

# Lecture 3

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

Mitesh M. Khapra

Department of Computer Science and Engineering  
Indian Institute of Technology Madras

## Acknowledgements

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

---

<sup>a</sup><http://neuralnetworksanddeeplearning.com/chap4.html>

# Module 3.1: Sigmoid Neuron

## 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 ...

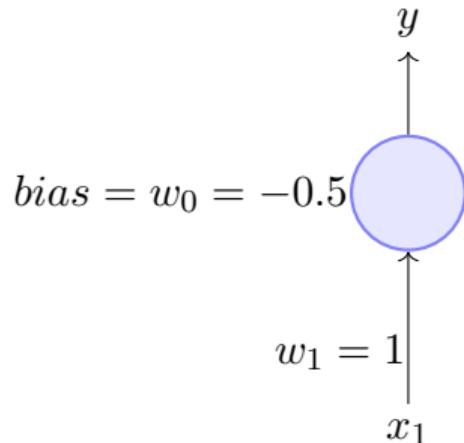
- Enough about boolean functions!
- What about arbitrary functions of the form  $y = f(x)$  where  $x \in \mathbb{R}^n$  (instead of  $\{0, 1\}^n$ ) and  $y \in \mathbb{R}$  (instead of  $\{0, 1\}$ ) ?
- Can we have a network which can (approximately) represent such functions ?

## The story ahead ...

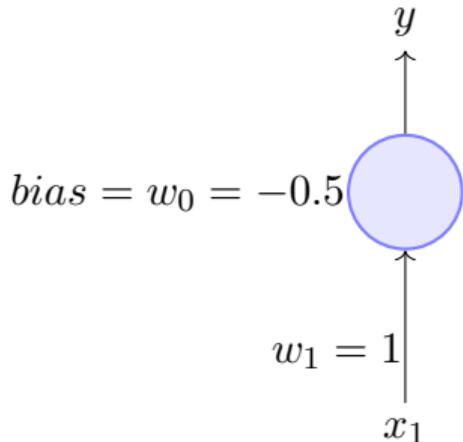
- Enough about boolean functions!
- What about arbitrary functions of the form  $y = f(x)$  where  $x \in \mathbb{R}^n$  (instead of  $\{0, 1\}^n$ ) and  $y \in \mathbb{R}$  (instead of  $\{0, 1\}$ ) ?
- Can we have a network which can (approximately) represent such functions ?
- Before answering the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

## Recall

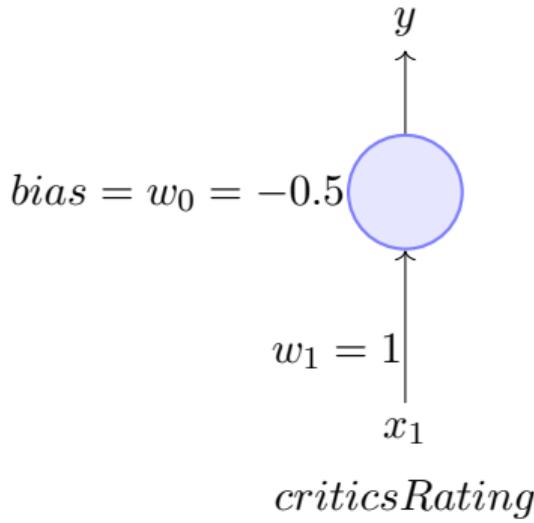
- A perceptron will fire if the weighted sum of its inputs is greater than the threshold ( $-w_0$ )



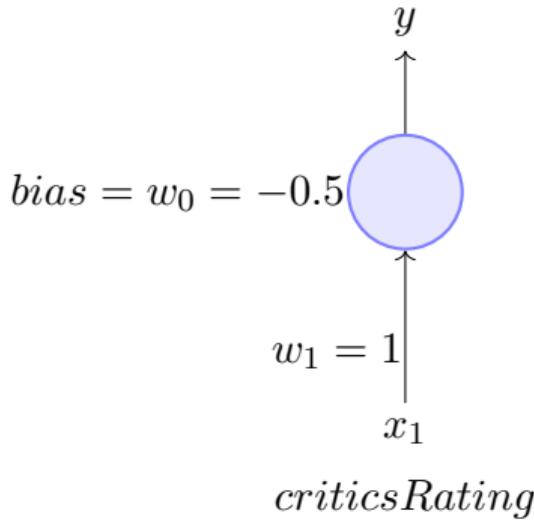
- The thresholding logic used by a perceptron is very harsh !



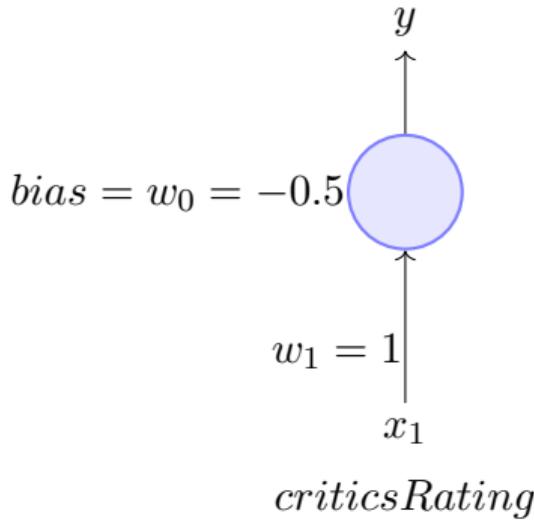
- The thresholding logic used by a perceptron is very harsh !
- For example, let us return to our problem of deciding whether we will like or dislike a movie



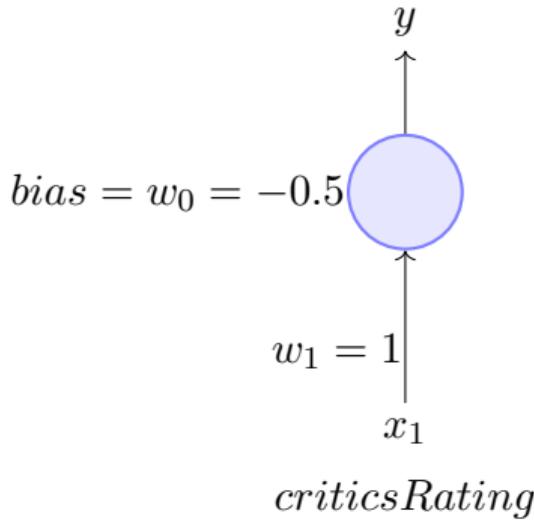
- The thresholding logic used by a perceptron is very harsh !
- For example, let us return to our problem of deciding whether we will like or dislike a movie
- Consider that we base our decision only on one input ( $x_1 = criticsRating$  which lies between 0 and 1)



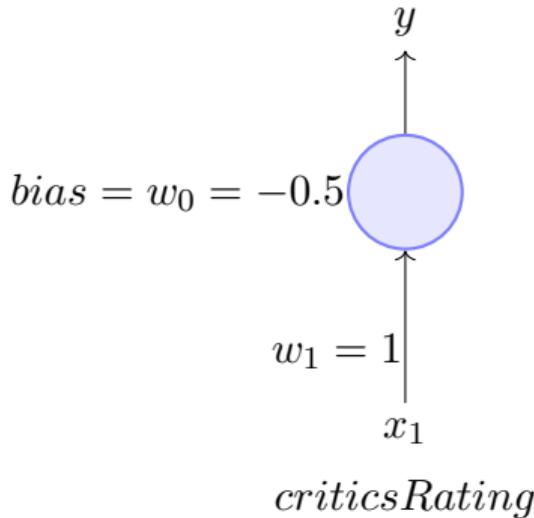
- The thresholding logic used by a perceptron is very harsh !
- For example, let us return to our problem of deciding whether we will like or dislike a movie
- Consider that we base our decision only on one input ( $x_1 = criticsRating$  which lies between 0 and 1)
- If the threshold is 0.5 ( $w_0 =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



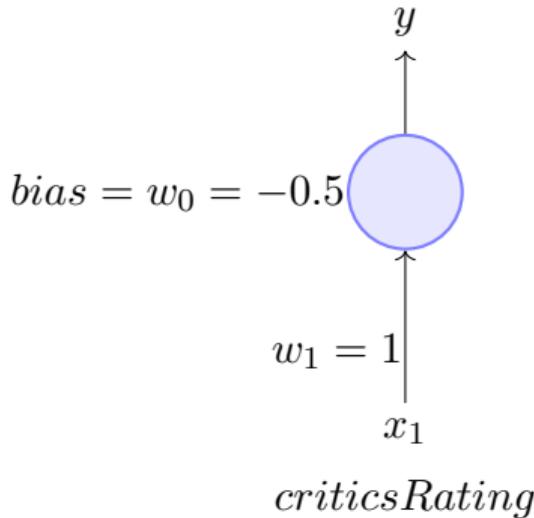
- The thresholding logic used by a perceptron is very harsh !
- For example, let us return to our problem of deciding whether we will like or dislike a movie
- Consider that we base our decision only on one input ( $x_1 = \text{criticsRating}$  which lies between 0 and 1)
- If the threshold is 0.5 ( $w_0 = -0.5$ ) and  $w_1 = 1$  then what would be the decision for a movie with  $\text{criticsRating} = 0.51$  ? (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  $criticsRating = 0.49$  ?

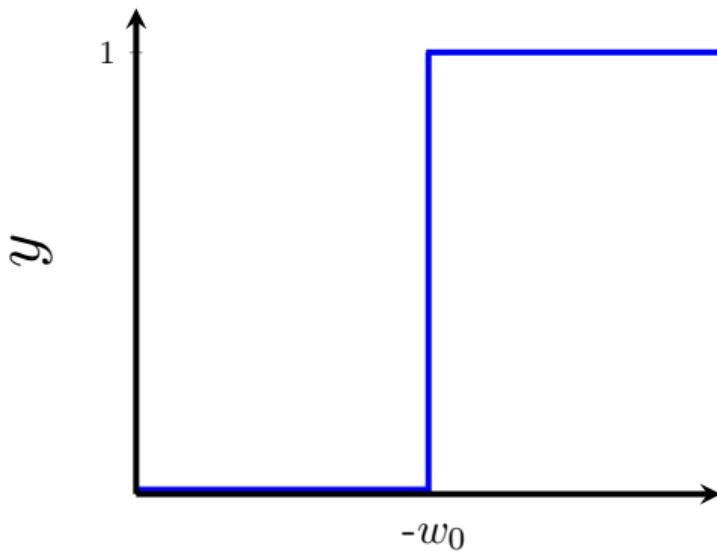


- The thresholding logic used by a perceptron is very harsh !
- For example, let us return to our problem of deciding whether we will like or dislike a movie
- Consider that we base our decision only on one input ( $x_1 = criticsRating$  which lies between 0 and 1)
- If the threshold is 0.5 ( $w_0 =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

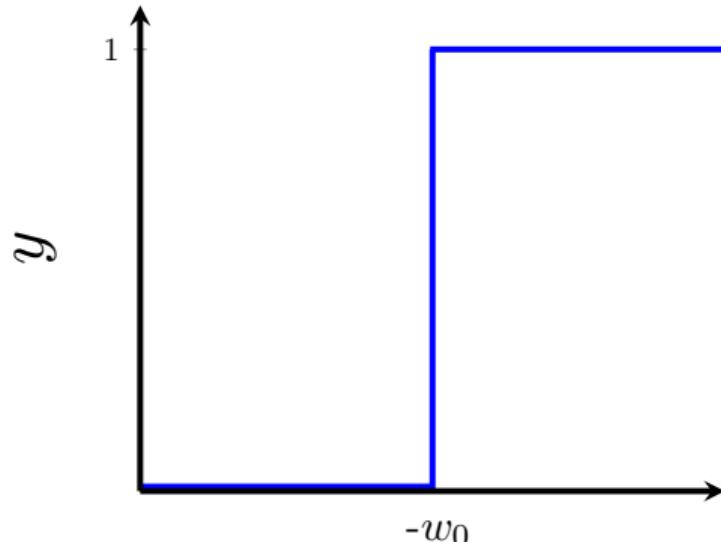


- The thresholding logic used by a perceptron is very harsh !
- For example, let us return to our problem of deciding whether we will like or dislike a movie
- Consider that we base our decision only on one input ( $x_1 = criticsRating$  which lies between 0 and 1)
- If the threshold is 0.5 ( $w_0 =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

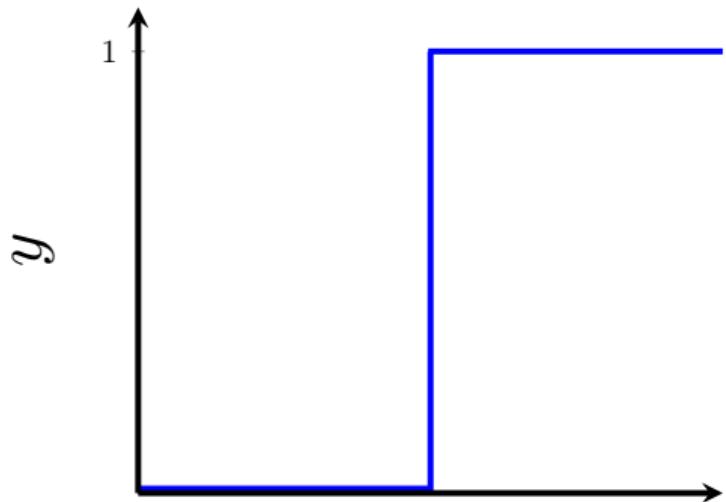
- This behavior is not a characteristic of the specific problem we chose or the specific weight and threshold that we chose



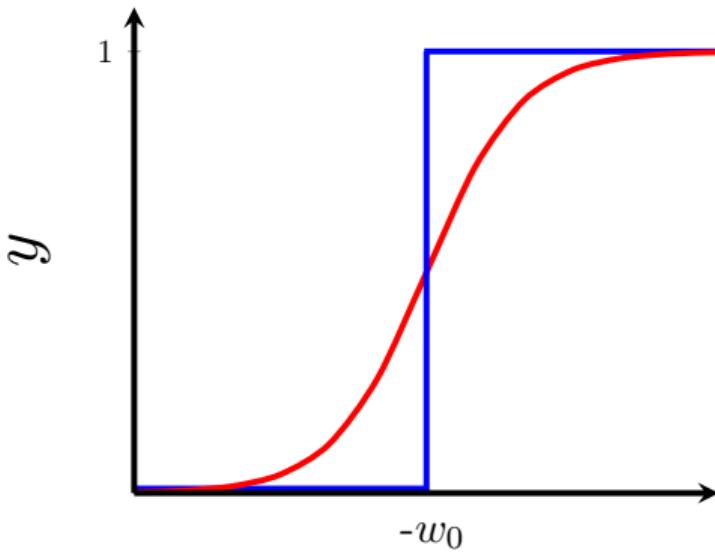
- This behavior is not a characteristic of the specific problem we chose or the specific weight and threshold that we chose
- It is a characteristic of the perceptron function itself which behaves like a step function



- 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 ( $-w_0$ )

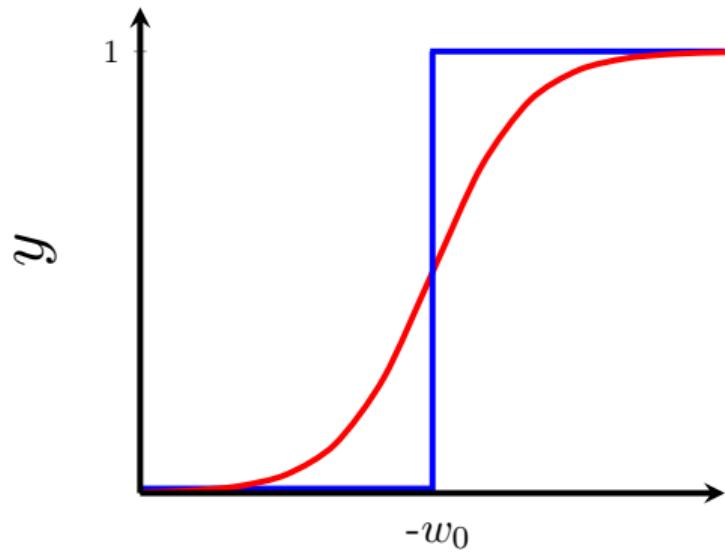


- This behavior is not a characteristic of the specific problem we chose or the specific weight and threshold that we chose
- It is a characteristic of the perceptron function itself which behaves like a step function
- There will always be this sudden change in the decision (from 0 to 1) when  $\sum_{i=1}^n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



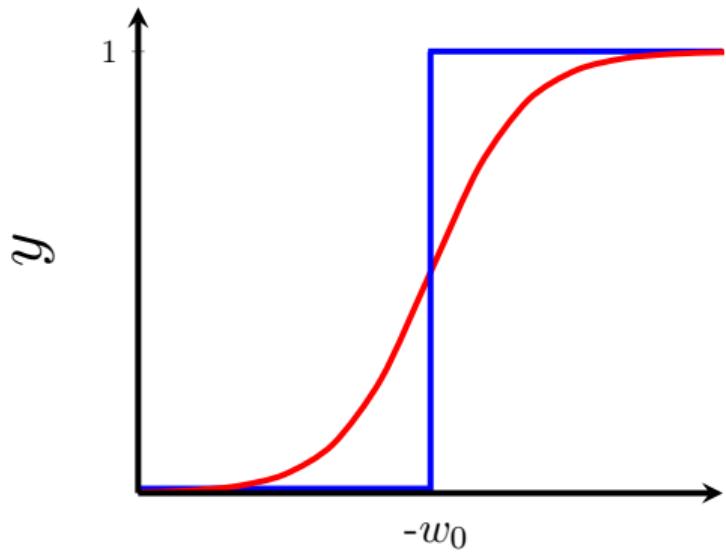
$$z = \sum_{i=1}^n w_i x_i$$

- 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

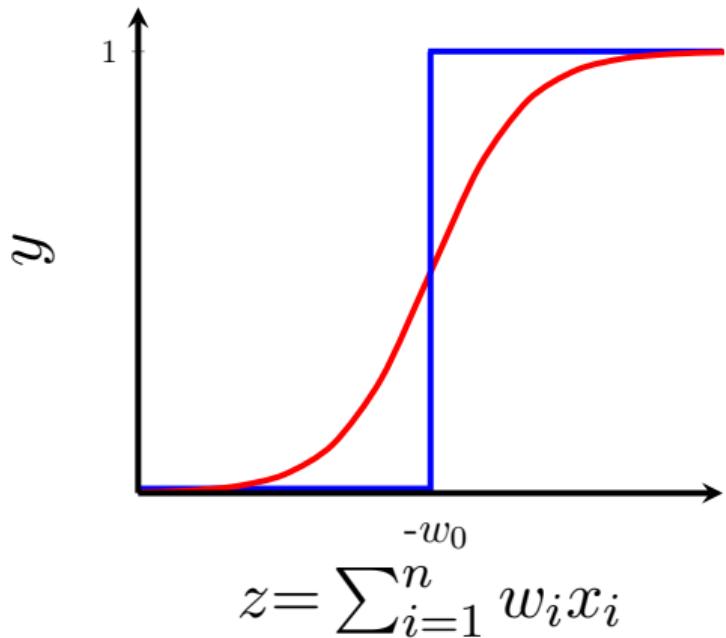
$$y = \frac{1}{1 + e^{-(w_0 + \sum_{i=1}^n w_i x_i)}}$$



- 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

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

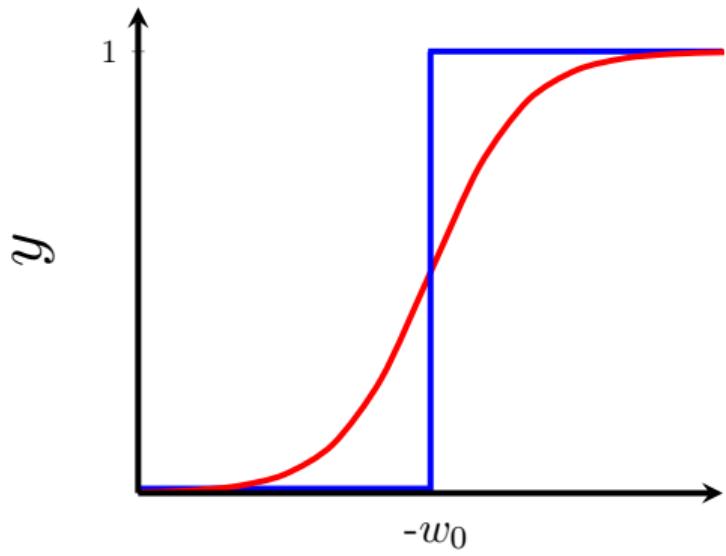
- We no longer see a sharp transition around the threshold  $-w_0$



- 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



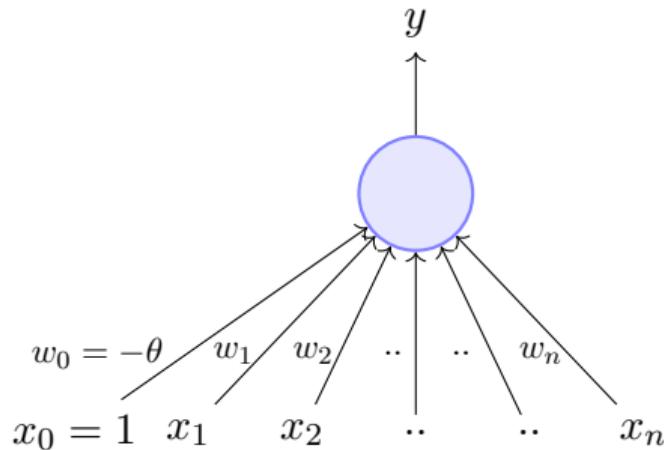
$$z = \sum_{i=1}^n w_i x_i$$

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

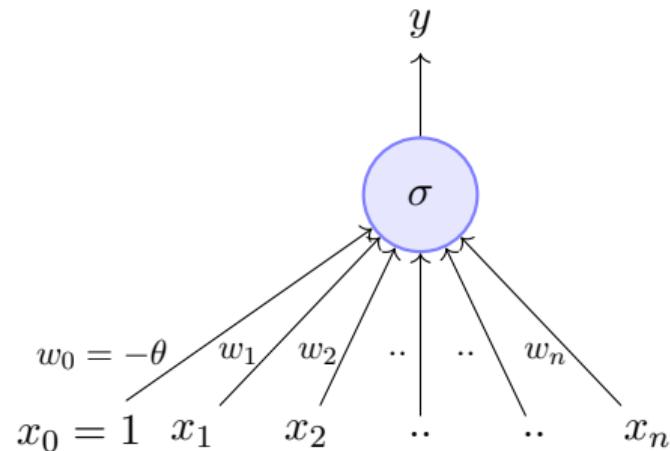
## Perceptron



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

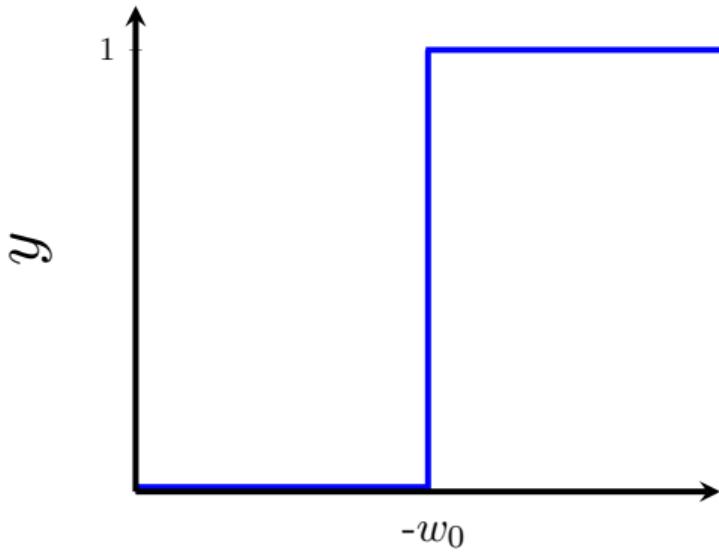
$$= 0 \quad if \sum_{i=0}^n w_i * x_i < 0$$

## Sigmoid (logistic) Neuron



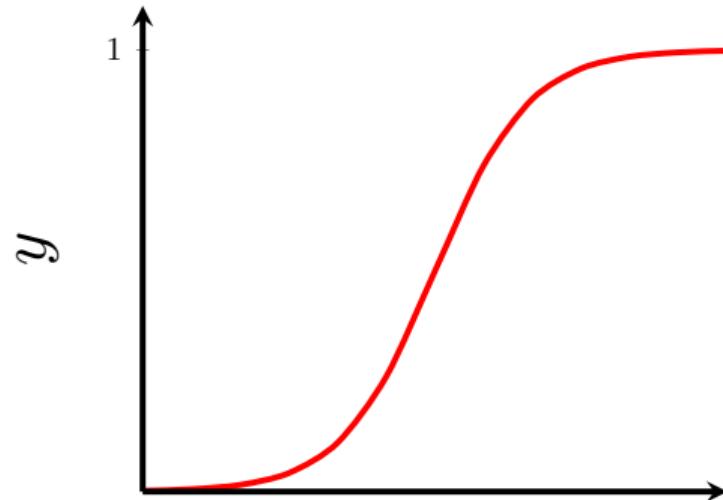
$$y = \frac{1}{1 + e^{-(\sum_{i=0}^n w_i x_i)}}$$

### Perceptron



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

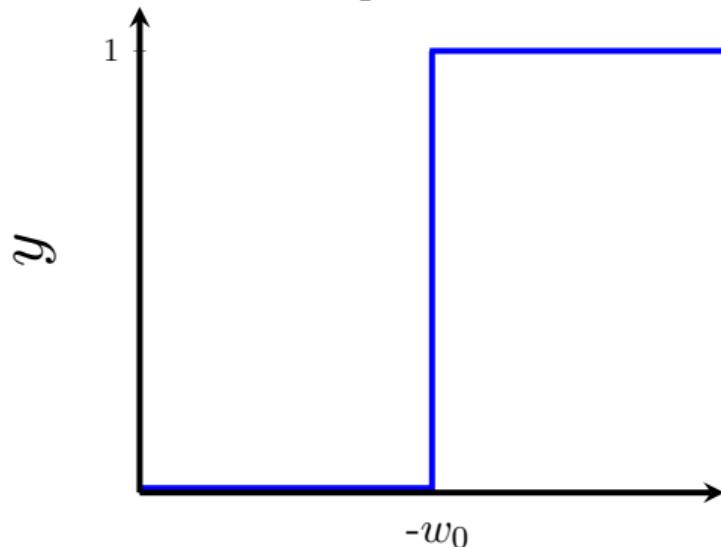
### Sigmoid Neuron



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

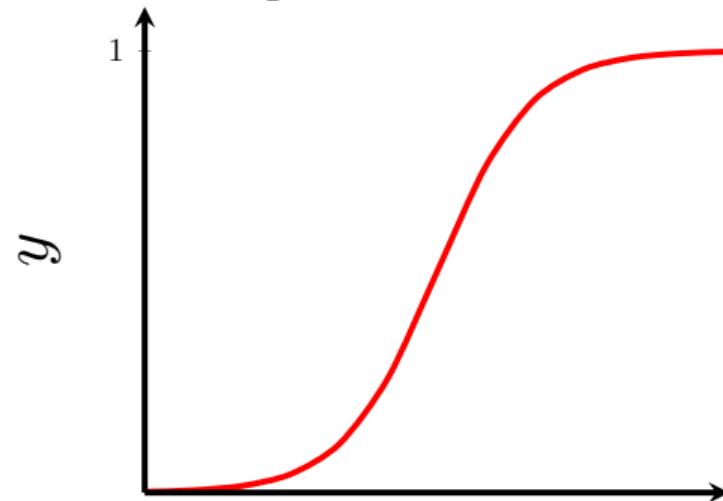
Not smooth, not continuous (at  $w_0$ ), **not differentiable**

### Perceptron



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

### Sigmoid Neuron



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

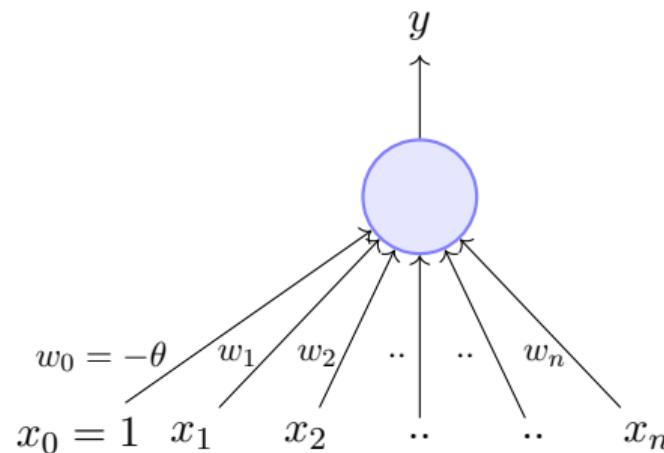
Not smooth, not continuous (at  $w_0$ ), **not differentiable**

Smooth, continuous, **differentiable**

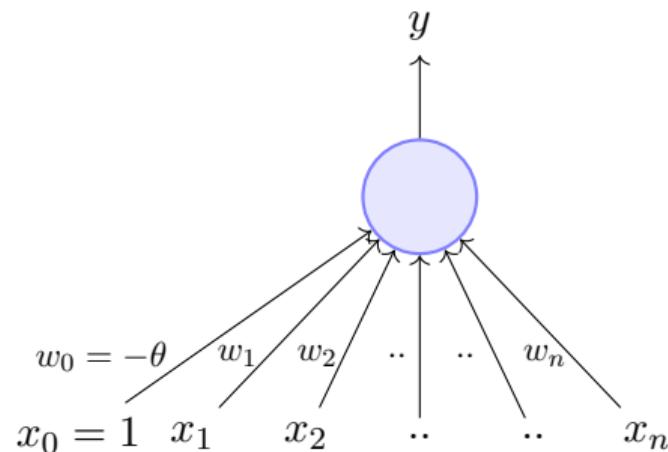
## Module 3.2: A typical Supervised Machine Learning Setup

- What next ?

## Sigmoid (logistic) Neuron

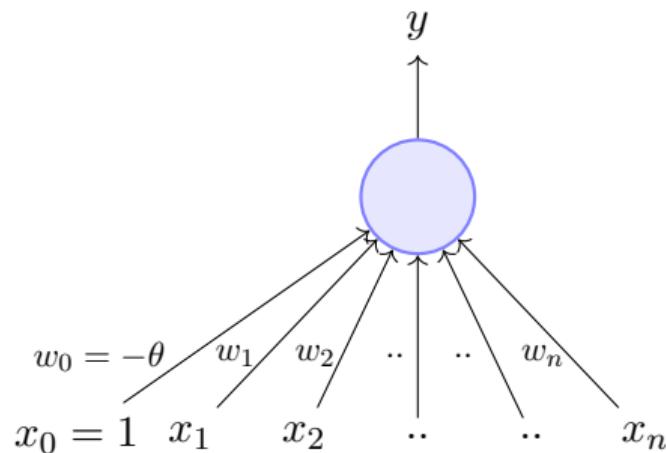


## Sigmoid (logistic) Neuron

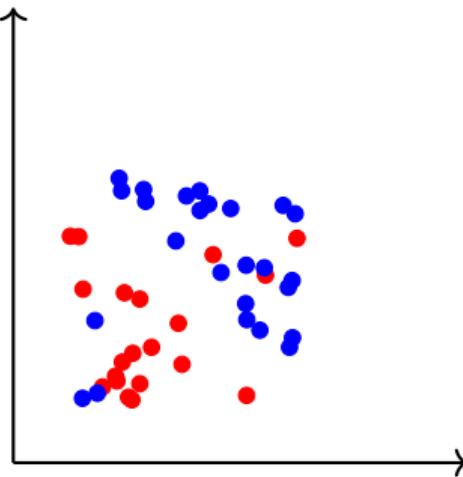


- What next ?
- Well, just as we had an algorithm for learning the weights of a perceptron, we also need a way of learning the weights of a sigmoid neuron

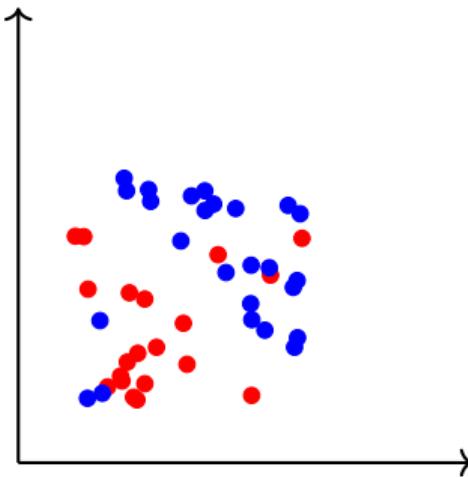
## Sigmoid (logistic) Neuron



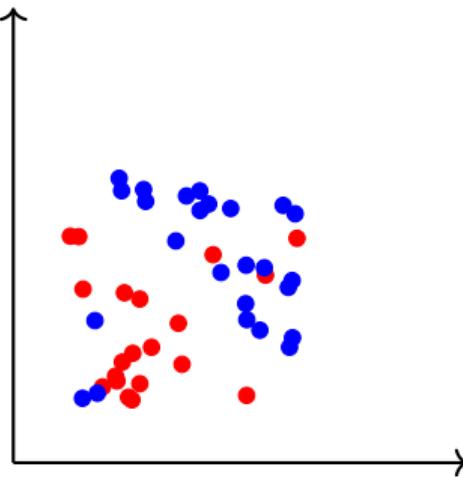
- What next ?
- Well, just as we had an algorithm for learning the weights of a perceptron, we also need a way of learning the weights of a sigmoid neuron
- Before we see such an algorithm we will revisit the concept of **error**



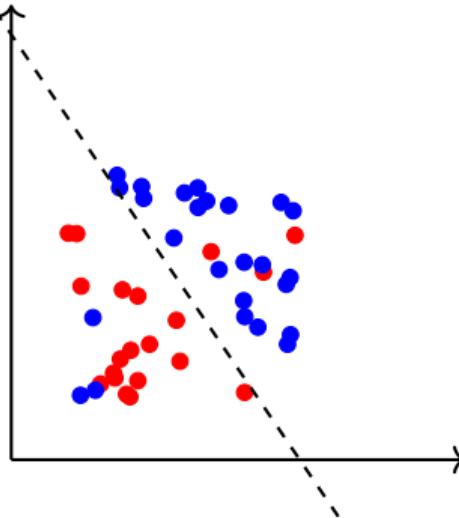
- Earlier we mentioned that a single perceptron cannot deal with this data because it is not linearly separable



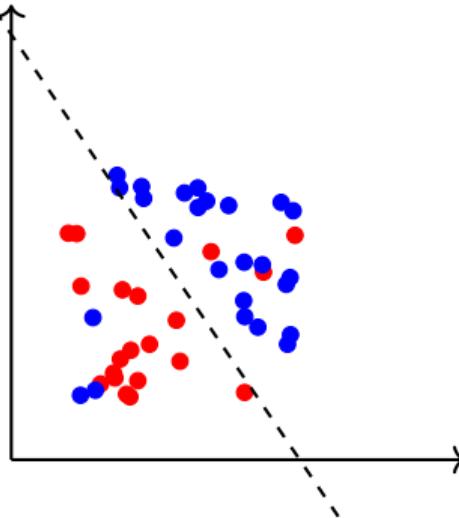
- Earlier we mentioned that a single perceptron cannot deal with this data because it is not linearly separable
- What does “cannot deal with” mean?



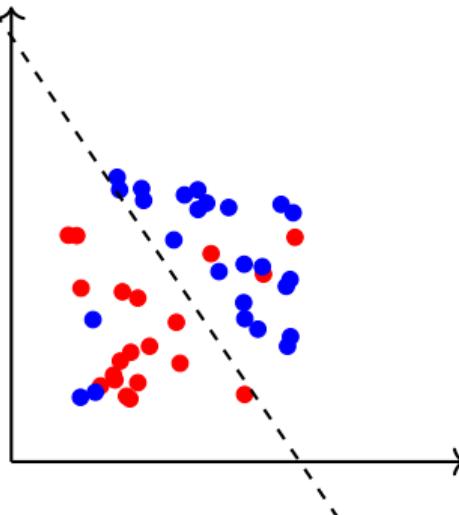
- Earlier we mentioned that a single perceptron cannot deal with this data because it is not linearly separable
- What does “cannot deal with” mean?
- What would happen if we use a perceptron model to classify this data ?



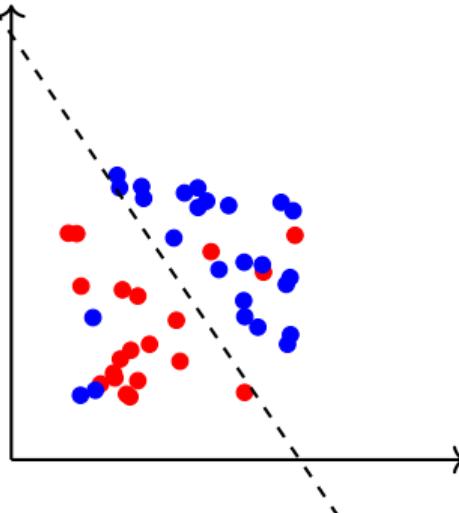
- Earlier we mentioned that a single perceptron cannot deal with this data because it is not linearly separable
- What does “cannot deal with” mean?
- What would happen if we use a perceptron model to classify this data ?
- We would probably end up with a line like this ...



- 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  $\mathbf{x}$  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  $\mathbf{x}$  and  $y$ . For example,

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

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  $\mathbf{x}$  and  $y$ . For example,

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

or  $\hat{y} = \mathbf{W}^T \mathbf{x}$

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  $\mathbf{x}$  and  $y$ . For example,

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

or  $\hat{y} = \mathbf{W}^T \mathbf{x}$

or  $\hat{y} = \mathbf{x}^T \mathbf{W} \mathbf{x}$

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  $\mathbf{x}$  and  $y$ . For example,

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

or  $\hat{y} = \mathbf{W}^T \mathbf{x}$

or  $\hat{y} = \mathbf{x}^T \mathbf{W} \mathbf{x}$

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  $\mathbf{x}$  and  $y$ . For example,

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

or  $\hat{y} = \mathbf{W}^T \mathbf{x}$

or  $\hat{y} = \mathbf{x}^T \mathbf{W} \mathbf{x}$

or just about any function

- **Parameters:** In all the above cases,  $\mathbf{W}$  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  $\mathbf{x}$  and  $y$ . For example,

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

or  $\hat{y} = \mathbf{W}^T \mathbf{x}$

or  $\hat{y} = \mathbf{x}^T \mathbf{W} \mathbf{x}$

or just about any function

- **Parameters:** In all the above cases,  $\mathbf{W}$  is a parameter which needs to be learned from the data
- **Learning algorithm:** An algorithm for learning the parameters ( $\mathbf{W}$ ) 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  $\mathbf{x}$  and  $y$ . For example,

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

$$\text{or } \hat{y} = \mathbf{W}^T \mathbf{x}$$

$$\text{or } \hat{y} = \mathbf{x}^T \mathbf{W} \mathbf{x}$$

or just about any function

- **Parameters:** In all the above cases,  $\mathbf{W}$  is a parameter which needs to be learned from the data
- **Learning algorithm:** An algorithm for learning the parameters ( $\mathbf{W}$ ) 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  $\mathbf{x}$  and  $y$ . For example,

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

or  $\hat{y} = \mathbf{W}^T \mathbf{x}$

or  $\hat{y} = \mathbf{x}^T \mathbf{W} \mathbf{x}$

or just about any function

- **Parameters:** In all the above cases,  $\mathbf{W}$  is a parameter which needs to be learned from the data
- **Learning algorithm:** An algorithm for learning the parameters ( $\mathbf{W}$ ) 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:**  $\{x_i = \text{movie}, y_i = \text{like/dislike}\}_{i=1}^n$

As an illustration, consider our movie example

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

As an illustration, consider our movie example

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

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

As an illustration, consider our movie example

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

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

- **Parameter:**  $\mathbf{W}$

As an illustration, consider our movie example

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

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

- **Parameter:**  $\mathbf{W}$
- **Learning algorithm:** Gradient Descent [we will see soon]

As an illustration, consider our movie example

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

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

- **Parameter:**  $\mathbf{W}$
- **Learning algorithm:** Gradient Descent [we will see soon]
- **Objective/Loss/Error function:**

As an illustration, consider our movie example

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

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

- **Parameter:**  $\mathbf{W}$
- **Learning algorithm:** Gradient Descent [we will see soon]
- **Objective/Loss/Error function:** One possibility is

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

As an illustration, consider our movie example

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

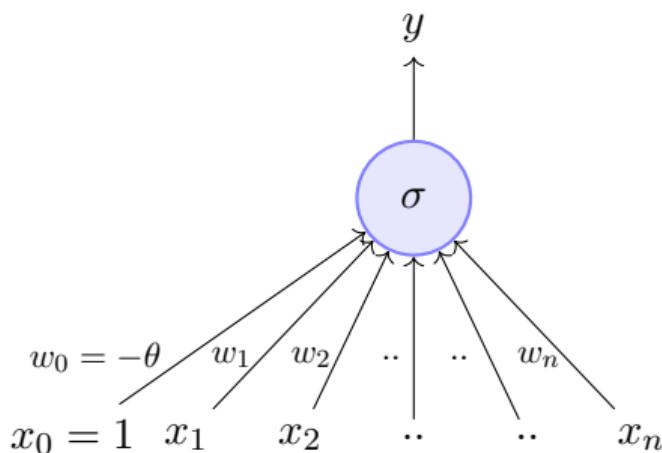
$$\hat{y} = \frac{1}{1 + e^{-(\mathbf{W}^T \mathbf{x})}}$$

- **Parameter:**  $\mathbf{W}$
- **Learning algorithm:** Gradient Descent [we will see soon]
- **Objective/Loss/Error function:** One possibility is

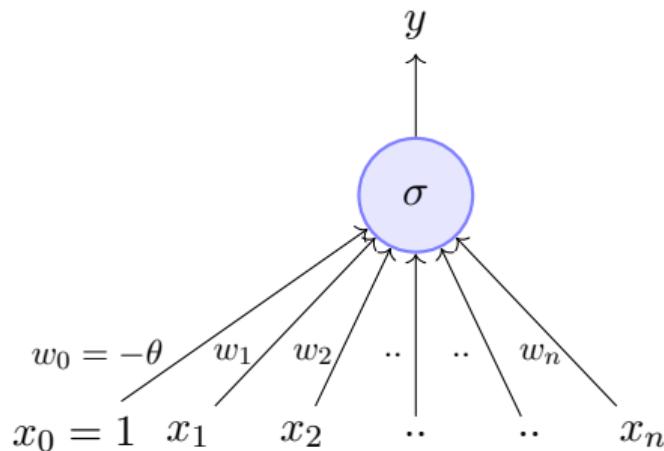
$$\mathcal{L}(\mathbf{W}) = \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

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

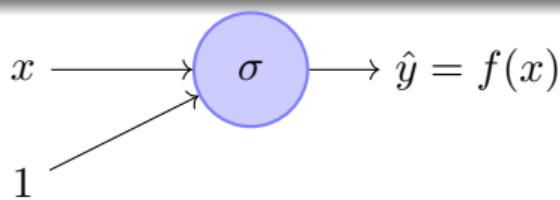
## Module 3.3: Learning Parameters: (Infeasible) guess work



- Keeping this supervised ML setup in mind, we will now focus on this **model** and discuss an **algorithm** for learning the **parameters** of this model from some given **data** using an appropriate **objective function**

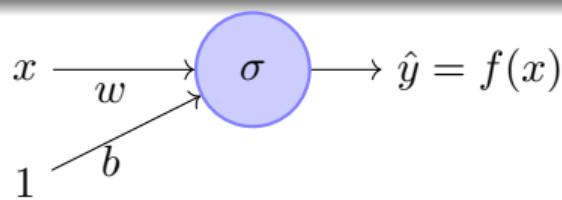


- Keeping this supervised ML setup in mind, we will now focus on this **model** and discuss an **algorithm** for learning the **parameters** of this model from some given **data** using an appropriate **objective function**
- $\sigma$  stands for the sigmoid function (logistic function in this case)



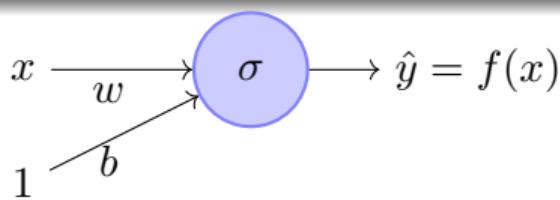
$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

- Keeping this supervised ML setup in mind, we will now focus on this **model** and discuss an **algorithm** for learning the **parameters** of this model from some given **data** using an appropriate **objective function**
- $\sigma$  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



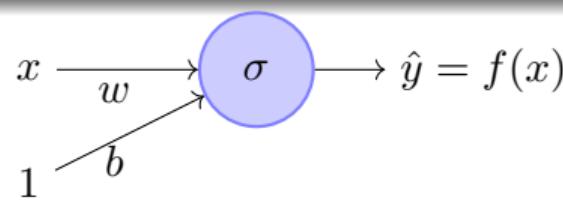
$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

- Keeping this supervised ML setup in mind, we will now focus on this **model** and discuss an **algorithm** for learning the **parameters** of this model from some given **data** using an appropriate **objective function**
- $\sigma$  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)

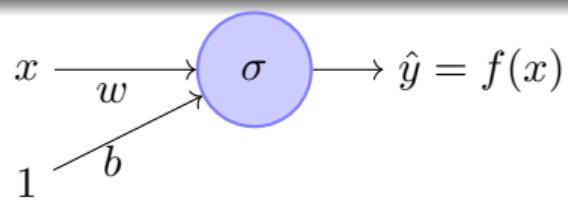


$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

- Keeping this supervised ML setup in mind, we will now focus on this **model** and discuss an **algorithm** for learning the **parameters** of this model from some given **data** using an appropriate **objective function**
- $\sigma$  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  $\text{criticsRating}(y)$  given  $\text{imdbRating}(x)$  (for no particular reason)



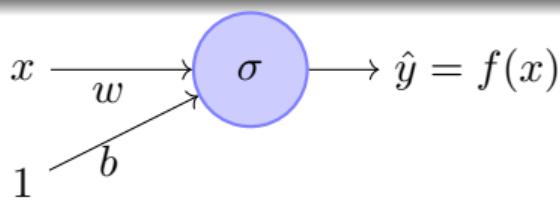
$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$



Input for training

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

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



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

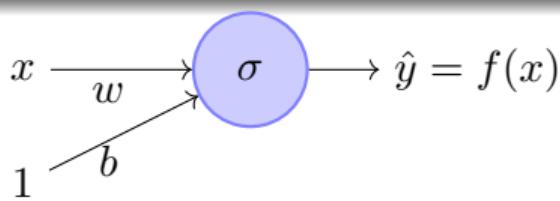
Input for training

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

Training objective

Find  $w$  and  $b$  such that:

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



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

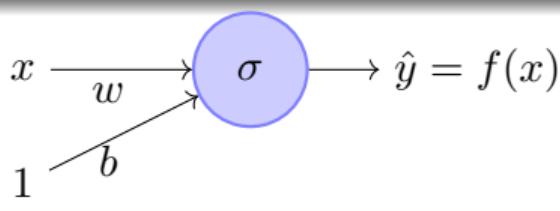
Input for training

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

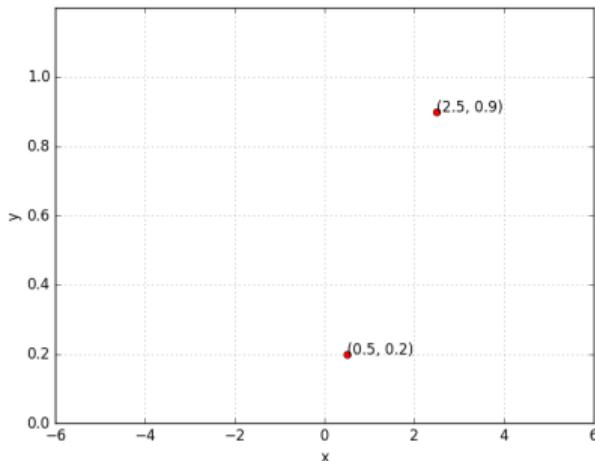
Training objective

Find  $w$  and  $b$  such that:

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

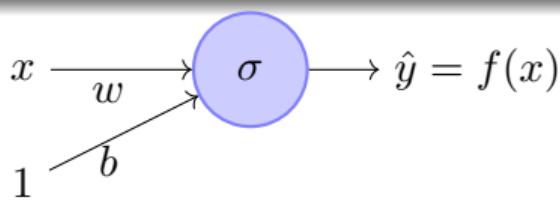


$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

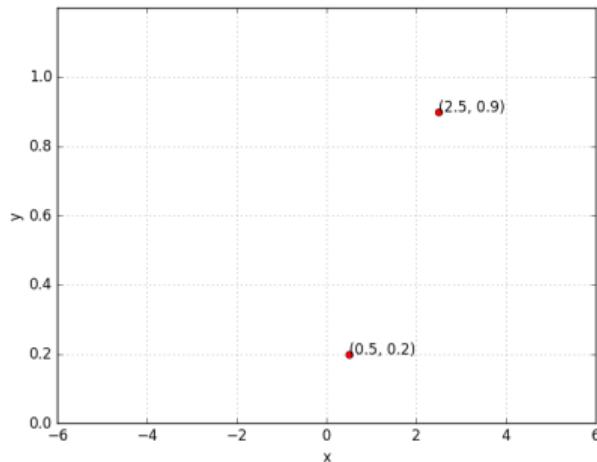


What does it mean to train the network?

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

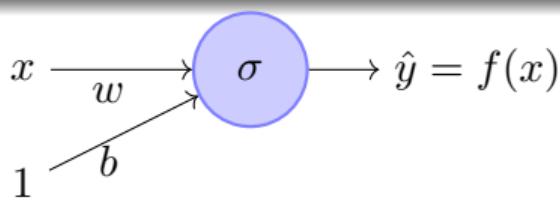


$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

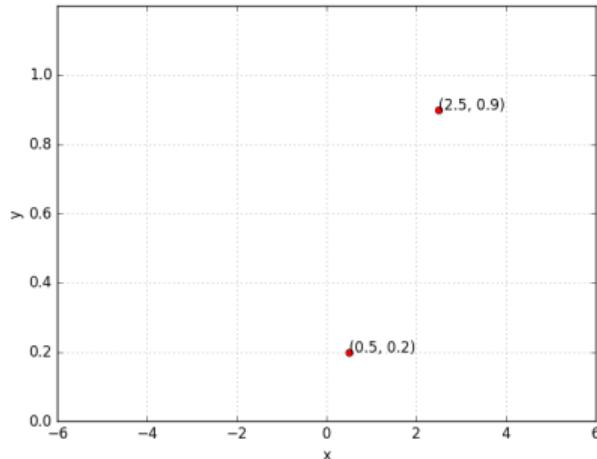


What does it mean to train the network?

- 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:

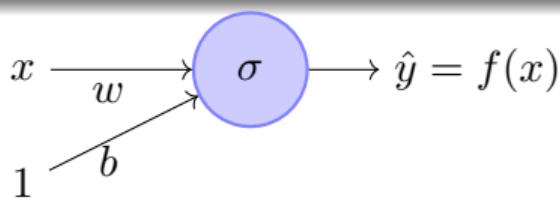


$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

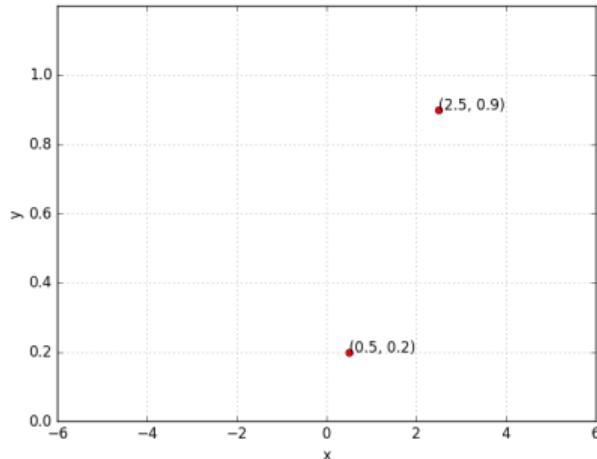


What does it mean to train the network?

- 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  $f(2.5) \rightarrow 0.9$



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

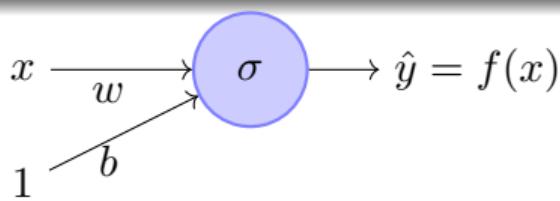


What does it mean to train the network?

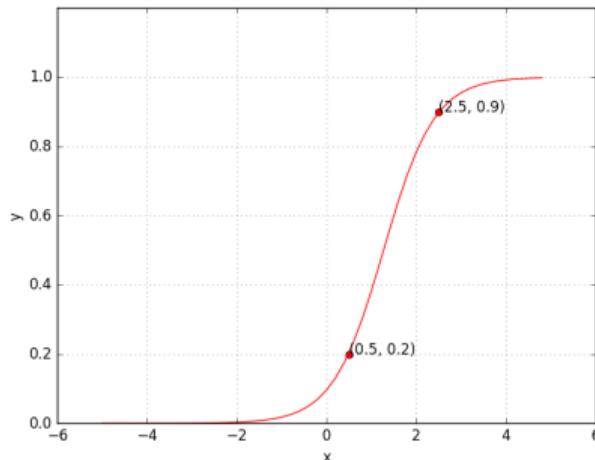
- Suppose we train the network with  $(x, y) = (0.5, 0.2)$  and  $(2.5, 0.9)$
- At the end of training we expect to find  $w^*$ ,  $b^*$  such that:
- $f(0.5) \rightarrow 0.2$  and  $f(2.5) \rightarrow 0.9$

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



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



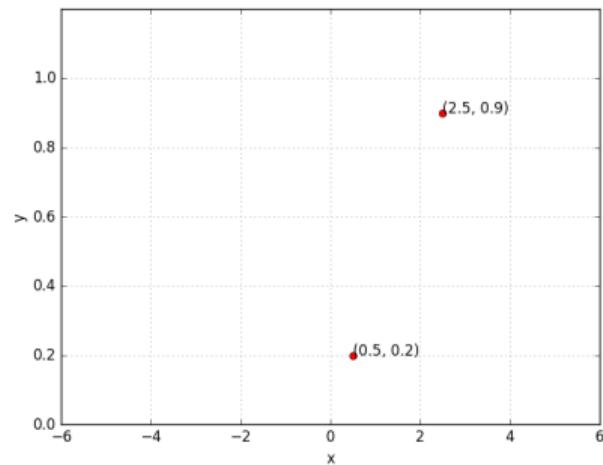
What does it mean to train the network?

- 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  $f(2.5) \rightarrow 0.9$

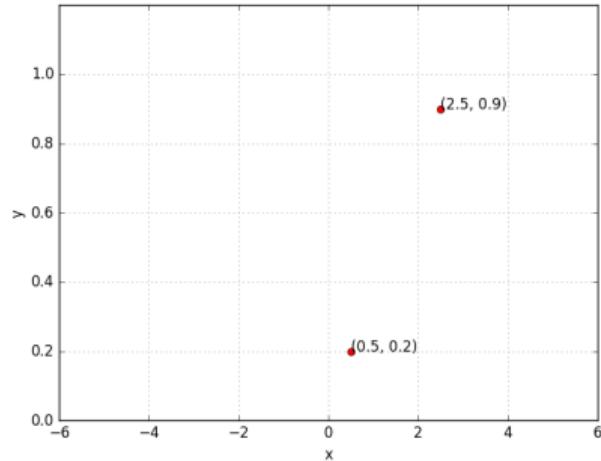
In other words...

