Artificial Neural Networks

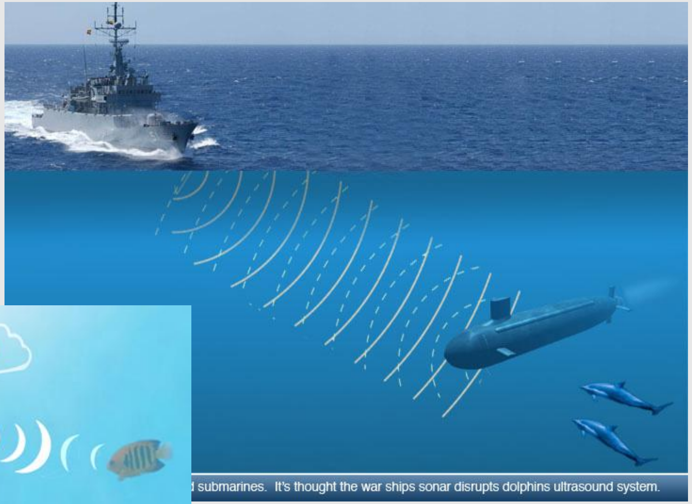
Machine Learning

Prof. Sandra Avila

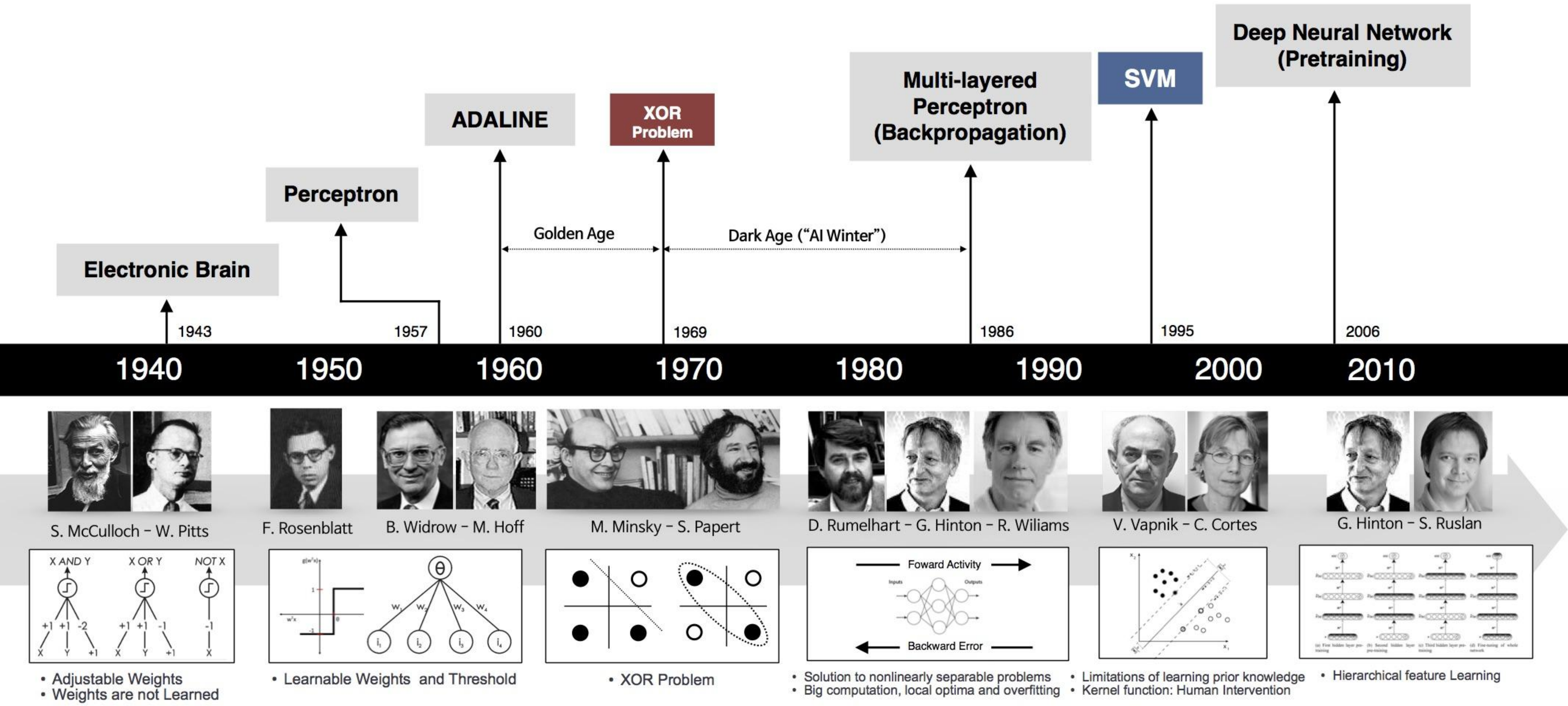
Institute of Computing (IC/Unicamp)

