Neural Networks

Ravi Kiran
CVIT, IIIT Hyderabad

15.02.2019

Rate of Change

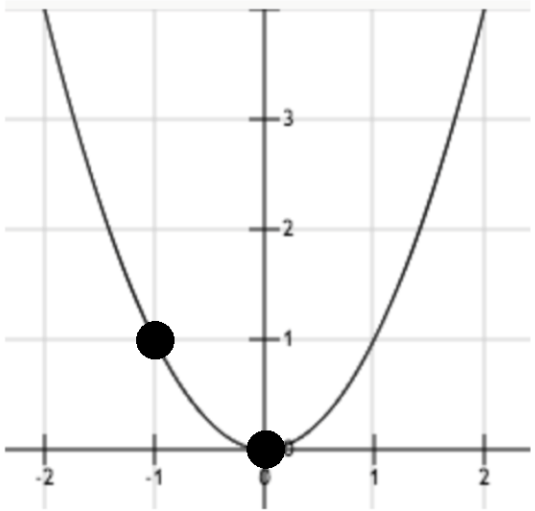
The slope of a line is also called the **rate of change** of the line.

$y = \frac{1}{2}x + 1$

Rate of Change

For nonlinear functions, the “rise over run” formula gives you the average rate of change between two points

$f(x) = x^2$

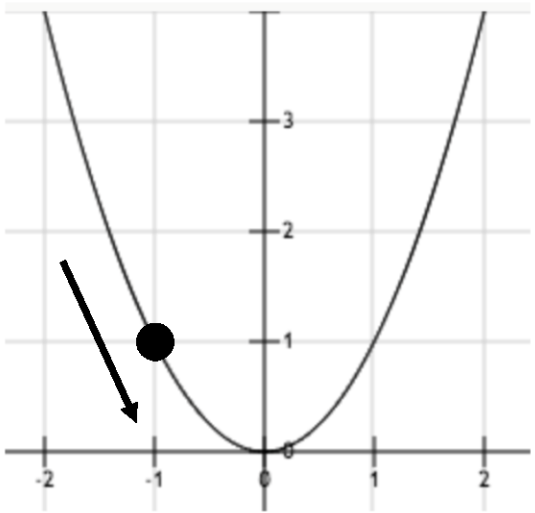


Average slope from x=-1 to x=0 is:
-1

Rate of Change

There is also a concept of rate of change at individual points (rather than two points)

$f(x) = x^2$

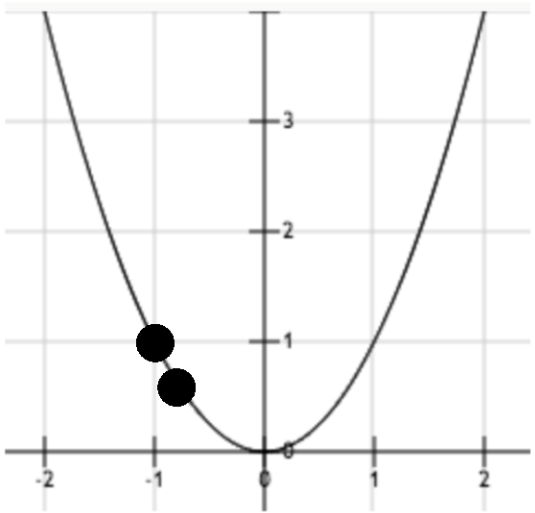


Slope at x=-1 is:
-2

Rate of Change

The slope at a point is called the **derivative** at that point

$f(x) = x^2$



Intuition:
Measure the slope between two points that are really close together

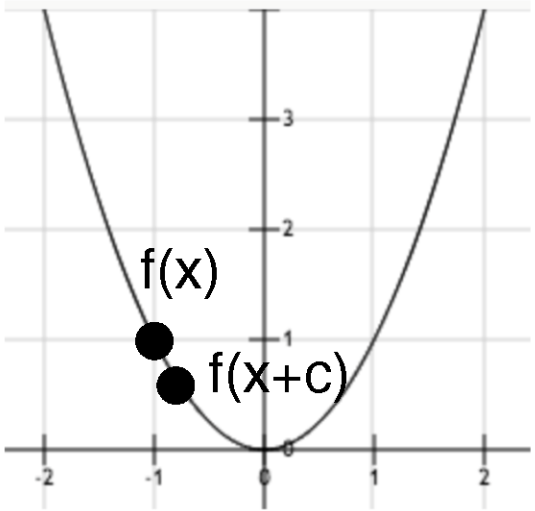
Rate of Change

The slope at a point is called the **derivative** at that point

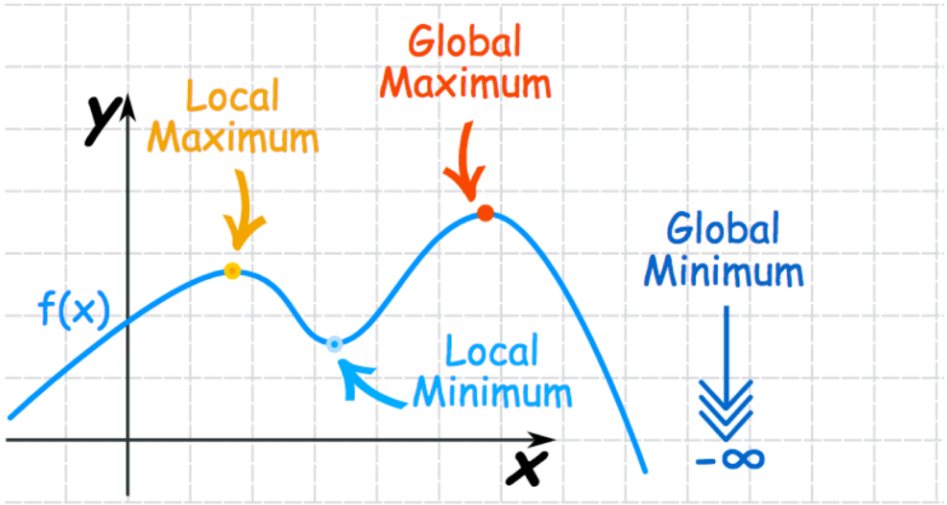
Intuition: Measure the slope between two points that are really close together

$$\frac{f(x + c) - f(x)}{c}$$

Limit as *c* goes to zero



Maxima and Minima



From: https://www.mathsisfun.com/algebra/functions-maxima-minima.html

Derivatives

The derivative of $f(x) = x^2$ is $2x$

Other ways of writing this:

$f'(x) = 2x$

$d/dx\ [x^2] = 2x$

$df/dx = 2x$

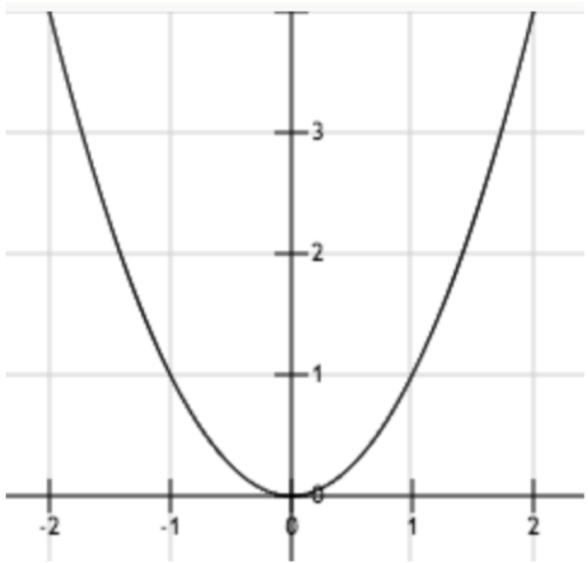
The derivative is also a function! It depends on the value of x.

- The rate of change is different at different points

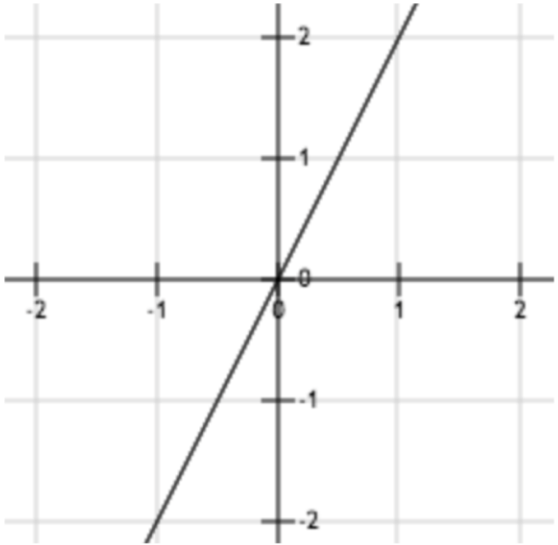
Derivatives

The derivative of $f(x) = x^2$ is $2x$

f(x)



f’(x)



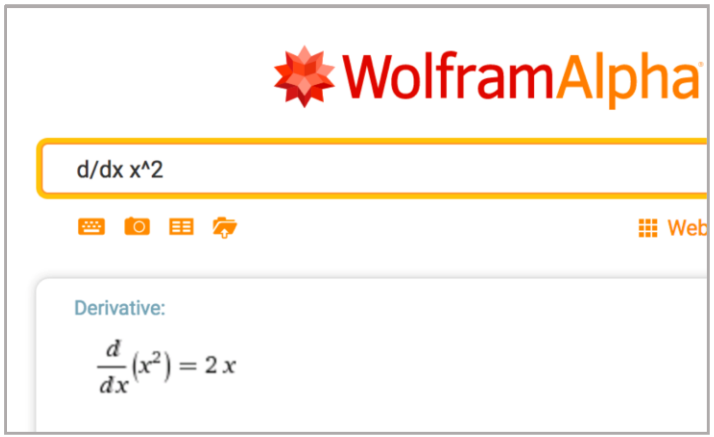
Derivatives

How to calculate a derivative?

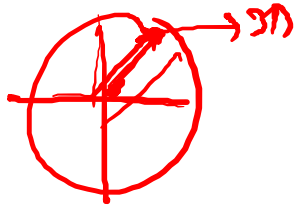
- Not going to do it in this class.

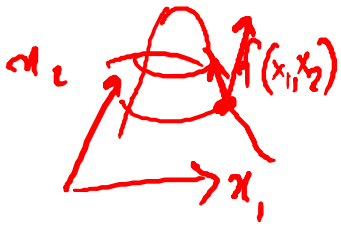
Some software can do it for you.

- Wolfram Alpha



Derivatives





What if a function has multiple arguments?

Ex: $f(x_1, x_2) = 3x_1 + 5x_2$

$df/dx_1 = 3 + 5x_2$ The derivative "with respect to" x_1

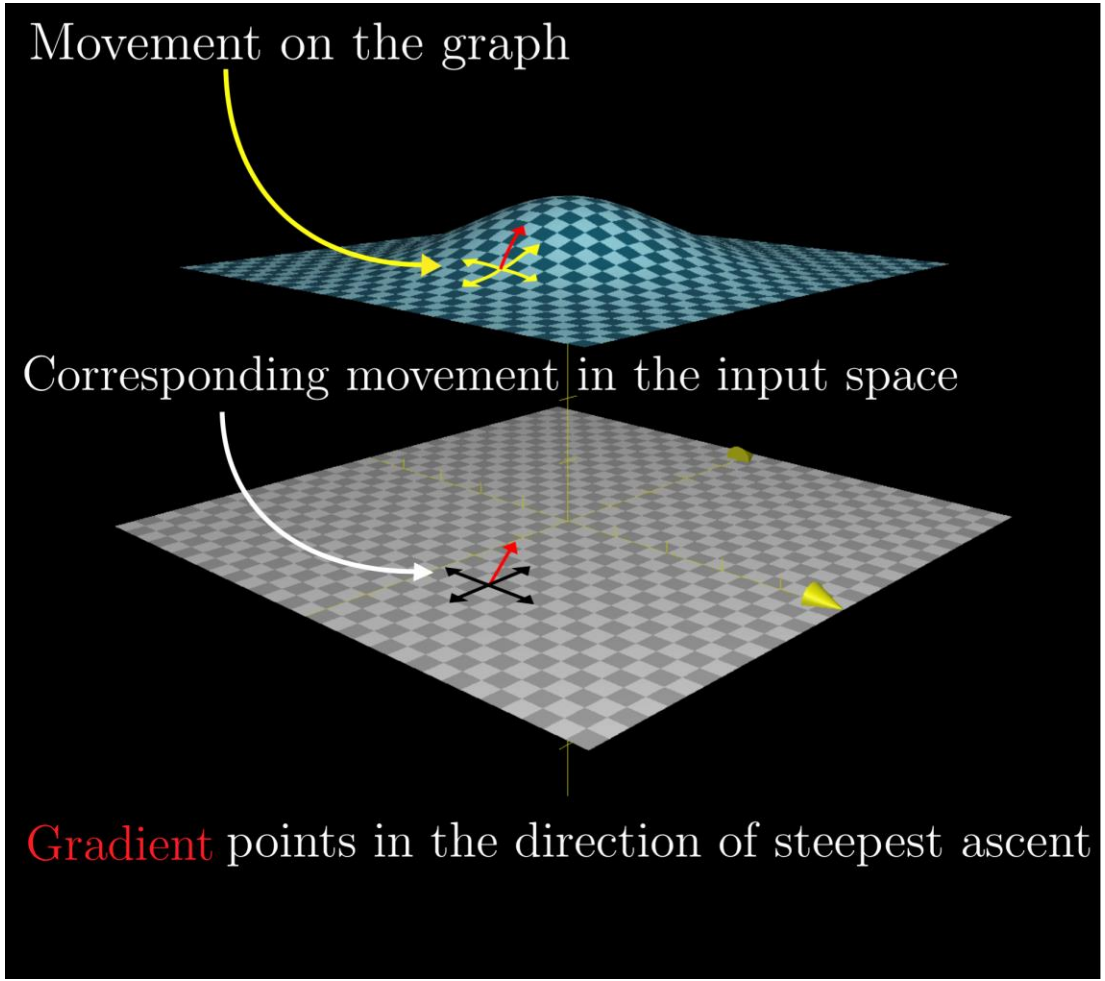
$df/dx_2 = 3x_1 + 5$ The derivative "with respect to" x_2

These two functions are called **partial derivatives**.

The vector of all partial derivatives for a function f is called the **gradient** of the function:

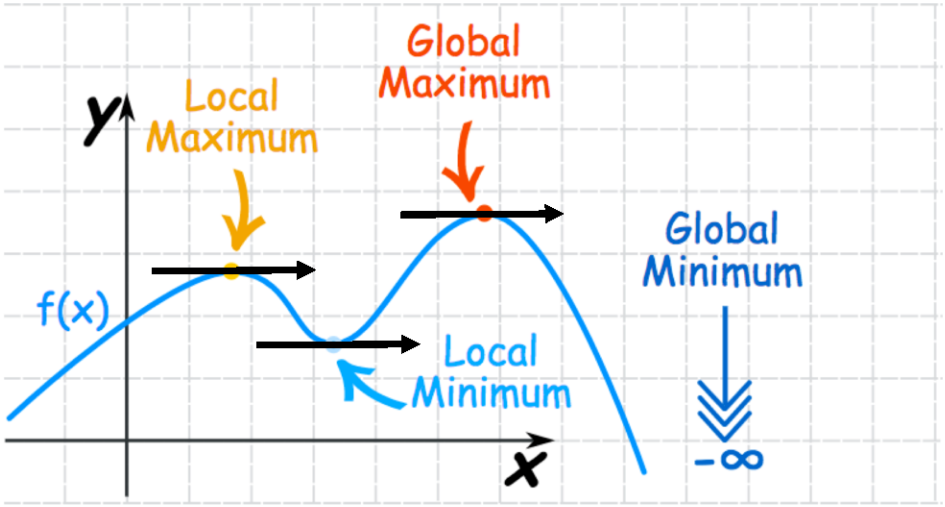
$$\nabla f(x_1, x_2) = < df/dx_1 , df/dx_2 >$$

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \vdots \end{bmatrix}$$



Finding Minima

The derivative is *zero* at any local maximum or minimum.



Finding Minima

The derivative is *zero* at any local maximum or minimum.

One way to find a minimum: set f'(x)=0 and solve for x.

- For most functions, there isn't a way to solve this.
- Instead: algorithmically search different values of x until you find one that results in a gradient near 0.

Finding Minima

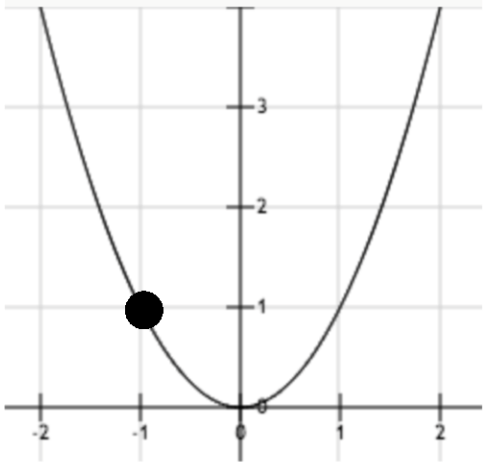
If the derivative is positive, the function is **increasing.**

- Don’t move in that direction, because you’ll be moving away from a trough.

If the derivative is negative, the function is **decreasing.**

- Keep going, since you’re getting closer to a trough

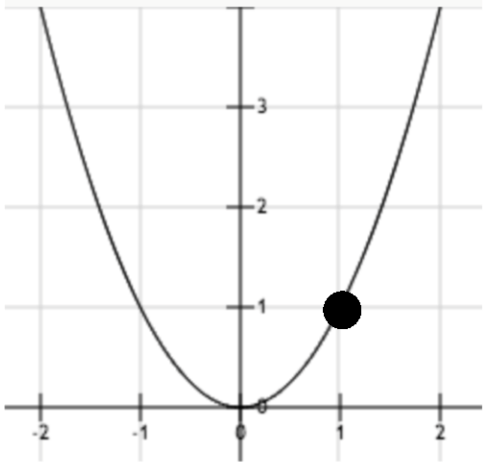
Finding Minima



f’(-1) = -2
At x=-1, the function is decreasing as x gets larger.
This is what we want, so let’s make x larger.
Increase x by the size of the gradient:

-1 + 2 = 1

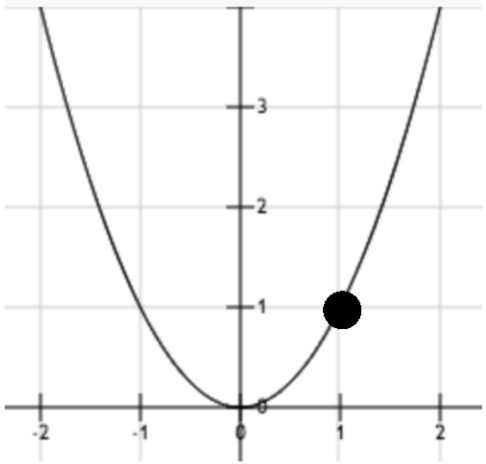
Finding Minima



f’(-1) = -2
At x=-1, the function is decreasing as x gets larger.
This is what we want, so let’s make x larger.
Increase x by the size of the gradient:

-1 + 2 = 1

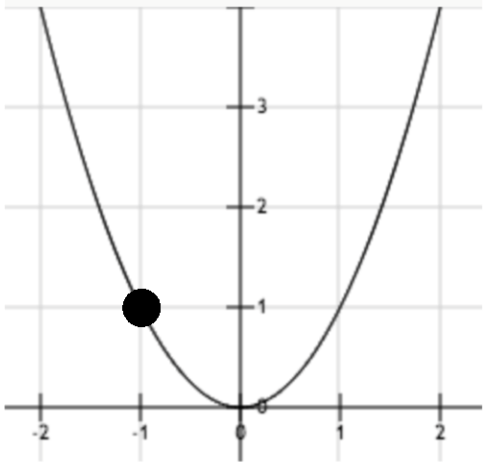
Finding Minima



f’(1) = 2
At x=1, the function is increasing as x gets larger. This is not what we want, so let’s make x smaller.
Decrease x by the size of the gradient:

1 - 2 = -1

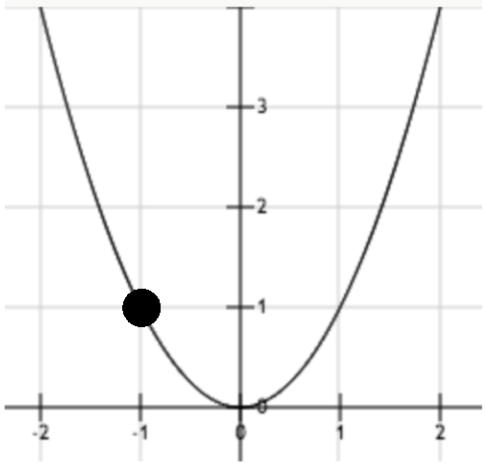
Finding Minima



f’(1) = 2
At x=1, the function is increasing as x gets larger. This is not what we want, so let’s make x smaller. Decrease x by the size of the gradient:

1 - 2 = -1

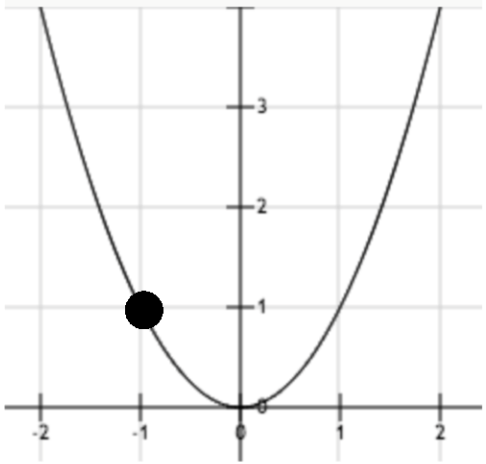
Finding Minima



We will keep jumping between the same two points this way.

