Lecture 3

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

January 13, 2017

Mitesh M. Khapra

Department of Computer Science and Engineering
Indian Institute of Technology Madras

The story ahead ...

- Enough about boolean functions!

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 $\{0,1\}$) ?

The story ahead ...

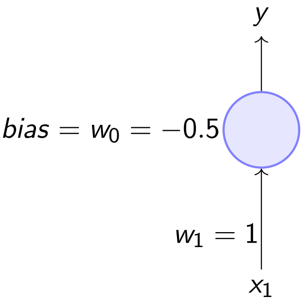
- Enough about boolean functions!
- What about arbitrary functions of the form $y = f(x)$ where $x \in \mathbb{R}^n$ (instead of $\{0,1\}^n$) and $y \in \mathbb{R}$ (instead of $\{0,1\}$) ?
- Can we have a network which can (approximately) represent such functions ?

The story ahead ...

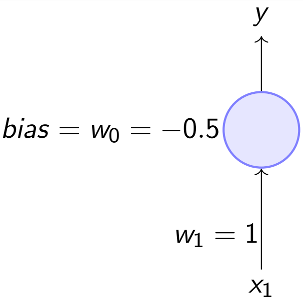
- Enough about boolean functions!
- What about arbitrary functions of the form $y = f(x)$ where $x \in \mathbb{R}^n$ (instead of $\{0,1\}^n$) and $y \in \mathbb{R}$ (instead of $\{0,1\}$) ?
- Can we have a network which can (approximately) represent such functions ?
- Before answering the above question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

Recall

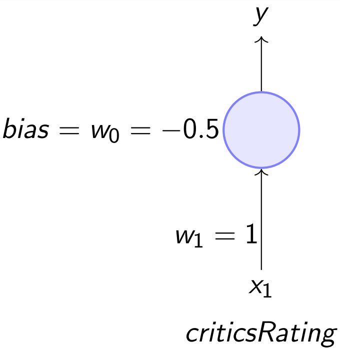
- A perceptron will fire if the weighted sum of its inputs is greater than the threshold (-w_0)



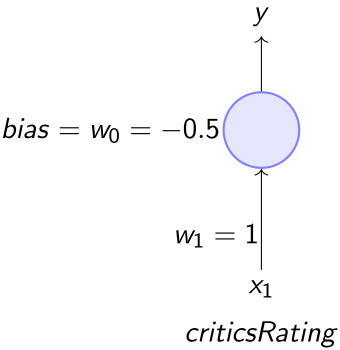
- The thresholding logic used by a perceptron is very harsh !



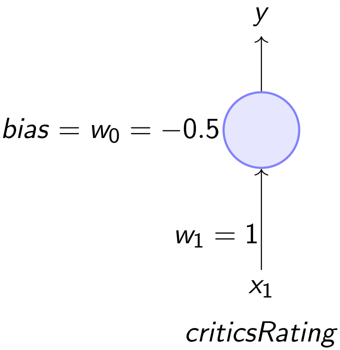
- The thresholding logic used by a perceptron is very harsh !
- For example, let us return to our problem of deciding whether we will like or dislike a movie



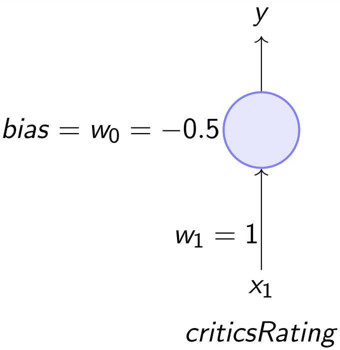
- The thresholding logic used by a perceptron is very harsh !
- For example, let us return to our problem of deciding whether we will like or dislike a movie
- Consider that we base our decision only on one input ($x_1 = criticsRating$ which lies between 0 and 1)



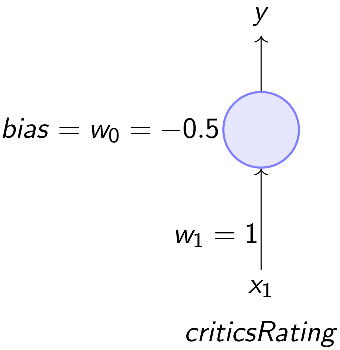
- The thresholding logic used by a perceptron is very harsh !
- For example, let us return to our problem of deciding whether we will like or dislike a movie
- Consider that we base our decision only on one input ($x_1 = criticsRating$ which lies between 0 and 1)
- If the threshold is 0.5 ($w_0 = -0.5$) and $w_1 = 1$ then what would be the decision for a movie with $criticsRating = 0.51$?



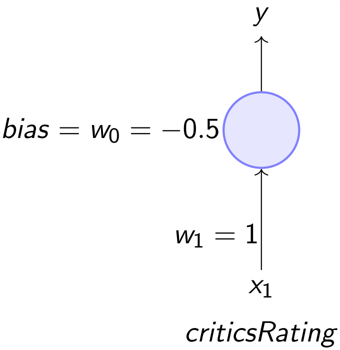
- The thresholding logic used by a perceptron is very harsh !
- For example, let us return to our problem of deciding whether we will like or dislike a movie
- Consider that we base our decision only on one input ($x_1 = criticsRating$ which lies between 0 and 1)
- If the threshold is 0.5 ($w_0 = -0.5$) and $w_1 = 1$ then what would be the decision for a movie with $criticsRating = 0.51$? (like)



- 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 $criticsRating = 0.49$?

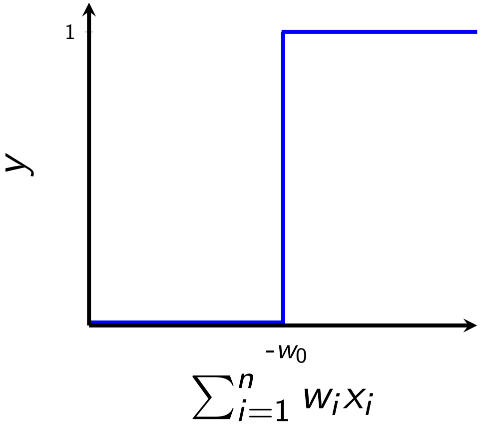


- The thresholding logic used by a perceptron is very harsh !
- For example, let us return to our problem of deciding whether we will like or dislike a movie
- Consider that we base our decision only on one input ($x_1 = criticsRating$ which lies between 0 and 1)
- If the threshold is 0.5 ($w_0 = -0.5$) 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)

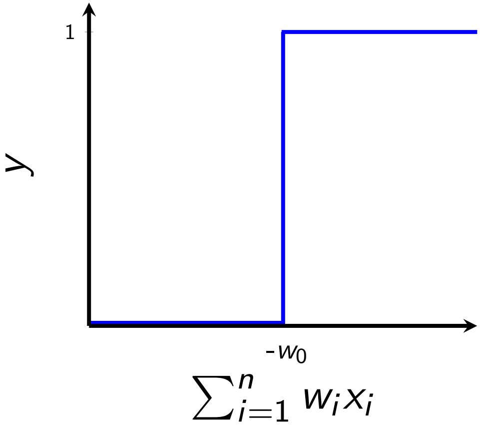


- The thresholding logic used by a perceptron is very harsh !
- For example, let us return to our problem of deciding whether we will like or dislike a movie
- Consider that we base our decision only on one input ($x_1 = criticsRating$ which lies between 0 and 1)
- If the threshold is 0.5 ($w_0 = -0.5$) 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

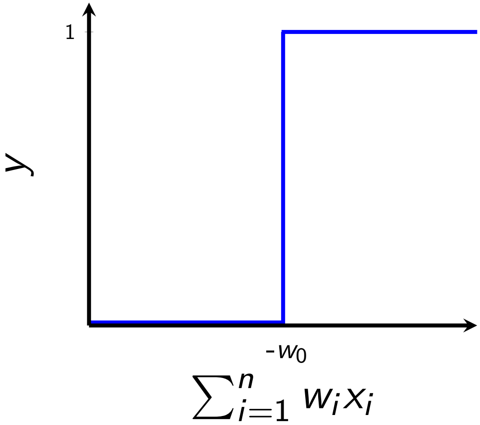
- This behavior is not a characteristic of the specific problem we chose or the specific weight and threshold that we chose



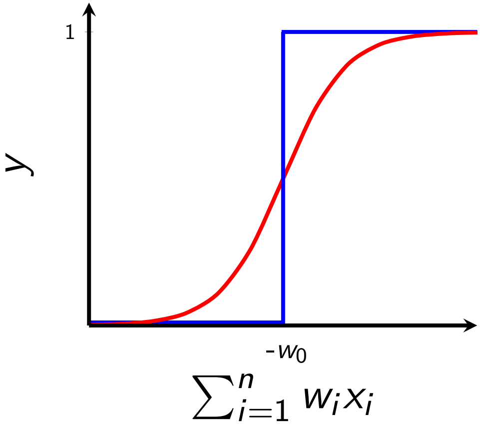
- This behavior is not a characteristic of the specific problem we chose or the specific weight and threshold that we chose
- It is a characteristic of the perceptron function itself which behaves like a step function



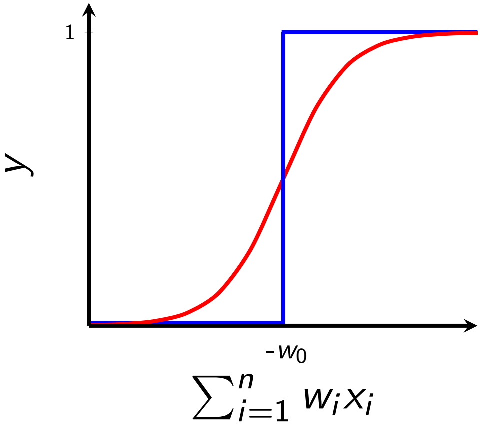
- This behavior is not a characteristic of the specific problem we chose or the specific weight and threshold that we chose
- It is a characteristic of the perceptron function itself which behaves like a step function
- There will always be this sudden change in the decision (from 0 to 1) when $\sum_{i=1}^{n}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



- 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

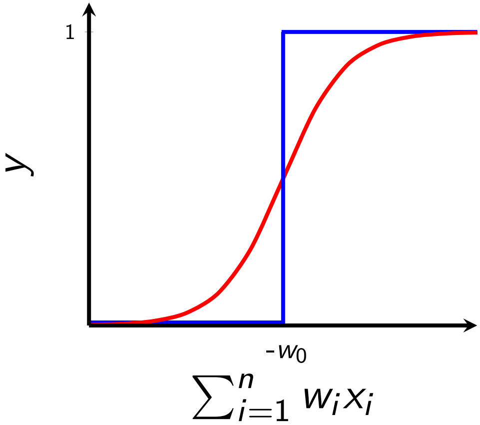


- Introducing sigmoid neurons where the output function is much smoother than the step function



- 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

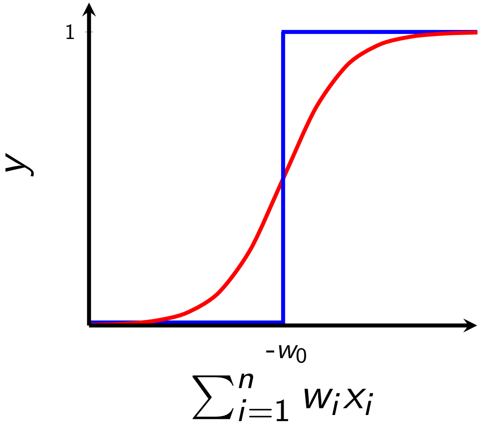
$$y = \frac{1}{1 + e^{-(w_0 + \sum_{i=1}^{n} w_i x_i)}}$$



- 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

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

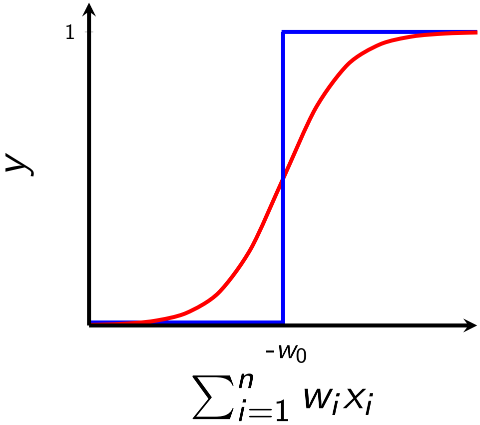
- We no longer see a sharp transition around the threshold $-w_0$



- 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

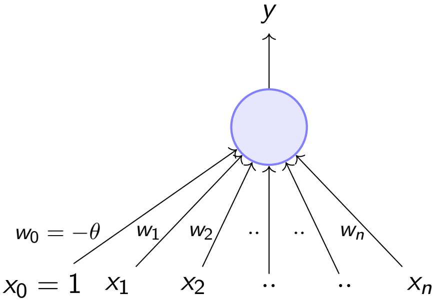


- Introducing sigmoid neurons where the output function is much smoother than the step function
- Here is one form of the sigmoid function called the logistic function

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

- 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

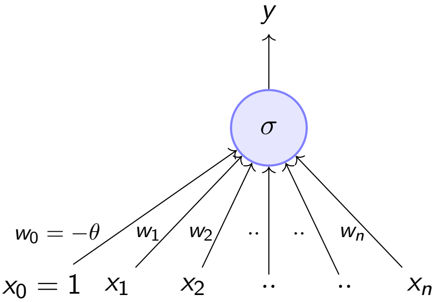
Perceptron



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

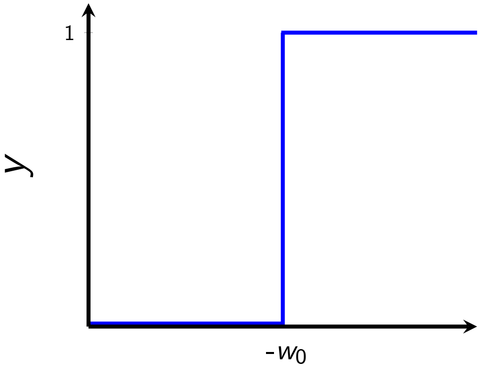
$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

Sigmoid (logistic) Neuron



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

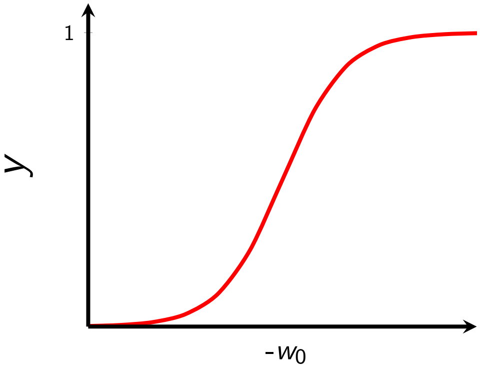
Perceptron



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

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

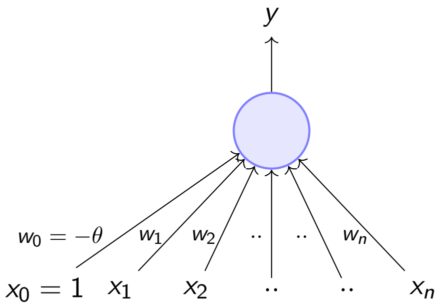
Sigmoid Neuron



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

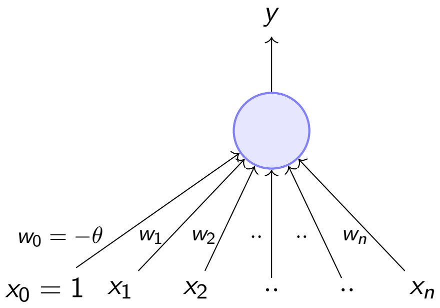
Smooth, continuous, **differentiable**

Sigmoid (logistic) Neuron



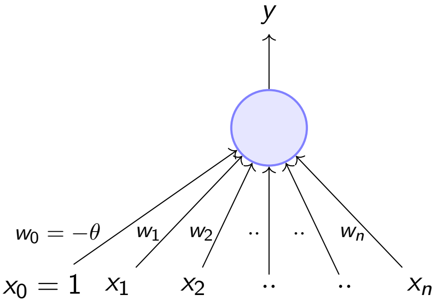
- What next ?

Sigmoid (logistic) Neuron

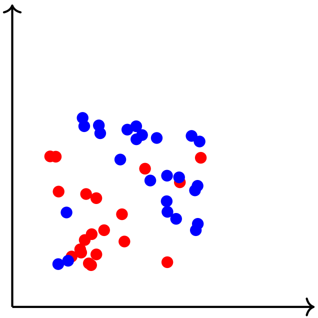


- What next ?
- Well, just as we had an algorithm for learning the weights of a perceptron, we also need a way of learning the weights of a sigmoid neuron

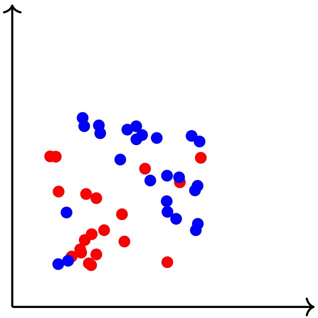
Sigmoid (logistic) Neuron



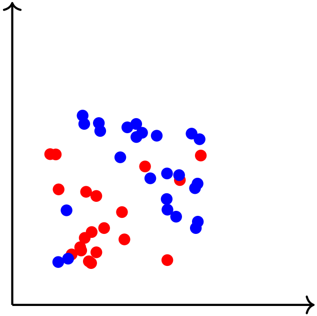
- What next ?
- Well, just as we had an algorithm for learning the weights of a perceptron, we also need a way of learning the weights of a sigmoid neuron
- Before we see such an algorithm we will revisit the concept of **error**



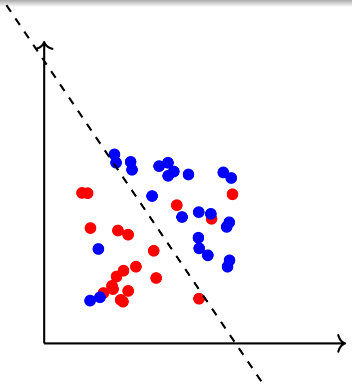
- Earlier we mentioned that a single perceptron cannot deal with this data because it is not linearly separable



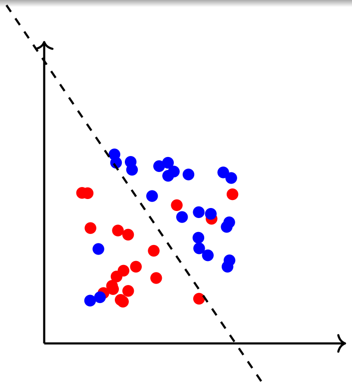
- Earlier we mentioned that a single perceptron cannot deal with this data because it is not linearly separable
- What does "cannot deal with" mean?



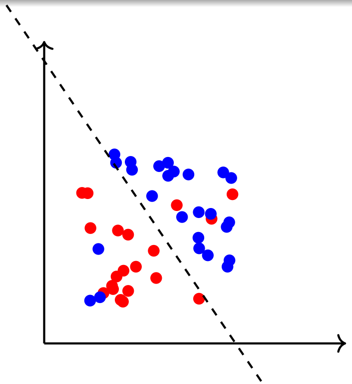
- Earlier we mentioned that a single perceptron cannot deal with this data because it is not linearly separable
- What does "cannot deal with" mean?
- What would happen if we use a perceptron model to classify this data ?



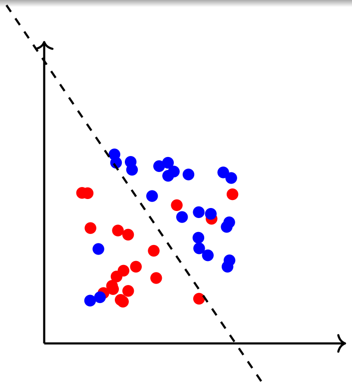
- Earlier we mentioned that a single perceptron cannot deal with this data because it is not linearly separable
- What does “cannot deal with” mean?
- What would happen if we use a perceptron model to classify this data ?
- We would probably end up with a line like this ...



- 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



- 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



- 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...

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

- **Data:** $\{x_i, y_i\}_{i=1}^{n}$

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 x and y. For example,

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 x and y. For example,

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

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 x and y. For example,

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

$$or \quad \hat{y} = w^T x$$

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 x and y. For example,

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

$$or \quad \hat{y} = w^T x$$

$$or \quad \hat{y} = (w^T x)^2$$

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 x and y. For example,

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

$$or \quad \hat{y} = w^T x$$

$$or \quad \hat{y} = (w^T x)^2$$

or just about any function

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 x and y. For example,

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

$$or \quad \hat{y} = w^T x$$

$$or \quad \hat{y} = (w^T x)^2$$

  or just about any function

- **Parameters:** In all the above cases, w is a parameter which needs to be learned from the data

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 x and y. For example,

$$\hat{y} = \frac{1}{1 + e^{-(w^T x)}}$$
$$or \quad \hat{y} = w^T x$$
$$or \quad \hat{y} = (w^T x)^2$$

  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.)

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 x and y. For example,

$$\hat{y} = \frac{1}{1 + e^{-(w^T x)}}$$
$$\text{or} \quad \hat{y} = w^T x$$
$$\text{or} \quad \hat{y} = (w^T x)^2$$

  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

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 x and y. For example,

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

$$or \quad \hat{y} = w^T x$$

$$or \quad \hat{y} = (w^T x)^2$$

  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

As an illustration, consider our movie example

- **Data:** $\{x_i = movie, y_i = like/dislike\}_{i=1}^{n}$

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 x and y (the probability of liking a movie).

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 x and y (the probability of liking a movie).

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

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 x and y (the probability of liking a movie).

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

- **Parameter:** w

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 x and y (the probability of liking a movie).

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

- **Parameter:** w
- **Learning algorithm:** Gradient Descent [we will see soon]

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 x and y (the probability of liking a movie).

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

- **Parameter:** w
- **Learning algorithm:** Gradient Descent [we will see soon]
- **Objective/Loss/Error 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 x and y (the probability of liking a movie).

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

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

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

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 x and y (the probability of liking a movie).

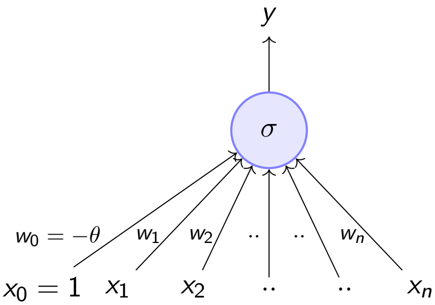
$$\hat{y} = \frac{1}{1 + e^{-(w^T x)}}$$

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

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

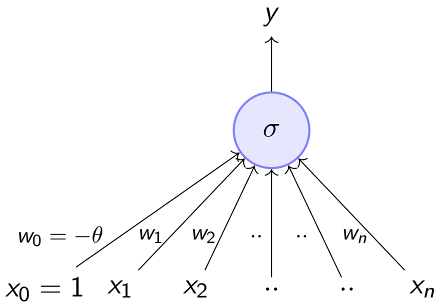
The learning algorithm should aim to find a w which minimizes the above function (squared error between y and $\hat{y}$)

Sigmoid (logistic) Neuron

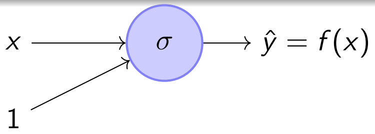


- With this setup in mind, we will now focus on this **model** and discuss an **algorithm** for learning the **parameters** of this model from some given **data**

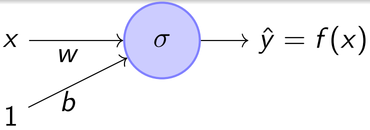
Sigmoid (logistic) Neuron



- With this 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**
- σ stands for the sigmoid function (logistic function in this case)

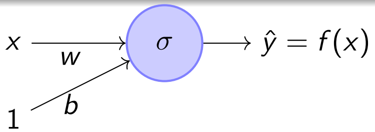


- With this setup in mind, we will now focus on this **model** and discuss an **algorithm** for learning the **parameters** of this model from some given **data**
- σ stands for the sigmoid function (logistic function in this case)
- For ease of explanation, we will consider a very simplified version of the model having just 1 input



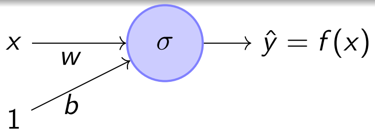
$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

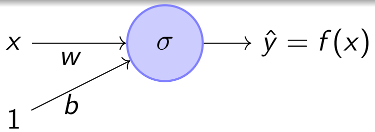
- With this setup in mind, we will now focus on this **model** and discuss an **algorithm** for learning the **parameters** of this model from some given **data**
- σ stands for the sigmoid function (logistic function in this case)
- For ease of explanation, we will consider a very simplified version 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)



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

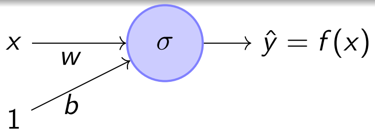
- With this setup in mind, we will now focus on this **model** and discuss an **algorithm** for learning the **parameters** of this model from some given **data**
- σ stands for the sigmoid function (logistic function in this case)
- For ease of explanation, we will consider a very simplified version 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)


x
w
σ
$\hat{y} = f(x)$
1
b



Input for training

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



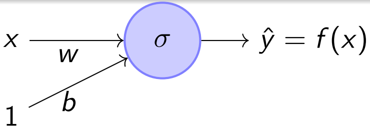
Input for training

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

Training objective

Find w and b such that:

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



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

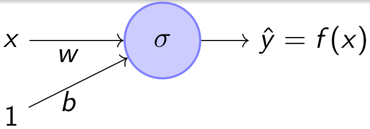
Input for training

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

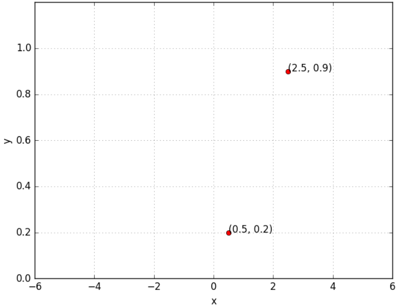
Training objective

Find w and b such that:

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



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

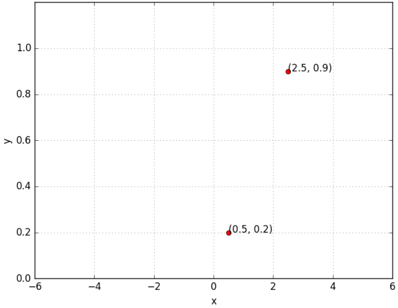


What does it mean to train the network?

- Suppose we train the network with $(x, y) = (0.5, 0.2)$ and $(2.5, 0.9)$

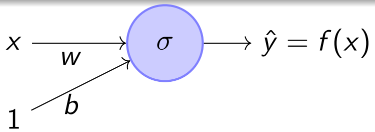
$$x \xrightarrow{w} \sigma \longrightarrow \hat{y} = f(x)$$
$$1 \xrightarrow{b} \sigma$$

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

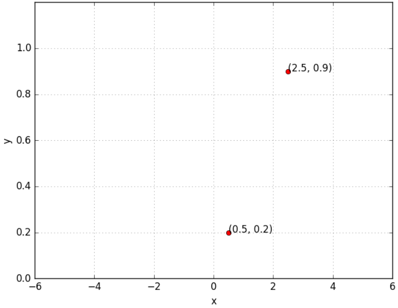


What does it mean to train the network?

- 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:

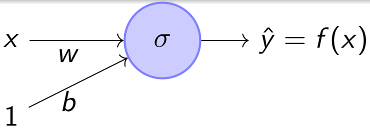


$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

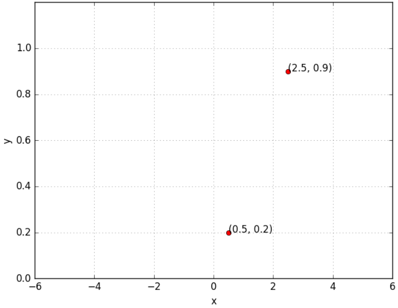


What does it mean to train the network?

- 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 $f(2.5) \rightarrow 0.9$



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

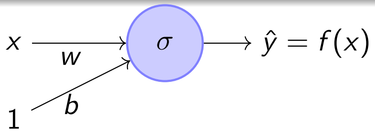


What does it mean to train the network?

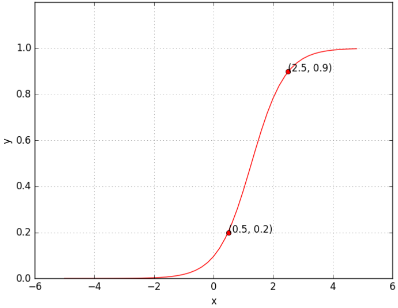
- Suppose we train the network with $(x, y) = (0.5, 0.2)$ and $(2.5, 0.9)$
- At the end of training we expect to find w*, b* such that:
- $f(0.5) \rightarrow 0.2$ and $f(2.5) \rightarrow 0.9$

In other words...

- We hope to find a sigmoid function such that $(0.5, 0.2)$ and $(2.5, 0.9)$ lie on this sigmoid



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



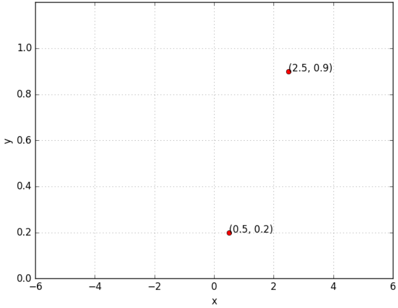
What does it mean to train the network?

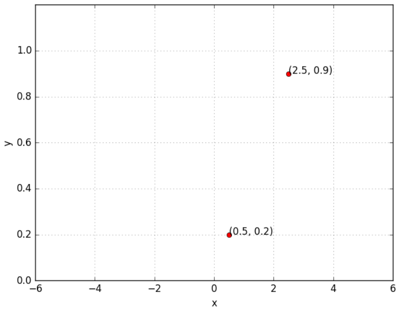
- Suppose we train the network with $(x, y) = (0.5, 0.2)$ and $(2.5, 0.9)$
- At the end of training we expect to find w*, b* such that:
- $f(0.5) \rightarrow 0.2$ and $f(2.5) \rightarrow 0.9$

In other words...

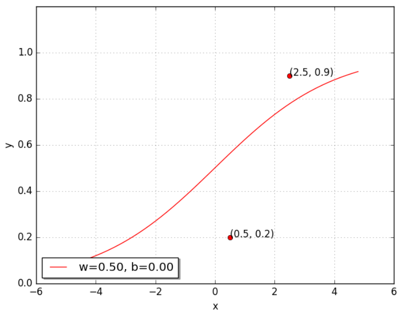
- We hope to find a sigmoid function such that $(0.5, 0.2)$ and $(2.5, 0.9)$ lie on this sigmoid

Lets see this in more detail....