MC886, September 11, 2019

Many inventions were inspired by Nature ...

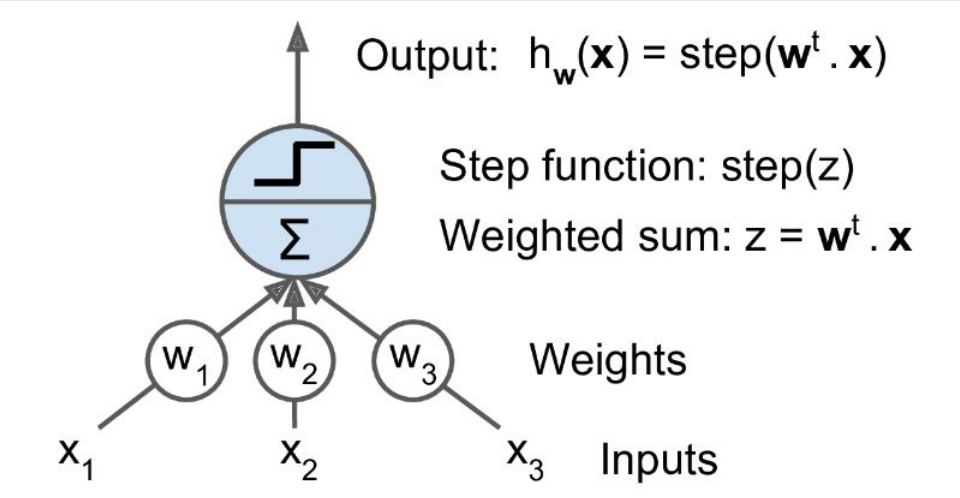


It seems logical to look at the brain's architecture for inspiration on how to build an intelligent machine.

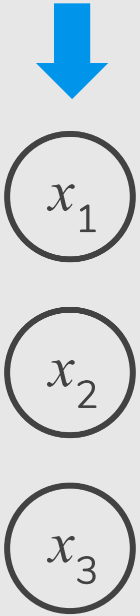

Electronic Brain
Perceptron
ADALINE
XOR Problem
Multi-layered Perceptron (Backpropagation)
SVM
Deep Neural Network (Pretraining)
Golden Age
Dark Age ("AI Winter")
1943
1957
1960
1969
1986
1995
2006
1940
1950
1960
1970
1980
1990
2000
2010
S. McCulloch – W. Pitts
F. Rosenblatt
B. Widrow – M. Hoff
M. Minsky – S. Papert
D. Rumelhart – G. Hinton – R. Wiliams
V. Vapnik – C. Cortes
G. Hinton – S. Ruslan
X AND Y
X OR Y
NOT X
Foward Activity
Inputs
Outputs
Backward Error
• Adjustable Weights
• Weights are not Learned
• Learnable Weights and Threshold
• XOR Problem
• Solution to nonlinearly separable problems
• Big computation, local optima and overfitting
• Limitations of learning prior knowledge
• Kernel function: Human Intervention
• Hierarchical feature Learning