We can fix this be using a learning rate or step size.

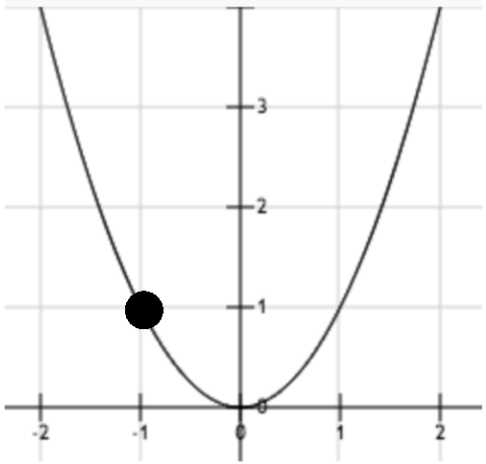
Finding Minima



$f'(-1) = -2$

$x \mathrel{+}= 2\eta =$

Finding Minima

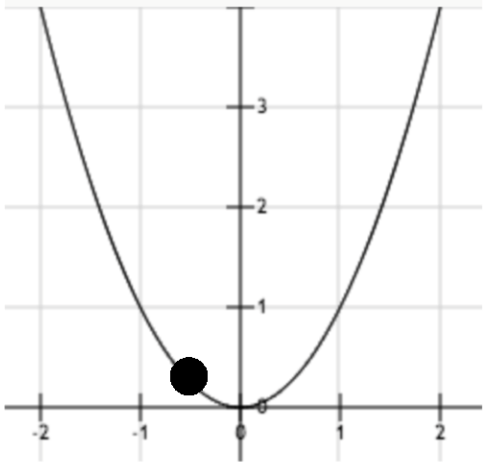


$f'(-1) = -2$

$x \mathrel{+}= 2\eta =$

Let's use $\eta = 0.25$.

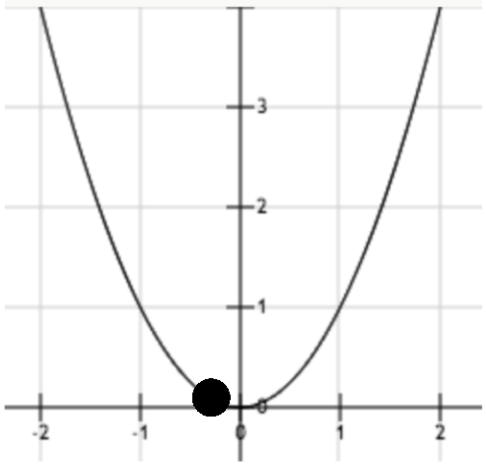
Finding Minima



$f'(-1) = -2$

$x = -1 + 2(.25) = -0.5$

Finding Minima



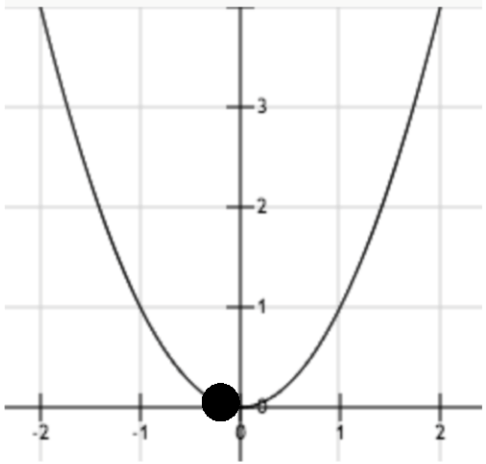
f’(-1) = -2

x = -1 + 2(.25) = -0.5

f’(-0.5) = -1

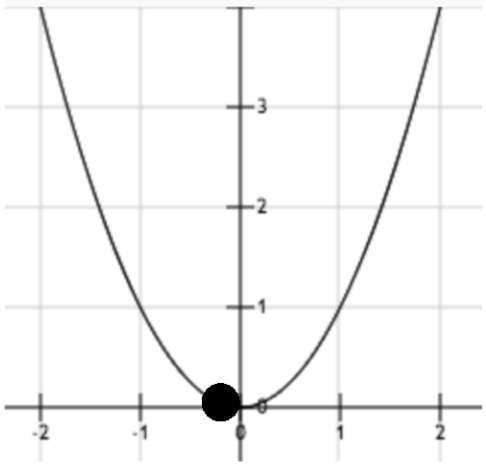
x = -0.5 + 1(.25) = -0.25

Finding Minima



f’(-1) = -2
x = -1 + 2(.25) = -0.5
f’(-0.5) = -1
x = -0.5 + 1(.25) = -0.25
f’(-0.25) = -0.5
x = -0.25 + 0.5(.25) = -0.125

Finding Minima



f’(-1) = -2
x = -1 + 2(.25) = -0.5
f’(-0.5) = -1
x = -0.5 + 1(.25) = -0.25
f’(-0.25) = -0.5
x = -0.25 + 0.5(.25) = -0.125

Eventually we’ll reach x=0.

Gradient Descent

1. Initialize the parameters $\mathbf{w}$ to some guess (usually all zeros, or random values)
2. Update the parameters:
   $\mathbf{w} = \mathbf{w} - \eta \nabla L(\mathbf{w})$
3. Update the learning rate η
4. Repeat steps 2-3 until $\nabla L(\mathbf{w})$ is close to zero.

Gradient Descent

Gradient descent is guaranteed to eventually find a *local* minimum if:

- the learning rate is decreased appropriately;
- a finite local minimum exists (i.e., the function doesn't keep decreasing forever).

Gradient Ascent

What if we want to find a local *maximum*?

Same idea, but the update rule moves the parameters in the opposite direction:

$$\mathbf{w} = \mathbf{w} + \eta \, \nabla L(\mathbf{w})$$

Learning Rate

In order to guarantee that the algorithm will converge, the learning rate should decrease over time. Here is a general formula.

At iteration t:

$\eta_t = c_1 / (t^a + c_2)$,

where $0.5 < a < 2$

$c1 > 0$

$c2 \geq 0$

Stopping Criteria

For most functions, you probably won't get the gradient to be exactly equal to **0** in a reasonable amount of time.

Once the gradient is sufficiently close to **0**, stop trying to minimize further.

How do we measure how close a gradient is to **0**?

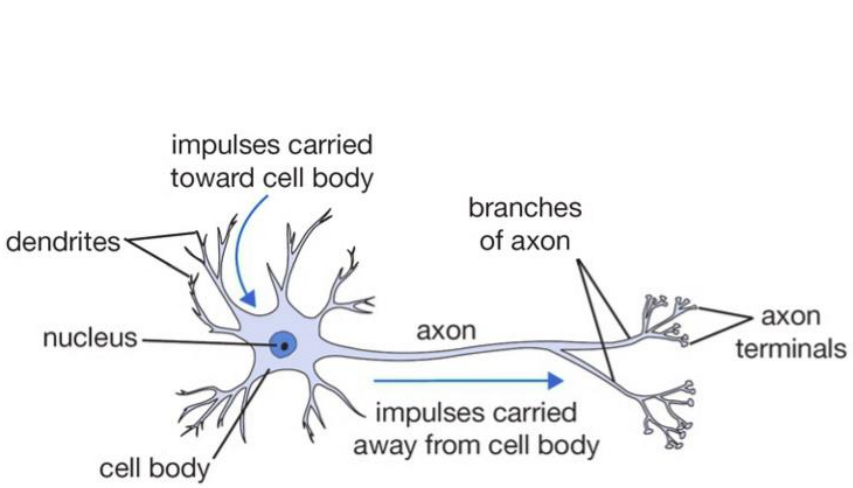
Stopping Criteria

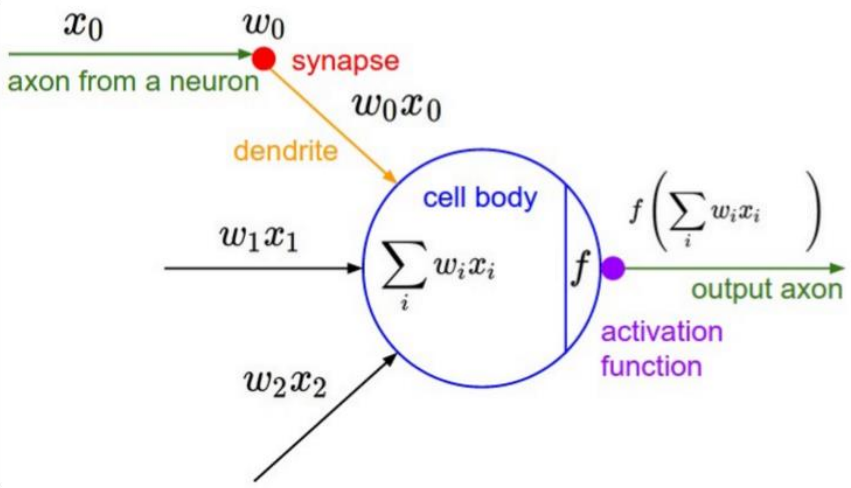
Stop when the norm of the gradient is below some threshold, θ:

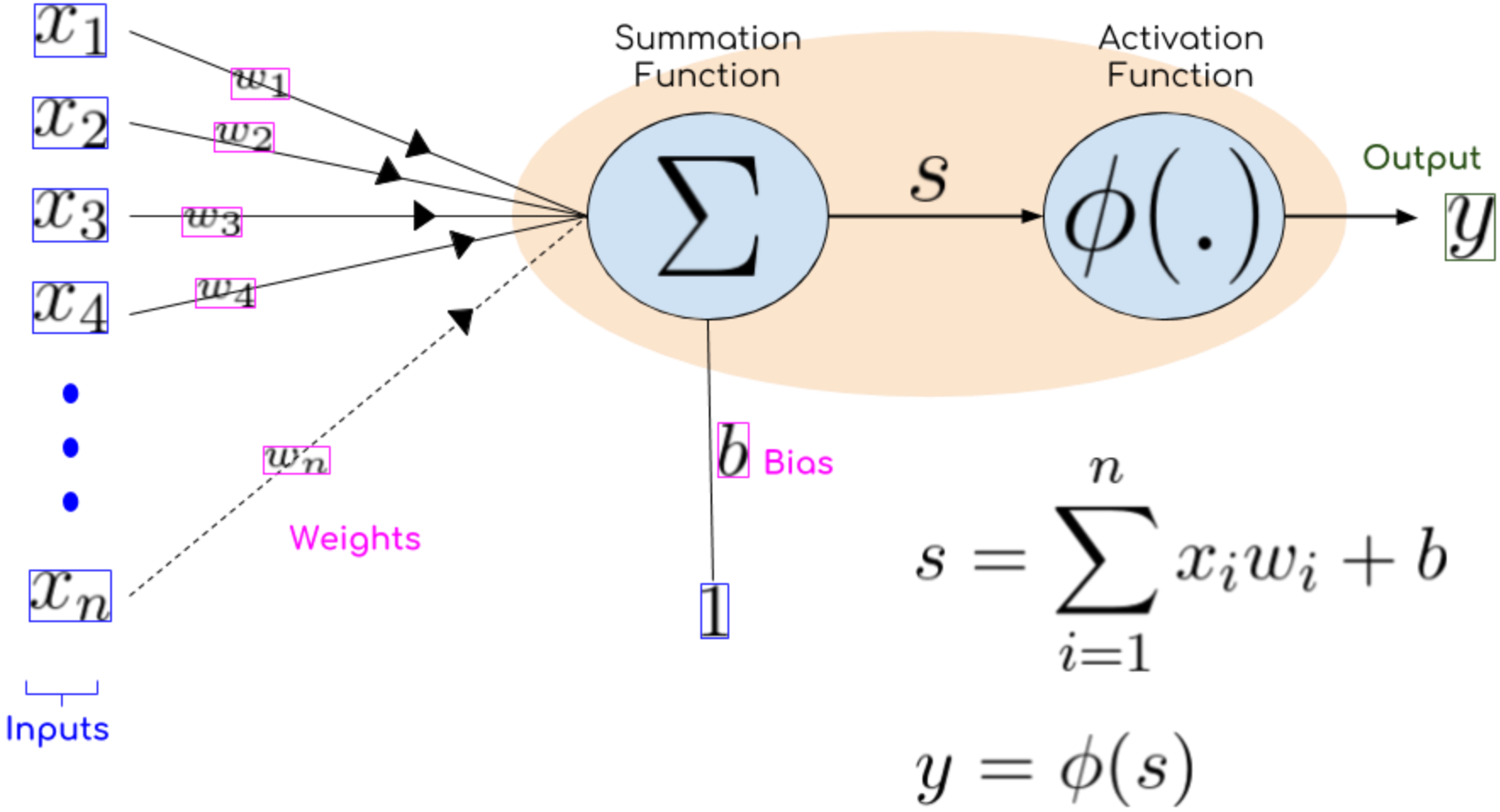
$$||\nabla L(\mathbf{w})|| < \theta$$

Common values of θ are around .01, but if it is taking too long, you can make the threshold larger.

Basic Building block :: A Neuron



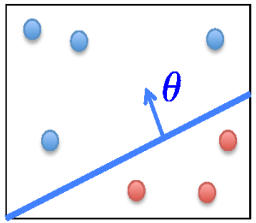



x_1
x_2
x_3
x_4
x_n
w_1
w_2
w_3
w_4
w_n
Weights
Inputs
Summation Function
$\sum$
s
Activation Function
$\phi(.)$
Output
y
b Bias
1
$s = \sum_{i=1}^{n} x_i w_i + b$
$y = \phi(s)$

Linear Classifiers

- **Linear classifiers**: represent decision boundary by hyperplane

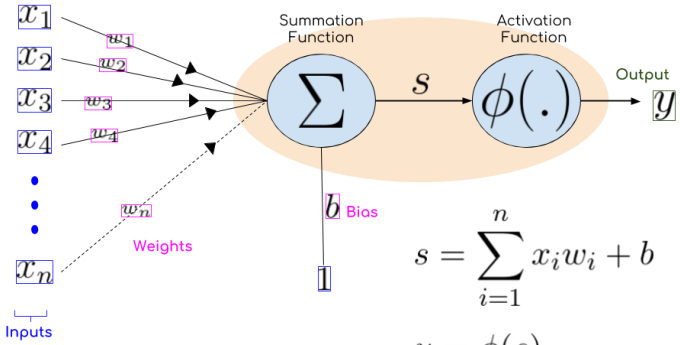
$$\boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{bmatrix} \quad \boldsymbol{x}^\mathsf{T} = \begin{bmatrix} 1 & x_1 & \dots & x_d \end{bmatrix}$$



$$h(\boldsymbol{x}) = \text{sign}(\boldsymbol{\theta}^\mathsf{T}\boldsymbol{x}) \quad \text{where} \quad \text{sign}(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases}$$

– Note that: $\boldsymbol{\theta}^\mathsf{T}\boldsymbol{x} > 0 \implies y = +1$

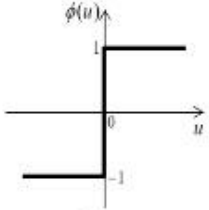
$\boldsymbol{\theta}^\mathsf{T}\boldsymbol{x} < 0 \implies y = -1$



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

$$y = \phi(s)$$

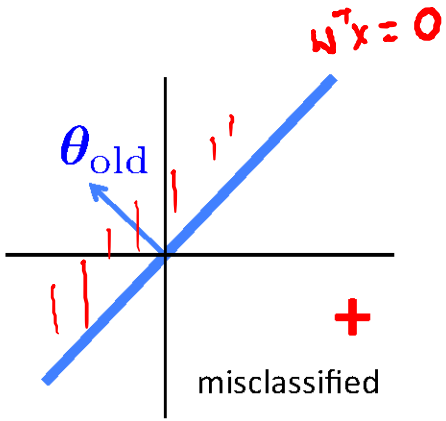
sign function

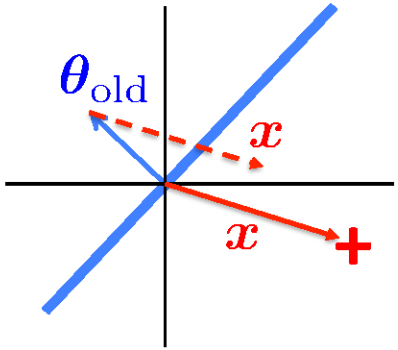


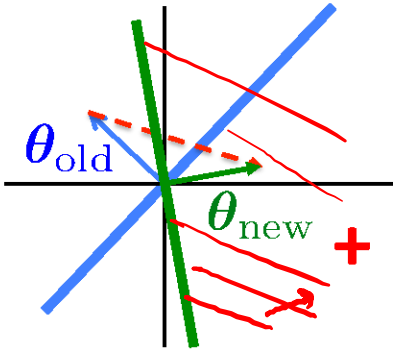
$$\phi_{sign}(u) = \begin{cases} 1 & \text{if } u \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

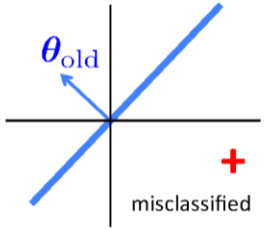
Perceptron Rule: If $\boldsymbol{x}^{(i)}$ is misclassified, do $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + y^{(i)}\boldsymbol{x}^{(i)}$

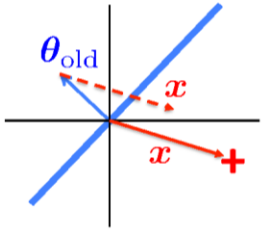
Perceptron Rule: If $\boldsymbol{x}^{(i)}$ is misclassified, do $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + y^{(i)}\boldsymbol{x}^{(i)}$