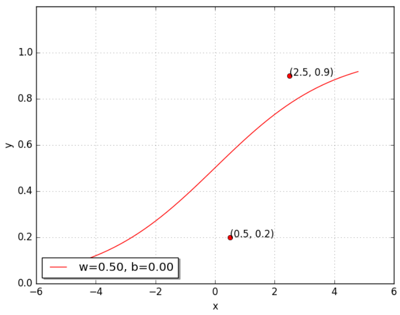

1.0
0.8
0.6
0.4
0.2
0.0
y
(2.5, 0.9)
(0.5, 0.2)
−6
−4
−2
0
2
4
6
x



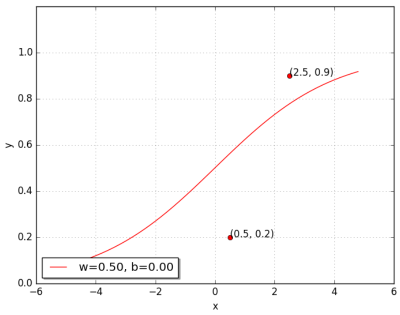
- Can we try to find such a $w*, b*$ manually



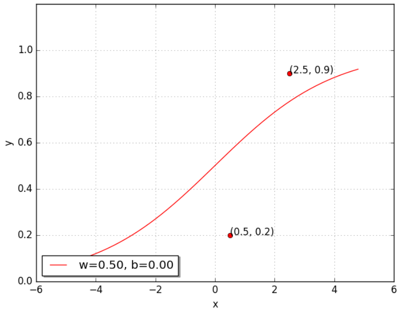
- Can we try to find such a $w*, b*$ manually
- Lets try a random guess.. (say, $w = 0.5, b = 0$)



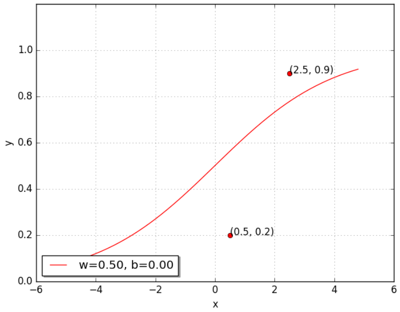
- Can we try to find such a $w*, b*$ manually
- Lets try a random guess.. (say, $w = 0.5, b = 0$)
- Clearly not good, but how bad is it ?



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

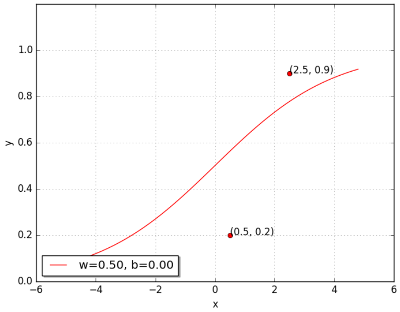


$$\mathscr{L}(w, b) = \frac{1}{2} * \sum_{i=1}^{N} (y_i - f(x_i))^2$$

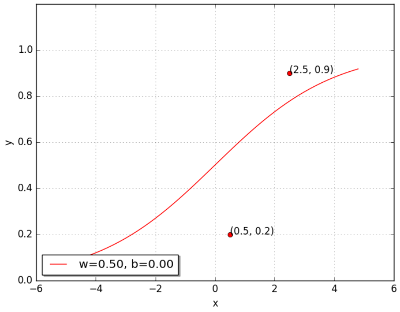


$$\mathscr{L}(w,b) = \frac{1}{2} * \sum_{i=1}^{N} (y_i - f(x_i))^2$$

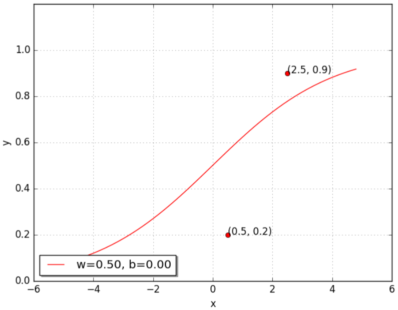
$$= \frac{1}{2} * (y_1 - f(x_1))^2 + (y_2 - f(x_2))^2$$



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



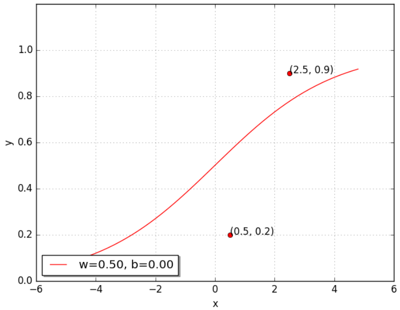
$$
\begin{aligned}
\mathscr{L}(w,b) &= \frac{1}{2} * \sum_{i=1}^{N}(y_i - f(x_i))^2 \\
&= \frac{1}{2} * (y_1 - f(x_1))^2 + (y_2 - f(x_2))^2 \\
&= \frac{1}{2} * (0.9 - f(2.5))^2 + (0.2 - f(0.5))^2 \\
&= 0.073
\end{aligned}
$$



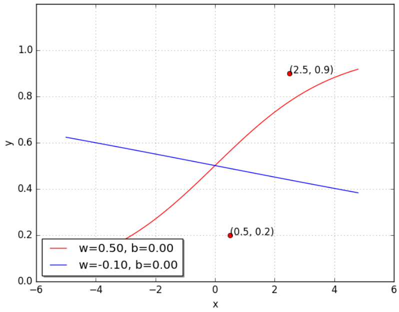
$$
\begin{aligned}
\mathscr{L}(w, b) &= \frac{1}{2} * \sum_{i=1}^{N} (y_i - f(x_i))^2 \\
&= \frac{1}{2} * (y_1 - f(x_1))^2 + (y_2 - f(x_2))^2 \\
&= \frac{1}{2} * (0.9 - f(2.5))^2 + (0.2 - f(0.5))^2 \\
&= 0.073
\end{aligned}
$$

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

Lets try some other values of w, b



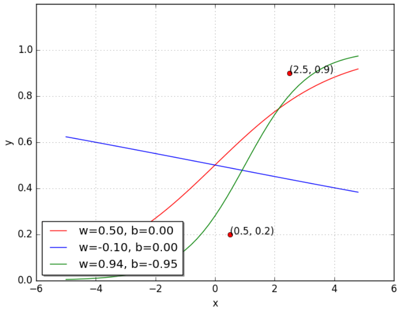
w	b	$\mathscr{L}(w,b)$
0.50	0.00	0.0730



Lets 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

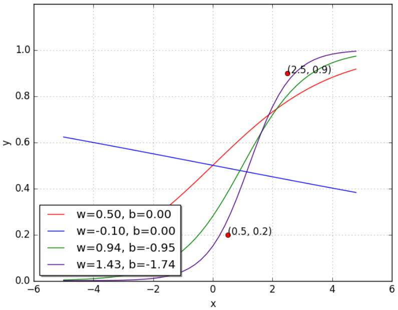
Oops!! this made things even worse...



Lets 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

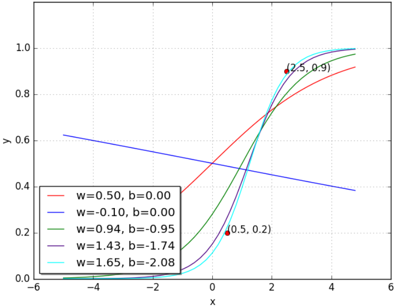
Perhaps it would help to push w and b in the other direction...



Lets 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

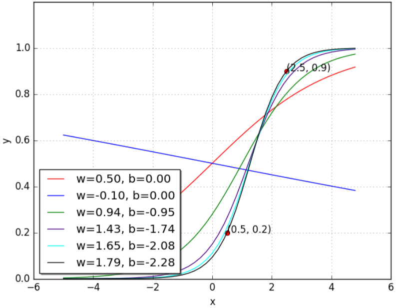
Lets keep going in this direction, *i.e.*, increase w and decrease b



Lets 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

Lets keep going in this direction, *i.e.*, increase w and decrease b



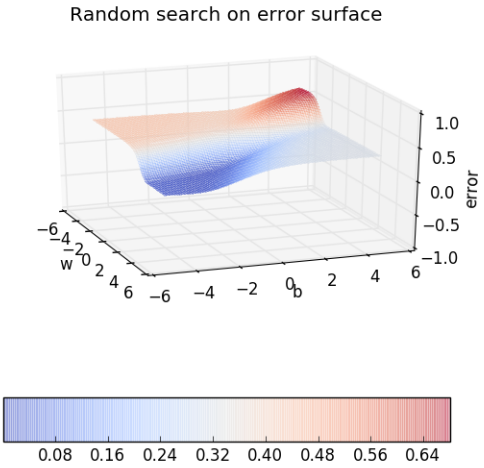
Lets try some other values of w, b

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

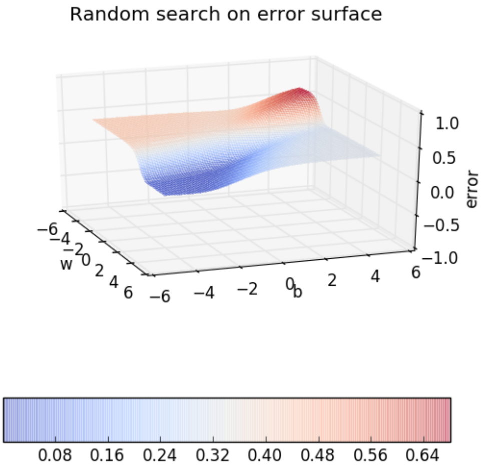
With some guess work and intuition we were able to find the right values for w and b

Lets look at something better than our "guess work" algorithm....

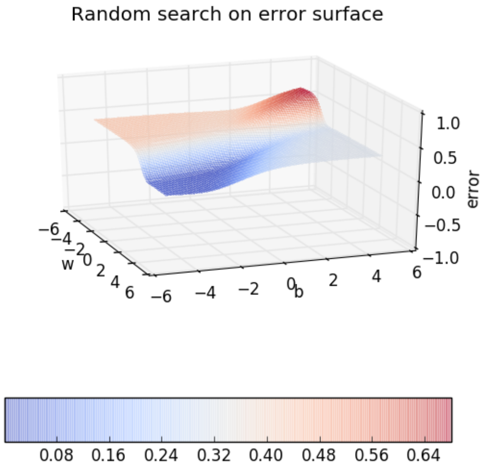
- Since we have only 2 points and 2 parameters $(w,\ b)$ we can easily plot $\mathscr{L}(w, b)$ for different values of $(w,\ b)$ and pick the one where $\mathscr{L}(w, b)$ is minimum



- 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

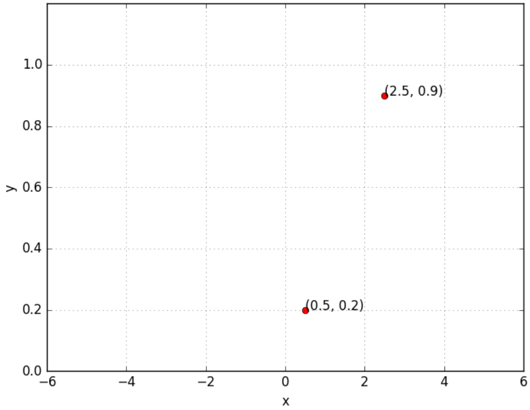


- Since we have only 2 points and 2 parameters $(w,\ b)$ we can easily plot $\mathscr{L}(w, b)$ for different values of $(w,\ b)$ and pick the one where $\mathscr{L}(w, b)$ is minimum
- But of course this becomes intractable once you have many more data points and many more parameters !!

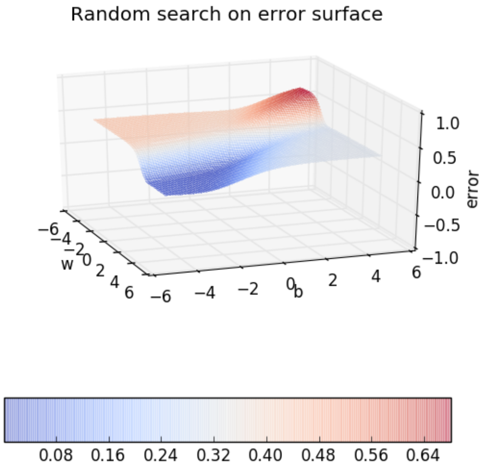


- Since we have only 2 points and 2 parameters $(w,\ b)$ we can easily plot $\mathscr{L}(w,b)$ for different values of $(w,\ b)$ and pick the one where $\mathscr{L}(w,b)$ is minimum
- But of course this becomes intractable once you have many more data points and many more parameters !!
- Further, even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

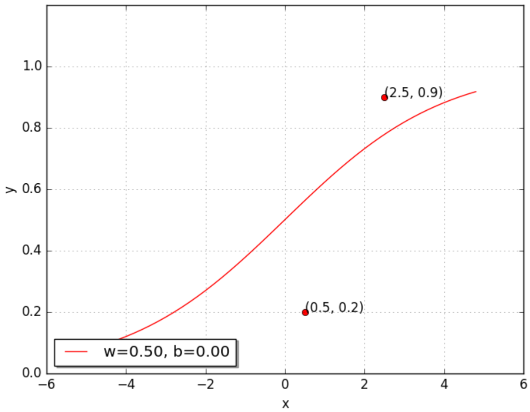
Lets look at the geometric interpretation of our "guess work" algorithm in terms of this error surface

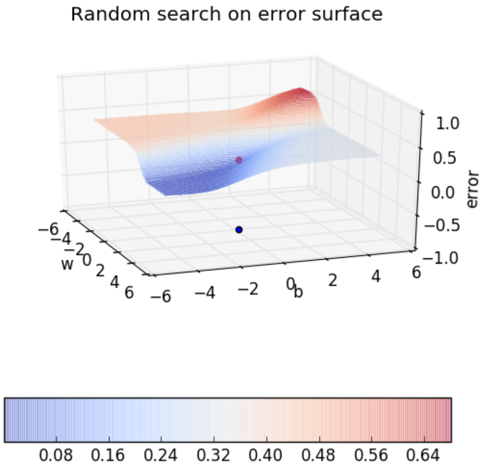


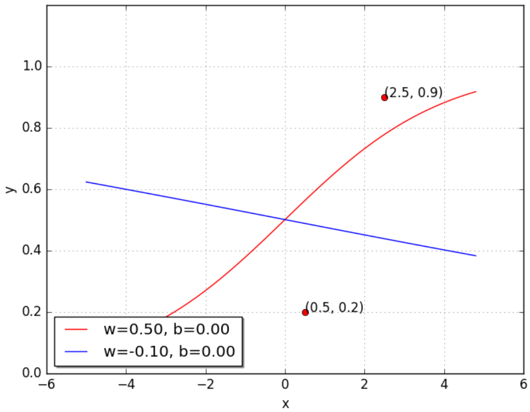
1.0
0.8
0.6
0.4
0.2
0.0
y
−6
−4
−2
0
2
4
6
x
(2.5, 0.9)
(0.5, 0.2)

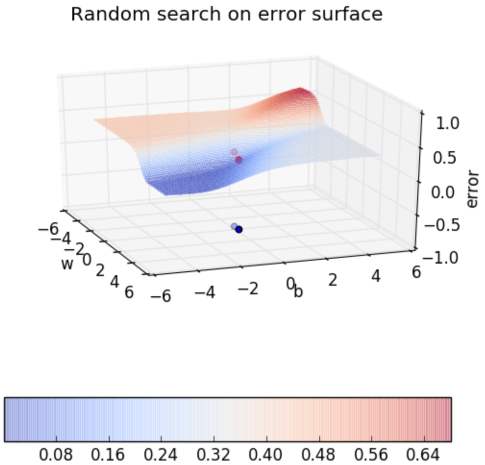


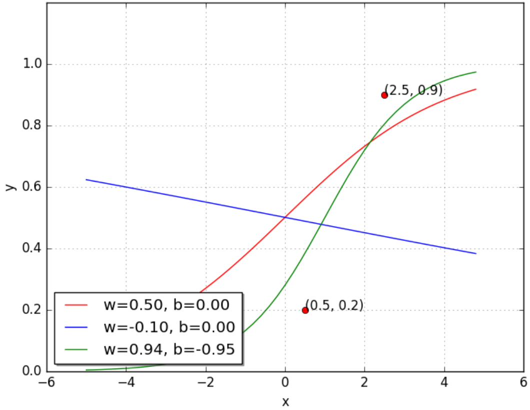
Random search on error surface
error
1.0
0.5
0.0
−0.5
−1.0
w
b
−6
−4
−2
0
2
4
6
0.08
0.16
0.24
0.32
0.40
0.48
0.56
0.64


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

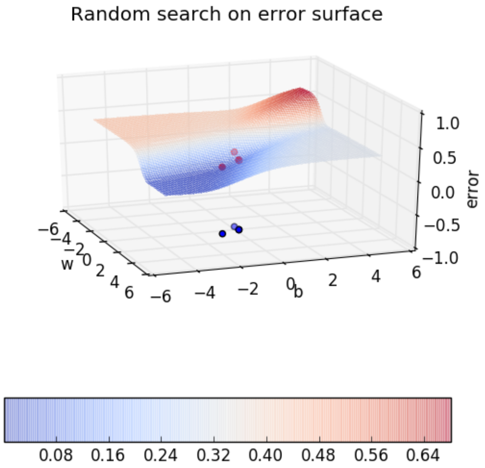

Random search on error surface
w
b
error
1.0
0.5
0.0
−0.5
−1.0
0.08
0.16
0.24
0.32
0.40
0.48
0.56
0.64


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

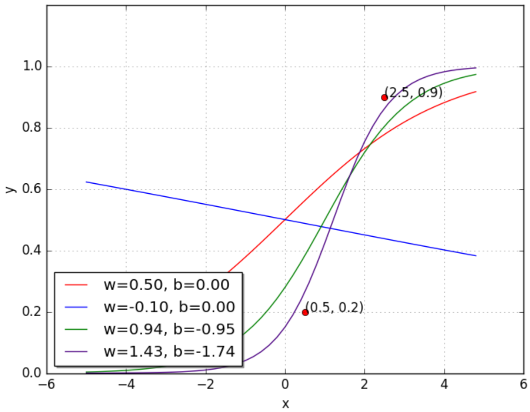

Random search on error surface
w
b
error
1.0
0.5
0.0
−0.5
−1.0
0.08
0.16
0.24
0.32
0.40
0.48
0.56
0.64



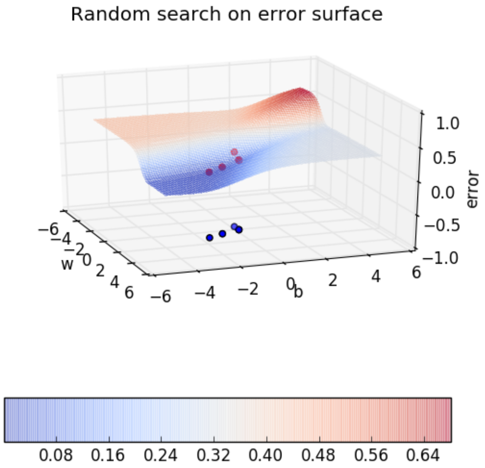
1.0
0.8
0.6
0.4
0.2
0.0
y
−6
−4
−2
0
2
4
6
x
(2.5, 0.9)
(0.5, 0.2)
w=0.50, b=0.00
w=-0.10, b=0.00
w=0.94, b=-0.95



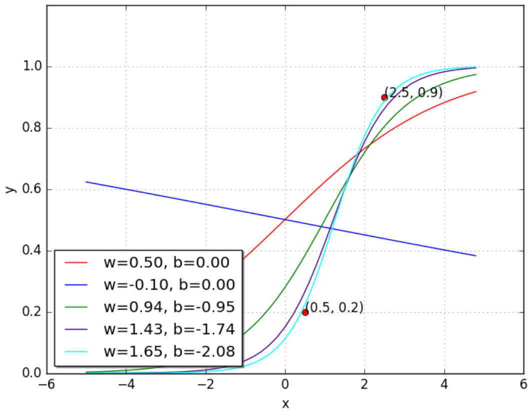
Random search on error surface
1.0
0.5
0.0
−0.5
−1.0
error
w
b
0.08
0.16
0.24
0.32
0.40
0.48
0.56
0.64



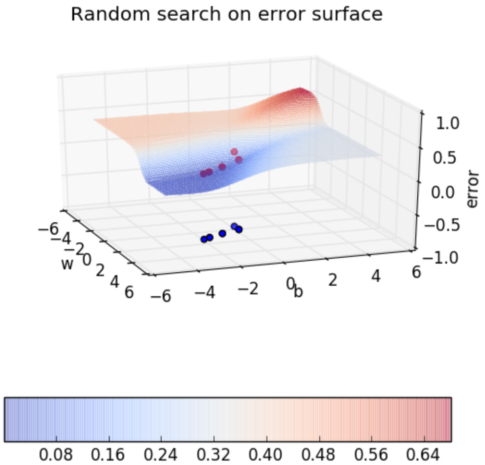
(2.5, 0.9)
(0.5, 0.2)
w=0.50, b=0.00
w=-0.10, b=0.00
w=0.94, b=-0.95
w=1.43, b=-1.74
y
x



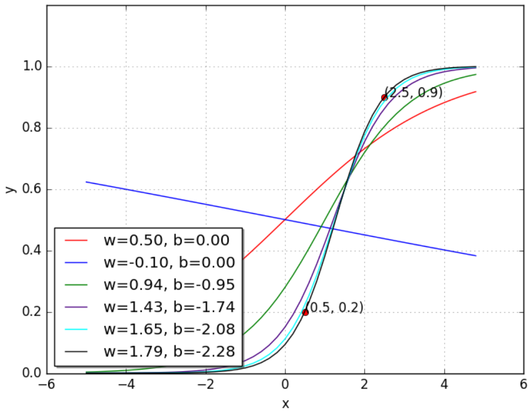
Random search on error surface
error
w
b
0.08
0.16
0.24
0.32
0.40
0.48
0.56
0.64



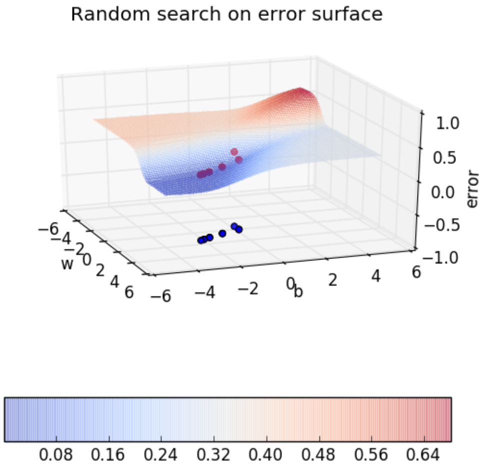
1.0
0.8
0.6
0.4
0.2
0.0
y
−6
−4
−2
0
2
4
6
x
(2.5, 0.9)
(0.5, 0.2)
w=0.50, b=0.00
w=-0.10, b=0.00
w=0.94, b=-0.95
w=1.43, b=-1.74
w=1.65, b=-2.08



Random search on error surface
1.0
0.5
0.0
−0.5
−1.0
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
w=0.94, b=-0.95
w=1.43, b=-1.74
w=1.65, b=-2.08
w=1.79, b=-2.28
y
x



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

Now lets 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

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

vector of parameters,
say, randomly initialized

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

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

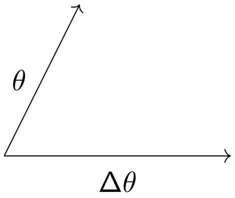
change in the
values of w, b

vector of parameters, say, randomly initialized

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

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

change in the values of w, b

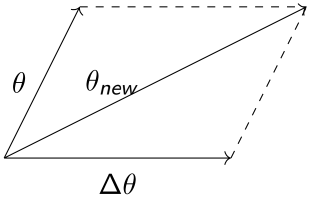


vector of parameters, say, randomly initialized

$\theta = [w, b]$

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

change in the values of w, b

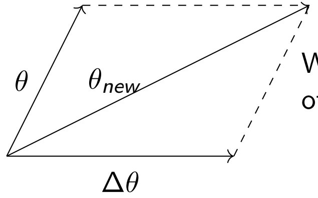


vector of parameters, say, randomly initialized

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

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

change in the values of w, b



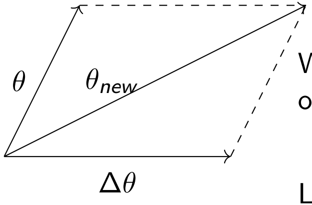
We moved in the direction of $\Delta\theta$

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 $\Delta\theta$

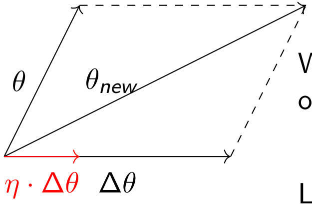
Lets be a bit conservative: move only by a small amount η

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 $\Delta\theta$

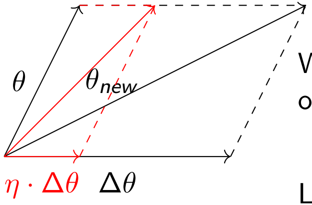
Lets be a bit conservative: move only by a small amount η

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 $\Delta\theta$

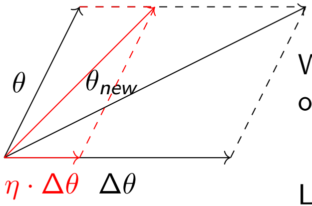
Lets be a bit conservative: move only by a small amount η

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 $\Delta\theta$

Lets be a bit conservative: move only by a small amount η

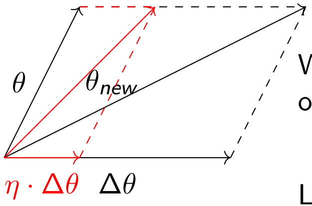
$$\theta_{new} = \theta + \eta \cdot \Delta\theta$$

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 $\Delta\theta$

Lets be a bit conservative: move only by a small amount η

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

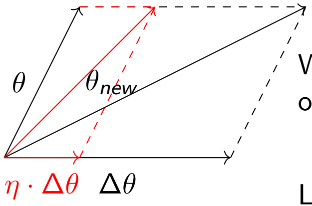
Question: What is the right $\Delta\theta$ to use ?

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 $\Delta\theta$

Lets 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,

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

$$\mathscr{L}(\theta + \eta u) = \mathscr{L}(\theta) + \eta * u^T \nabla \mathscr{L}(\theta) + \frac{\eta^2}{2!} * u^T \nabla^2 \mathscr{L}(\theta) u + \frac{\eta^3}{3!} * \ldots + \frac{\eta^4}{4!} * \ldots$$

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

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

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

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

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

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

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

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

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,

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

Okay, so we have,

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

But, what is the range of $u^T \nabla \mathscr{L}(\theta)$?

Okay, so we have,

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

But, what is the range of $u^T \nabla \mathscr{L}(\theta)$? Lets see....

Okay, so we have,

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

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

Okay, so we have,

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

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

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

Okay, so we have,

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

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

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

Okay, so we have,

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

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

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

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

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

Okay, so we have,

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

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

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

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

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

Thus, $\mathscr{L}(\theta + \eta u) - \mathscr{L}(\theta) = u^T \nabla \mathscr{L}(\theta) = k * cos(\beta)$ 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

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

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

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

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

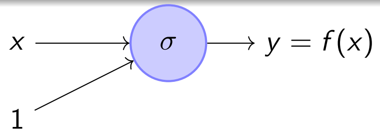
- Lets create an algorithm from this rule ...

- Lets create an algorithm from this rule ...

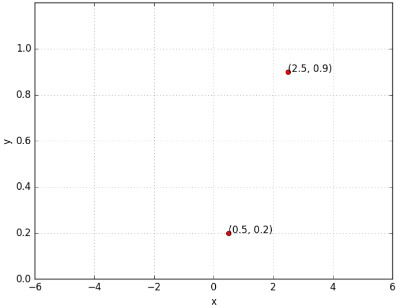
Algorithm: gradient_descent()

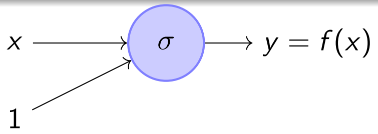
$t \leftarrow 0$;
$max\_iterations \leftarrow 1000$;
while $t < max\_iterations$ **do**
  $w_{t+1} \leftarrow w_t - \eta \nabla w_t$;
  $b_{t+1} \leftarrow b_t - \eta \nabla b_t$;
end

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

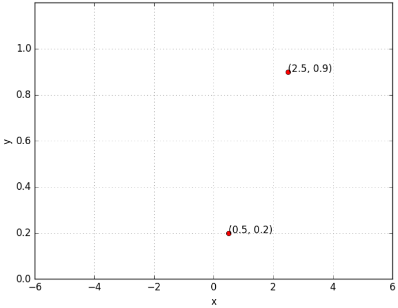


$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

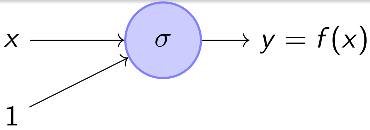




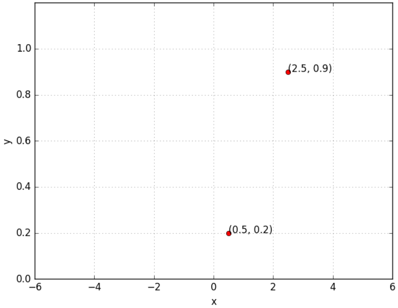
$$f(x) = \frac{1}{1+e^{-(w \cdot x+b)}}$$



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

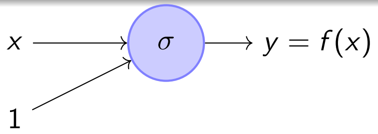


$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

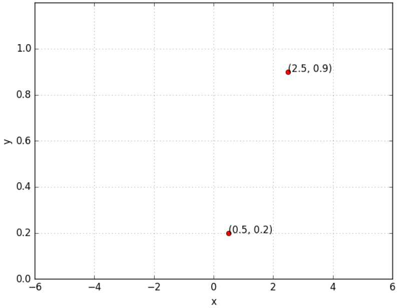


Let's assume there is only 1 point to fit (x, y)

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



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



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

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

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

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

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

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

$$= \frac{1}{2} * [2 * (f(x) - y) * \frac{\partial}{\partial w}(f(x) - y)]$$

$$= (f(x) - y) * \frac{\partial}{\partial w}(f(x))$$

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

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

$$\frac{\partial}{\partial w}\Big(\frac{1}{1 + e^{-(wx+b)}}\Big)$$

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

$$\begin{aligned}&\frac{\partial}{\partial w}\left(\frac{1}{1 + e^{-(wx+b)}}\right) \\ &= \frac{-1}{(1 + e^{-(wx+b)})^2}\frac{\partial}{\partial w}(e^{-(wx+b)}))\end{aligned}$$

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

$$\frac{\partial}{\partial w}\Big(\frac{1}{1 + e^{-(wx+b)}}\Big)$$
$$= \frac{-1}{(1 + e^{-(wx+b)})^2}\frac{\partial}{\partial w}(e^{-(wx+b)}))$$
$$= \frac{-1}{(1 + e^{-(wx+b)})^2} * (e^{-(wx+b)})\frac{\partial}{\partial w}(-(wx + b)))$$

$$\nabla w = \frac{\partial}{\partial w}[\frac{1}{2} * (f(x) - y)^2]$$
$$= \frac{1}{2} * [2 * (f(x) - y) * \frac{\partial}{\partial w}(f(x) - y)]$$
$$= (f(x) - y) * \frac{\partial}{\partial w}(f(x))$$
$$= (f(x) - y) * \frac{\partial}{\partial w}\left(\frac{1}{1 + e^{-(wx+b)}}\right)$$

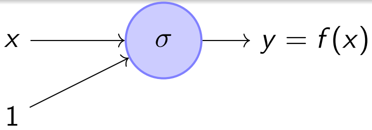
$$\frac{\partial}{\partial w}\left(\frac{1}{1 + e^{-(wx+b)}}\right)$$
$$= \frac{-1}{(1 + e^{-(wx+b)})^2}\frac{\partial}{\partial w}(e^{-(wx+b)}))$$
$$= \frac{-1}{(1 + e^{-(wx+b)})^2} * (e^{-(wx+b)})\frac{\partial}{\partial w}(-(wx + b)))$$
$$= \frac{-1}{(1 + e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1 + e^{-(wx+b)})} * (-x)$$
$$= \frac{1}{(1 + e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1 + e^{-(wx+b)})} * (x)$$