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

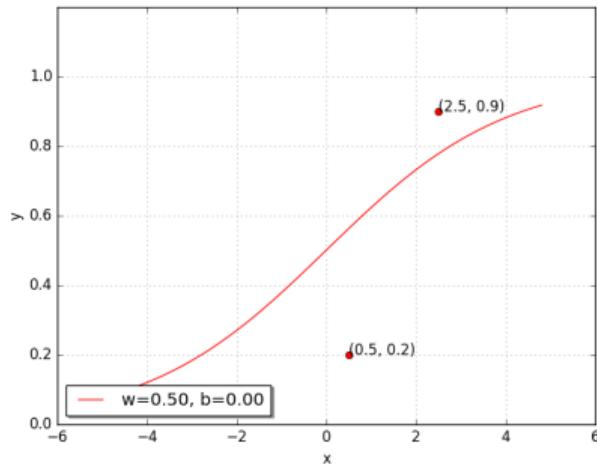
*Let us see this in more detail....*



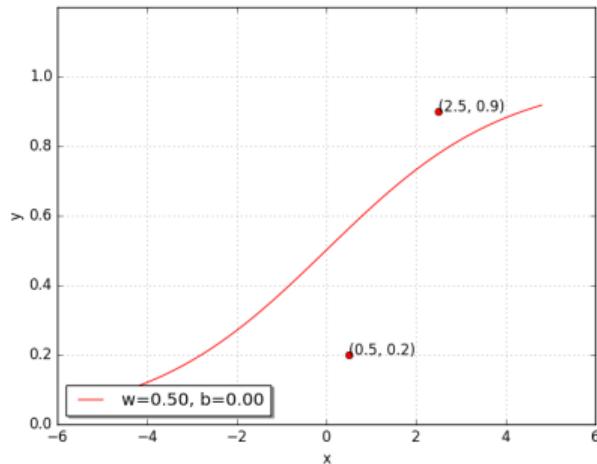
- Can we try to find such a  $w^*, b^*$  manually



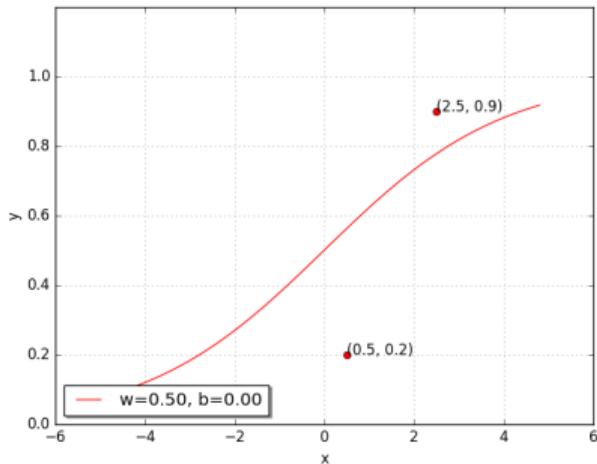
- Can we try to find such a  $w^*, b^*$  manually
- Let us try a random guess.. (say,  $w = 0.5, b = 0$ )

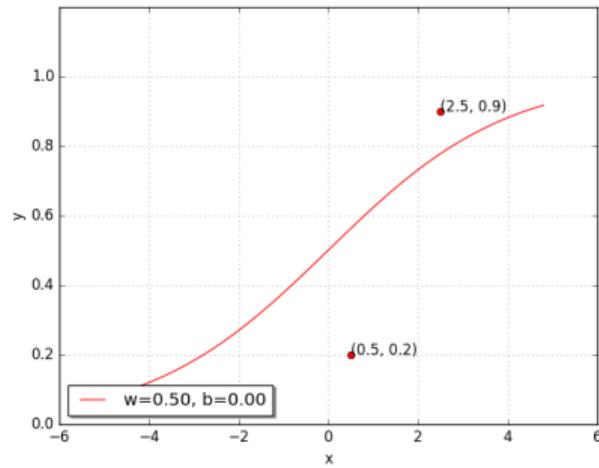


- Can we try to find such a  $w^*, b^*$  manually
- Let us try a random guess.. (say,  $w = 0.5, b = 0$ )
- Clearly not good, but how bad is it ?

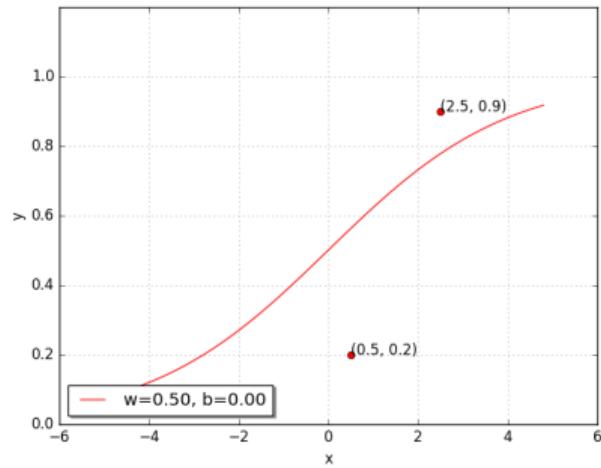


- Can we try to find such a  $w^*, b^*$  manually
- Let us try a random guess.. (say,  $w = 0.5, b = 0$ )
- Clearly not good, but how bad is it ?
- Let us revisit  $\mathcal{L}(w, b)$  to see how bad it is ...

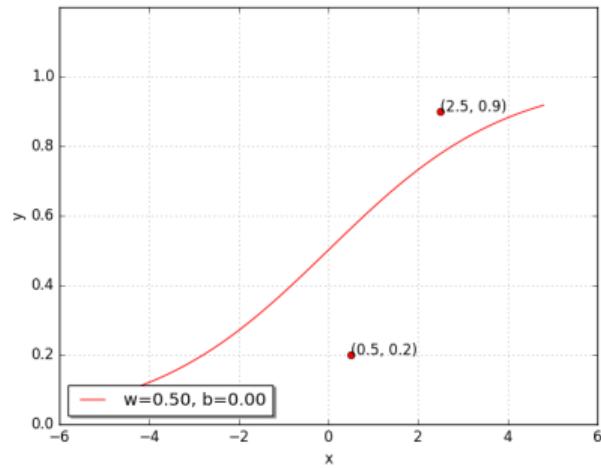




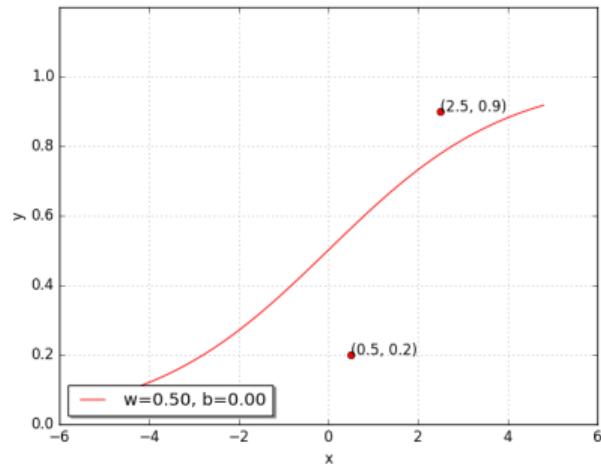
$$\mathcal{L}(w, b) = \frac{1}{2} * \sum_{i=1}^N (y_i - f(x_i))^2$$



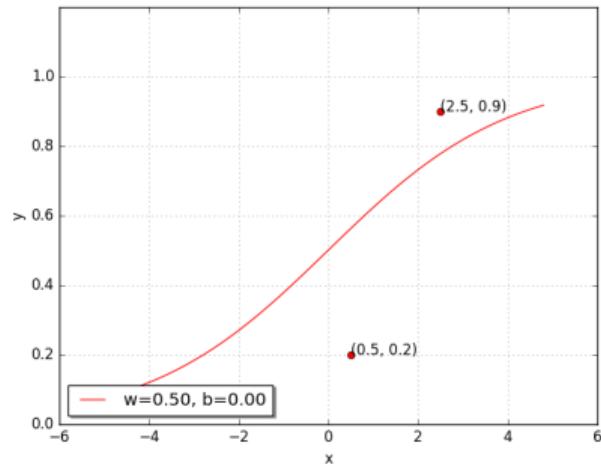
$$\begin{aligned}\mathcal{L}(w, b) &= \frac{1}{2} * \sum_{i=1}^N (y_i - f(x_i))^2 \\ &= \frac{1}{2} * (y_1 - f(x_1))^2 + (y_2 - f(x_2))^2\end{aligned}$$



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



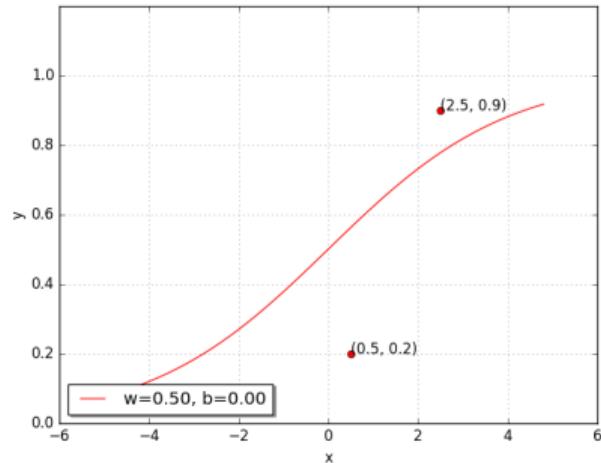
$$\begin{aligned}
 \mathcal{L}(w, b) &= \frac{1}{2} * \sum_{i=1}^N (y_i - f(x_i))^2 \\
 &= \frac{1}{2} * (y_1 - f(x_1))^2 + (y_2 - f(x_2))^2 \\
 &= \frac{1}{2} * (0.9 - f(2.5))^2 + (0.2 - f(0.5))^2 \\
 &= 0.073
 \end{aligned}$$



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

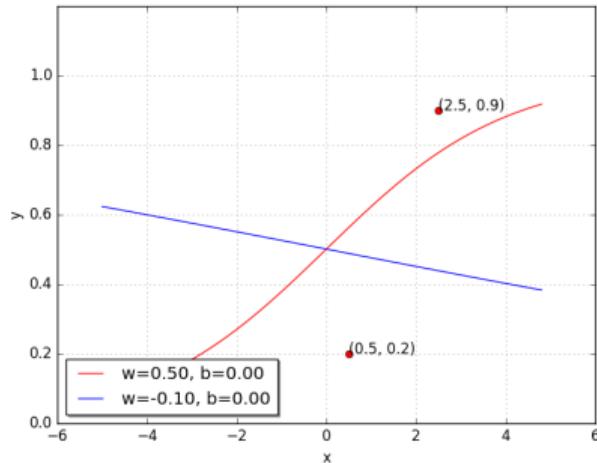
We want  $\mathcal{L}(w, b)$  to be as close to 0 as possible

Let us try some other values of w, b



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

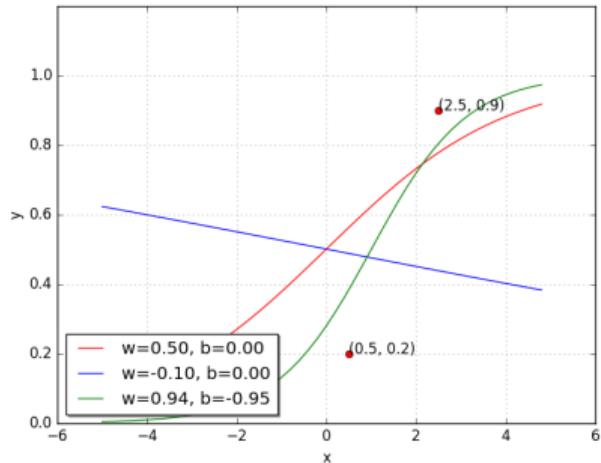
Let us try some other values of w, b



$w$	$b$	$\mathcal{L}(w, b)$
0.50	0.00	0.0730
-0.10	0.00	0.1481

Oops!! this made things even worse...

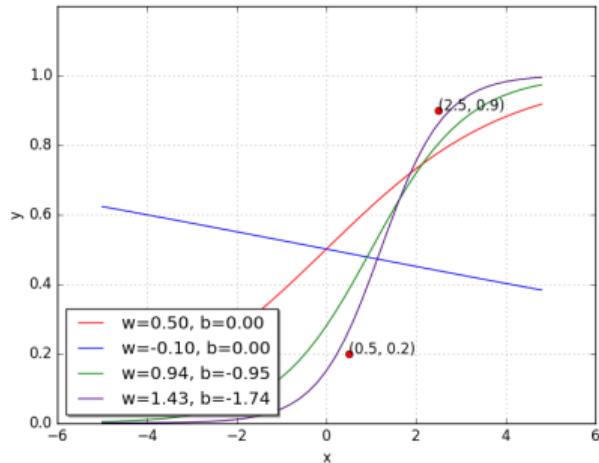
Let us try some other values of  $w$ ,  $b$



$w$	$b$	$\mathcal{L}(w, b)$
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...

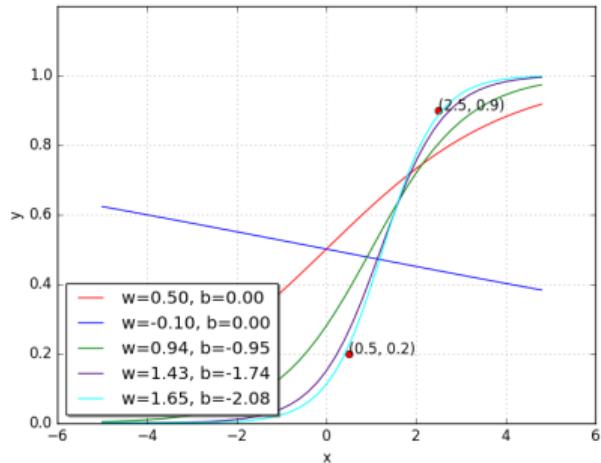
Let us try some other values of  $w$ ,  $b$



$w$	$b$	$\mathcal{L}(w, b)$
0.50	0.00	0.0730
-0.10	0.00	0.1481
0.94	-0.94	0.0214
1.42	-1.73	0.0028

Let us keep going in this direction, *i.e.*, increase  $w$  and decrease  $b$

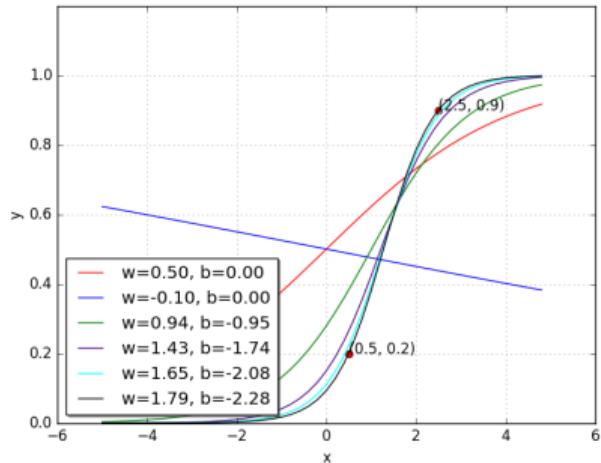
Let us try some other values of  $w$ ,  $b$



$w$	$b$	$\mathcal{L}(w, b)$
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

Let us keep going in this direction, *i.e.*, increase  $w$  and decrease  $b$

Let us try some other values of  $w$ ,  $b$



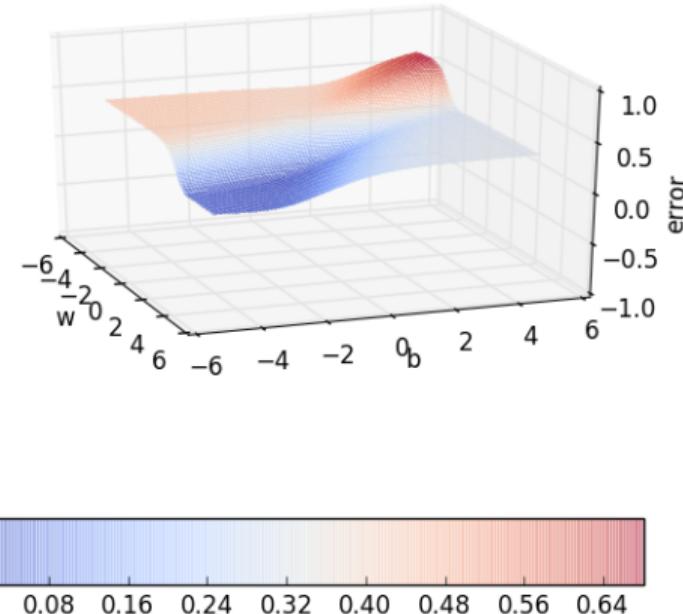
$w$	$b$	$\mathcal{L}(w, b)$
0.50	0.00	0.0730
-0.10	0.00	0.1481
0.94	-0.94	0.0214
1.42	-1.73	0.0028
1.65	-2.08	0.0003
1.78	-2.27	0.0000

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

*Let us look at something better than our “guess work” algorithm....*

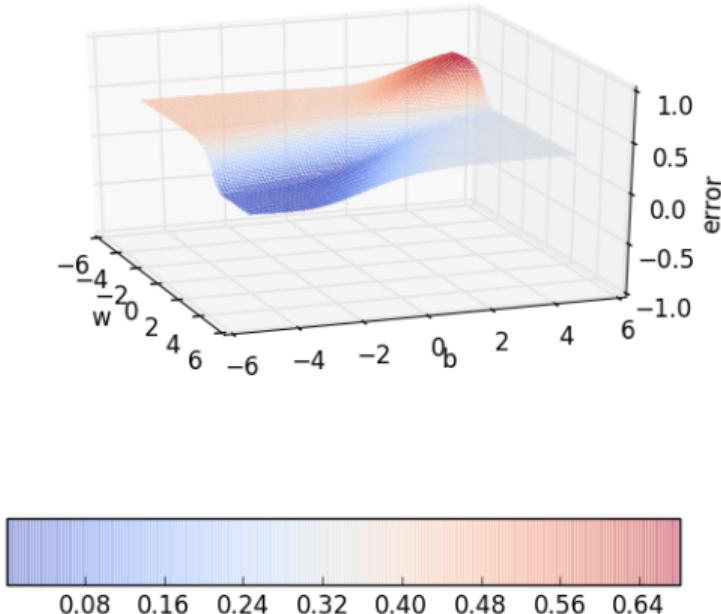
- Since we have only 2 points and 2 parameters ( $w$ ,  $b$ ) we can easily plot  $\mathcal{L}(w, b)$  for different values of  $(w, b)$  and pick the one where  $\mathcal{L}(w, b)$  is minimum

### Random search on error surface



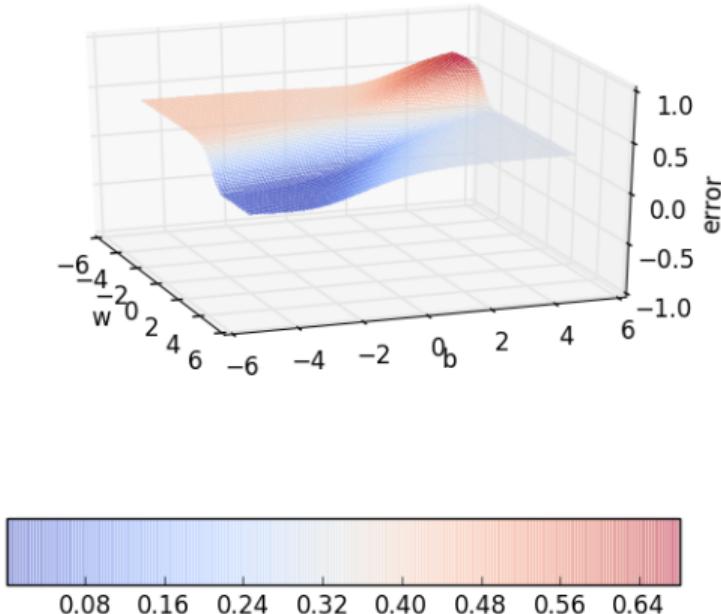
- Since we have only 2 points and 2 parameters ( $w, b$ ) we can easily plot  $\mathcal{L}(w, b)$  for different values of  $(w, b)$  and pick the one where  $\mathcal{L}(w, b)$  is minimum

### Random search on error surface



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

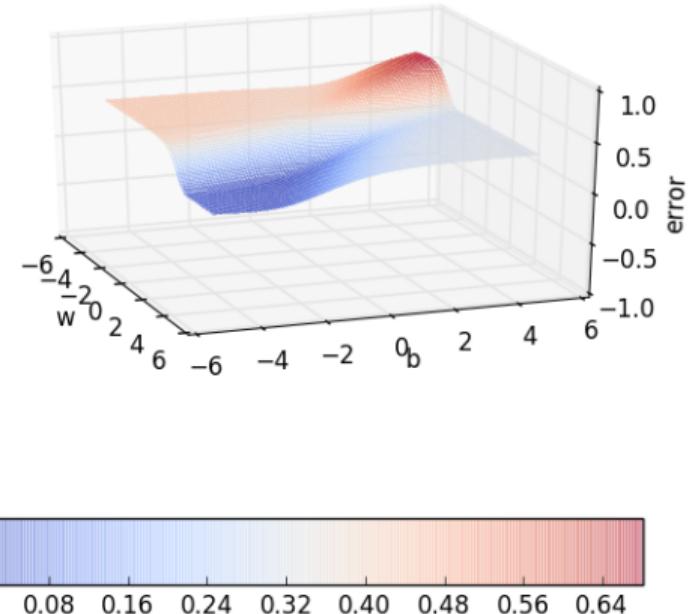
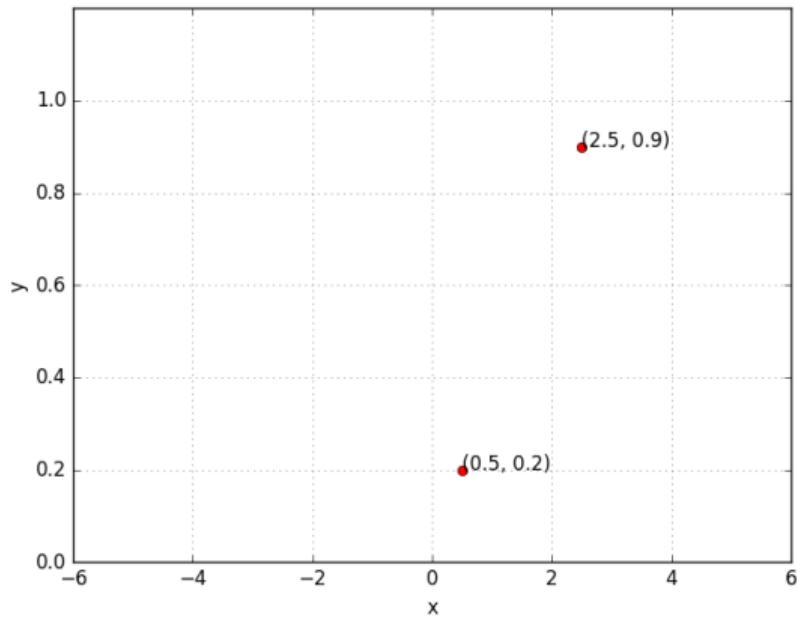
### Random search on error surface

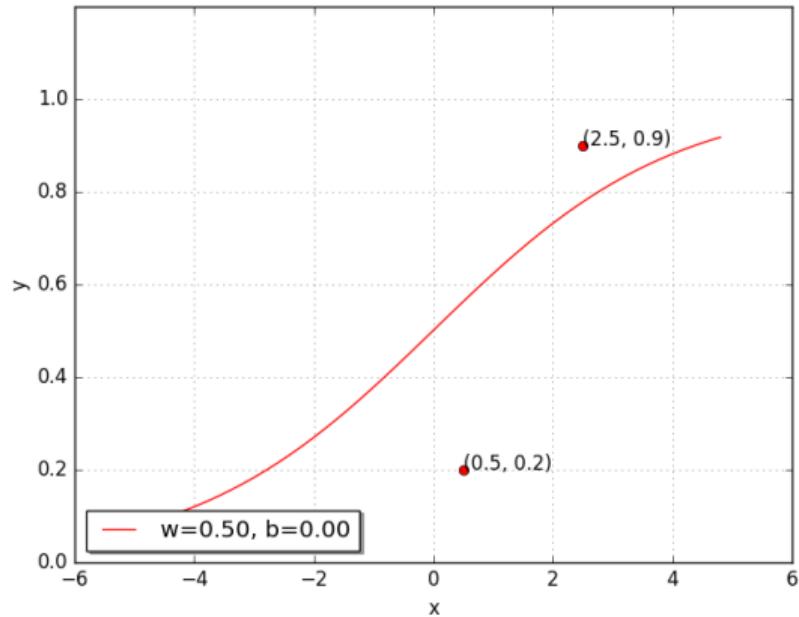


- Since we have only 2 points and 2 parameters ( $w, b$ ) we can easily plot  $\mathcal{L}(w, b)$  for different values of  $(w, b)$  and pick the one where  $\mathcal{L}(w, b)$  is minimum
- But of course this becomes intractable once you have many more data points and many more parameters !!
- Further,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

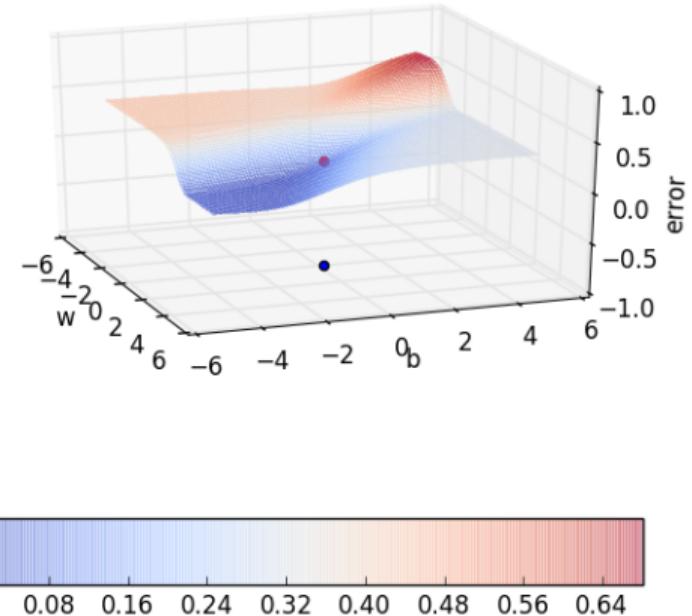
*Let us look at the geometric interpretation of our “guess work” algorithm in terms of this error surface*

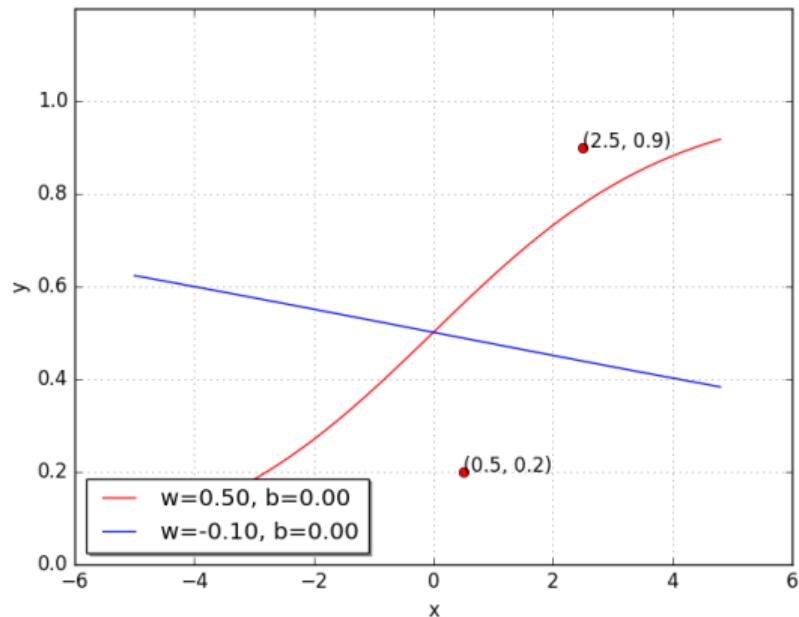
## Random search on error surface



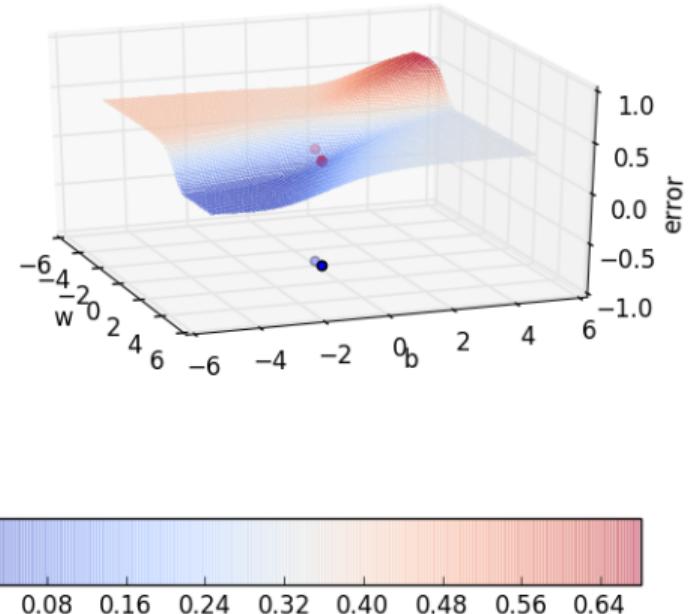


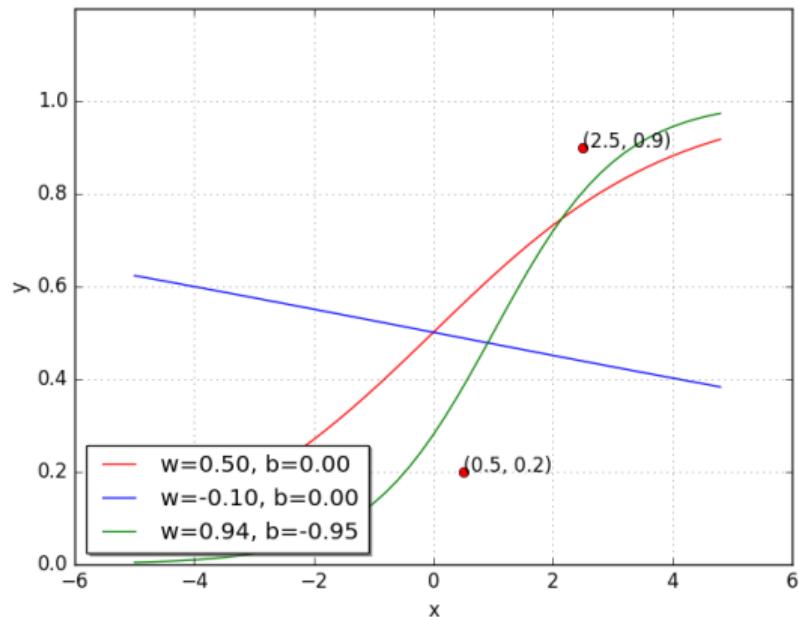
Random search on error surface



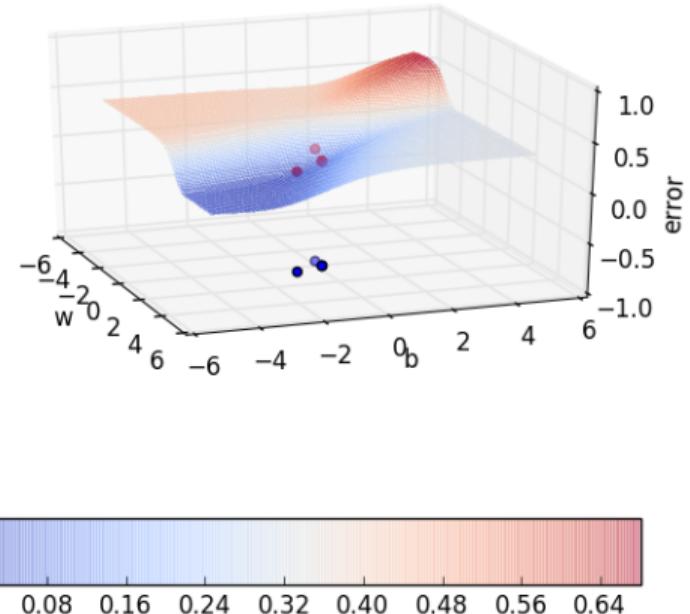


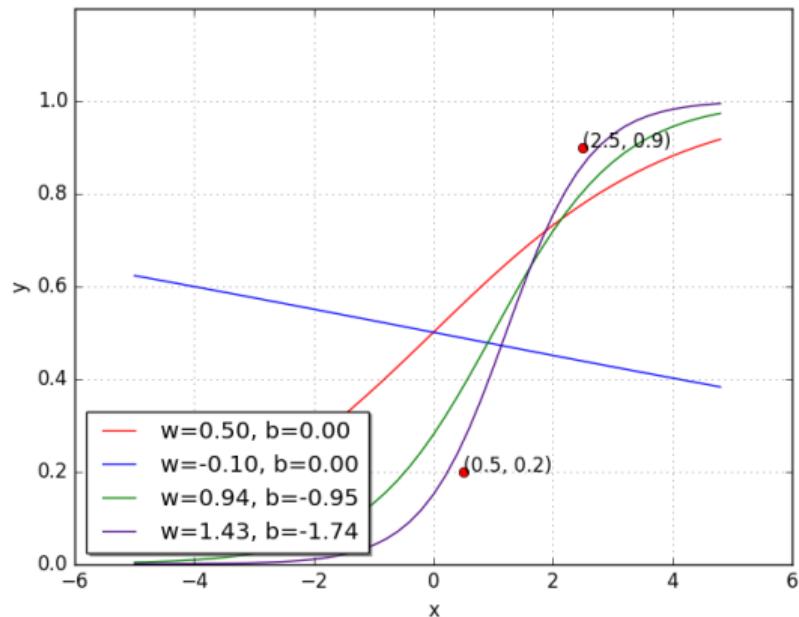
## Random search on error surface



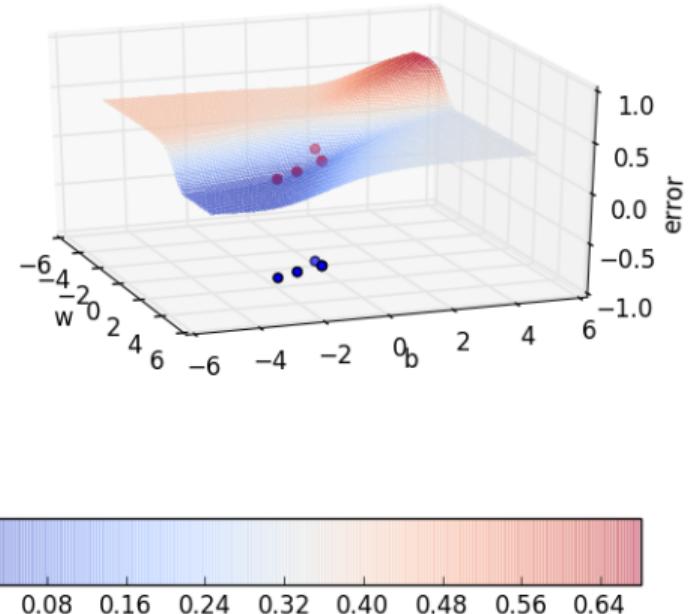


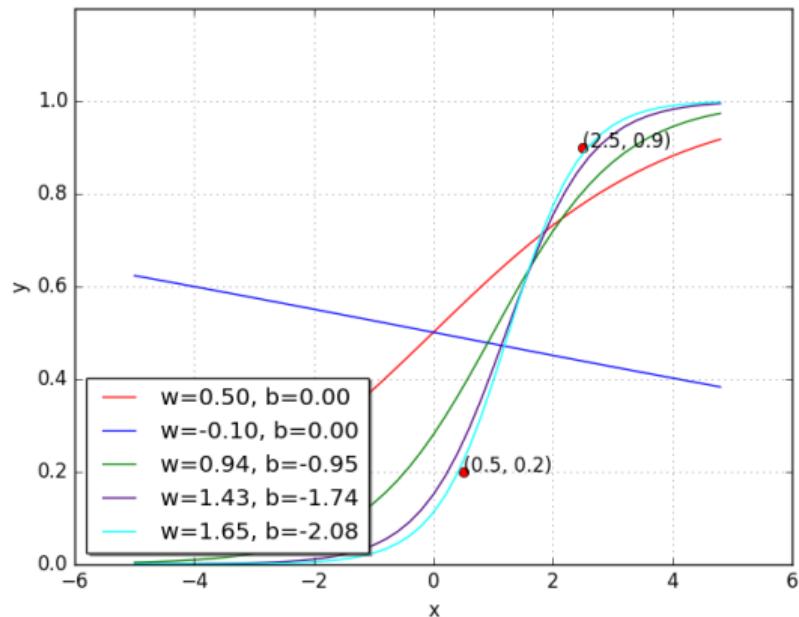
## Random search on error surface



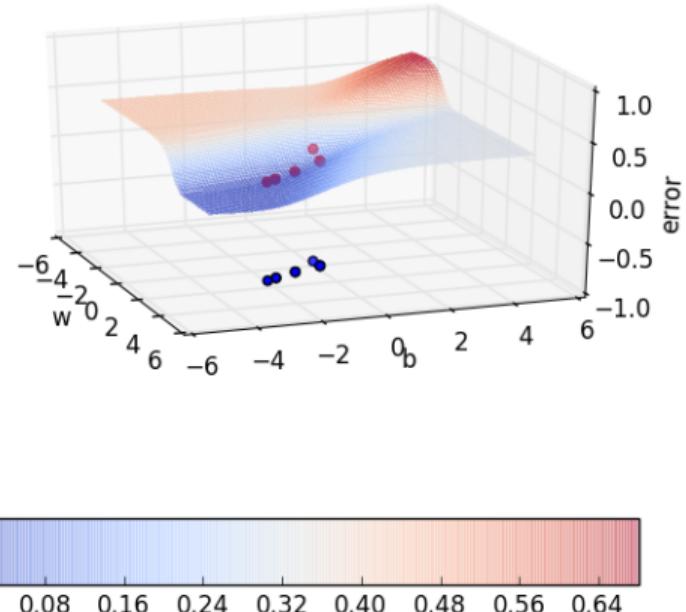


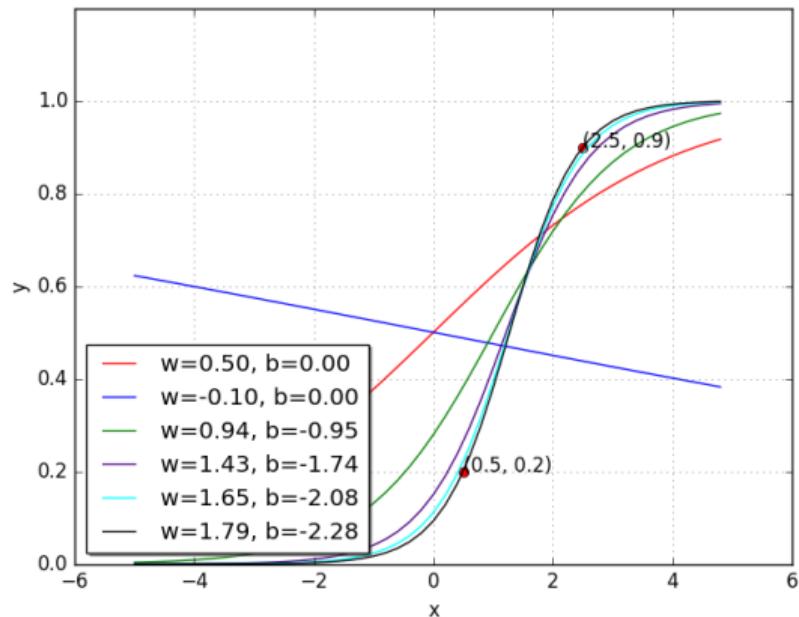
## Random search on error surface



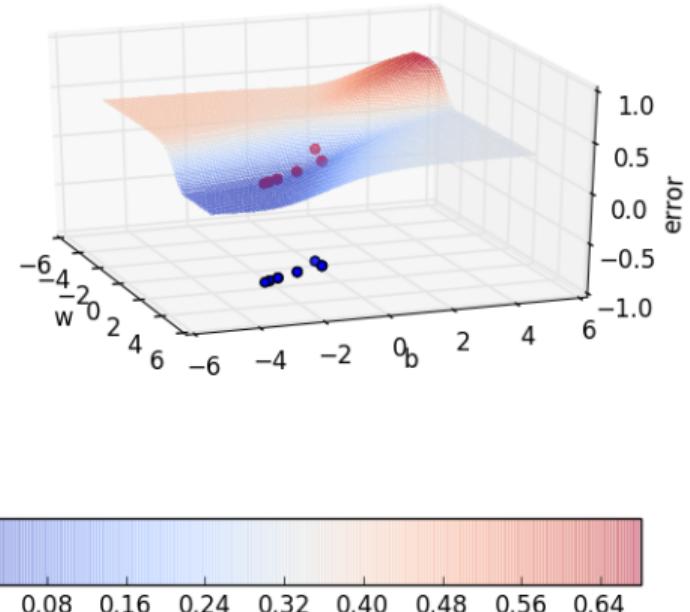


## Random search on error surface





## Random search on error surface



## Module 3.4: Learning Parameters : Gradient Descent

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

## Goal

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

vector of parameters,  
say, randomly initialized

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

vector of parameters,  
say, randomly initial-  
ized

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

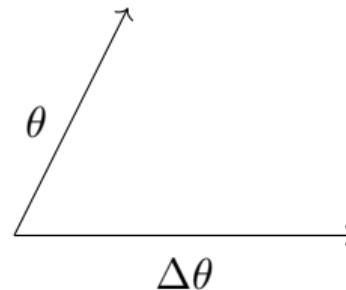
$$\Delta\theta = [\Delta w, \Delta b]$$

change in the  
values of w, b

vector of parameters,  
say, randomly initial-  
ized

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

change in the  
values of w, b

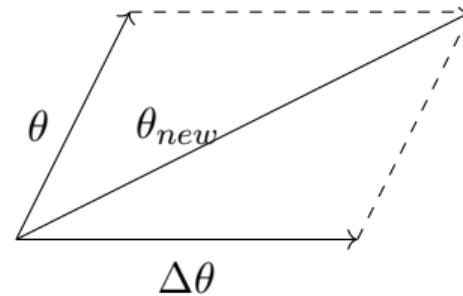


vector of parameters,  
say, randomly initialized

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

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

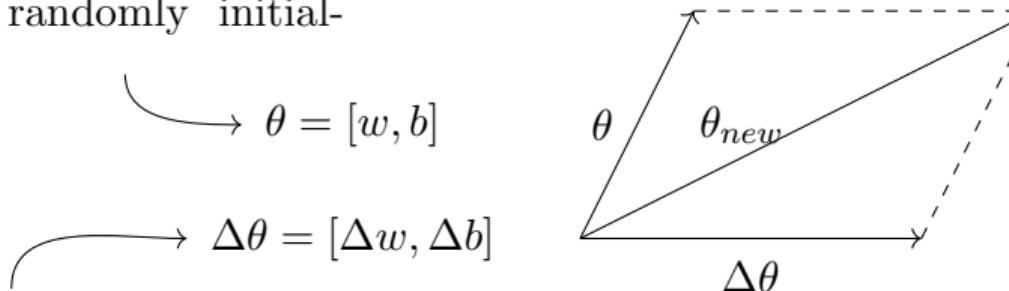
change in the  
values of w, b



vector of parameters,  
say, randomly initialized

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

change in the  
values of w, b

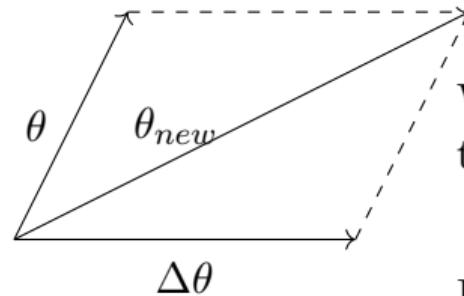


vector of parameters,  
say, randomly initialized

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

change in the  
values of w, b

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



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

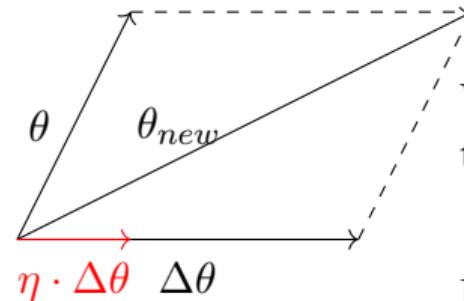
Let us be a bit conservative:  
move only by a small amount  $\eta$

vector of parameters,  
say, randomly initialized

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

change in the  
values of w, b

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



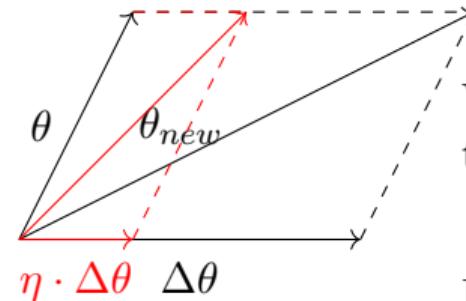
We moved in the direction of  $\Delta\theta$

Let us be a bit conservative:  
move only by a small amount  $\eta$

vector of parameters,  
say, randomly initialized

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

change in the  
values of w, b



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

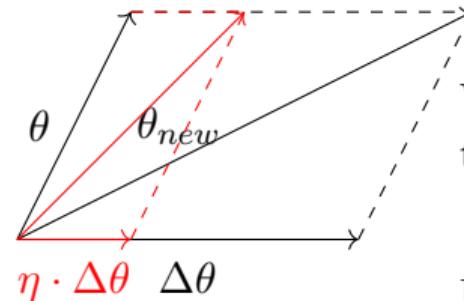
Let us be a bit conservative: move only by a small amount  $\eta$

vector of parameters,  
say, randomly initialized

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

change in the  
values of w, b

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



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

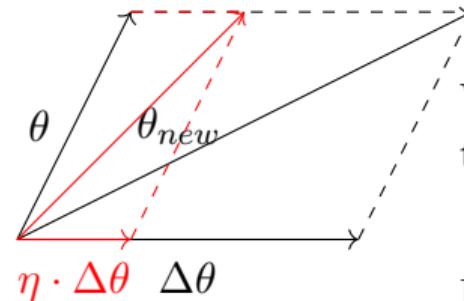
Let us be a bit conservative: move only by a small amount  $\eta$

vector of parameters,  
say, randomly initialized

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

change in the  
values of w, b

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



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

Let us be a bit conservative: move only by a small amount  $\eta$

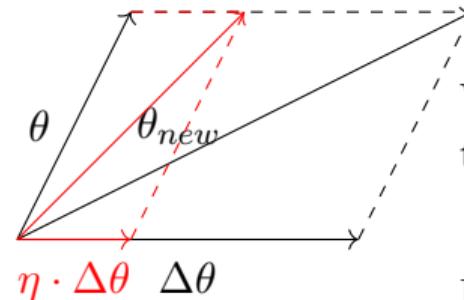
**Question:** What is the right  $\Delta\theta$  to use?  
?

vector of parameters,  
say, randomly initialized

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

change in the  
values of w, b

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



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

Let us be a bit conservative: move only by a small amount  $\eta$

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

**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,

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

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

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

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

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

Note that the move ( $\eta u$ ) would be favorable only if,

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

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

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

Note that the move ( $\eta u$ ) would be favorable only if,

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

This implies,

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

Okay, so we have,

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

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

Okay, so we have,

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

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

Okay, so we have,

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

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

Let  $\beta$  be the angle between  $u^T$  and  $\nabla \mathcal{L}(\theta)$ , then we know that,

Okay, so we have,

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

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

Let  $\beta$  be the angle between  $u^T$  and  $\nabla \mathcal{L}(\theta)$ , then we know that,

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

Okay, so we have,

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

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

Let  $\beta$  be the angle between  $u^T$  and  $\nabla \mathcal{L}(\theta)$ , then we know that,

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

Okay, so we have,

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

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

Let  $\beta$  be the angle between  $u^T$  and  $\nabla \mathcal{L}(\theta)$ , then we know that,

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

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

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

Okay, so we have,

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

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

Let  $\beta$  be the angle between  $u^T$  and  $\nabla \mathcal{L}(\theta)$ , then we know that,

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

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

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

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

## Gradient Descent Rule

- The direction  $u$  that we intend to move in should be at  $180^\circ$  w.r.t. the gradient

## Gradient Descent Rule

- The direction  $u$  that we intend to move in should be at  $180^\circ$  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  $180^\circ$  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,  $\nabla w_t = \frac{\partial \mathcal{L}(w, b)}{\partial w}$  at  $w = w_t, b = b_t$ ,  $\nabla b = \frac{\partial \mathcal{L}(w, b)}{\partial b}$  at  $w = w_t, b = b_t$

## Gradient Descent Rule

- The direction  $u$  that we intend to move in should be at  $180^\circ$  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,  $\nabla w_t = \frac{\partial \mathcal{L}(w, b)}{\partial w}$  at  $w = w_t, b = b_t$ ,  $\nabla b = \frac{\partial \mathcal{L}(w, b)}{\partial b}$  at  $w = w_t, b = b_t$

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

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

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

---

**Algorithm:** gradient\_descent()

---

```
t ← 0;  
max_iterations ← 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
```

---

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

---

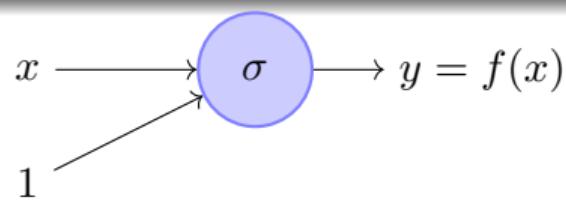
**Algorithm:** gradient\_descent()

---

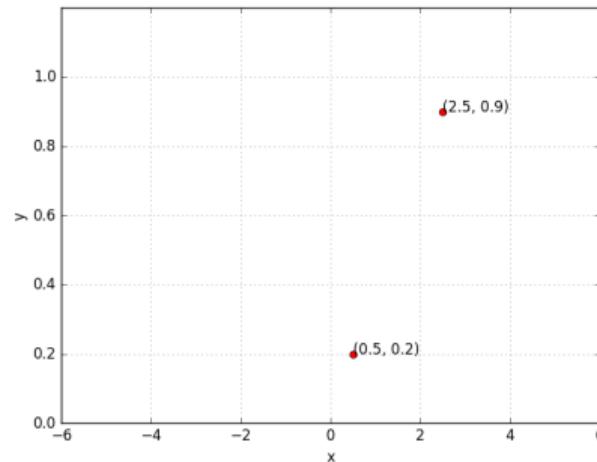
```
t ← 0;  
max_iterations ← 1000;  
while  $t < max\_iterations$  do  
    |  $w_{t+1} \leftarrow w_t - \eta \nabla w_t;$   
    |  $b_{t+1} \leftarrow b_t - \eta \nabla b_t;$   
end
```

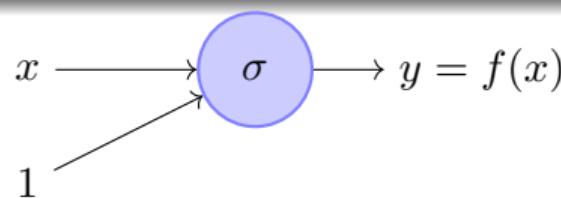
---

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



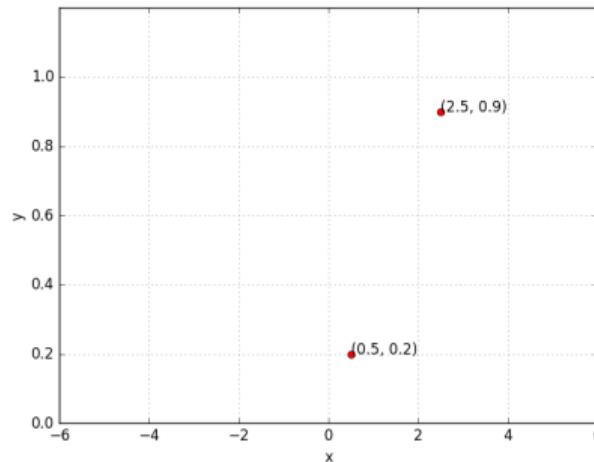
$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

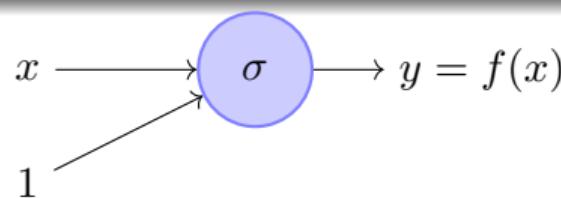




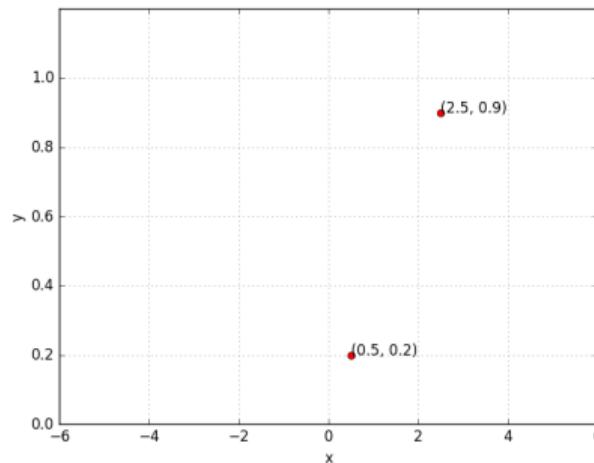
$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

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



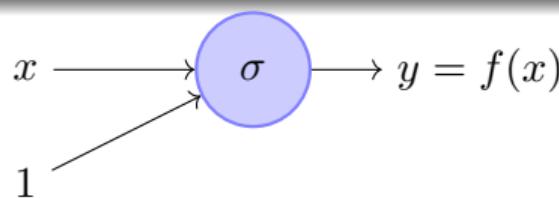


$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

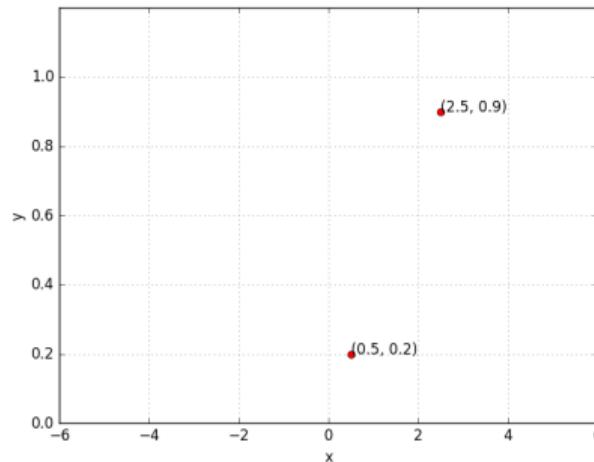


Let's assume there is only 1 point to fit  
 $(x, y)$

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



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



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

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

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

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

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

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

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

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

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

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

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

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

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

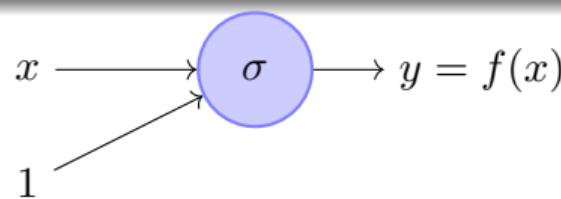
$$\begin{aligned}
 &\frac{\partial}{\partial w} \left( \frac{1}{1 + e^{-(wx+b)}} \right) \\
 &= \frac{-1}{(1 + e^{-(wx+b)})^2} \frac{\partial}{\partial w} (e^{-(wx+b)}) \\
 &= \frac{-1}{(1 + e^{-(wx+b)})^2} * (e^{-(wx+b)}) \frac{\partial}{\partial w} (-wx - b) \\
 &= \frac{-1}{(1 + e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1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