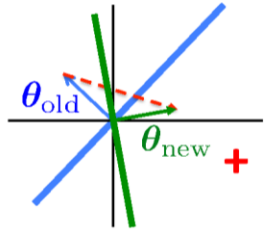












Why the Perceptron Update Works

- Consider the misclassified example (y = +1)
  - Perceptron wrongly thinks that $\boldsymbol{\theta}_{\text{old}}^{\mathsf{T}}\boldsymbol{x} < 0$
- Update:

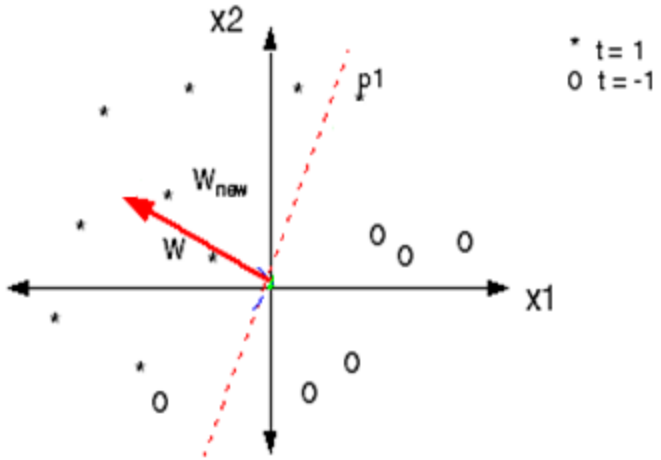
$$\boldsymbol{\theta}_{\text{new}} = \boldsymbol{\theta}_{\text{old}} + y\boldsymbol{x} = \boldsymbol{\theta}_{\text{old}} + \boldsymbol{x} \qquad (\text{since } y = +1)$$

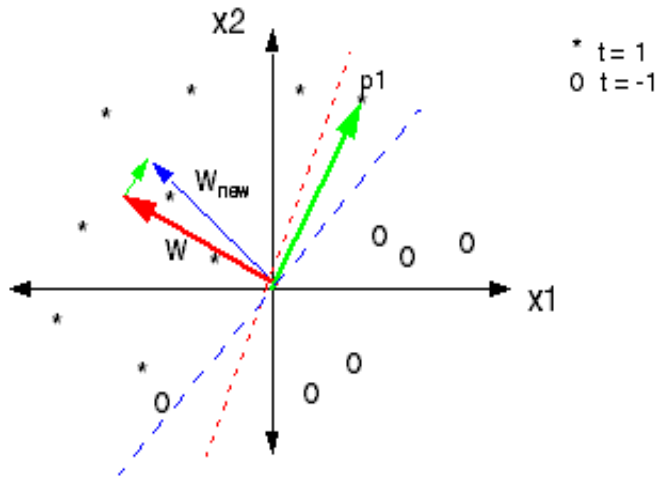
- Note that

$$\begin{aligned}\boldsymbol{\theta}_{\text{new}}^{\mathsf{T}}\boldsymbol{x} &= (\boldsymbol{\theta}_{\text{old}} + \boldsymbol{x})^{\mathsf{T}}\boldsymbol{x} \\ &= \boldsymbol{\theta}_{\text{old}}^{\mathsf{T}}\boldsymbol{x} + \boldsymbol{x}^{\mathsf{T}}\boldsymbol{x}\end{aligned}$$

$\boldsymbol{x}^{\mathsf{T}}\boldsymbol{x} = \|\boldsymbol{x}\|_2^2 > 0$

- Therefore, $\boldsymbol{\theta}_{\text{new}}^{\mathsf{T}}\boldsymbol{x}$ is less negative than $\boldsymbol{\theta}_{\text{old}}^{\mathsf{T}}\boldsymbol{x}$
  - So, we are making ourselves more correct on this example!


x2
x1
p1
W
W_new
* t = 1
0 t = -1


x2
x1
p1
W
W_new
* t = 1
0 t = -1

Perceptron Rule: If $\boldsymbol{x}^{(i)}$ is misclassified, do $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + y^{(i)}\boldsymbol{x}^{(i)}$

The Perceptron Cost Function

- Prediction is correct if $y^{(i)}\boldsymbol{x}^{(i)}\boldsymbol{\theta} > 0$

Perceptron Rule: If $\boldsymbol{x}^{(i)}$ is misclassified, do $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + y^{(i)}\boldsymbol{x}^{(i)}$

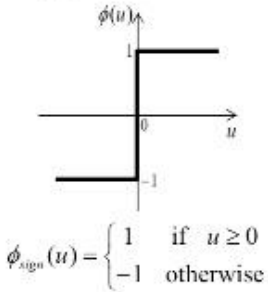
The Perceptron Cost Function

- Prediction is correct if $y^{(i)}\boldsymbol{x}^{(i)}\boldsymbol{\theta} > 0$
- 0/1 loss

$$J_{0/1}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \ell(sign(x^{(i)}\boldsymbol{\theta}), y^{(i)})$$

where $\ell()$ is 0 if the prediction is correct, 1 otherwise

sign function



Based on slide by Alan Fern

Perceptron Rule: If $\boldsymbol{x}^{(i)}$ is misclassified, do $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + y^{(i)}\boldsymbol{x}^{(i)}$

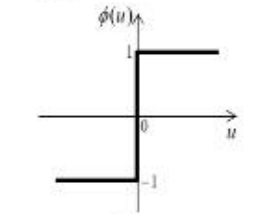
The Perceptron Cost Function

- Prediction is correct if $y^{(i)}\boldsymbol{x}^{(i)}\boldsymbol{\theta} > 0$
- 0/1 loss

$$J_{0/1}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \ell(sign(x^{(i)}\boldsymbol{\theta}), y^{(i)})$$

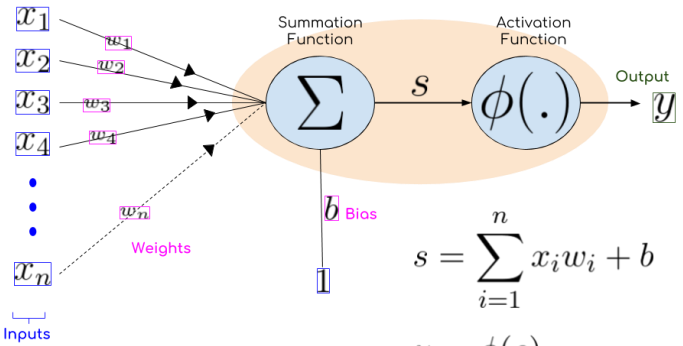
where $\ell()$ is 0 if the prediction is correct, 1 otherwise

sign function



$$\phi_{sign}(u) = \begin{cases} 1 & \text{if } u \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

$J(\theta)$ $\frac{\partial J(\theta)}{\partial \theta}$ $(f+g)' = f' + g'$ $\frac{\partial \ell(sign(x^{(i)}\theta, y^{(i)}))}{\partial \theta}$



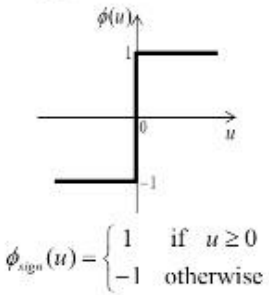
$$s = \sum_{i=1}^{n} x_i w_i + b$$

$$y = \phi(s)$$

Perceptron Rule: If $\boldsymbol{x}^{(i)}$ is misclassified, do $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + y^{(i)}\boldsymbol{x}^{(i)}$

The Perceptron Cost Function

sign function

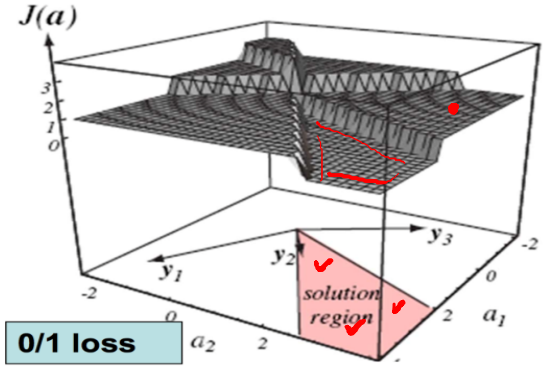


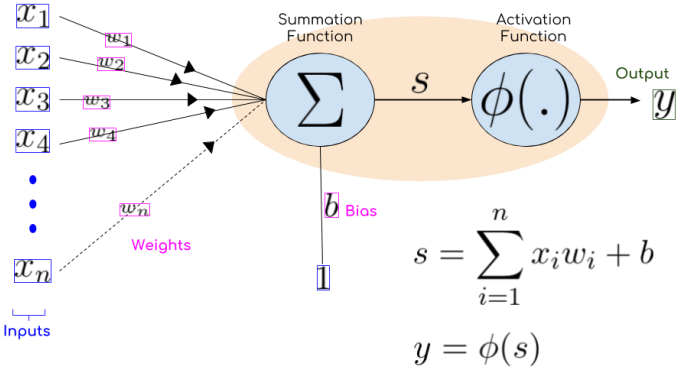
- Prediction is correct if $y^{(i)}\boldsymbol{x}^{(i)}\boldsymbol{\theta} > 0$
- 0/1 loss

$$J_{0/1}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \ell(sign(x^{(i)}\boldsymbol{\theta}), y^{(i)})$$

where $\ell()$ is 0 if the prediction is correct, 1 otherwise

$a = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$



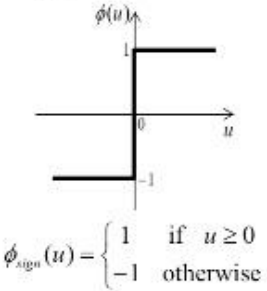


Based on slide by Alan Fern

Perceptron Rule: If $\boldsymbol{x}^{(i)}$ is misclassified, do $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + y^{(i)}\boldsymbol{x}^{(i)}$

The Perceptron Cost Function

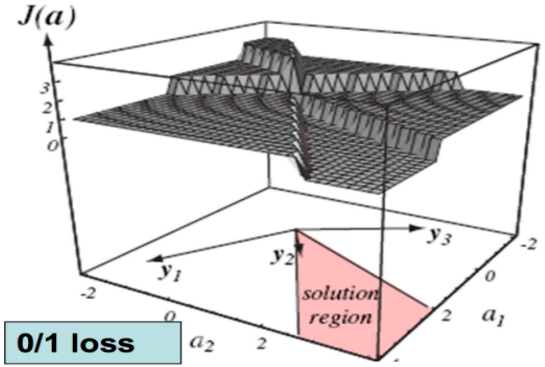
sign function



- Prediction is correct if $y^{(i)}\boldsymbol{x}^{(i)}\boldsymbol{\theta} > 0$
- 0/1 loss

$$J_{0/1}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \ell(sign(x^{(i)}\boldsymbol{\theta}), y^{(i)})$$

where $\ell()$ is 0 if the prediction is correct, 1 otherwise



Doesn't produce a useful gradient

No gradient ➔ We cannot use gradient-based optimization methods. Does NOT mean 'unsolvable'

Based on slide by Alan Fern

Batch Perceptron

Given training data $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$
Let $\boldsymbol{\theta} \leftarrow [0, 0, \ldots, 0]$
Repeat:
    Let $\boldsymbol{\Delta} \leftarrow [0, 0, \ldots, 0]$
    for $i = 1 \ldots n$, do
        if $y^{(i)}\boldsymbol{x}^{(i)}\boldsymbol{\theta} \leq 0$ // prediction for ith instance is incorrect
            $\boldsymbol{\Delta} \leftarrow \boldsymbol{\Delta} + y^{(i)}\boldsymbol{x}^{(i)}$
    $\boldsymbol{\Delta} \leftarrow \boldsymbol{\Delta}/n$ // compute average update
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\boldsymbol{\Delta}$
Until $\|\boldsymbol{\Delta}\|_2 < \epsilon$

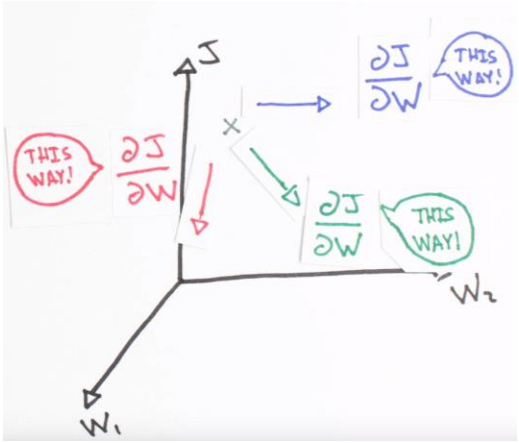
- Simplest case: α = 1 and don't normalize, yields the fixed increment perceptron
- Guaranteed to find a separating hyperplane if one exists

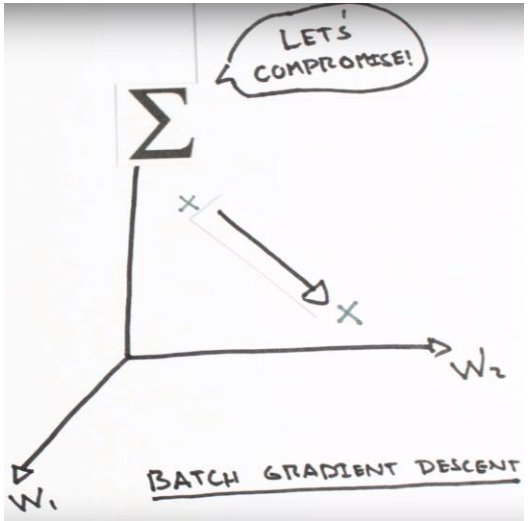
Batch Perceptron

Given training data $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$
Let $\boldsymbol{\theta} \leftarrow [0, 0, \ldots, 0]$
Repeat:
    Let $\boldsymbol{\Delta} \leftarrow [0, 0, \ldots, 0]$
    for $i = 1 \ldots n$, do
        if $y^{(i)}\boldsymbol{x}^{(i)}\boldsymbol{\theta} \leq 0$ // prediction for ith instance is incorrect
            $\boldsymbol{\Delta} \leftarrow \boldsymbol{\Delta} + y^{(i)}\boldsymbol{x}^{(i)}$
    $\boldsymbol{\Delta} \leftarrow \boldsymbol{\Delta}/n$ // compute average update
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\boldsymbol{\Delta}$
Until $\|\boldsymbol{\Delta}\|_2 < \epsilon$

- Simplest case: α = 1 and don't normalize, yields the fixed increment perceptron
- Guaranteed to find a separating hyperplane if one exists

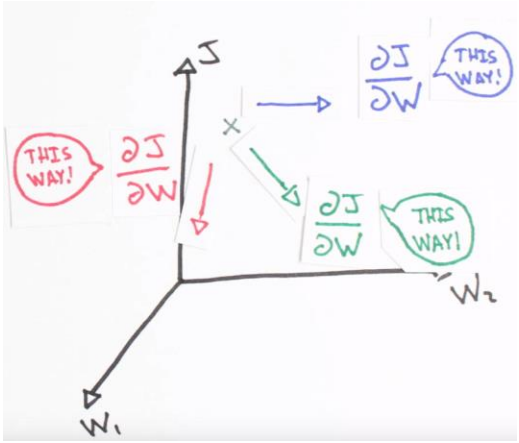
Based on slide by Alan Fern





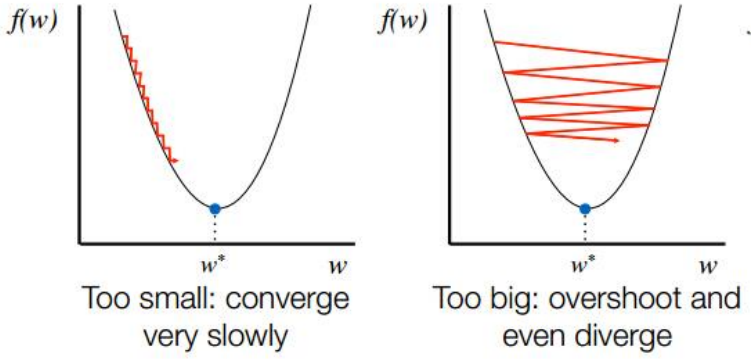
Batch Perceptron

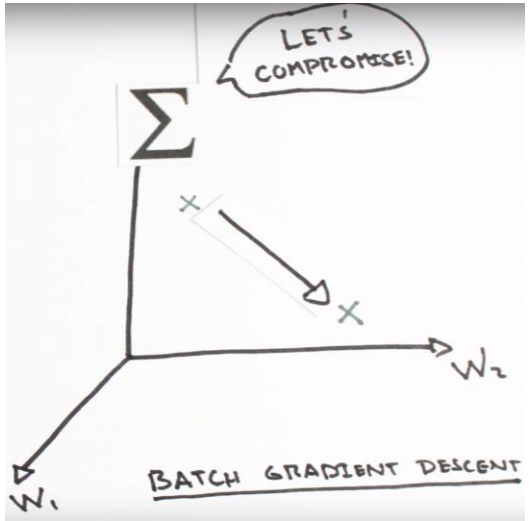
Given training data $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$
Let $\boldsymbol{\theta} \leftarrow [0, 0, \ldots, 0]$
Repeat:
    Let $\boldsymbol{\Delta} \leftarrow [0, 0, \ldots, 0]$
    for $i = 1 \ldots n$, do
        if $y^{(i)}\boldsymbol{x}^{(i)}\boldsymbol{\theta} \leq 0$ // prediction for i^{th} instance is incorrect
            $\boldsymbol{\Delta} \leftarrow \boldsymbol{\Delta} + y^{(i)}\boldsymbol{x}^{(i)}$
    $\boldsymbol{\Delta} \leftarrow \boldsymbol{\Delta}/n$ // compute average update
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\boldsymbol{\Delta}$
Until $\|\boldsymbol{\Delta}\|_2 < \epsilon$



- Simplest case: α = 1 and don't normalize, yields the fixed increment perceptron
- Guaranteed to find a separating hyperplane if one exists

Based on slide by Alan Fern





Online Perceptron Algorithm

Let $\boldsymbol{\theta} \leftarrow [0, 0, \ldots, 0]$
Repeat:
    Receive training example $(\boldsymbol{x}^{(i)}, y^{(i)})$
    if $y^{(i)}\boldsymbol{x}^{(i)}\boldsymbol{\theta} \leq 0$ // prediction is incorrect
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + y^{(i)}\boldsymbol{x}^{(i)}$

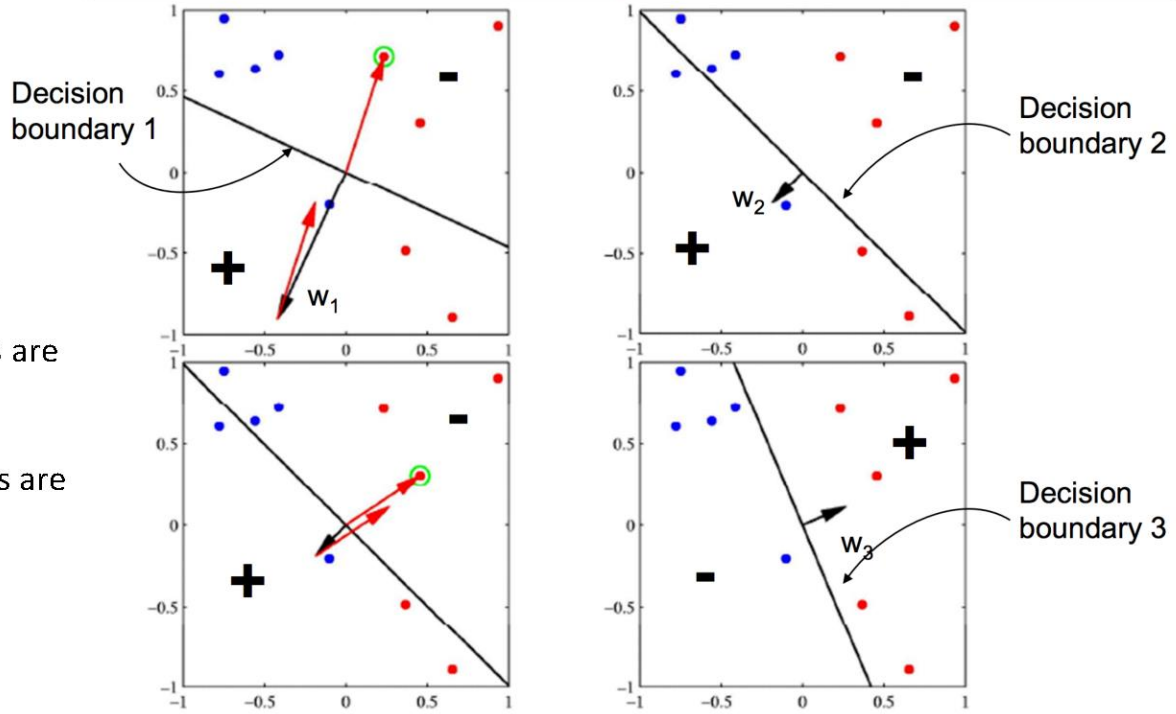
Online learning – the learning mode where the model update is performed each time a single observation is received

Batch learning – the learning mode where the model update is performed after observing the entire training set

Based on slide by Alan Fern

Online Perceptron Algorithm

When an error is made, moves the weight in a direction that corrects the error



Red points are labeled +

Blue points are labeled -

Animation: https://www.youtube.com/watch?v=vGwemZhPlsA

Gradient Descent

1. Initialize the parameters $\mathbf{w}$ to some guess (usually all zeros, or random values)
2. Update the parameters:
   $$\mathbf{w} = \mathbf{w} - \eta \nabla L(\mathbf{w})$$
   $$\eta = c_1 / (t^a + c_2)$$
3. Repeat step 2 until $\|\nabla L(\mathbf{w})\| < \theta$ or until the maximum number of iterations is reached.