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

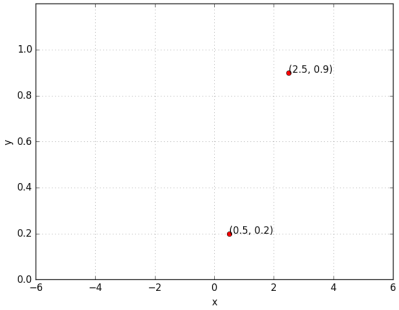
$$\frac{\partial}{\partial w}\Big(\frac{1}{1 + e^{-(wx+b)}}\Big)$$
$$= \frac{-1}{(1 + e^{-(wx+b)})^2}\frac{\partial}{\partial w}(e^{-(wx+b)}))$$
$$= \frac{-1}{(1 + e^{-(wx+b)})^2} * (e^{-(wx+b)})\frac{\partial}{\partial w}(-(wx + b)))$$
$$= \frac{-1}{(1 + e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1 + e^{-(wx+b)})}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

$$\nabla w = \frac{\partial}{\partial w}[\frac{1}{2} * (f(x) - y)^2]$$
$$= \frac{1}{2} * [2 * (f(x) - y) * \frac{\partial}{\partial w}(f(x) - y)]$$
$$= (f(x) - y) * \frac{\partial}{\partial w}(f(x))$$
$$= (f(x) - y) * \frac{\partial}{\partial w}\left(\frac{1}{1 + e^{-(wx+b)}}\right)$$
$$= (f(x) - y) * f(x) * (1 - f(x)) * x$$

$$\frac{\partial}{\partial w}\left(\frac{1}{1 + e^{-(wx+b)}}\right)$$
$$= \frac{-1}{(1 + e^{-(wx+b)})^2}\frac{\partial}{\partial w}(e^{-(wx+b)}))$$
$$= \frac{-1}{(1 + e^{-(wx+b)})^2} * (e^{-(wx+b)})\frac{\partial}{\partial w}(-(wx + b)))$$
$$= \frac{-1}{(1 + e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1 + e^{-(wx+b)})} * (-x)$$
$$= \frac{1}{(1 + e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1 + e^{-(wx+b)})} * (x)$$
$$= f(x) * (1 - f(x)) * x$$



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

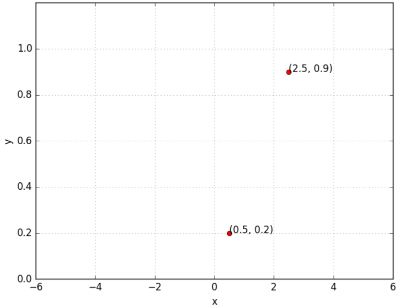


So if there is only 1 point (x, y), we have,

$$x \longrightarrow \sigma \longrightarrow y = f(x)$$

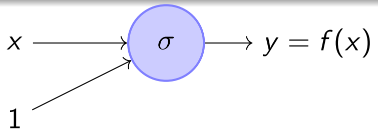
1

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

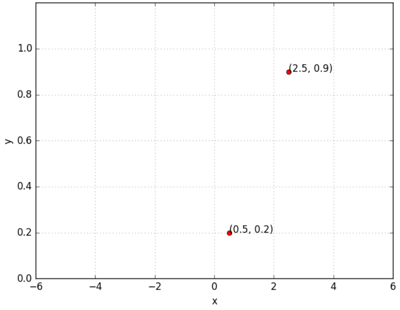


So if there is only 1 point (x, y), we have,

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



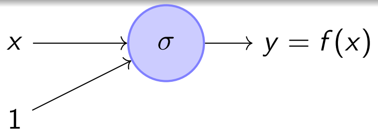
$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$



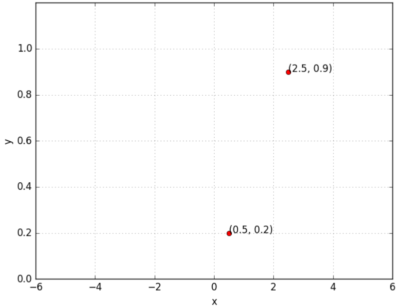
So if there is only 1 point (x, y), we have,

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

For two points,



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

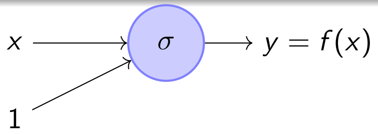


So if there is only 1 point (x, y), we have,

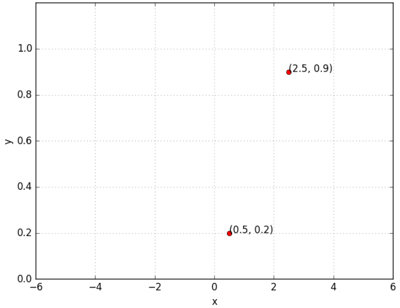
$$\nabla w = (f(x) - y) * f(x) * (1 - f(x)) * x$$

For two points,

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



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



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,

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

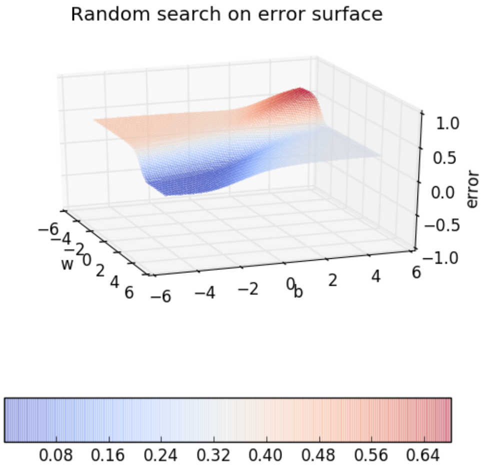
$$\nabla b = \sum_{i=1}^{2} (f(x_i) - y_i) * f(x_i) * (1 - f(x_i))$$

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

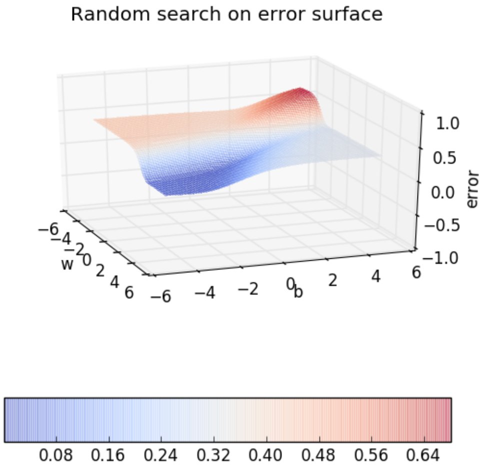
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

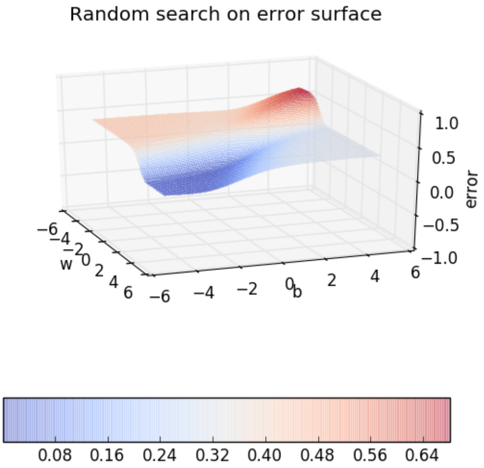
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



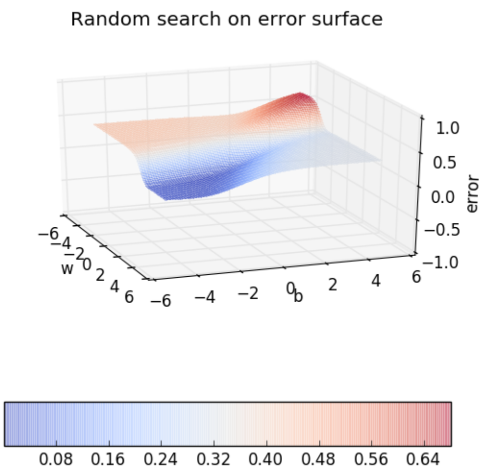
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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)
```



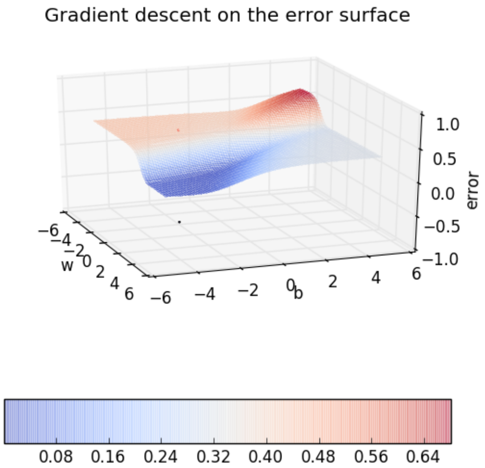
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



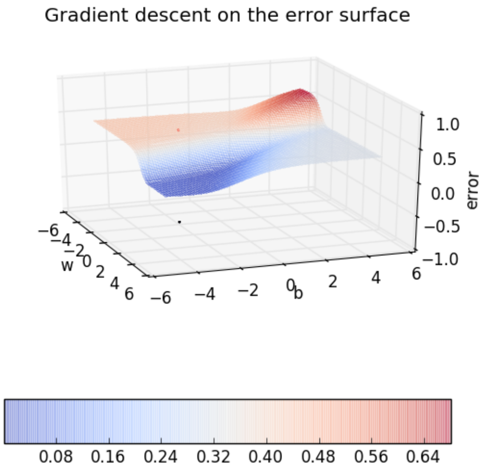
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



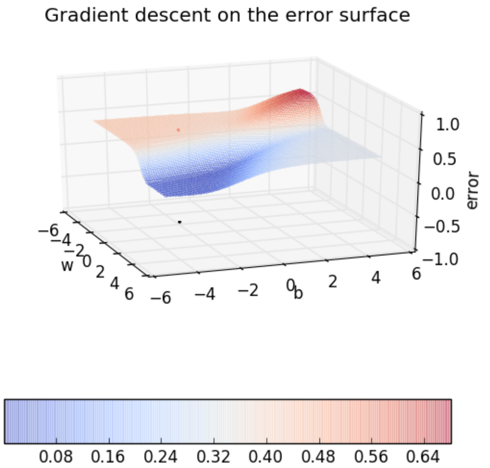
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



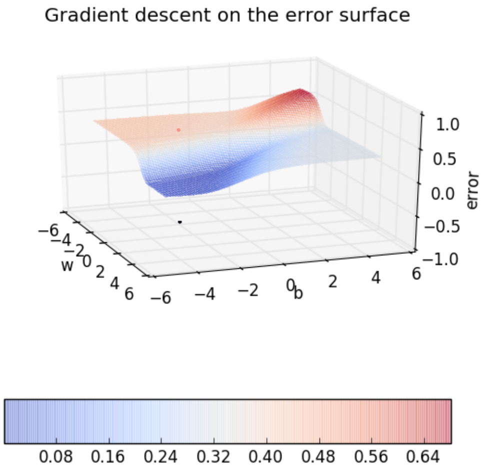
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



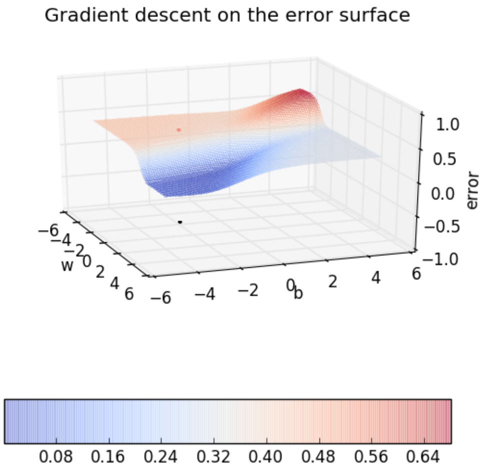
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



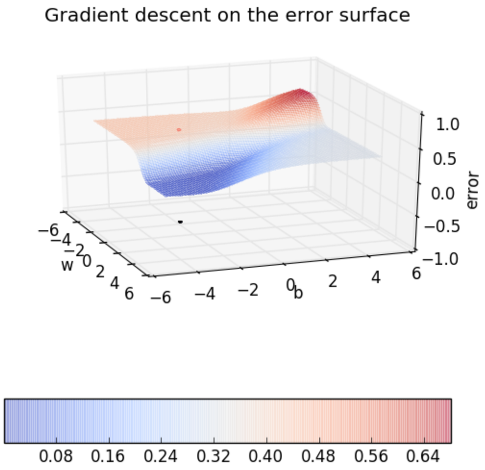
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



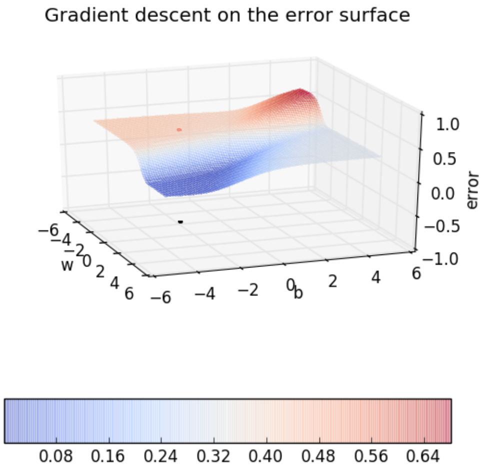
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



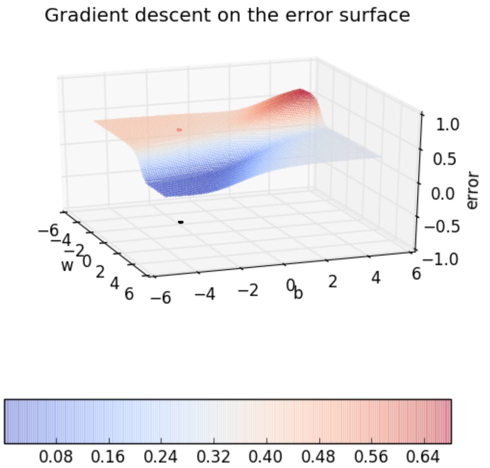
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



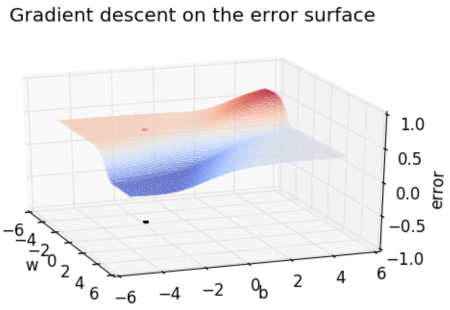
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

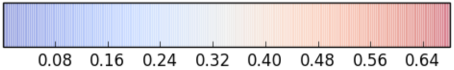


```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



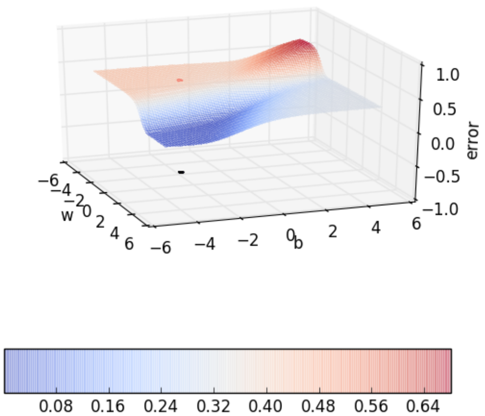
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```





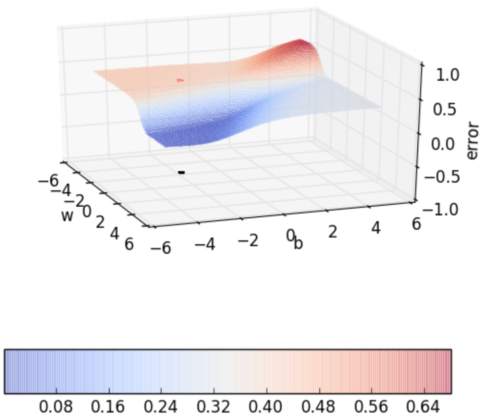
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

Gradient descent on the error surface

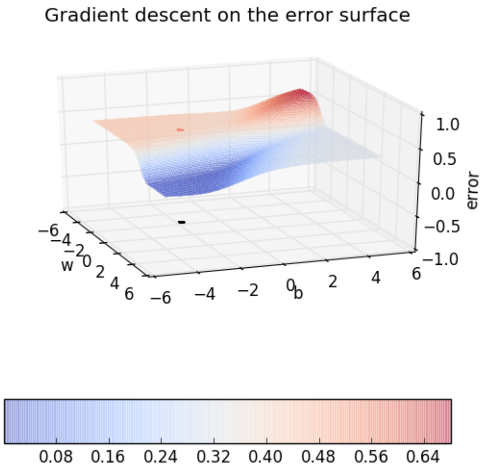


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

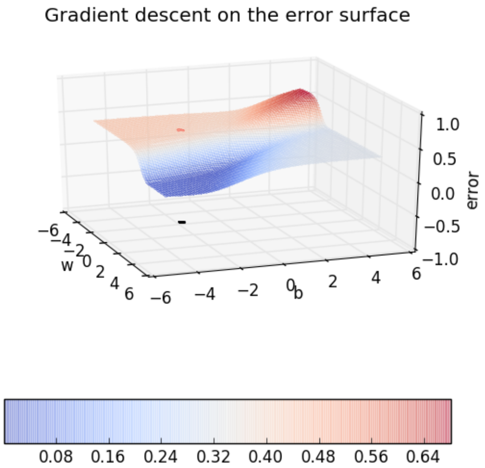
Gradient descent on the error surface



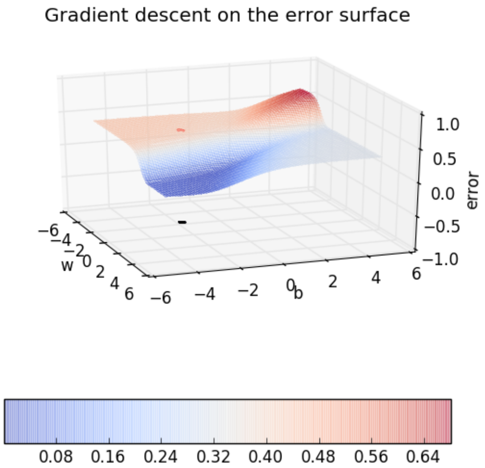
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



```
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
```

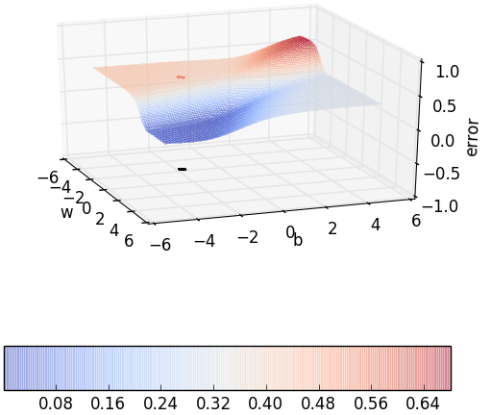


```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

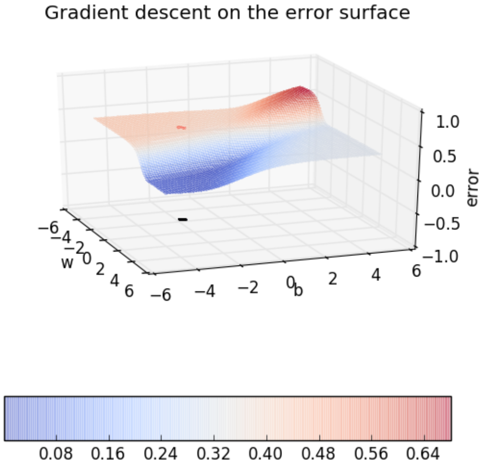


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

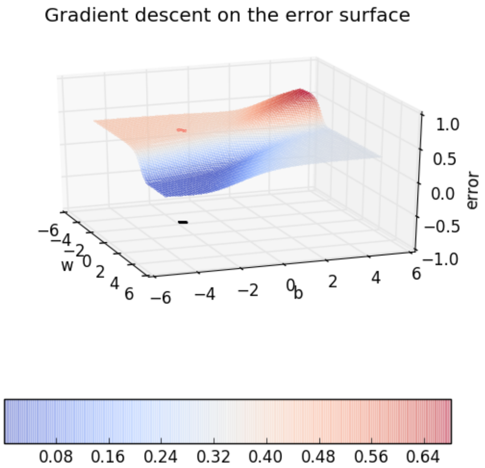
Gradient descent on the error surface



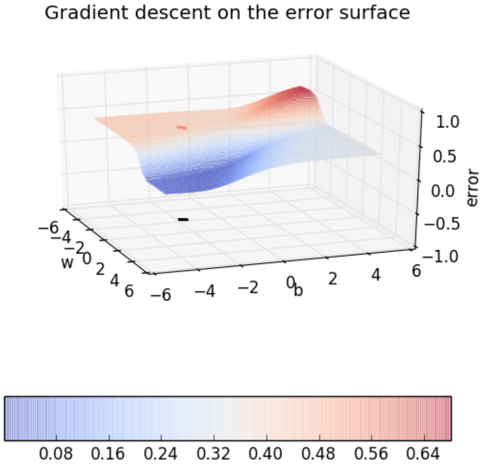
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



```python
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
```

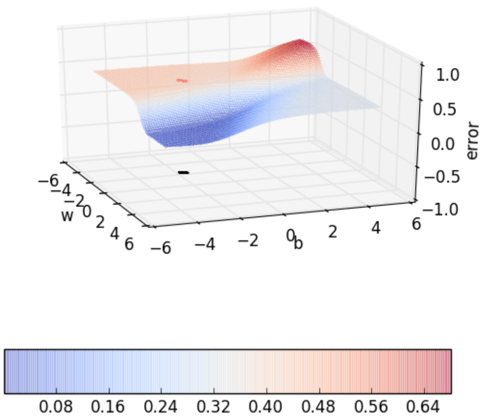


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

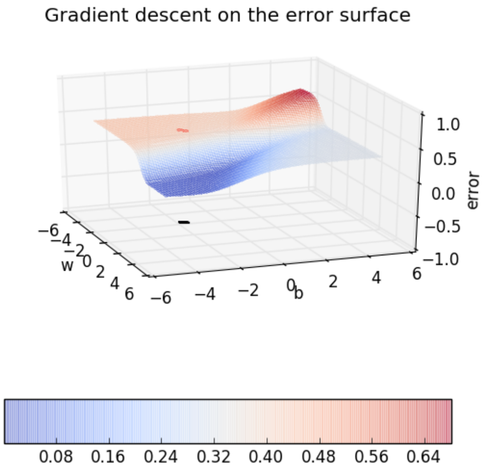


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

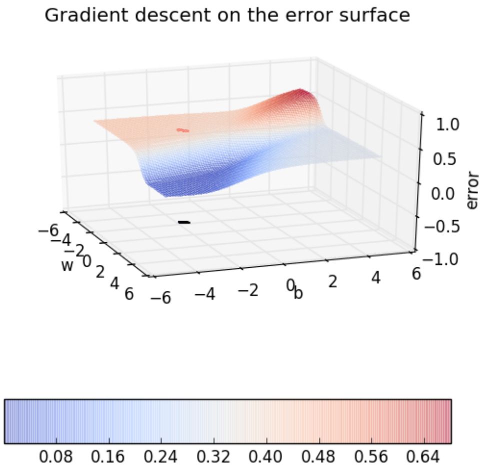
Gradient descent on the error surface



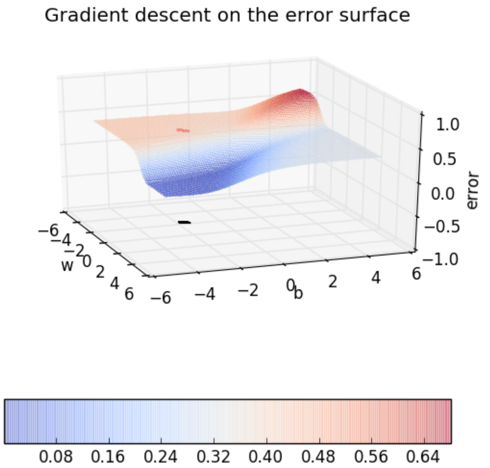
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



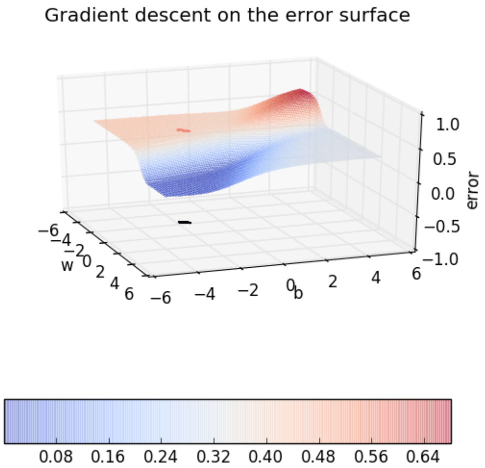
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



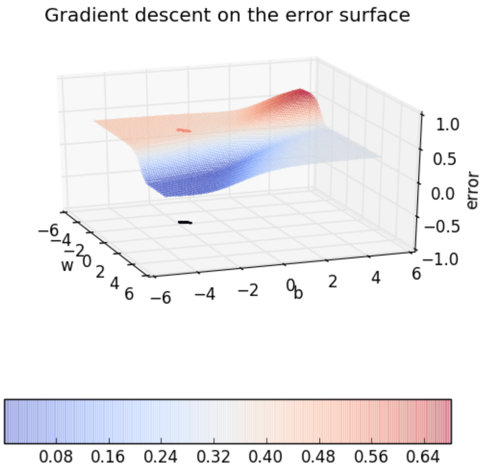
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



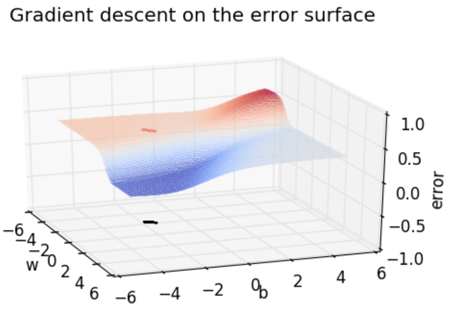
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

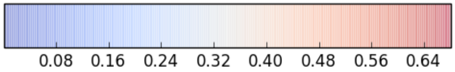


```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

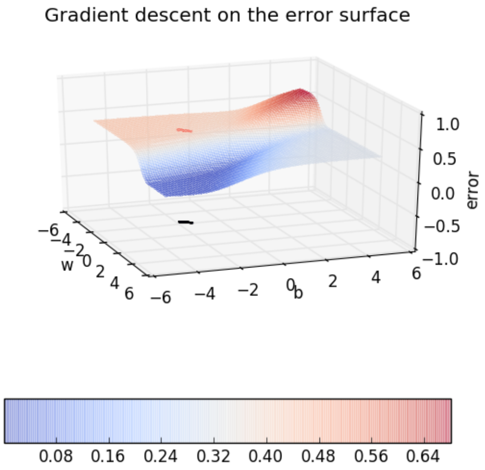


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



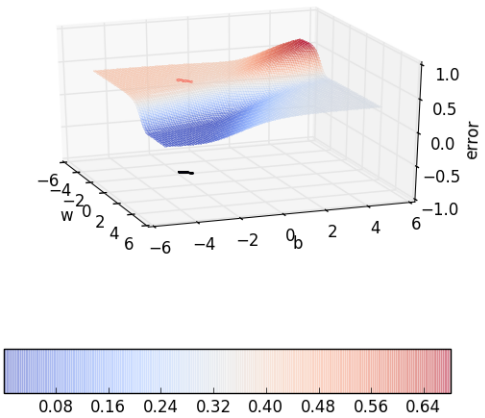


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

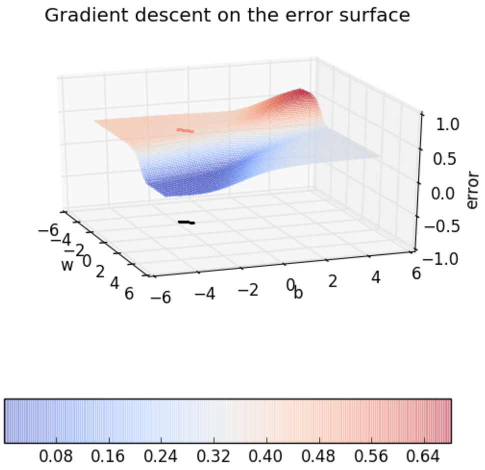


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

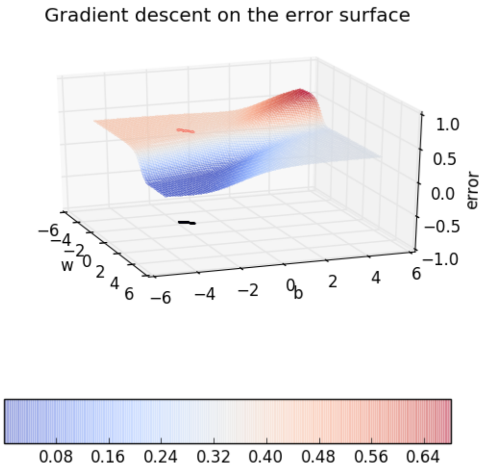
Gradient descent on the error surface



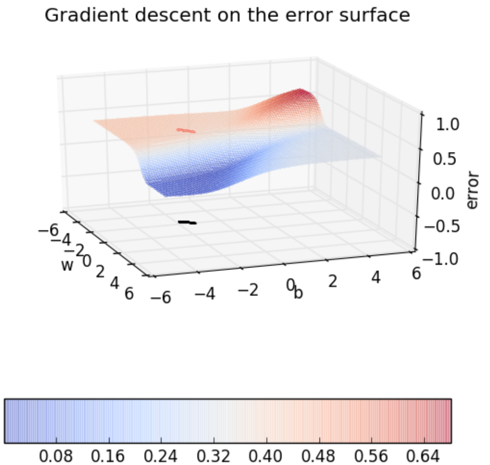
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



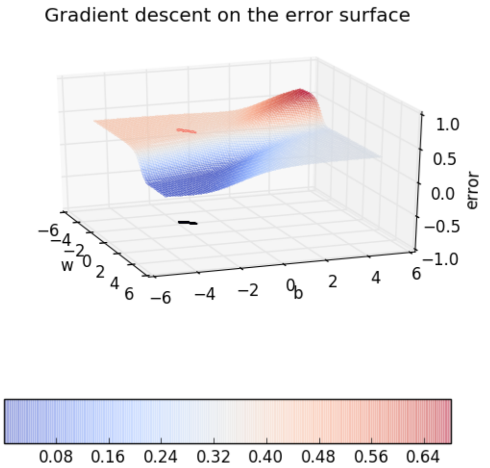
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



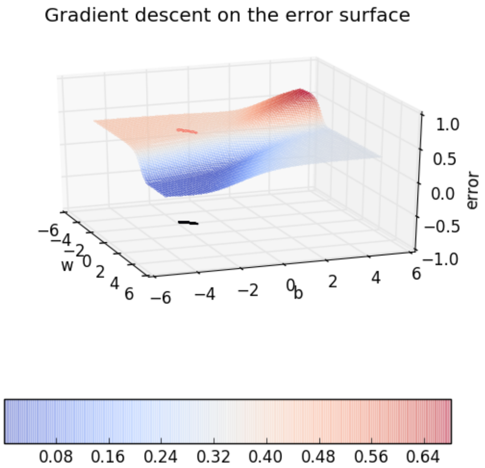
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