The Perceptron

Neuron Model: Logistic Unit

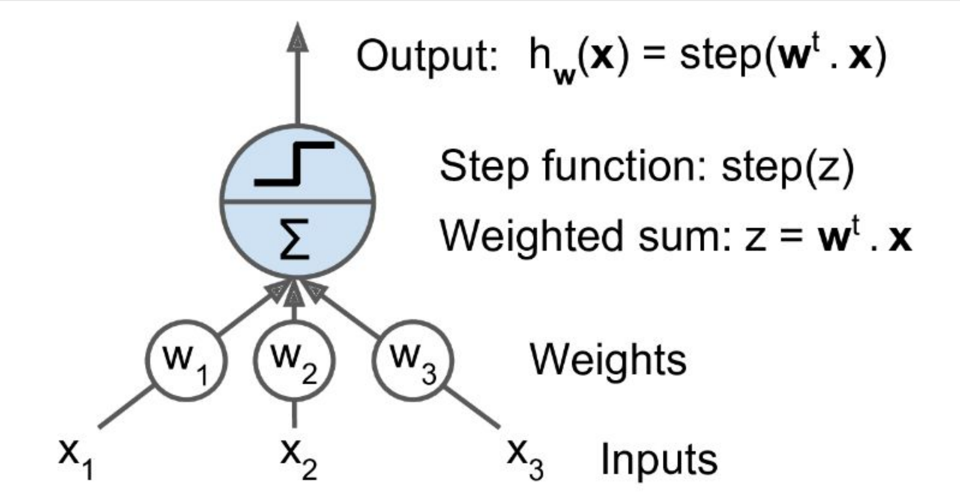


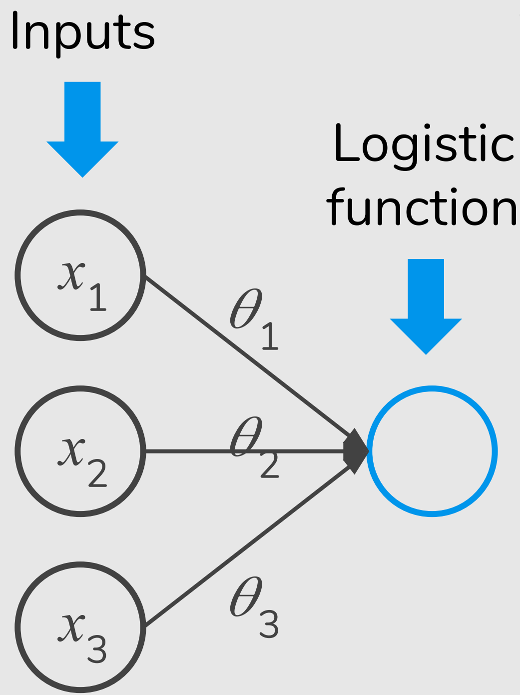
Inputs



Neuron Model: Logistic Unit

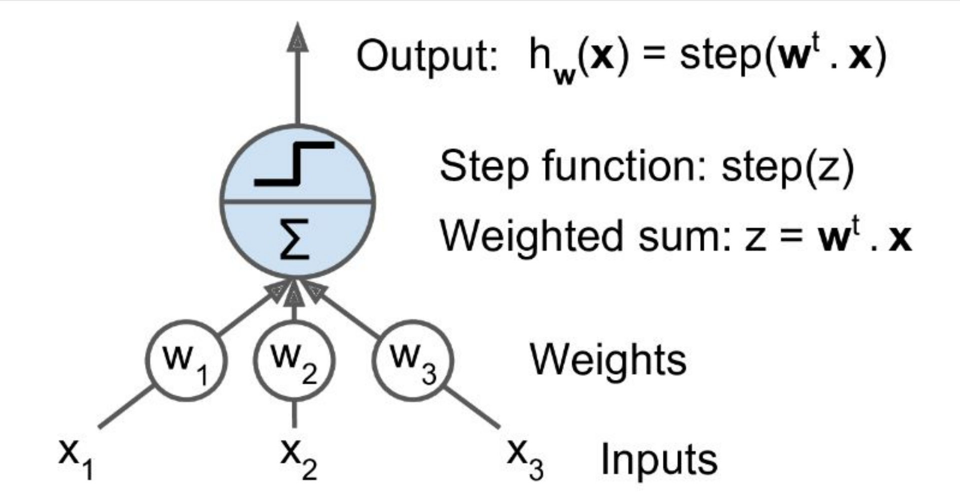
$$\frac{1}{1+e^{-\theta^T x}}$$