Stochastic Gradient Descent

A variant of gradient descent makes updates using an approximate of the gradient that is only based on one instance at a time.

$$L_i(\mathbf{w}) = (y_i - \mathbf{w}^T\mathbf{x_i})^2$$

$$dL_i/dw_j = -2\, x_{ij}\, (y_i - \mathbf{w}^T\mathbf{x_i})$$

Stochastic Gradient Descent

General algorithm for SGD:

1. Iterate through the instances in a random order
   a) For each instance x_i, update the weights based on the gradient of the loss for that instance only:

$$\mathbf{w} = \mathbf{w} - \eta \nabla L_i(\mathbf{w}; \mathbf{x}_i)$$

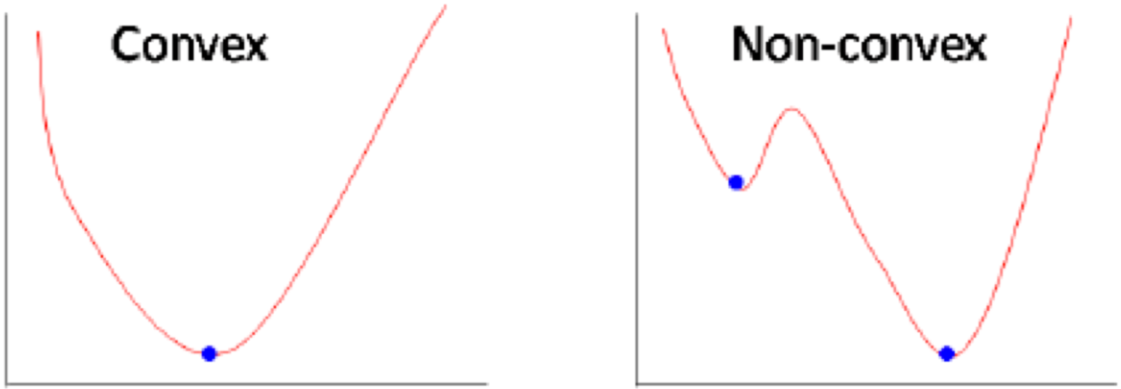
The gradient for one instance's loss is an approximation to the true gradient

- stochastic = random
  The *expected* gradient is the true gradient

Convexity

How do you know if you've found the global minimum, or just a local minimum?

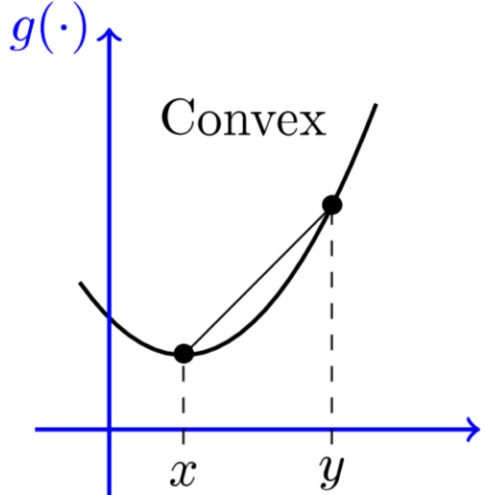
A **convex** function has only one minimum:



Convexity

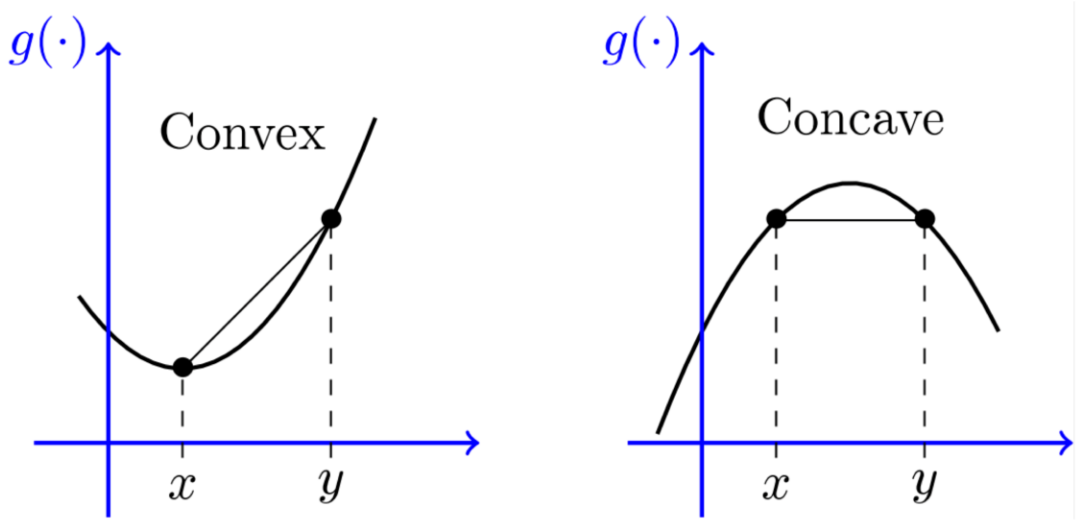
How do you know if you've found the global minimum, or just a local minimum?

A **convex** function has only one minimum:



Convexity

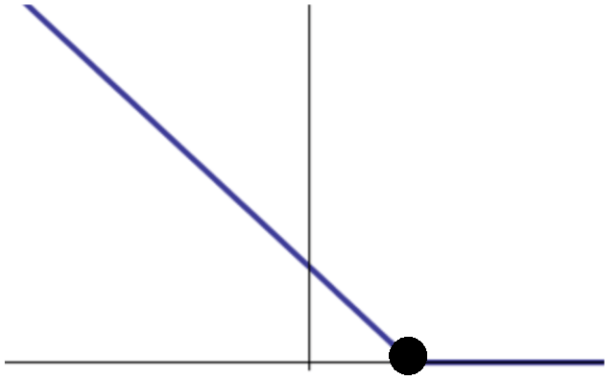
A **concave** function has only one maximum:

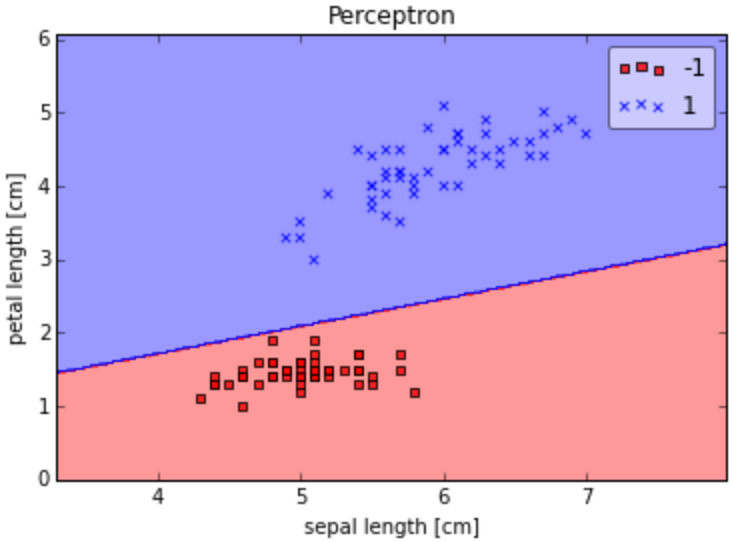


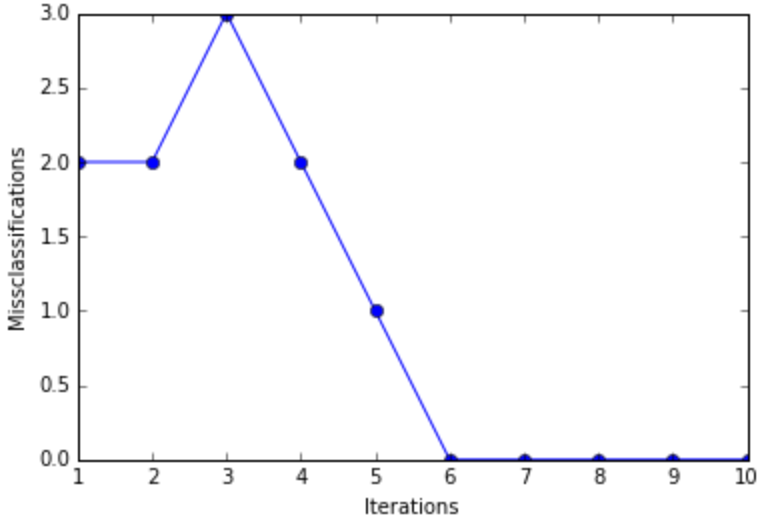
Convexity

Squared error is a convex loss function, as is the perceptron loss.

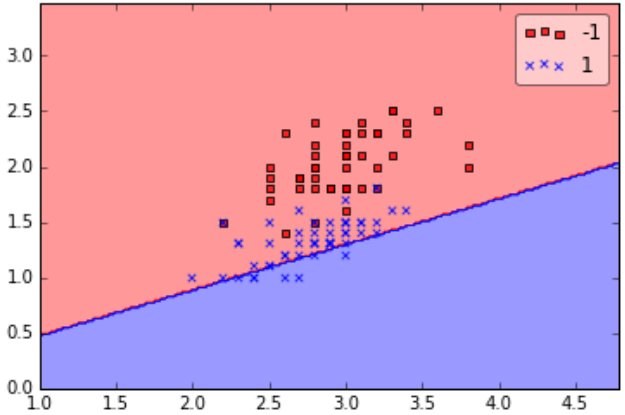
Note: convexity means there is only one minimum value, but there may be multiple parameters that result in that minimum value.

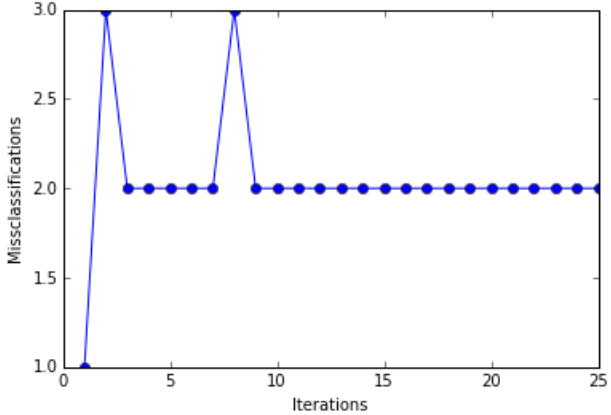



Perceptron
-1
1
petal length [cm]
sepal length [cm]
0
1
2
3
4
5
6
4
5
6
7

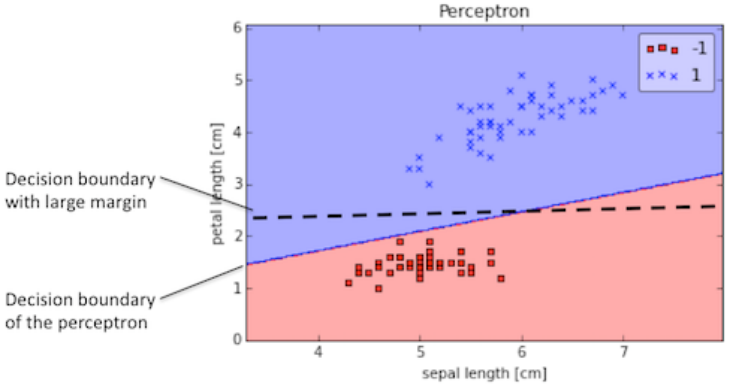

Missclassifications
Iterations
0.0
0.5
1.0
1.5
2.0
2.5
3.0
1
2
3
4
5
6
7
8
9
10

Problem #1: Assumes linear separability





Problem #2: Does not give the 'best' margin



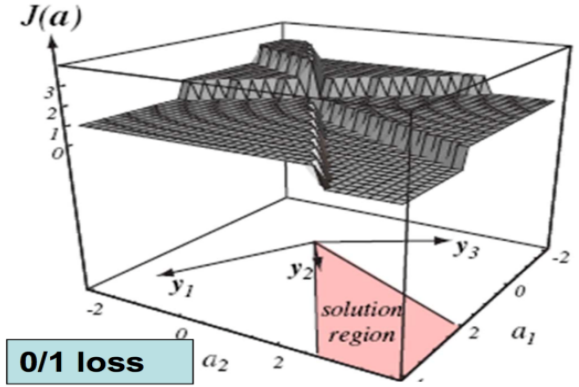
Improving the perceptron

Problem #0: Non-differentiable loss function

- Could have used 0/1 loss

$$J_{0/1}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \ell(sign(x^{(i)}\boldsymbol{\theta}), y^{(i)})$$

where $\ell()$ is 0 if the prediction is correct, 1 otherwise



Doesn't produce a useful gradient

Based on slide by Alan Fern

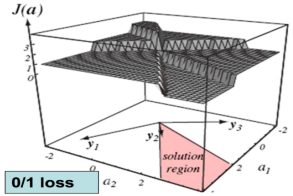
Improving the perceptron

Problem #0: Non-differentiable loss function

- Could have used 0/1 loss

$$J_{0/1}(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \ell(sign(x^{(i)}\boldsymbol{\theta}), y^{(i)})$$

where $\ell()$ is 0 if the prediction is correct, 1 otherwise



Doesn't produce a useful gradient

Based on slide by Alan Fern

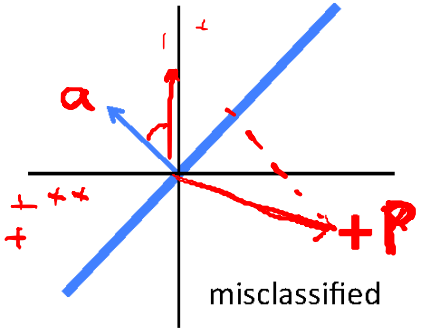
Solution: Minimize the 'misclassification distance'

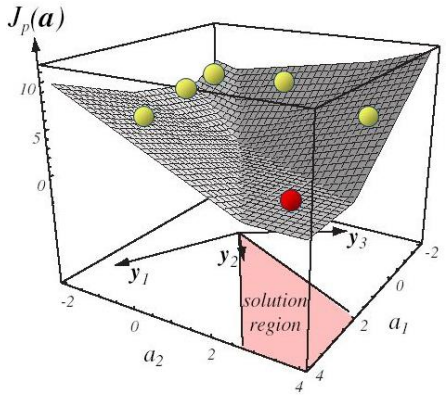
$$J_p(\mathbf{a}) = \sum_{\mathbf{x}\in\mathbb{M}} (-\mathbf{a}^{\mathbf{T}}\mathbf{x})$$

$a^T x = 0$

$a^T p$

$\mathbb{M}$ ➡ Set of misclassified samples

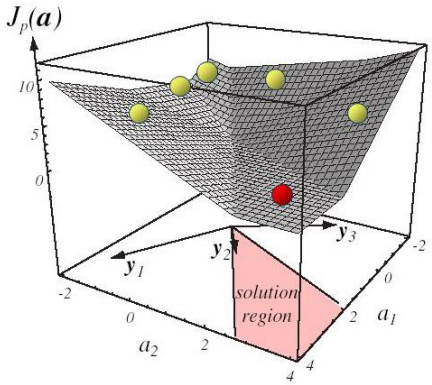




Solution: Minimize the ‘misclassification distance’

$$J_p(\mathbf{a}) = \sum_{\mathbf{x}\in\mathbb{M}} (-\mathbf{a}^{\mathbf{T}}\mathbf{x})$$

$\mathbb{M}$ ➡ Set of misclassified samples



$$\nabla J_p(\mathbf{a}) = \sum_{\mathbf{x}\in\mathbb{M}} (-\mathbf{x})$$

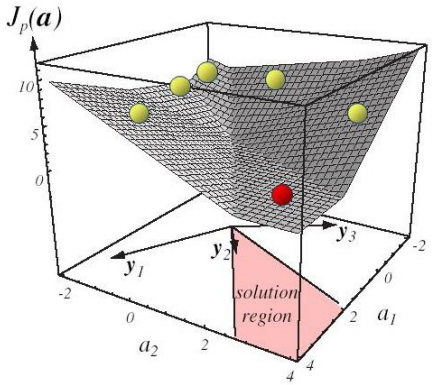
$$a^{(k+1)} = a^k - \eta^k \nabla J_p(a)$$

$$a^{(k+1)} = a^k + \eta^k \sum_{\mathbf{x}\in\mathbb{M}} \mathbf{x}$$

Solution: Minimize the 'misclassification distance'

$$J_p(\mathbf{a}) = \sum_{\mathbf{x}\in\mathbb{M}} (-\mathbf{a}^\mathbf{T}\mathbf{x})$$

$\mathbb{M}$ ➡ Set of misclassified samples



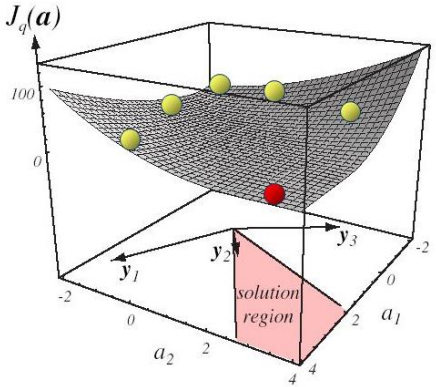
$$\nabla J_p(\mathbf{a}) = \sum_{\mathbf{x}\in\mathbb{M}} (-\mathbf{x})$$

$$a^{(k+1)} = a^k - \eta^k \nabla J_p(a)$$

$$a^{(k+1)} = a^k + \eta^k \sum_{\mathbf{x}\in\mathbb{M}} \mathbf{x}$$

Solution: Minimize the **square** of 'misclassification distance'

$$J_q(\mathbf{a}) = \sum_{\mathbf{x}\in\mathbb{M}} (\mathbf{a}^\mathbf{T}\mathbf{x})^2$$