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

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

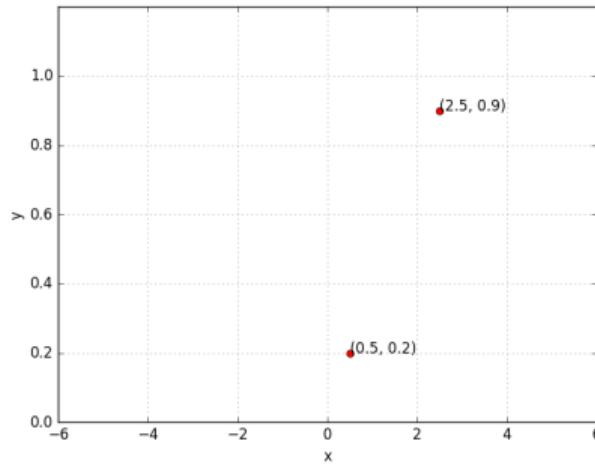
$$\begin{aligned}
\nabla w &= \frac{\partial}{\partial w} \left[ \frac{1}{2} * (f(x) - y)^2 \right] \\
&= \frac{1}{2} * [2 * (f(x) - y) * \frac{\partial}{\partial w} (f(x) - y)] \\
&= (f(x) - y) * \frac{\partial}{\partial w} (f(x)) \\
&= (f(x) - y) * \frac{\partial}{\partial w} \left( \frac{1}{1 + e^{-(wx+b)}}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

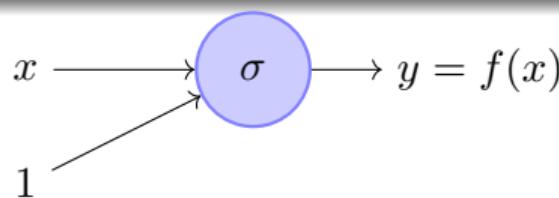
$$\begin{aligned}
&\frac{\partial}{\partial w} \left( \frac{1}{1 + e^{-(wx+b)}} \right) \\
&= \frac{-1}{(1 + e^{-(wx+b)})^2} \frac{\partial}{\partial w} (e^{-(wx+b)}) \\
&= \frac{-1}{(1 + e^{-(wx+b)})^2} * (e^{-(wx+b)}) \frac{\partial}{\partial w} (-wx - b) \\
&= \frac{-1}{(1 + e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1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



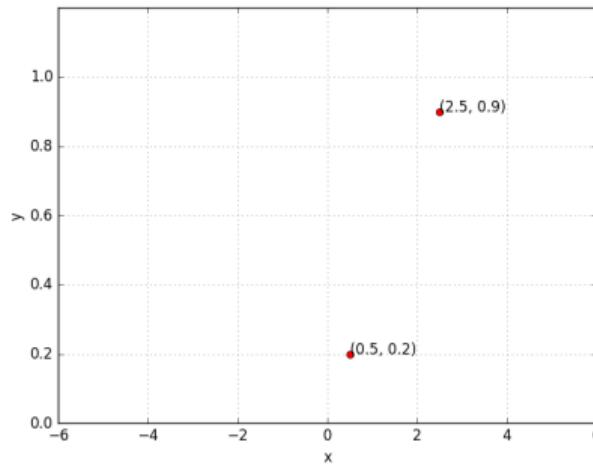
$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

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



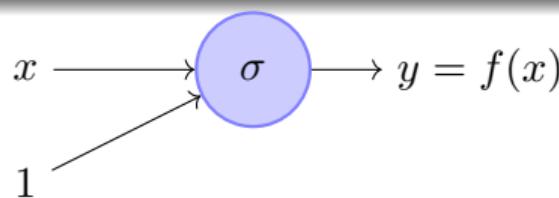


$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$

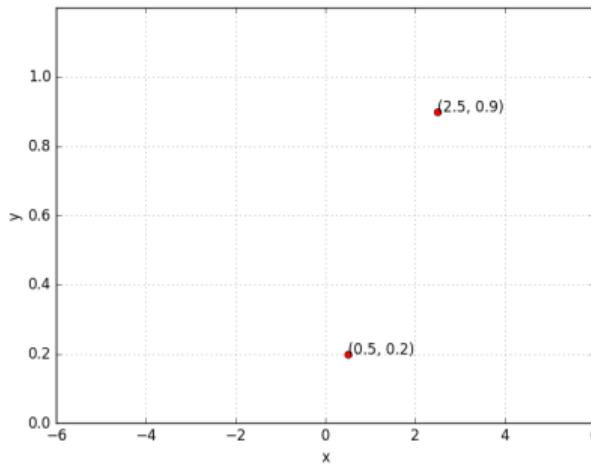


So if there is only 1 point  $(x, y)$ , we have,

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



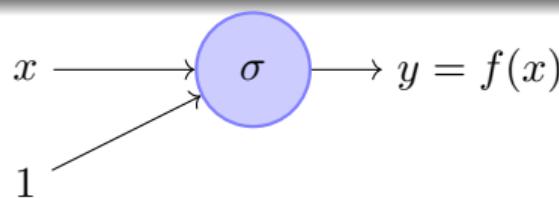
$$f(x) = \frac{1}{1+e^{-(w \cdot x + b)}}$$



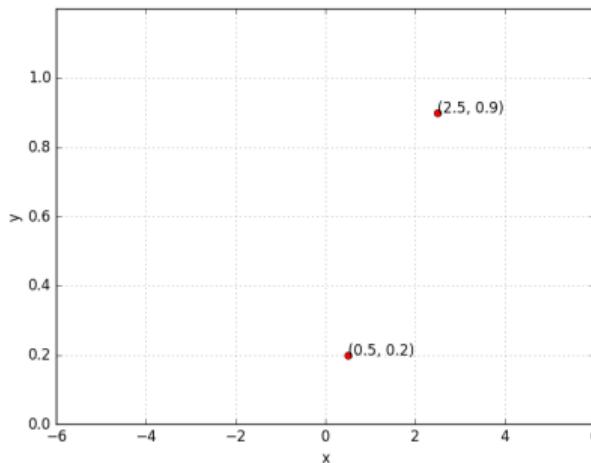
So if there is only 1 point  $(x, y)$ , we have,

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

For two points,



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

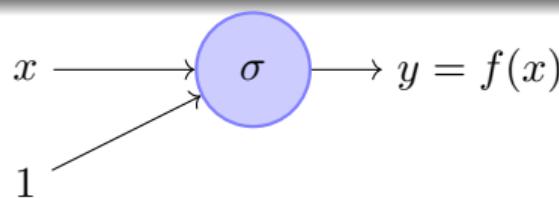


So if there is only 1 point  $(x, y)$ , we have,

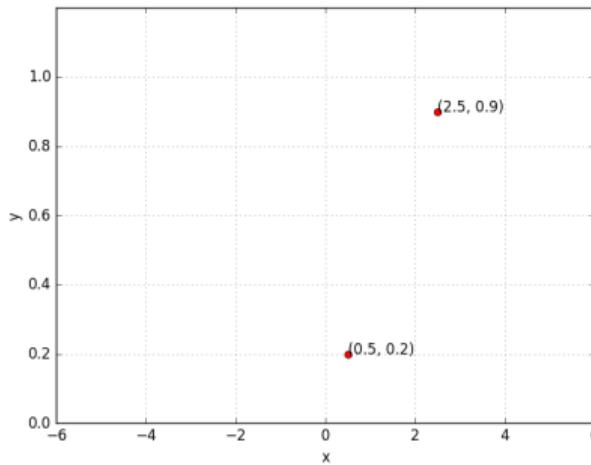
$$\nabla w = (f(x) - y) * f(x) * (1 - f(x)) * x$$

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]
```

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

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

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

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

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

```

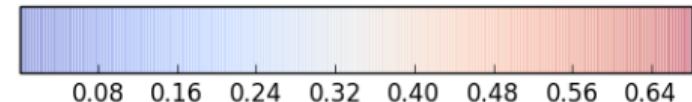
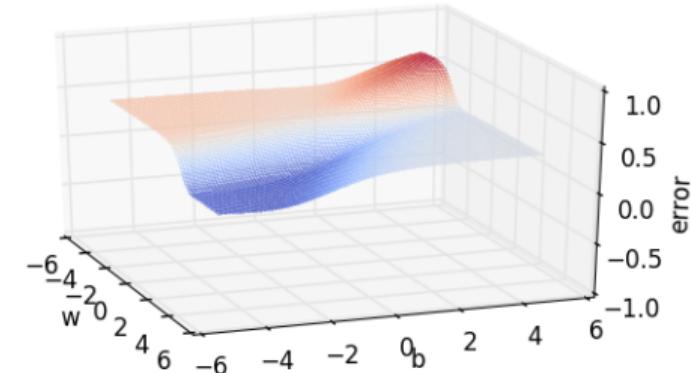
X = [0.5, 2.5]
Y = [0.2, 0.9]

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

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

```

Random search on error surface



```

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

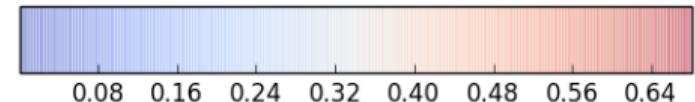
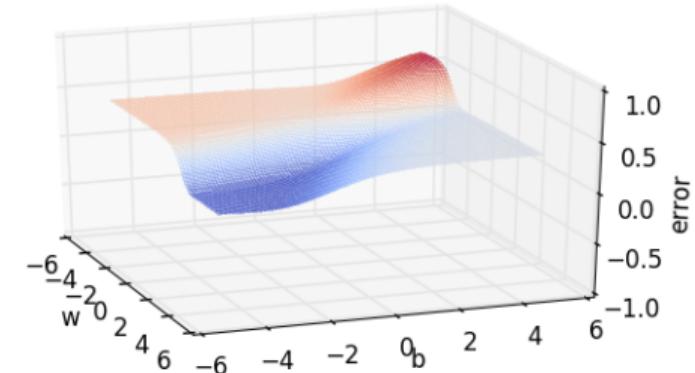
def f(w,b,x) : #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x + b)))

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)

```

Random search on 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)))

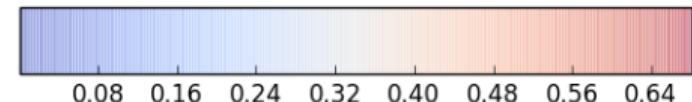
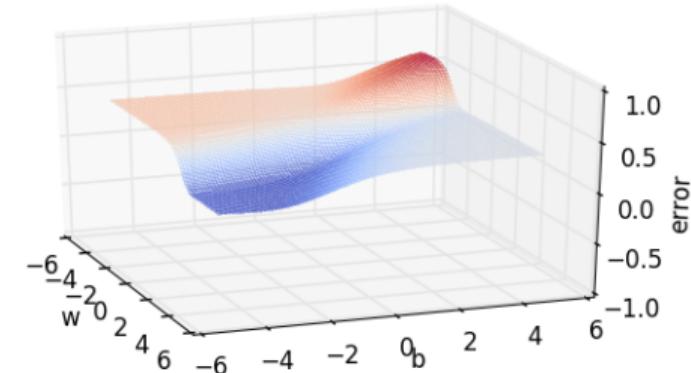
def error (w, b) :
    err = 0.0
    for x,y in zip(X,Y) :
        fx = f(w,b,x)
        err += 0.5 * (fx - y) ** 2
    return err

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

```

Random search on 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

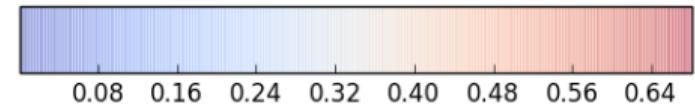
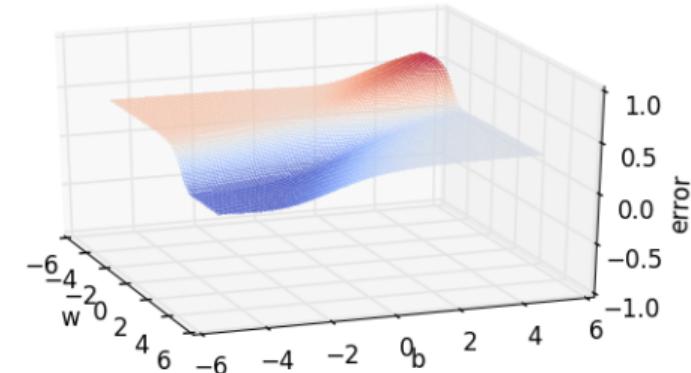
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

Random search on 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

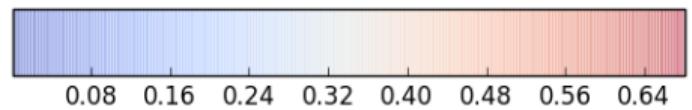
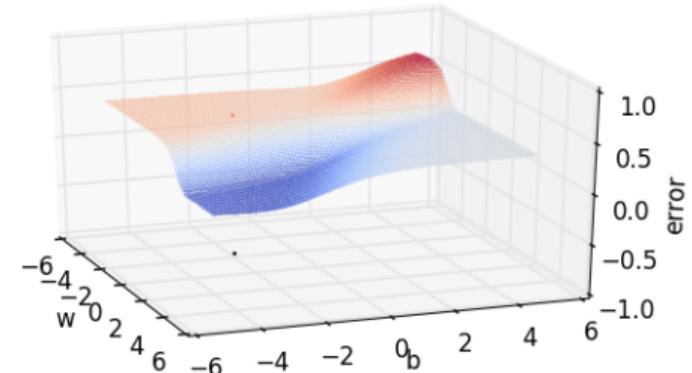
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

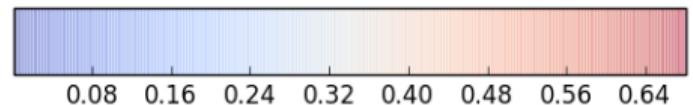
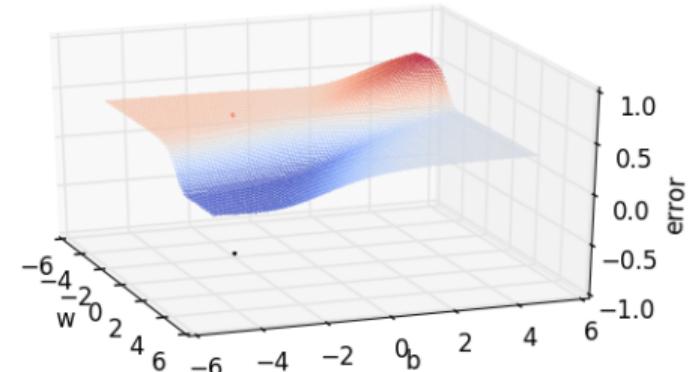
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

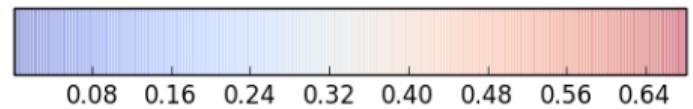
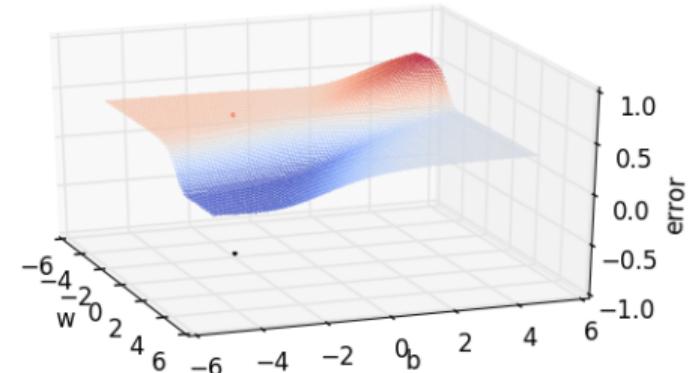
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

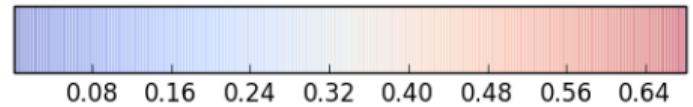
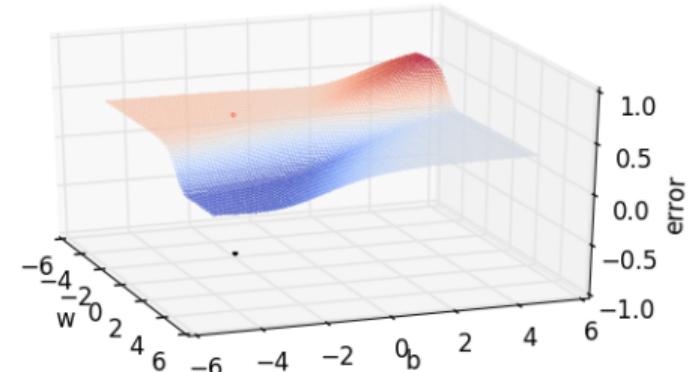
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

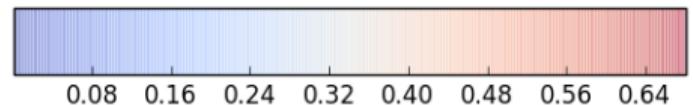
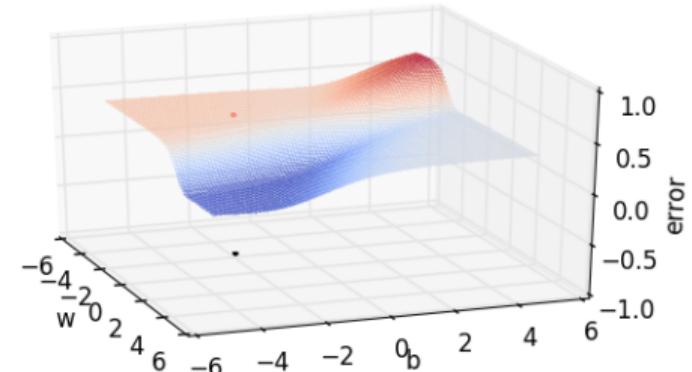
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

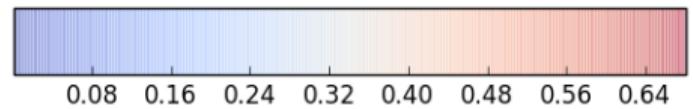
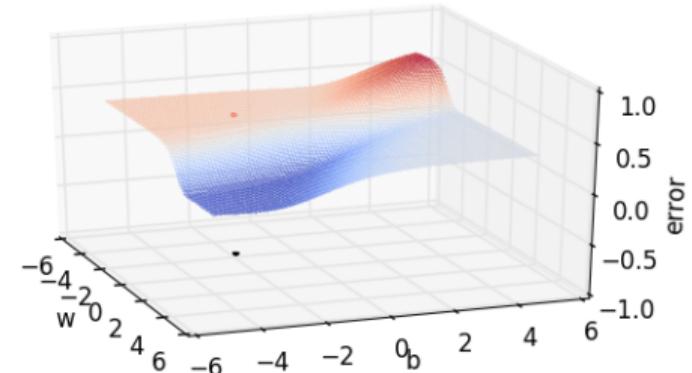
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

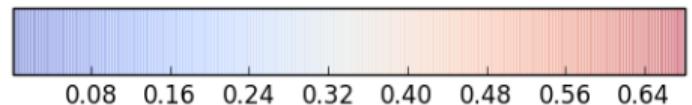
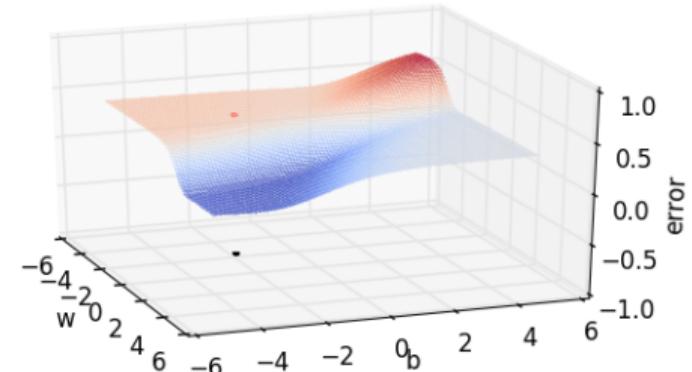
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

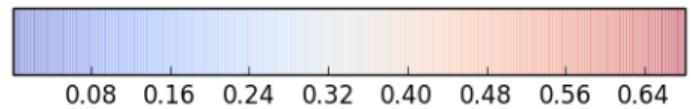
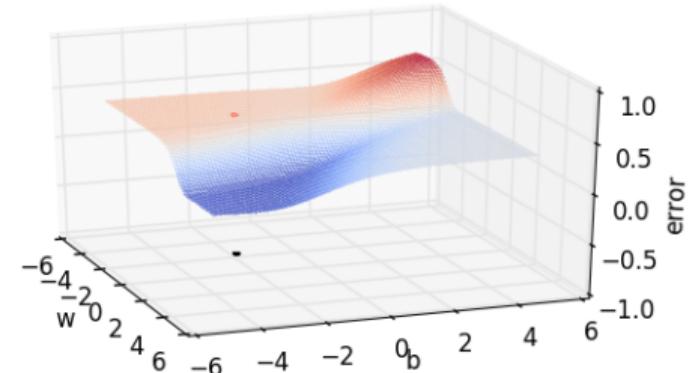
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

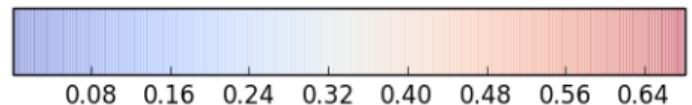
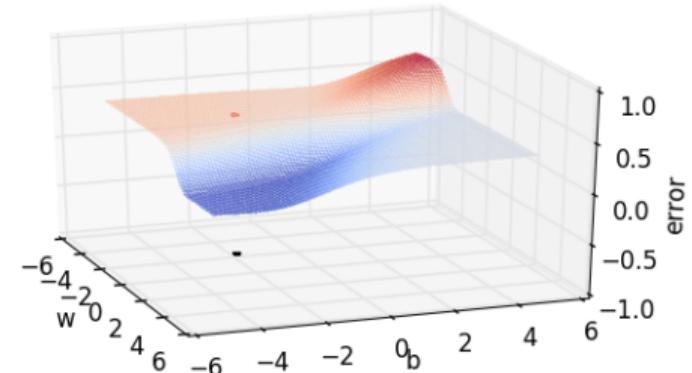
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

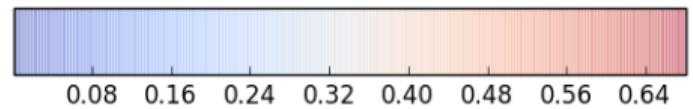
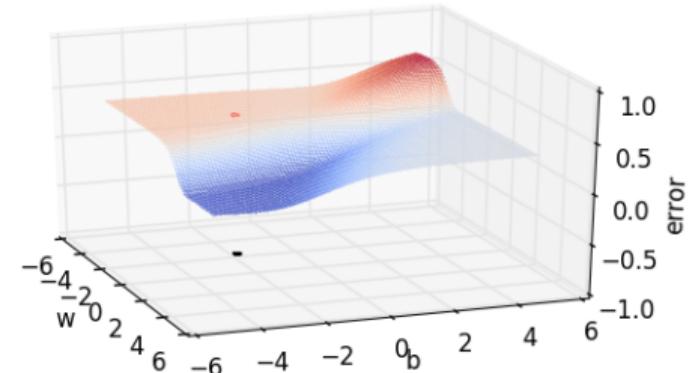
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

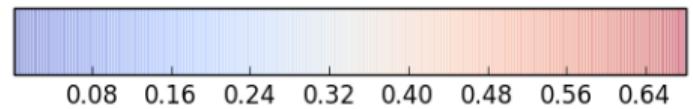
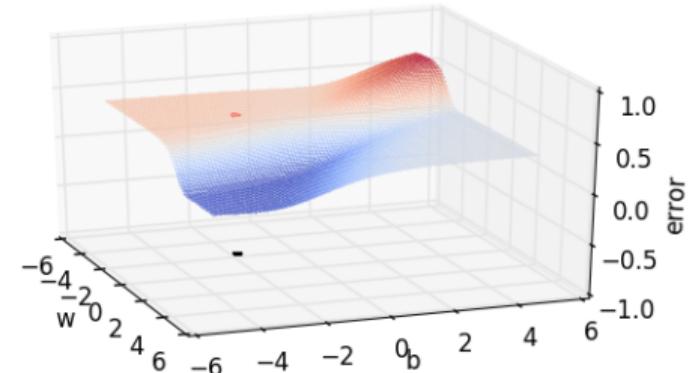
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

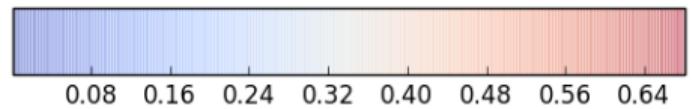
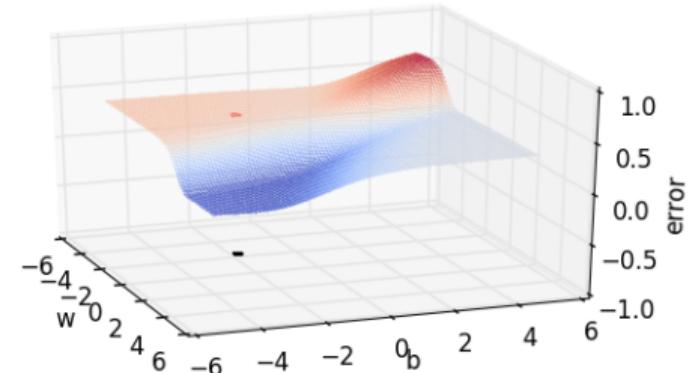
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

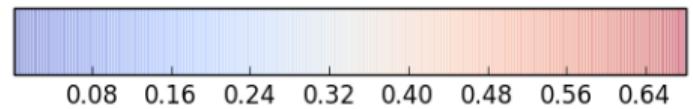
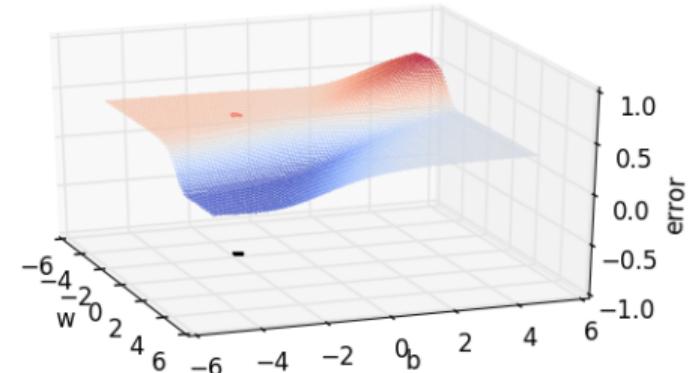
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

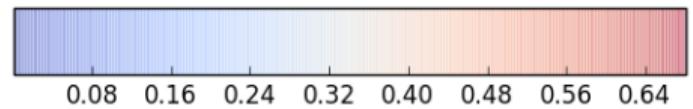
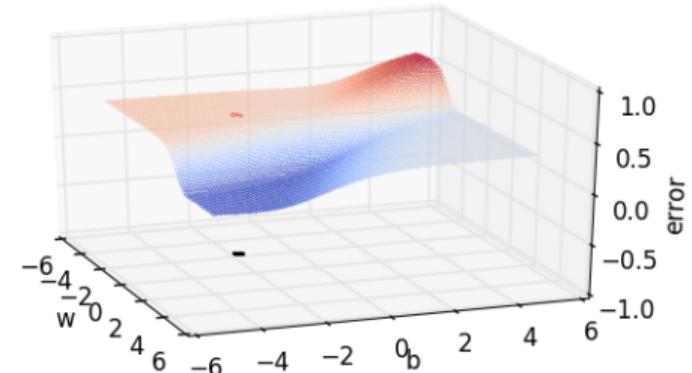
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

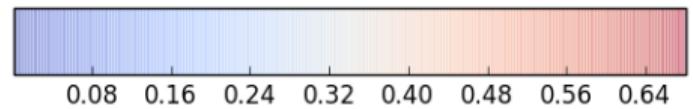
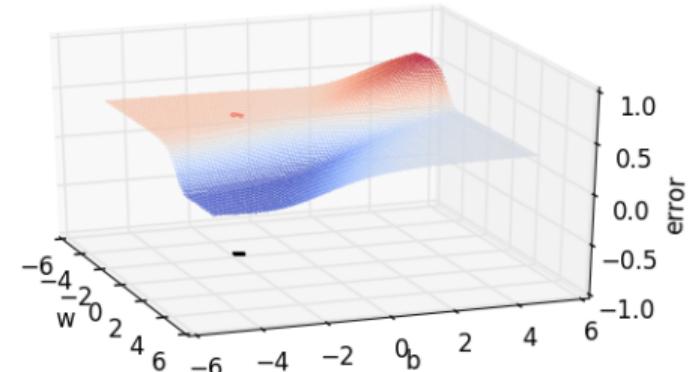
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

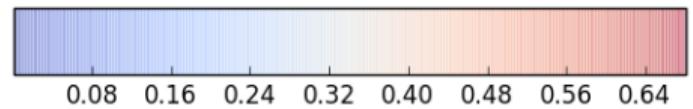
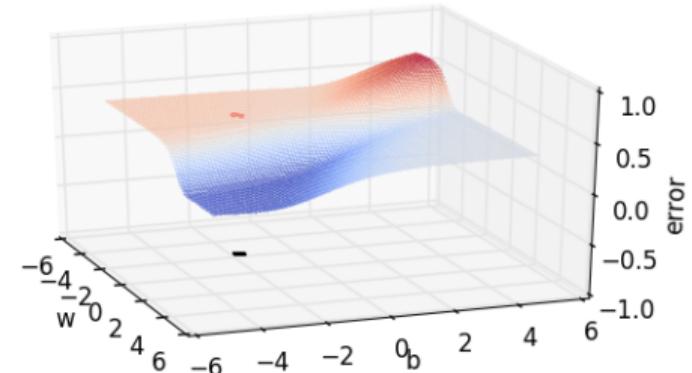
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

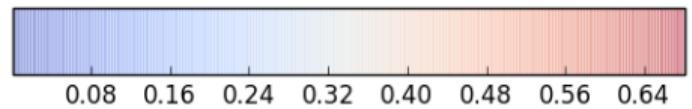
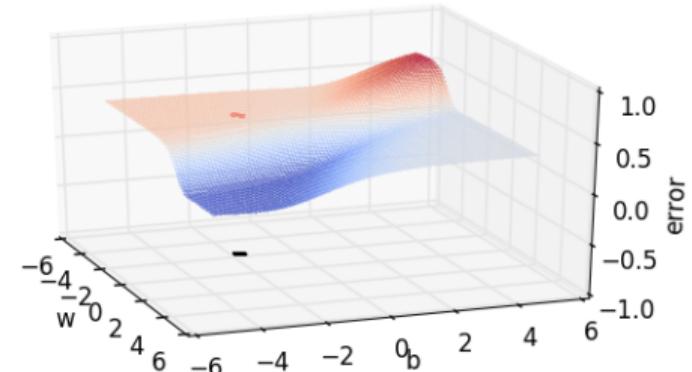
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

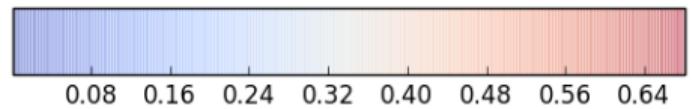
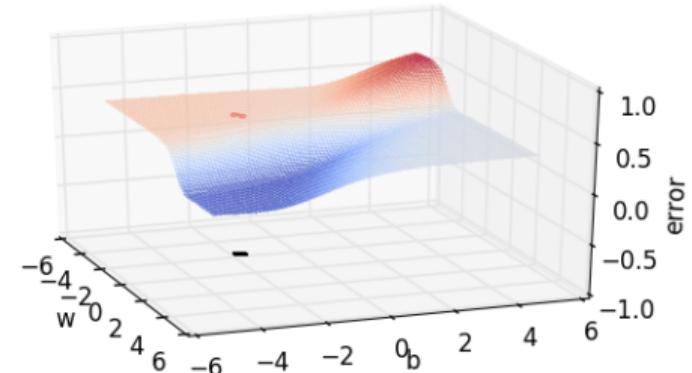
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

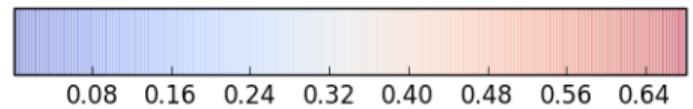
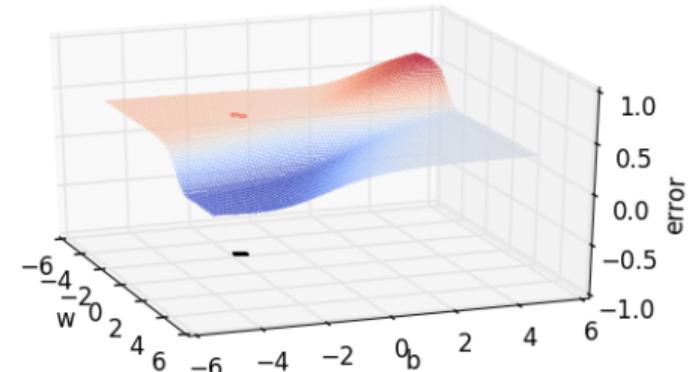
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

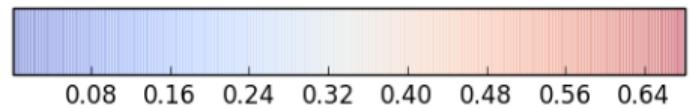
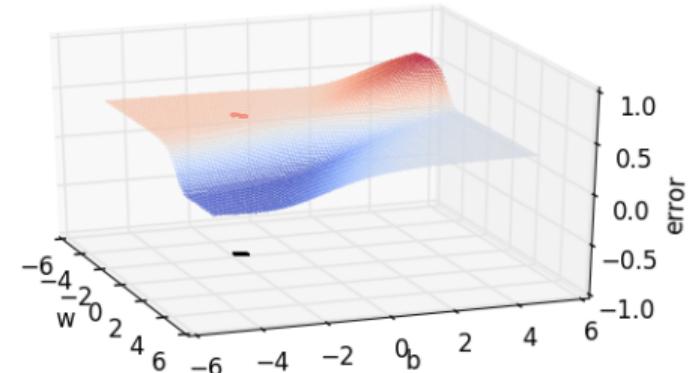
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

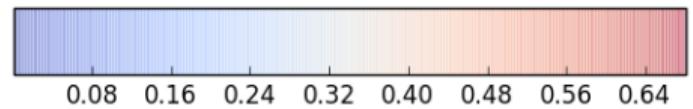
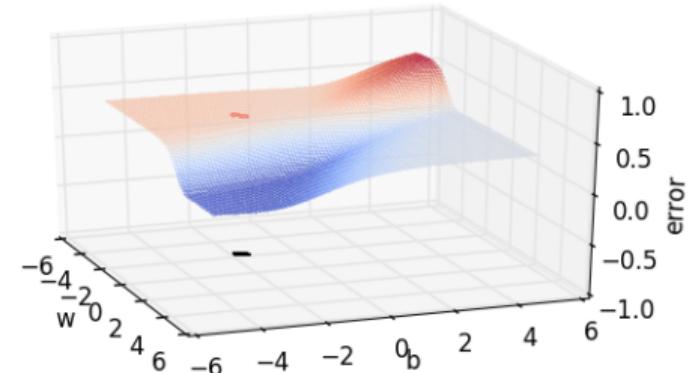
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

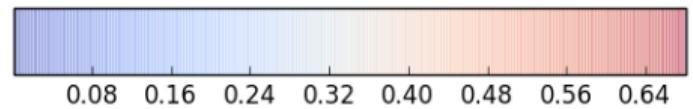
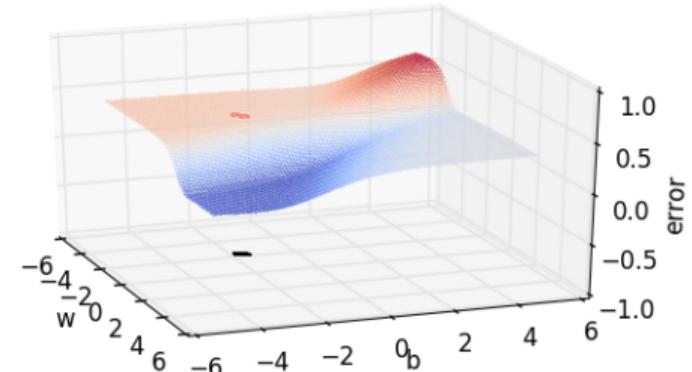
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

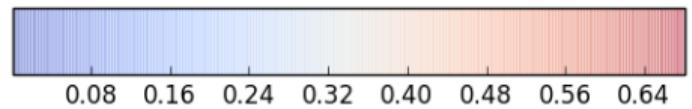
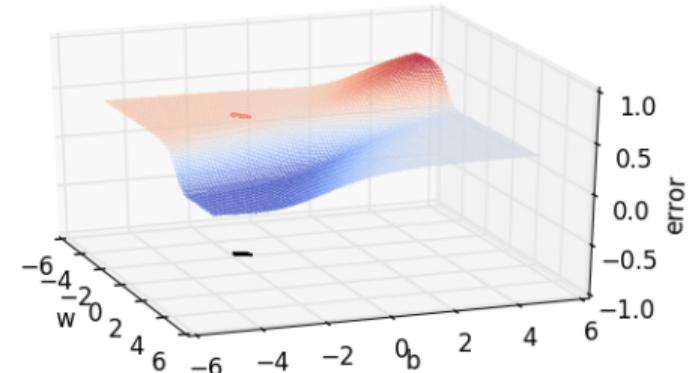
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

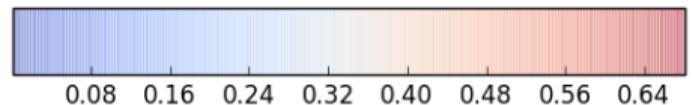
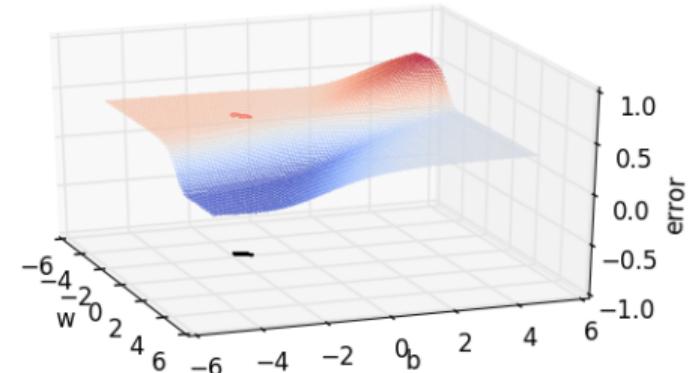
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

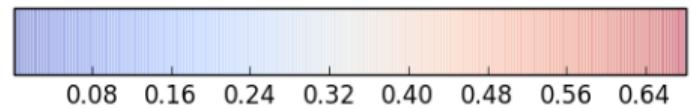
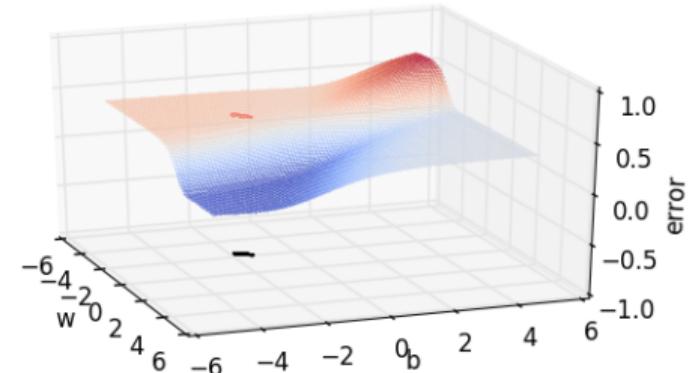
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

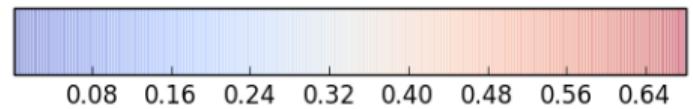
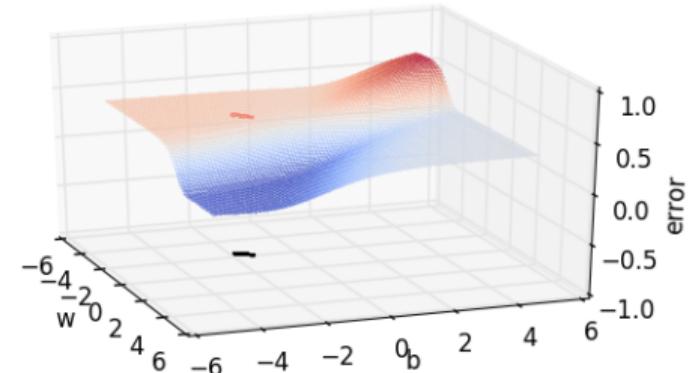
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

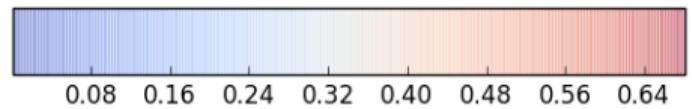
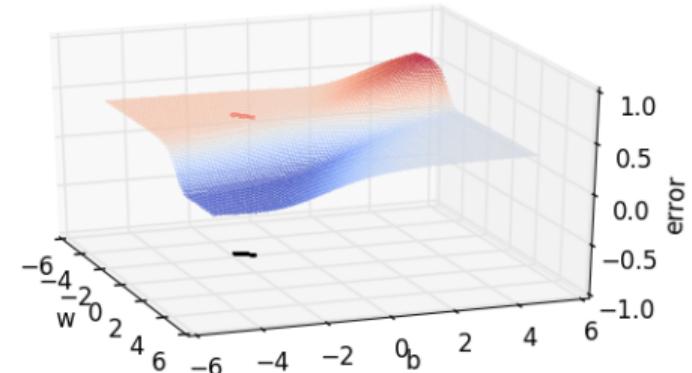
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

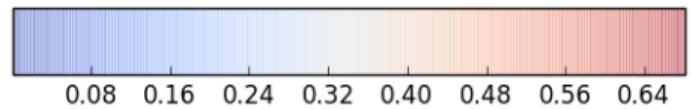
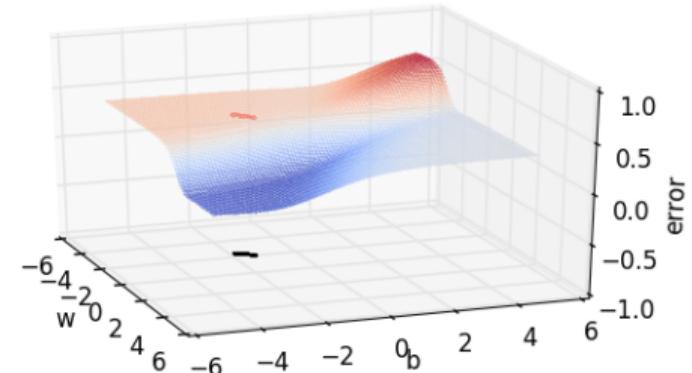
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

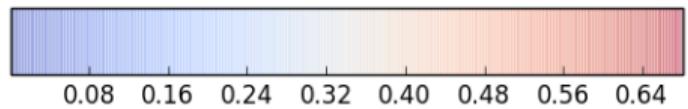
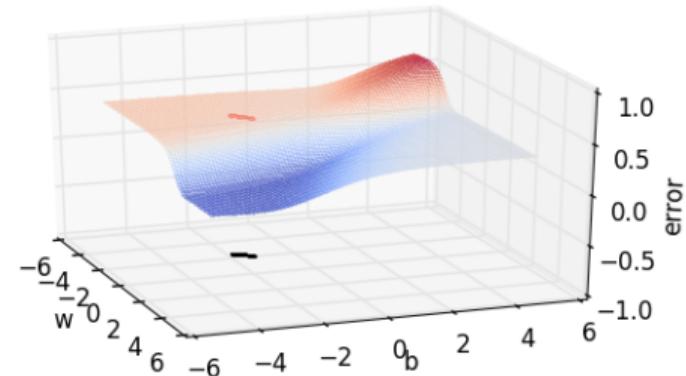
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

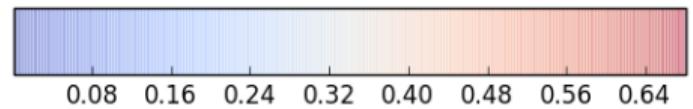
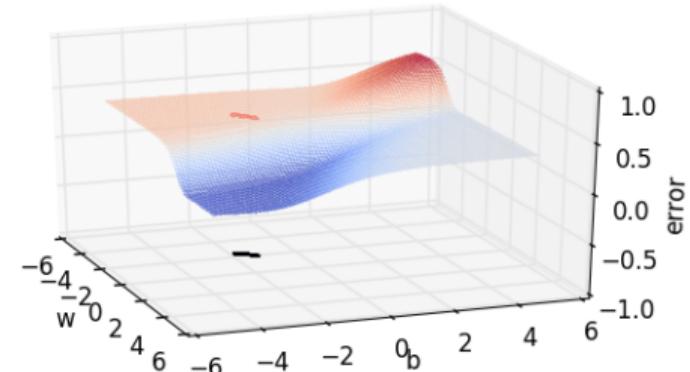
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

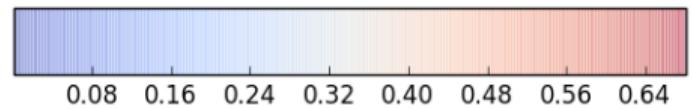
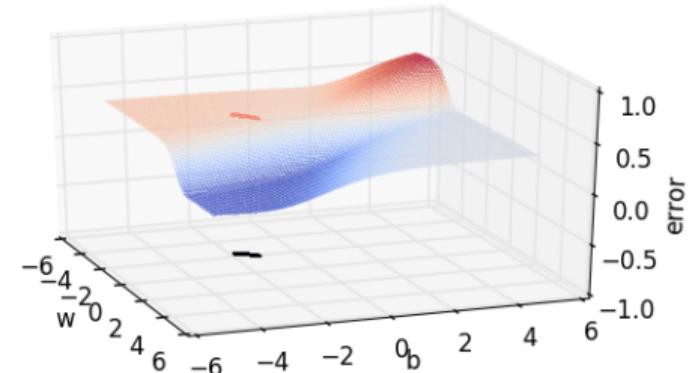
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

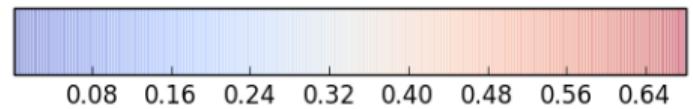
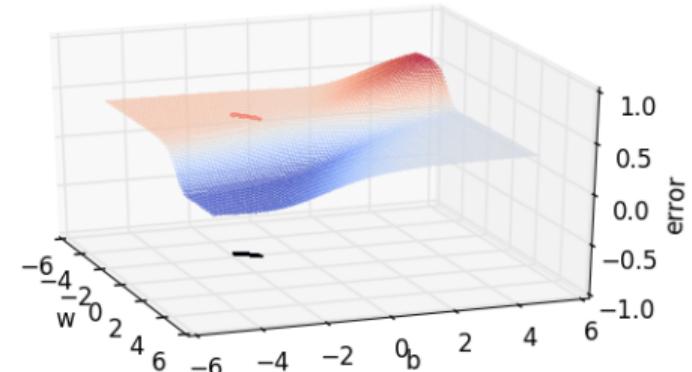
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

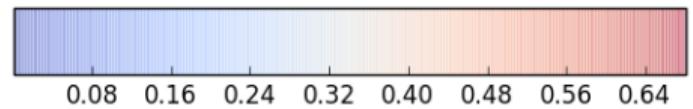
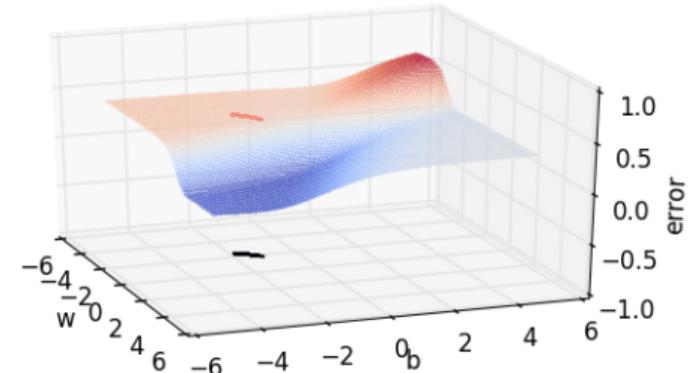
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

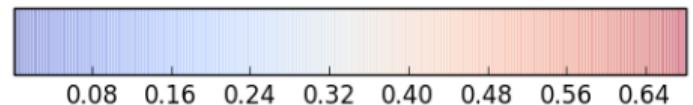
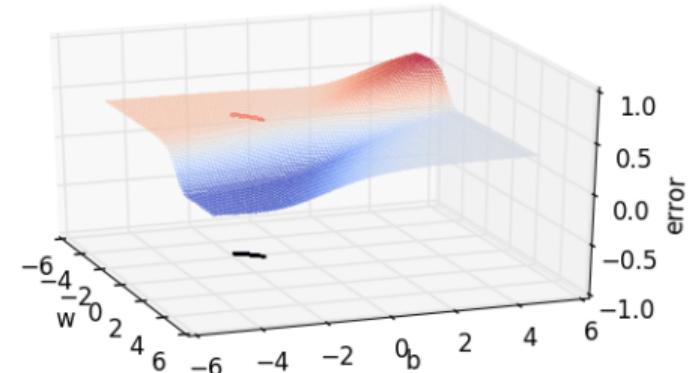
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

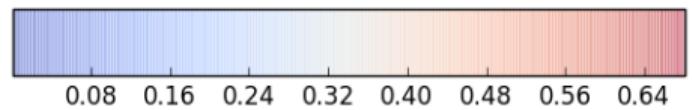
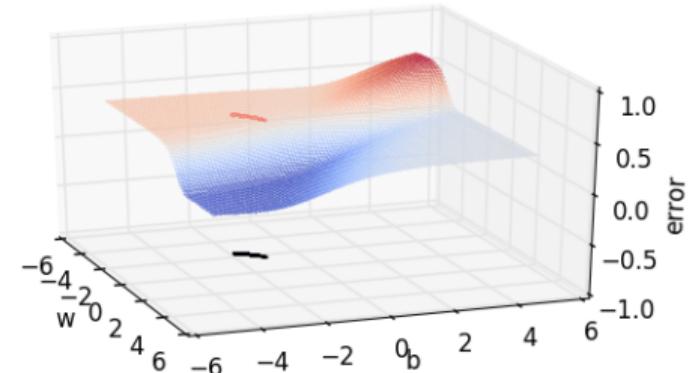
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

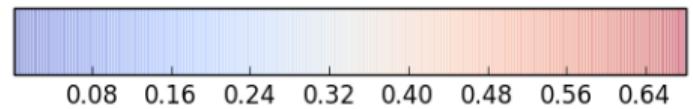
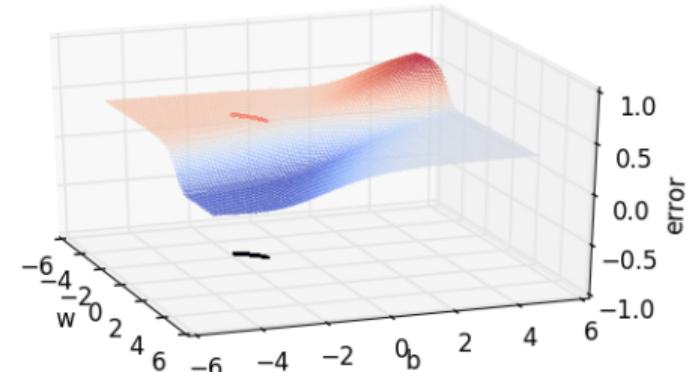
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

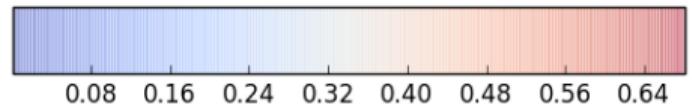
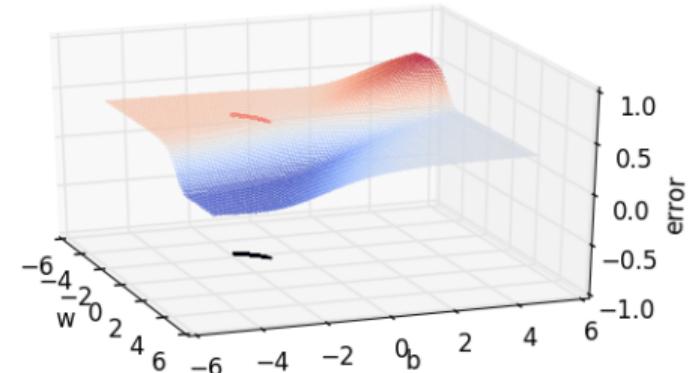
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

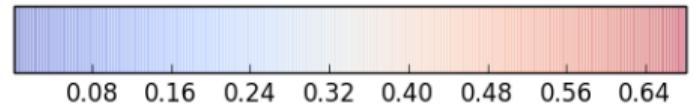
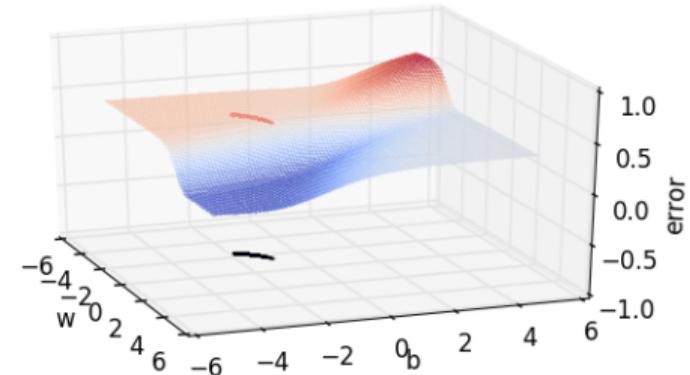
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

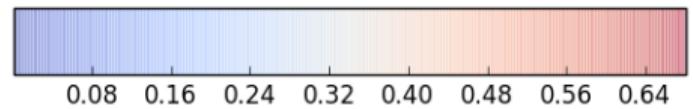
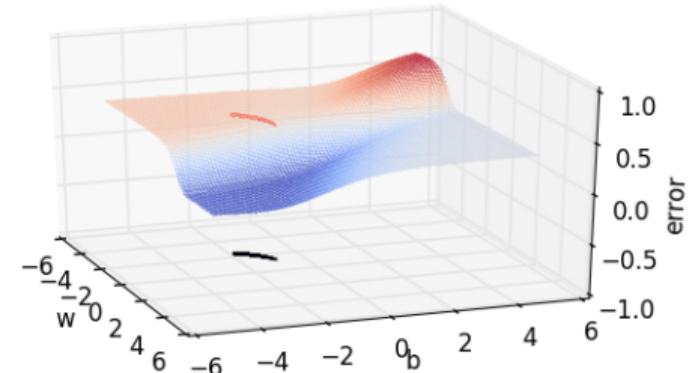
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

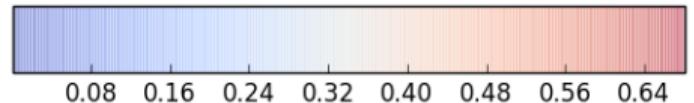
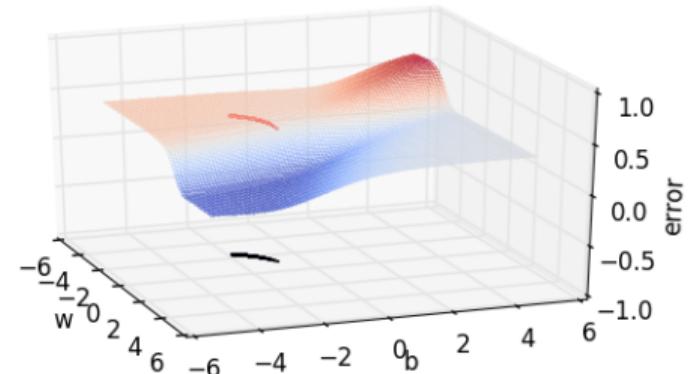
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

