximate any function by summing up many such towers



- For example, we could approximate the following function using a sum of several towers

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


w=50,b=-100
+
+1
+1
−
−
+1
-1
+1
-1
w1=100,w2=1,b=200
w1=100,w2=1,b=-200
w1=1,w2=100,b=200
w1=1,w2=100,b=-200
x1
x2
1

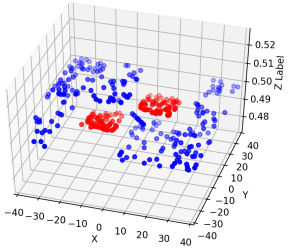

w=3,b=-6
+1
+1
+1
-1
+1
-1
w1=2.5,w2=0,b=5
w1=2.5,w2=0,b=-5
w1=0,w2=2.5,b=5
w1=0,w2=2.5,b=-5
x1
x2
1

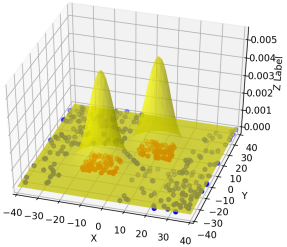
Think

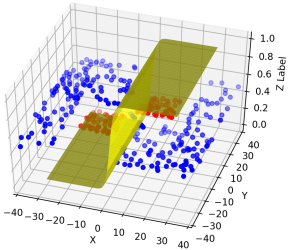
- For 1 dimensional input we needed 2 neurons to construct a tower
- For 2 dimensional input we needed 4 neurons to construct a tower
- How many neurons will you need to construct a tower in n dimensions ?

Time to retrospect

- Why do we care about approximating any arbitrary function ?
- Can we tie all this back to the classification problem that we have been dealing with ?







- This is what we actually want
- 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 the blue points from the red points !!