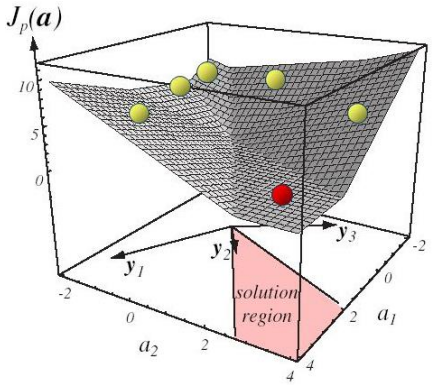
$$\nabla J_q(a) =$$

$$a^{(k+1)} =$$

Solution: Minimize the 'misclassification distance'

$$J_p(\mathbf{a}) = \sum_{\mathbf{x}\in\mathbb{M}} (-\mathbf{a}^\mathbf{T}\mathbf{x})$$

$\mathbb{M}$ ➡ Set of misclassified samples



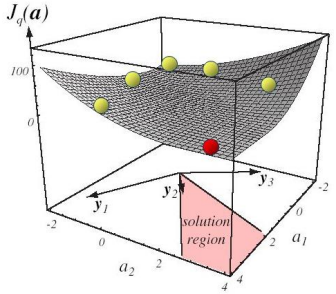
$$\nabla J_p(\mathbf{a}) = \sum_{\mathbf{x}\in\mathbb{M}} (-\mathbf{x})$$

$$a^{(k+1)} = a^k - \eta^k \nabla J_p(a)$$

$$a^{(k+1)} = a^k + \eta^k \sum_{\mathbf{x}\in\mathbb{M}} \mathbf{x}$$

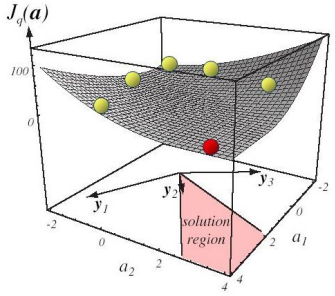
Solution: Minimize the **square** of 'misclassification distance'

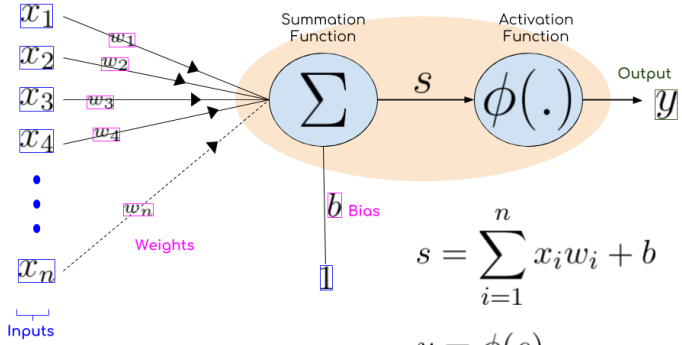
$$J_q(\mathbf{a}) = \sum_{\mathbf{x}\in\mathbb{M}} (\mathbf{a}^\mathbf{T}\mathbf{x})^2$$



Solution: Minimize the **square** of 'misclassification distance'

$$J_q(\mathbf{a}) = \sum_{\mathbf{x} \in \mathbb{M}} (\mathbf{a}^{\mathbf{T}}\mathbf{x})^2$$

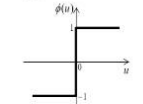




$$s = \sum_{i=1}^{n} x_i w_i + b$$

$$y = \phi(s)$$

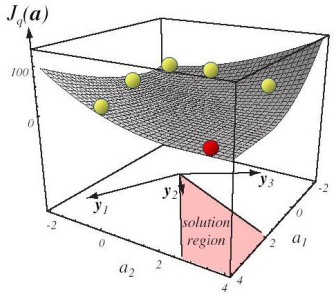
sign function

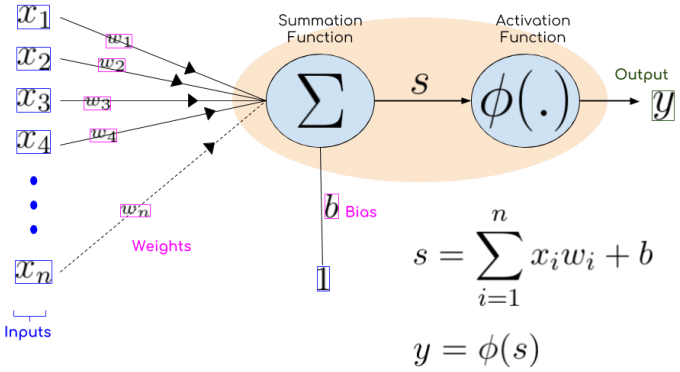


$$\phi_{sign}(u) = \begin{cases} 1 & \text{if } u \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

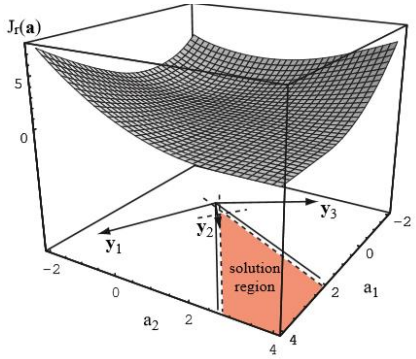
Solution: Minimize the **square** of 'misclassification distance'

$$J_q(\mathbf{a}) = \sum_{\mathbf{x}\in\mathbb{M}} (\mathbf{a^T x})^2$$

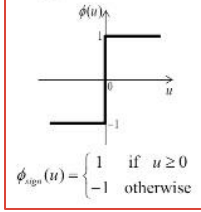




$$J_r(\mathbf{a}) = \sum_{\mathbf{x}\in\mathbb{M}} \frac{(\mathbf{a^T x} - \mathbf{b})^2}{||x||^2}$$

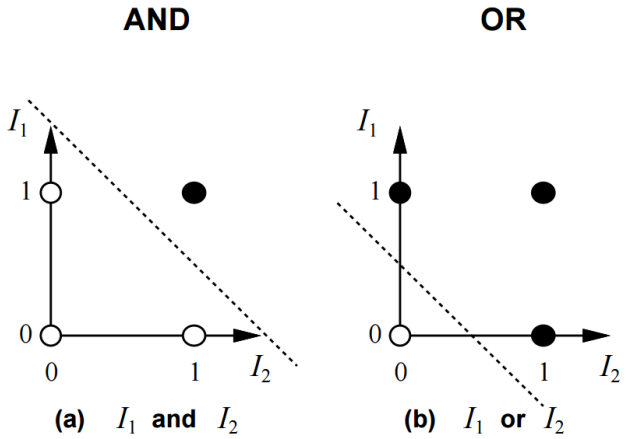


sign function

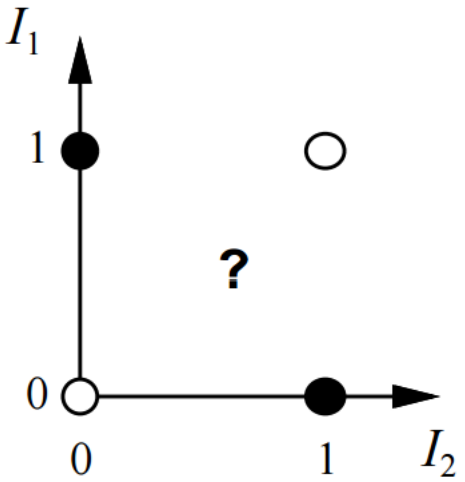


Loss function is changing. Activation function has remained fixed ('sign').

Can a Perceptron Separate Distributions Which are Not Linearly Separable ???



Linearly Separable Classes



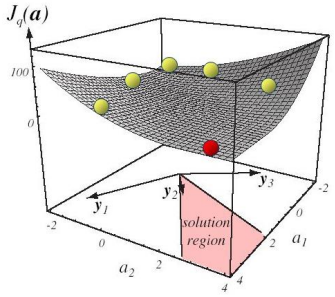
(c) I_1 xor I_2

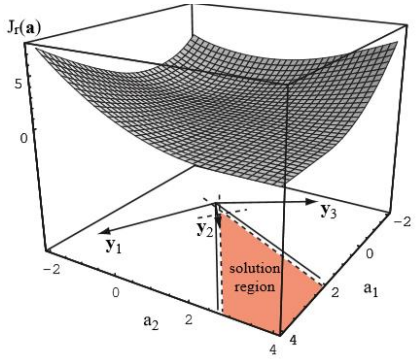
Solution: Minimize the **square** of 'misclassification distance'

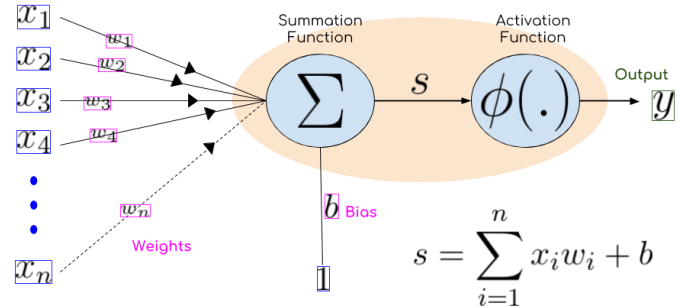
$$J_q(\mathbf{a}) = \sum_{\mathbf{x}\in\mathbb{M}} (\mathbf{a}^{\mathbf{T}}\mathbf{x})^2$$

Converges even when classes are not separable

$$J_r(\mathbf{a}) = \sum_{\mathbf{x}\in\mathbb{M}} \frac{(\mathbf{a}^{\mathbf{T}}\mathbf{x} - \mathbf{b})^2}{||x||^2}$$



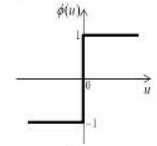




$$s = \sum_{i=1}^{n} x_i w_i + b$$

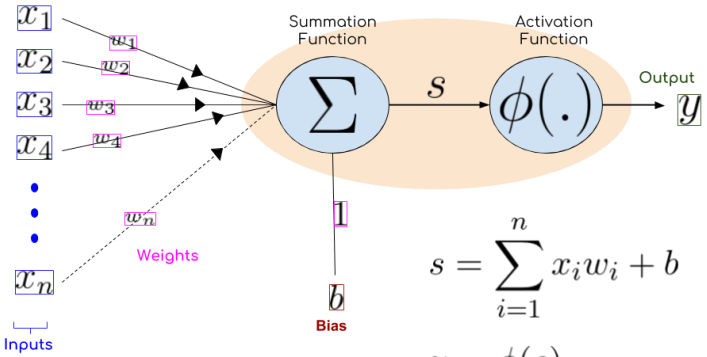
$$y = \phi(s)$$

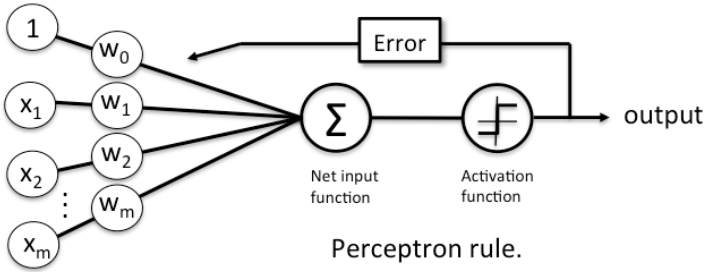
sign function



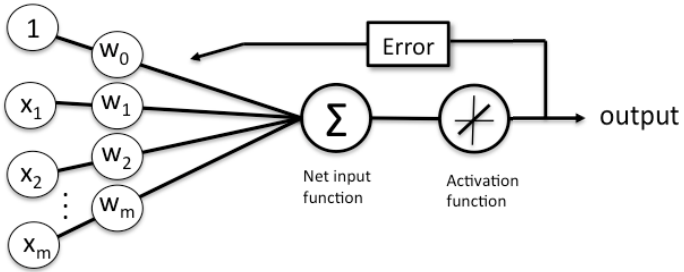
$$\phi_{sign}(u) = \begin{cases} 1 & \text{if } u \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

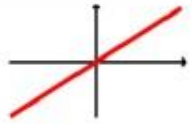
Regression





Perceptron rule.



Activation Function	Equation	Example	1D Graph
Linear	$\phi(z) = z$	Adaline, linear regression	

Regression — Gradient Descent Update

$$\mathcal{L}(\hat{y}, y) = (\hat{y} - y)^2$$

$$J_i = \mathcal{L}(\hat{y}_i - y_i)^2$$

$$J_i = \mathcal{L}(\hat{y}, y)$$

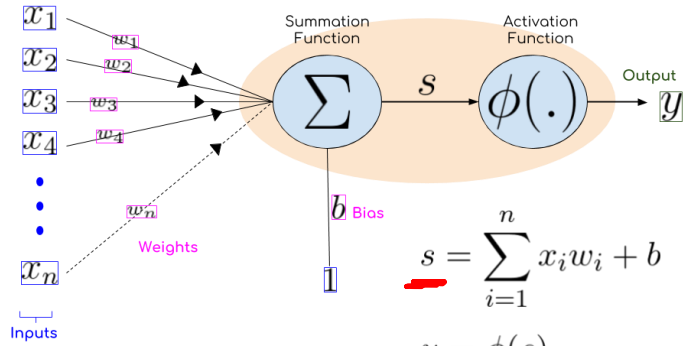
$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w}$$

$$\Delta = 1 \cdot 2(\hat{y} - y) \otimes$$

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

$$w = w + \Delta$$

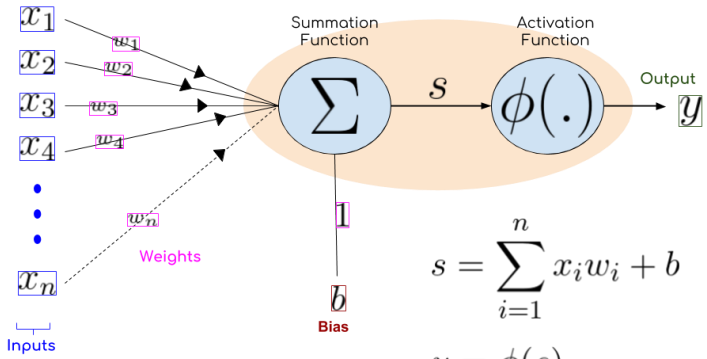
Activation Function	Equation
Linear	$\phi(z) = z$



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

$$y = \phi(s)$$

ACTIVATION FUNCTIONS

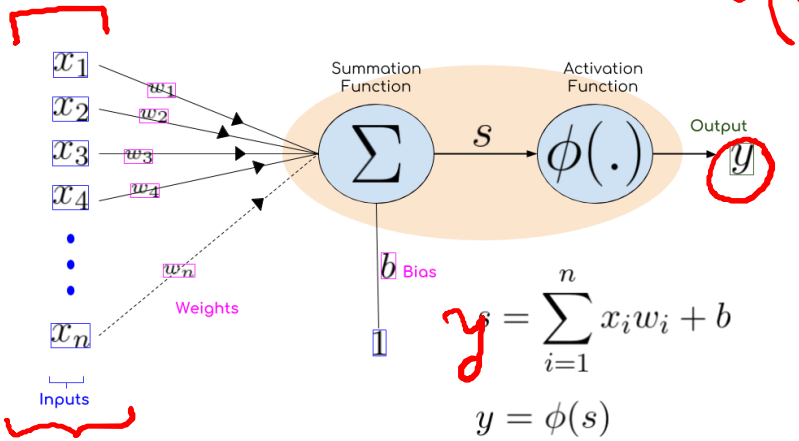


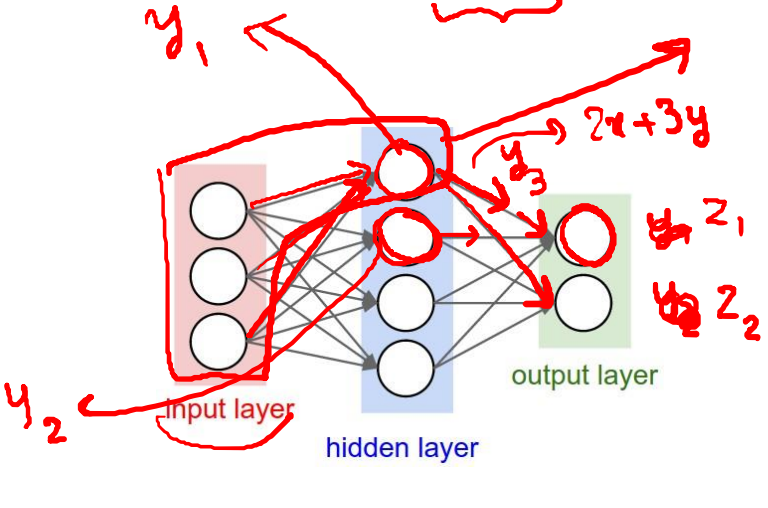
$$s = \sum_{i=1}^{n} x_i w_i + b$$

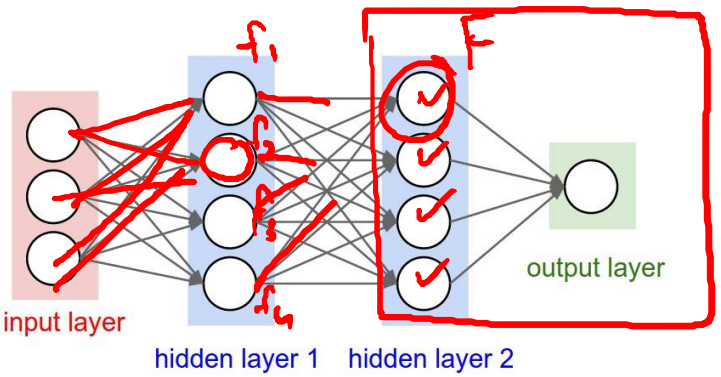
$$y = \phi(s)$$

Activation Function	Equation	Example	1D Graph
Linear	$\phi(z) = z$	Adaline, linear regression	
Unit Step (Heaviside Function)	$\phi(z) = \begin{cases} 0 & z < 0 \\ 0.5 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Sign (signum)	$\phi(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Piece-wise Linear	$\phi(z) = \begin{cases} 0 & z \leq -½ \\ z + ½ & -½ \leq z \leq ½ \\ 1 & z \geq ½ \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression,	
Hyperbolic Tangent (tanh)	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$		
ReLU	$\phi(z) = \begin{cases} 0 & z < 0 \\ z & z > 0 \end{cases}$		

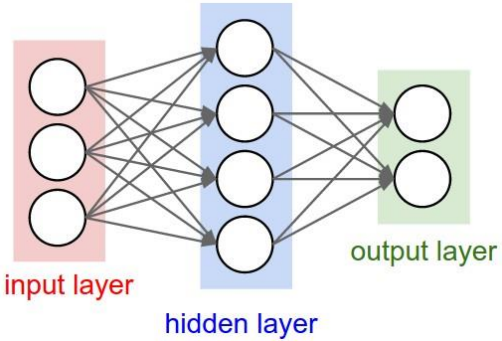
Why Use Only One Neuron ?