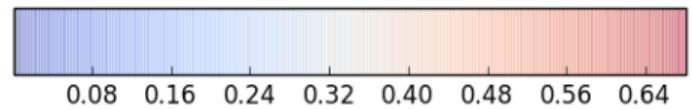
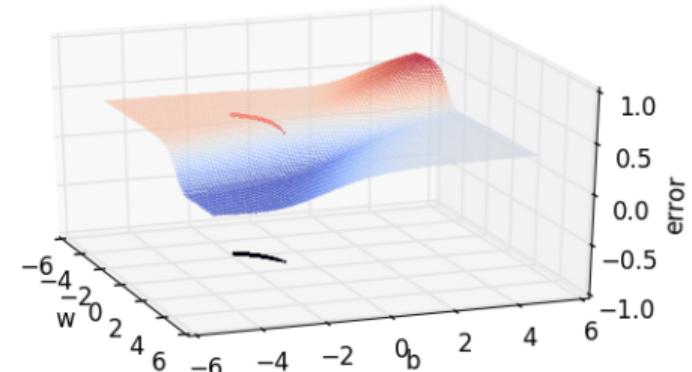
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

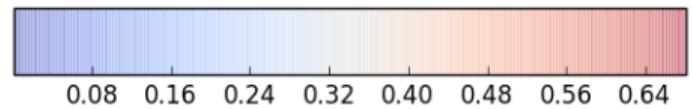
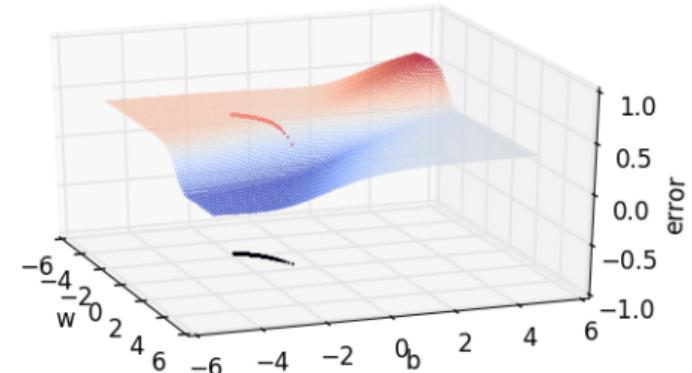
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

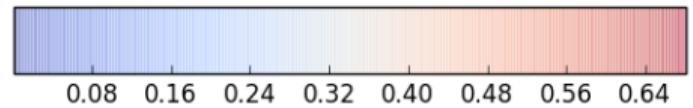
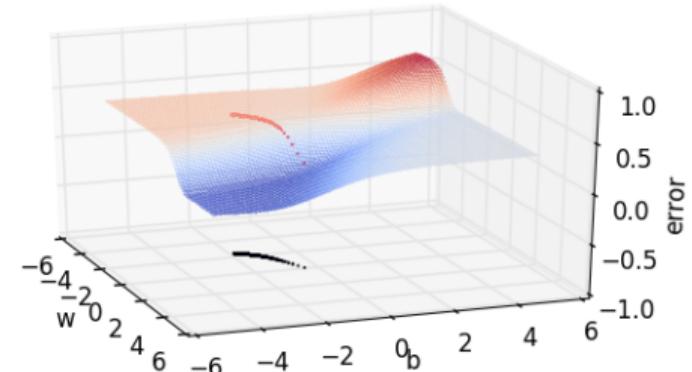
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

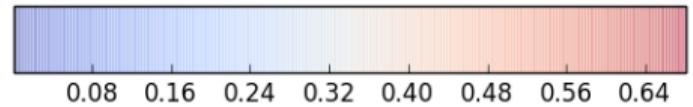
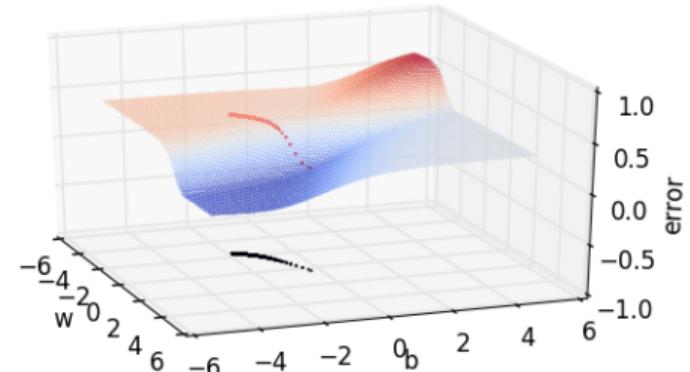
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

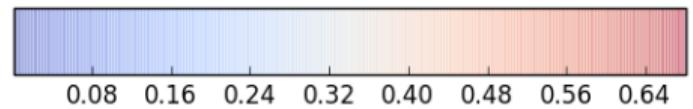
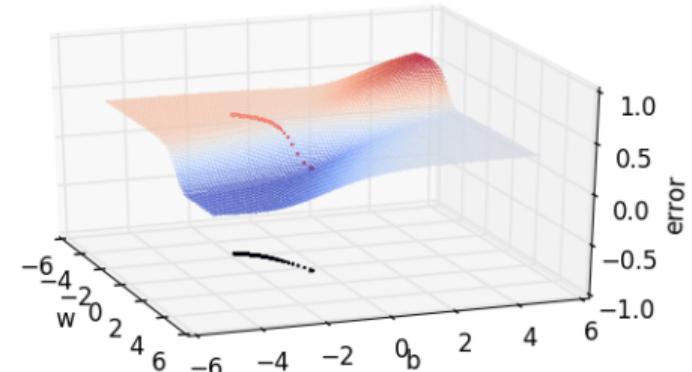
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

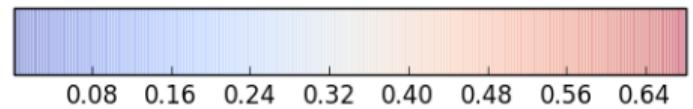
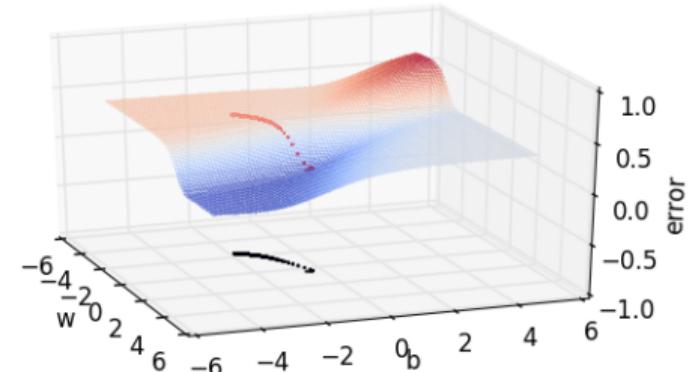
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

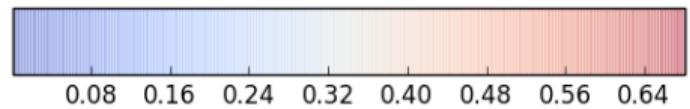
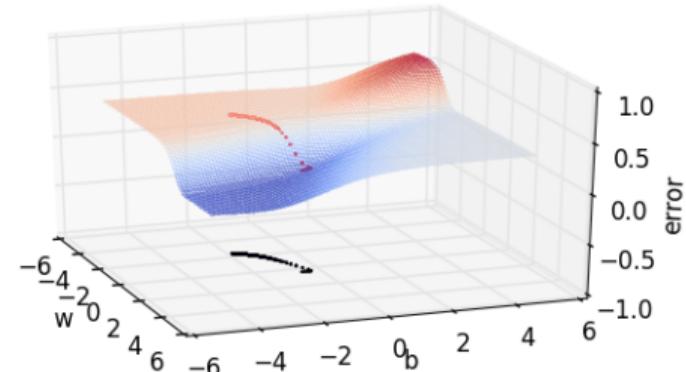
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

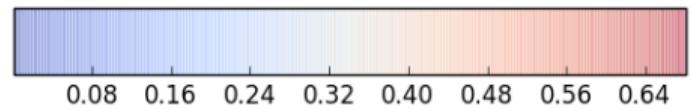
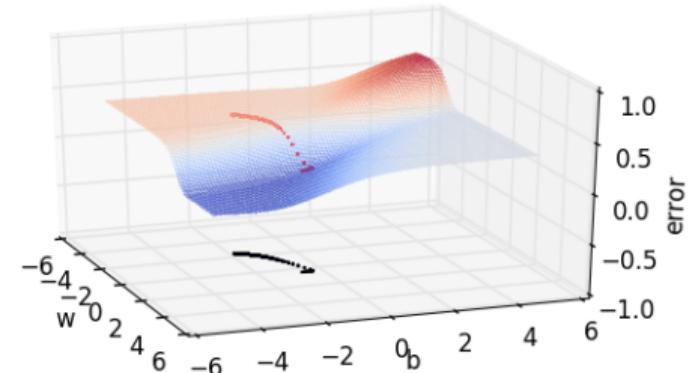
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

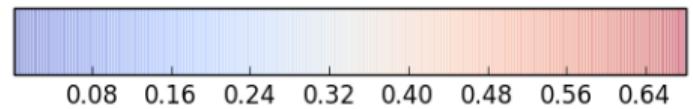
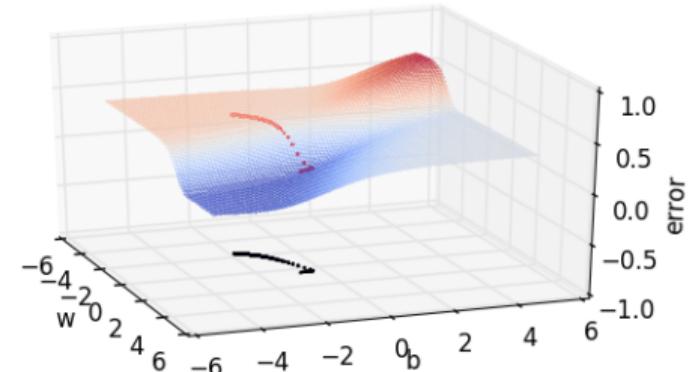
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

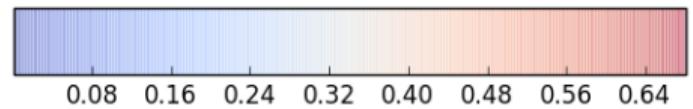
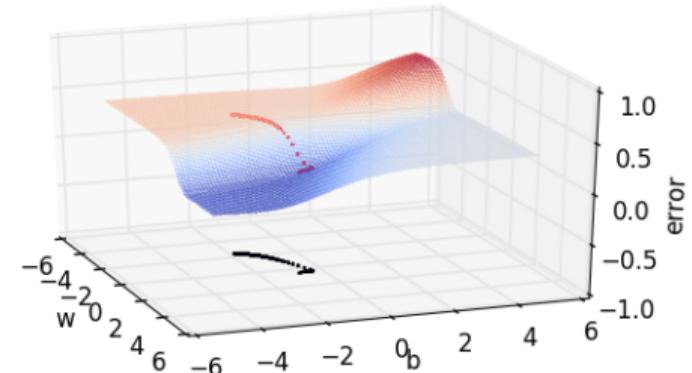
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

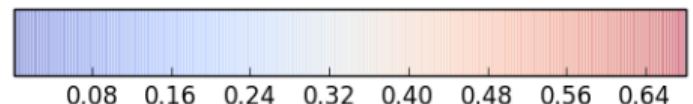
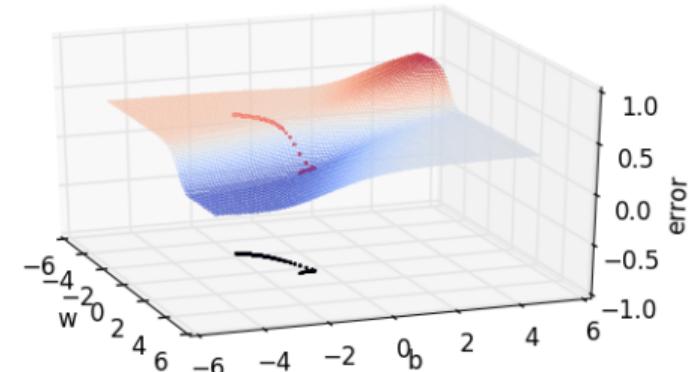
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

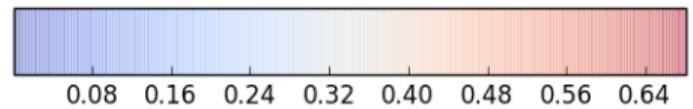
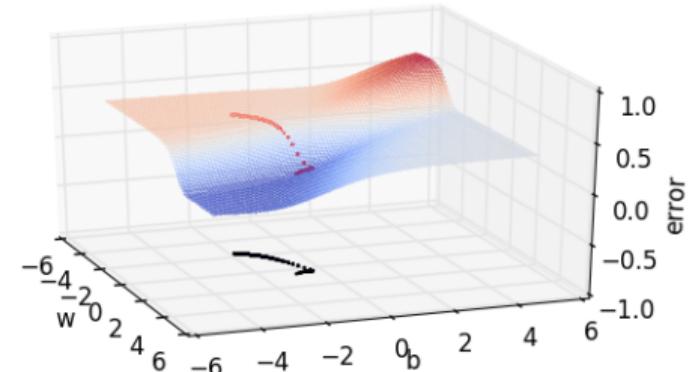
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

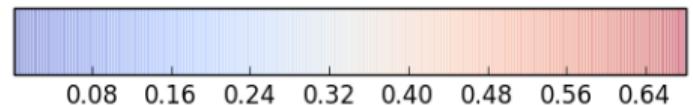
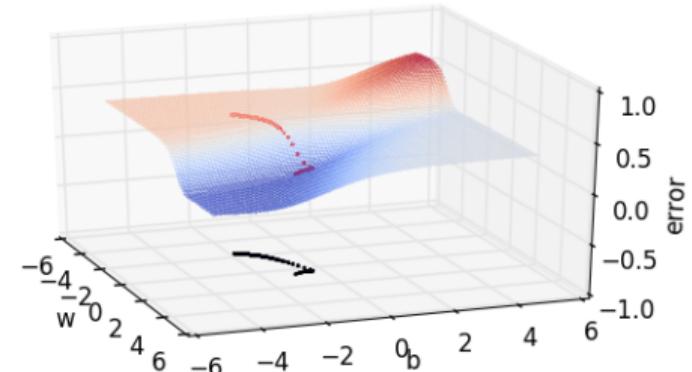
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

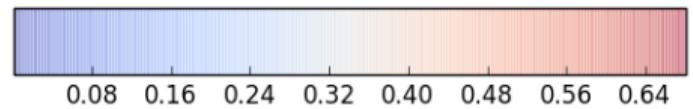
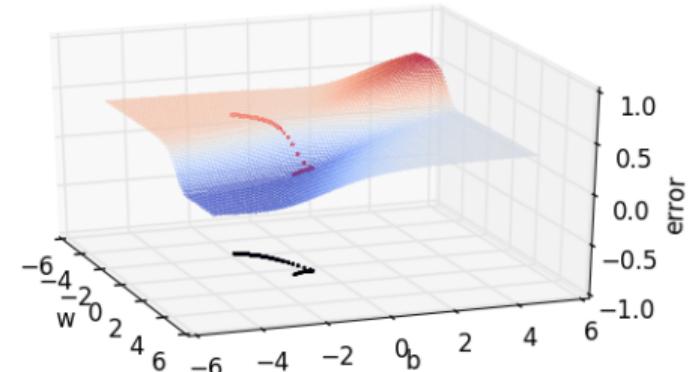
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

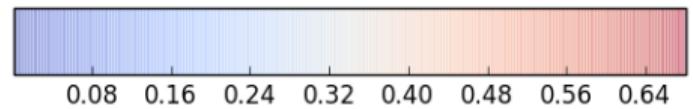
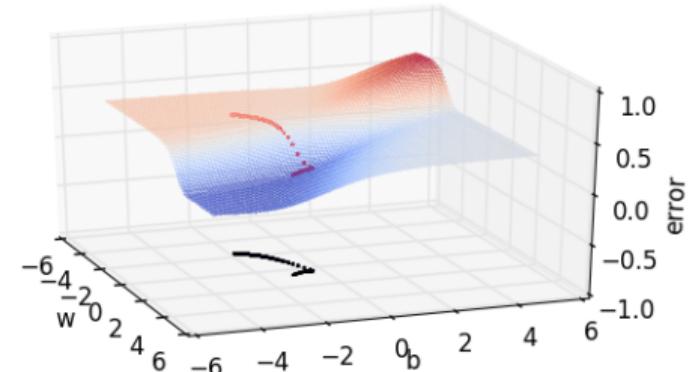
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

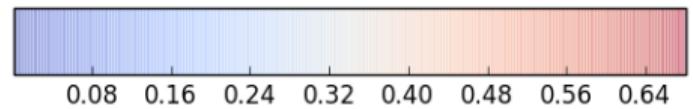
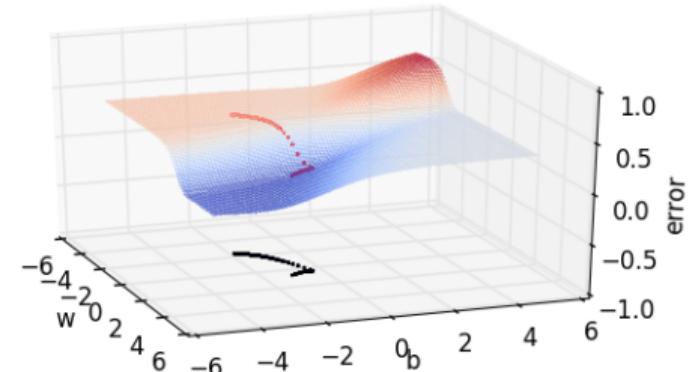
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

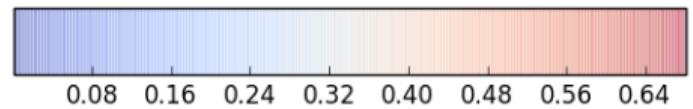
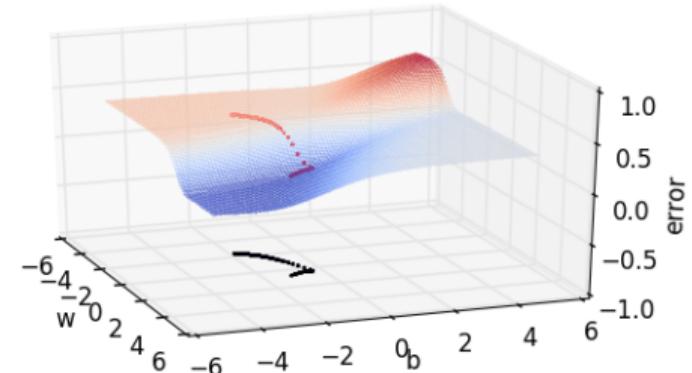
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

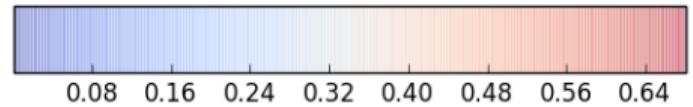
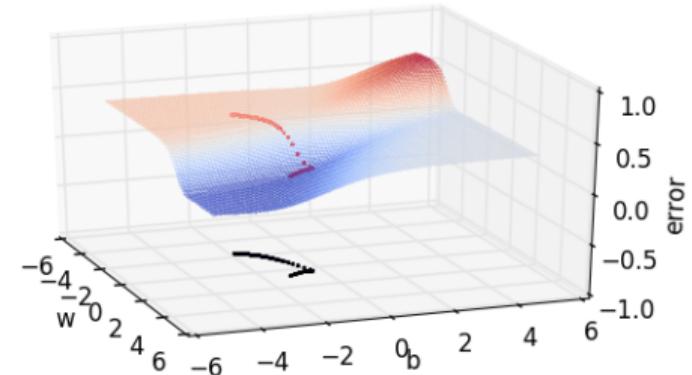
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

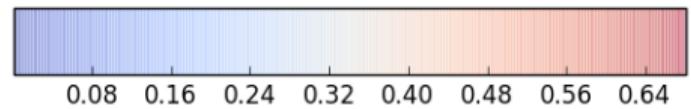
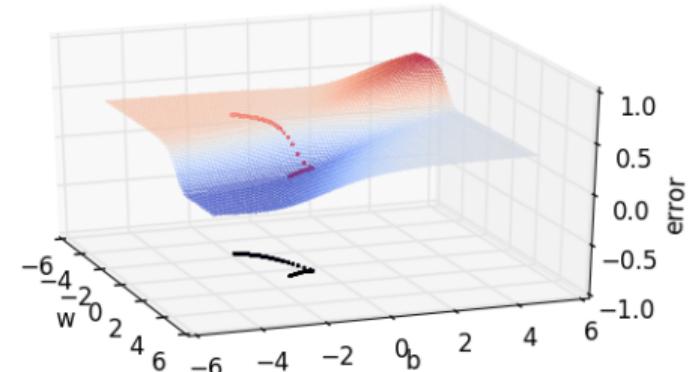
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

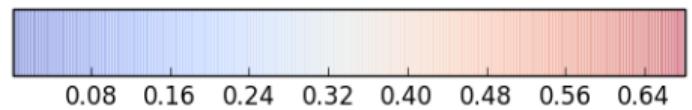
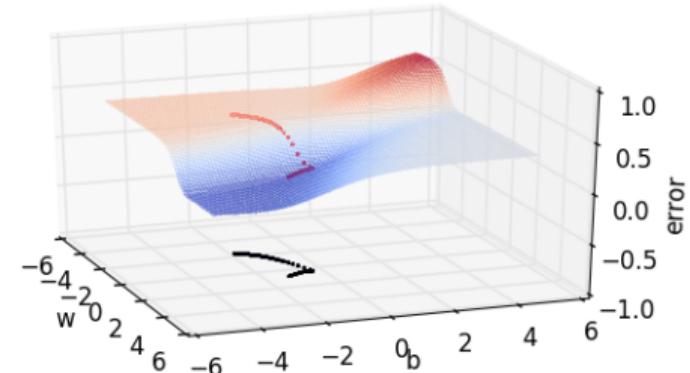
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

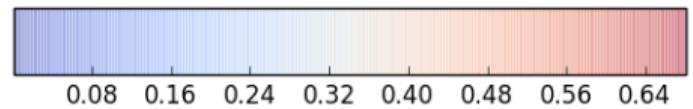
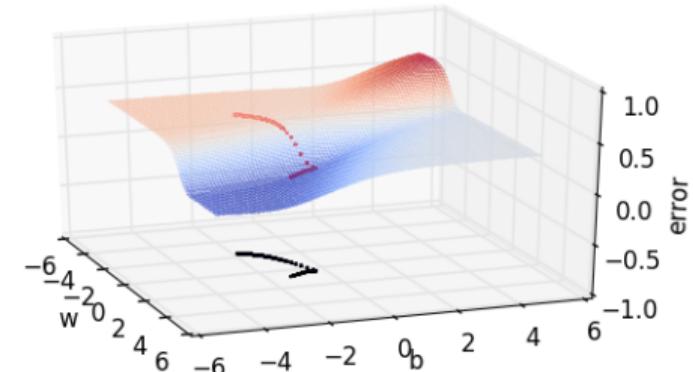
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

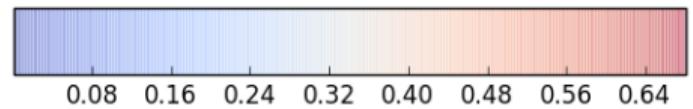
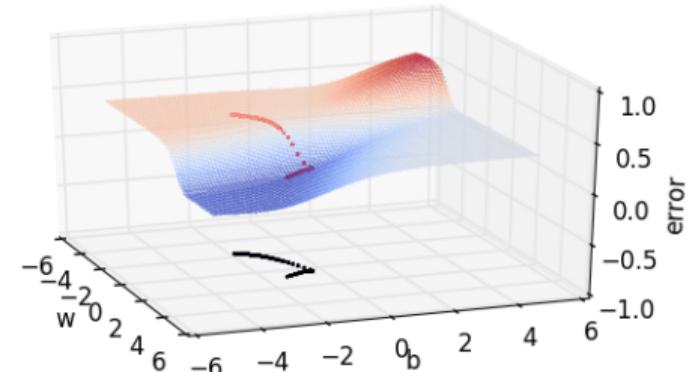
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

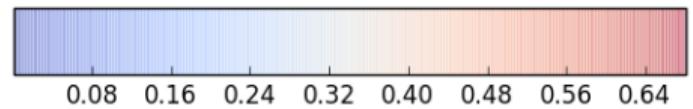
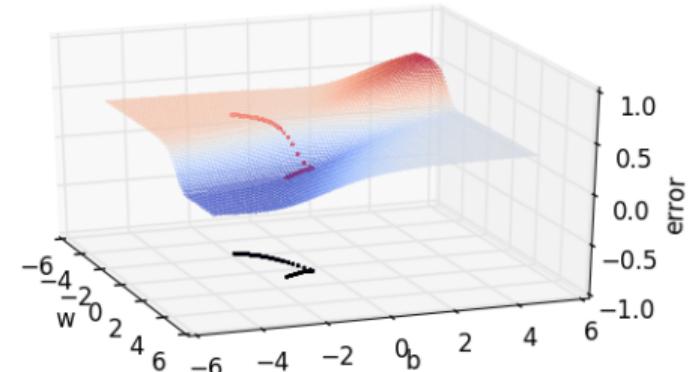
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

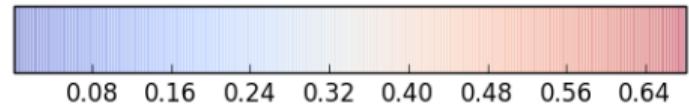
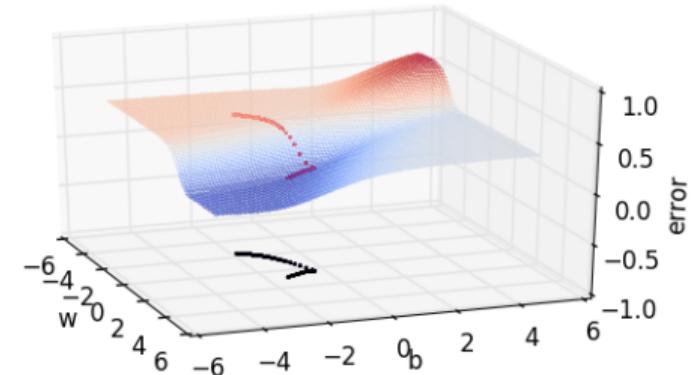
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

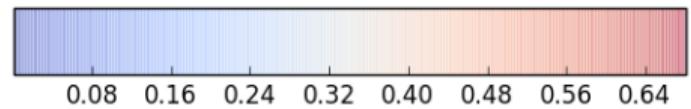
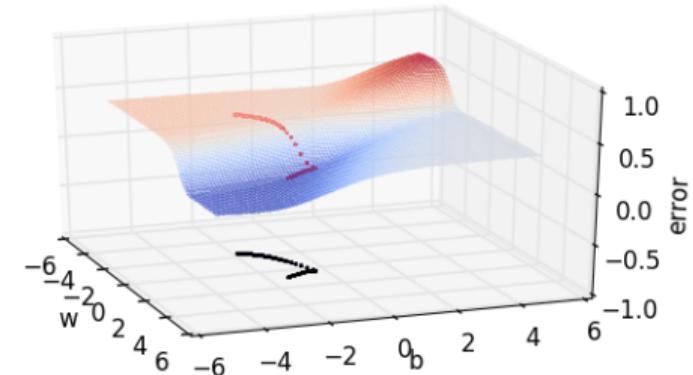
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

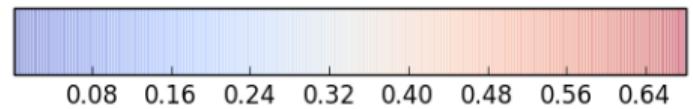
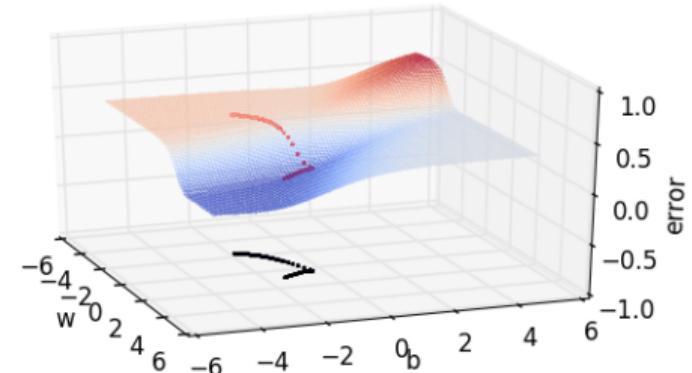
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

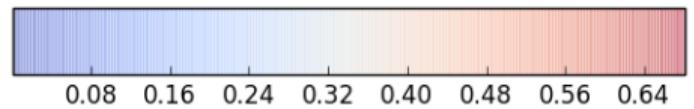
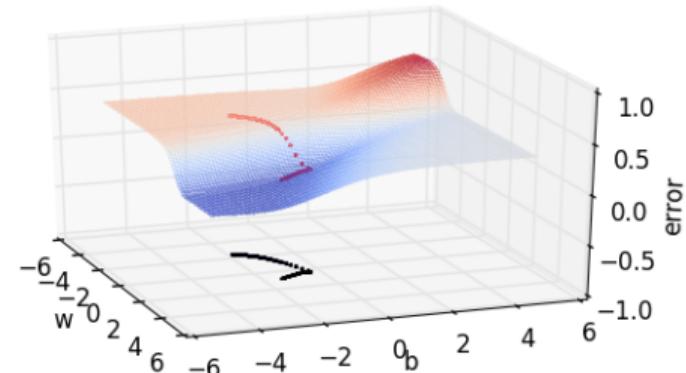
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

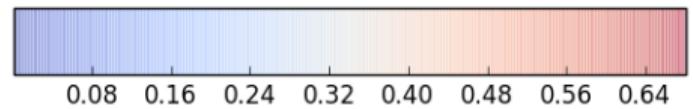
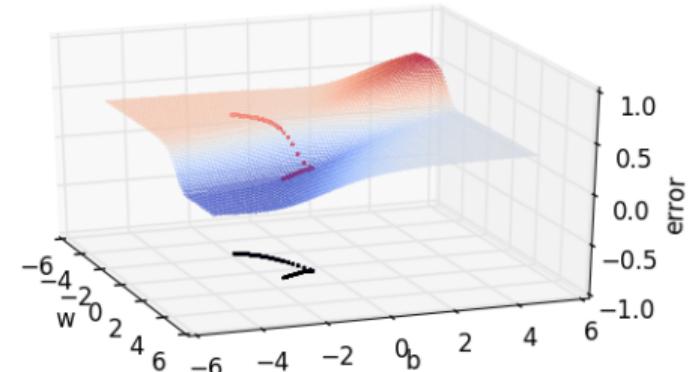
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

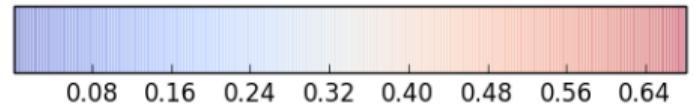
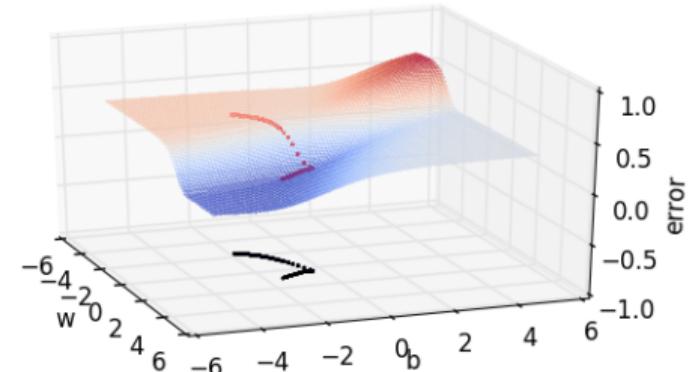
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

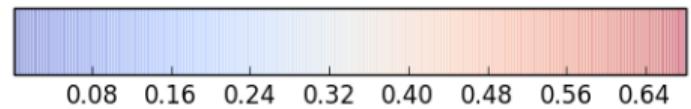
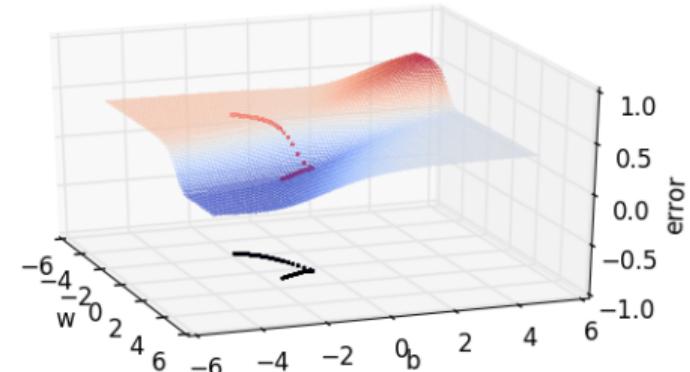
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

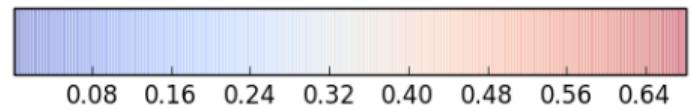
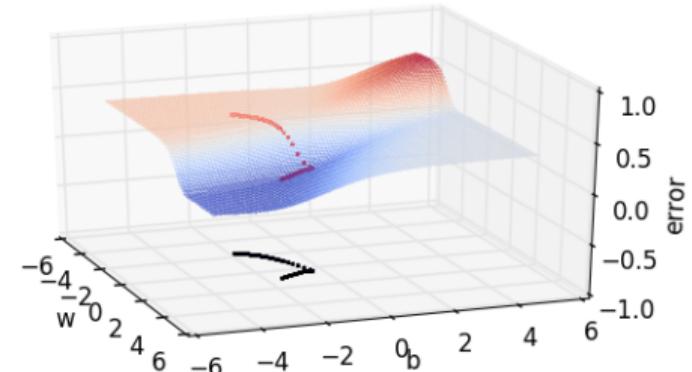
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

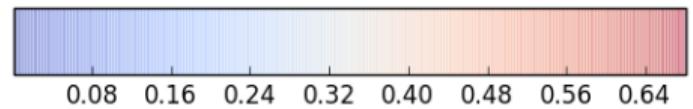
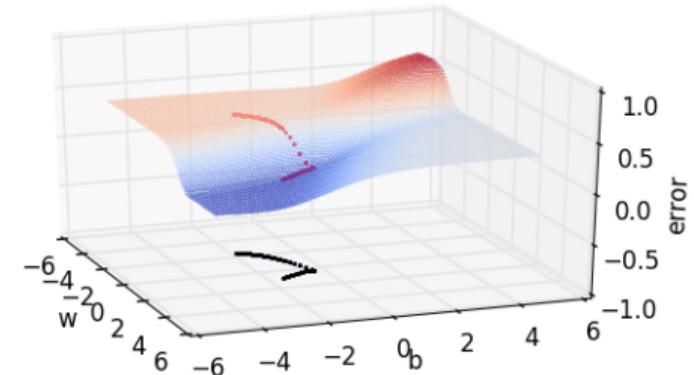
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

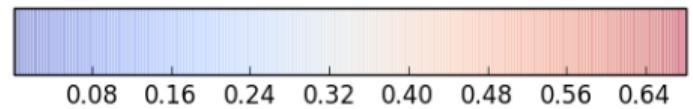
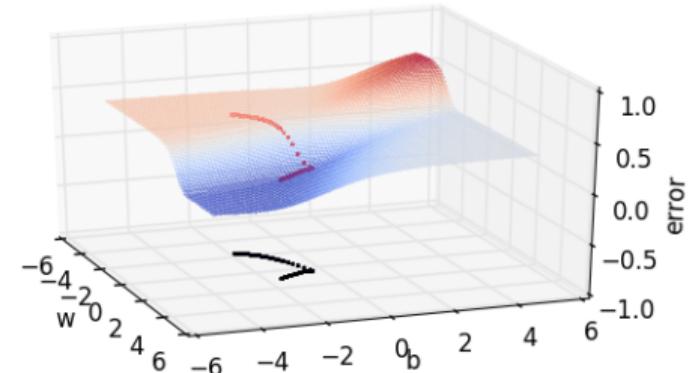
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

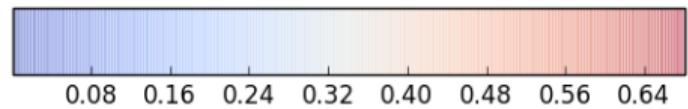
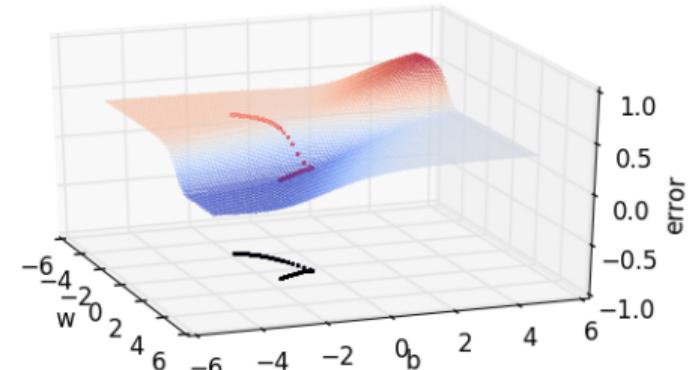
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

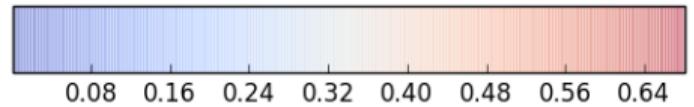
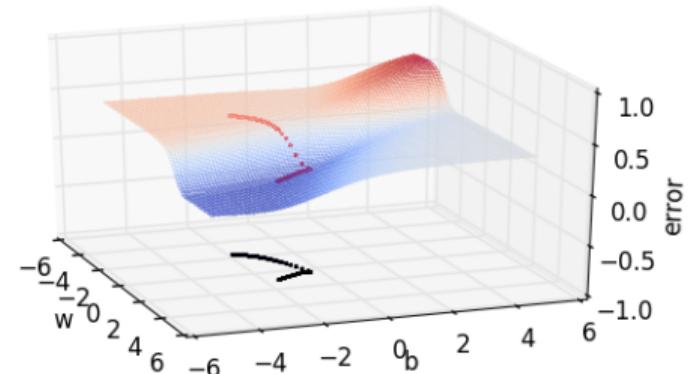
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

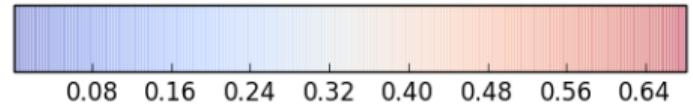
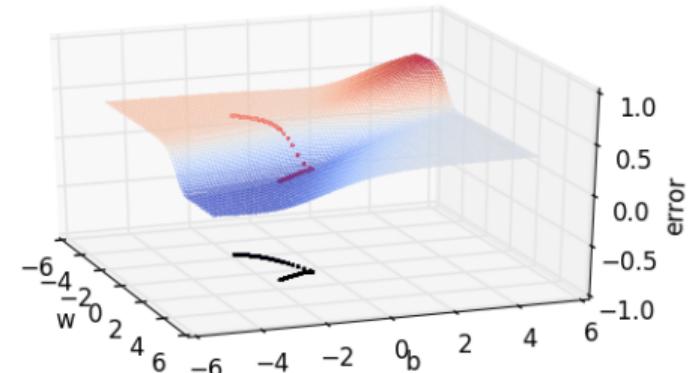
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

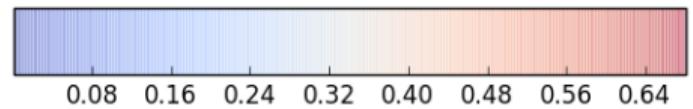
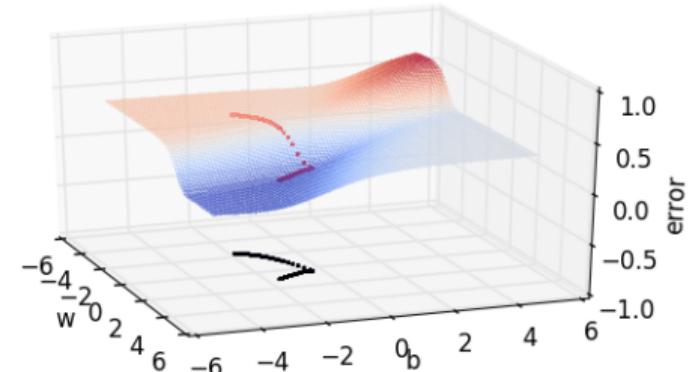
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

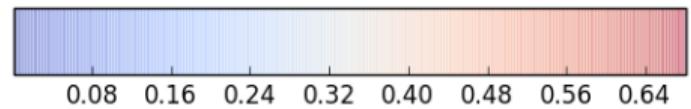
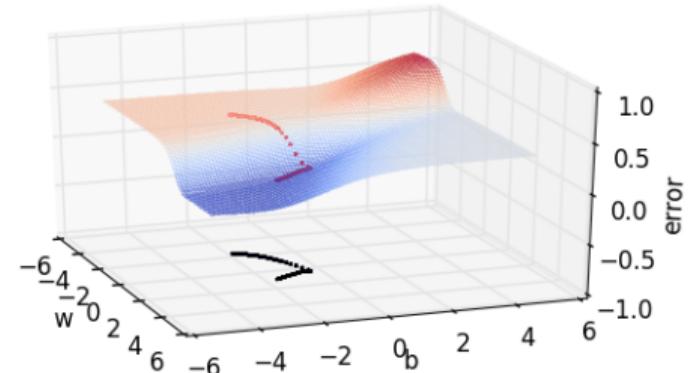
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

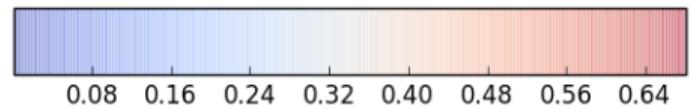
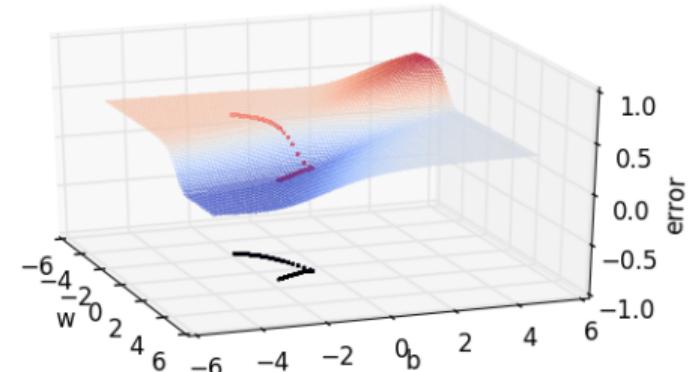
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

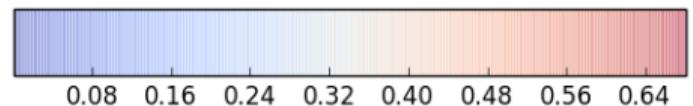
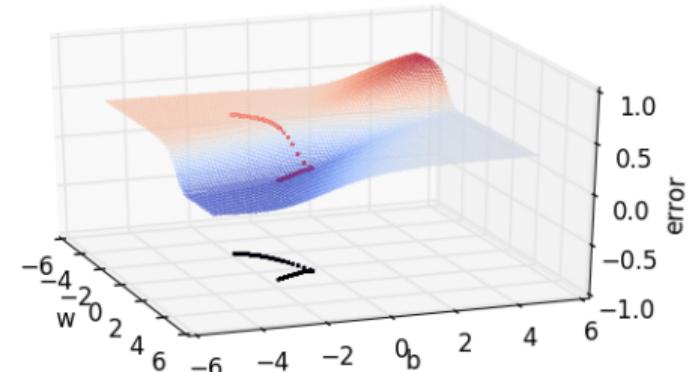
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

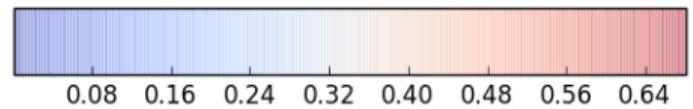
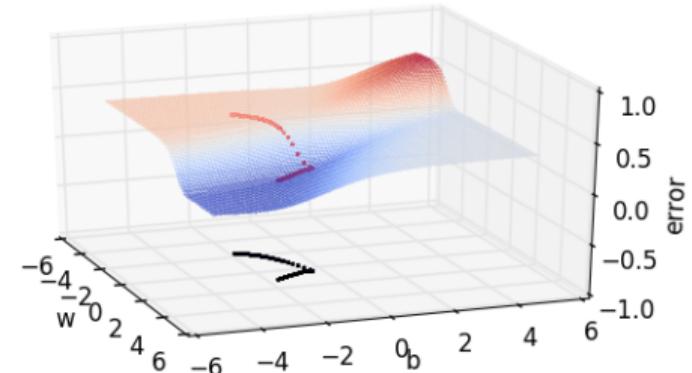
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

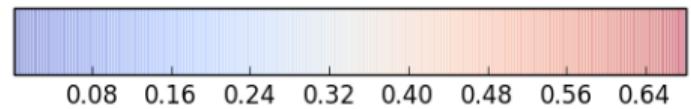
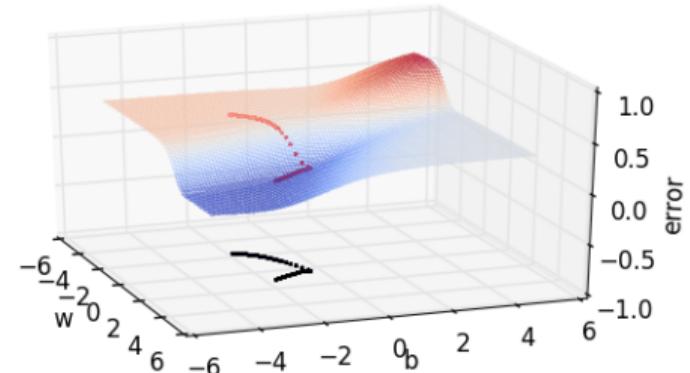
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

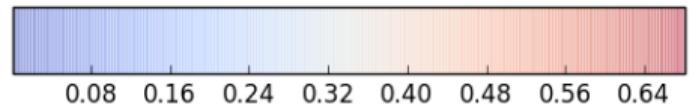
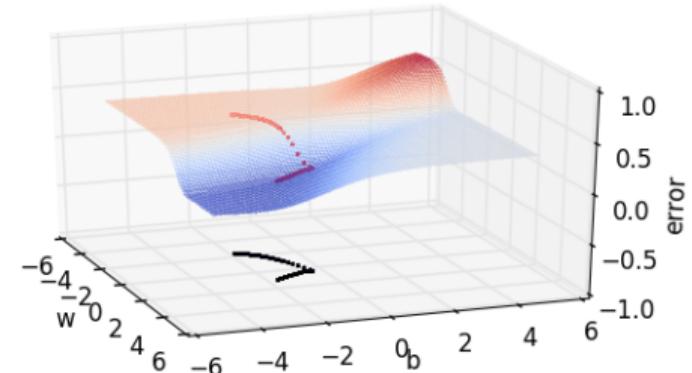
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

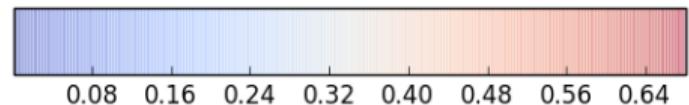
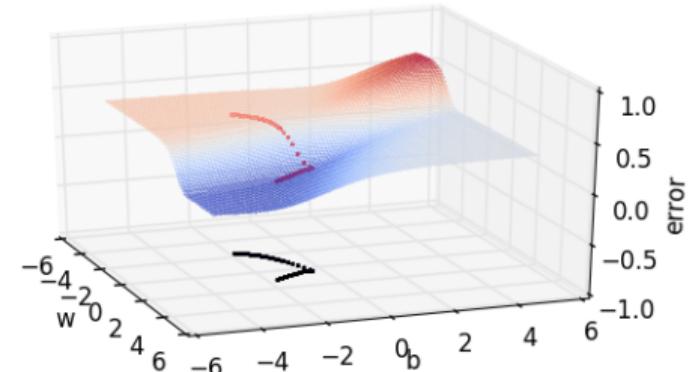
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

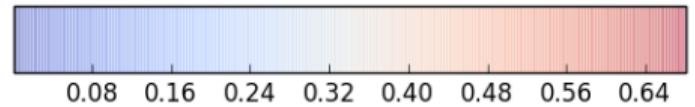
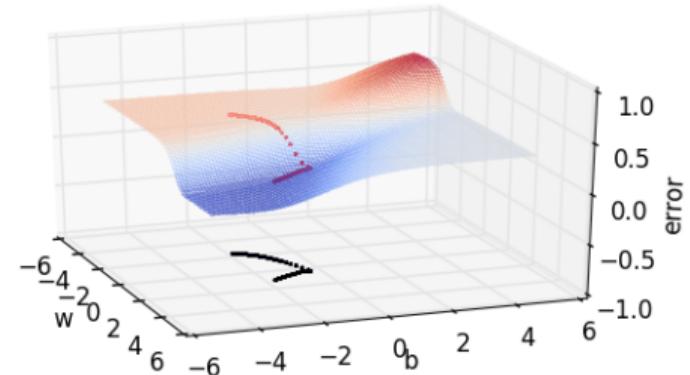
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

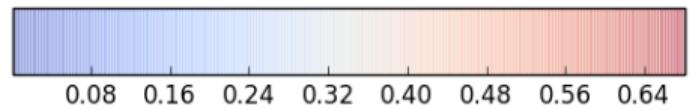
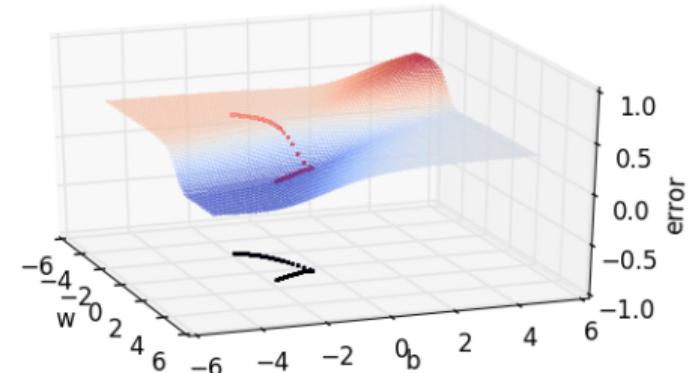
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

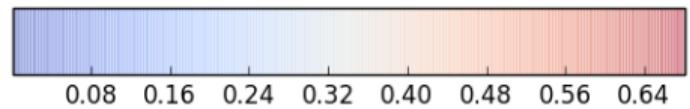
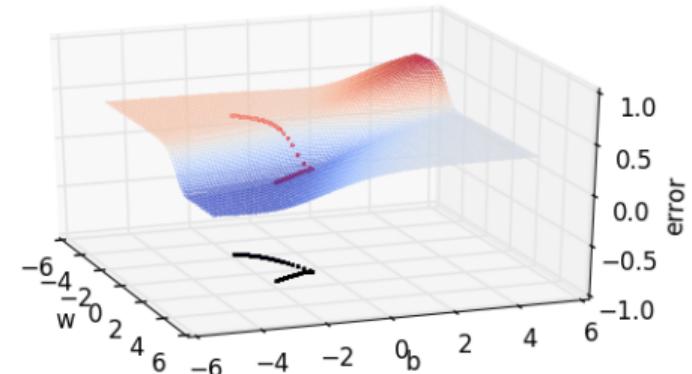
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

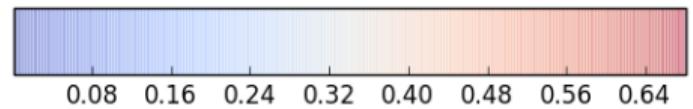
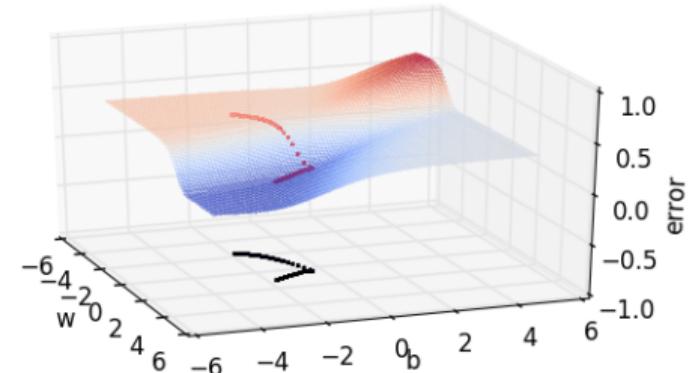
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

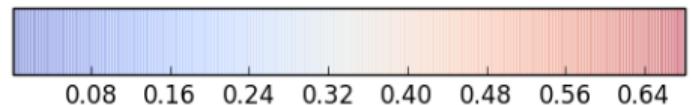
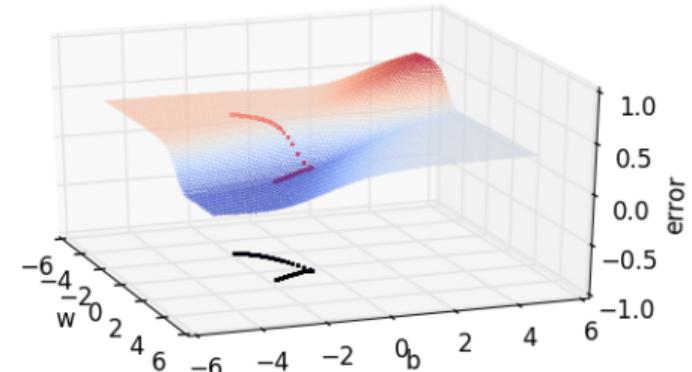
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

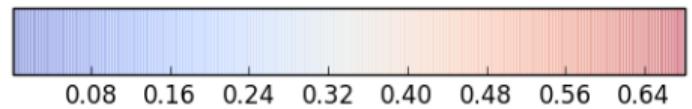
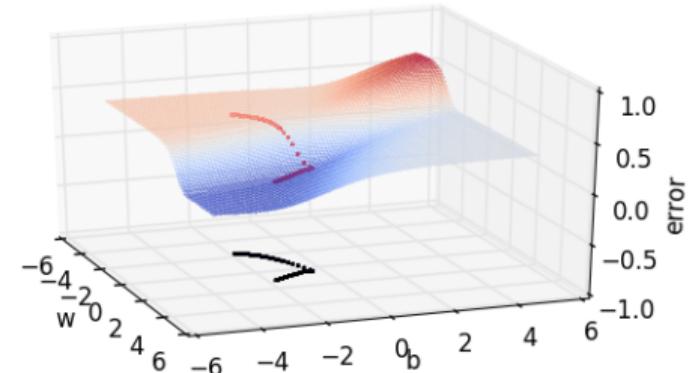
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

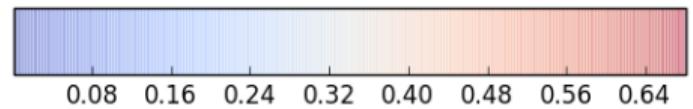
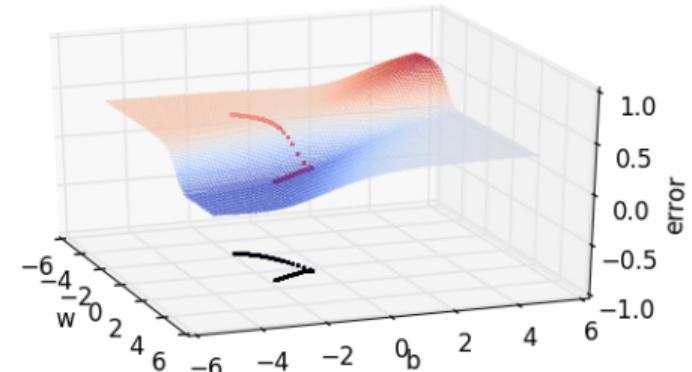
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