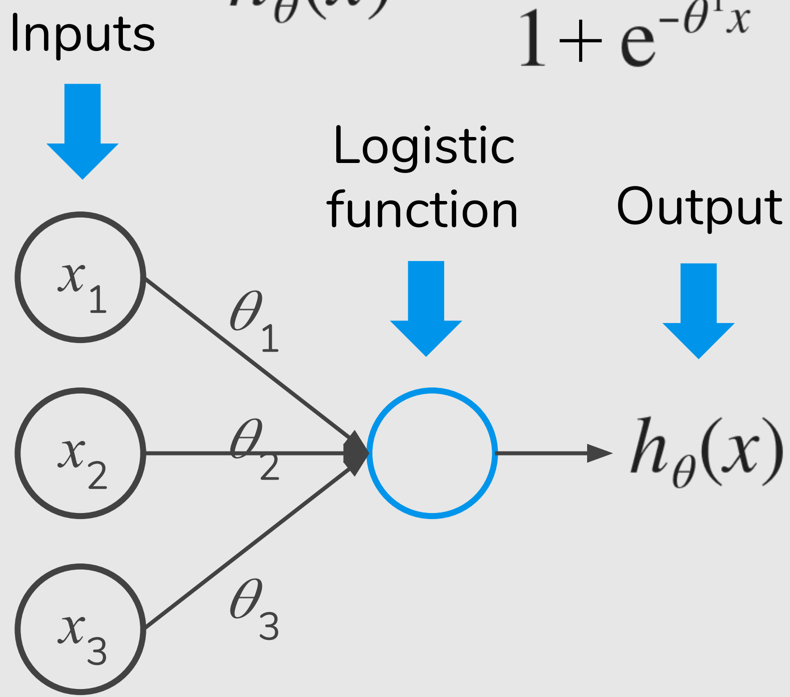




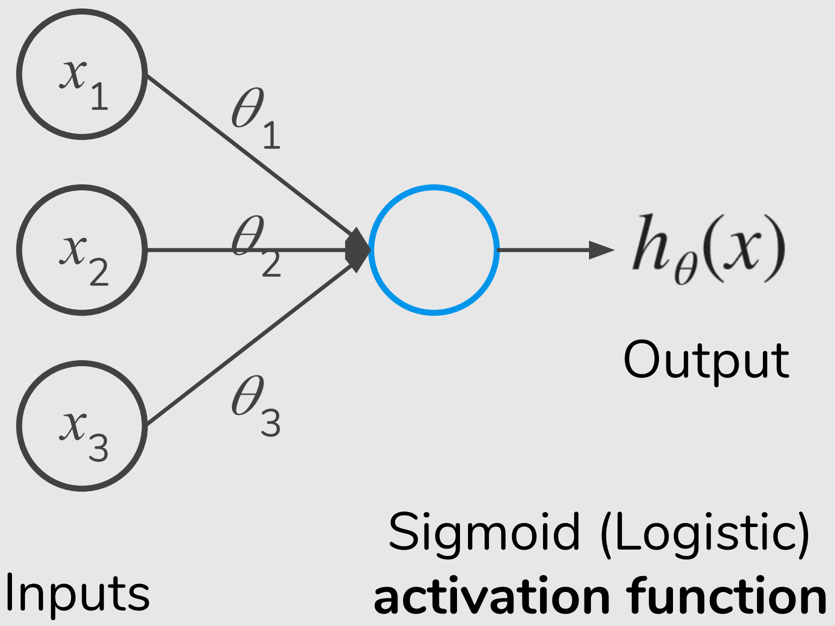
Neuron Model: Logistic Unit

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$





Neuron Model: Logistic Unit

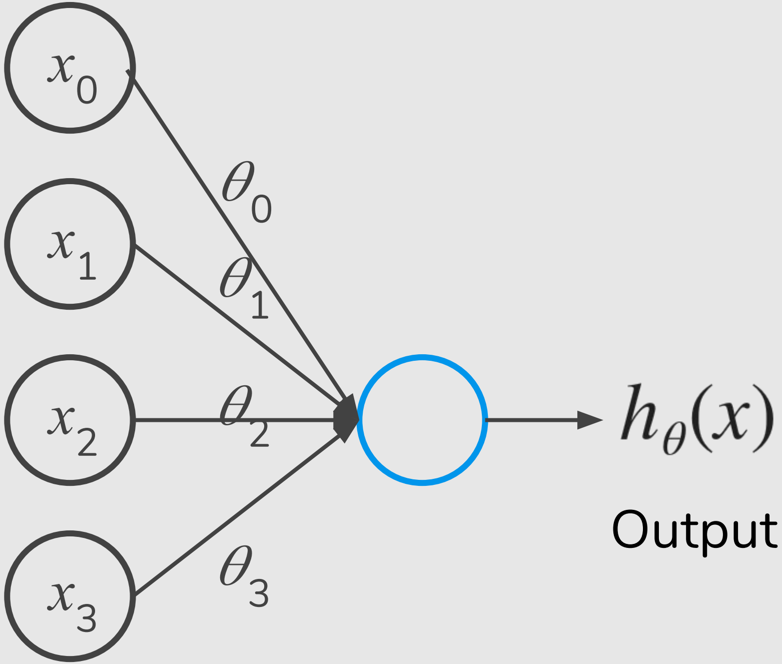