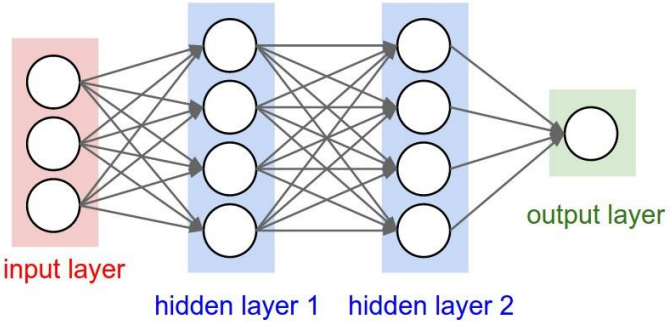


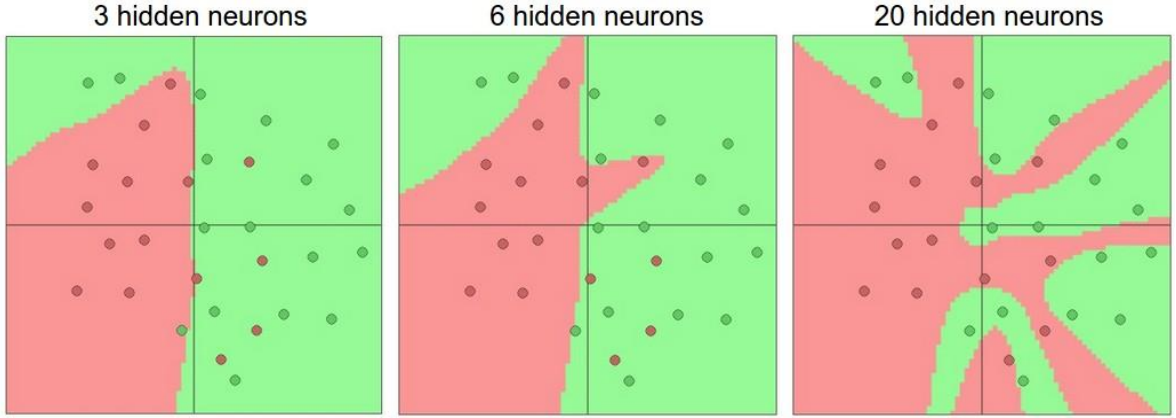




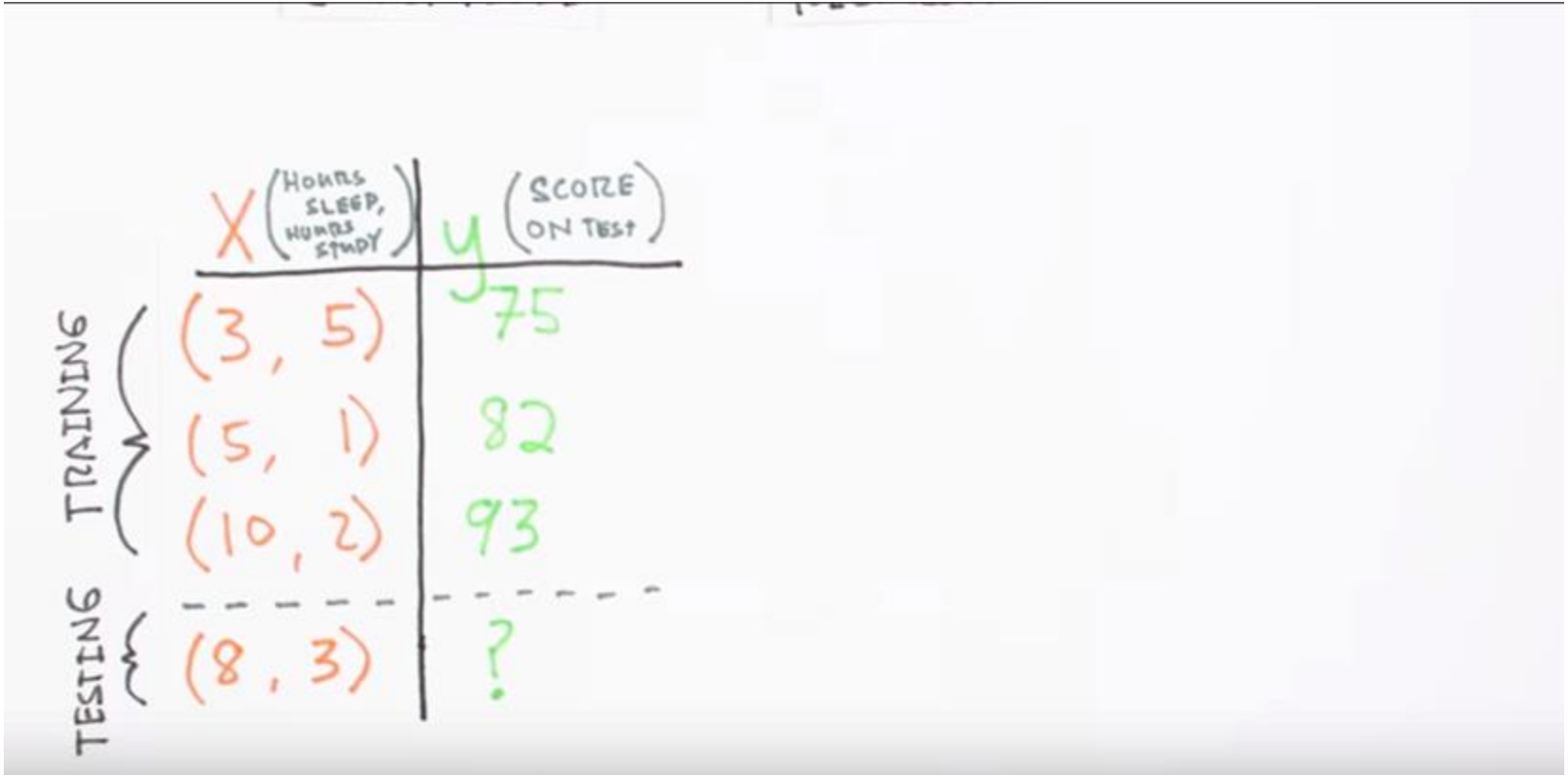
Why Use Only One Neuron ?



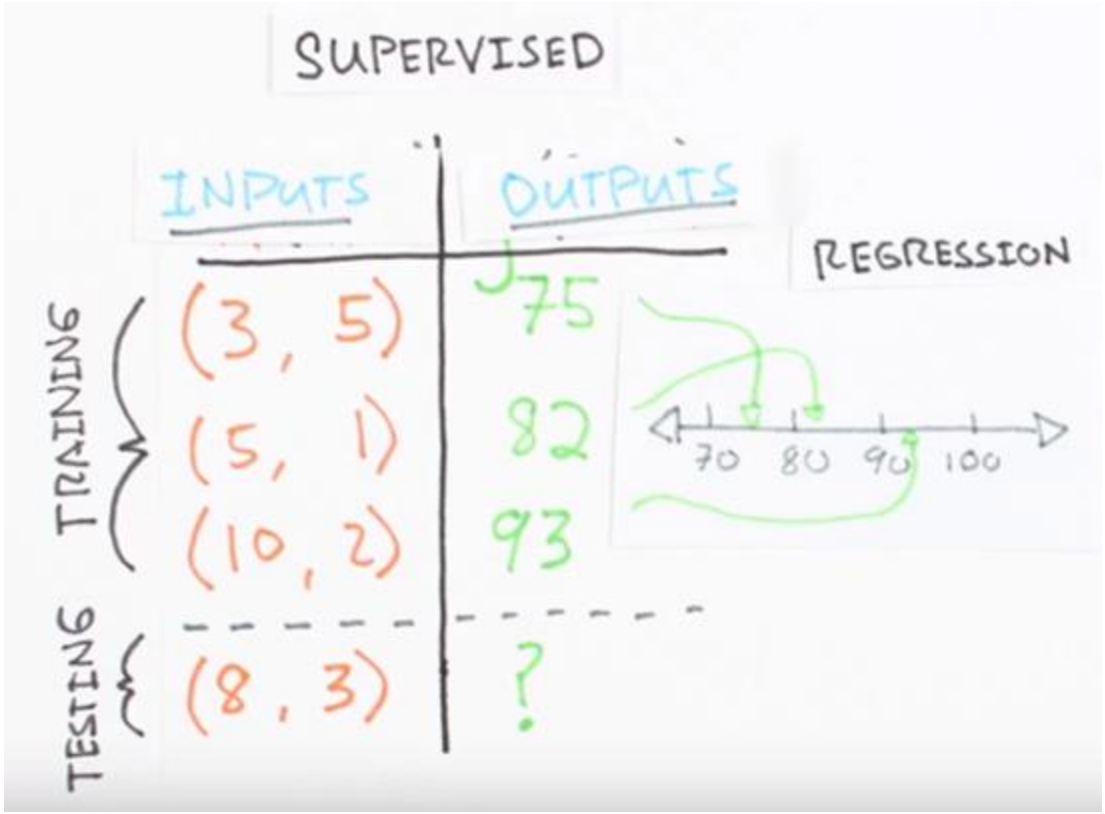




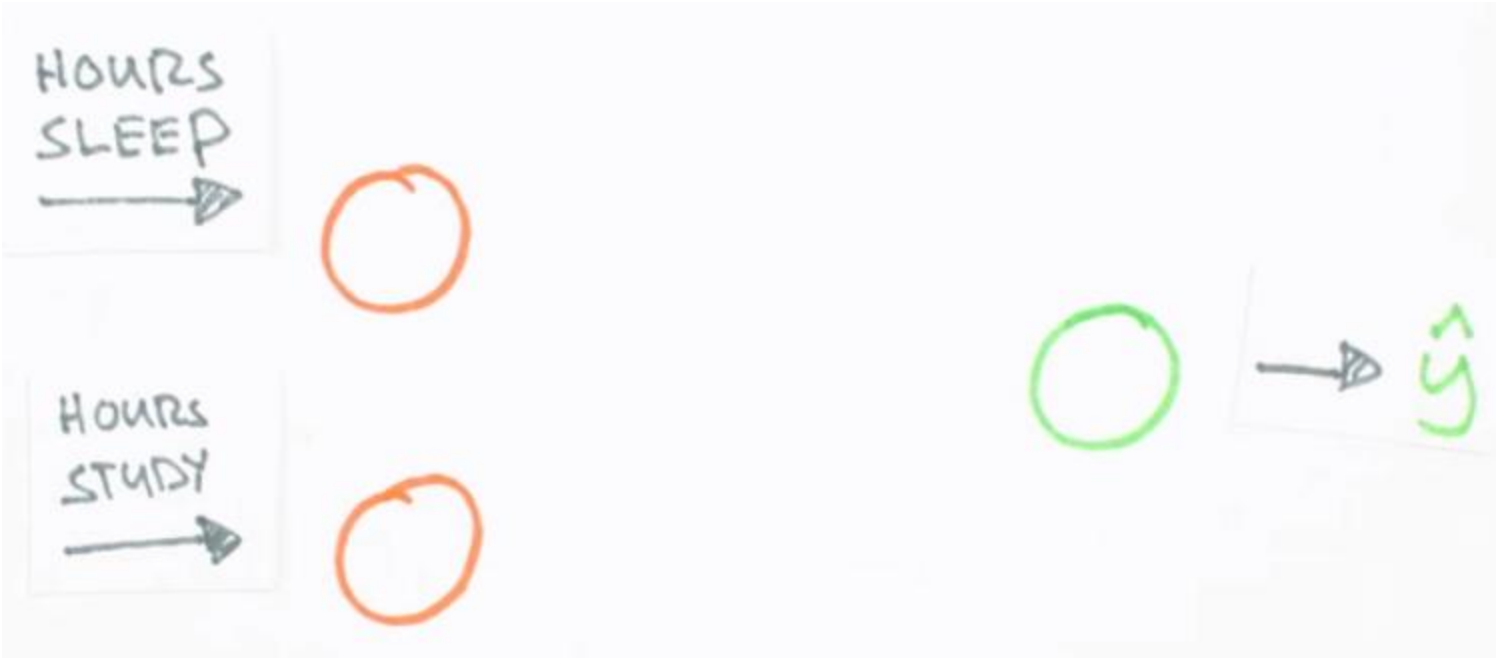
Multi-Neuron Networks



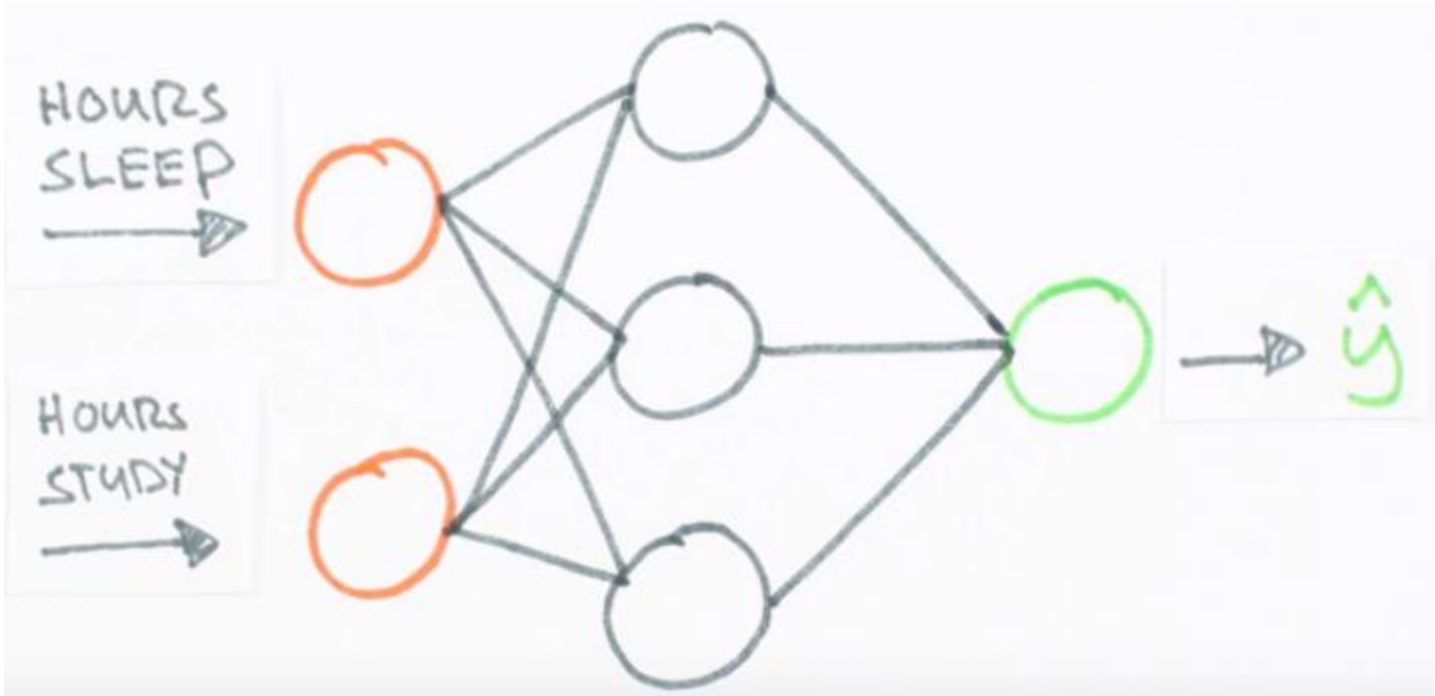
Multi-Neuron Networks



Multi-Neuron Networks :: Architecture

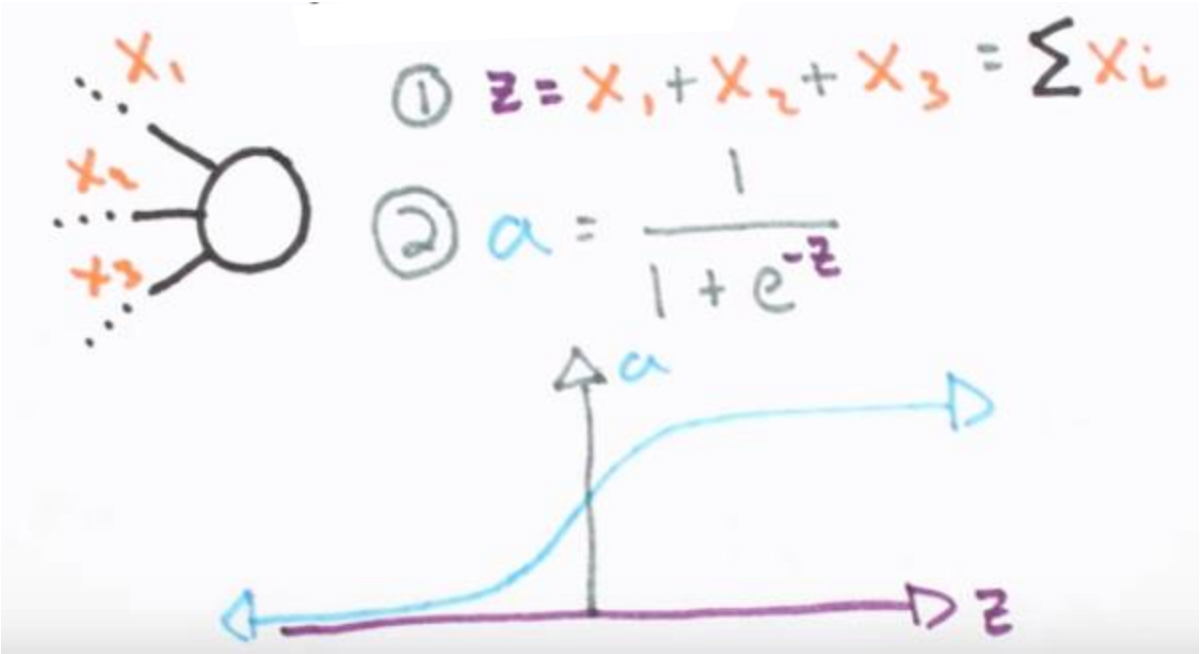


Multi-Neuron Networks :: Architecture



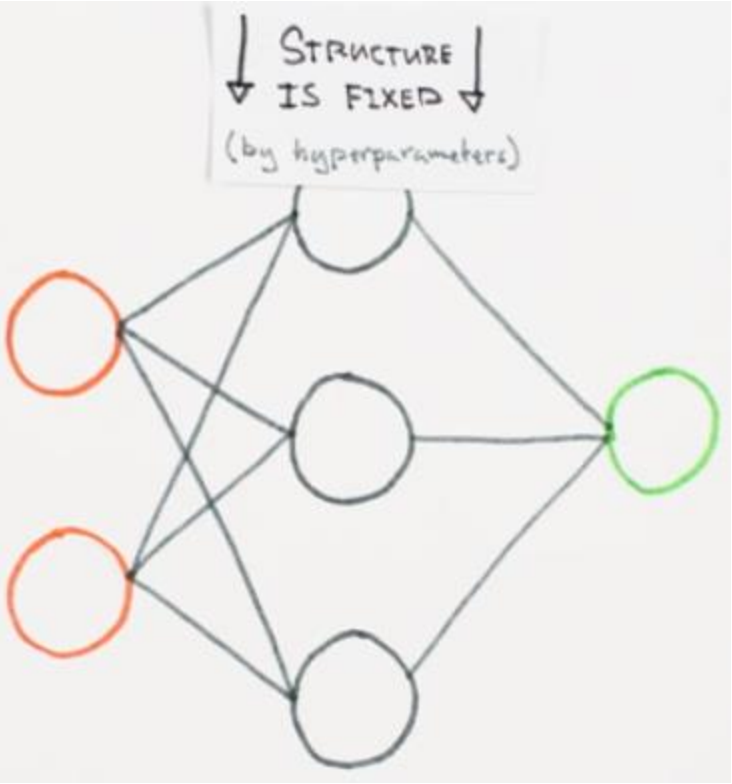
Multi-Neuron Networks :: Architecture

Neuron



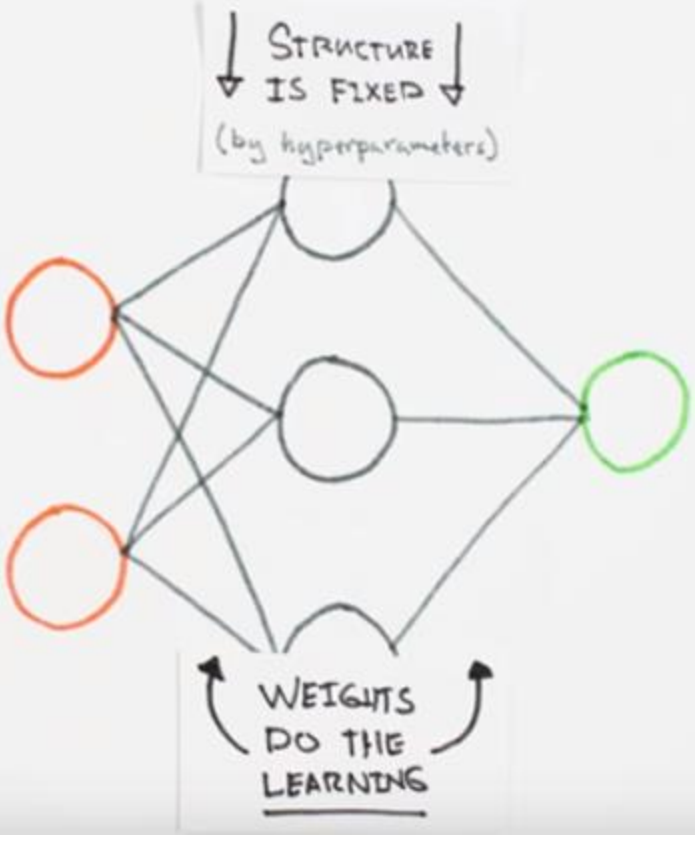
Multi-Neuron Networks :: Architecture

```
In [ ]: class Neural_Network(object):
            def __init__(self):
                #Define HyperParameters
                self.inputLayerSize = 2
                self.outputLayerSize = 1
                self.hiddenLayerSize = 3
```