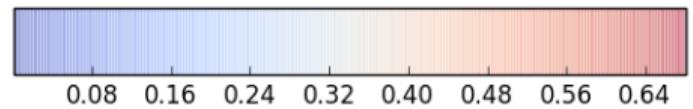
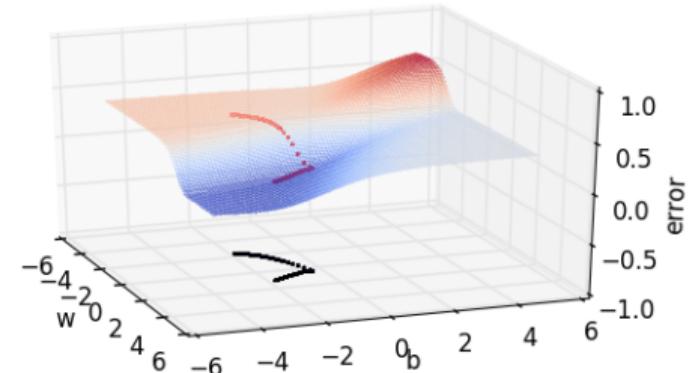
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

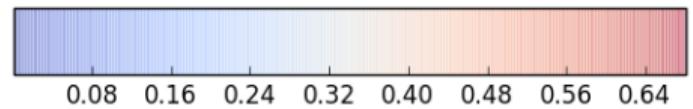
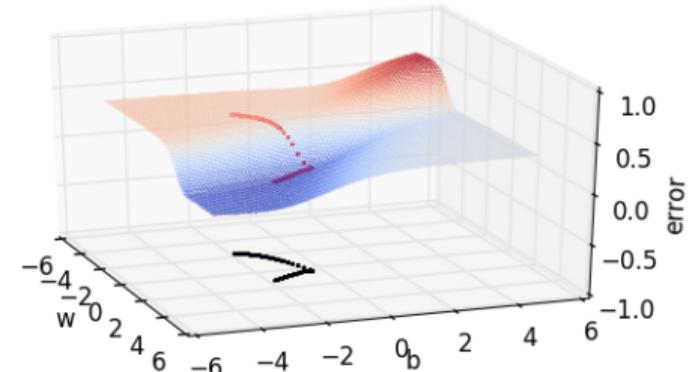
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

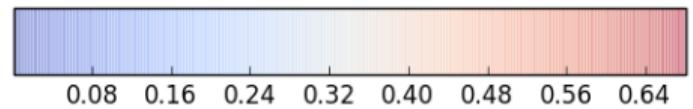
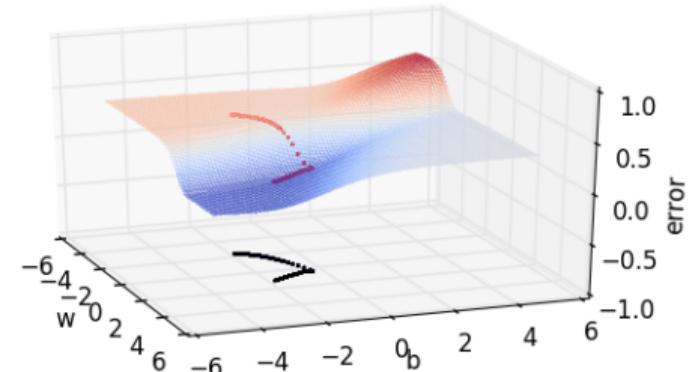
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

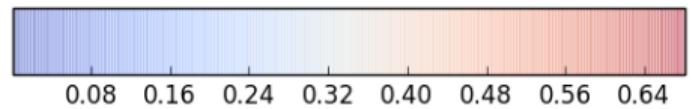
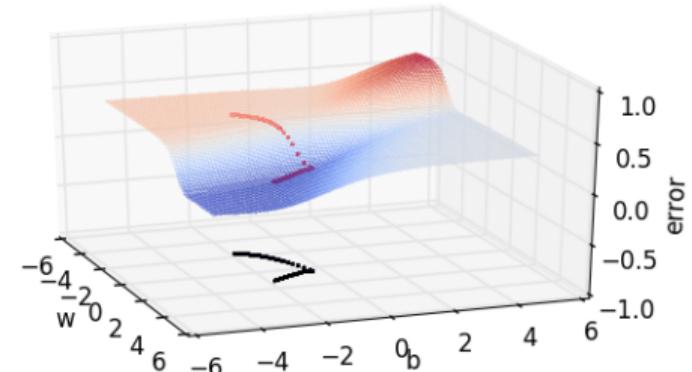
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

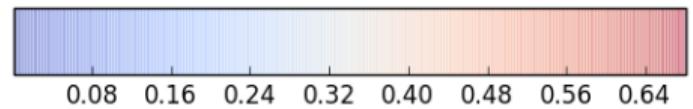
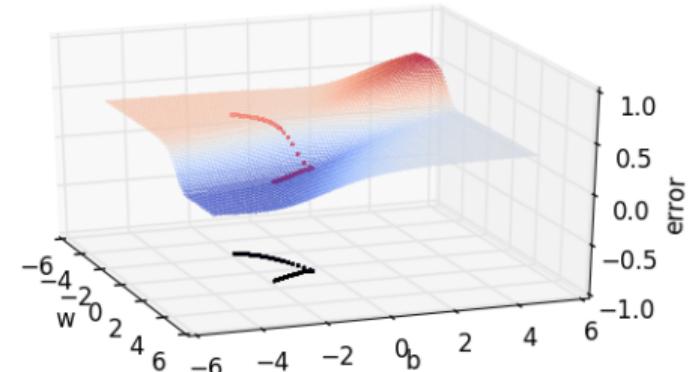
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

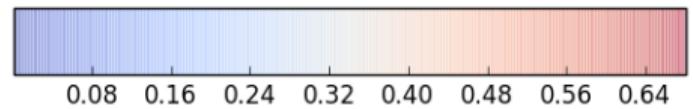
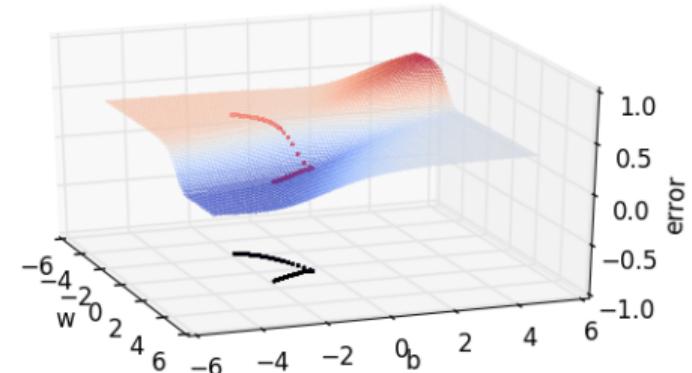
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

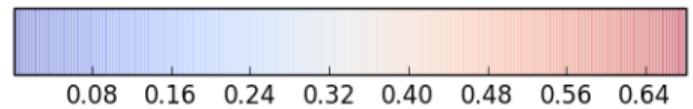
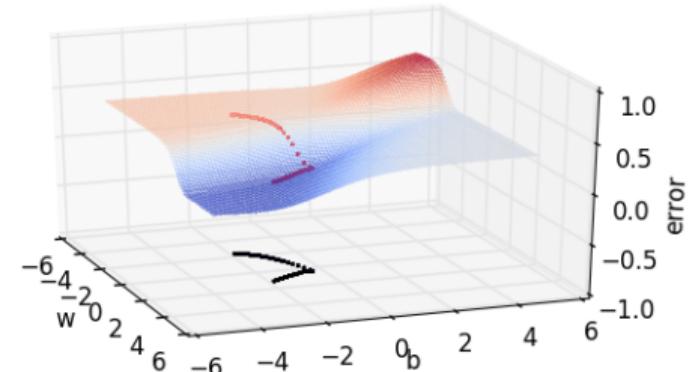
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## 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

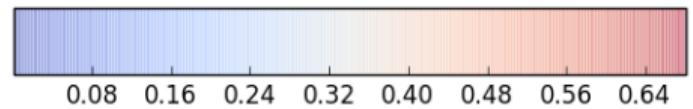
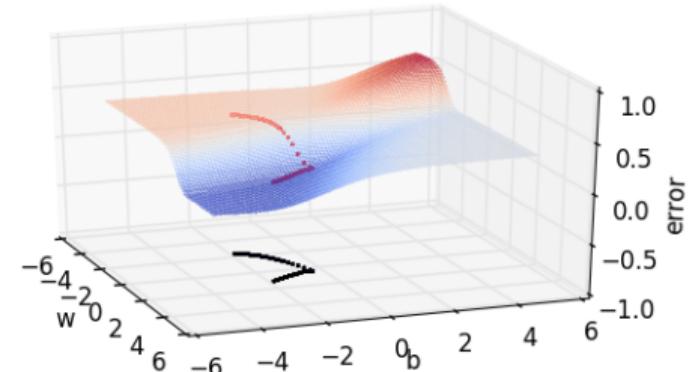
def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