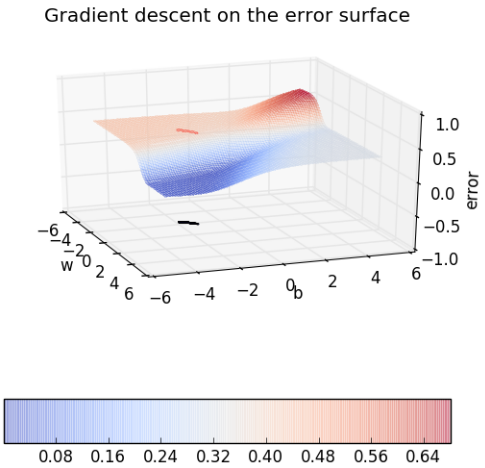
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



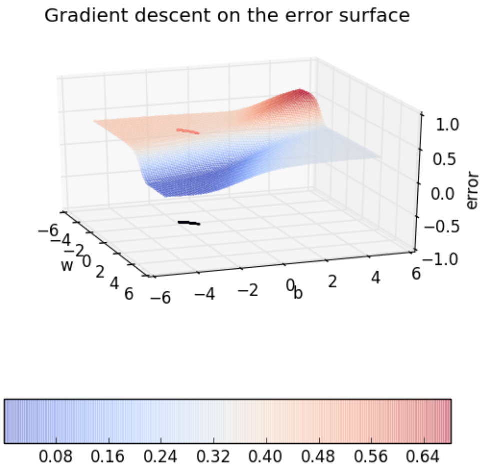
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



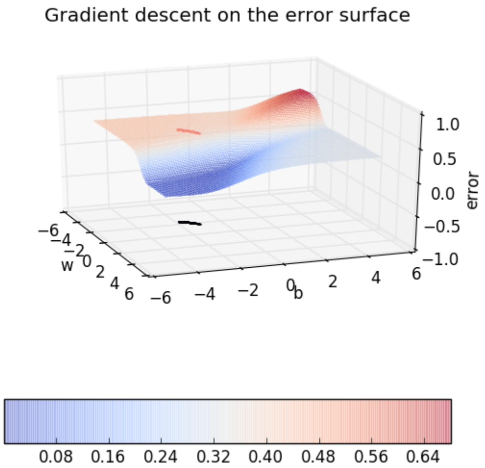
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



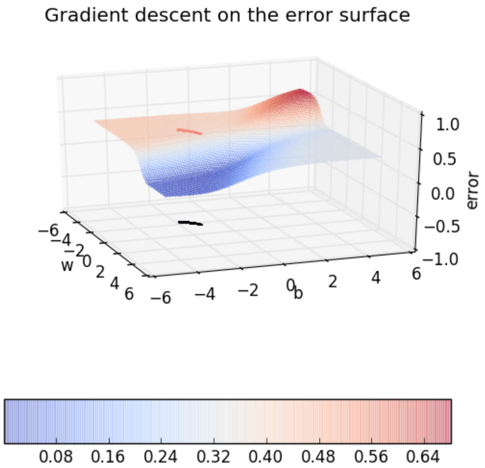
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



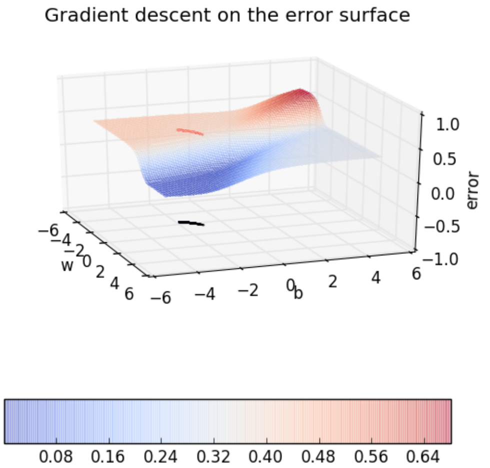
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



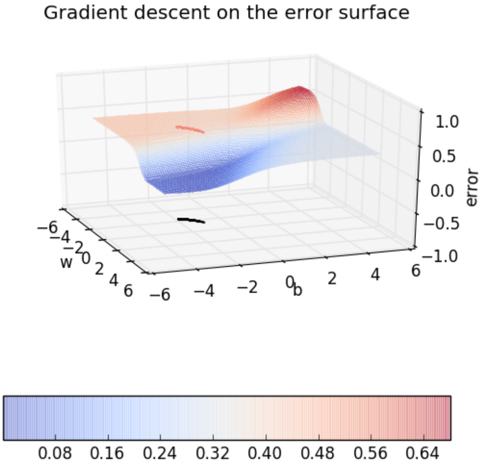
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



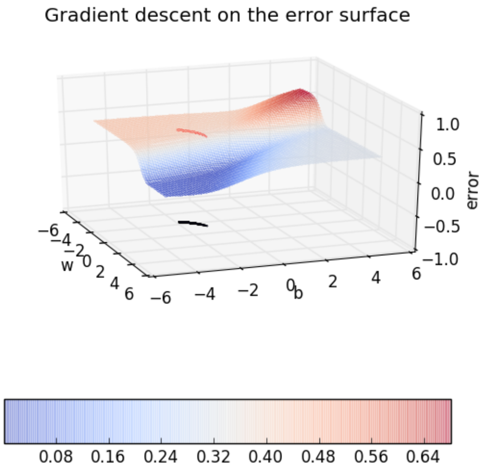
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



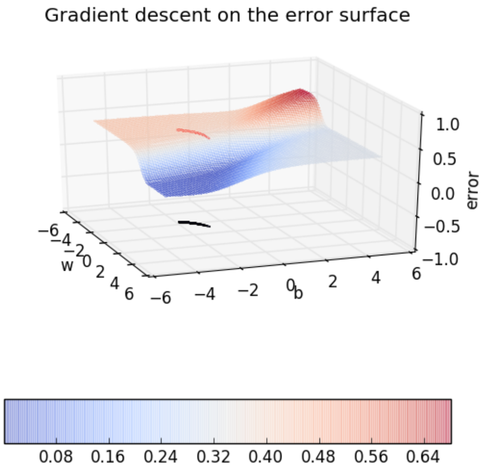
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



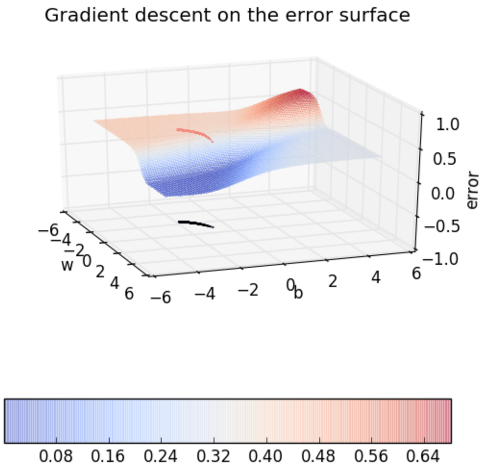
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



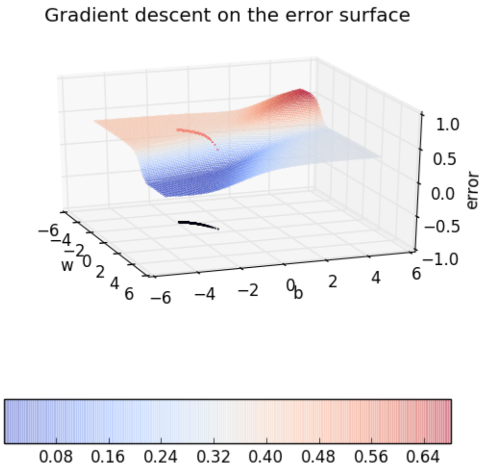
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



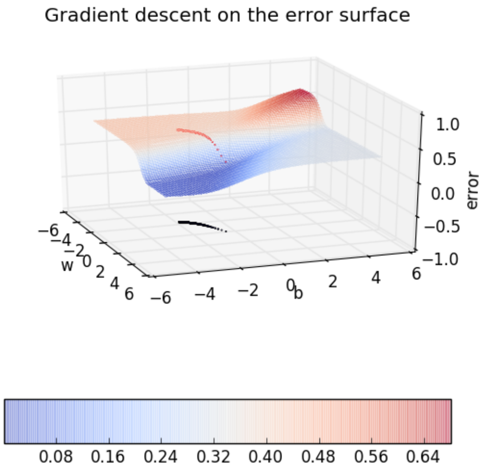
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



```
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
```

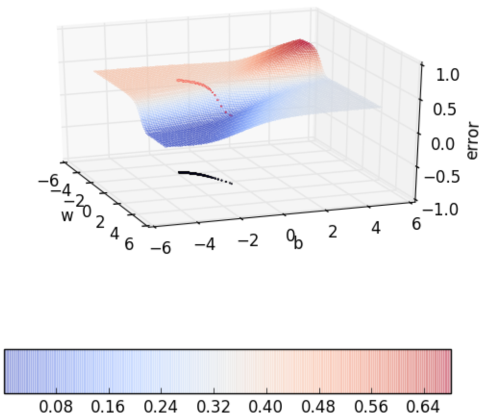


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

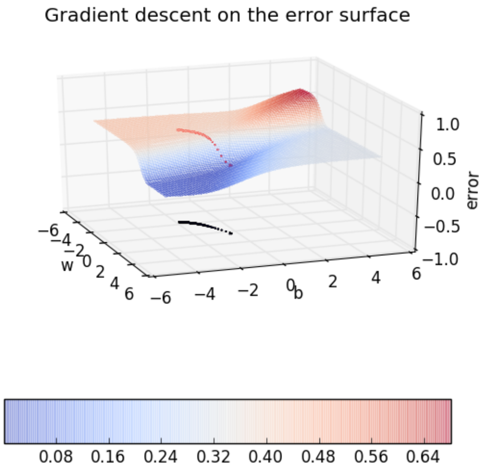


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

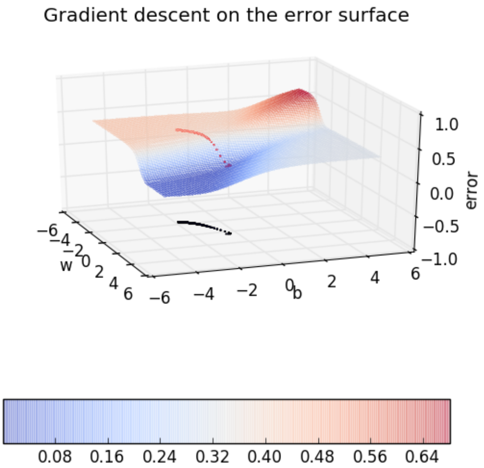
Gradient descent on the error surface



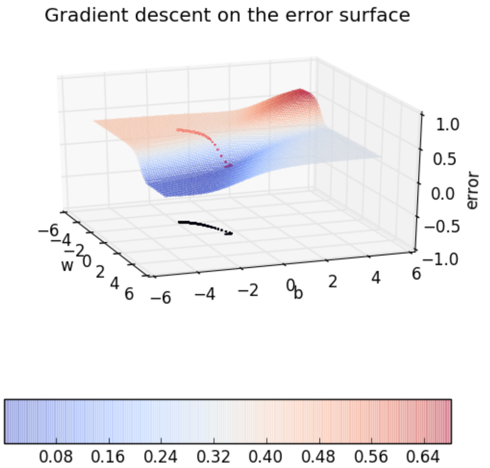
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



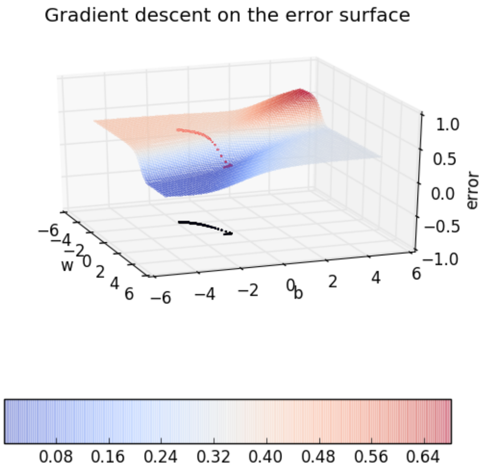
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



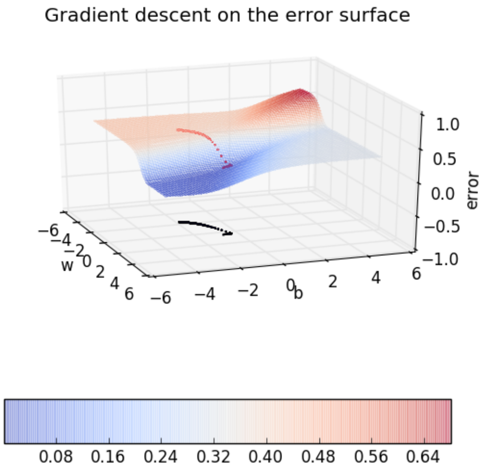
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



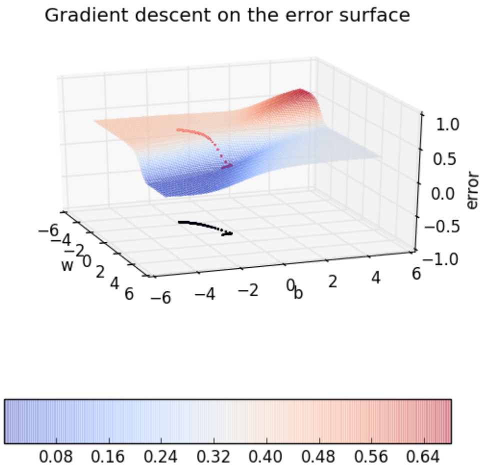
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



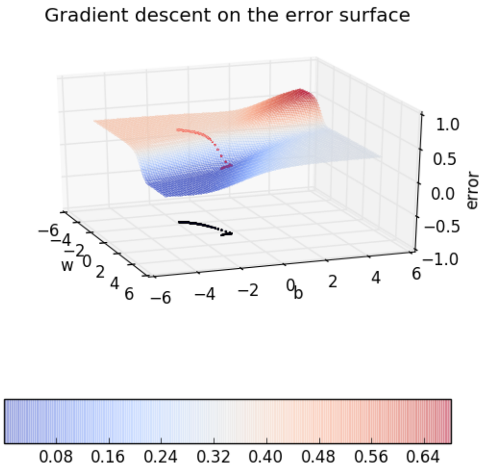
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



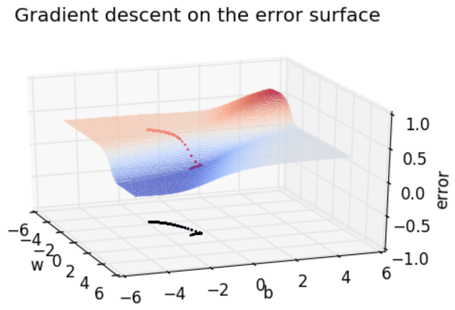
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

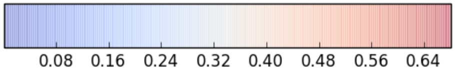


```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



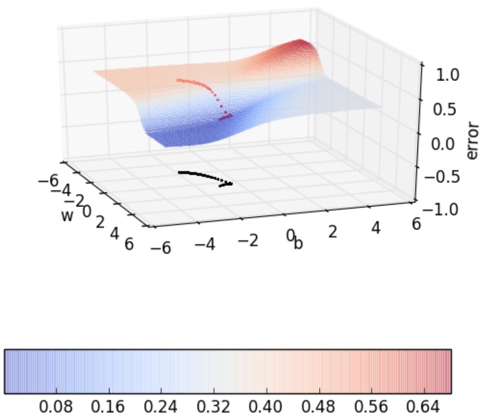
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



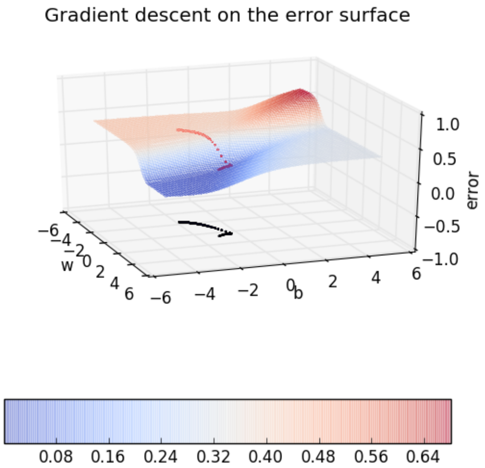


```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

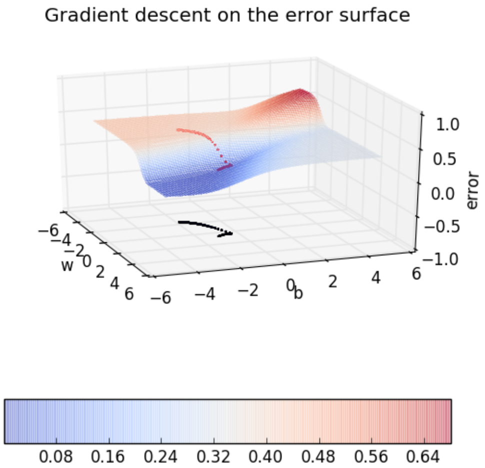
Gradient descent on the error surface



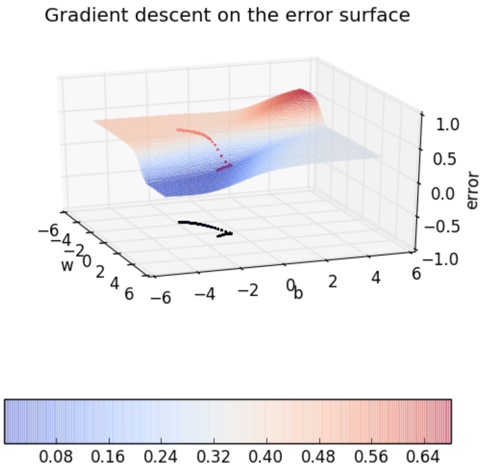
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



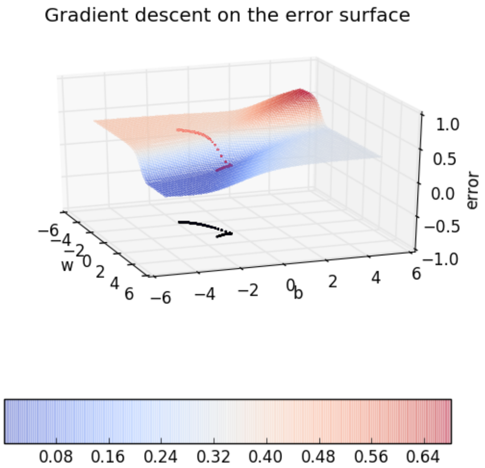
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



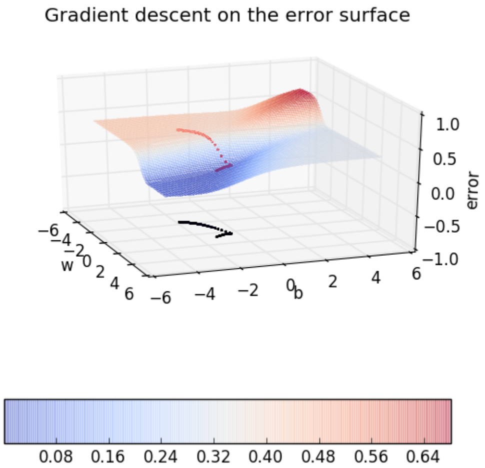
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



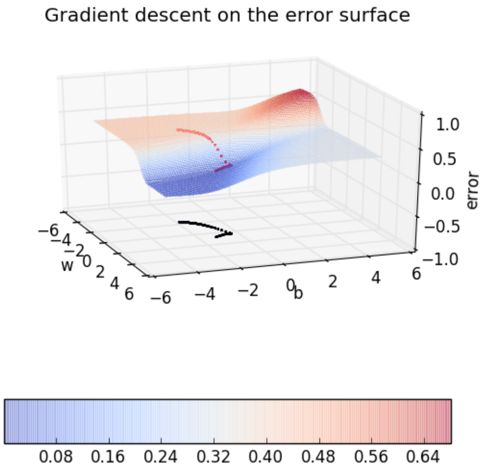
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



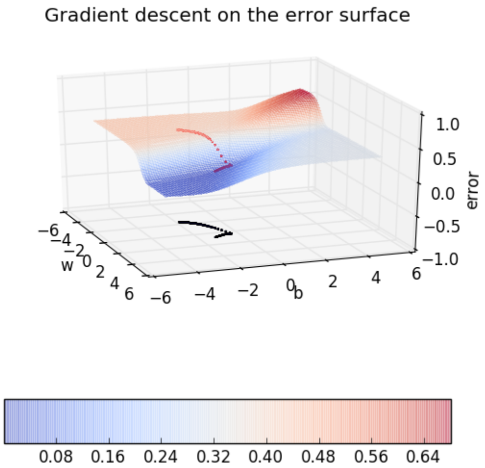
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



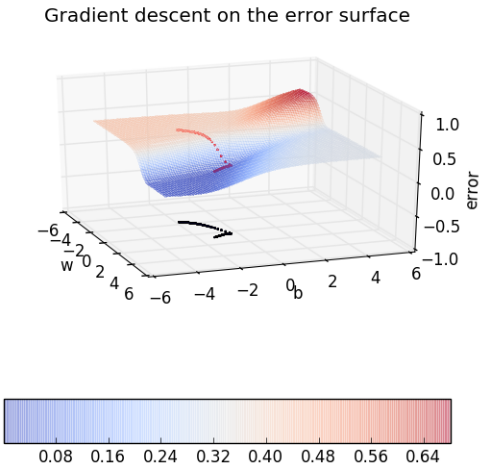
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



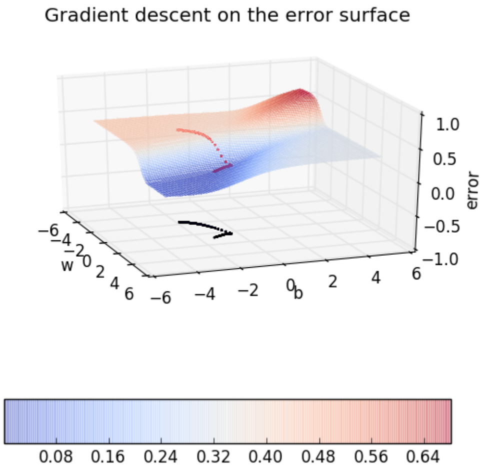
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



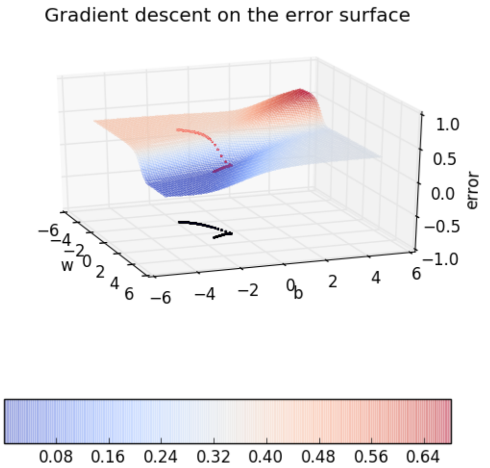
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



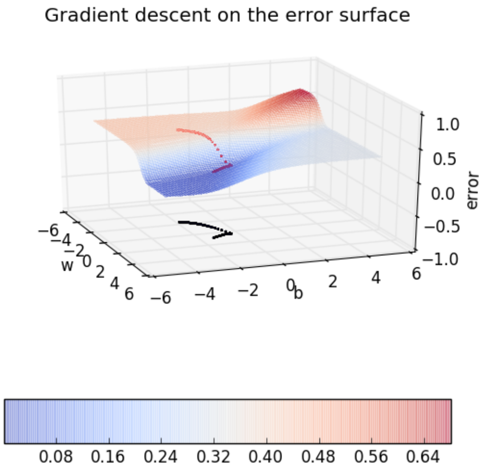
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



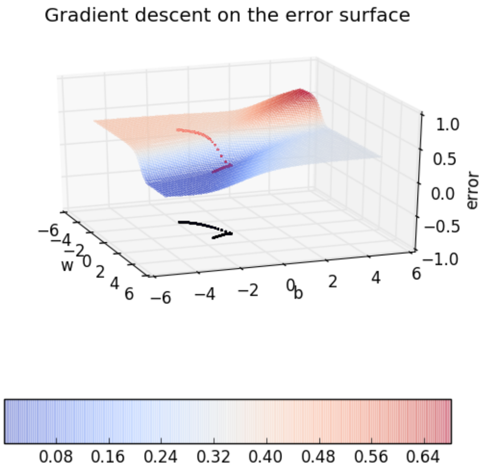
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



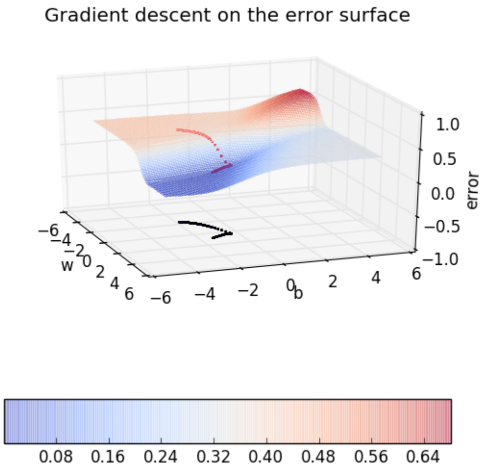
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



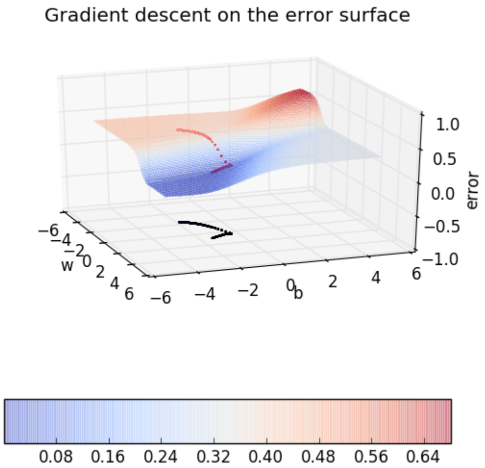
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



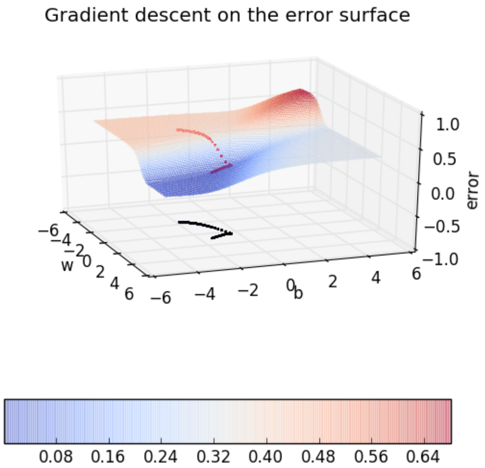
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



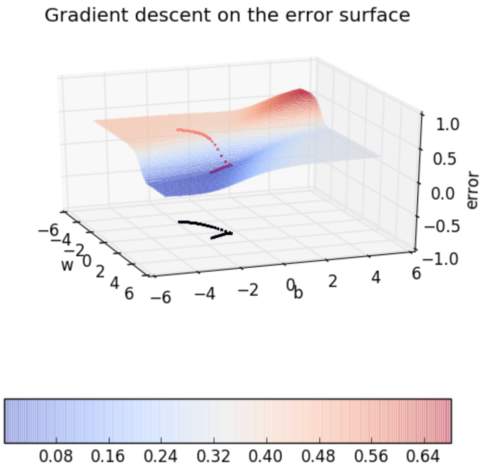
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



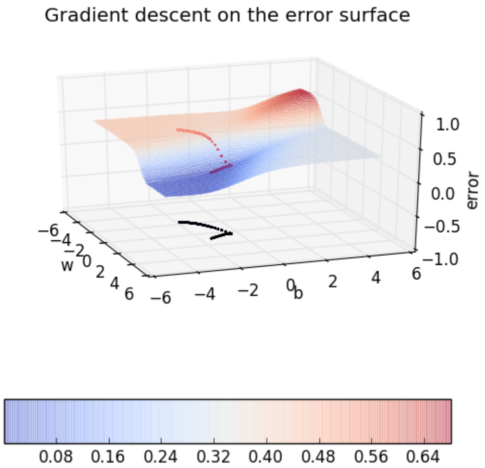
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



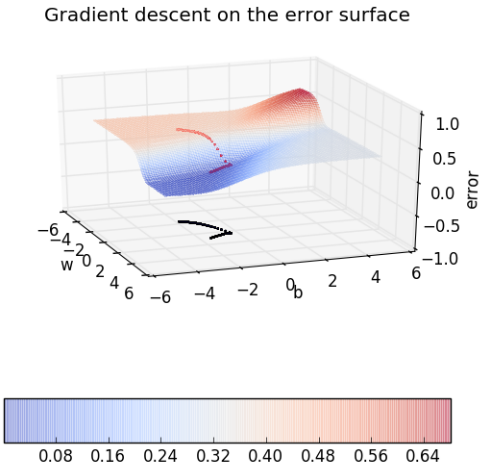
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



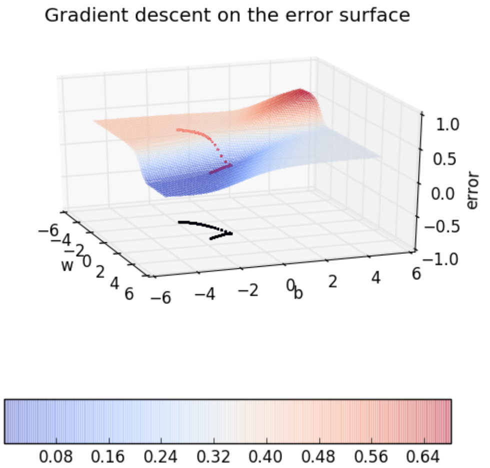
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



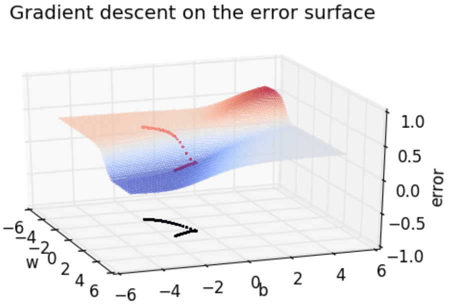
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