weights

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

Neuron Model: Logistic Unit



Inputs

Sigmoid (Logistic)
activation function

weights

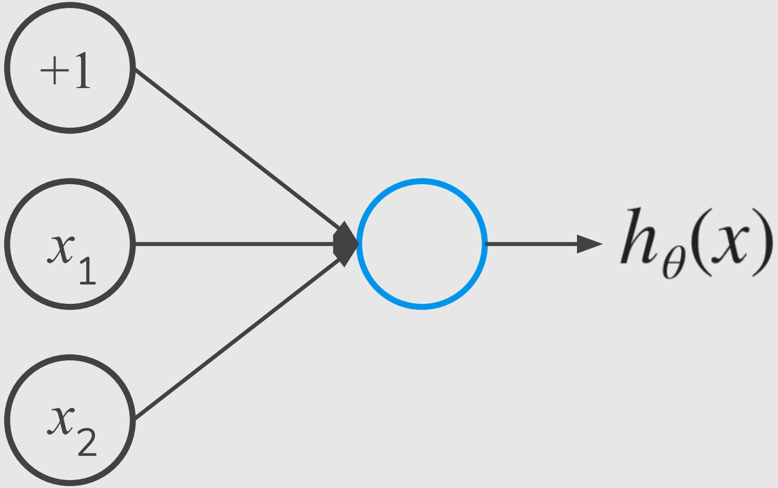
$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