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

```

## Gradient descent on the error surface



- 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 ?

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

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

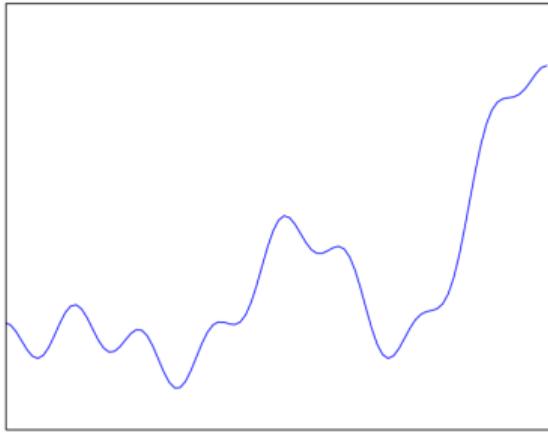
In other words, there is a guarantee that for any function  $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$ , we can always find a neural network (with 1 hidden layer containing enough neurons) whose output  $g(x)$  satisfies  $|g(x) - f(x)| < \epsilon$  !!

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

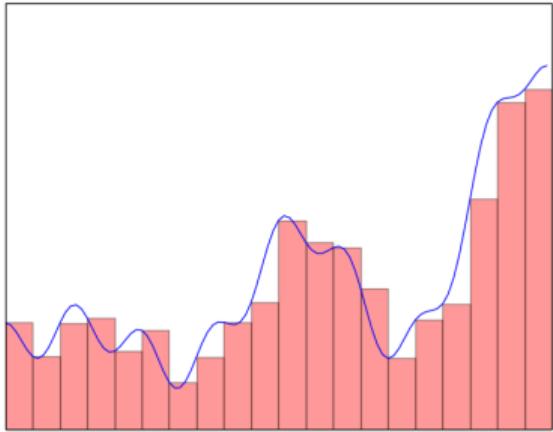
- See this link\* for an excellent illustration of this proof
- The discussion in the next few slides is based on the ideas presented at the above link

---

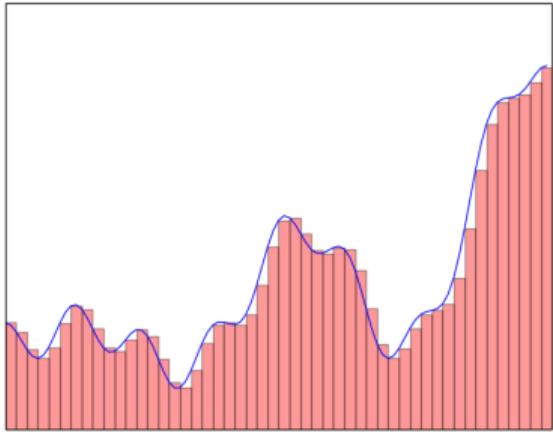
\*<http://neuralnetworksanddeeplearning.com/chap4.html>



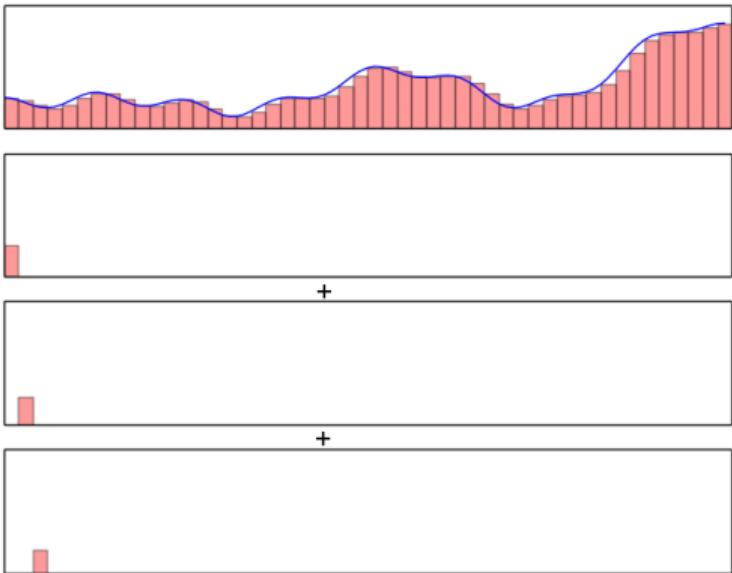
- We are interested in knowing whether a network of neurons can be used to represent an arbitrary function (like the one shown in the figure)



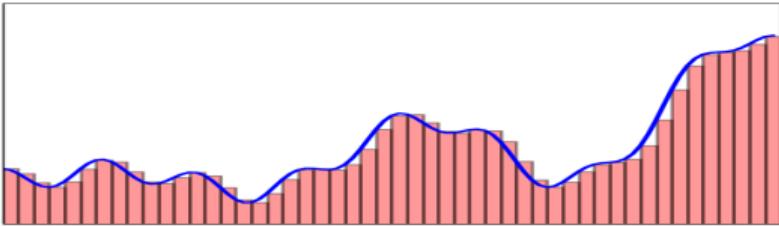
- We are interested in knowing whether a network of neurons can be used to represent an arbitrary function (like the one shown in the figure)
- We observe that such an arbitrary function can be approximated by several “tower” functions



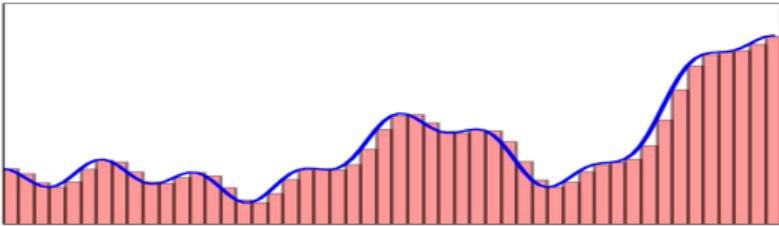
- We are interested in knowing whether a network of neurons can be used to represent an arbitrary function (like the one shown in the figure)
- We observe that such an arbitrary function can be approximated by several “tower” functions
- More the number of such “tower” functions, better the approximation



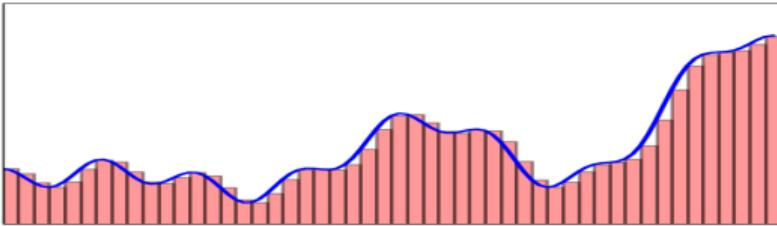
- We are interested in knowing whether a network of neurons can be used to represent an arbitrary function (like the one shown in the figure)
- We observe that such an arbitrary function can be approximated by several “tower” functions
- More the number of such “tower” functions, better the approximation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



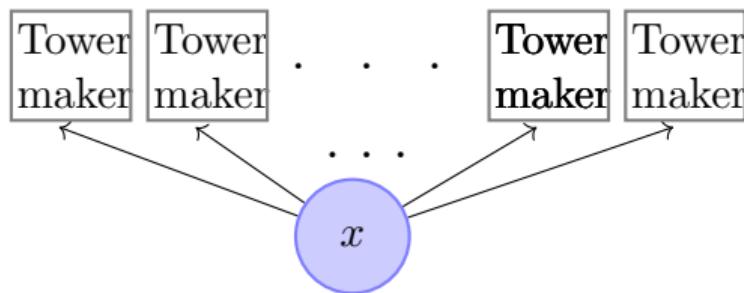
- We make a few observations

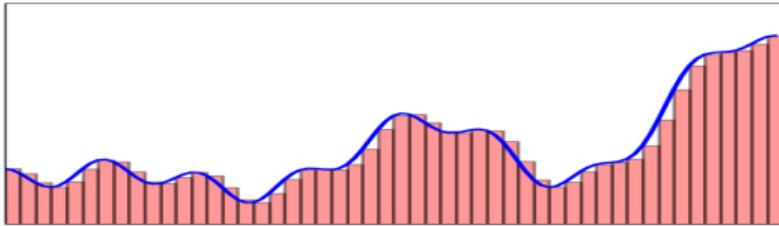


- We make a few observations
- All these “tower” functions are similar and only differ in their heights and positions on the x-axis

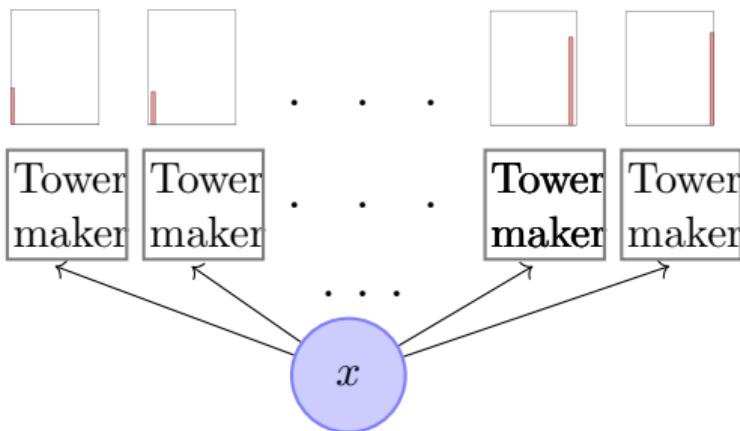


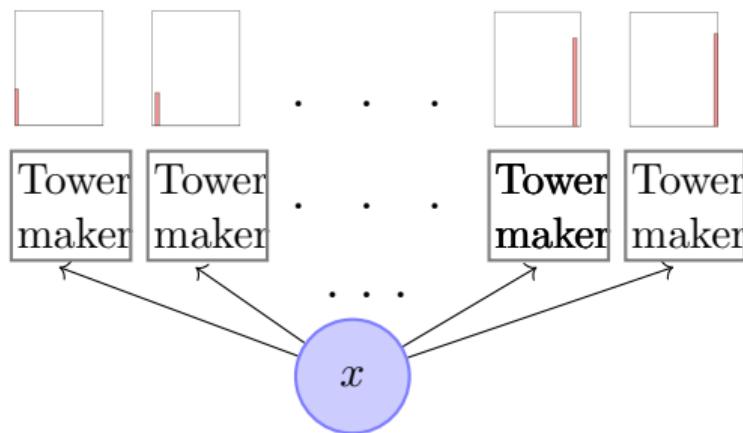
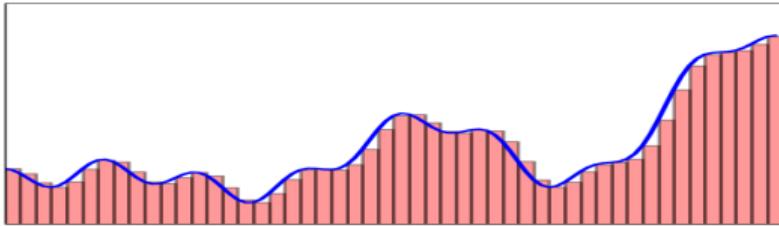
- We make a few observations
- All these “tower” functions are similar and only differ in their heights and positions on the x-axis
- Suppose there is a black box which takes the original input ( $x$ ) and constructs these tower functions



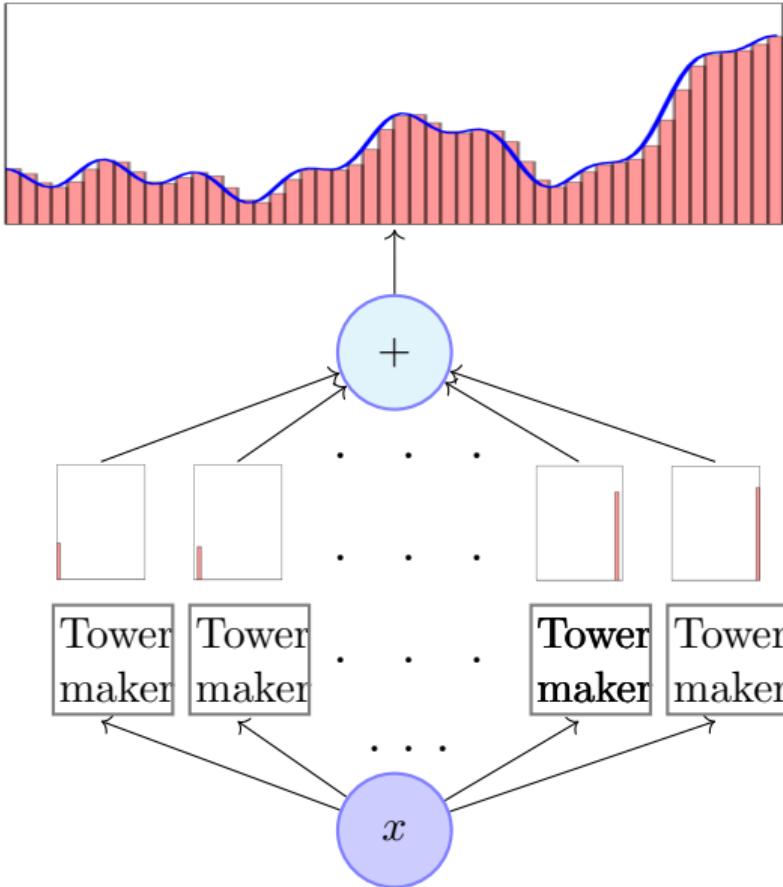


- We make a few observations
- All these “tower” functions are similar and only differ in their heights and positions on the x-axis
- Suppose there is a black box which takes the original input ( $x$ ) and constructs these tower functions

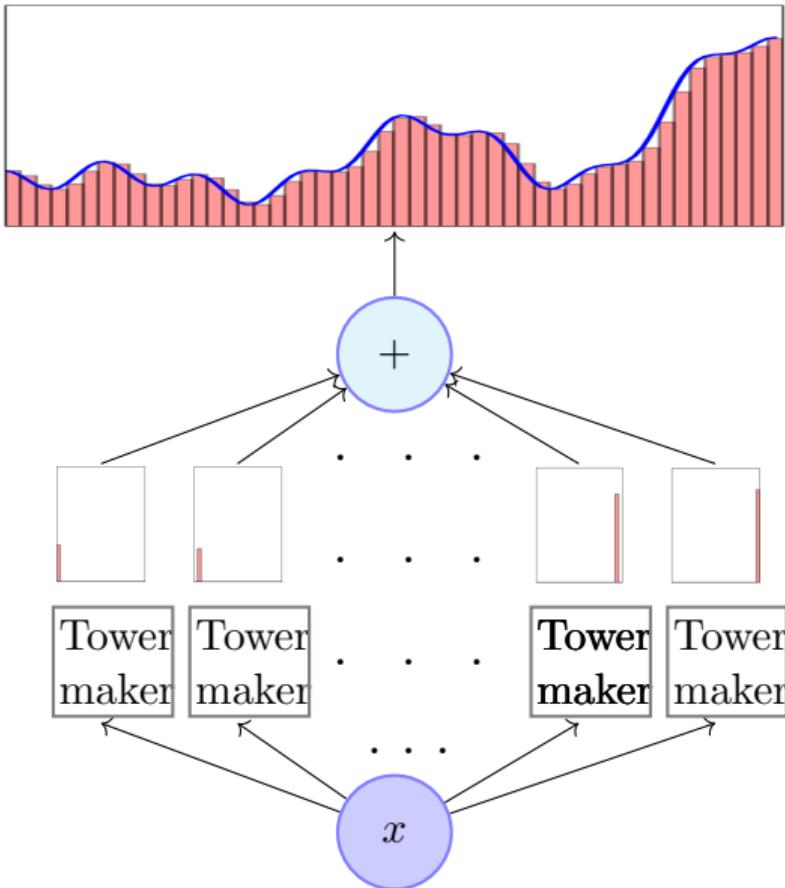




- We make a few observations
- All these “tower” functions are similar and only differ in their heights and positions on the x-axis
- Suppose there is a black box which takes the original input ( $x$ ) and constructs these tower functions
- We can then have a simple network which can just add them up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

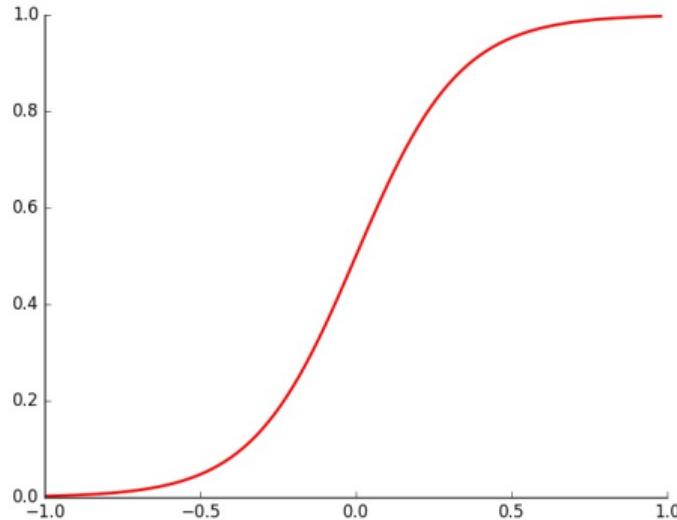


- We make a few observations
- All these “tower” functions are similar and only differ in their heights and positions on the x-axis
- Suppose there is a black box which takes the original input ( $x$ ) and constructs these tower functions
- We can then have a simple network which can just add them up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

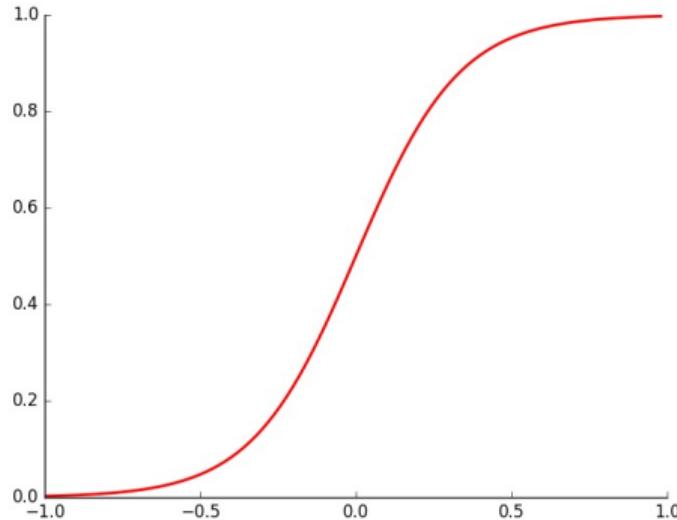


- We make a few observations
  - All these “tower” functions are similar and only differ in their heights and positions on the x-axis
  - Suppose there is a black box which takes the original input ( $x$ ) and constructs these tower functions
  - We can then have a simple network which can just add them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

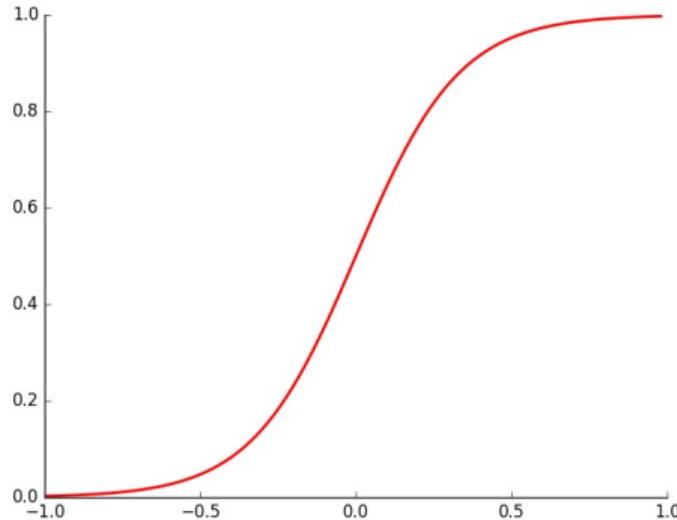
We will figure this out over the next few slides ...



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

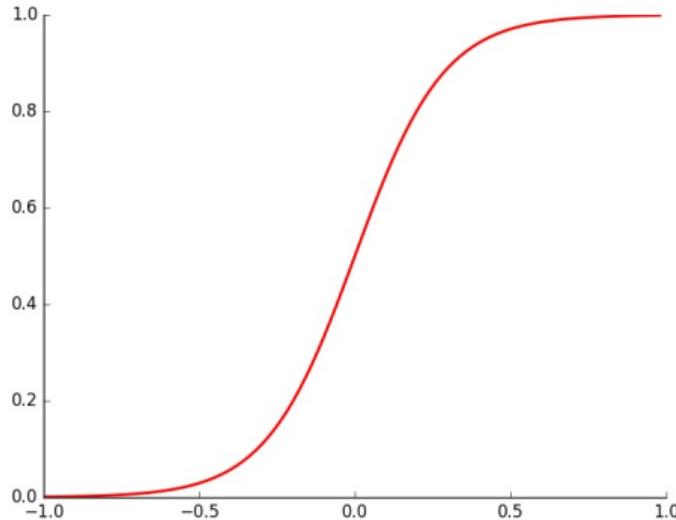


- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$



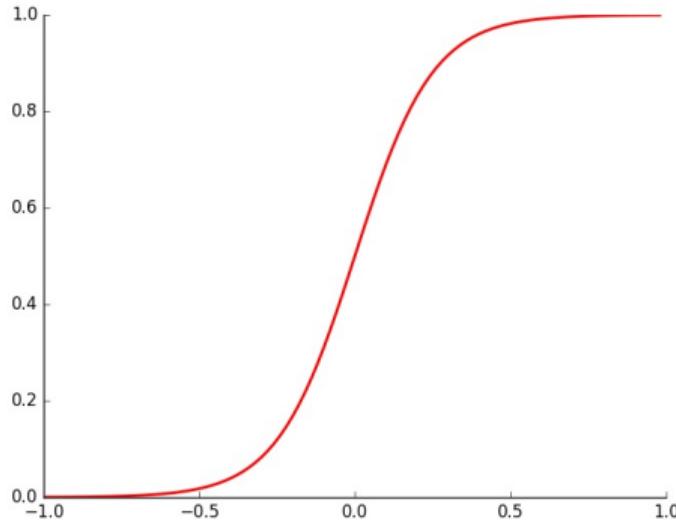
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 0$$



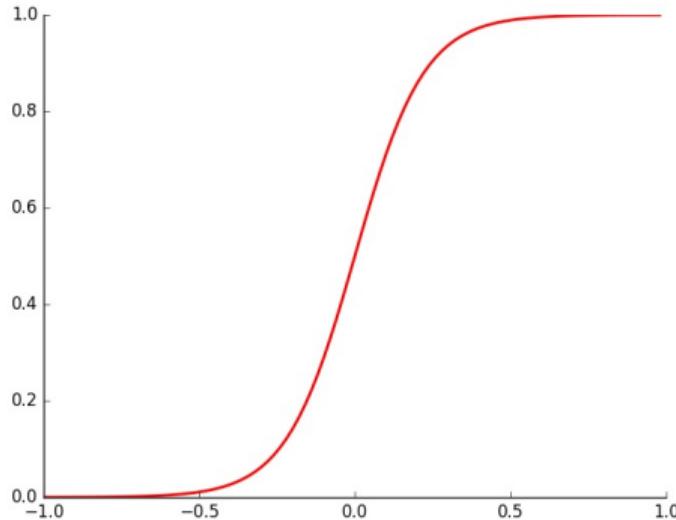
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 1$$



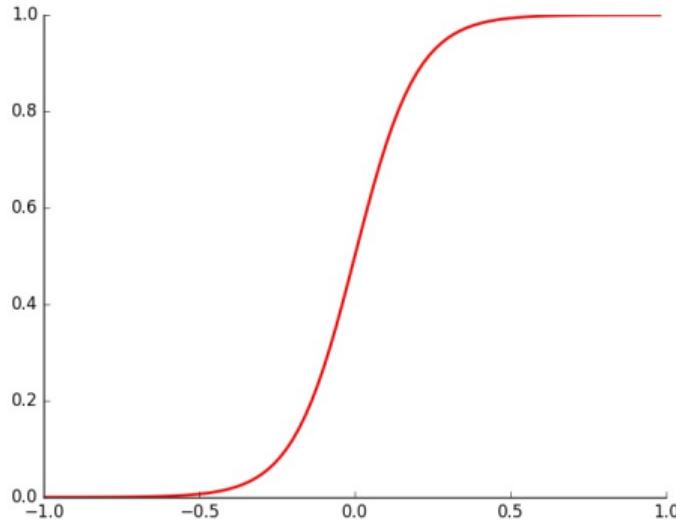
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 2$$



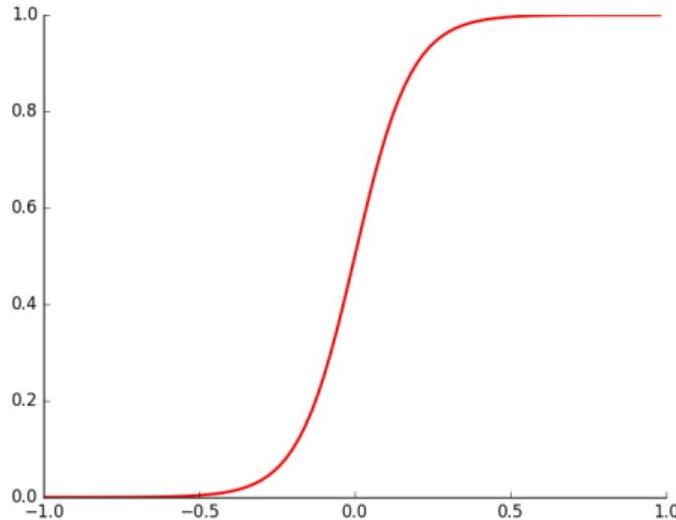
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 3$$



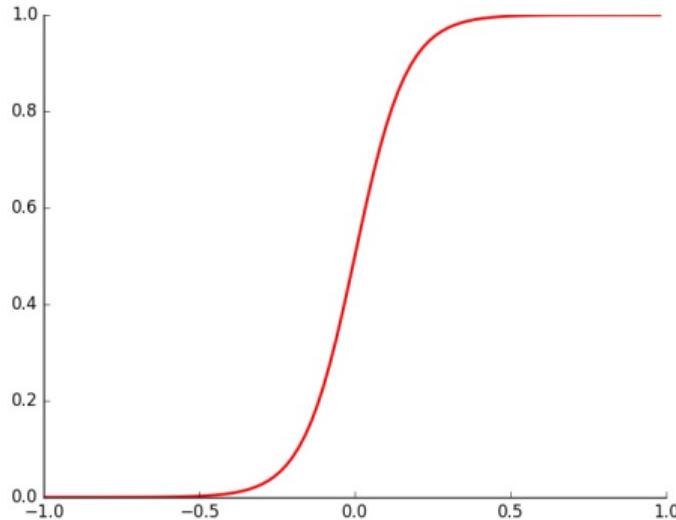
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 4$$



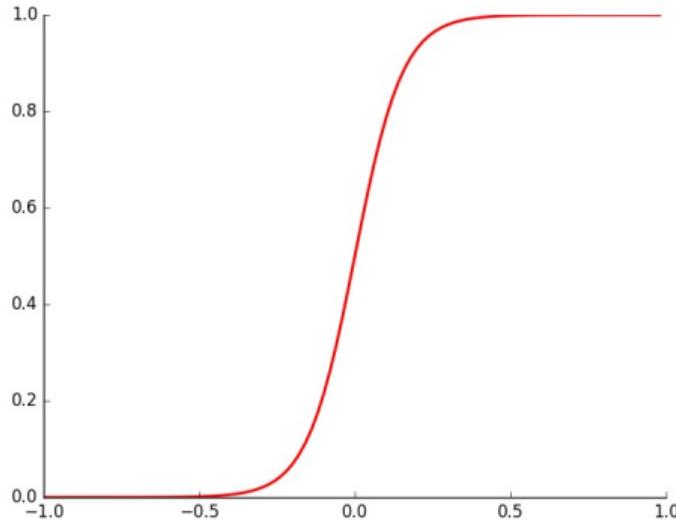
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 5$$



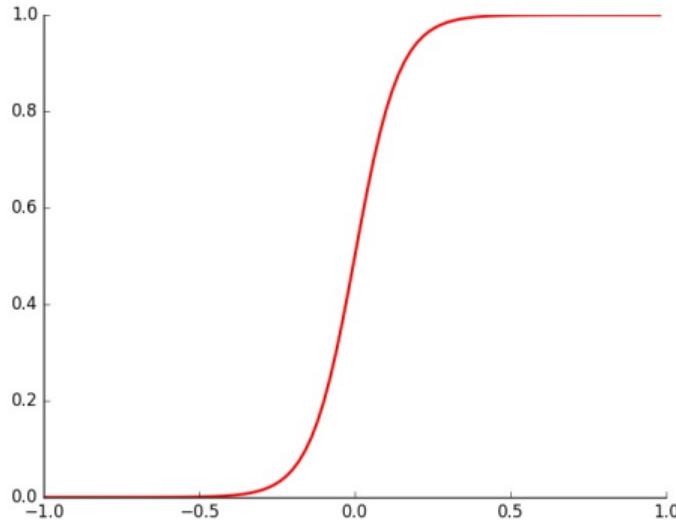
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 6$$



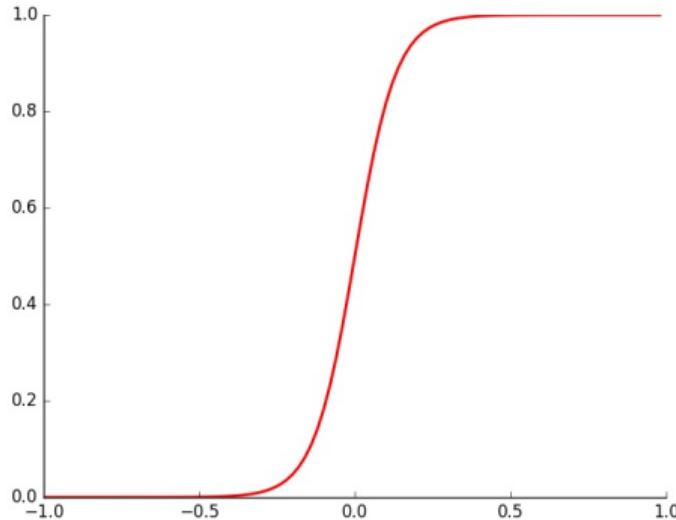
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 7$$



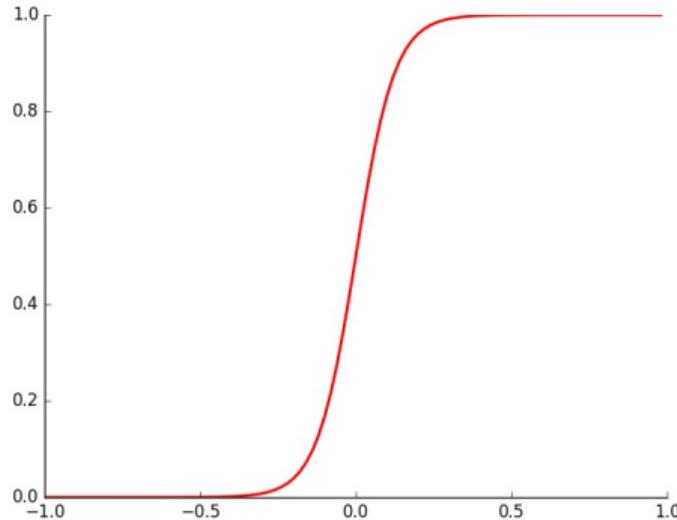
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1 - \exp(-wx - b)} \quad w = 50, b = 8$$



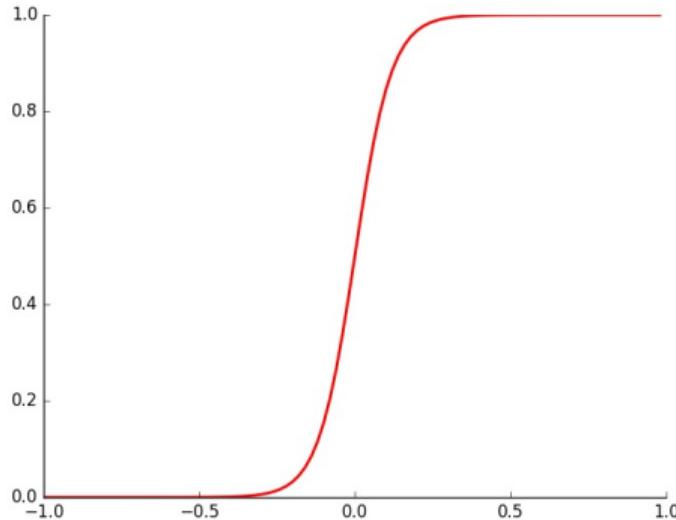
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 9$$



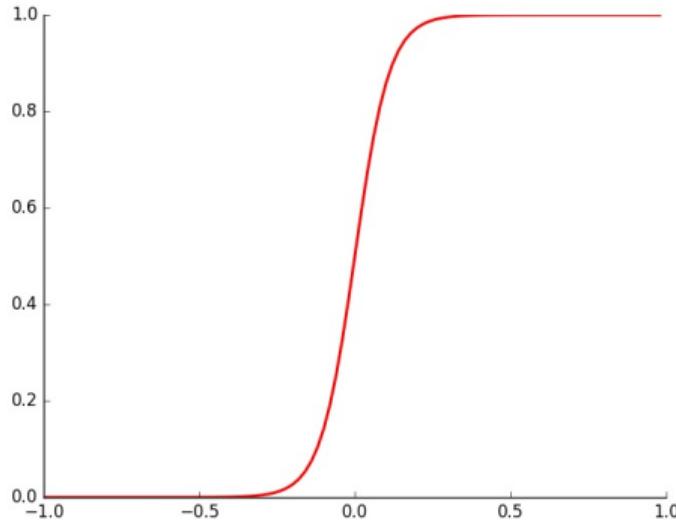
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 10$$



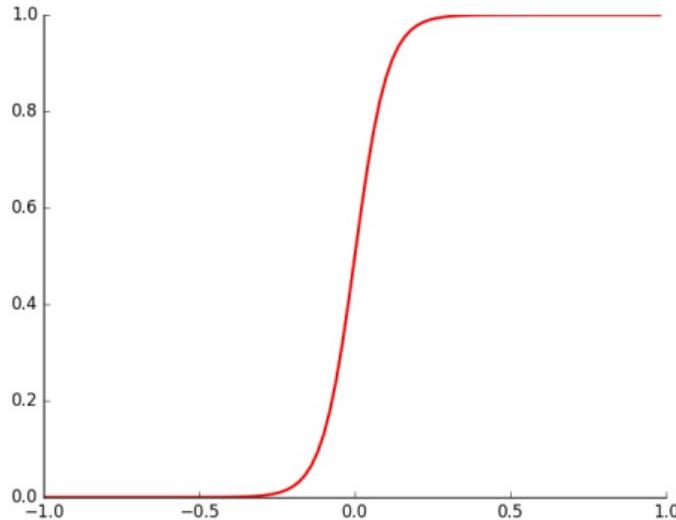
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 11$$



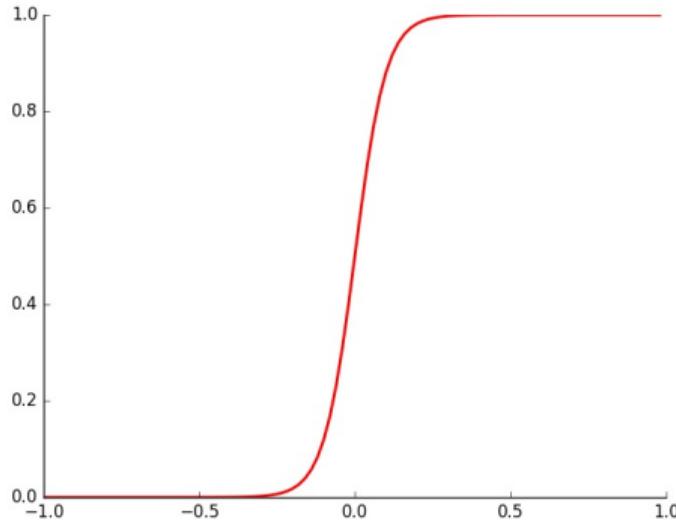
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 12$$



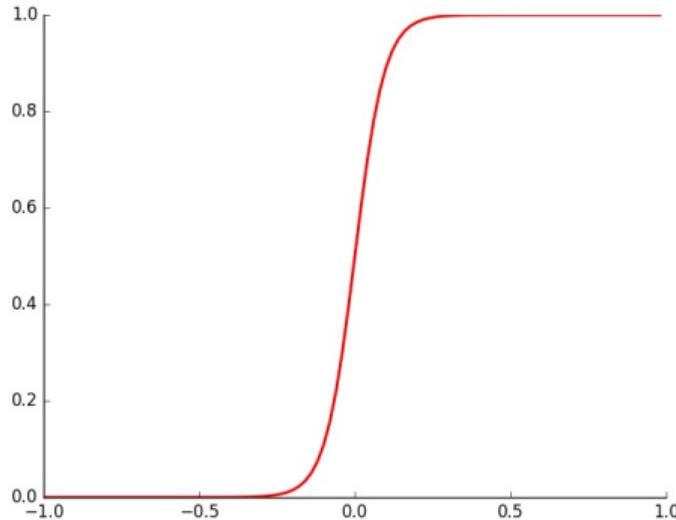
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 13$$



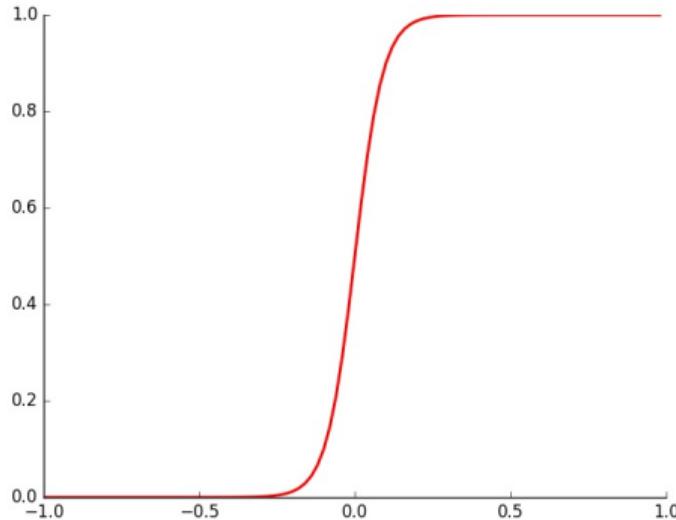
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 14$$



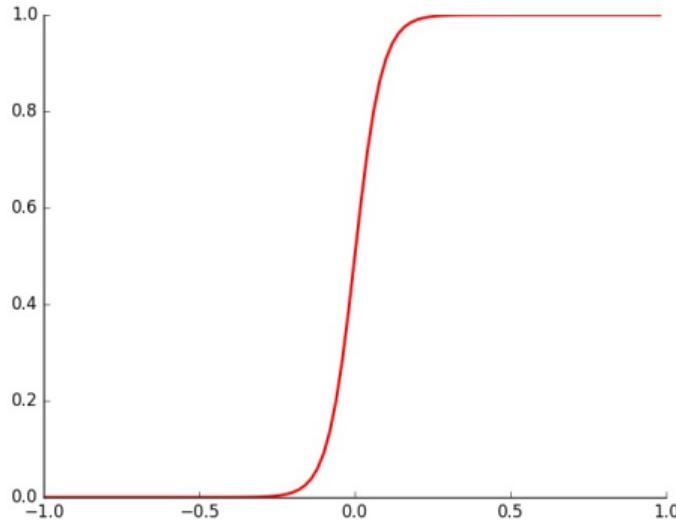
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 15$$



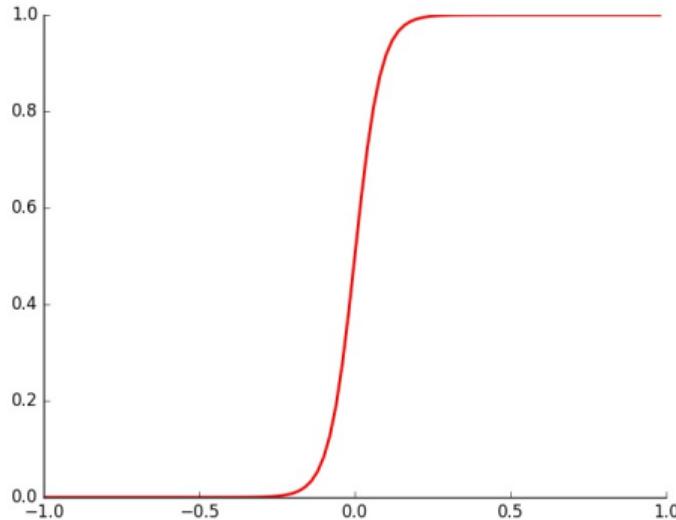
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 16$$



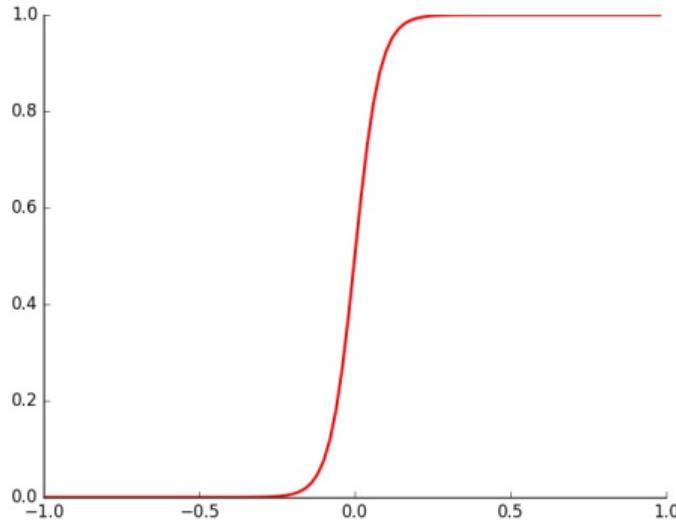
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1 - \exp(-wx - b)} \quad w = 50, b = 17$$



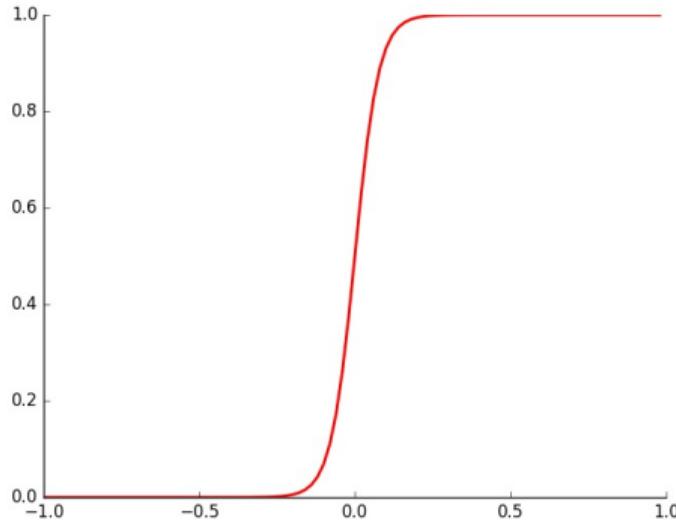
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 18$$



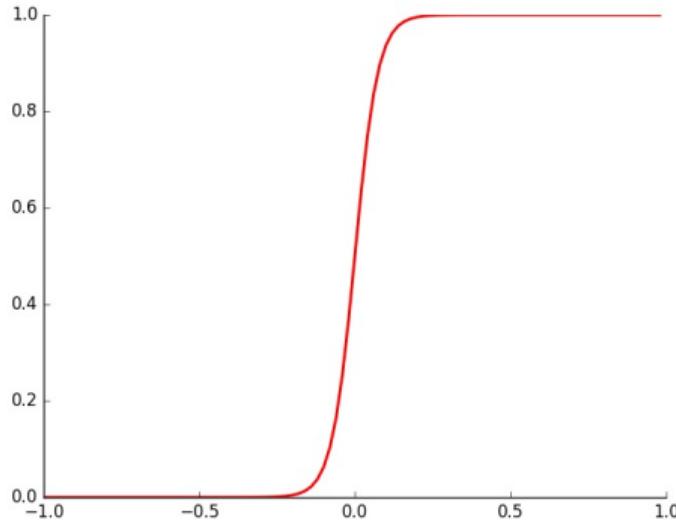
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1 - \exp(-wx - b)} \quad w = 50, b = 19$$



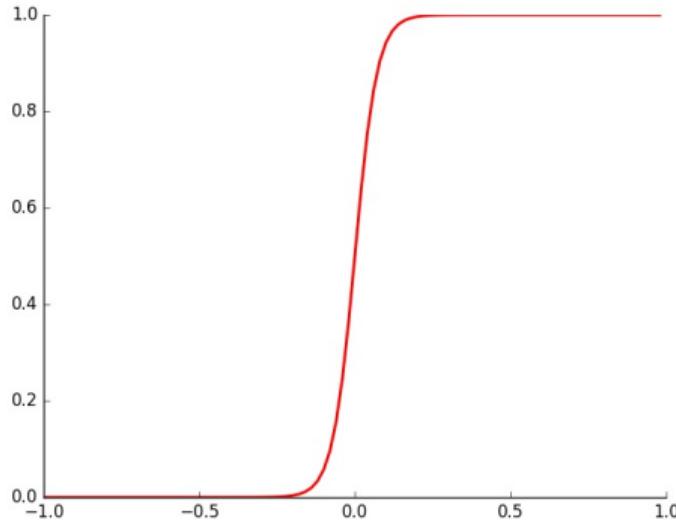
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 20$$



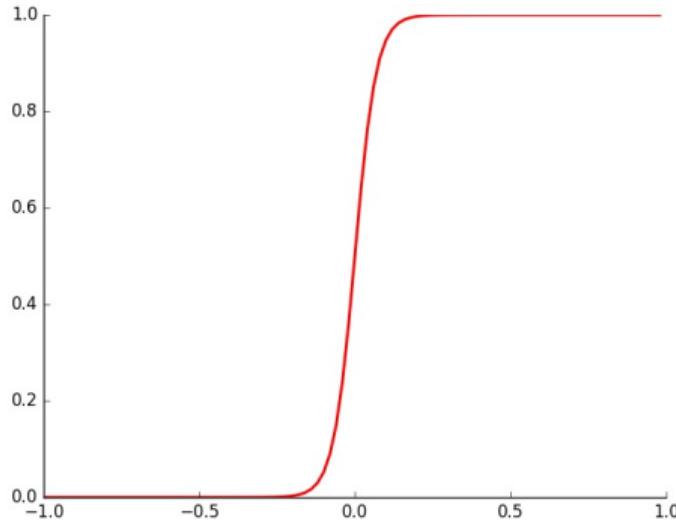
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 21$$



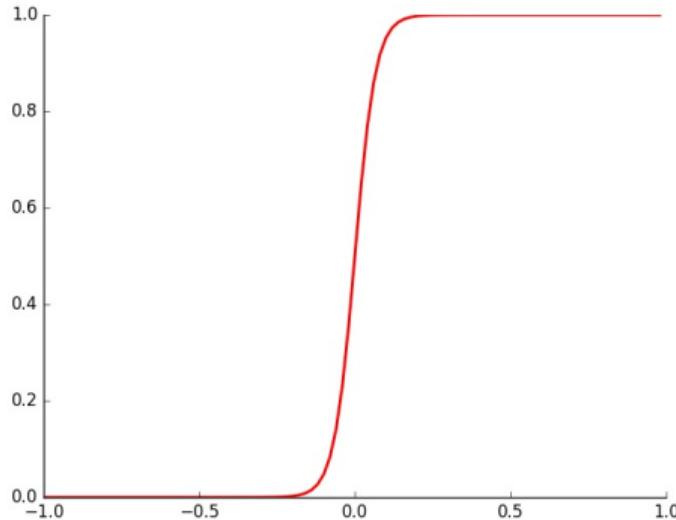
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 22$$



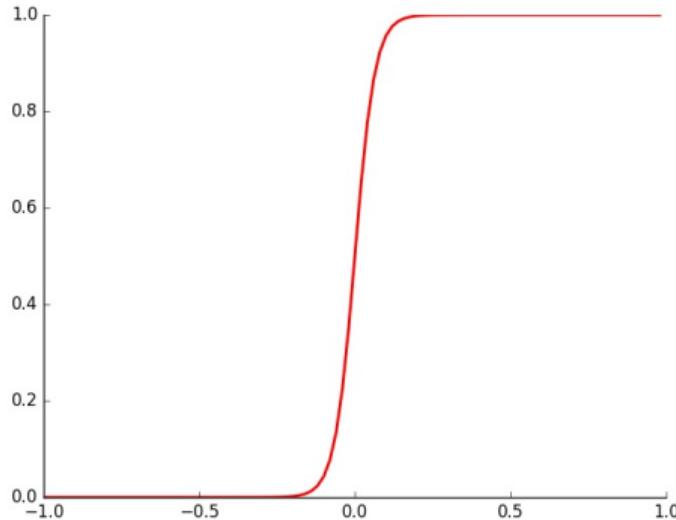
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 23$$



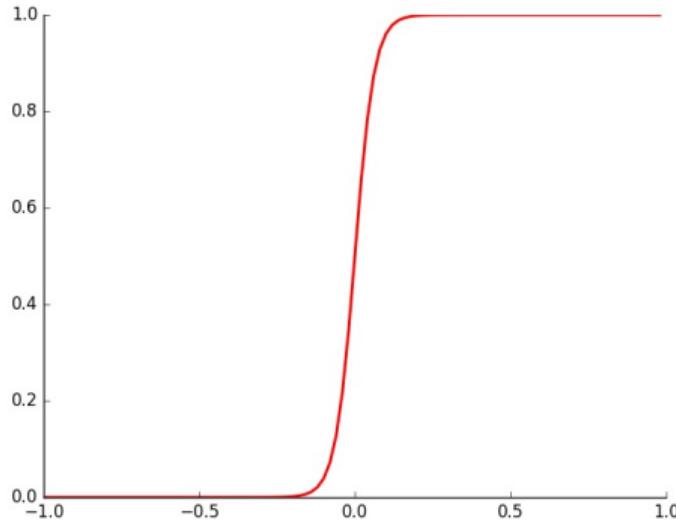
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 24$$



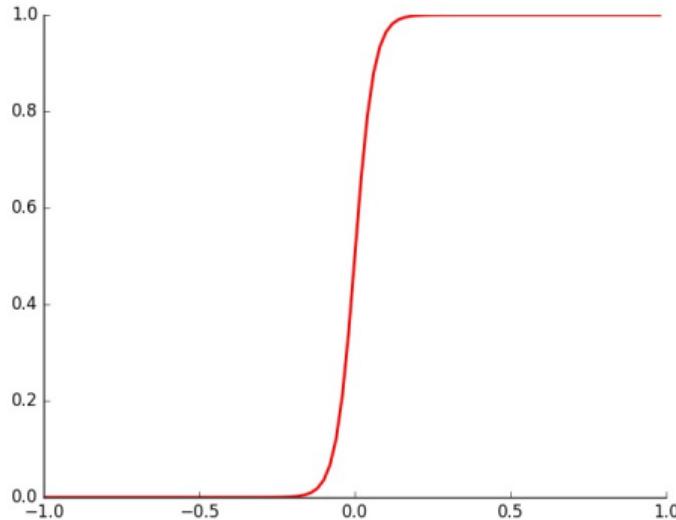
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 25$$



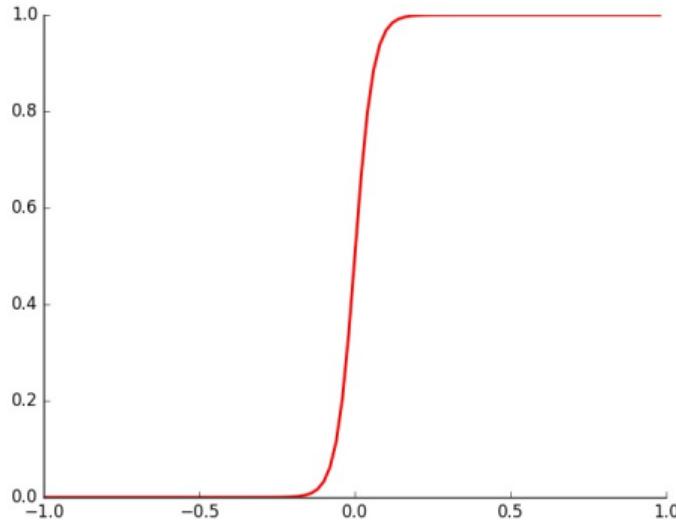
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1 - \exp(-wx + b)} \quad w = 50, b = 26$$



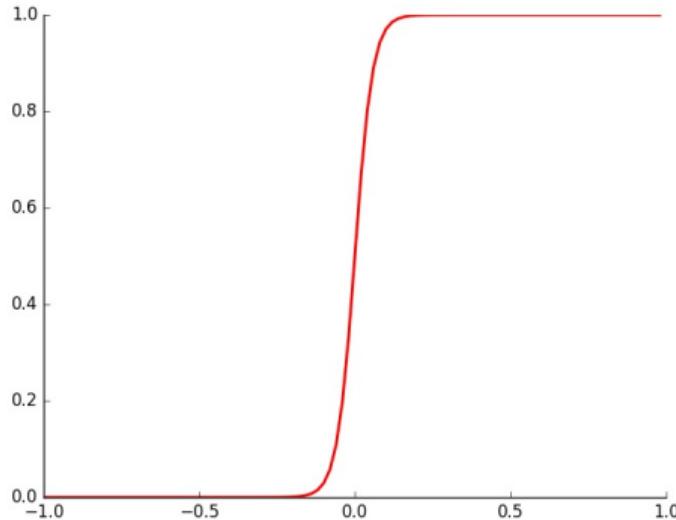
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 27$$



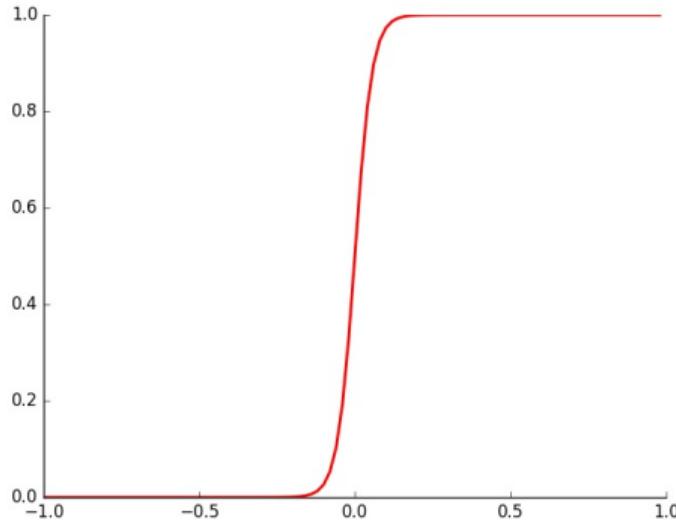
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 28$$



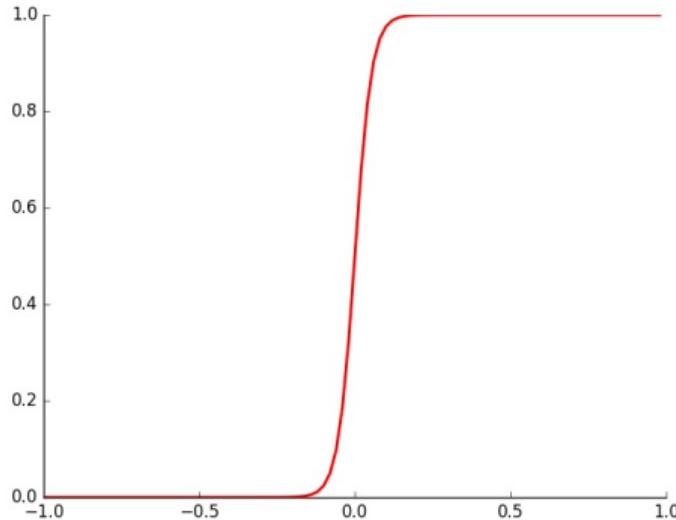
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 29$$



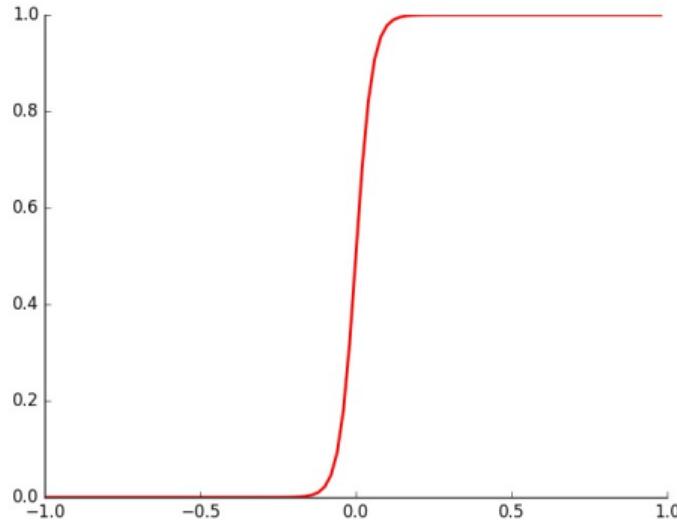
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 30$$



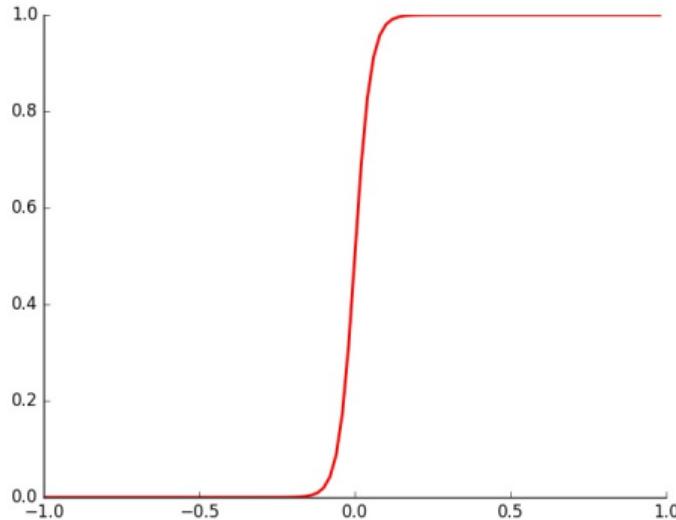
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 31$$



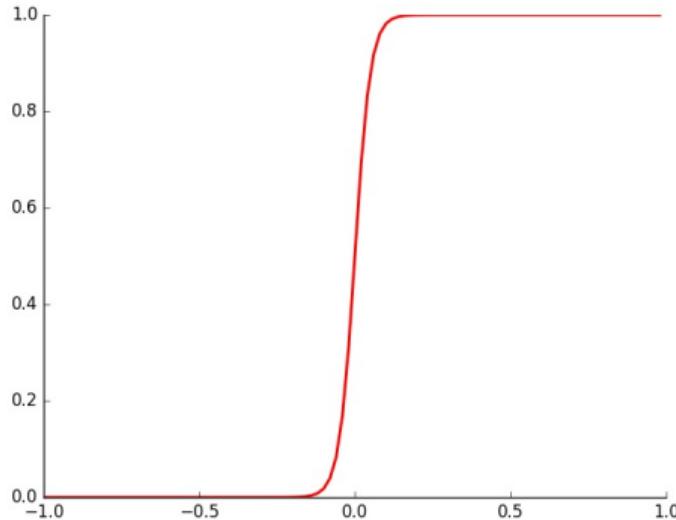
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 32$$



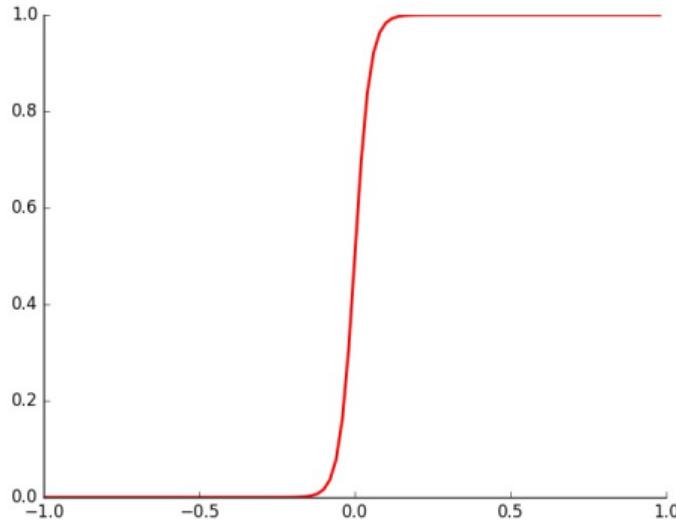
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 33$$



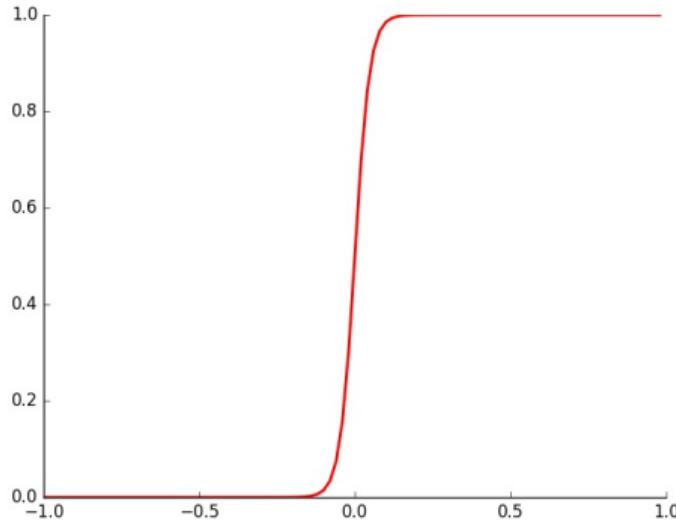
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 34$$



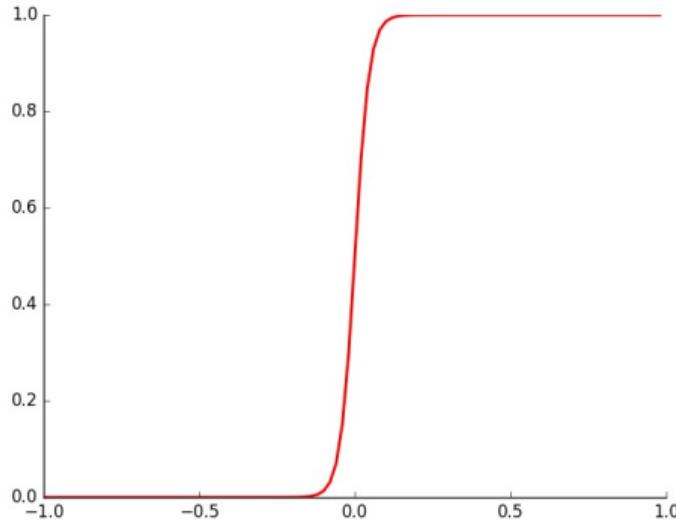
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 35$$



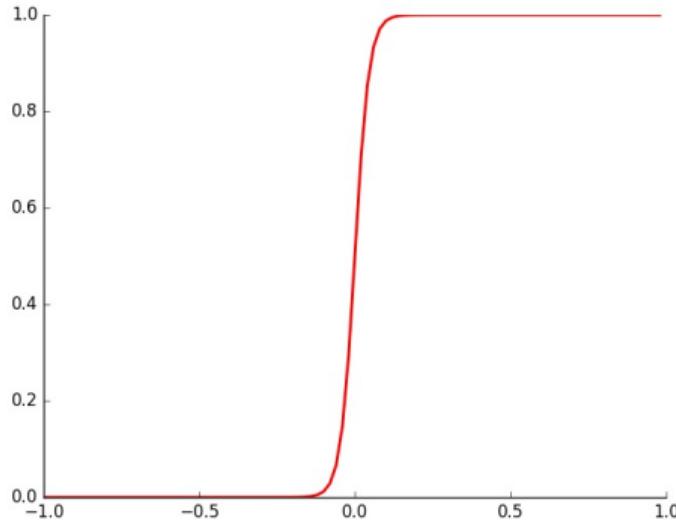
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 36$$



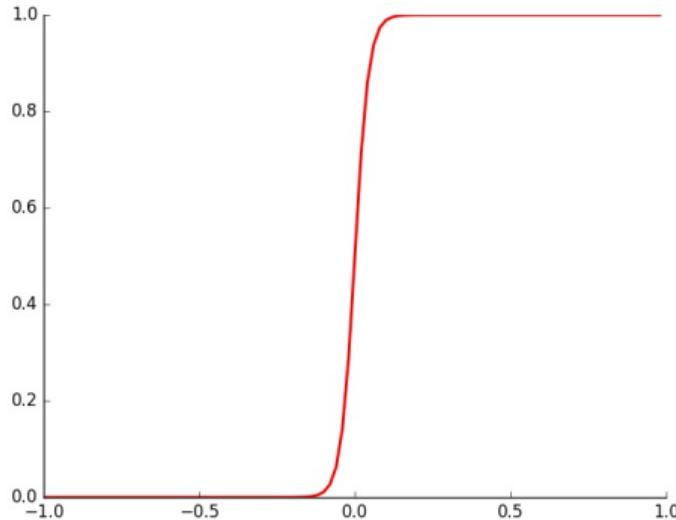
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 37$$



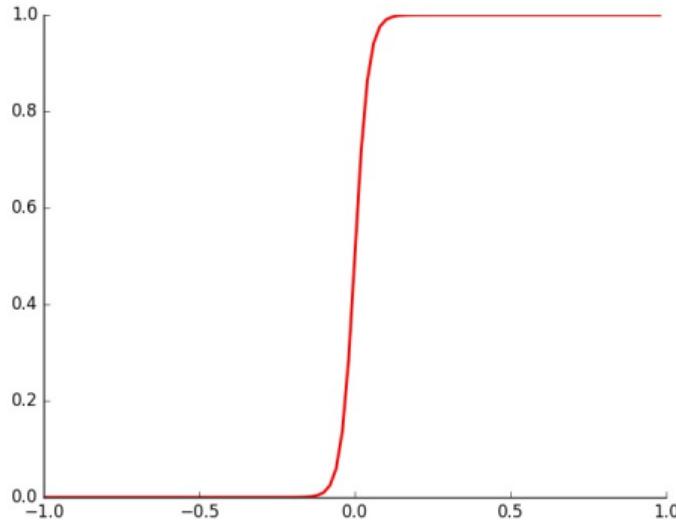
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 38$$



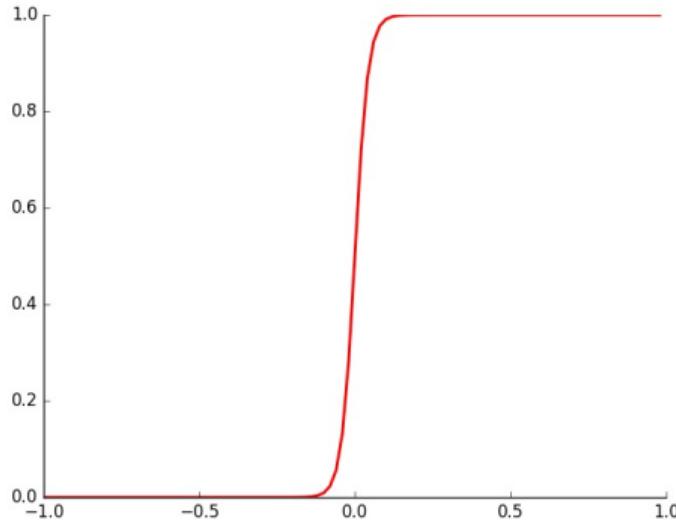
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 39$$



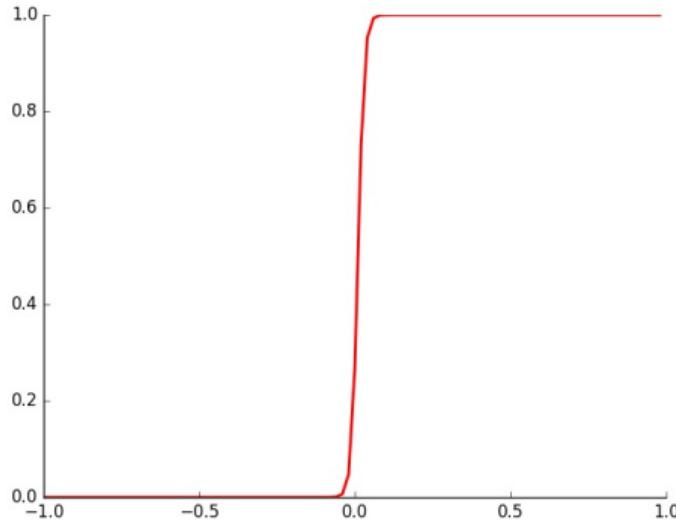
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 40$$



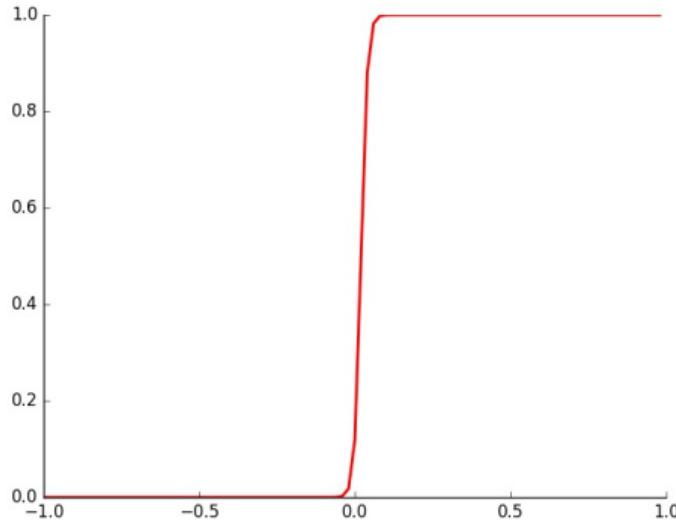
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 41$$



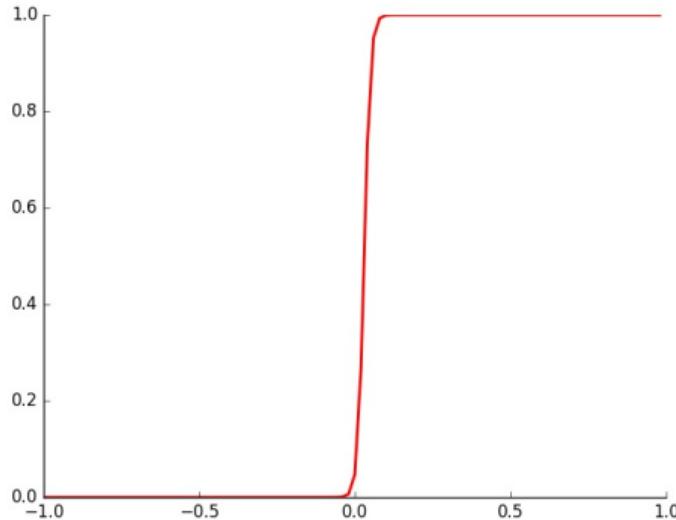
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 1$$



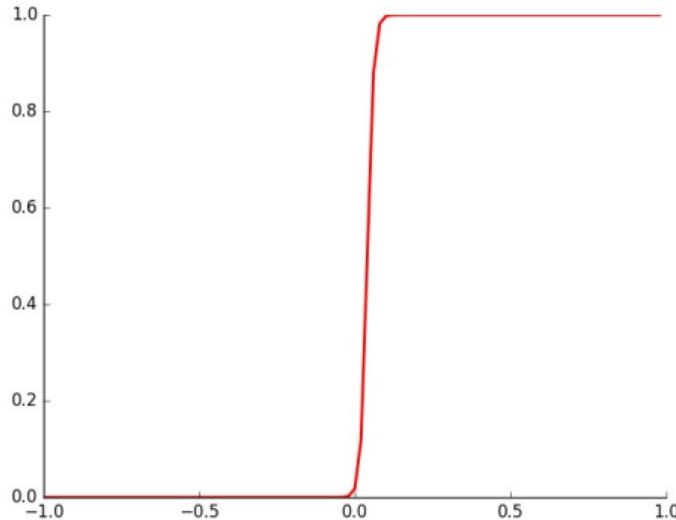
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 2$$



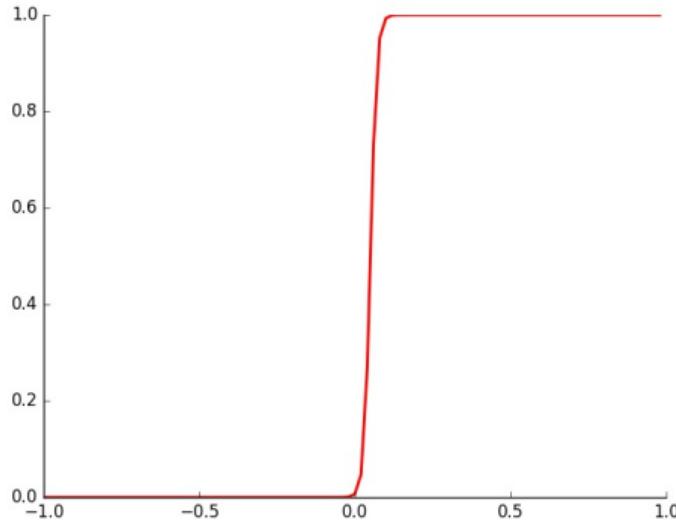
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 3$$



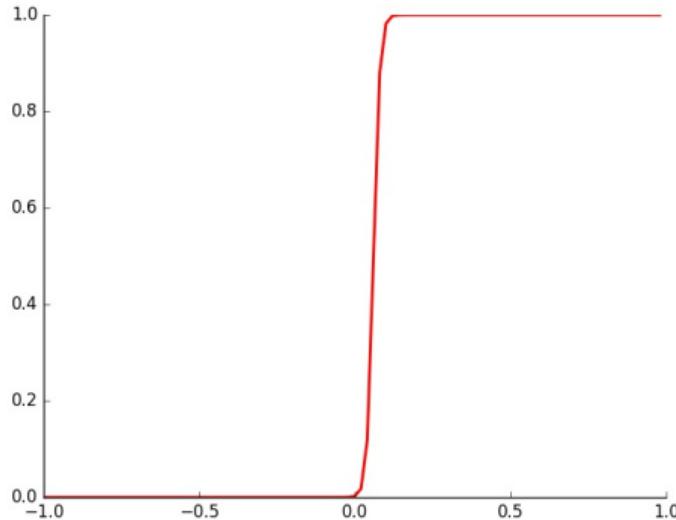
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 4$$



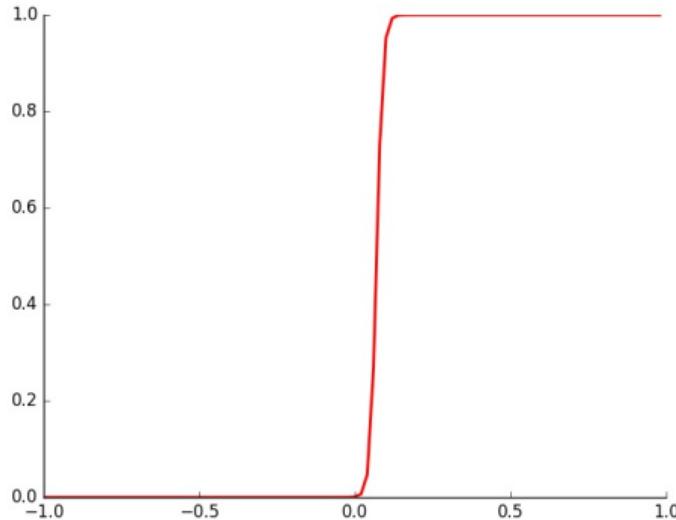
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 5$$



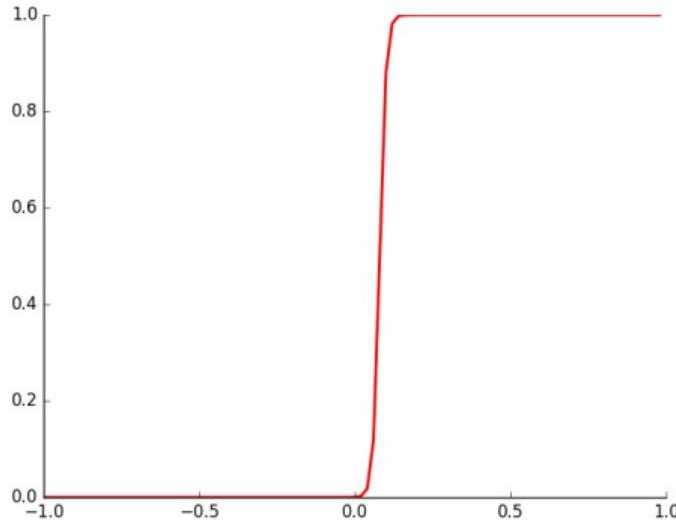
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 6$$



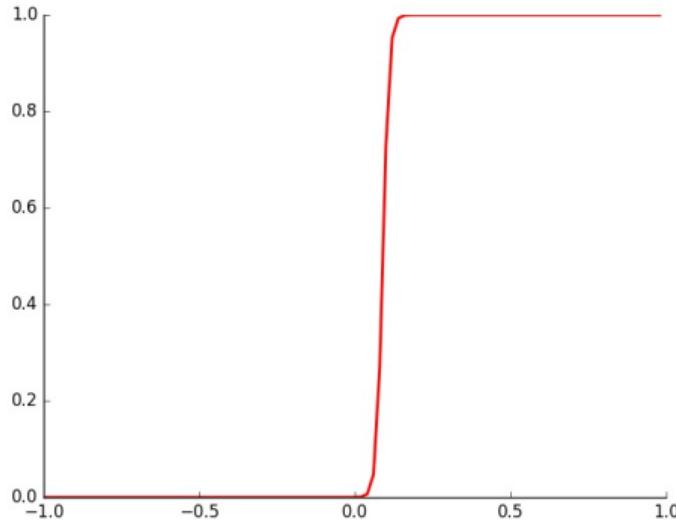
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 7$$