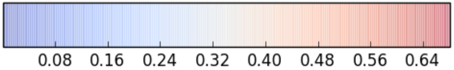


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

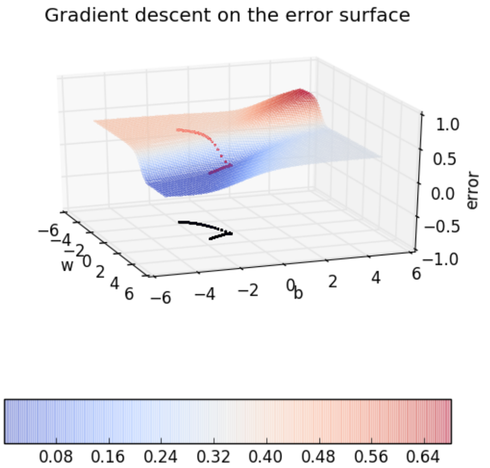


```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



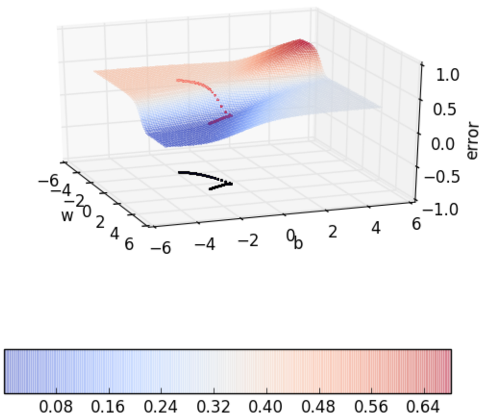


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

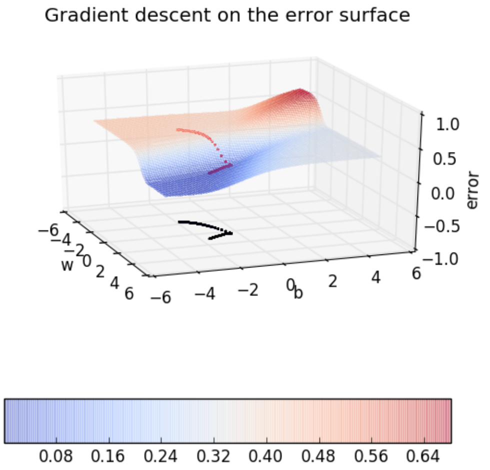


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

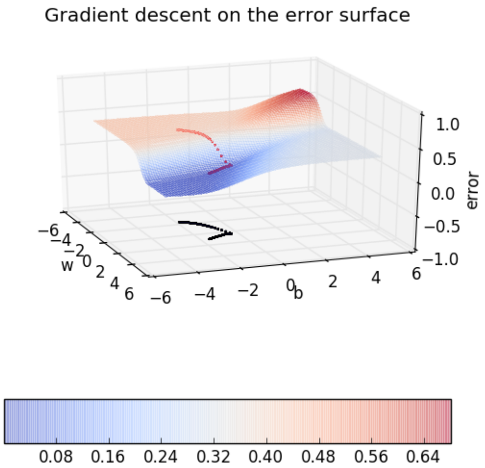
Gradient descent on the error surface



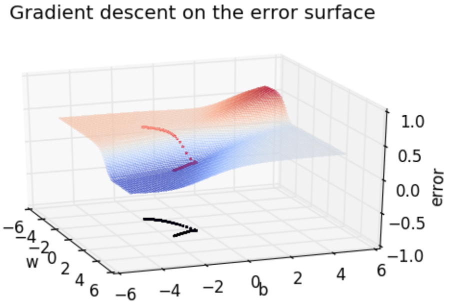
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

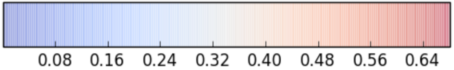


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

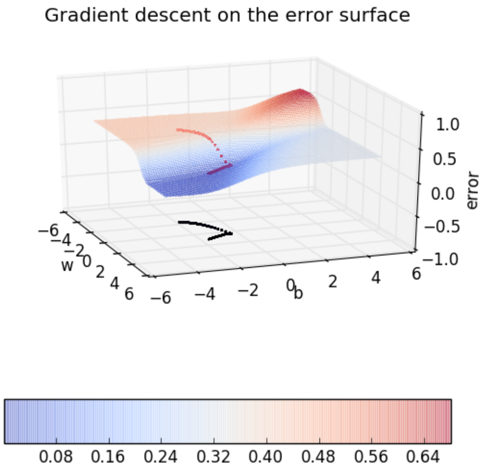


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

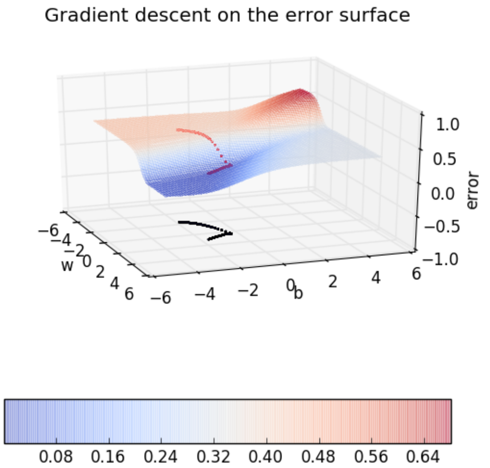




```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

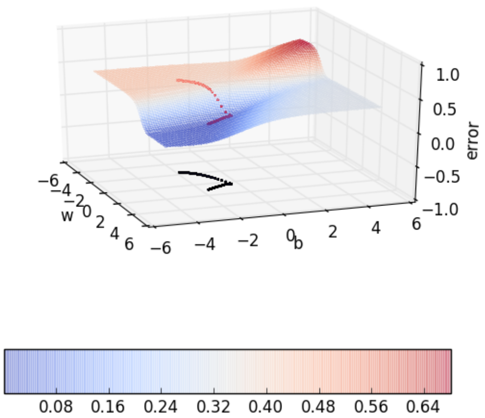


```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

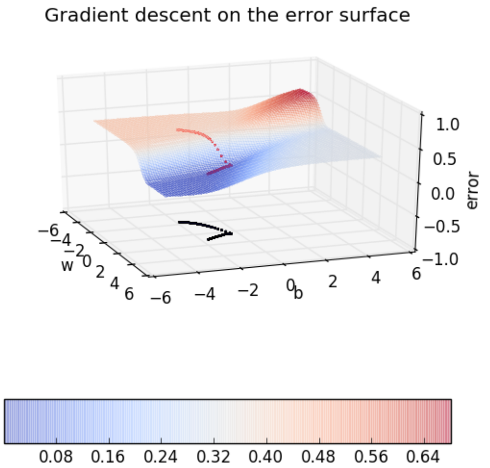


```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

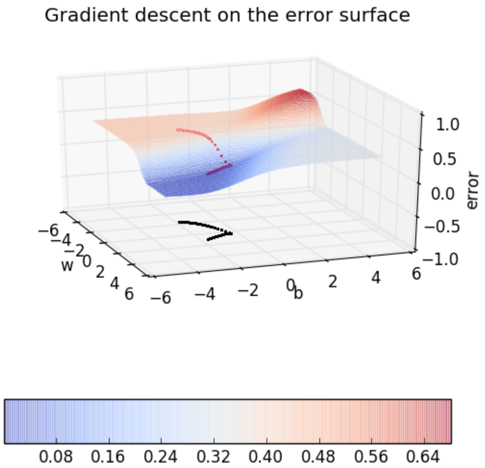
Gradient descent on the error surface



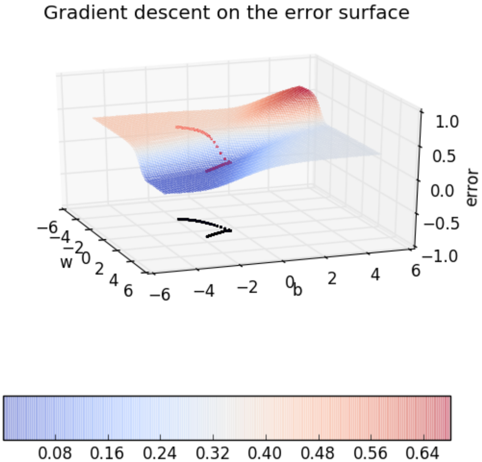
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



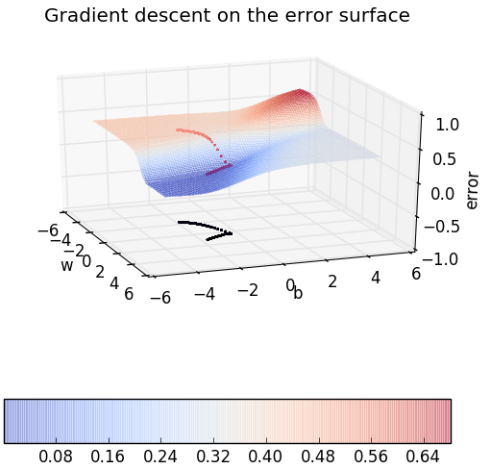
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



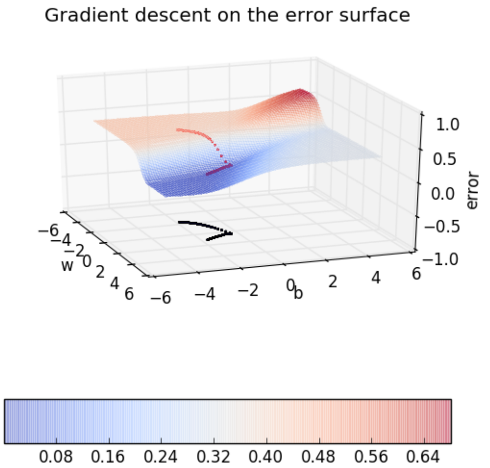
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



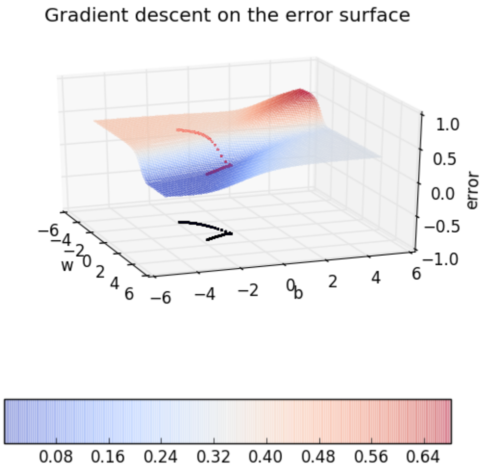
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



```
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
```

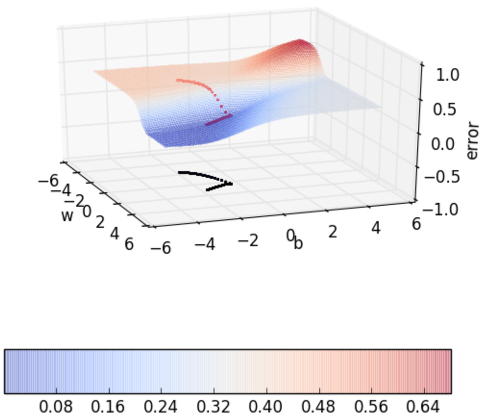


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

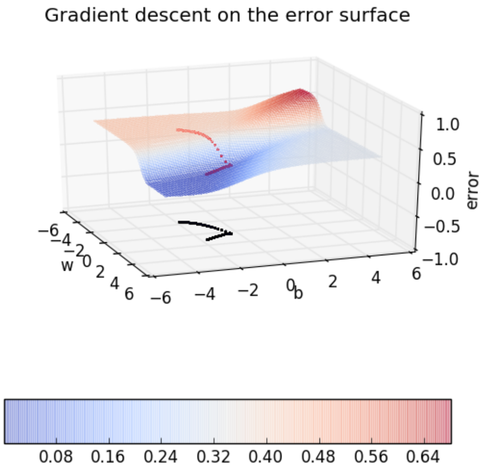


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

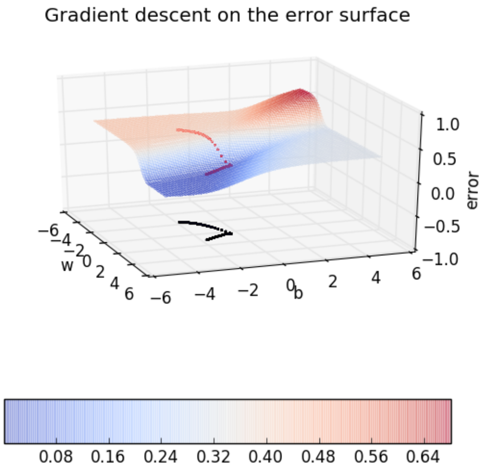
Gradient descent on the error surface



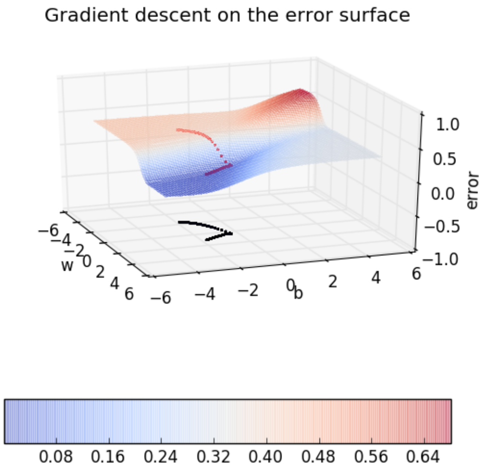
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



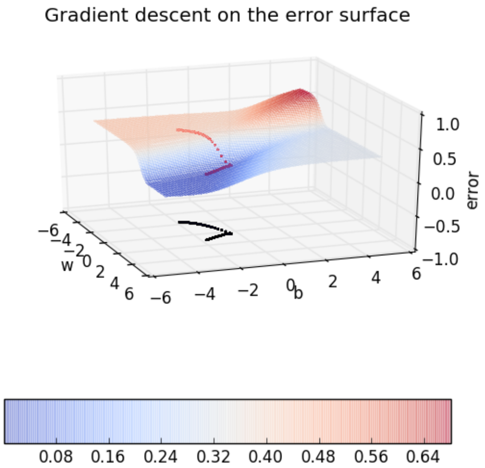
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



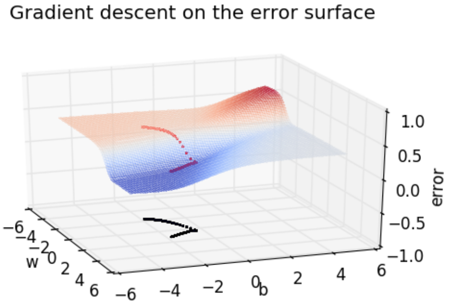
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

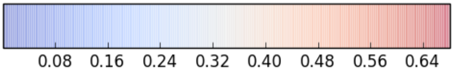


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

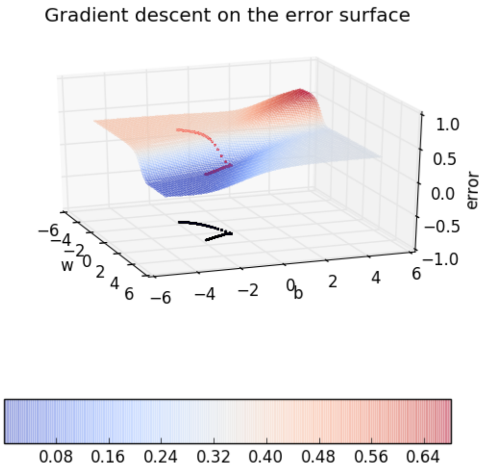


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

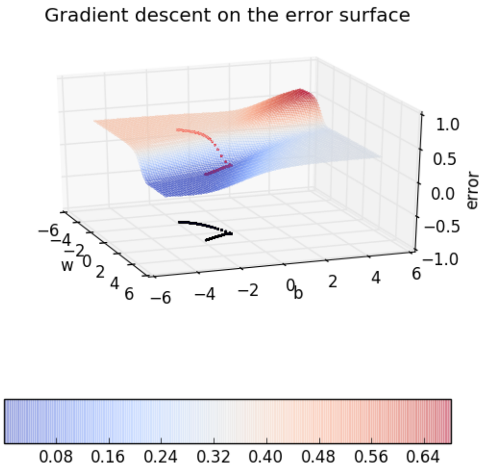




```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

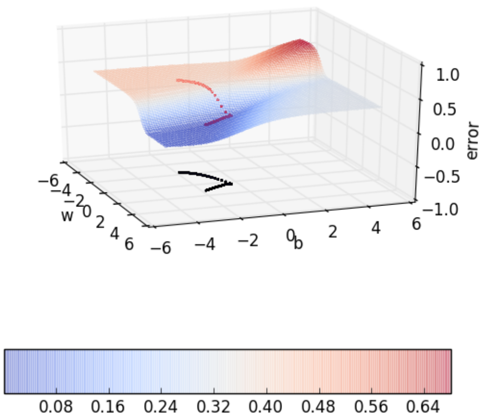


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

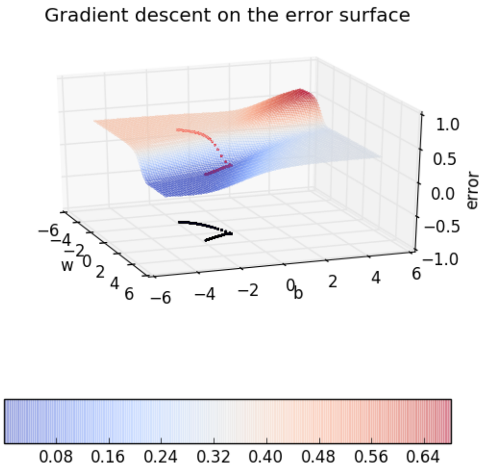