Multi-Neuron Networks :: Architecture

```
In [ ]: class Neural_Network(object):
            def __init__(self):
                #Define HyperParameters
                self.inputLayerSize = 2
                self.outputLayerSize = 1
                self.hiddenLayerSize = 3
```



Multi-Neuron Networks :: training

Initialize network with random weights

While [not converged]

Do forward prop

Do backprop and determine change in weights

Update all weights in all layers

Multi-Neuron Networks :: training

Initialize network with random weights

While [not converged]

Do forward prop

Do backprop and determine change in weights

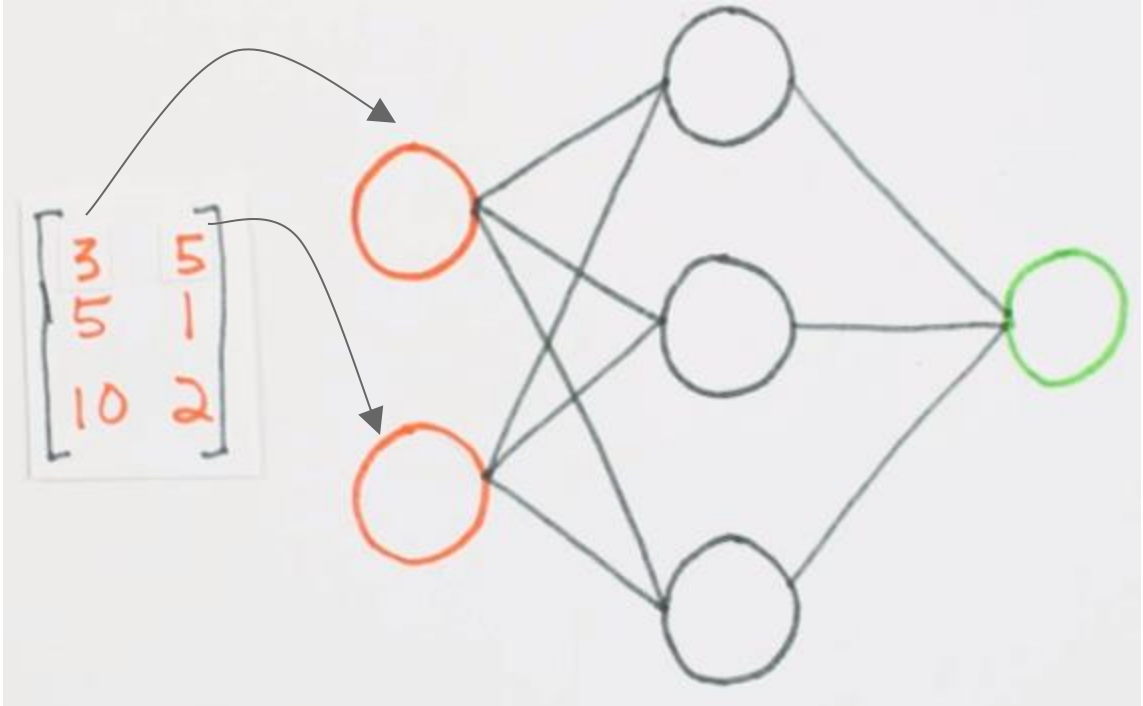
Update all weights in all layers

```
Initialize weights w ; // Random or 0

Until [all examples correctly classified]
        For each training sample (x,y)
                Compute yt := xTw
                 if y == yt // Correctly classified
                        continue ;
                else // Update weights
                        dw = (y - yt) * x ;
                        w   = w + dw;
                EndIf
        EndFor
EndUntil
```

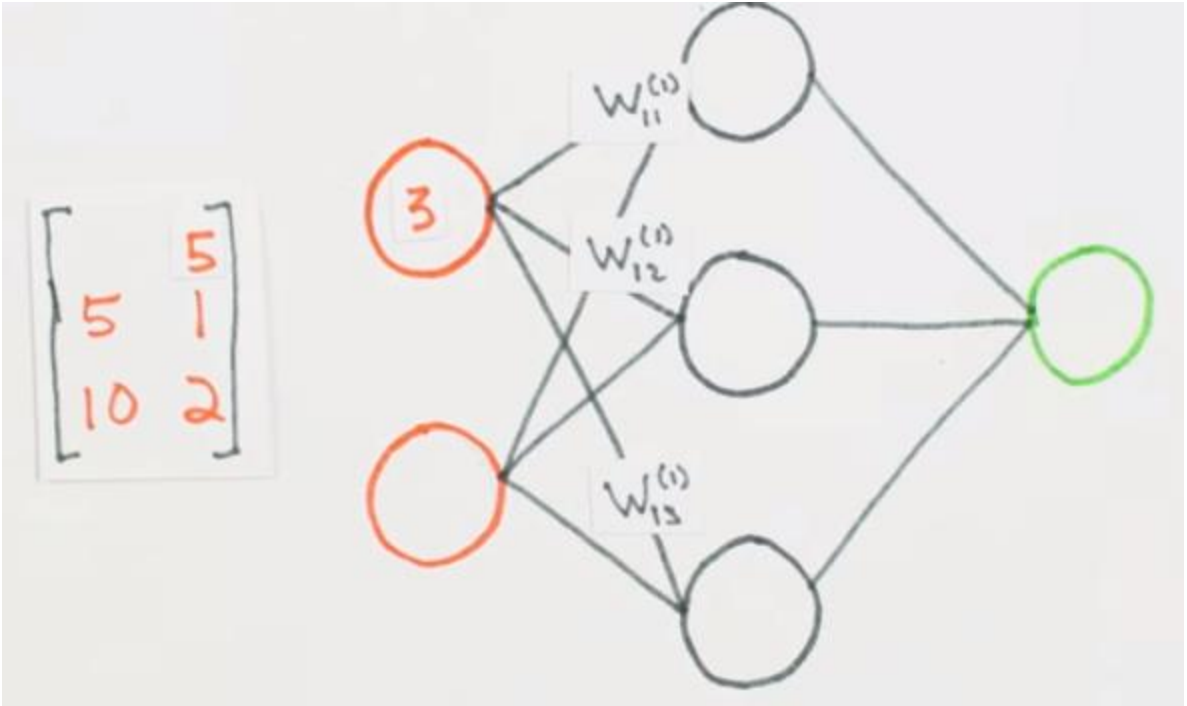
Multi-Neuron Networks :: forward propagation

```
In [1]: class Neural_Network(object):
            def __init__(self):
                #Define HyperParameters
                self.inputLayerSize = 2
                self.outputLayerSize = 1
                self.hiddenLayerSize = 3

            def forward(self, X):
                #Propagate inputs through network
```

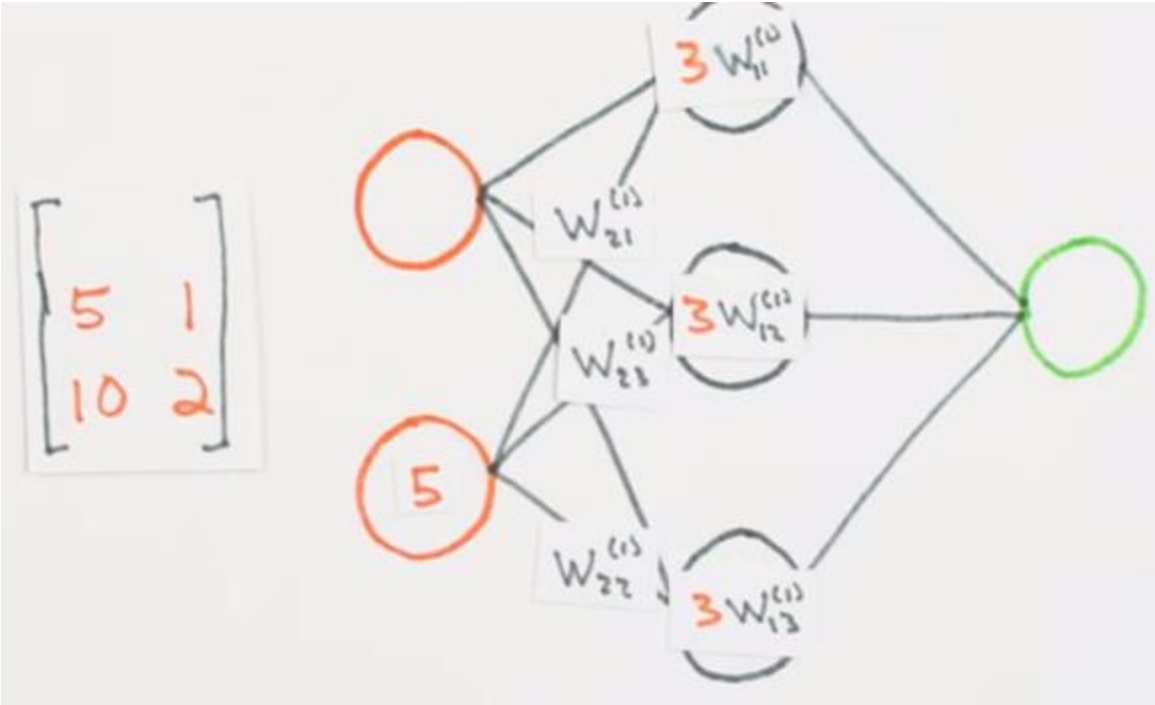
Multi-Neuron Networks :: forward propagation



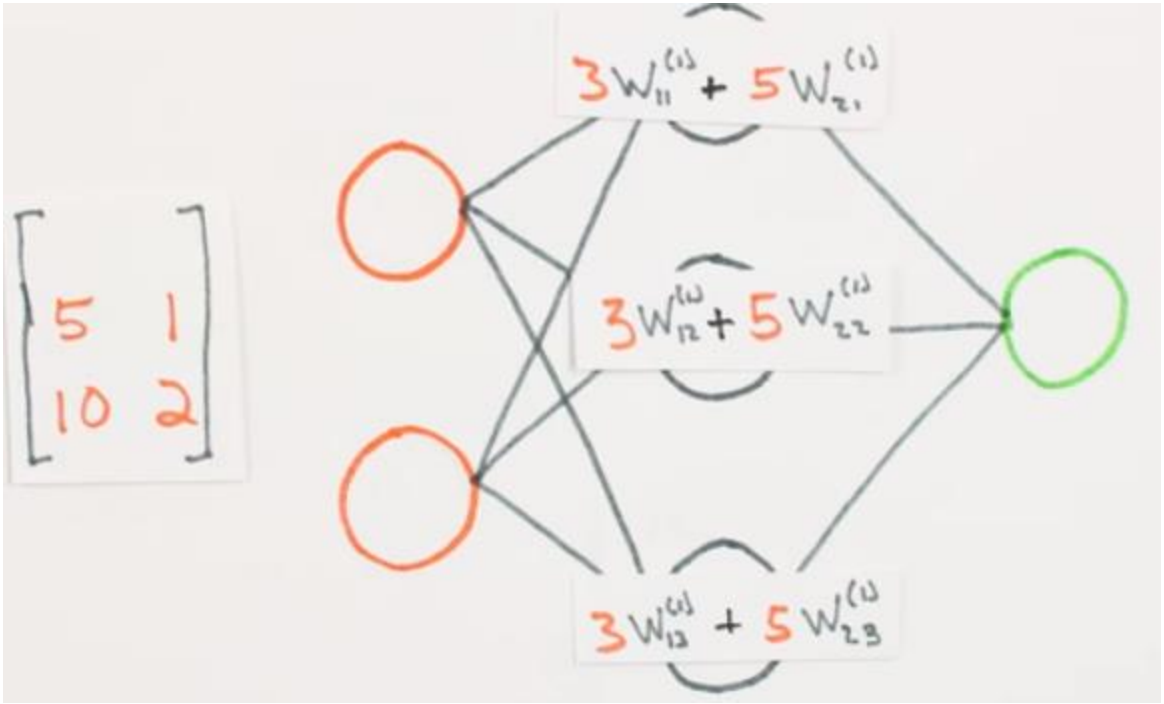
Multi-Neuron Networks :: forward propagation



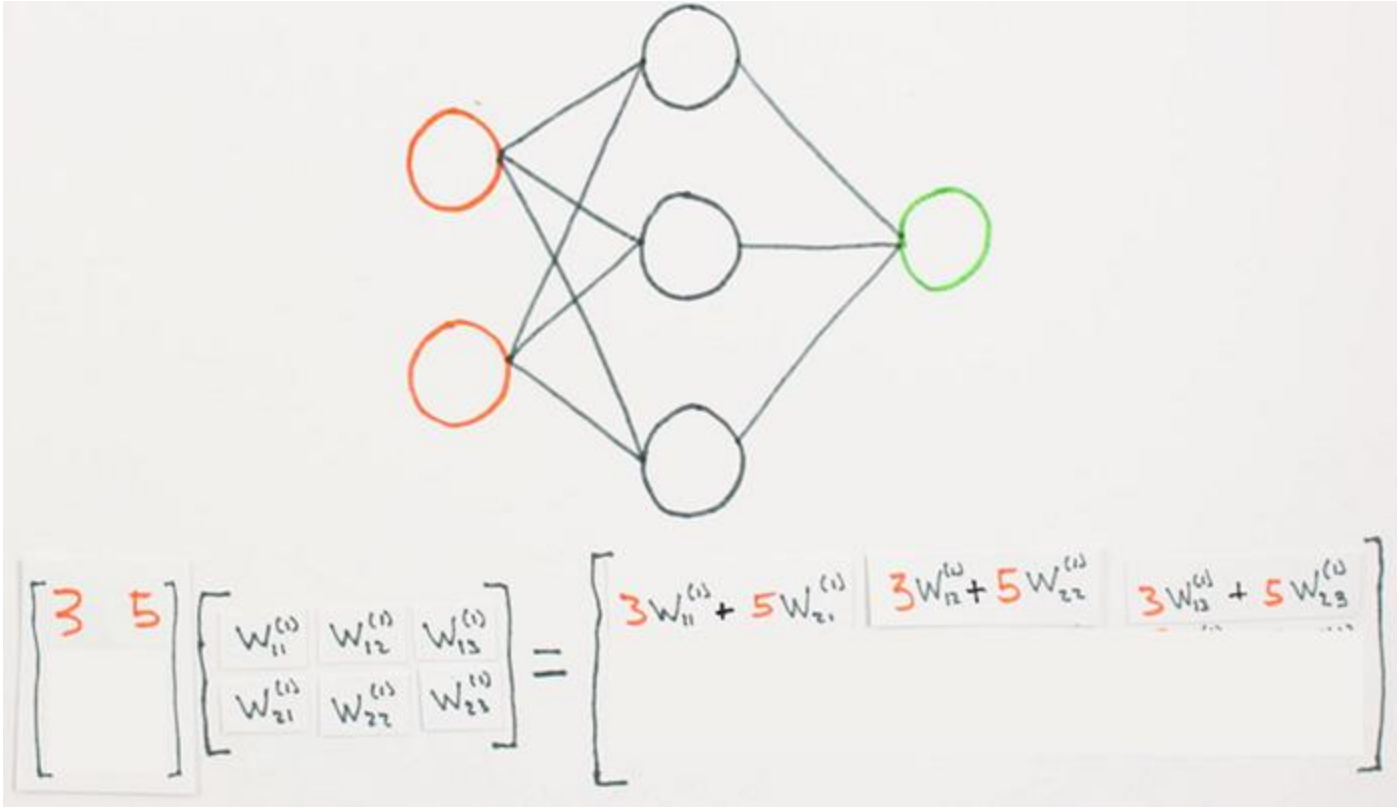
Multi-Neuron Networks :: forward propagation



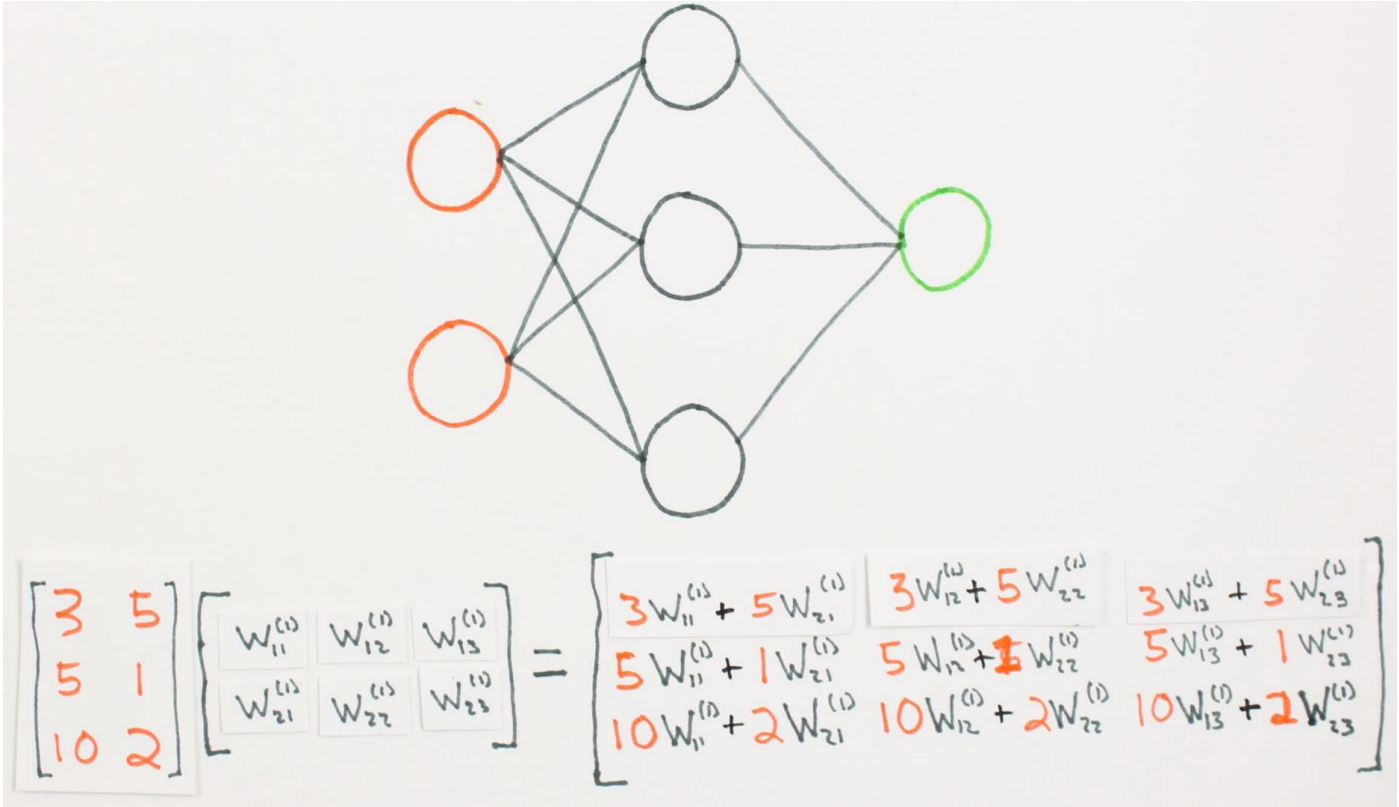
Multi-Neuron Networks :: forward propagation