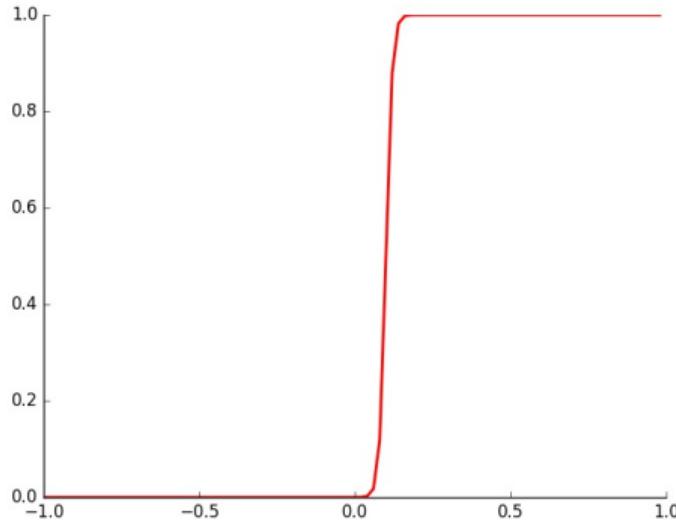
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 8$$



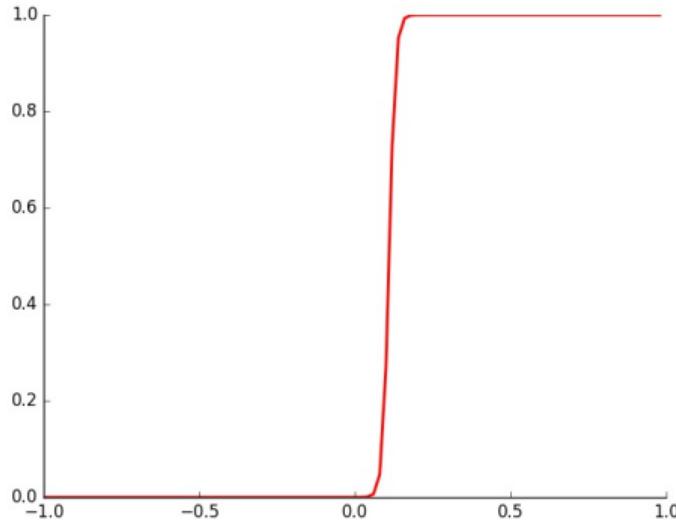
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 9$$



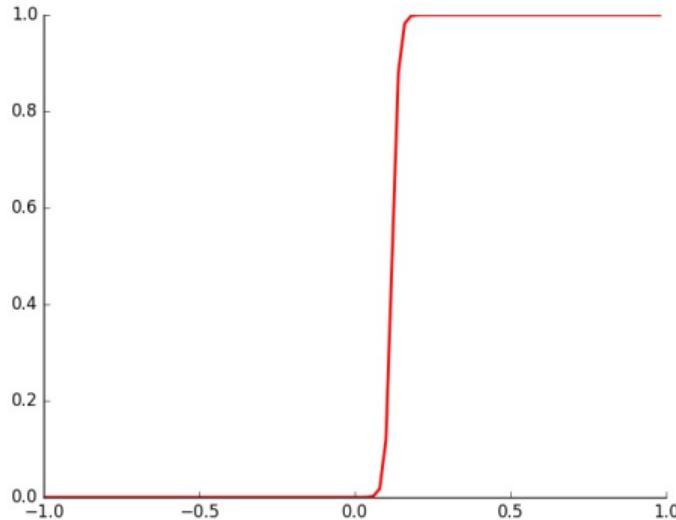
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 10$$



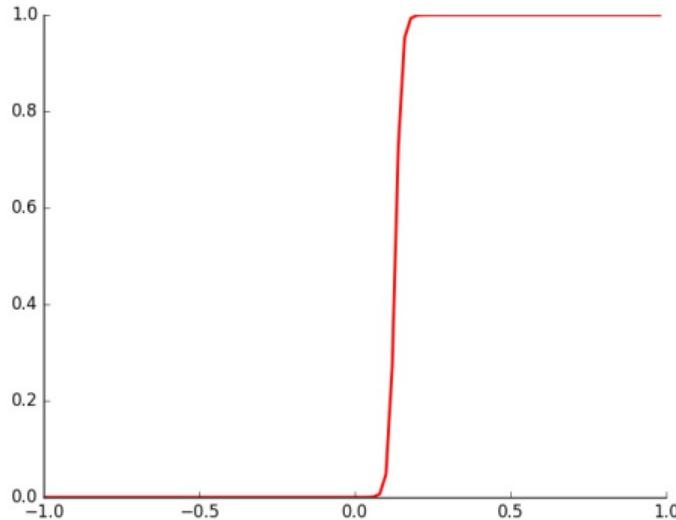
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 11$$



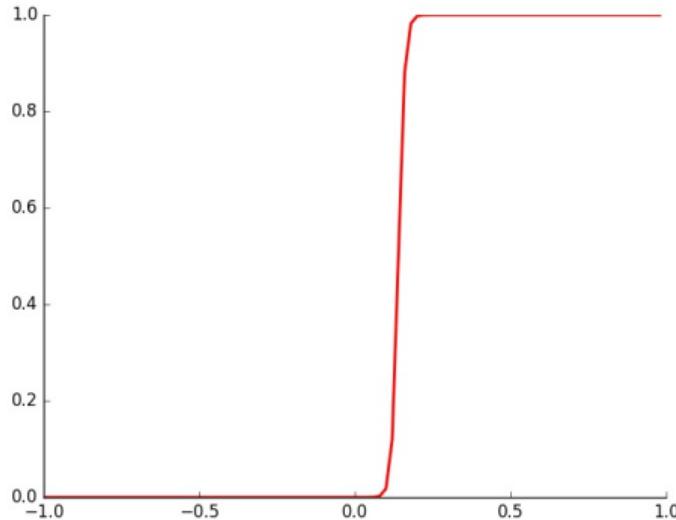
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 12$$



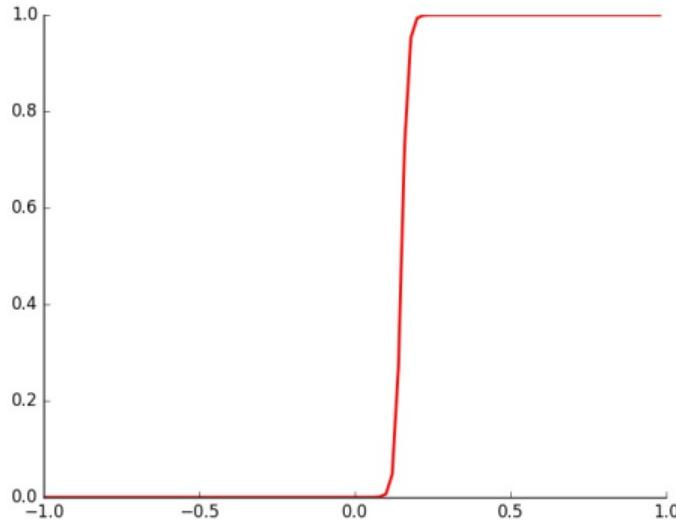
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 13$$



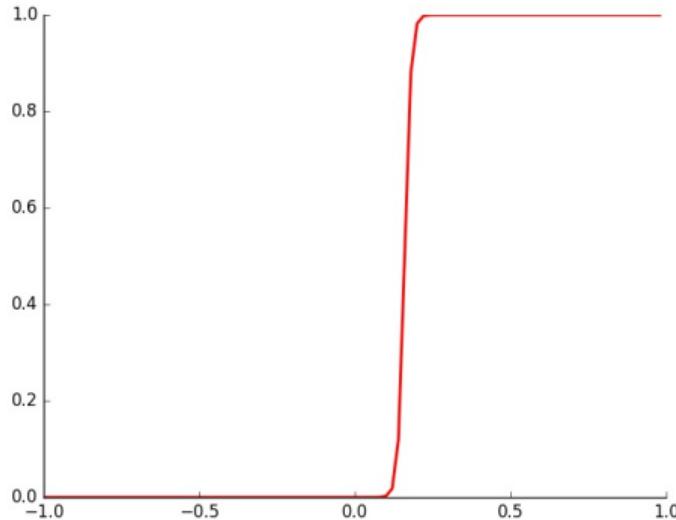
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 14$$



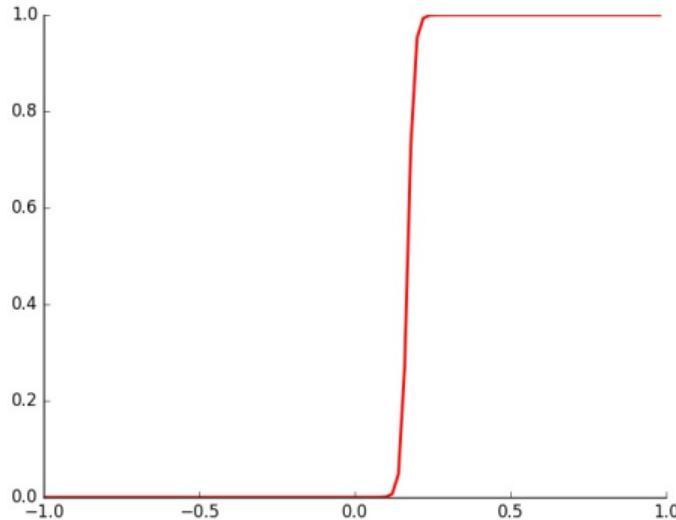
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 15$$



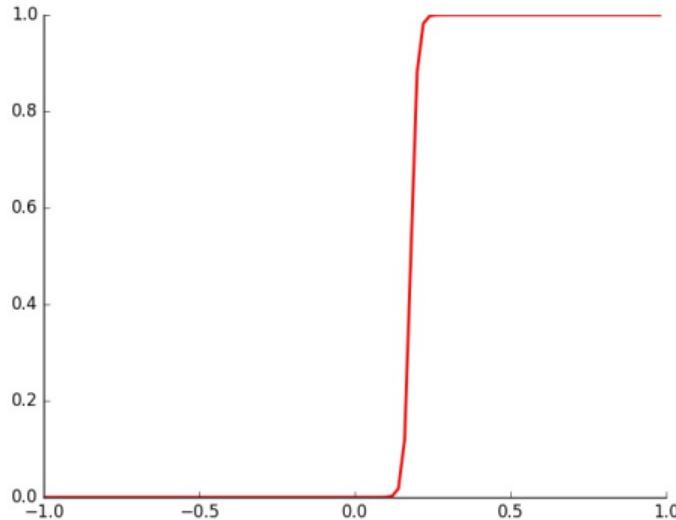
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 16$$



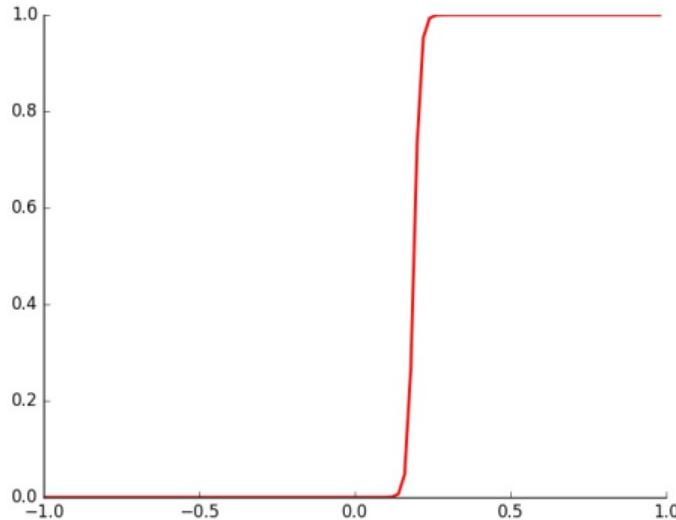
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 17$$



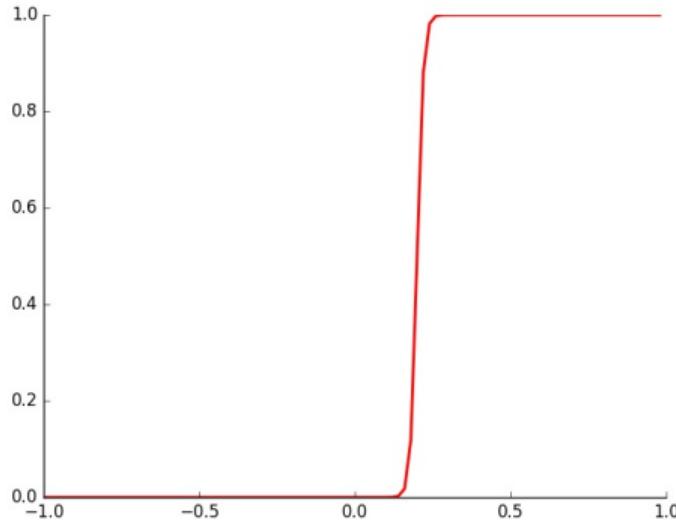
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 18$$



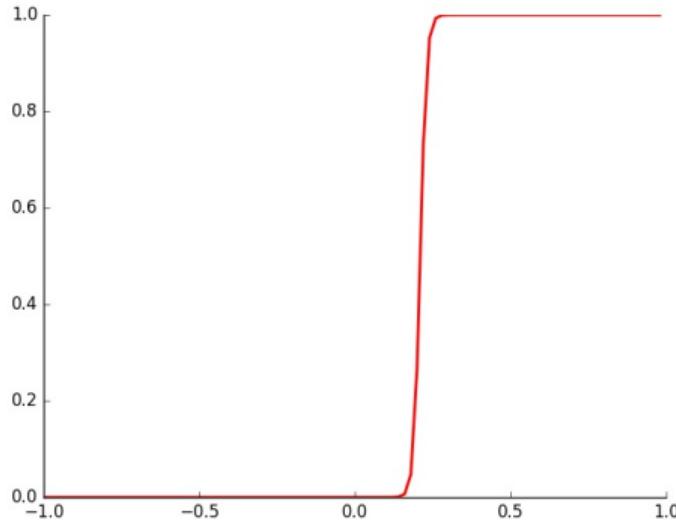
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 19$$



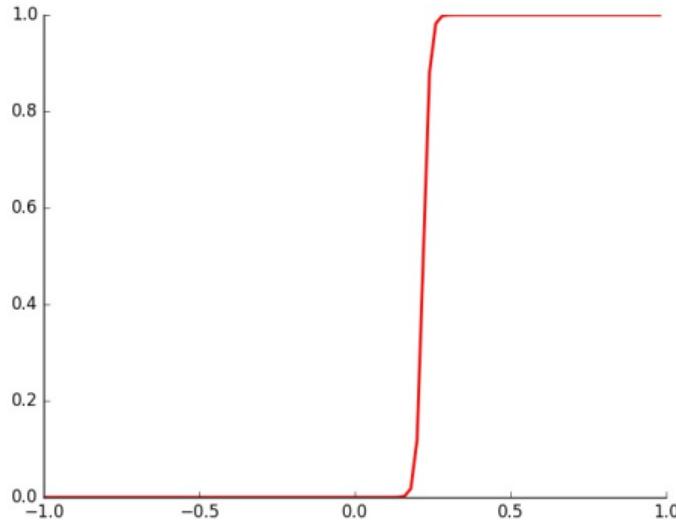
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 20$$



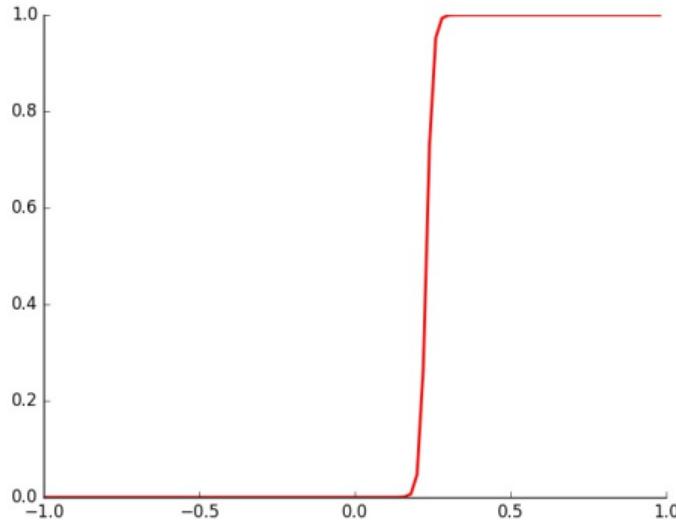
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 21$$



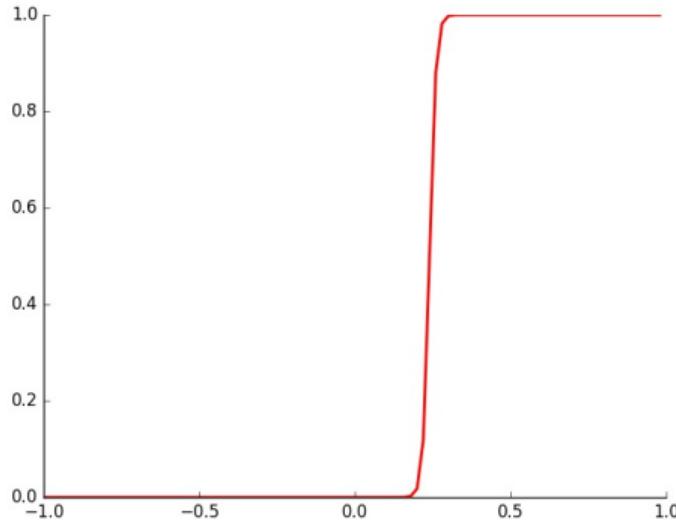
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 22$$



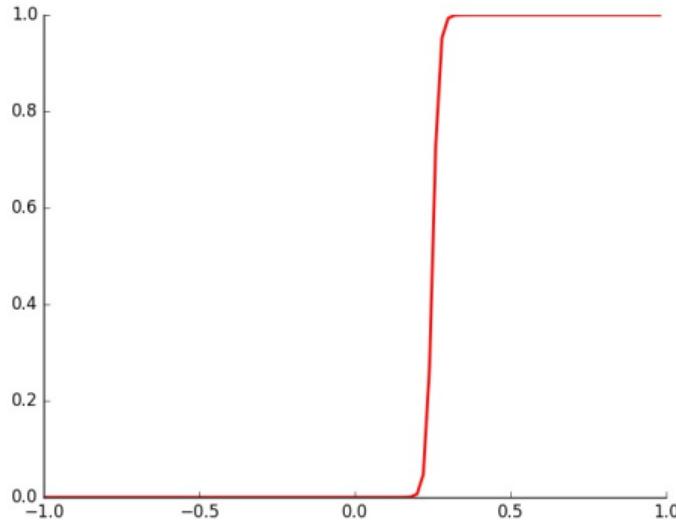
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 23$$



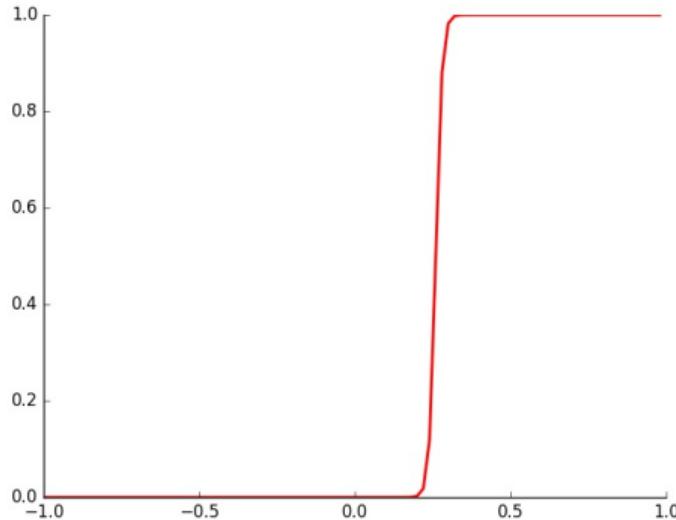
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 24$$



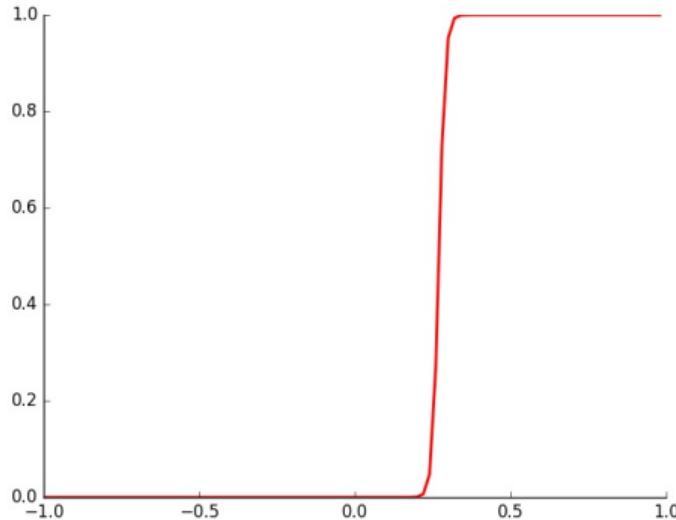
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 25$$



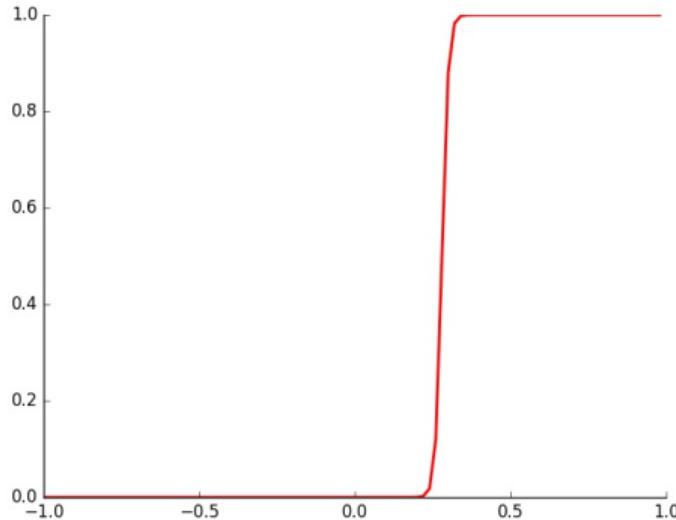
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 26$$



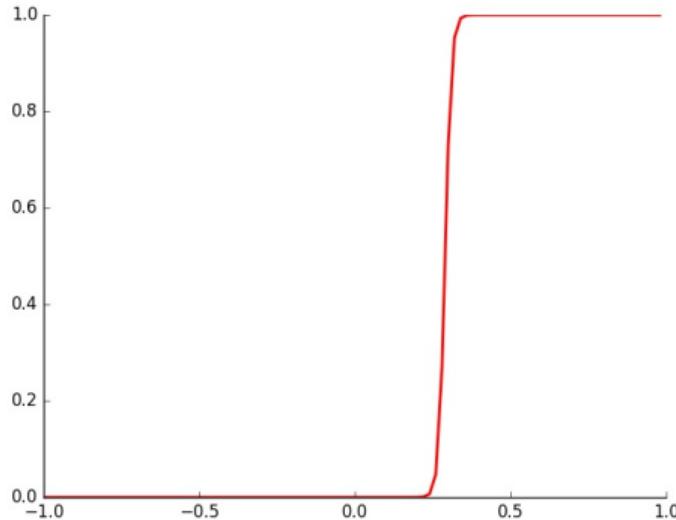
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 27$$



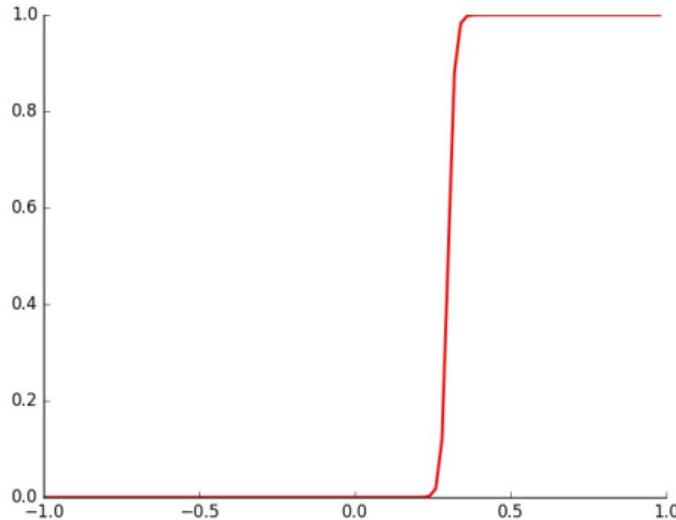
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 28$$



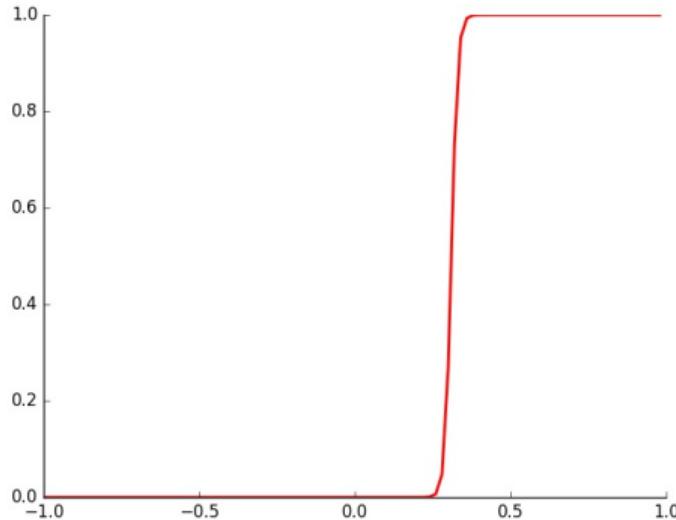
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 29$$



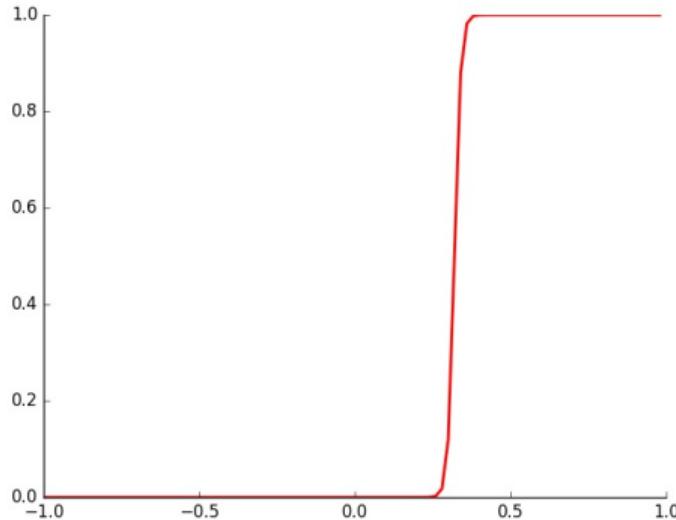
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 30$$



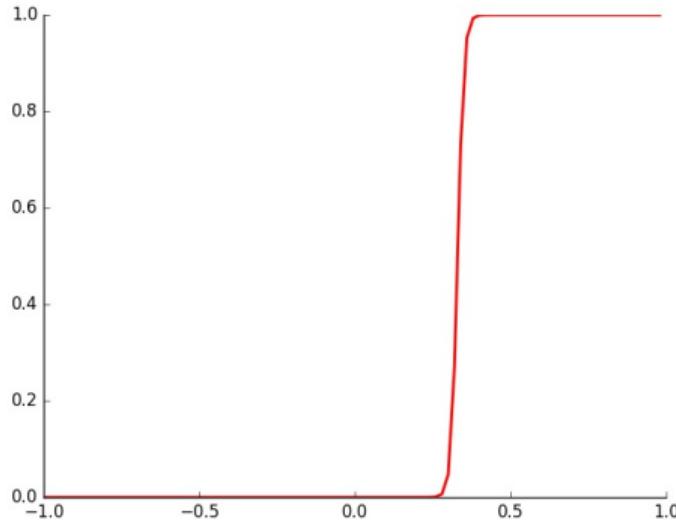
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 31$$



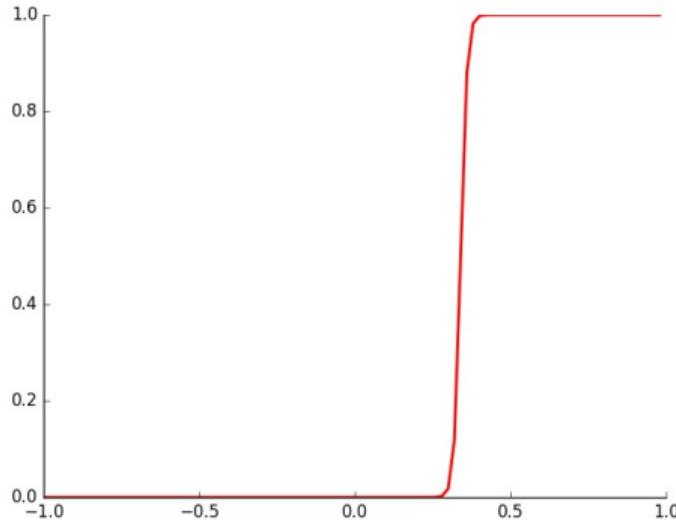
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 32$$



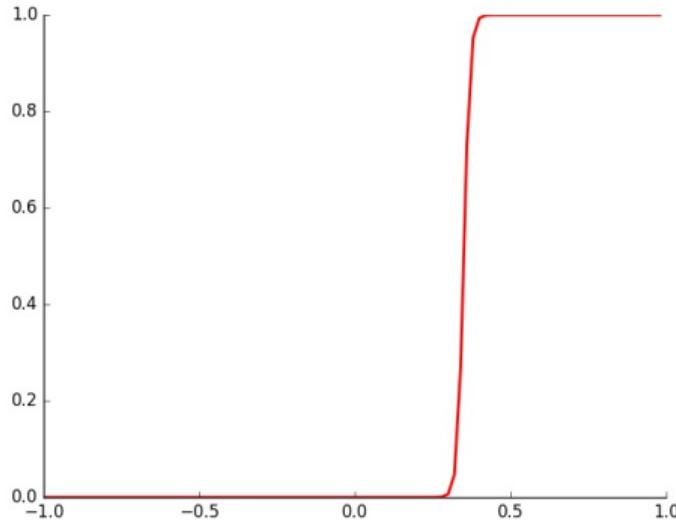
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 33$$



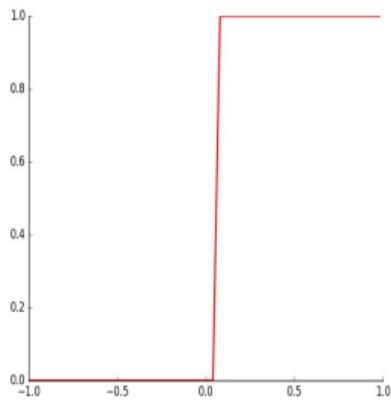
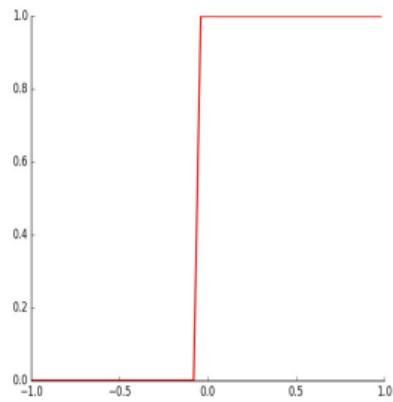
- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

$$\sigma(x) = \frac{1}{1 - \exp^{(wx+b)}} \quad w = 50, b = 34$$

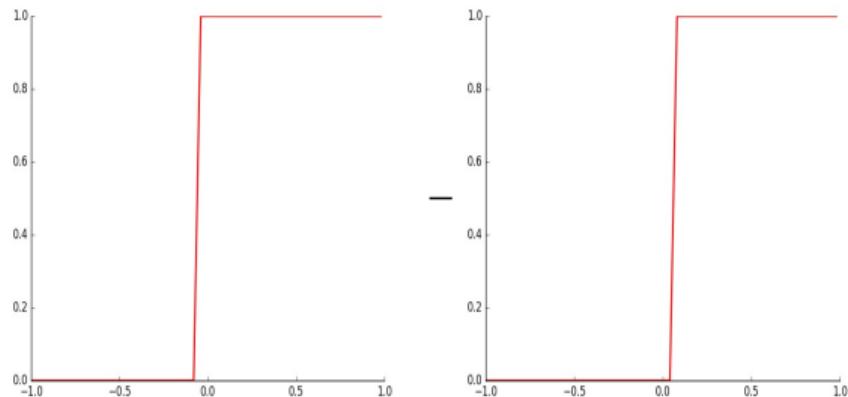


- If we take the logistic function and set  $w$  to a very high value we will recover the step function
- Let us see what happens as we change the value of  $w$
- Further we can adjust the value of  $b$  to control the position on the x-axis at which the function transitions from 0 to 1

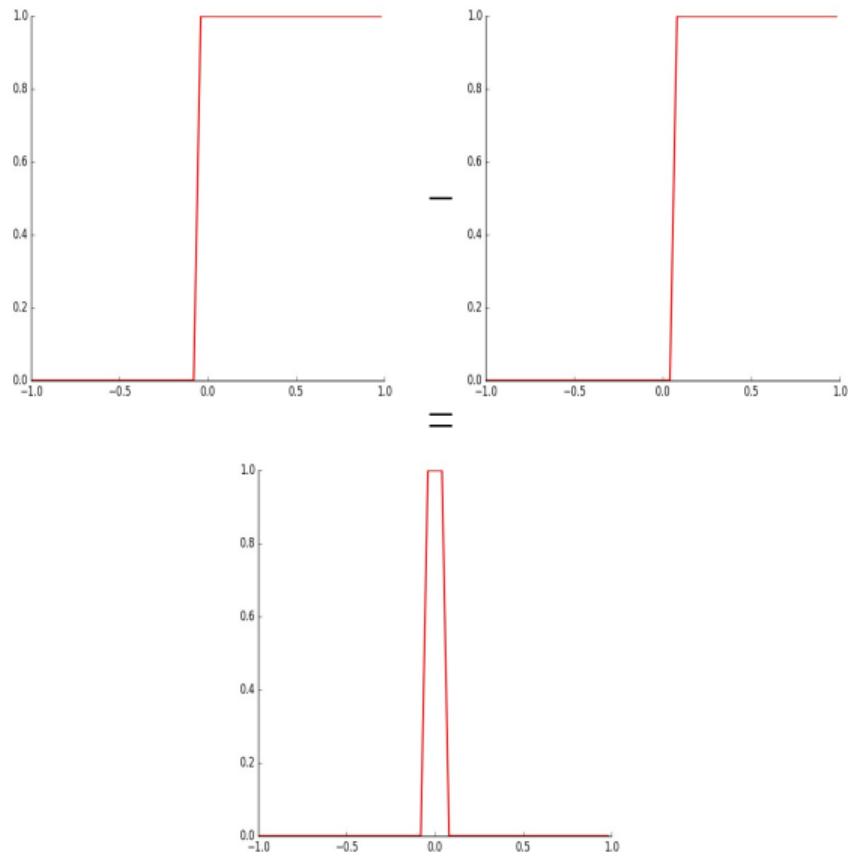
$$\sigma(x) = \frac{1}{1-\exp^{(wx+b)}} \quad w = 50, b = 35$$



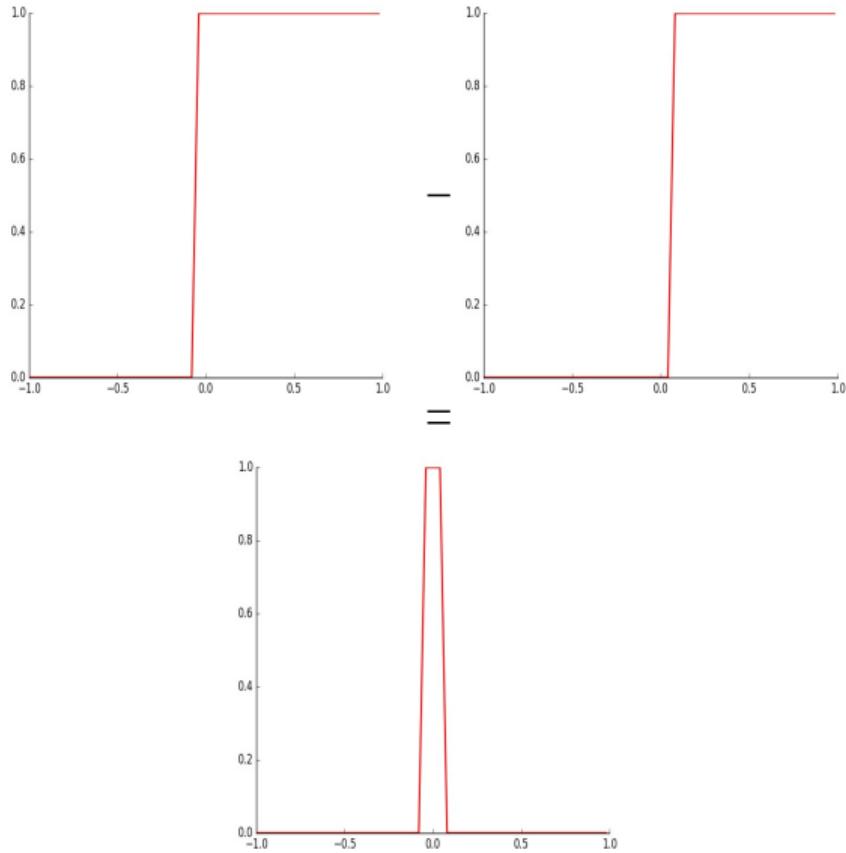
- Now let us see what we get by taking two such sigmoid functions (with different  $b$ 's) and subtracting one from the other



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

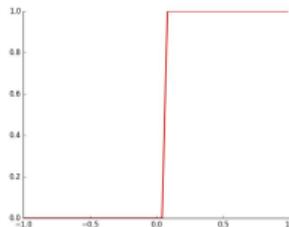
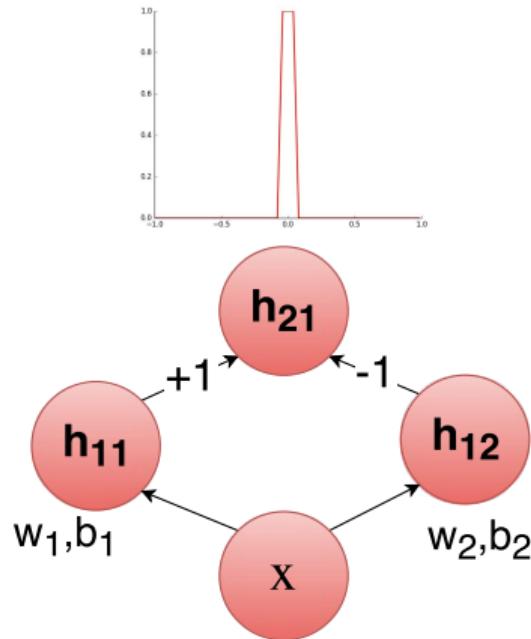
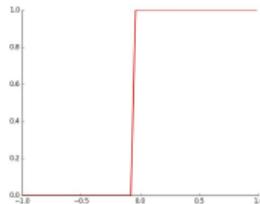


- Now let us see what we get by taking two such sigmoid functions (with different  $b$ 's) and subtracting one from the other

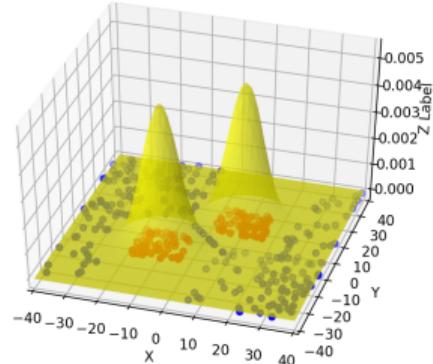


- Now let us see what we get by taking two such sigmoid functions (with different  $b$ 's) and subtracting one from the other
- Voila! We have our tower function !!

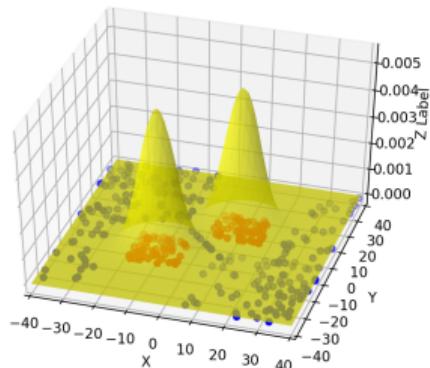
- Can we come up with a neural network to represent this operation of subtracting one sigmoid function from another ?



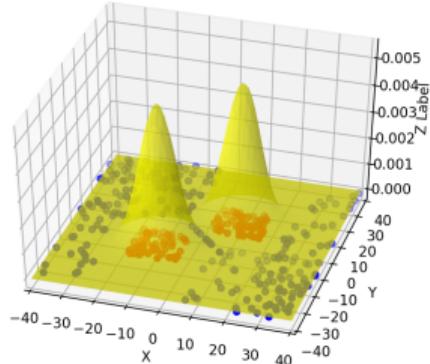
- What if we have more than one input?

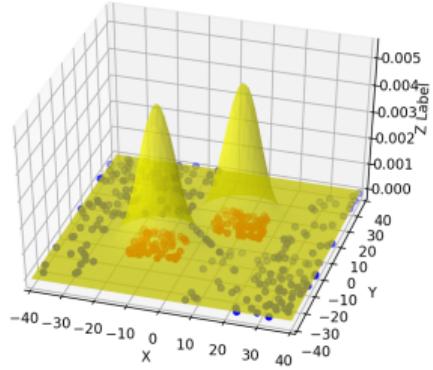


- What if we have more than one input?
- Suppose we are trying to take a decision about whether we will find oil at a particular location on the ocean bed(Yes/No)

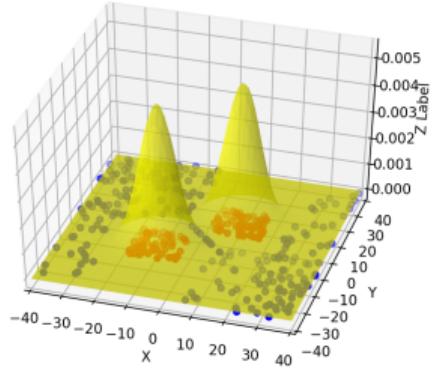


- What if we have more than one input?
- Suppose we are trying to take a decision about whether we will find oil at a particular location on the ocean bed(Yes/No)
- Further, suppose we base our decision on two factors: Salinity ( $x_1$ ) and Pressure ( $x_2$ )





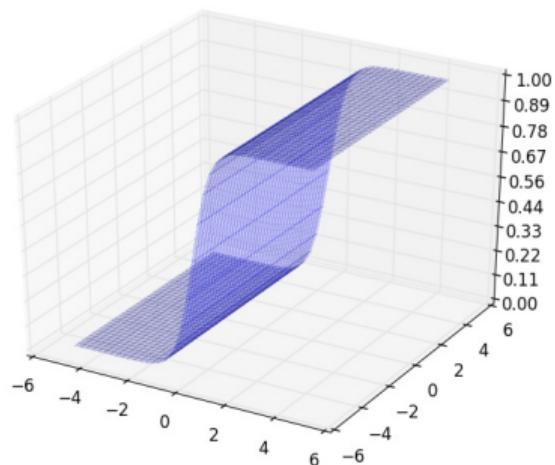
- What if we have more than one input?
- Suppose we are trying to take a decision about whether we will find oil at a particular location on the ocean bed(Yes/No)
- Further, suppose we base our decision on two factors: Salinity ( $x_1$ ) and Pressure ( $x_2$ )
- We are given some data and it seems that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



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

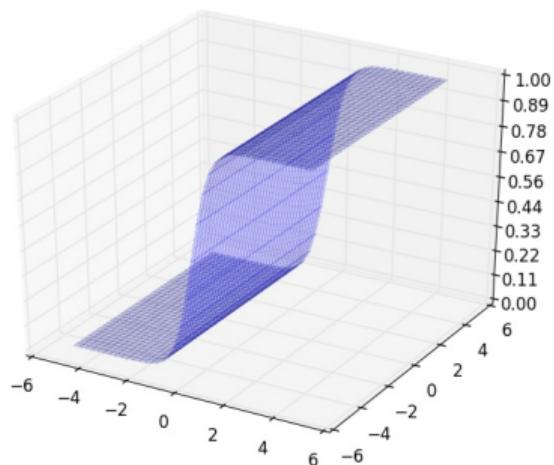
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

- This is what a 2-dimensional sigmoid looks like

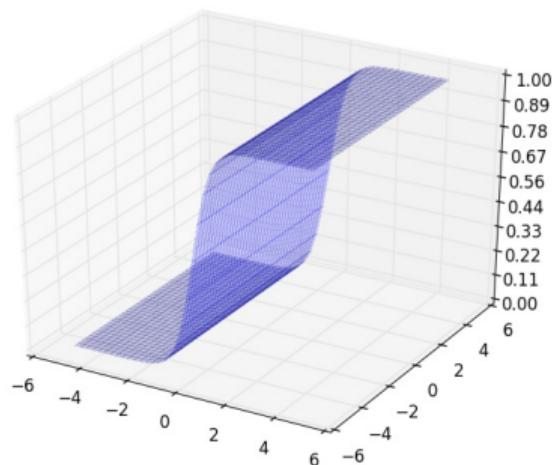


$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

- This is what a 2-dimensional sigmoid looks like
- We need to figure out how to get a tower in this case



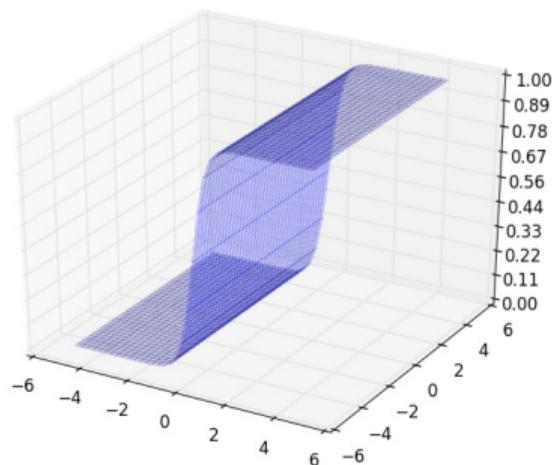
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

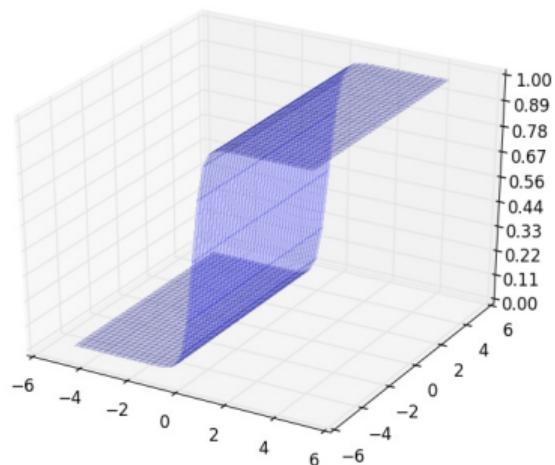
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

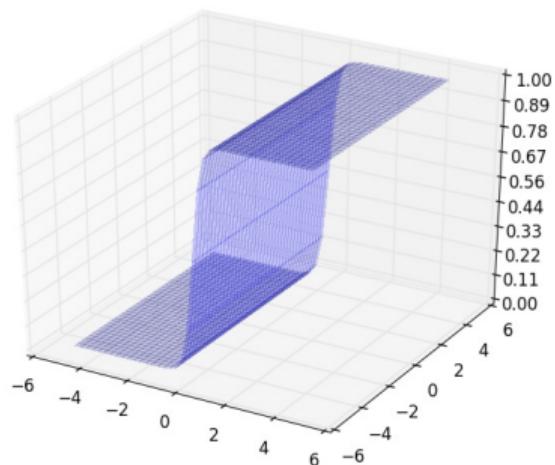
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

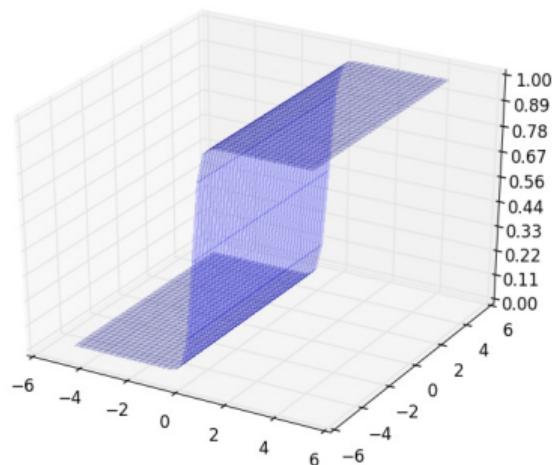
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

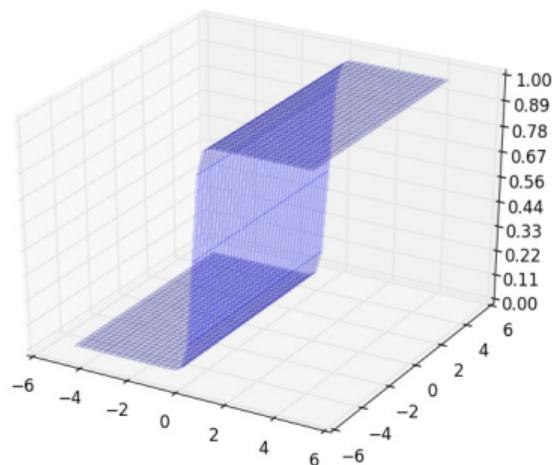
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

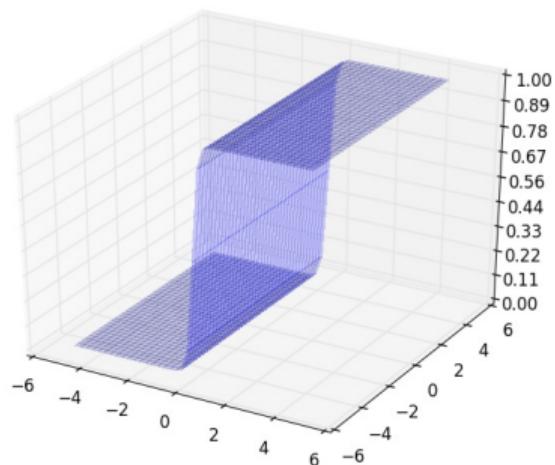
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

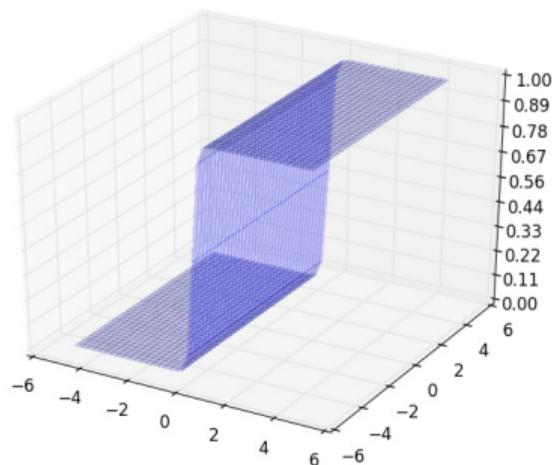
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

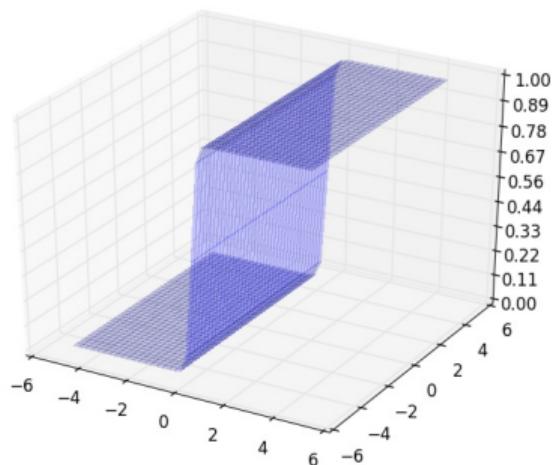
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

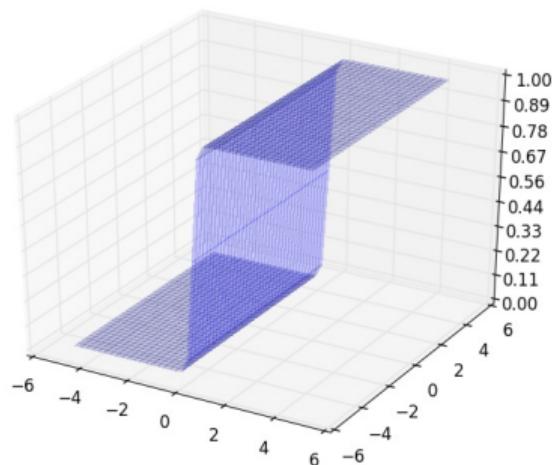
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

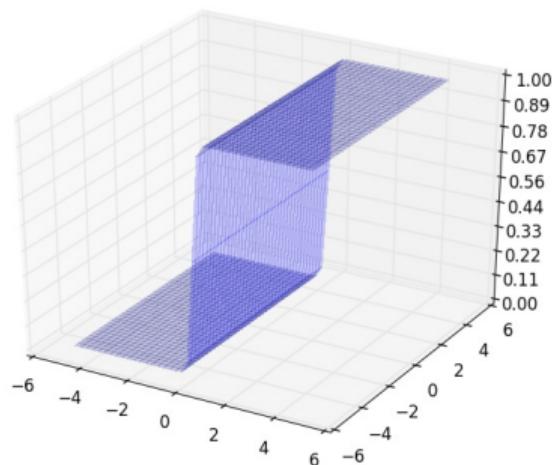
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

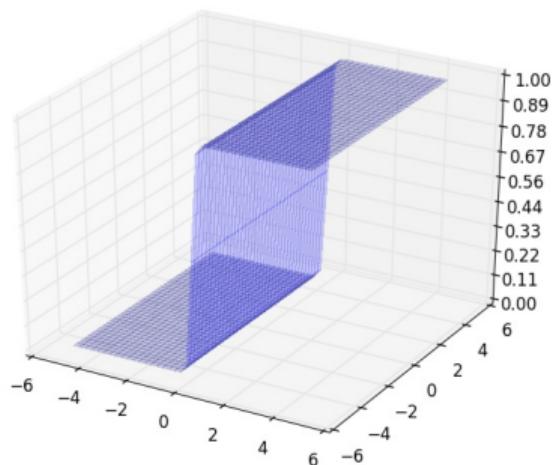
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

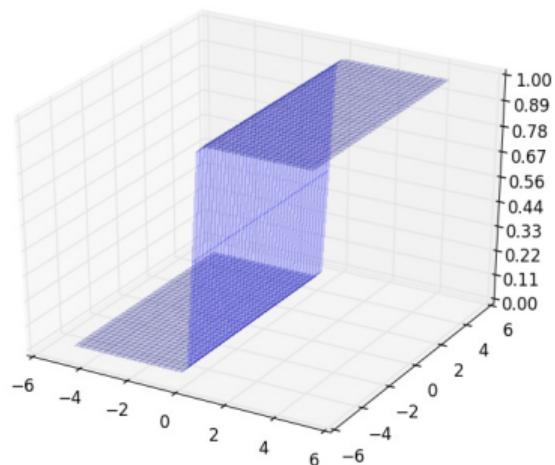
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

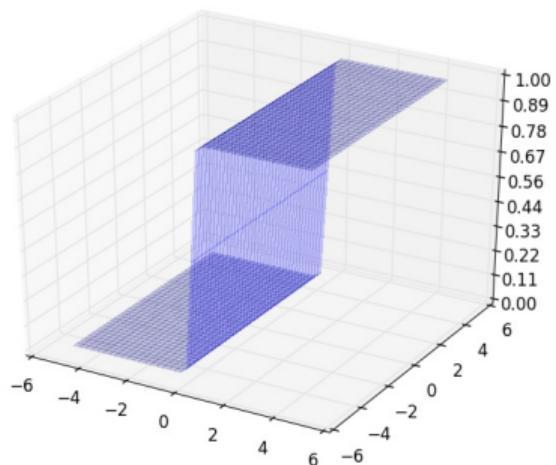
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

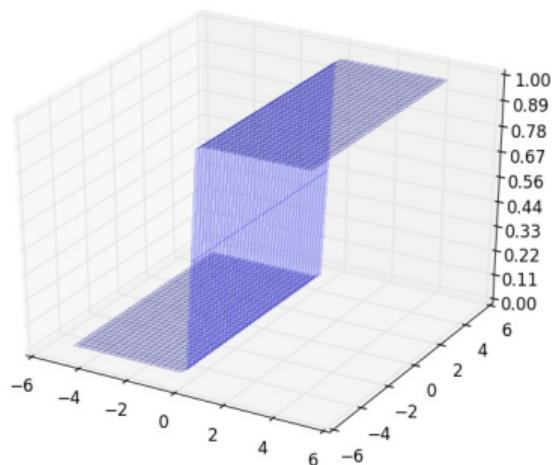
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

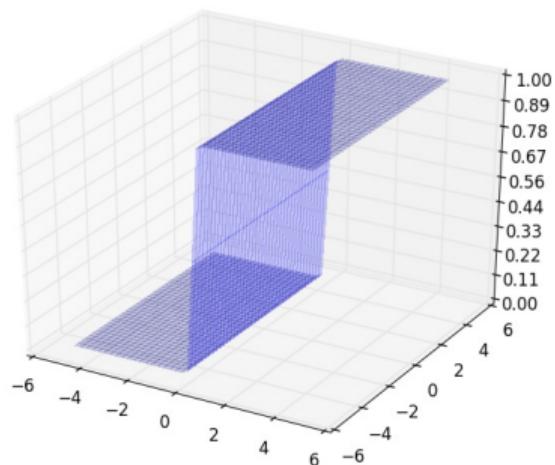
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

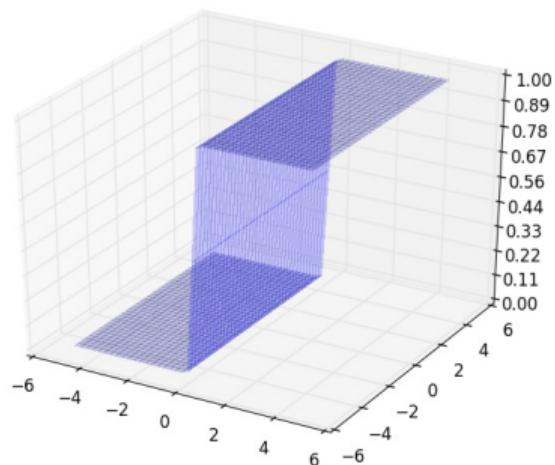
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

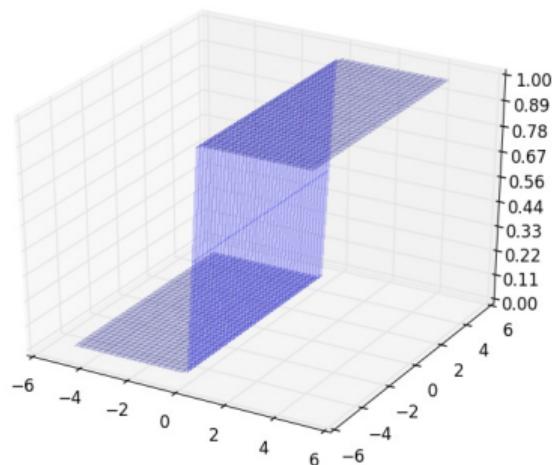
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

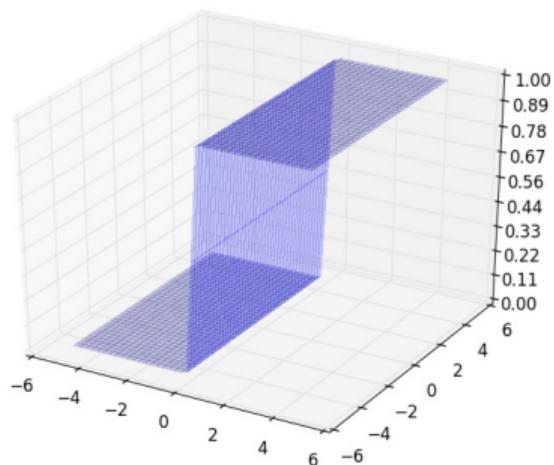
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

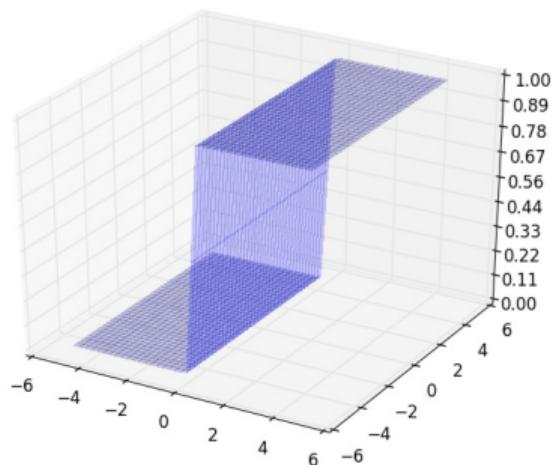
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

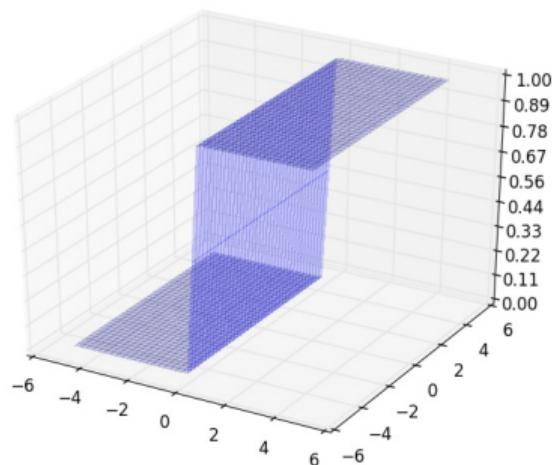
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

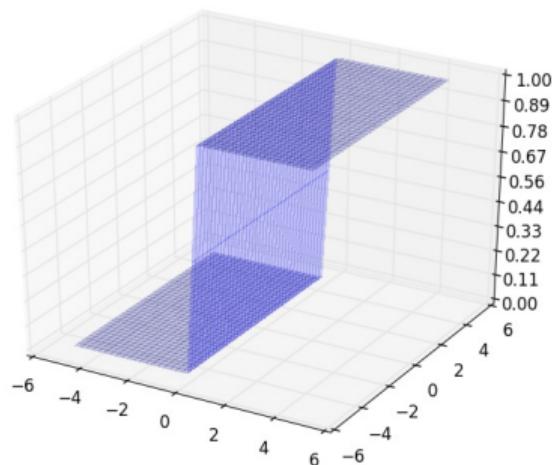
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

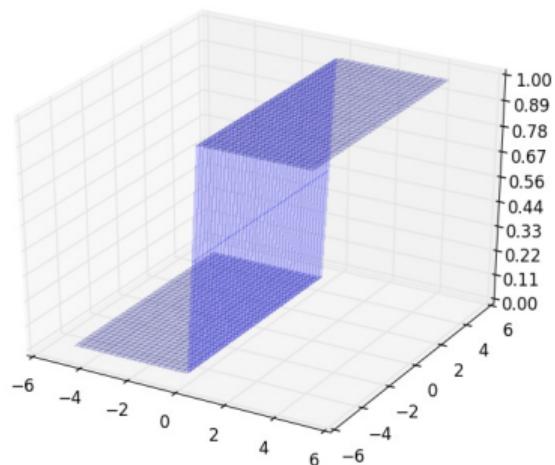
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

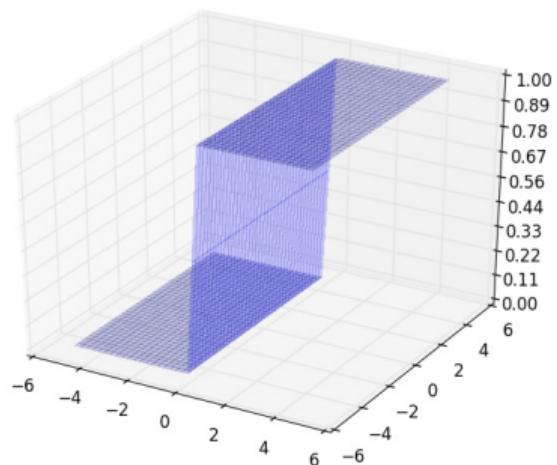
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

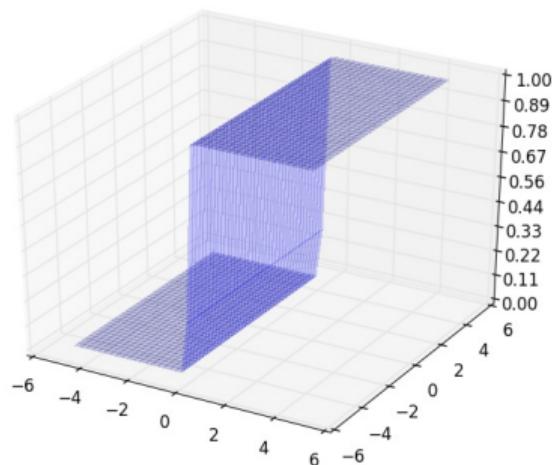
- This is what a 2-dimensional sigmoid looks like
- We need to figure out how to get a tower in this case
- First, let us set  $w_2$  to 0 and see if we can get a two dimensional step function

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

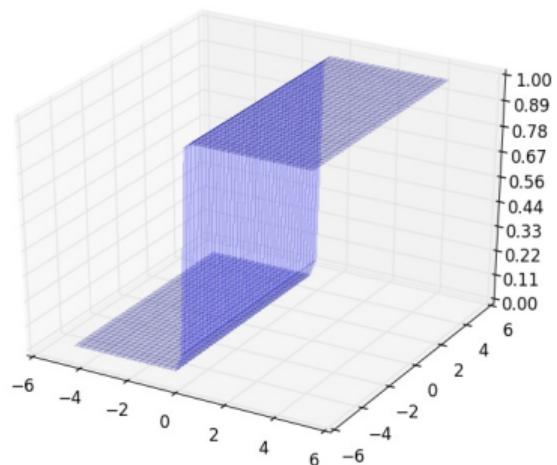
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

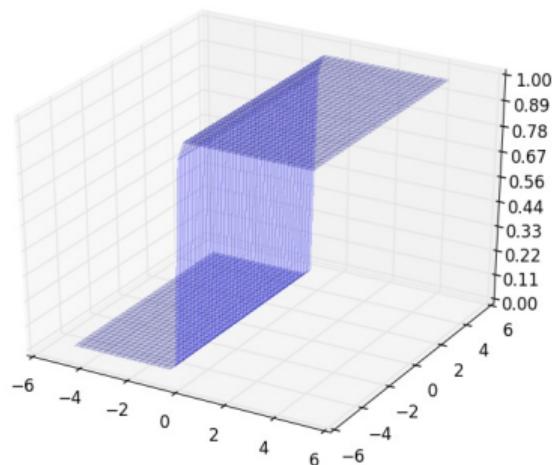
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

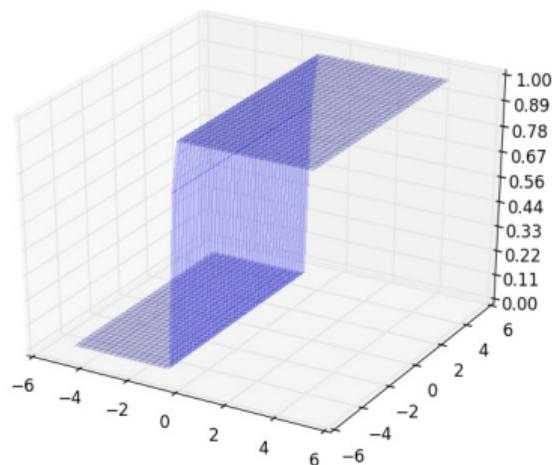
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

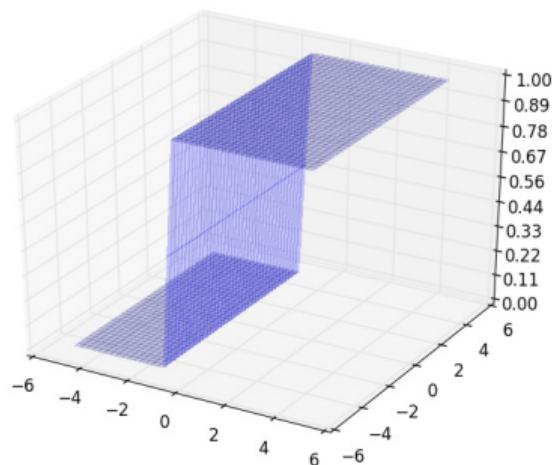
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

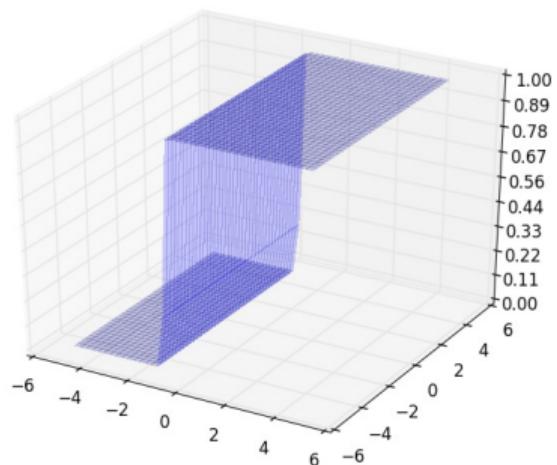
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

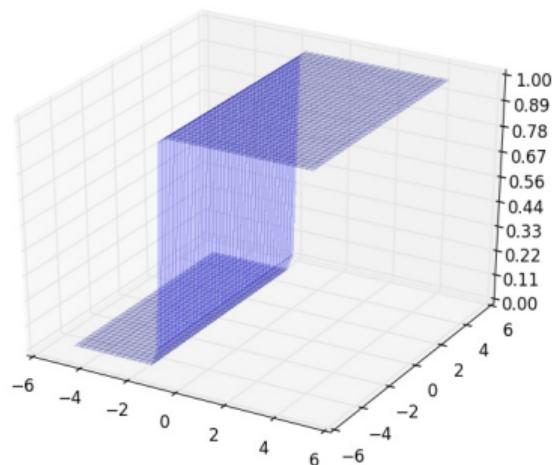
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

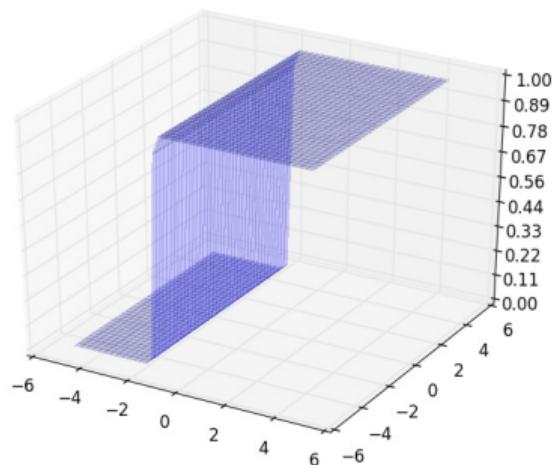
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

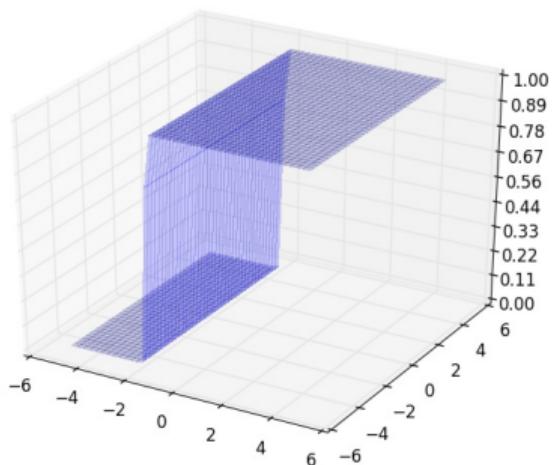
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$



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

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

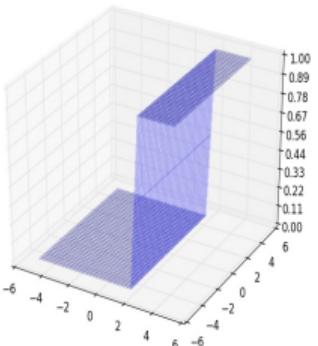
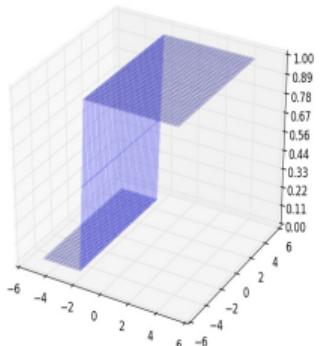
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

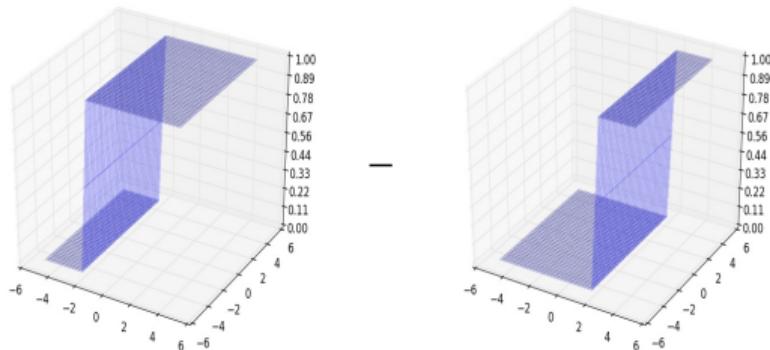


$$w_1 = 25, w_2 = 0, b = 45$$

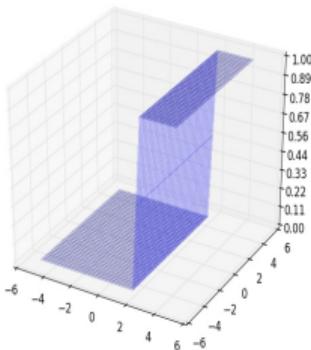
- This is what a 2-dimensional sigmoid looks like
- We need to figure out how to get a tower in this case
- First, let us set  $w_2$  to 0 and see if we can get a two dimensional step function
- What would happen if we change  $b$  ?

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

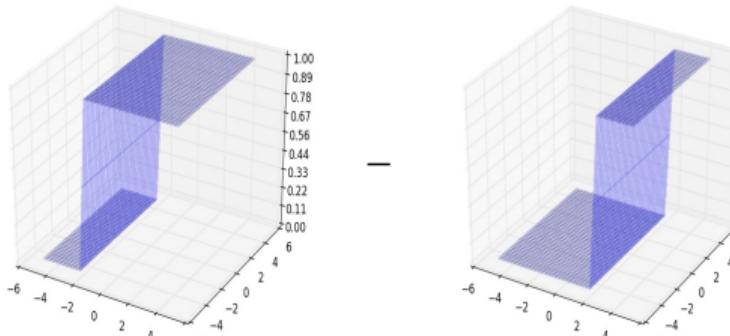




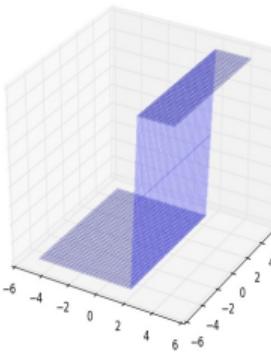