```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```

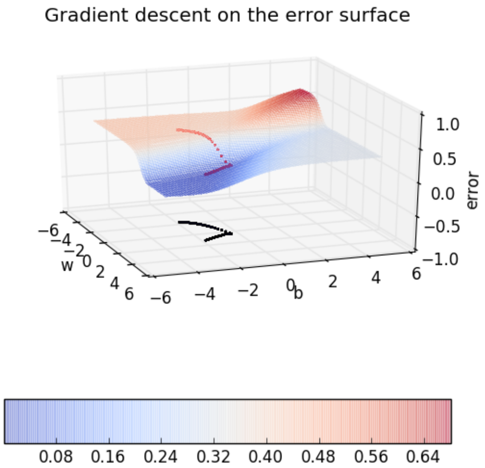
Gradient descent on the error surface



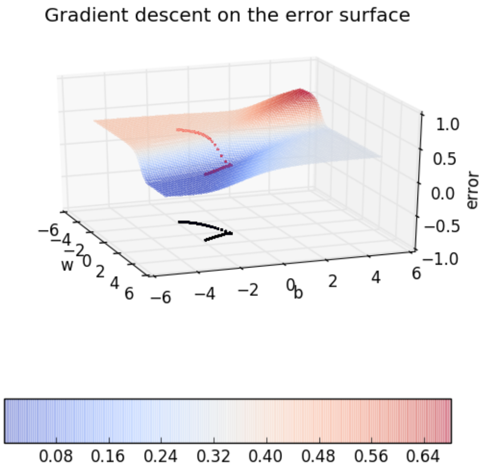
```
X = [0.5, 2.5]
Y = [0.2, 0.9]

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
```



```
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
```



```
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

- 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

- 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 ?

Representation power of a multilayer network of perceptrons

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

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

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 $|g(x) - f(x)| < \epsilon$!!

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

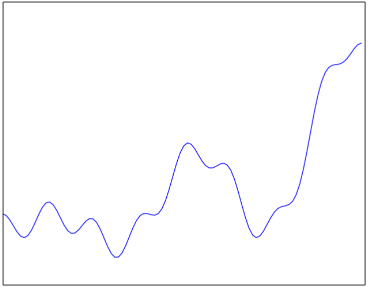
A multilayer network of neurons with a single hidden layer can be used to approximate any continuous function to any desired precision

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 $|g(x) - f(x)| < \epsilon$!!

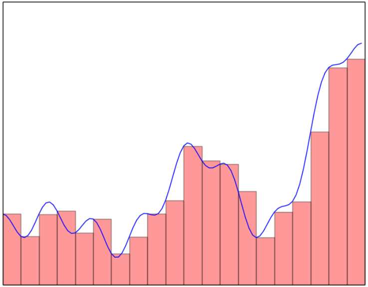
Proof: We will see an illustrative proof of this... [Cybenko, 1989], [Hornik, 1991]

- 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

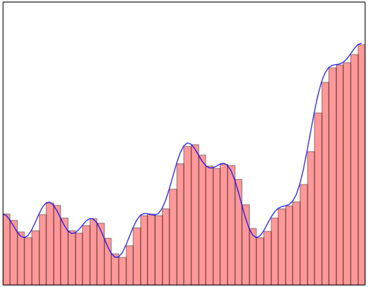
*http://neuralnetworksanddeeplearning.com/chap4.html



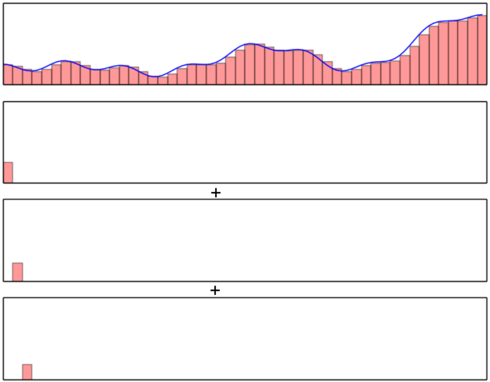
- We are interested in knowing whether a network of neurons can be used to represent an arbitrary function (like the one shown in the figure)



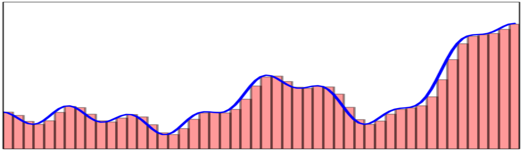
- We are interested in knowing whether a network of neurons can be used to represent an arbitrary function (like the one shown in the figure)
- We observe that such an arbitrary function can be approximated by several "tower" functions



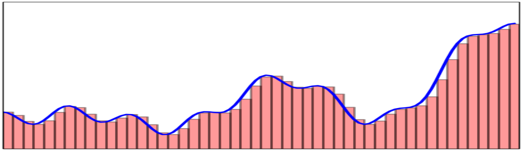
- We are interested in knowing whether a network of neurons can be used to represent an arbitrary function (like the one shown in the figure)
- We observe that such an arbitrary function can be approximated by several "tower" functions
- More the number of such "tower" functions, better the approximation



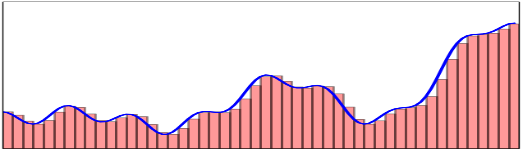
- We are interested in knowing whether a network of neurons can be used to represent an arbitrary function (like the one shown in the figure)
- We observe that such an arbitrary function can be approximated by several “tower” functions
- More the number of such “tower” functions, better the approximation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



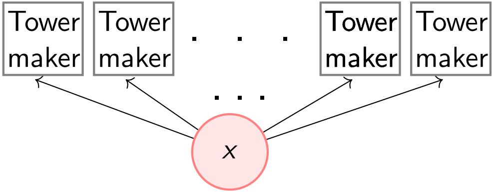
- We make a few observations

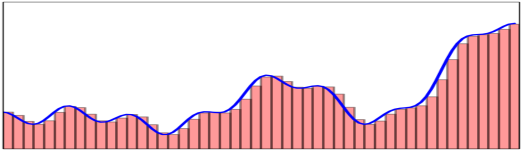


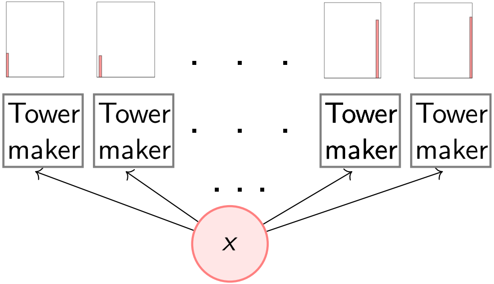
- We make a few observations
- All these "tower" functions are similar and only differ in their heights and positions on the x-axis



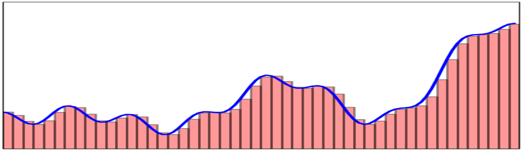
- We make a few observations
- All these "tower" functions are similar and only differ in their heights and positions on the x-axis
- Suppose there is a black box which takes the original input (x) and constructs these tower functions

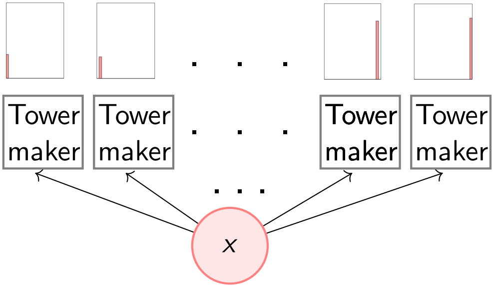




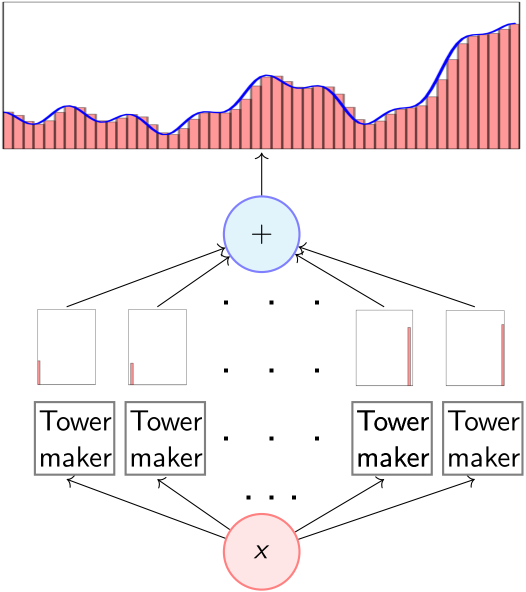


- We make a few observations
- All these "tower" functions are similar and only differ in their heights and positions on the x-axis
- Suppose there is a black box which takes the original input (x) and constructs these tower functions

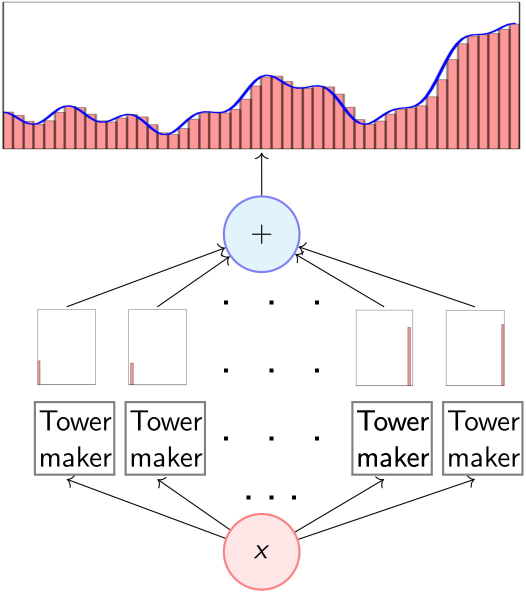




- We make a few observations
- All these "tower" functions are similar and only differ in their heights and positions on the x-axis
- Suppose there is a black box which takes the original input (x) and constructs these tower functions
- We can then have a simple network which can just add them up to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

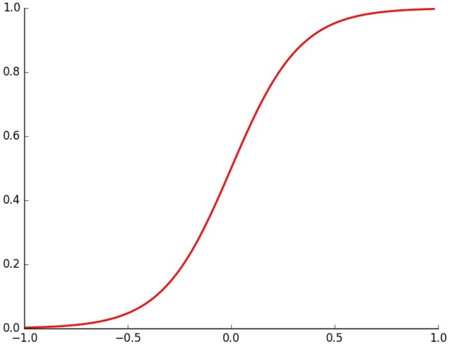


- We make a few observations
- All these "tower" functions are similar and only differ in their heights and positions on the x-axis
- Suppose there is a black box which takes the original input (x) and constructs these tower functions
- We can then have a simple network which can just add them up to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

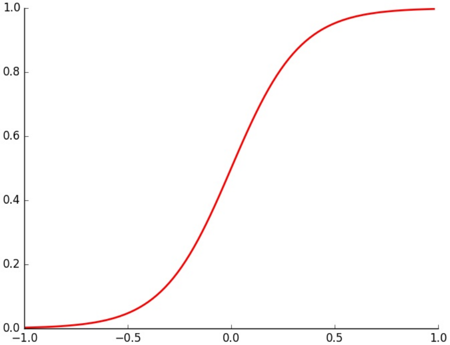


- We make a few observations
- All these "tower" functions are similar and only differ in their heights and positions on the x-axis
- Suppose there is a black box which takes the original input (x) and constructs these tower functions
- We can then have a simple network which can just add them up to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

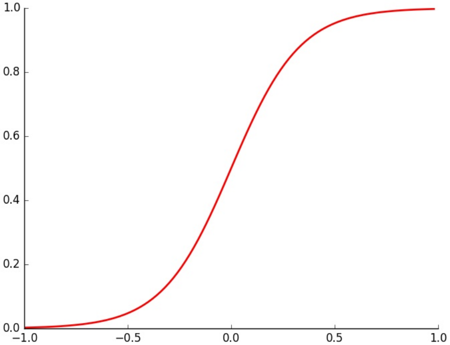
We will figure this out over the next few slides ...



- If we take the logistic function and set w to a very high value we will recover the step function

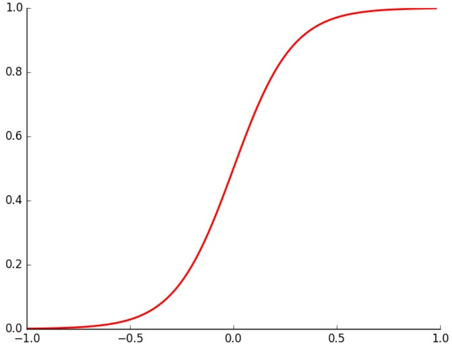


- If we take the logistic function and set w to a very high value we will recover the step function
- Let us see what happens as we change the value of w



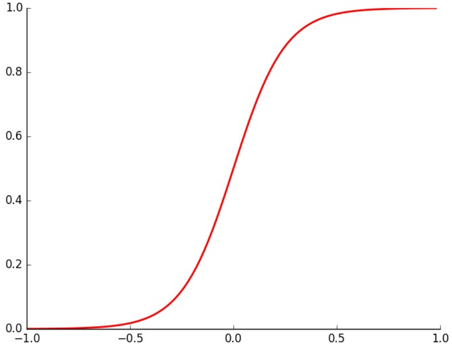
$w = 6, b = 0, 3$

- 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



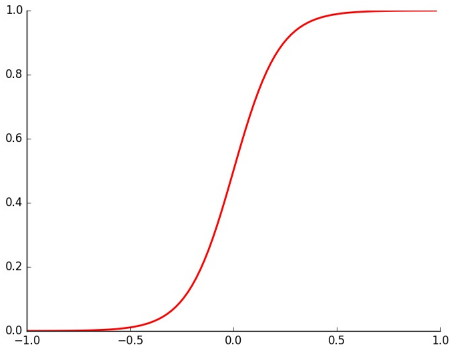
$w = 7, b = 0, 4$

- 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



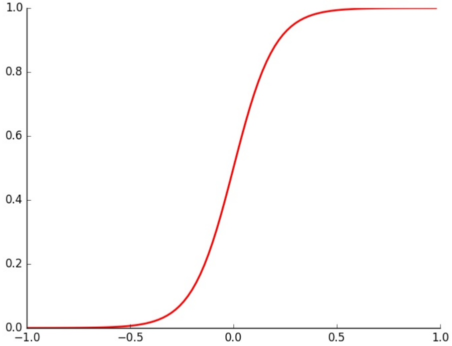
$w = 8, b = 0, 5$

- 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



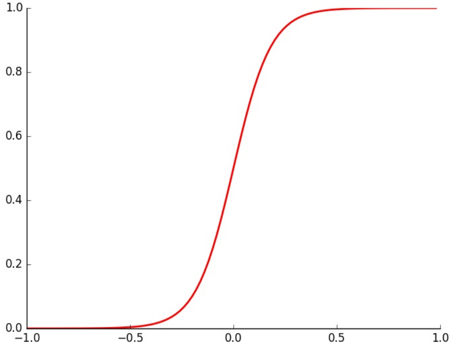
$w = 9, b = 0, 6$

- 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



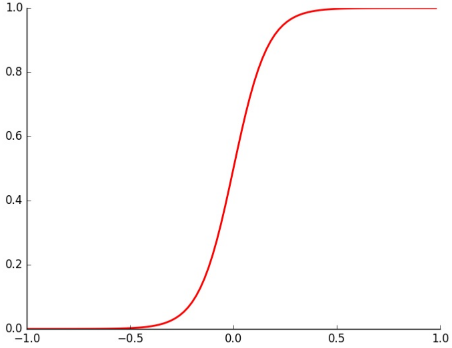
$w = 10, b = 0, 7$

- 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



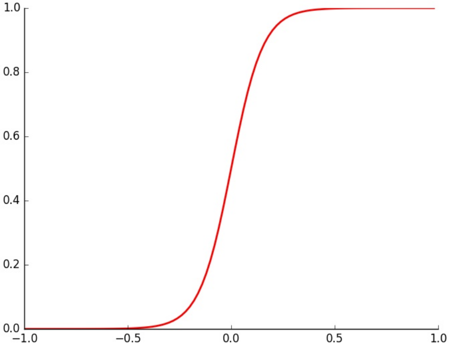
$w = 11, b = 0, 8$

- 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



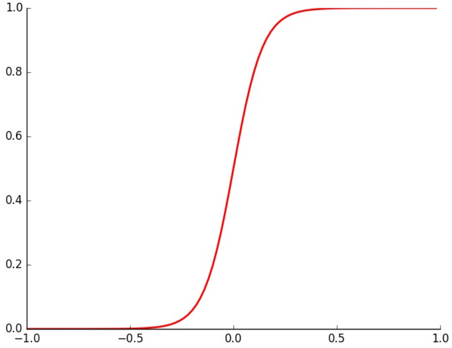
$w = 12, b = 0,9$

- 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



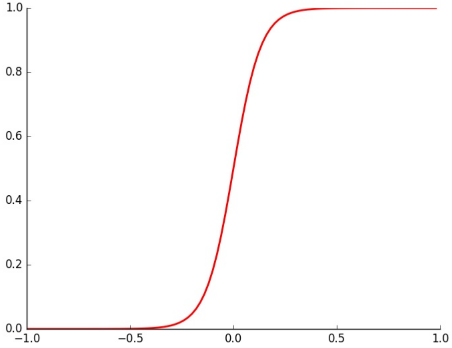
$w = 13, b = 0, 10$

- 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



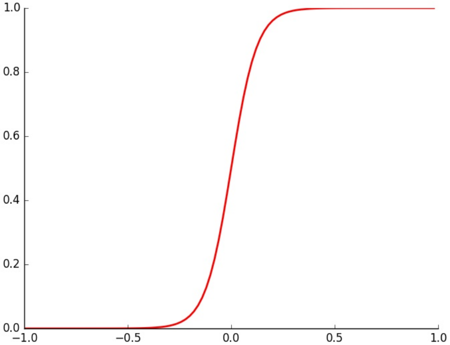
$w = 14, b = 0, 11$

- 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



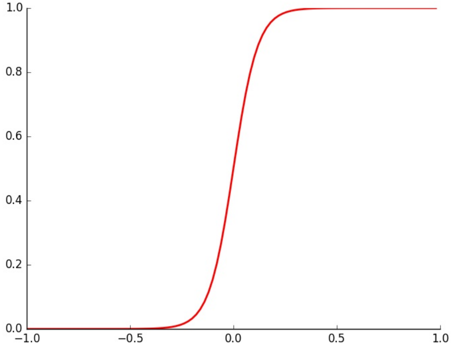
$w = 15, b = 0, 12$

- 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



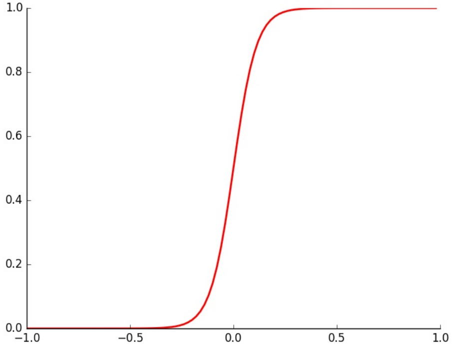
$w = 16, b = 0, 13$

- 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



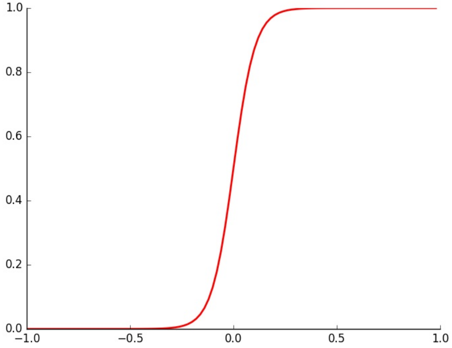
$w = 17, b = 0, 14$

- 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



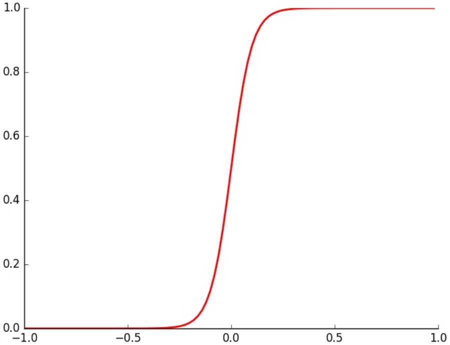
$w = 18, b = 0, 15$

- 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



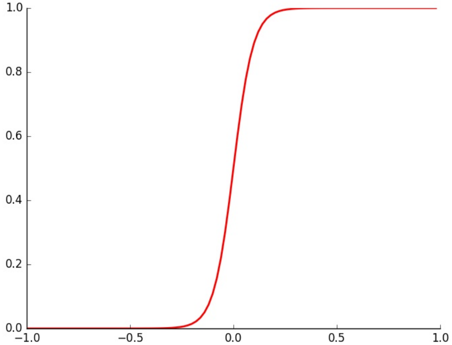
$w = 19, b = 0,16$

- 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



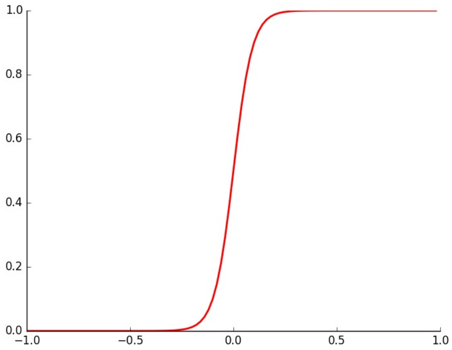
$w = 20, b = 0, 17$

- 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



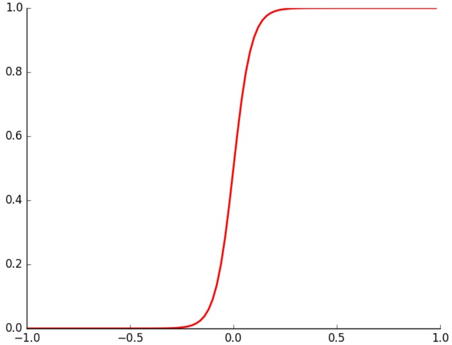
$w = 21, b = 0, 18$

- 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



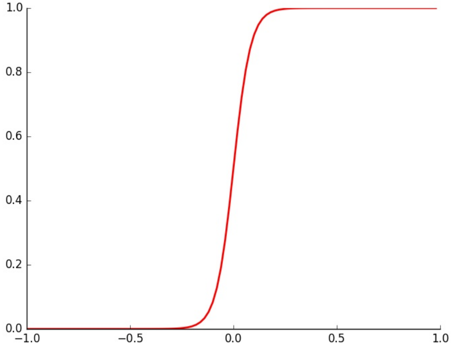
$$w = 22, b = 0, 19$$

- 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



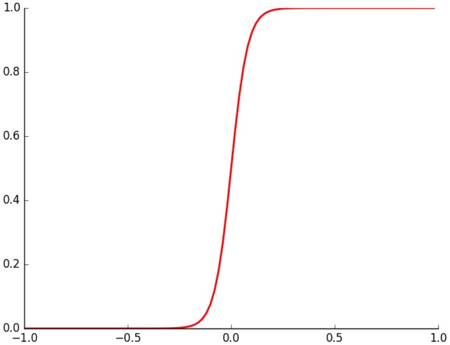
$w = 23, b = 0, 20$

- 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



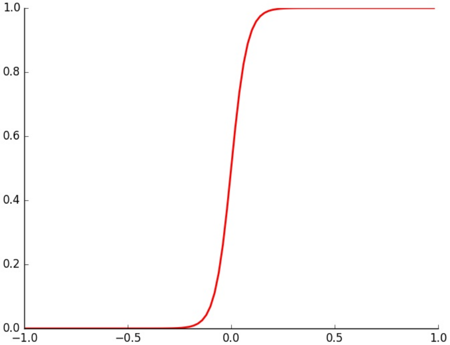
$w = 24, b = 0, 21$

- 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



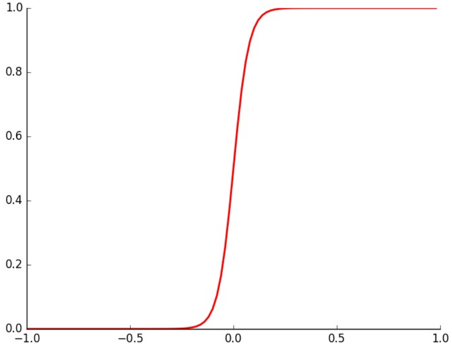
$w = 25, b = 0, 22$

- 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



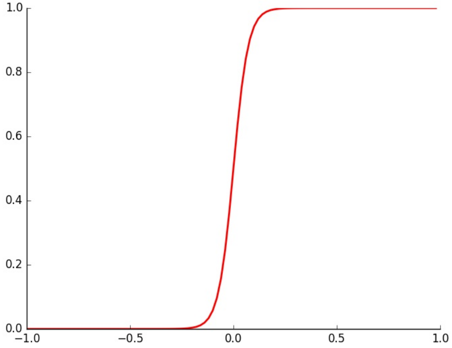
$w = 26, b = 0,23$

- 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



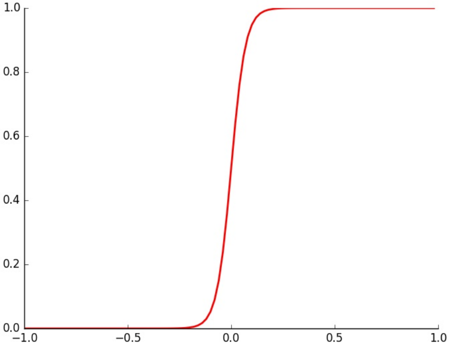
$w = 27, b = 0, 24$

- 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



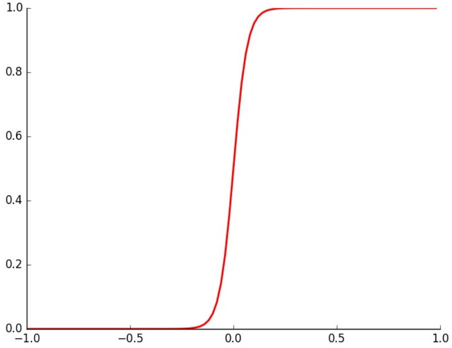
$w = 28, b = 0,25$

- 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



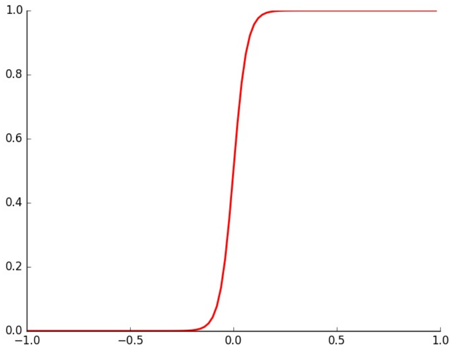
$w = 29, b = 0, 26$

- 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



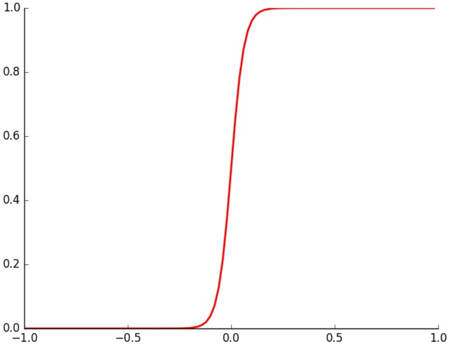
$w = 30, b = 0, 27$

- 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



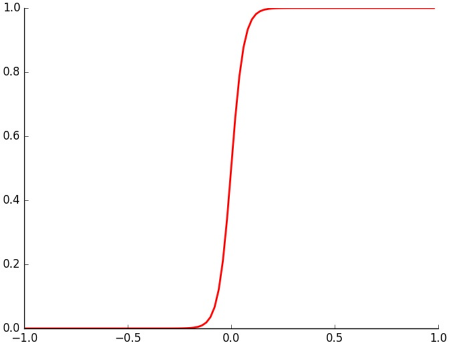
$w = 31, b = 0, 28$

- 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



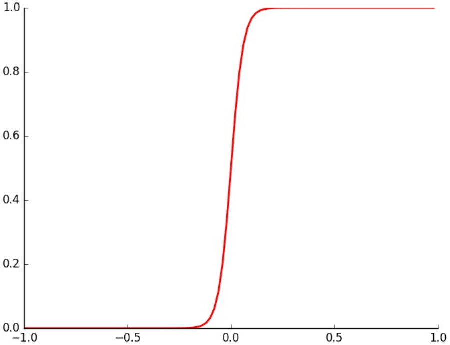
$w = 32, b = 0, 29$

- 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



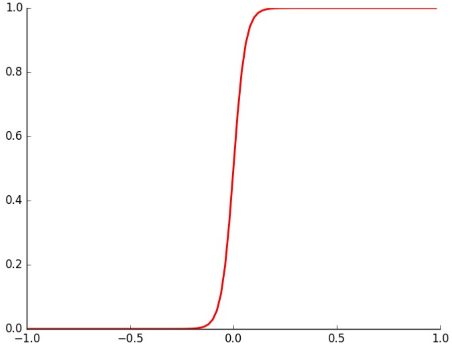
$w = 33, b = 0, 30$

- 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



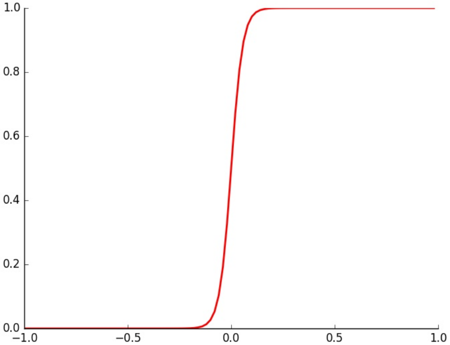
$w = 34, b = 0, 31$

- 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



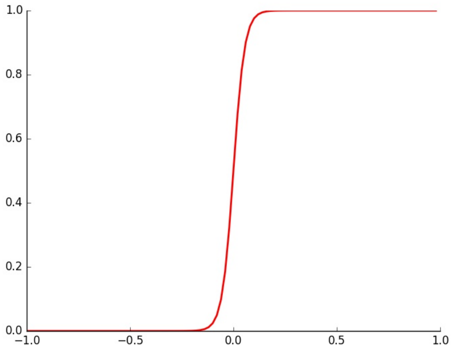
$w = 35, b = 0, 32$

- 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



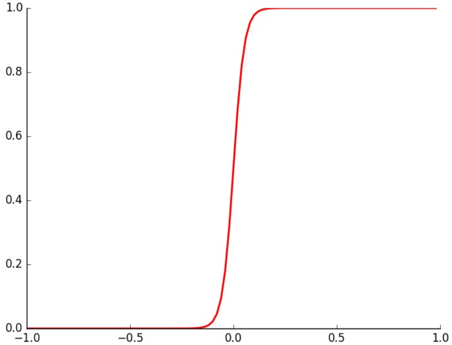
$w = 36, b = 0, 33$

- 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



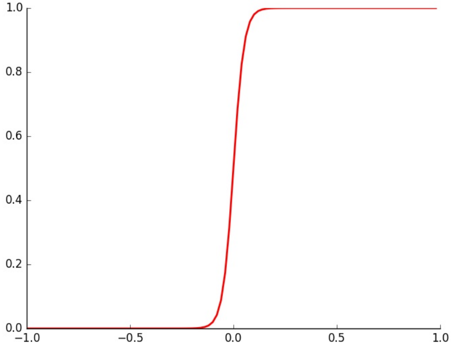
$w = 37, b = 0, 34$

- 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



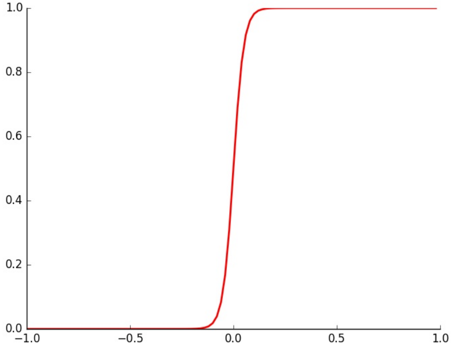
$w = 38, b = 0, 35$

- 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



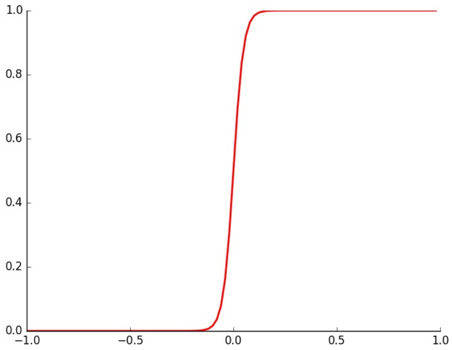
$w = 39, b = 0, 36$

- 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



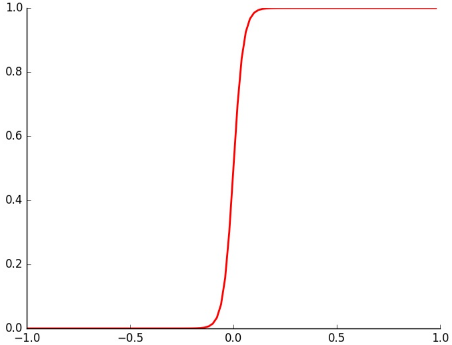
$w = 40, b = 0, 37$

- 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



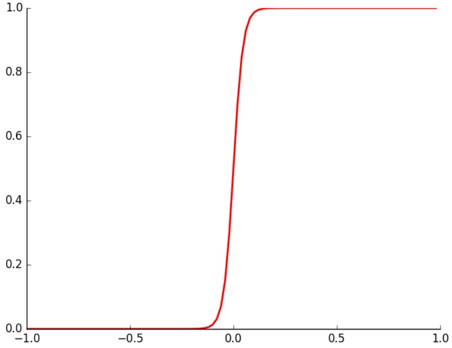
$w = 41, b = 0, 38$

- 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



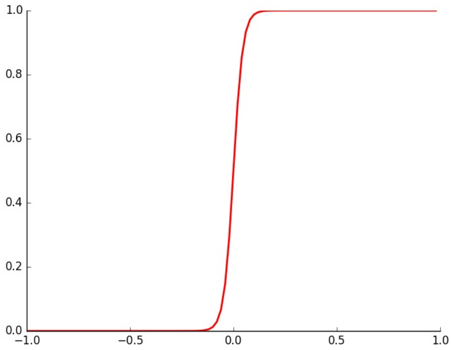
$w = 42, b = 0, 39$

- 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



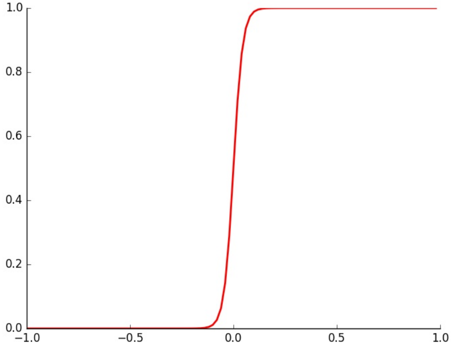
$w = 43, b = 0, 40$

- 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



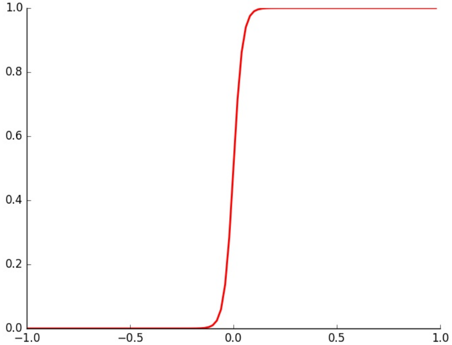
$w = 44, b = 0, 41$

- 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



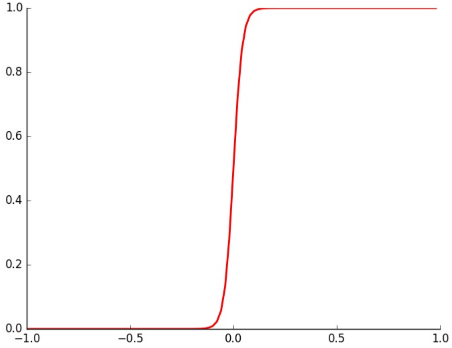
$w = 45, b = 0,42$

- 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



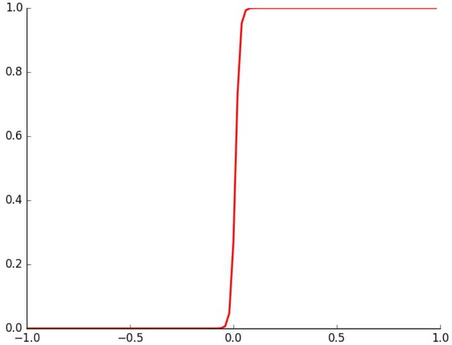
$w = 46, b = 0, 43$

- 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



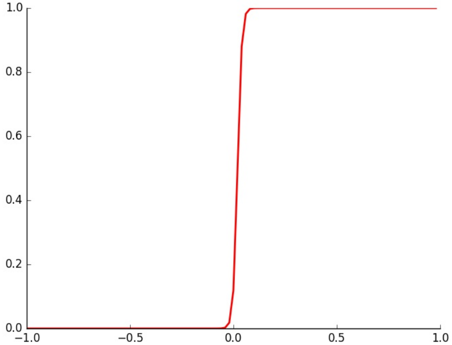
$w = 47, b = 0, 44$

- 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



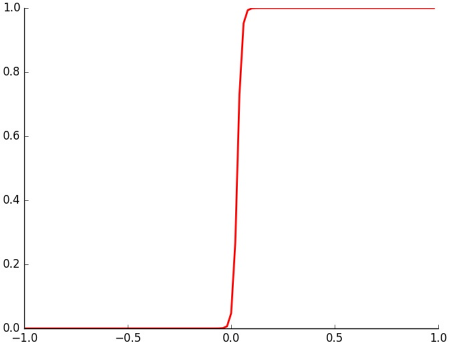
$w = 50, b = 1$

- 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



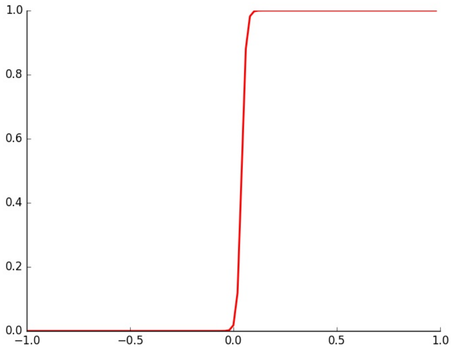
$w = 50, b = 2$

- 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



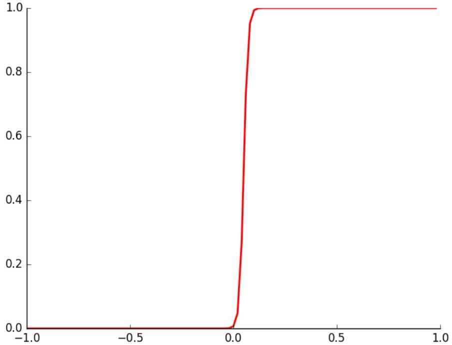