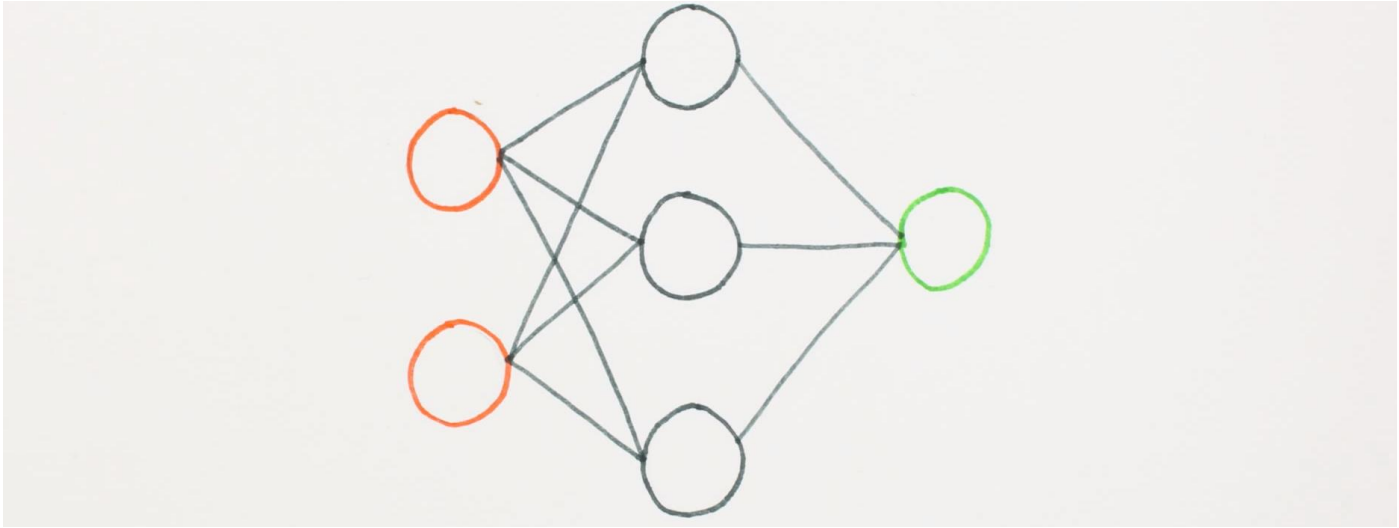
Multi-Neuron Networks :: forward propagation



Multi-Neuron Networks :: forward propagation

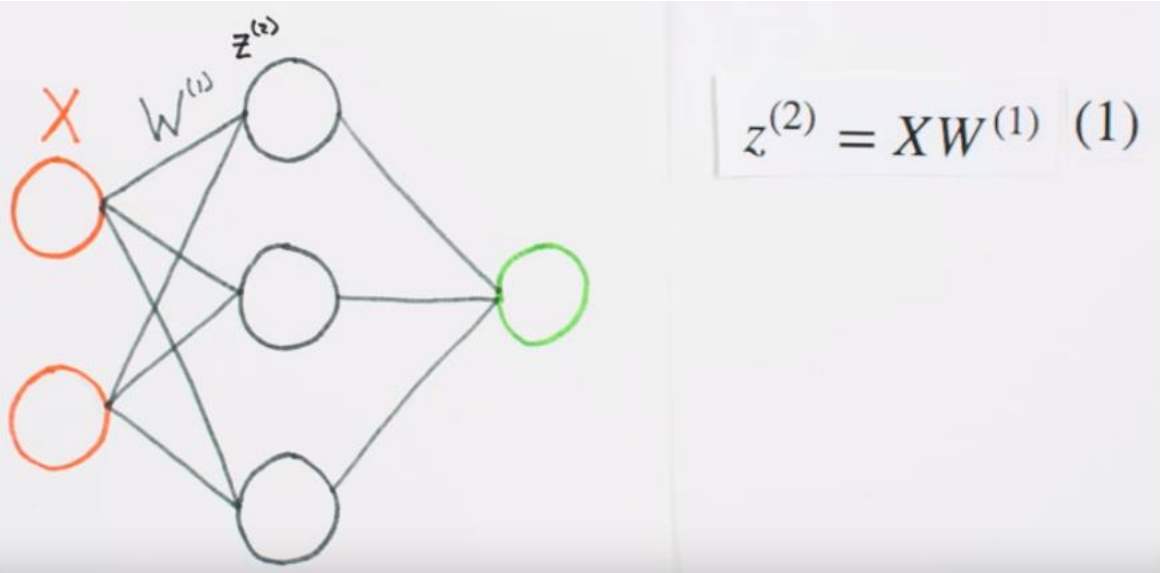


Multi-Neuron Networks :: forward propagation

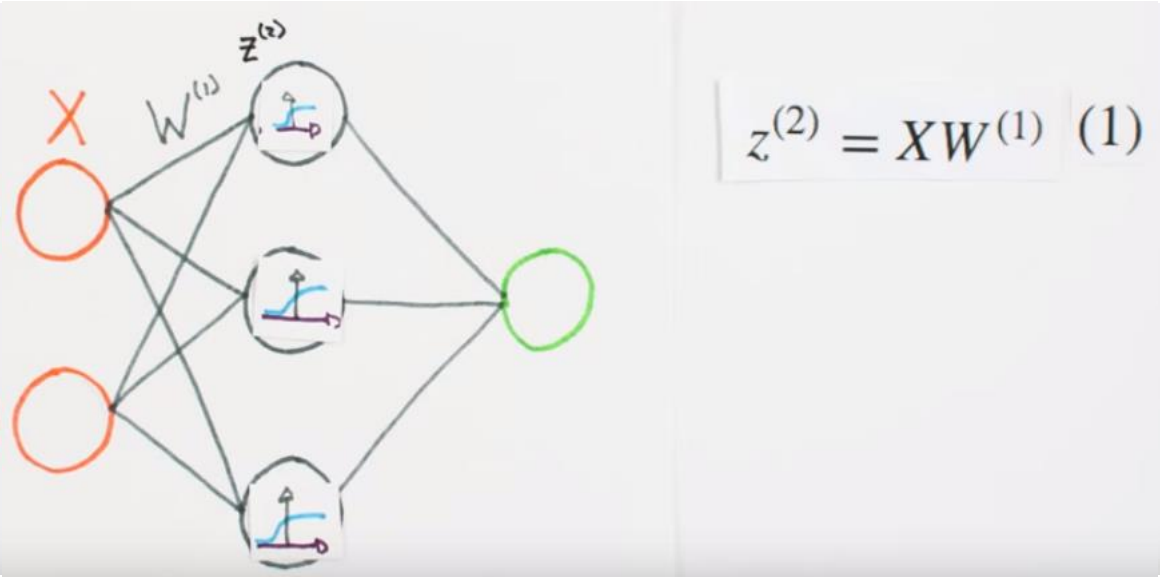


$$X \quad W^{(1)} = Z^{(2)}$$

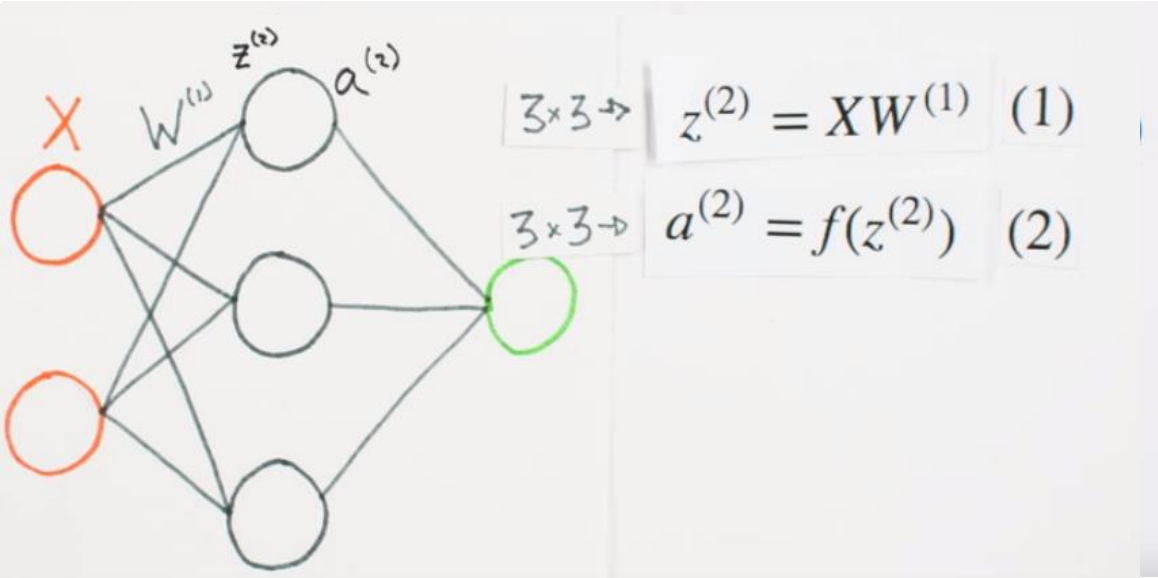
Multi-Neuron Networks :: forward propagation



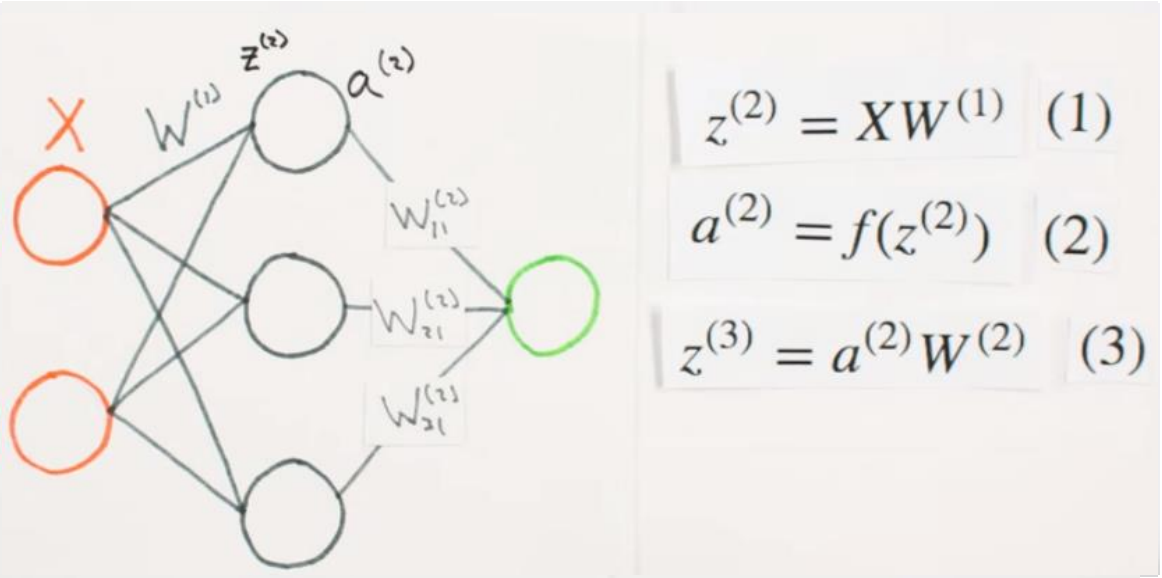
Multi-Neuron Networks :: forward propagation



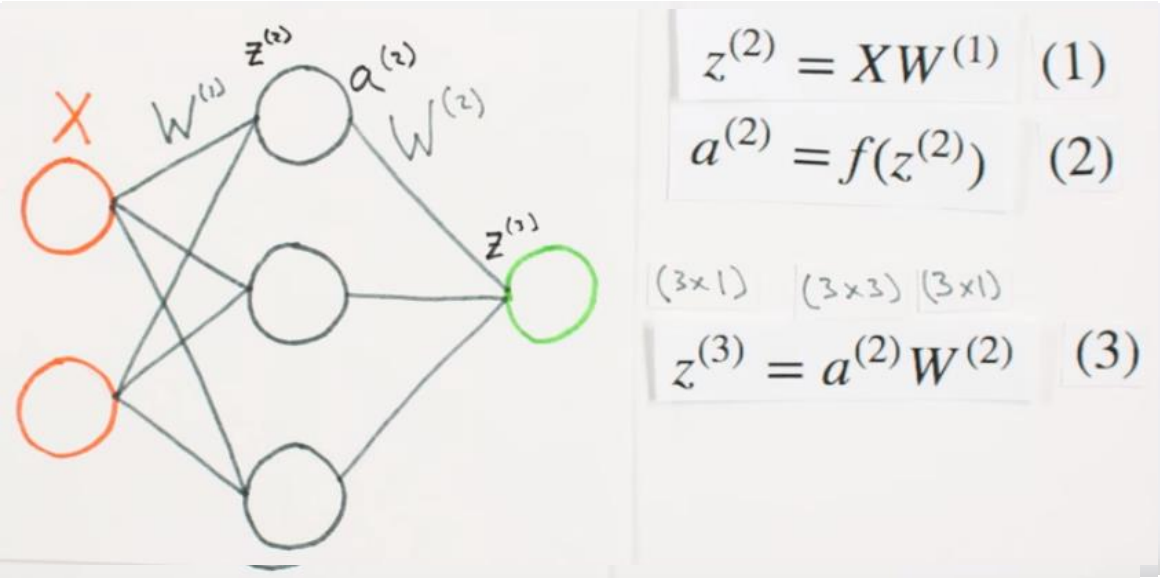
Multi-Neuron Networks :: forward propagation



Multi-Neuron Networks :: forward propagation



Multi-Neuron Networks :: forward propagation

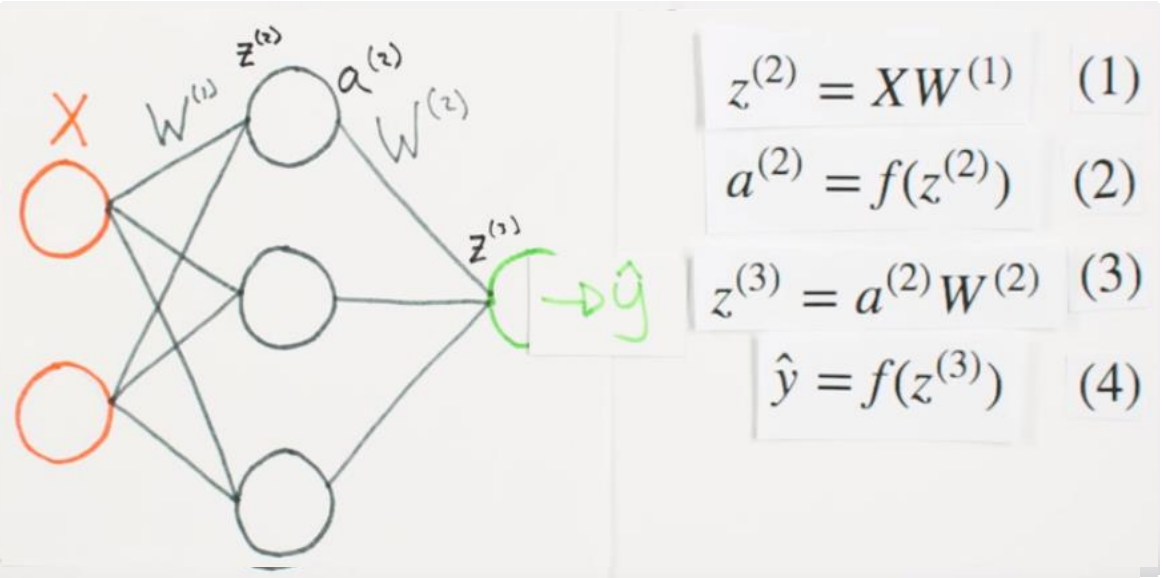


$$z^{(2)} = XW^{(1)} \quad (1)$$

$$a^{(2)} = f(z^{(2)}) \quad (2)$$

$$z^{(3)} = a^{(2)}W^{(2)} \quad (3)$$

Multi-Neuron Networks :: forward propagation



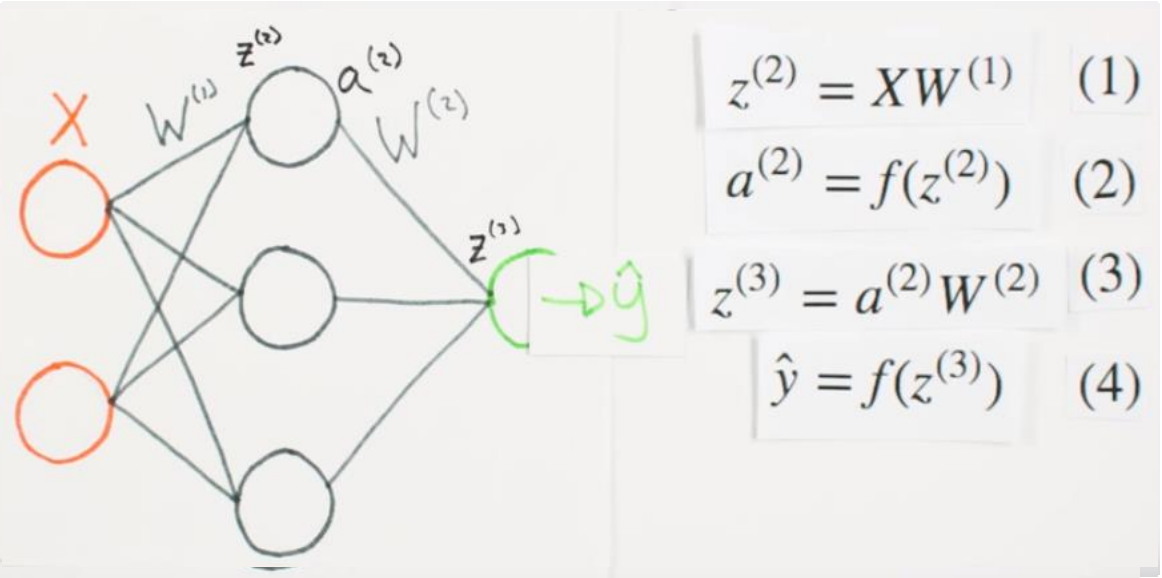
$$z^{(2)} = XW^{(1)} \quad (1)$$

$$a^{(2)} = f(z^{(2)}) \quad (2)$$

$$z^{(3)} = a^{(2)}W^{(2)} \quad (3)$$

$$\hat{y} = f(z^{(3)}) \quad (4)$$

Multi-Neuron Networks :: forward propagation



$$z^{(2)} = XW^{(1)} \quad (1)$$

$$a^{(2)} = f(z^{(2)}) \quad (2)$$

$$z^{(3)} = a^{(2)}W^{(2)} \quad (3)$$

$$\hat{y} = f(z^{(3)}) \quad (4)$$

Resources

- https://playground.tensorflow.org/
- https://betterexplained.com/articles/derivatives-product-power-chain/