- What if we take two such step functions (with different  $b$  values) and subtract one from the other



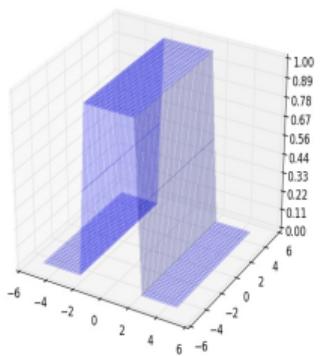
- What if we take two such step functions (with different  $b$  values) and subtract one from the other

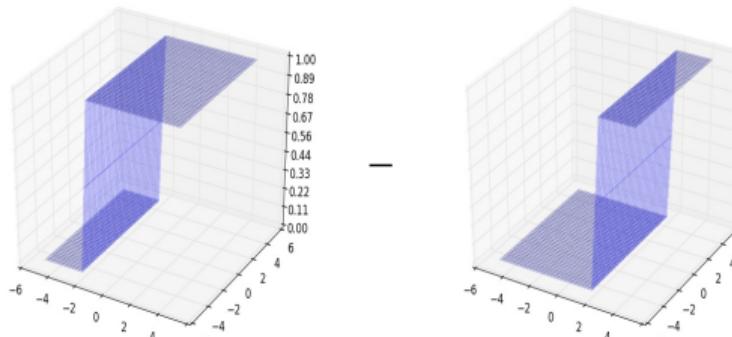


-

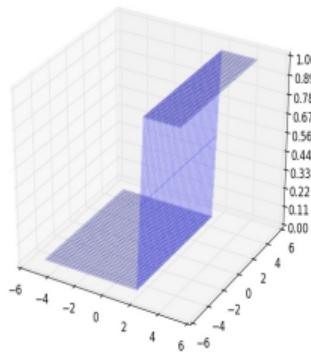


=



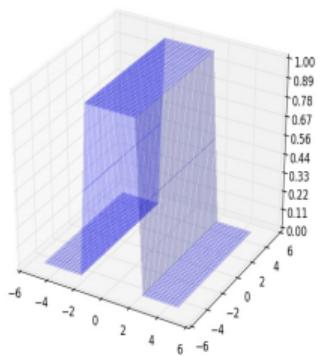


—



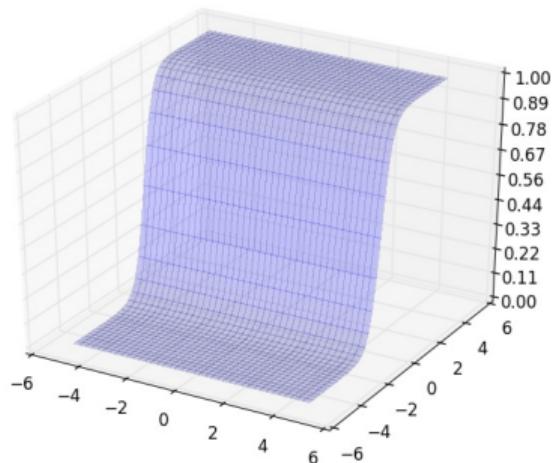
- What if we take two such step functions (with different  $b$  values) and subtract one from the other
- We still don't get a tower (or we get a tower which is open from two sides)

=



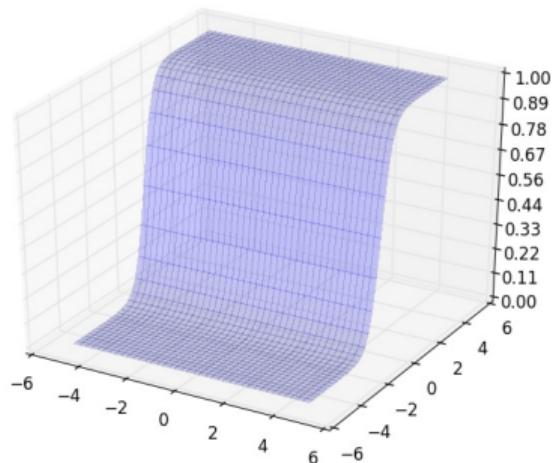
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



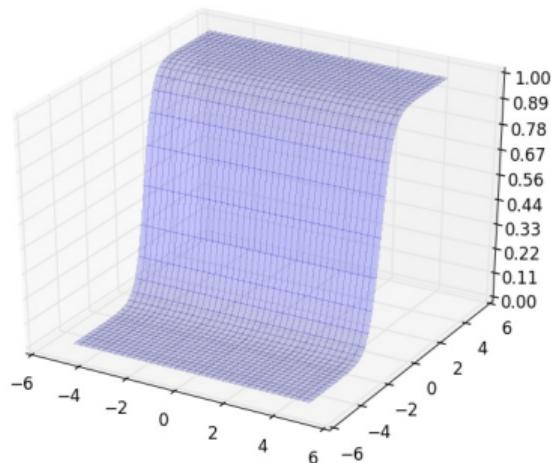
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

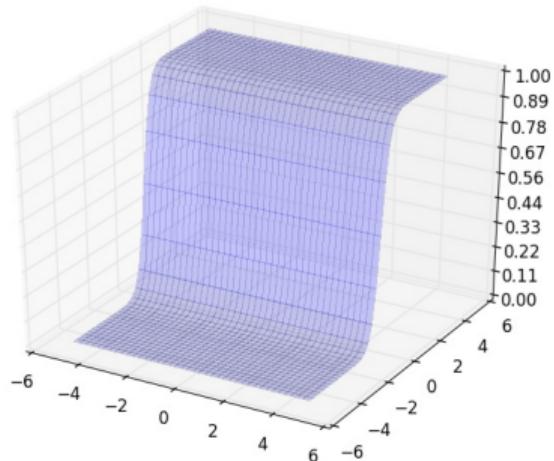
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

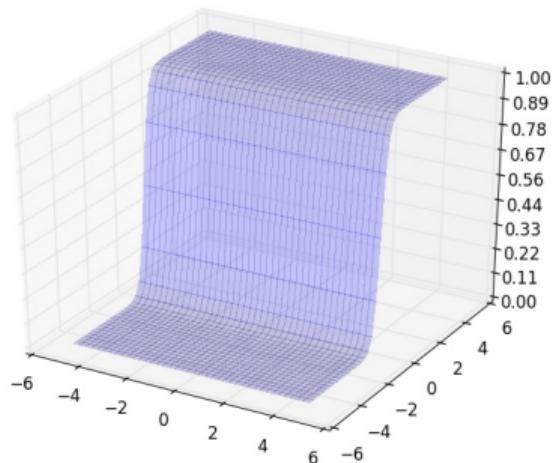
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

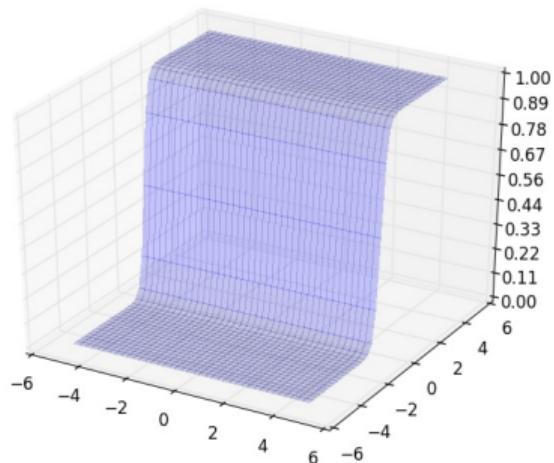
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

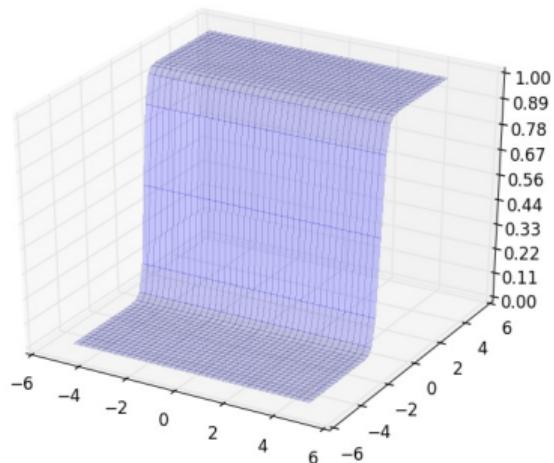
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

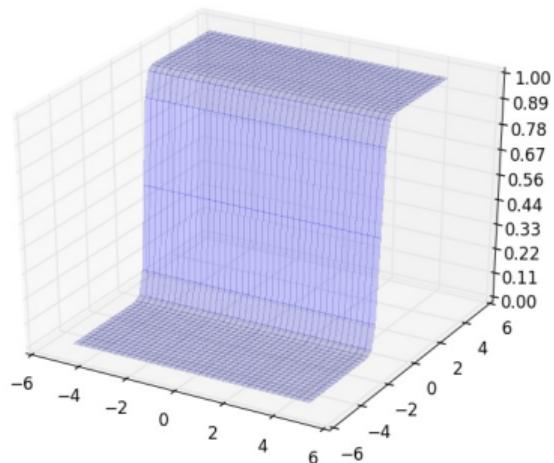
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

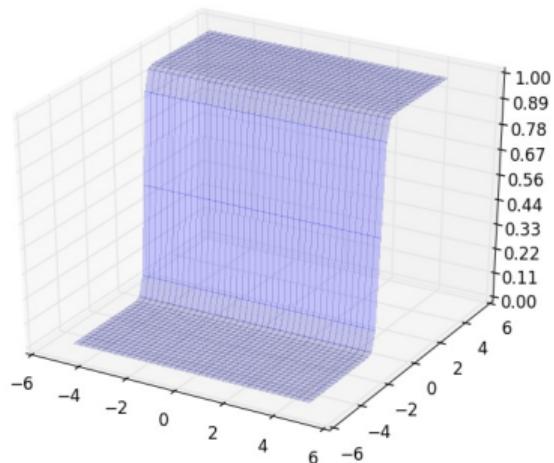
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

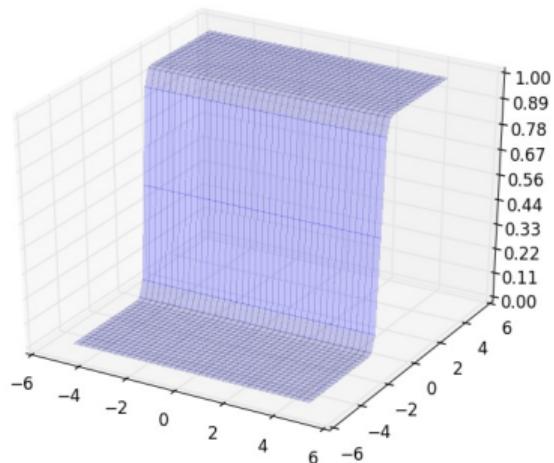
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

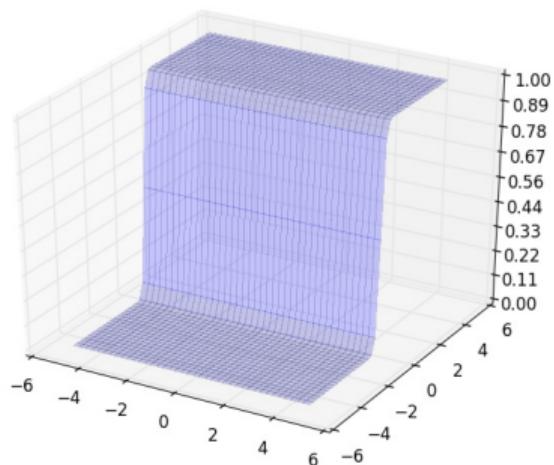
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

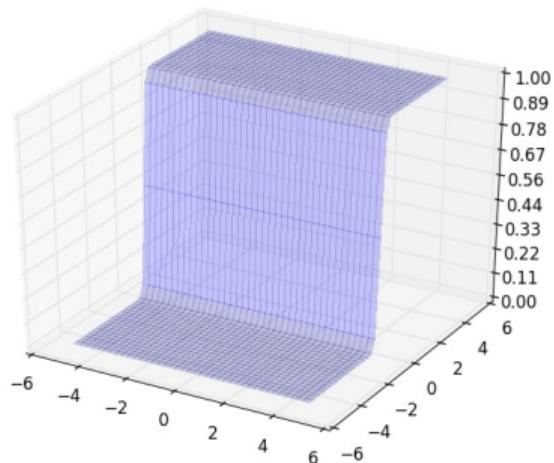
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

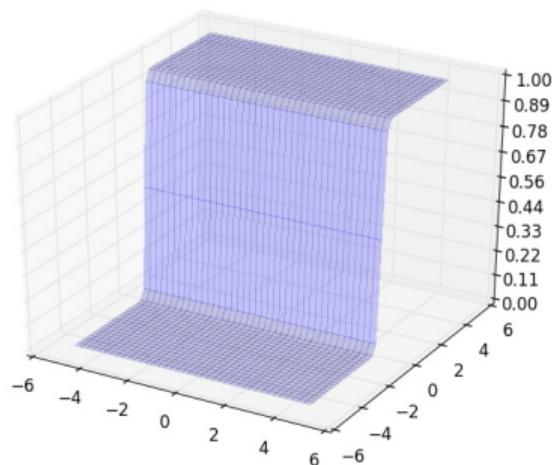
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

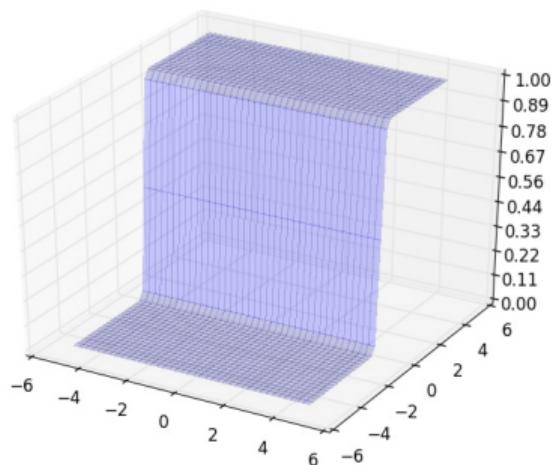
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

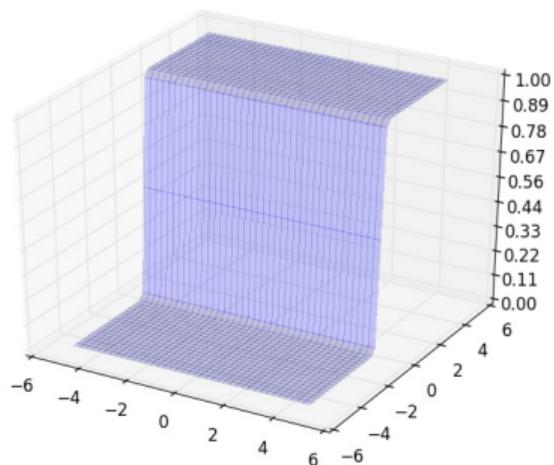
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

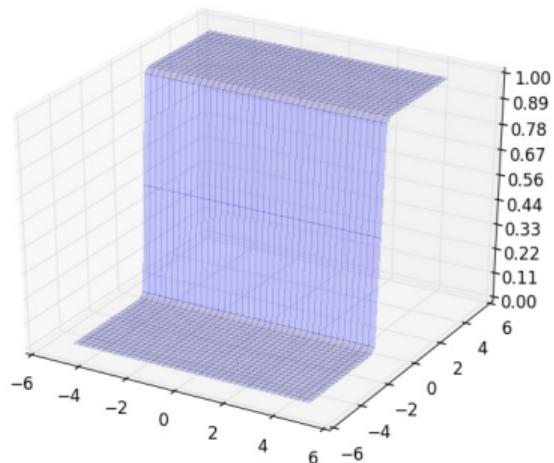
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

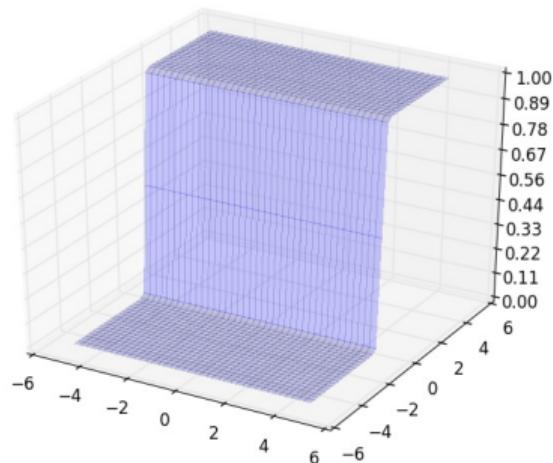
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

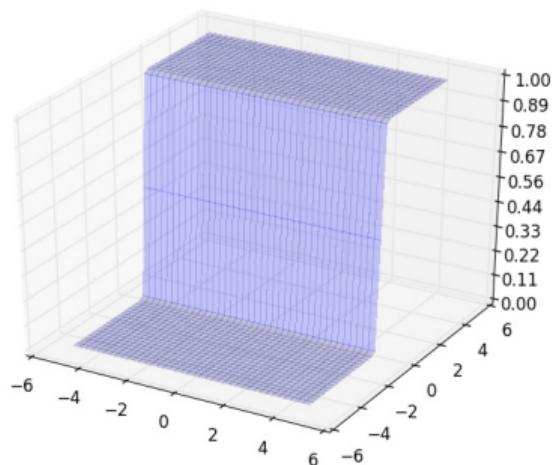
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

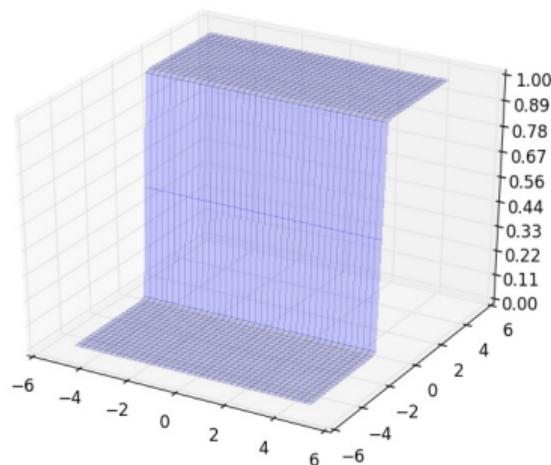
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

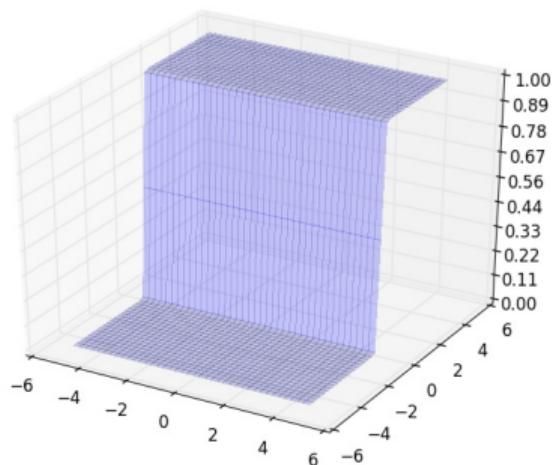
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

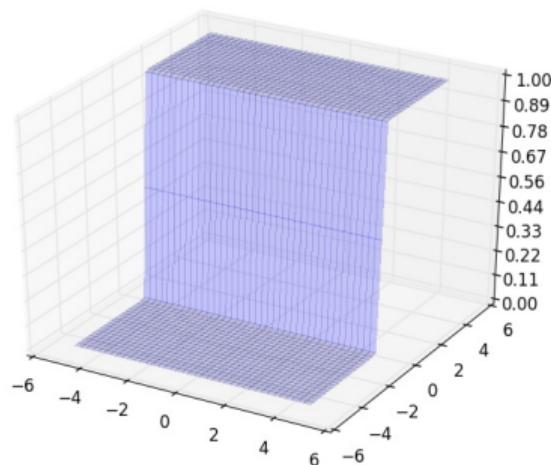
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

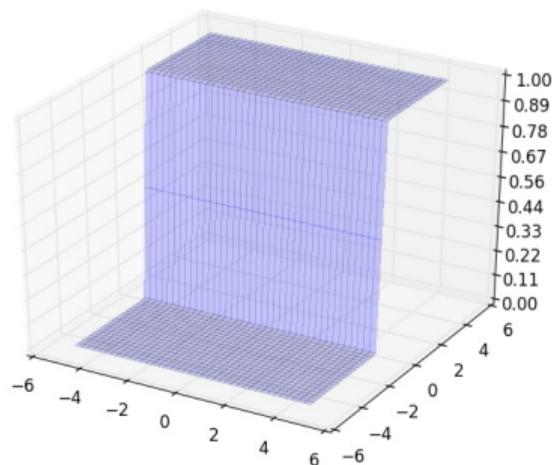
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

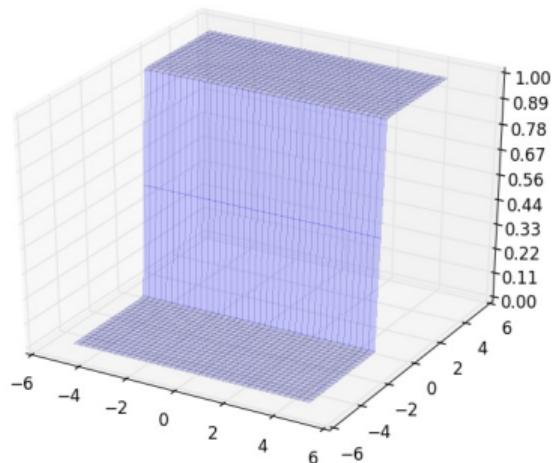
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

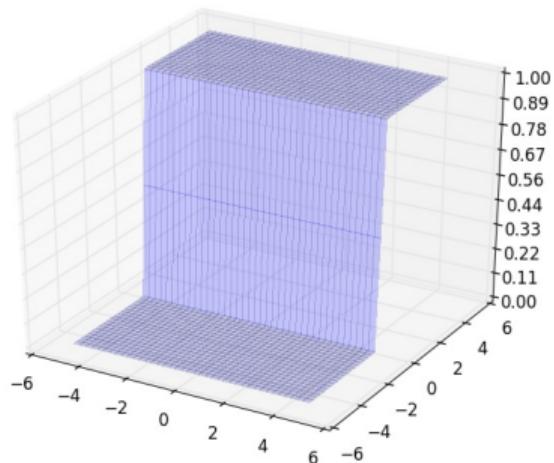
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

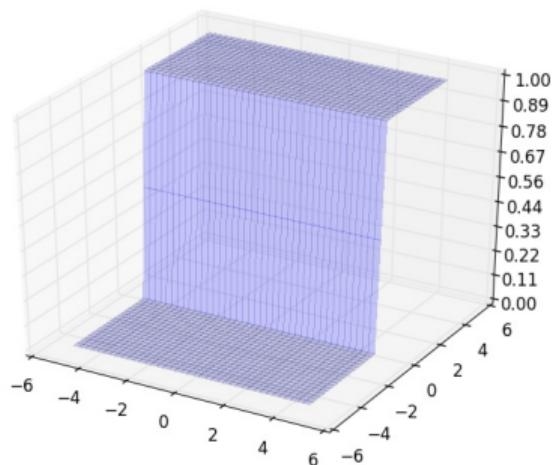
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

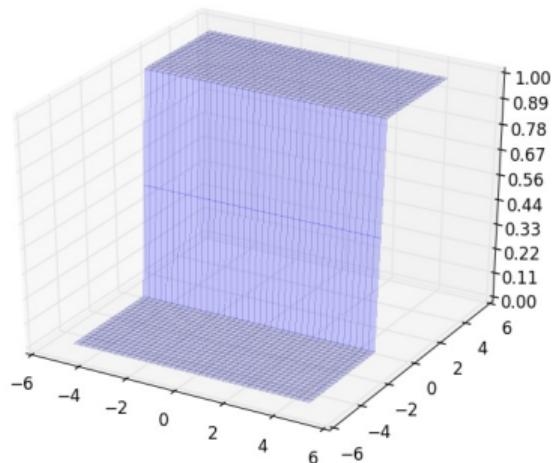
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation



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

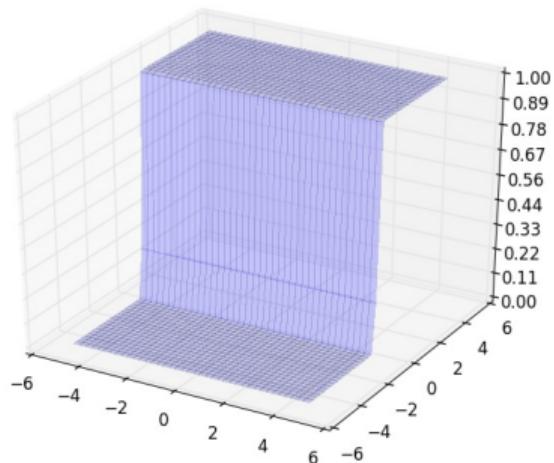
$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

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



$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

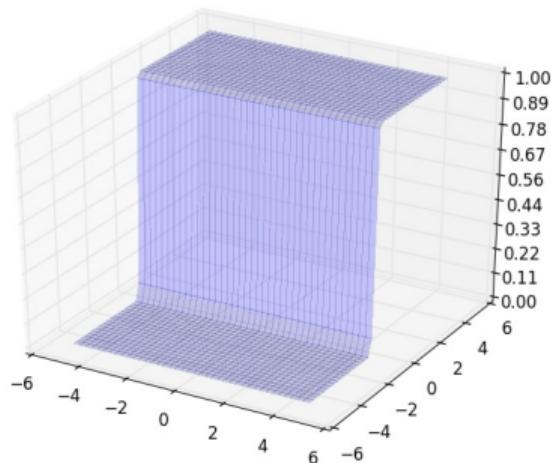
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation
- And now we change  $b$



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

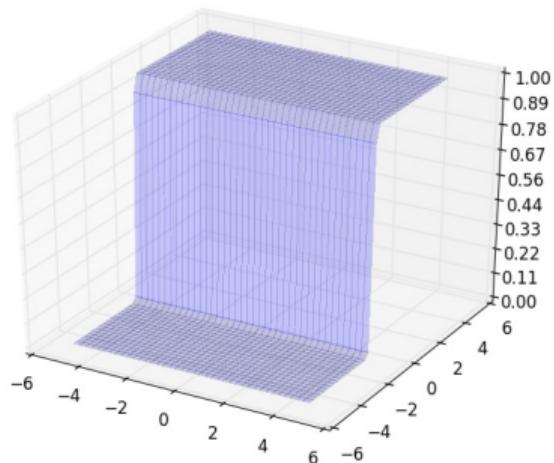
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation
- And now we change  $b$



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

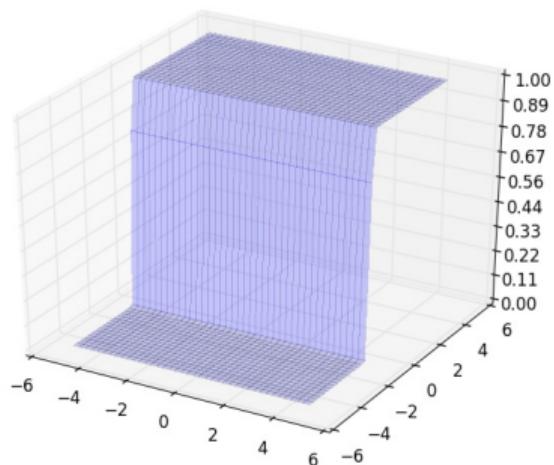
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation
- And now we change  $b$



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

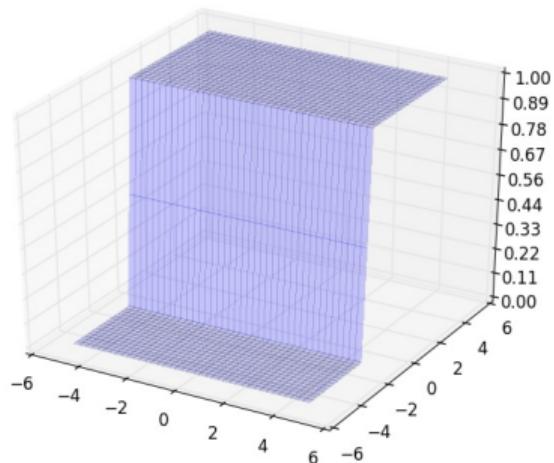
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation
- And now we change  $b$



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

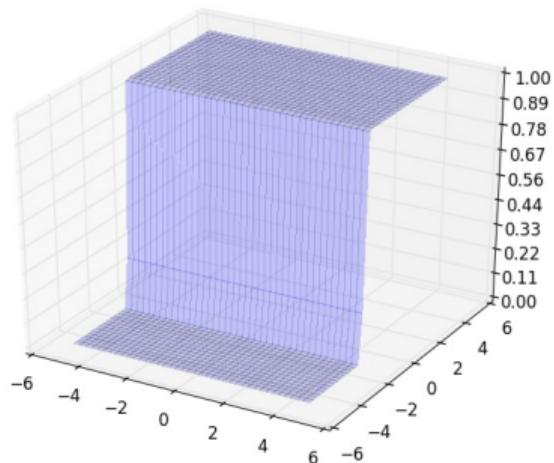
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation
- And now we change  $b$



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

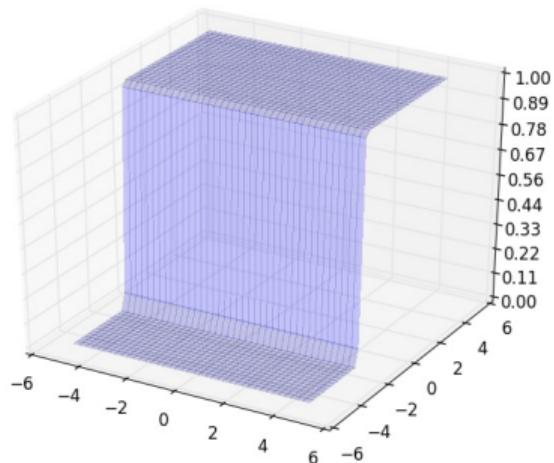
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation
- And now we change  $b$



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

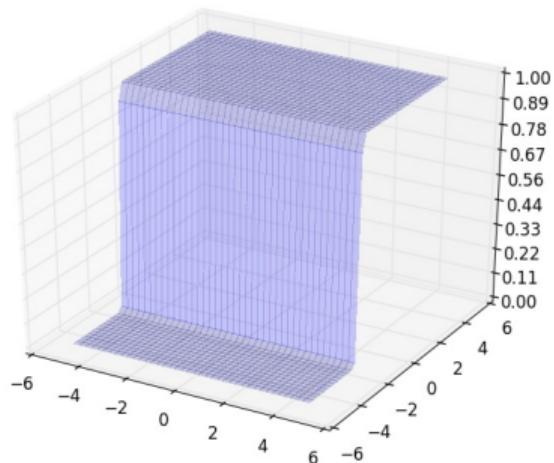
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation
- And now we change  $b$



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

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

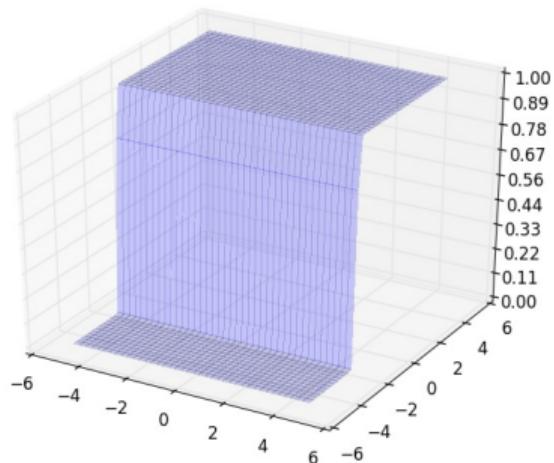
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation
- And now we change  $b$



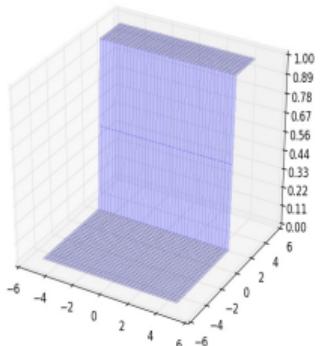
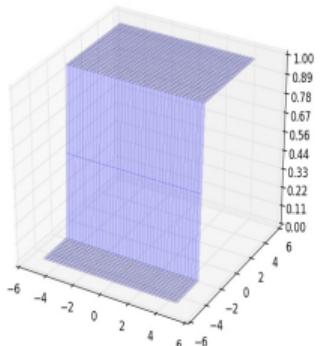
$$w_1 = 0, w_2 = 25, b = 40$$

$$y = \frac{1}{1 + \exp^{-(w_1x_1 + w_2x_2 + b)}}$$

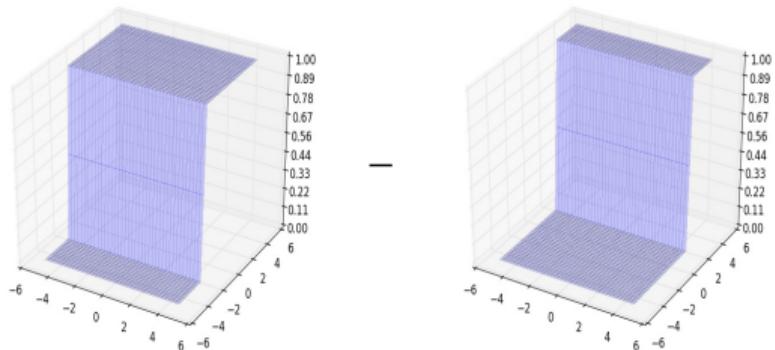
- Now let us set  $w_1$  to 0 and adjust  $w_2$  to get a 2-dimensional step function with a different orientation
- And now we change  $b$



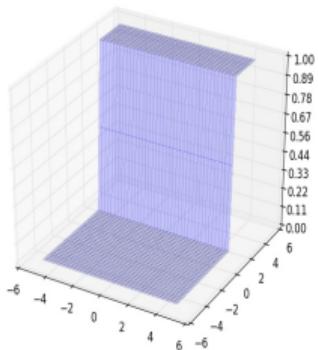
$$w_1 = 0, w_2 = 25, b = 45$$



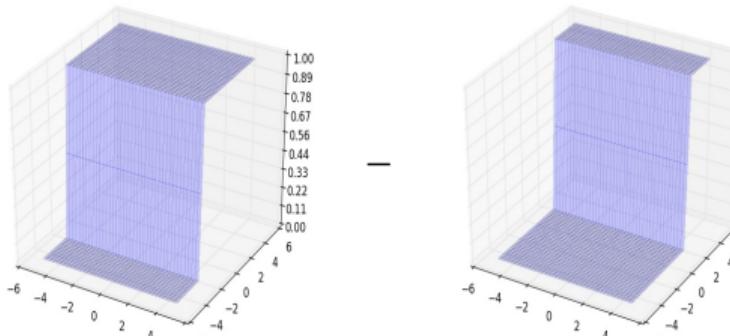
- Again, what if we take two such step functions (with different  $b$  values) and subtract one from the other



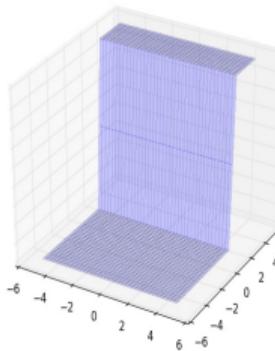
- Again, what if we take two such step functions (with different  $b$  values) and subtract one from the other



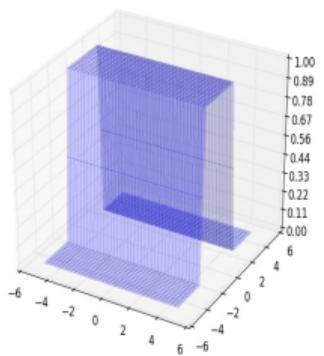
- Again, what if we take two such step functions (with different  $b$  values) and subtract one from the other

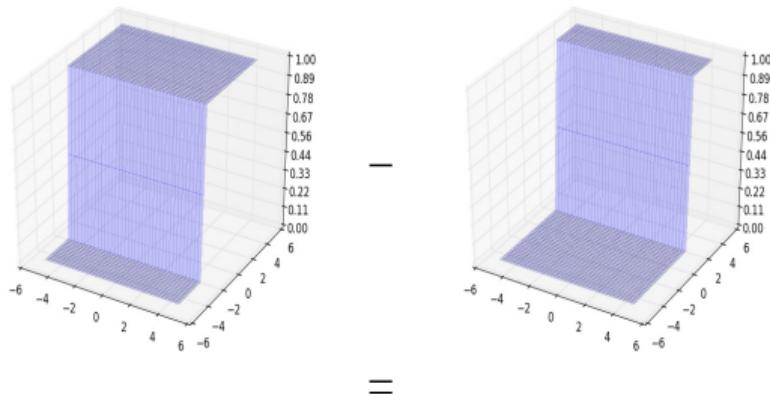


-

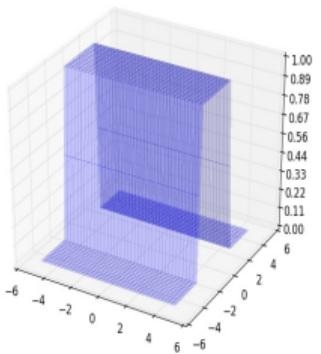


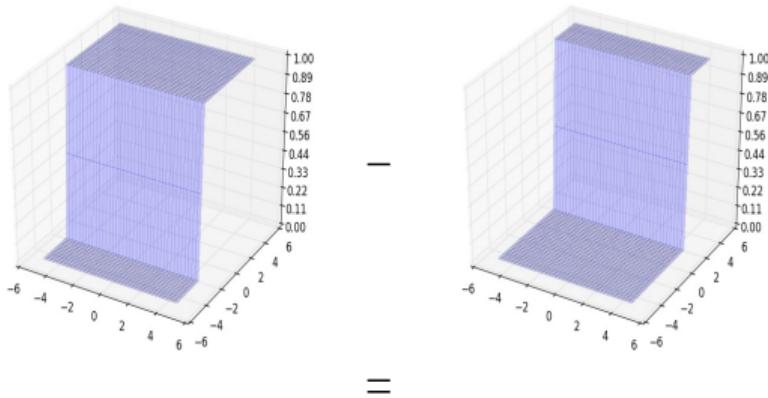
=



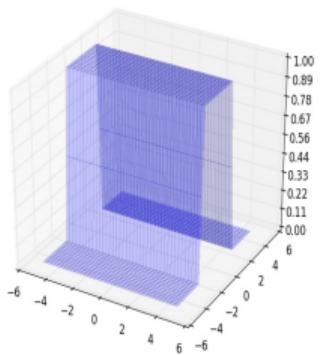


- Again, what if we take two such step functions (with different  $b$  values) and subtract one from the other
- We still don't get a tower (or we get a tower which is open from two sides)

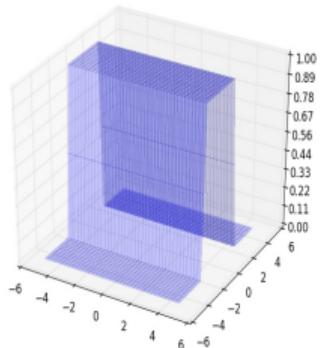
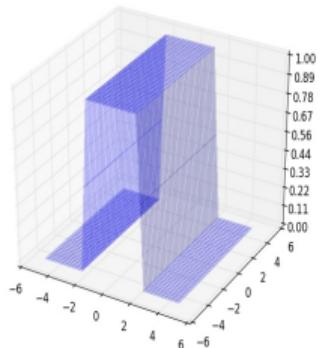


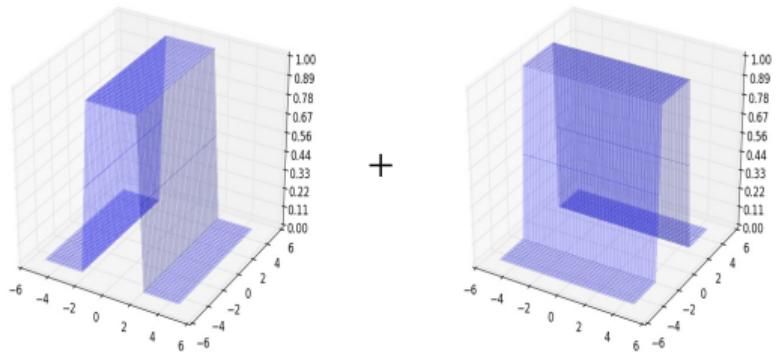


- Again, what if we take two such step functions (with different  $b$  values) and subtract one from the other
- We still don't get a tower (or we get a tower which is open from two sides)
- Notice that this open tower has a different orientation from the previous one



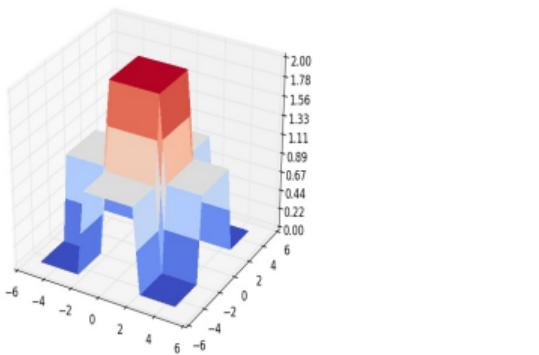
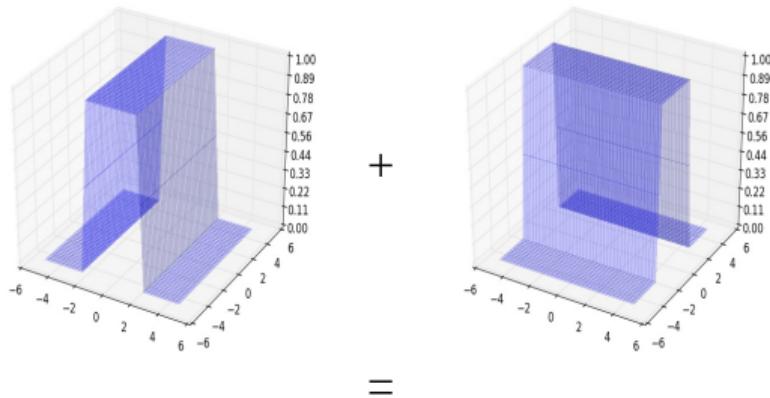
- Now what will we get by adding two such open towers ?

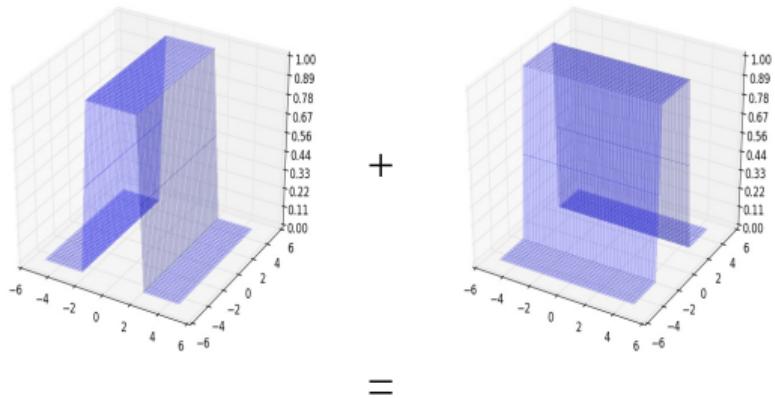




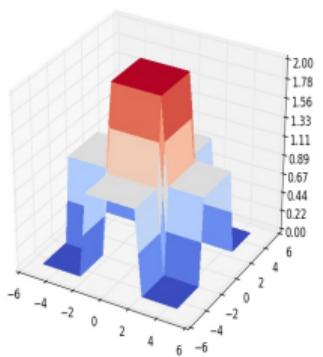
- Now what will we get by adding two such open towers ?

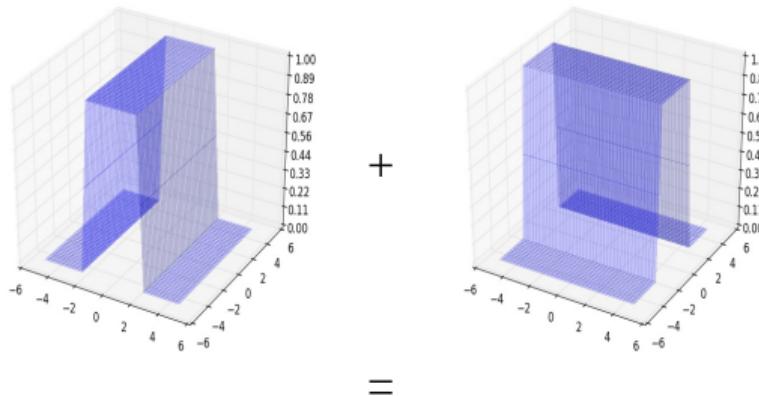
- Now what will we get by adding two such open towers ?



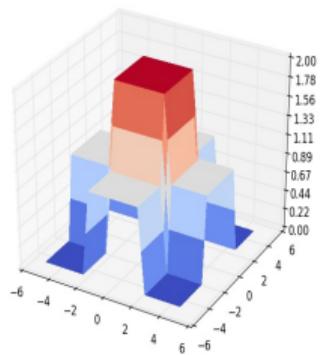


- Now what will we get by adding two such open towers ?
  - We get a tower standing on an elevated base

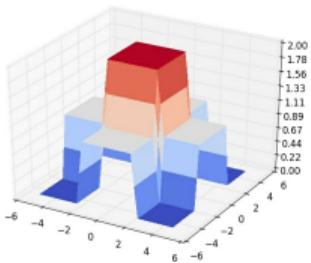
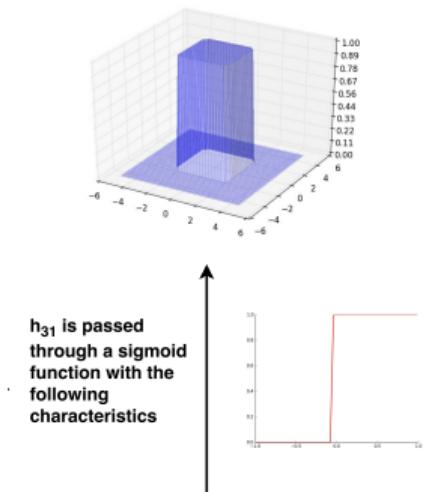




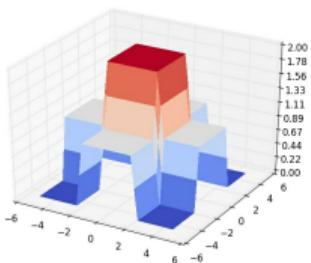
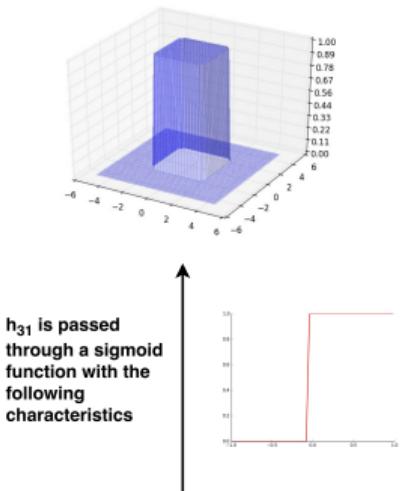
- Now what will we get by adding two such open towers ?
- We get a tower standing on an elevated base
- We can now pass this output through another sigmoid neuron to get the desired tower !



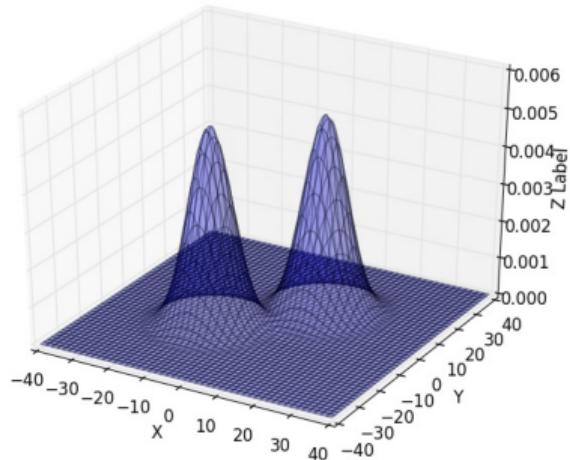
- Now what will we get by adding two such open towers ?
- We get a tower standing on an elevated base
- We can now pass this output through another sigmoid neuron to get the desired tower !



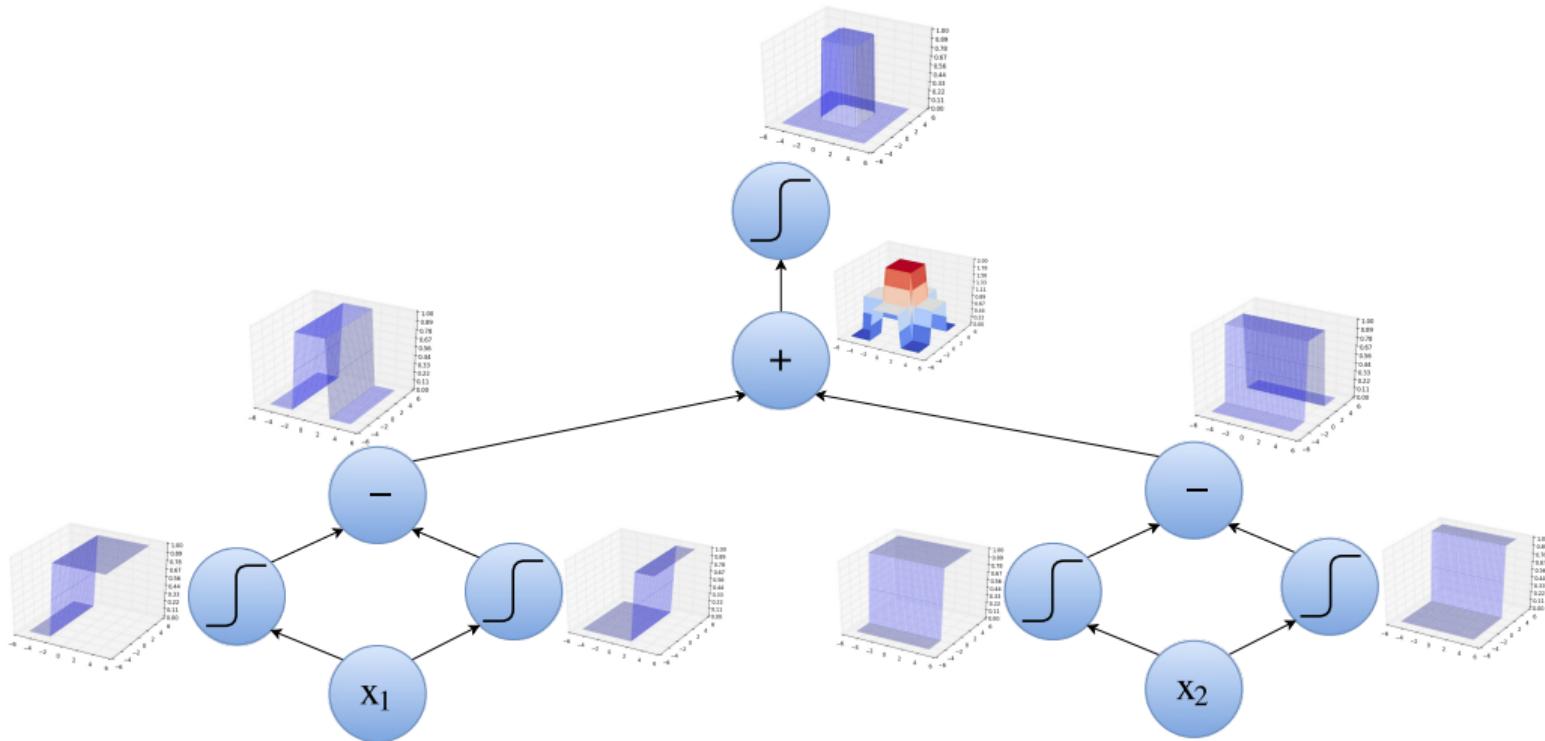
- Now what will we get by adding two such open towers ?
- We get a tower standing on an elevated base
- We can now pass this output through another sigmoid neuron to get the desired tower !
- We can now approximate any function by summing up many such towers

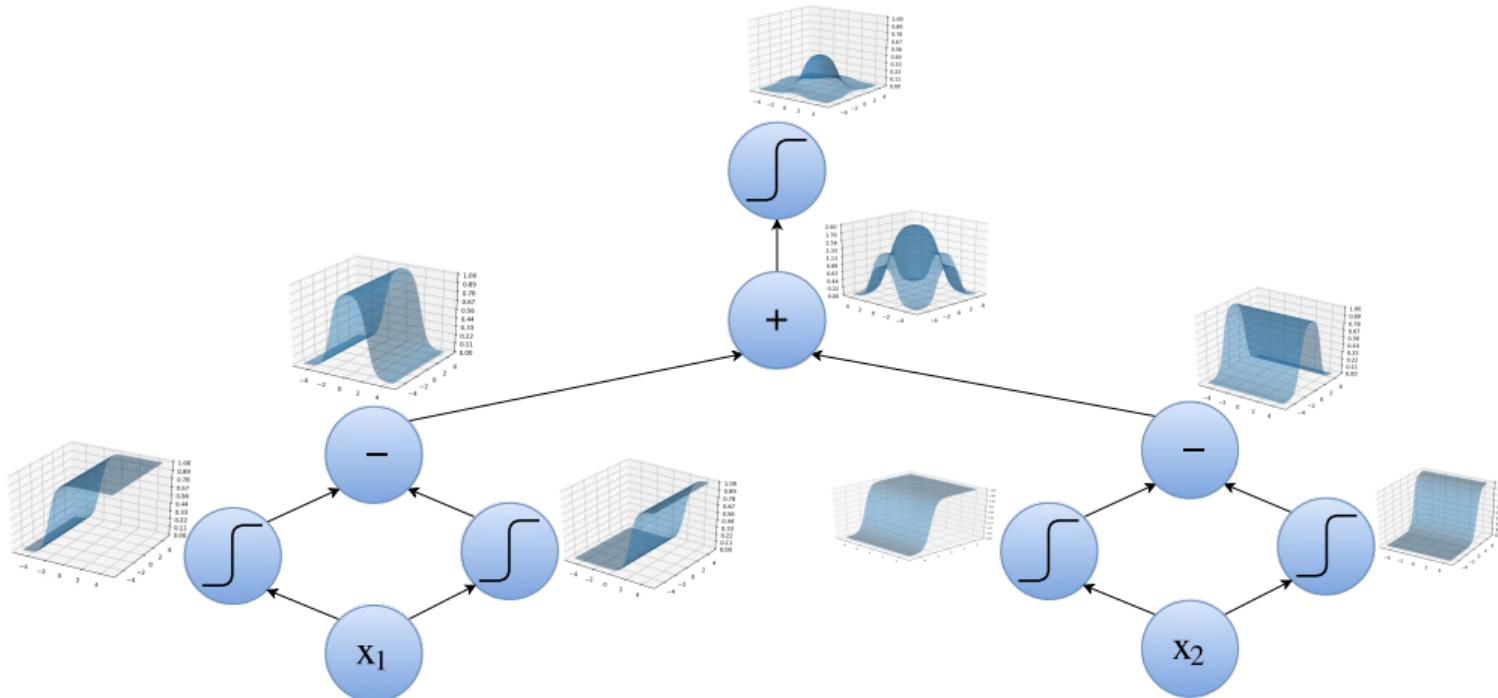


- For example, we could approximate the following function using a sum of several towers



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



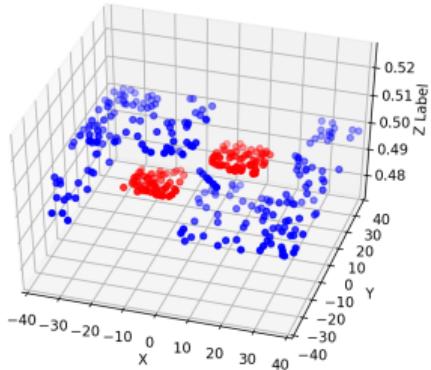


## Think

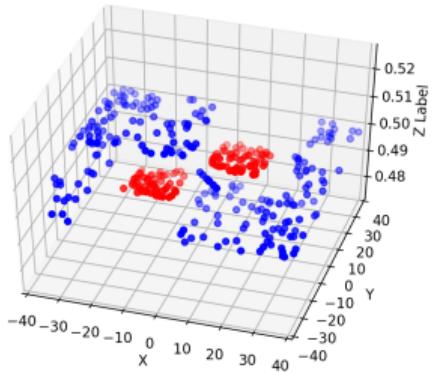
- For 1 dimensional input we needed 2 neurons to construct a tower
- For 2 dimensional input we needed 4 neurons to construct a tower
- How many neurons will you need to construct a tower in  $n$  dimensions ?

## Time to retrospect

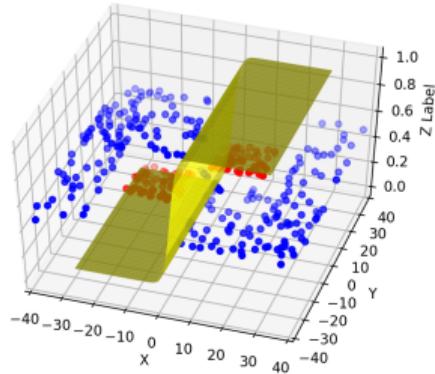
- Why do we care about approximating any arbitrary function ?
- Can we tie all this back to the classification problem that we have been dealing with ?



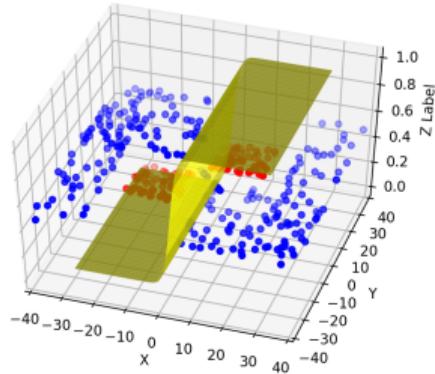
- We are interested in separating the blue points from the red points



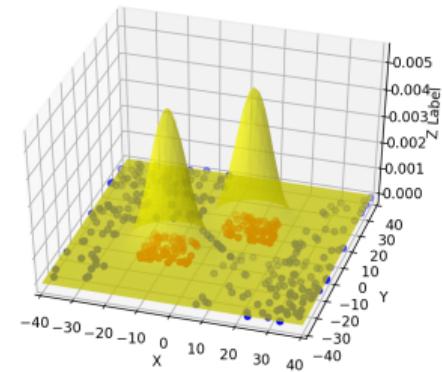
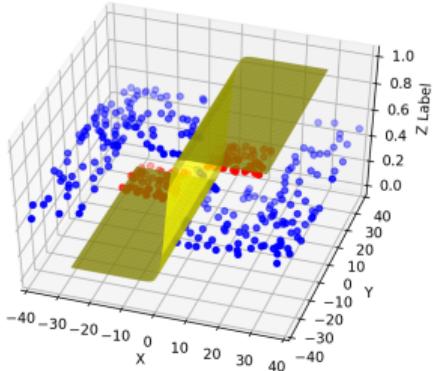
- We are interested in separating the blue points from the red points
- Suppose we use a single sigmoidal neuron to approximate the relation between  $x = [x_1, x_2]$  and  $y$



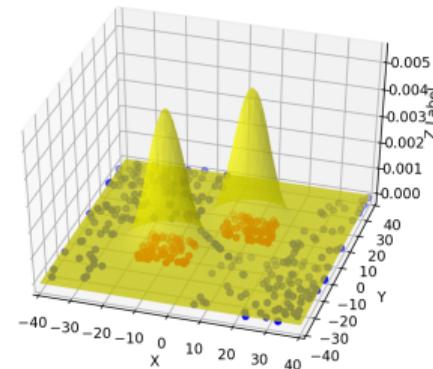
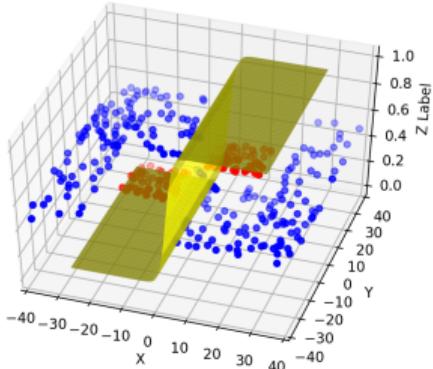
- We are interested in separating the blue points from the red points
- Suppose we use a single sigmoidal neuron to approximate the relation between  $x = [x_1, x_2]$  and  $y$



- 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)

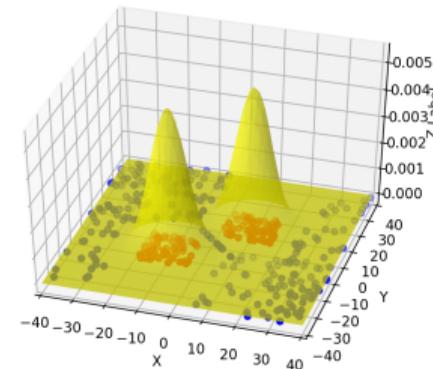
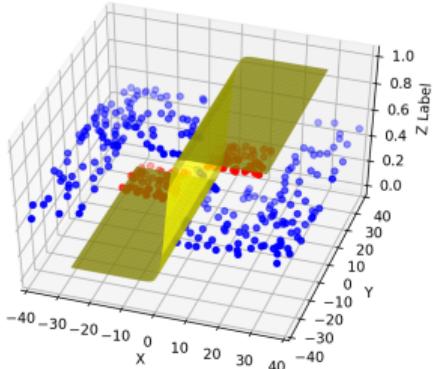


- We are interested in separating the blue points from the red points
- Suppose we use a single sigmoidal neuron to approximate the relation between  $x = [x_1, x_2]$  and  $y$
- Obviously, there will be errors (some blue points get classified as 1 and some red points get classified as 0)
- This is what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