$w = 50, b = 3$

- 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



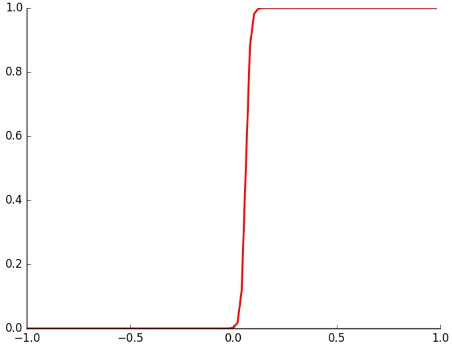
$w = 50, b = 4$

- 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



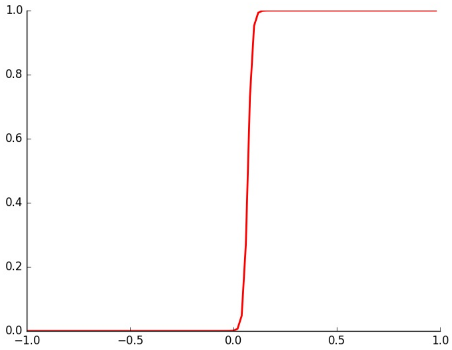
$w = 50, b = 5$

- 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



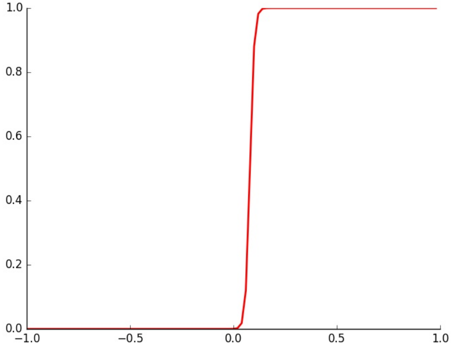
$w = 50, b = 6$

- 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



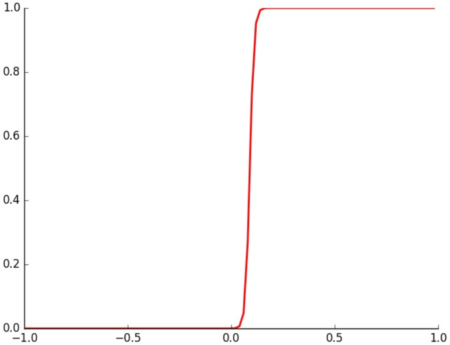
$w = 50, b = 7$

- 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



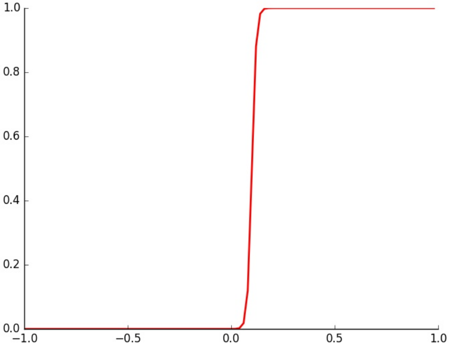
$w = 50, b = 8$

- 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



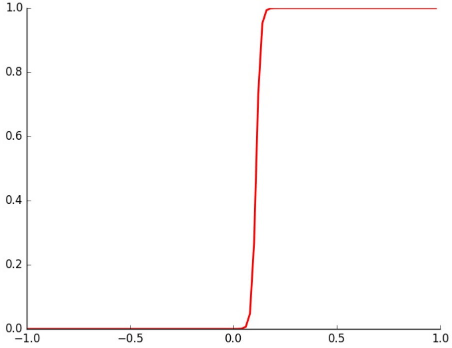
$w = 50, b = 9$

- 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



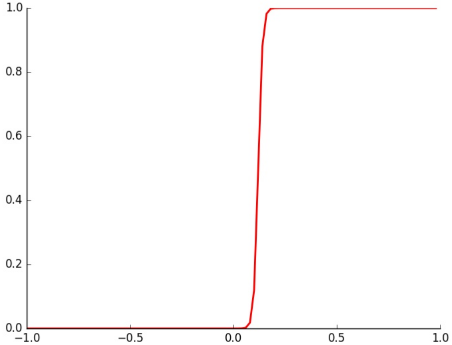
$w = 50, b = 10$

- 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



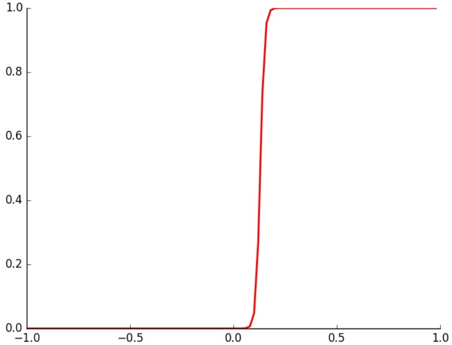
$w = 50, b = 11$

- 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



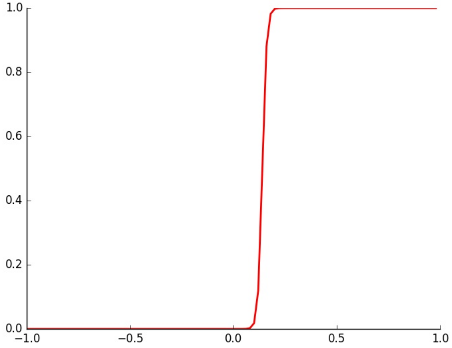
$w = 50, b = 12$

- 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



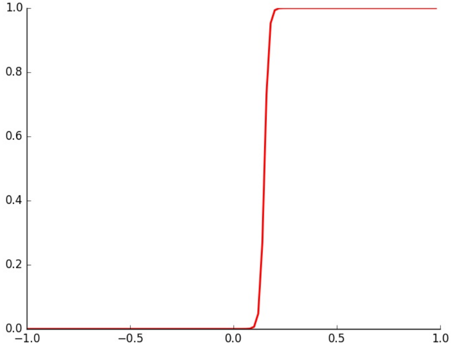
$w = 50, b = 13$

- 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



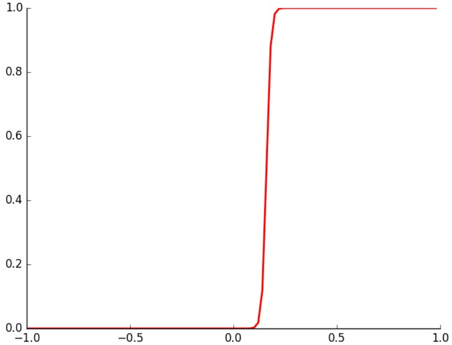
$w = 50, b = 14$

- 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



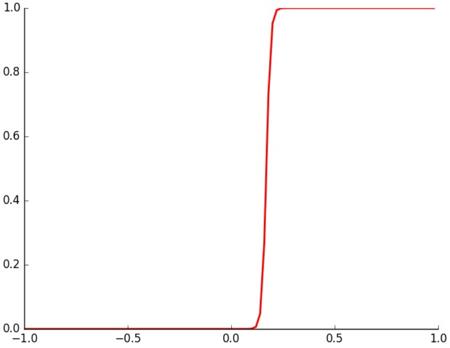
$w = 50, b = 15$

- 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



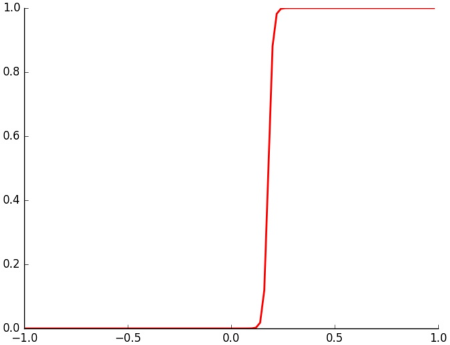
$w = 50, b = 16$

- 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



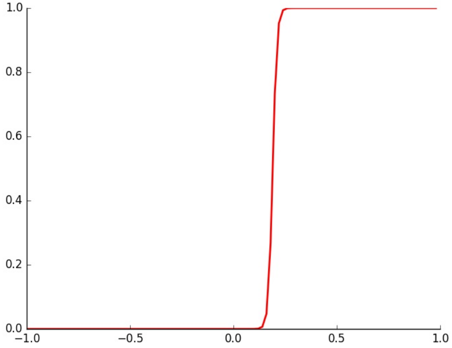
$w = 50, b = 17$

- 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



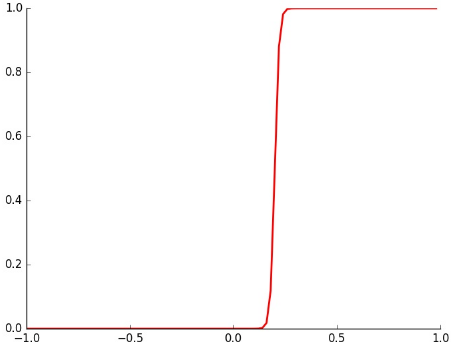
$w = 50, b = 18$

- 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



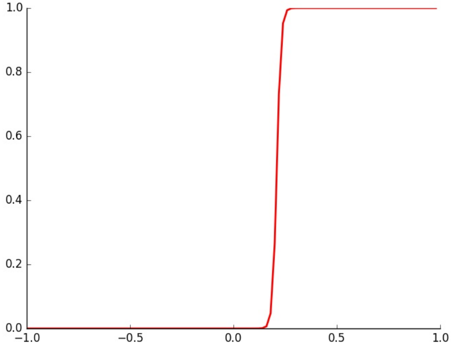
$w = 50, b = 19$

- 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



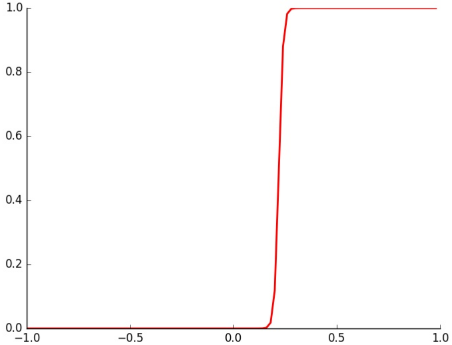
$w = 50, b = 20$

- 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



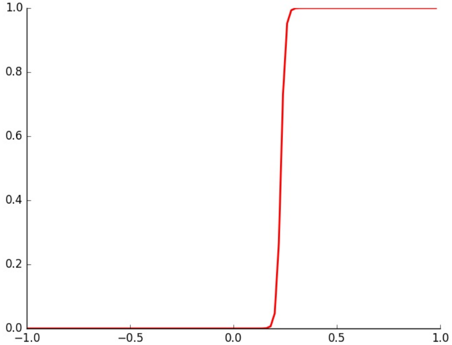
$w = 50, b = 21$

- 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



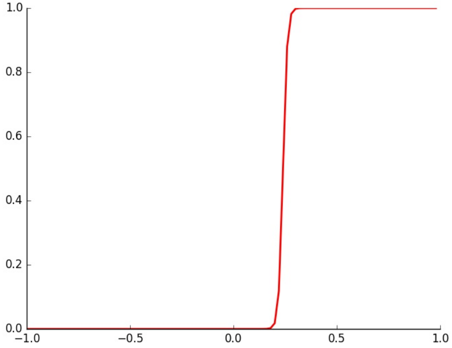
$w = 50, b = 22$

- 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



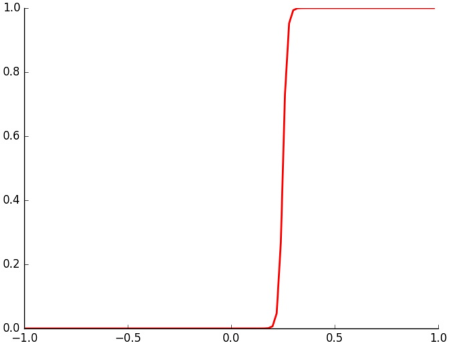
$w = 50, b = 23$

- 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



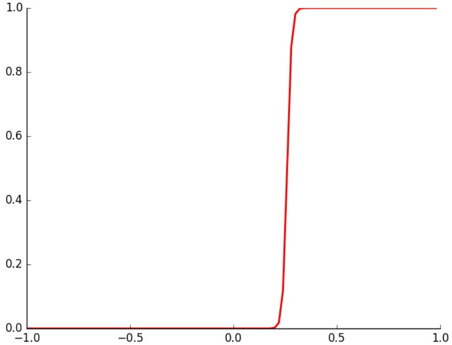
$w = 50, b = 24$

- 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



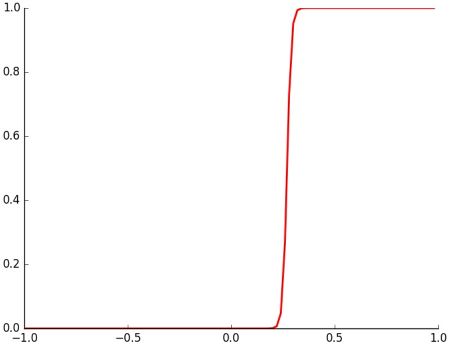
$w = 50, b = 25$

- 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



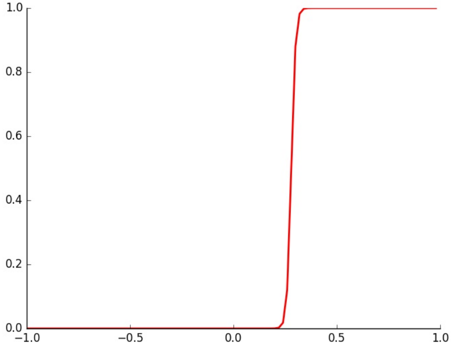
$w = 50, b = 26$

- 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



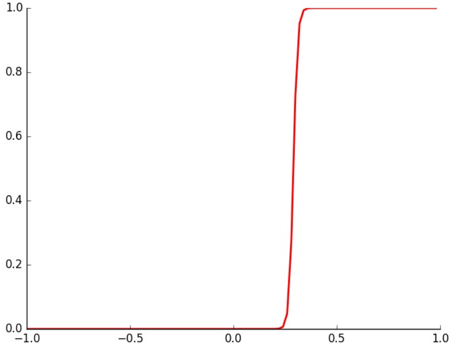
$w = 50, b = 27$

- 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



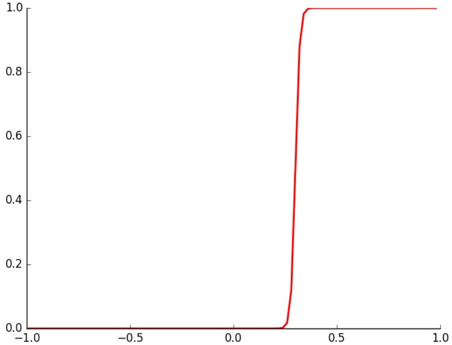
$w = 50, b = 28$

- 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



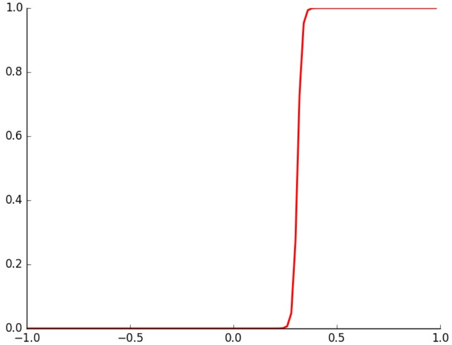
$w = 50, b = 29$

- 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



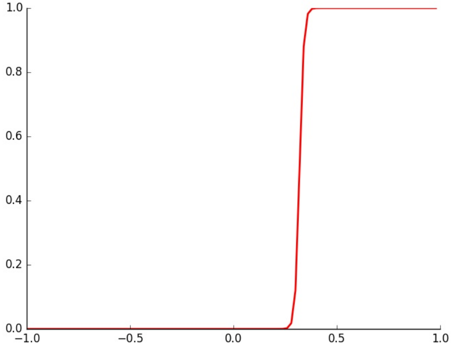
$w = 50, b = 30$

- 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



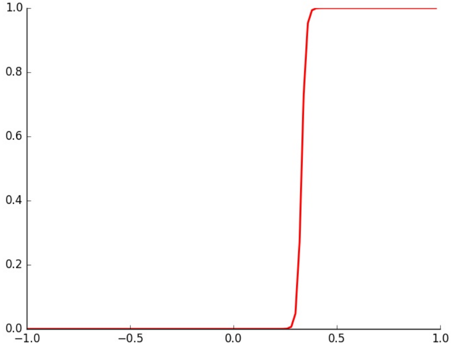
$w = 50, b = 31$

- 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



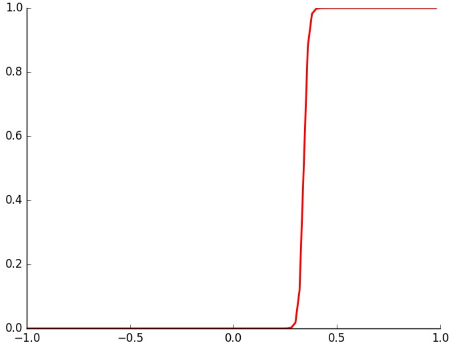
$w = 50, b = 32$

- 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



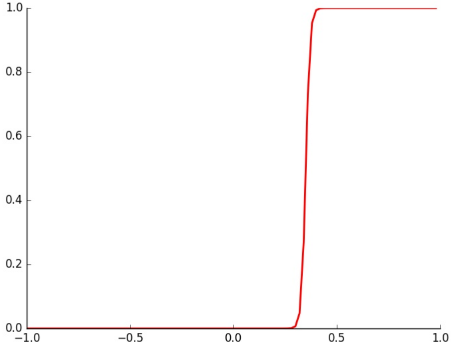
$w = 50, b = 33$

- 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



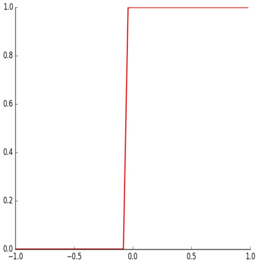
$w = 50, b = 34$

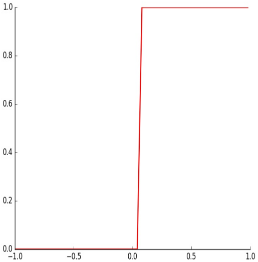
- If we take the logistic function and set w to a very high value we will recover the step function
- Let us see what happens as we change the value of w
- Further we can adjust the value of b to control the position on the x-axis at which the function transitions from 0 to 1



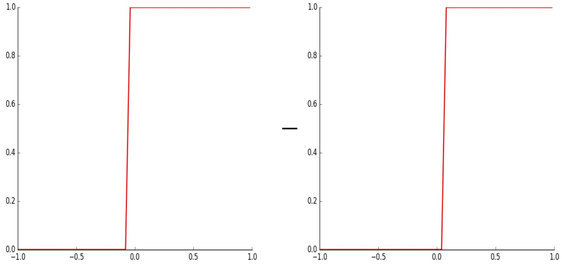
$w = 50, b = 35$

- 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

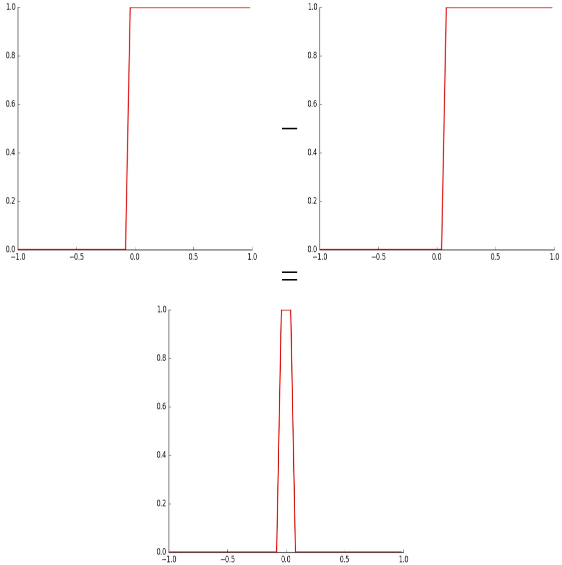




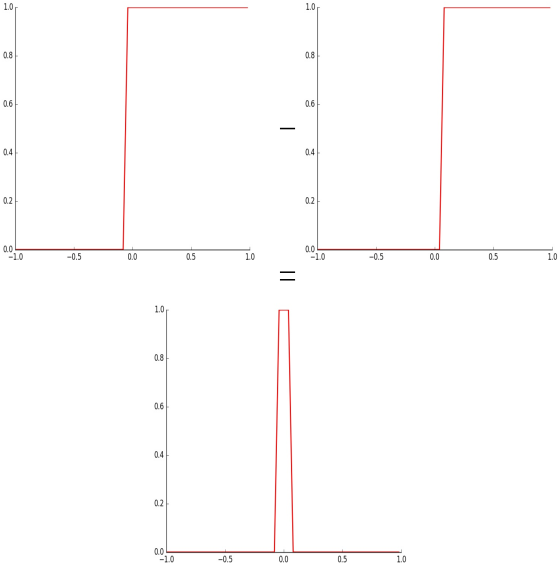
- Now lets see what we get by taking two such sigmoid functions (with different *b*s) and subtracting one from the other



- Now lets see what we get by taking two such sigmoid functions (with different *b*s) and subtracting one from the other

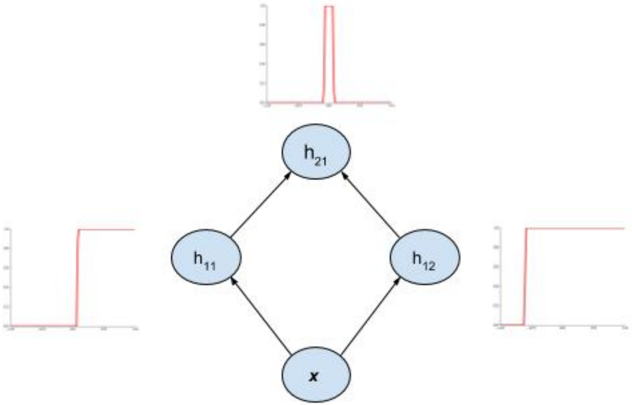


- Now lets see what we get by taking two such sigmoid functions (with different *b*s) and subtracting one from the other

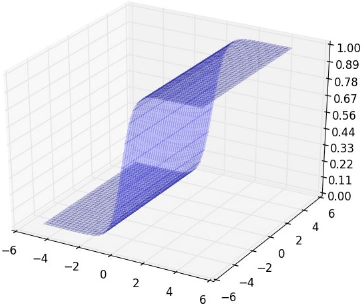


- Now lets see what we get by taking two such sigmoid functions (with different bs) and subtracting one from the other
- Voila! We have our tower function !!

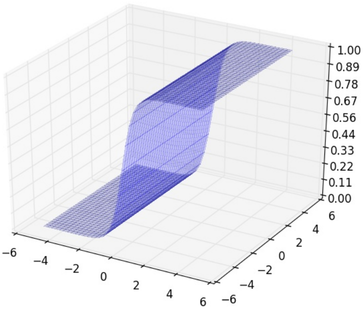
- Can we come up with a neural network to represent this operation of subtracting one sigmoid function from another ?


h21
h11
h12
x

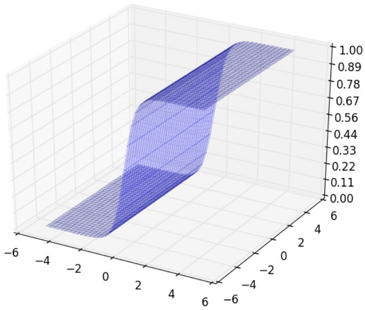
What if we have more than 1 input ?



- This is what a 3-dimensional sigmoid looks like

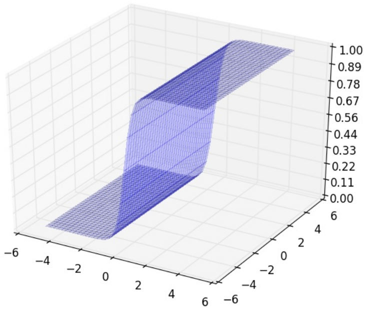


- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower



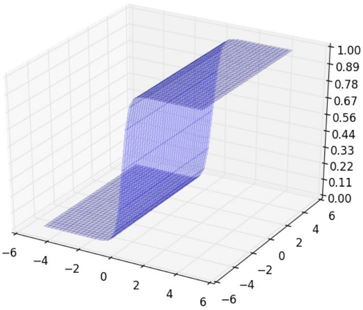
$$w_1 = 2, w_2 = 0, b = 0$$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



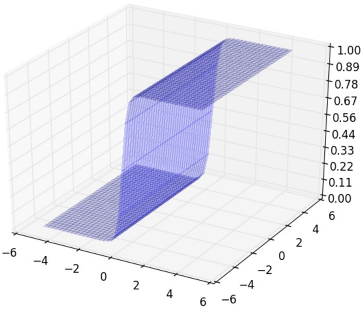
$w_1 = 3, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



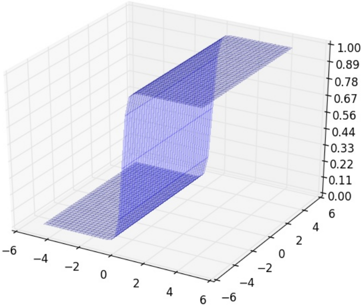
$w_1 = 4, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



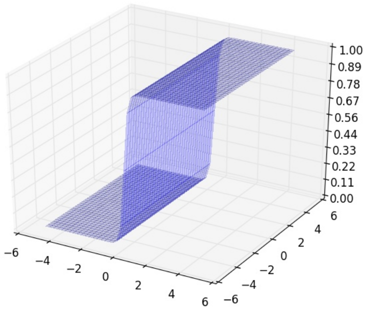
$w_1 = 5, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



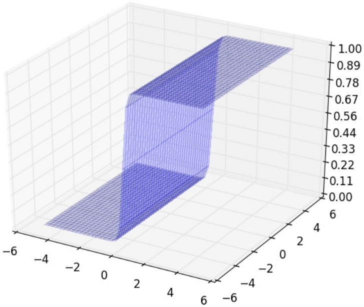
$w_1 = 6, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



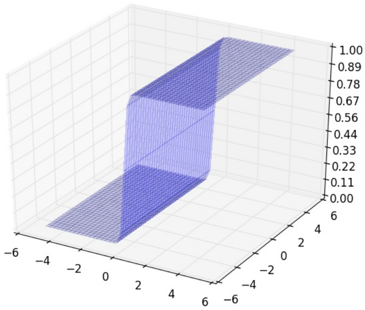
$w_1 = 7, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



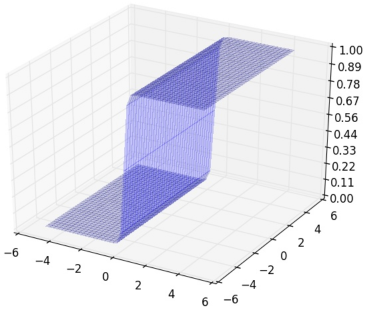
$w_1 = 8, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



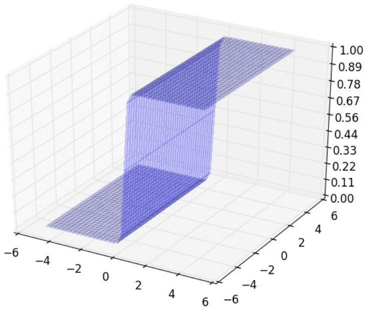
$w_1 = 9, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



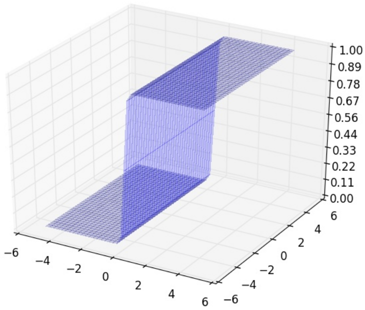
$w_1 = 10, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



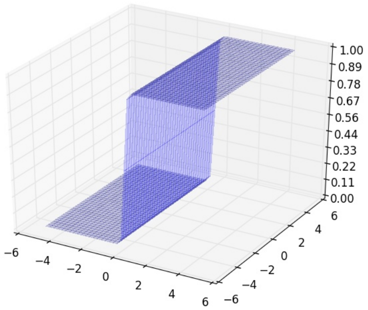
$w_1 = 11, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



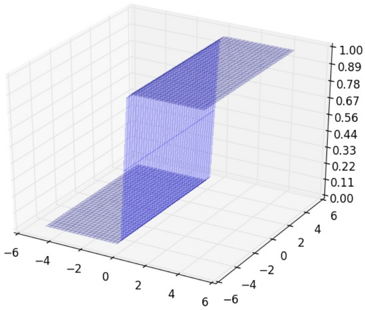
$w_1 = 12, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



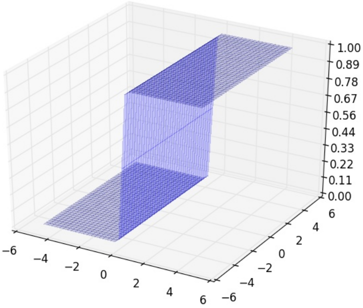
$w_1 = 13, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



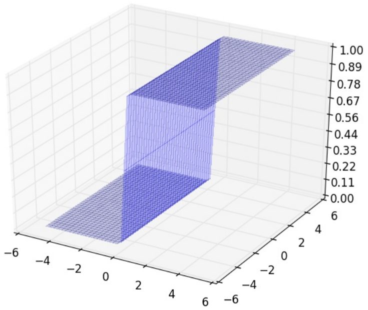
$w_1 = 14, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



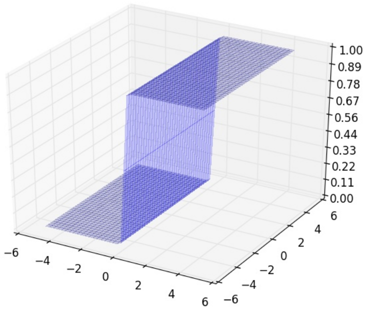
$w_1 = 15, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



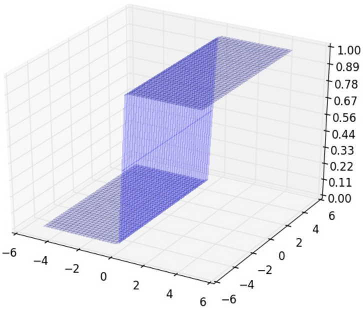
$w_1 = 16, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



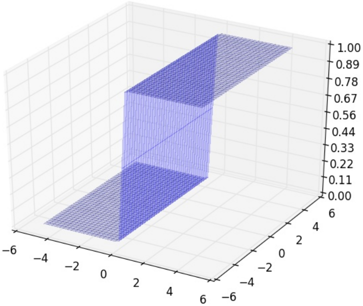
$w_1 = 17, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



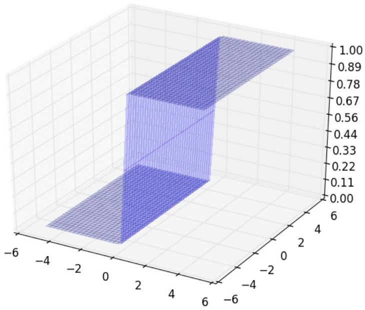
$w_1 = 18, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



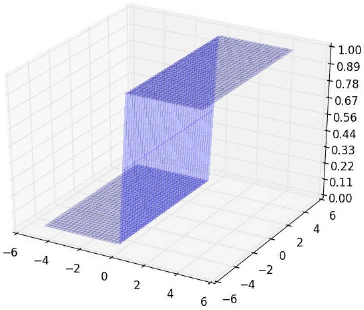
$w_1 = 19, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



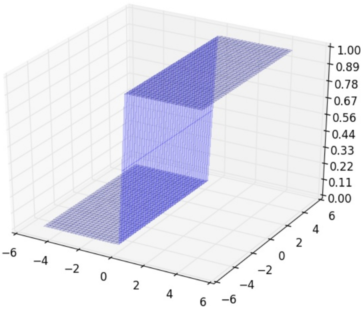
$w_1 = 20, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



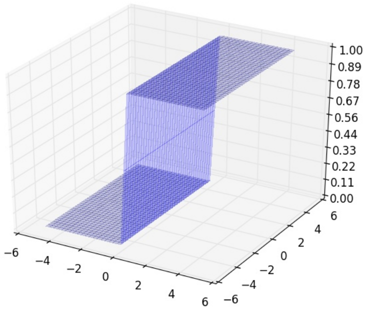
$w_1 = 21, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



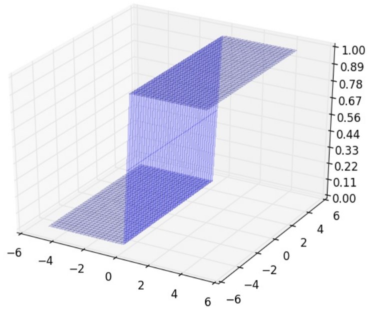
$w_1 = 22, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function



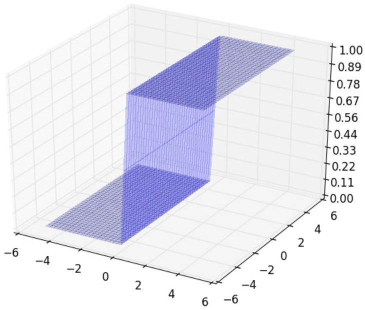
$w_1 = 23, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function

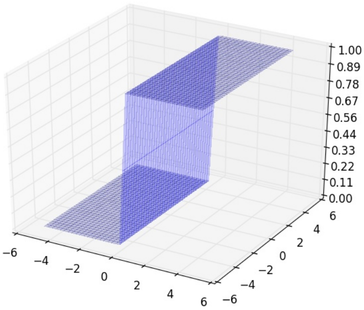


$$w_1 = 24, w_2 = 0, b = 0$$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function

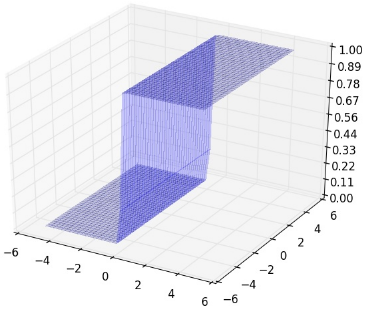


- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function
- What would happen if we change b ?



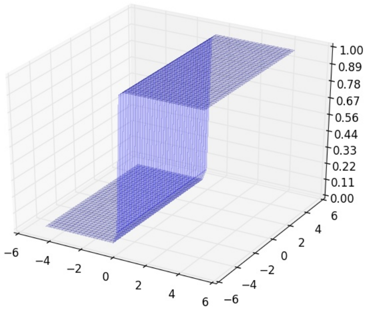
$w_1 = 25, w_2 = 0, b = 0$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- 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 ?



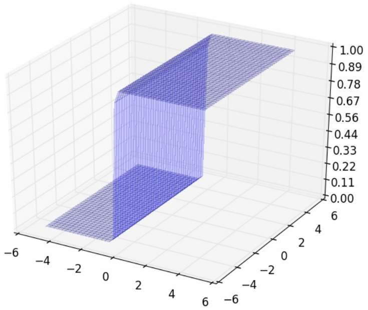
$w_1 = 25, w_2 = 0, b = 5$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- 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 ?



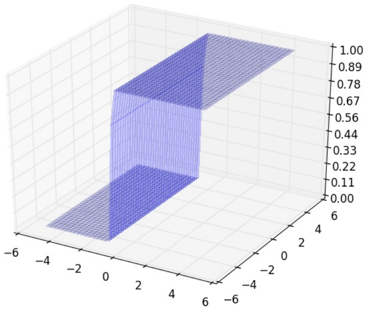
$w_1 = 25, w_2 = 0, b = 10$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- 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 ?



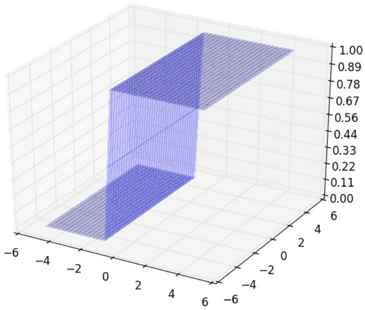
$$w_1 = 25, w_2 = 0, b = 15$$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- 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 ?



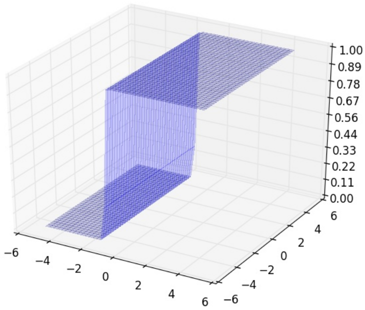
$w_1 = 25, w_2 = 0, b = 20$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- 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 ?



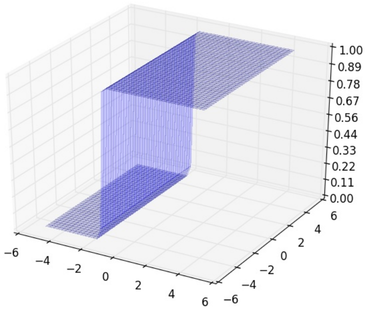
$w_1 = 25, w_2 = 0, b = 25$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- 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 ?



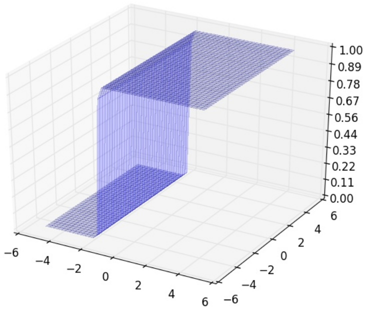
$w_1 = 25, w_2 = 0, b = 30$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- 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 ?



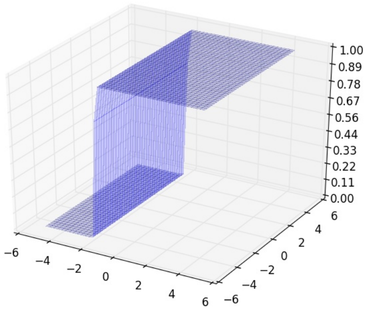
$w_1 = 25, w_2 = 0, b = 35$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- 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 ?



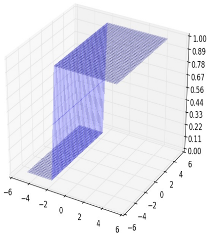
$w_1 = 25, w_2 = 0, b = 40$

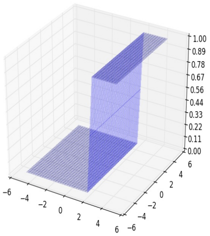
- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- First, let us set w_2 to 0 and see if we can get a two dimensional step function
- What would happen if we change b ?



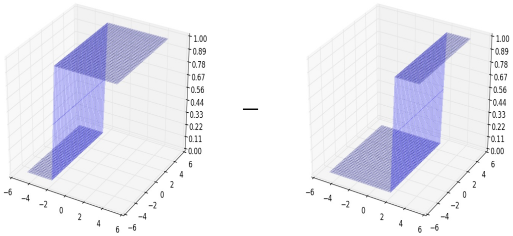
$w_1 = 25, w_2 = 0, b = 45$

- This is what a 3-dimensional sigmoid looks like
- We need to figure out how to get a 3-dimensional tower
- 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 ?

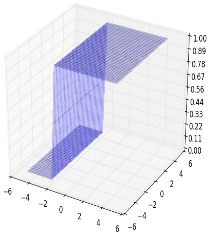




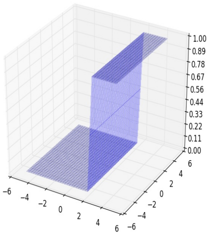
- What if we take two such step functions (with different b values) and subtract one from the other



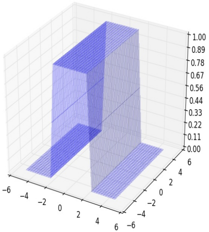
- What if we take two such step functions (with different b values) and subtract one from the other



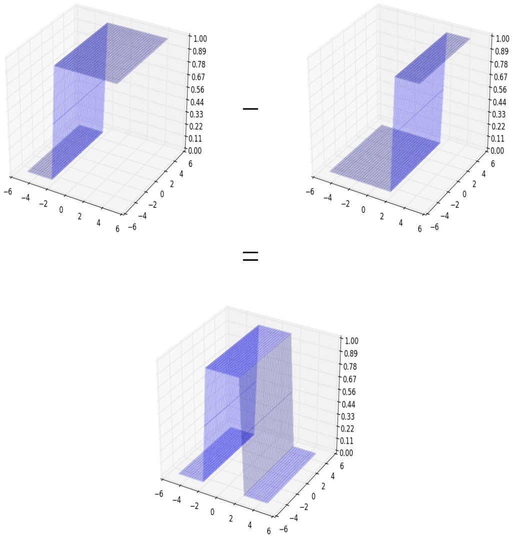
−



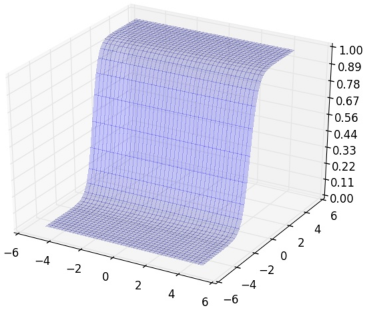
=



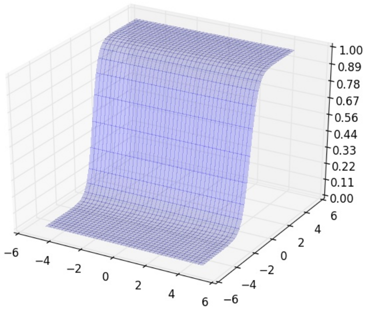
- What if we take two such step functions (with different b values) and subtract one from the other



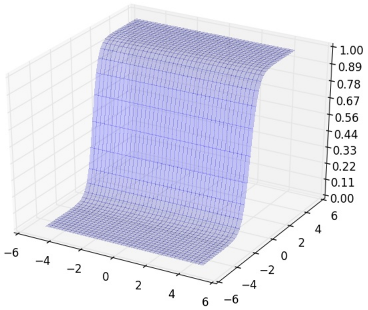
- What if we take two such step functions (with different b values) and subtract one from the other
- We still don't get a tower (or we get a tower which is open from two sides)



- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation

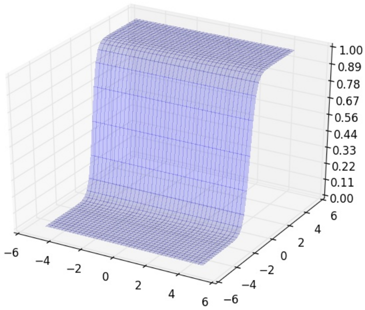


- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation



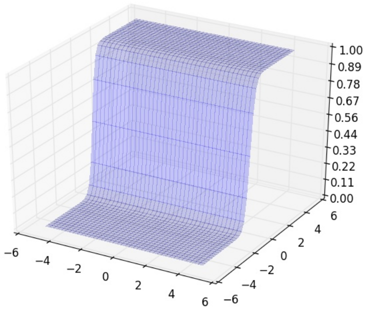
$w_1 = 0, w_2 = 2, b = 0$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation



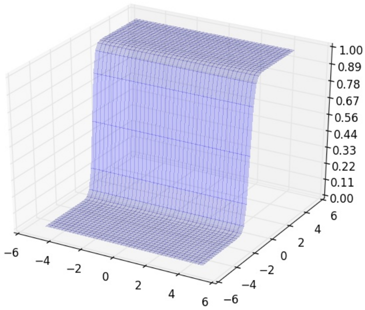
$w_1 = 0, w_2 = 3, b = 0$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation



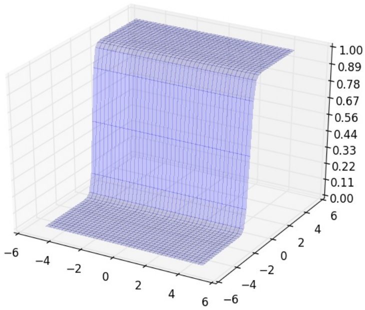
$w_1 = 0, w_2 = 4, b = 0$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation



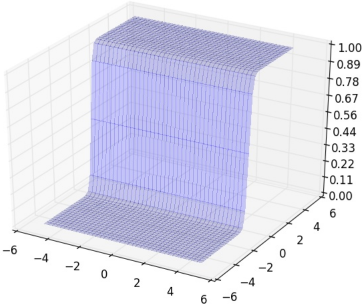
$w_1 = 0, w_2 = 5, b = 0$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation



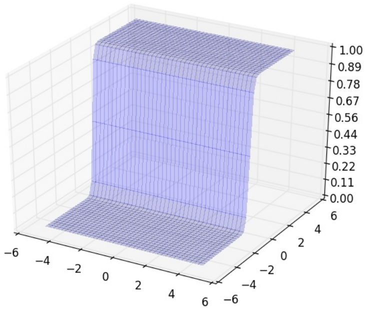
$w_1 = 0, w_2 = 6, b = 0$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation



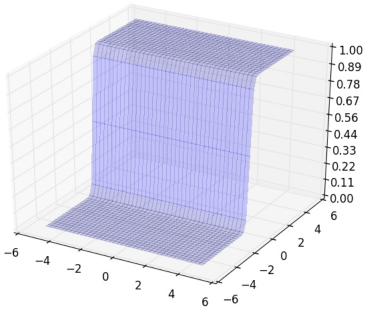
$w_1 = 0, w_2 = 7, b = 0$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation



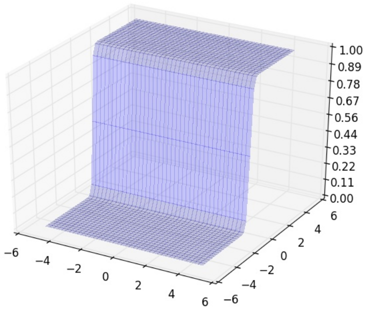
$w_1 = 0, w_2 = 8, b = 0$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation



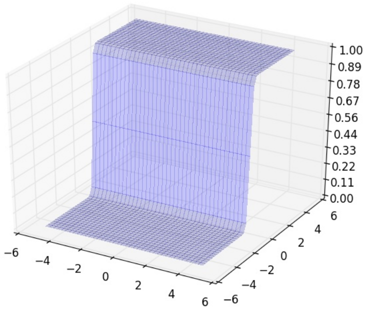
$w_1 = 0, w_2 = 9, b = 0$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation



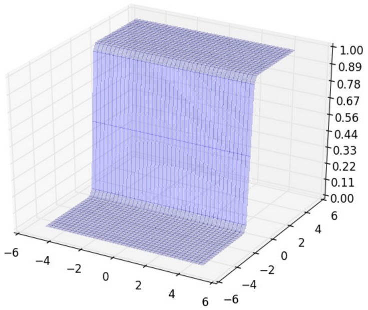
$$w_1 = 0, w_2 = 10, b = 0$$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation



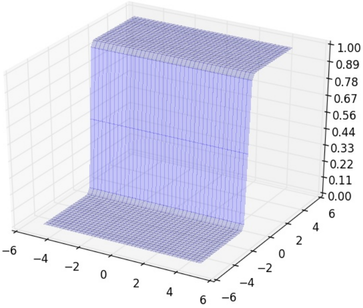
$w_1 = 0, w_2 = 11, b = 0$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation



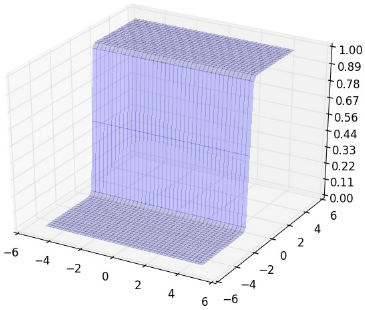
$$w_1 = 0, w_2 = 12, b = 0$$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation



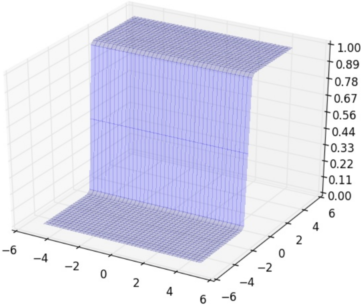
$w_1 = 0, w_2 = 13, b = 0$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation



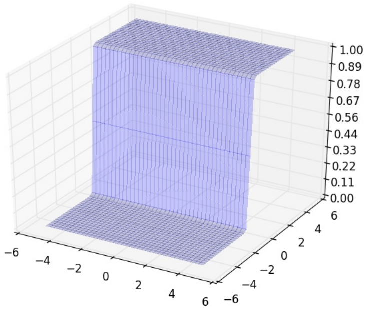
$w_1 = 0, w_2 = 14, b = 0$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation



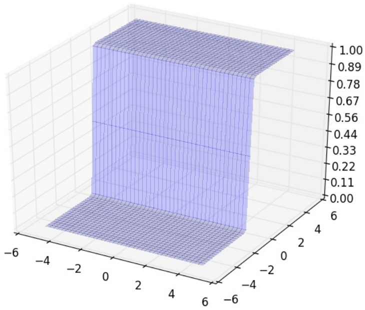
- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation

$w_1 = 0, w_2 = 15, b = 0$



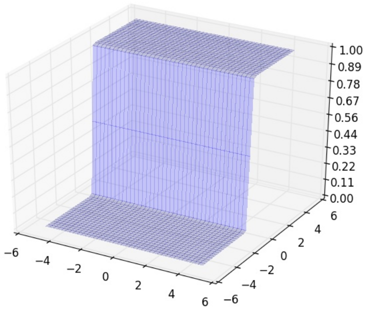
$$w_1 = 0, w_2 = 16, b = 0$$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation



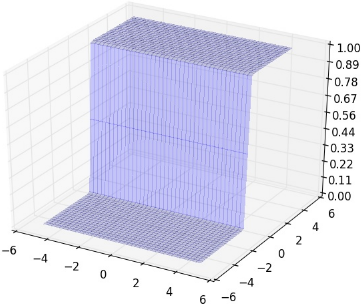
$w_1 = 0, w_2 = 17, b = 0$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation



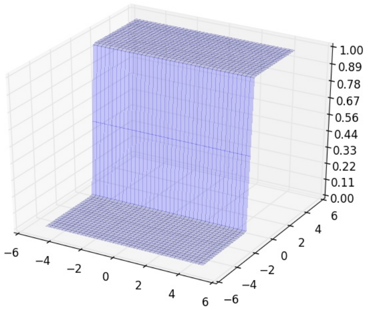
$w_1 = 0, w_2 = 18, b = 0$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation



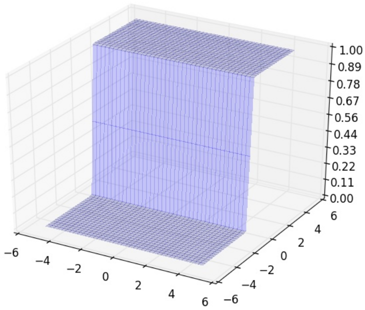
- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation

$$w_1 = 0, w_2 = 19, b = 0$$



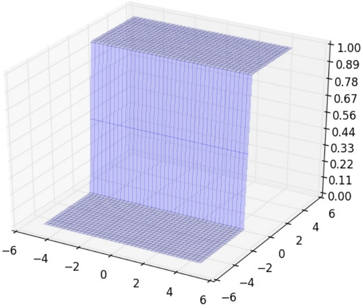
$w_1 = 0, w_2 = 20, b = 0$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation



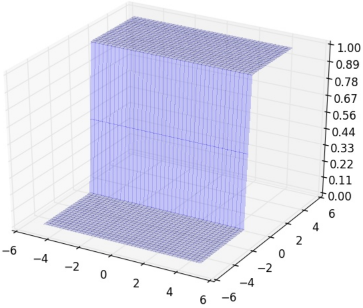
- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation

$w_1 = 0, w_2 = 21, b = 0$



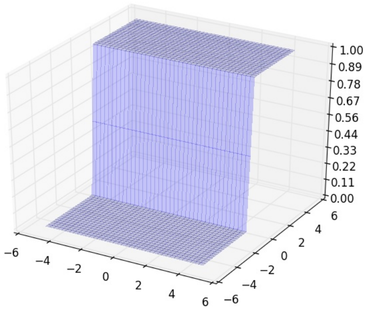
$w_1 = 0, w_2 = 22, b = 0$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation



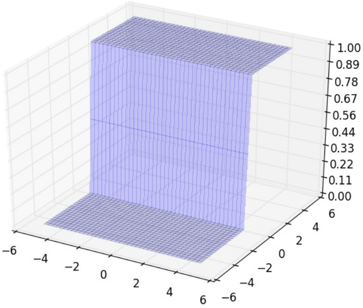
$w_1 = 0, w_2 = 23, b = 0$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation

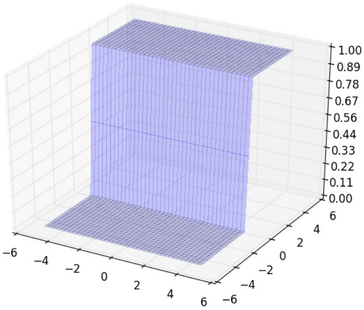


$w_1 = 0, w_2 = 24, b = 0$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation

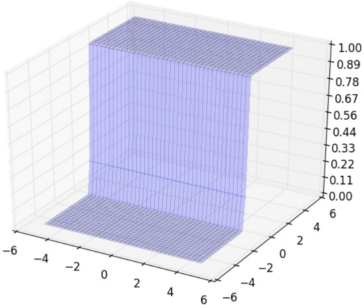


- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation
- And now we change b



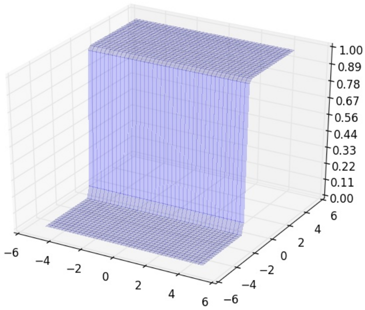
$w_1 = 0, w_2 = 25, b = 0$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation
- And now we change b



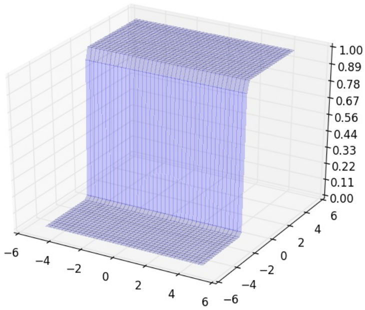
$w_1 = 0, w_2 = 25, b = 5$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation
- And now we change b



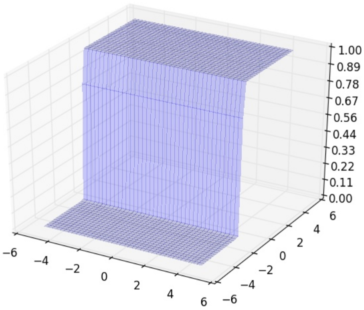
$w_1 = 0, w_2 = 25, b = 10$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation
- And now we change b



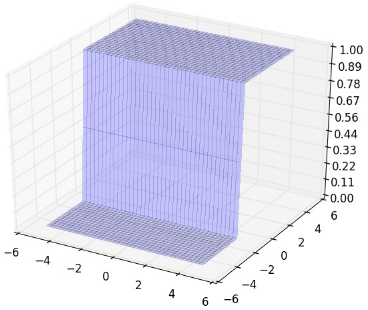
$w_1 = 0, w_2 = 25, b = 15$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation
- And now we change b



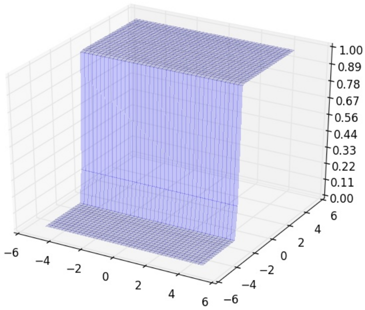
$w_1 = 0, w_2 = 25, b = 20$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation
- And now we change b



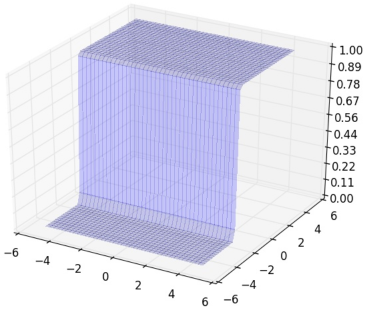
$w_1 = 0, w_2 = 25, b = 25$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation
- And now we change b



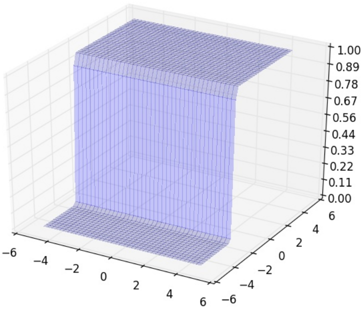
$w_1 = 0, w_2 = 25, b = 30$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation
- And now we change b



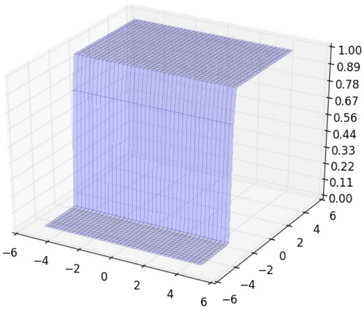
$w_1 = 0, w_2 = 25, b = 35$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation
- And now we change b



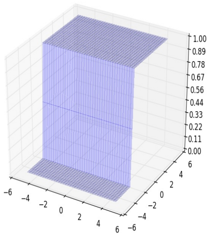
$w_1 = 0, w_2 = 25, b = 40$

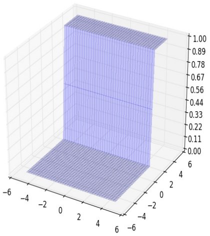
- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation
- And now we change b



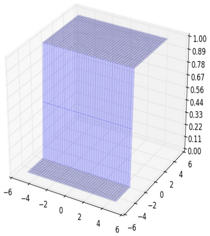
$w_1 = 0, w_2 = 25, b = 45$

- Now let us set w_1 to 0 and adjust w_2 to get a 3-dimensional step function with a different orientation
- And now we change b

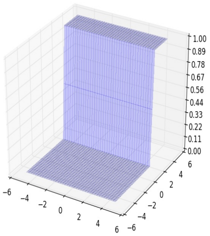




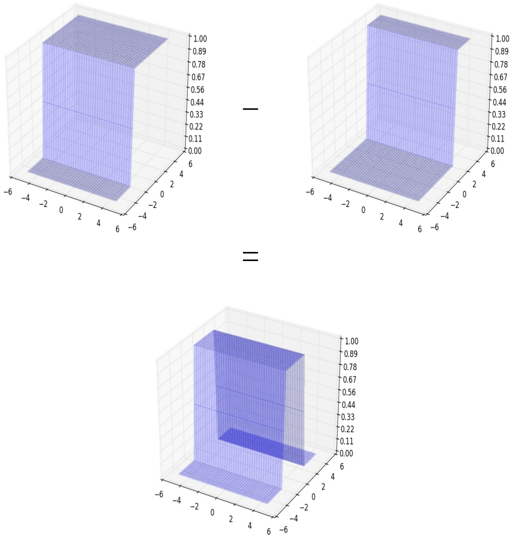
- Again, what if we take two such step functions (with different b values) and subtract one from the other



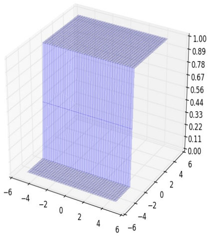
−



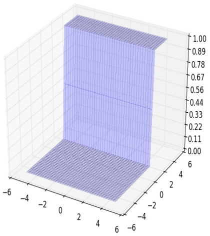
- Again, what if we take two such step functions (with different b values) and subtract one from the other



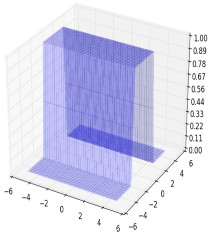
- Again, what if we take two such step functions (with different b values) and subtract one from the other



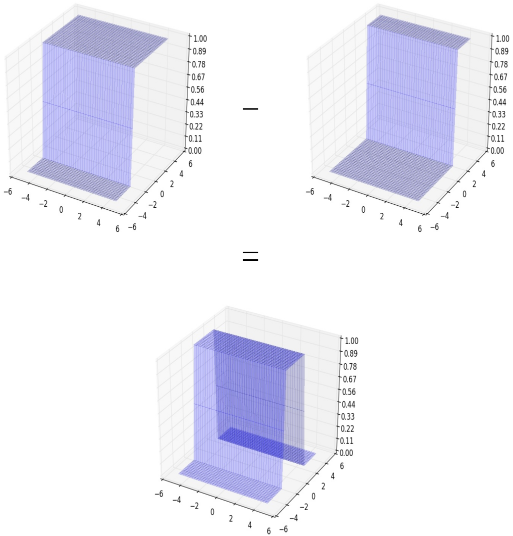
$-$



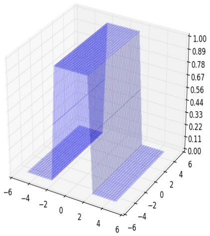
$=$

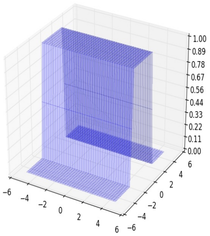


- Again, what if we take two such step functions (with different b values) and subtract one from the other
- We still don't get a tower (or we get a tower which is open from two sides)

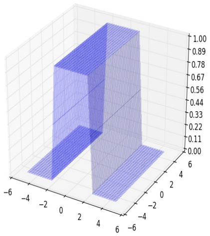


- Again, what if we take two such step functions (with different b values) and subtract one from the other
- We still don't get a tower (or we get a tower which is open from two sides)
- Notice that this open tower has a different orientation from the previous one

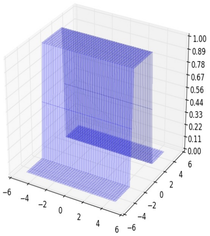




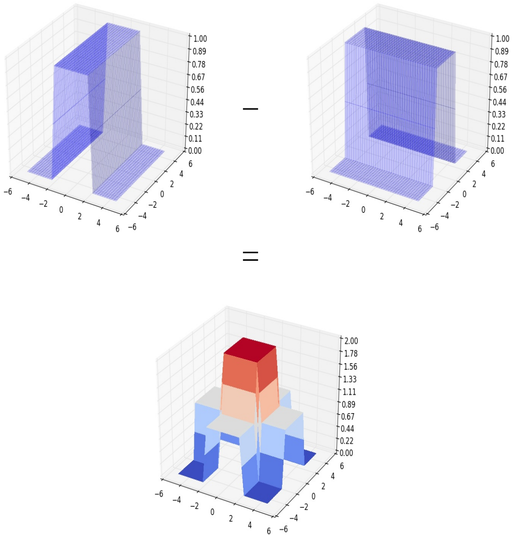
- Now what will we get by adding two such open towers ?



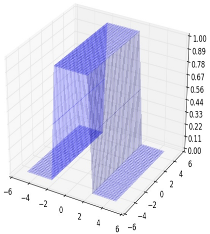
−



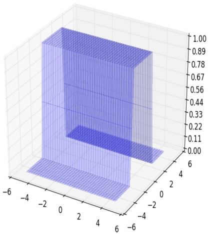
- Now what will we get by adding two such open towers ?



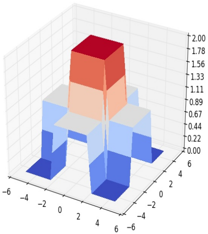
- Now what will we get by adding two such open towers ?



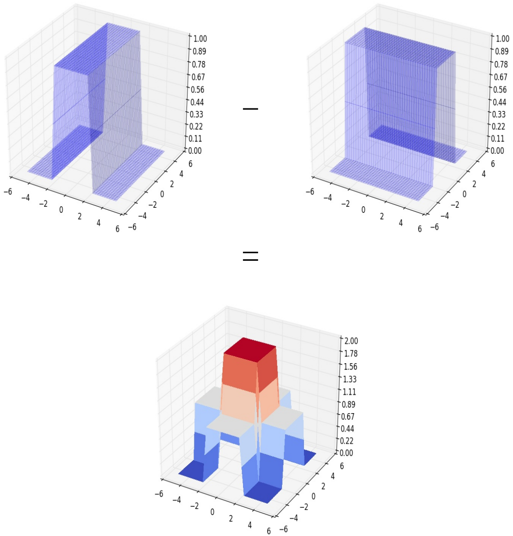
−



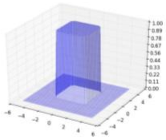
=

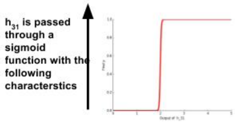


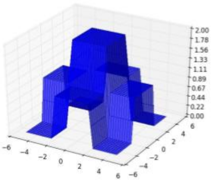
- Now what will we get by adding two such open towers ?
- We get a tower standing on an elevated base



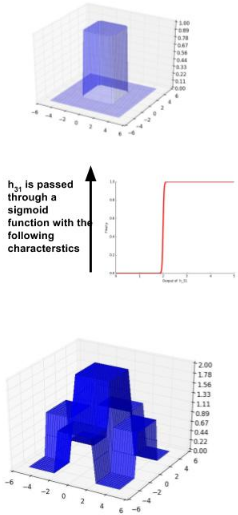
- Now what will we get by adding two such open towers ?
- We get a tower standing on an elevated base
- We can now pass this output through another sigmoid neuron to get the desired tower !



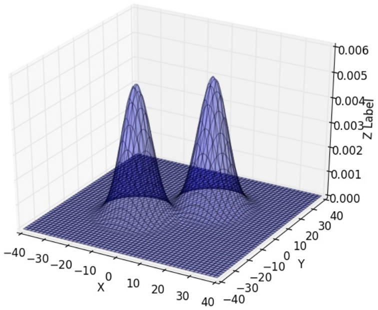




- Now what will we get by adding two such open towers ?
- We get a tower standing on an elevated base
- We can now pass this output through another sigmoid neuron to get the desired tower !

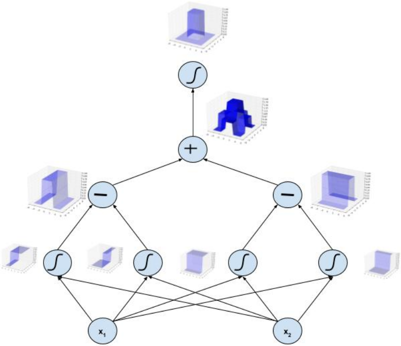


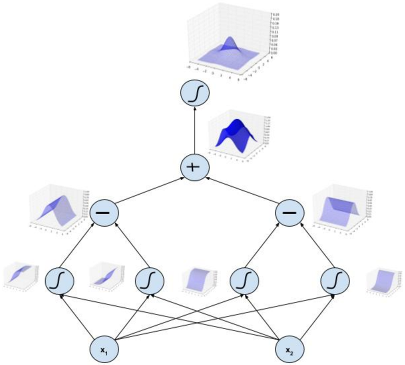
- Now what will we get by adding two such open towers ?
- We get a tower standing on an elevated base
- We can now pass this output through another sigmoid neuron to get the desired tower !
- We can now approximate any function by summing up many such towers



- 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 ?


+
−
−
x_1
x_2

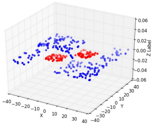

+
−
−
x_1
x_2

Think

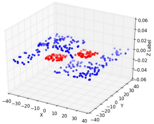
- For 1 dimensional input we needed 2 neurons to construct a tower
- For 2 dimensional input we needed 4 neurons to construct a tower
- How many neurons will you need to construct a tower in n dimensions ?

Time to retrospect

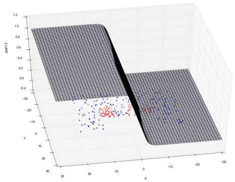
- Why do we care about approximating any arbitrary function ?
- Can we tie all this back to the classification problem that we have been dealing with ?



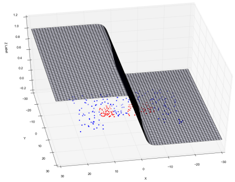
- We are interested in separating the blue points from the red points



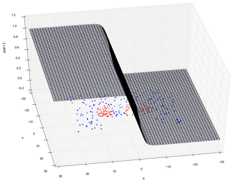
- We are interested in separating the blue points from the red points
- Suppose we use a single sigmoidal neuron to approximate the relation between $x = [x_1, x_2]$ and y

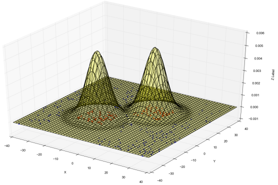


- We are interested in separating the blue points from the red points
- Suppose we use a single sigmoidal neuron to approximate the relation between $x = [x_1, x_2]$ and y



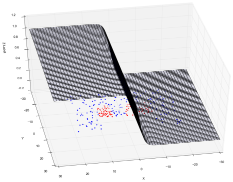
- We are interested in separating the blue points from the red points
- Suppose we use a single sigmoidal neuron to approximate the relation between $x = [x_1, x_2]$ and y
- Obviously, there will be errors (some blue points get classified as 1 and some red points get classified as 0)

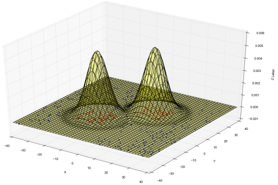




- We are interested in separating the blue points from the red points
- Suppose we use a single sigmoidal neuron to approximate the relation between $x = [x_1, x_2]$ and y
- Obviously, there will be errors (some blue points get classified as 1 and some red points get classified as 0)

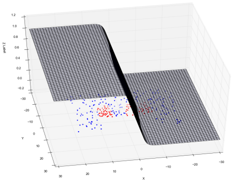
- This is what we actually want

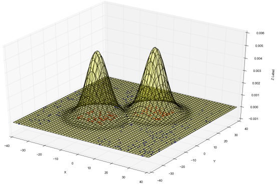




- We are interested in separating the blue points from the red points
- Suppose we use a single sigmoidal neuron to approximate the relation between $x = [x_1, x_2]$ and y
- Obviously, there will be errors (some blue points get classified as 1 and some red points get classified as 0)

- 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





- We are interested in separating the blue points from the red points
- Suppose we use a single sigmoidal neuron to approximate the relation between $x = [x_1, x_2]$ and y
- Obviously, there will be errors (some blue points get classified as 1 and some red points get classified as 0)

- 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 !!