Examples

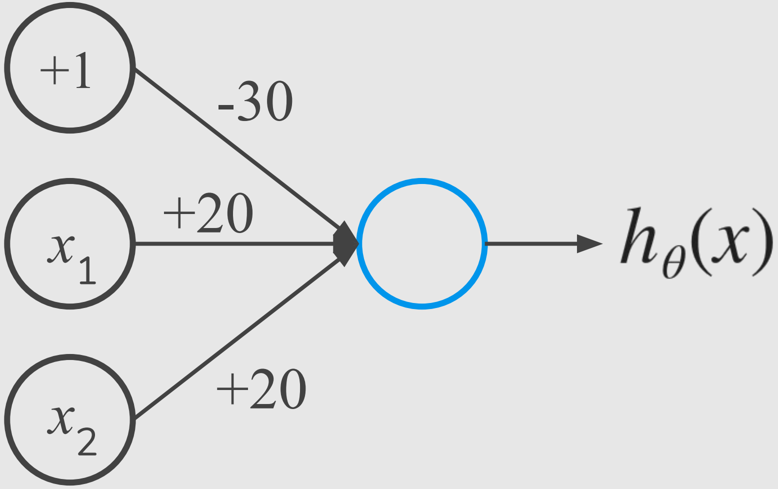
Simple Example: AND

$x_1, x_2 \in \{0,1\}$　　$y = x_1 \text{ AND } x_2$



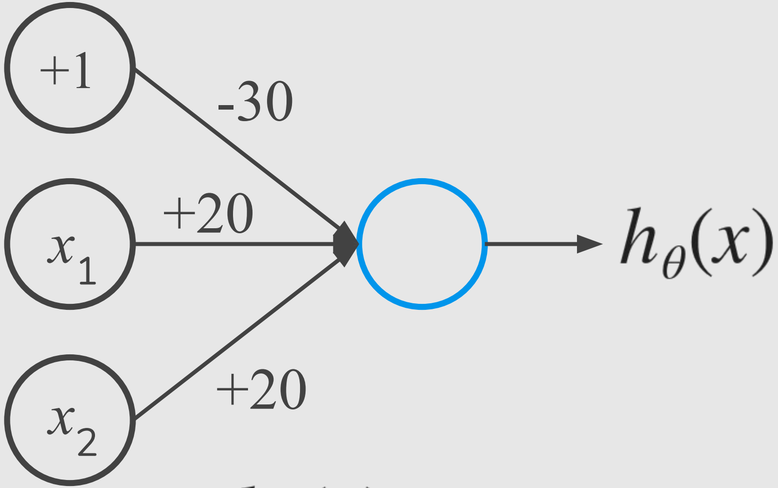
Simple Example: AND

$x_1, x_2 \in \{0,1\}$ $\quad$ $y = x_1 \text{ AND } x_2$



Simple Example: AND

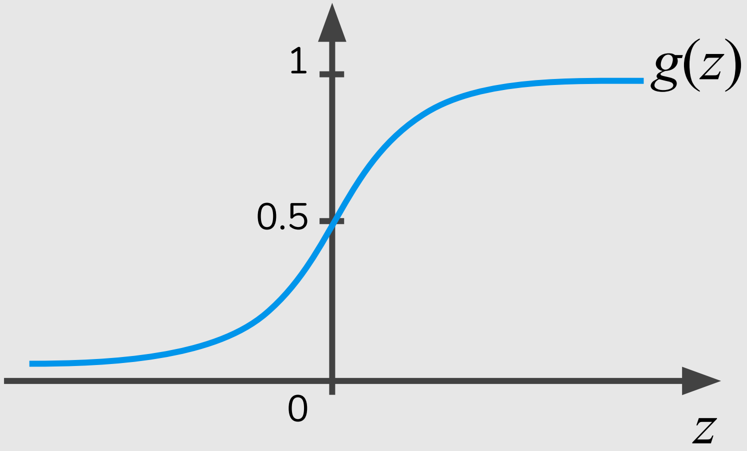
$x_1, x_2 \in \{0,1\}$ $\quad y = x_1 \text{ AND } x_2$

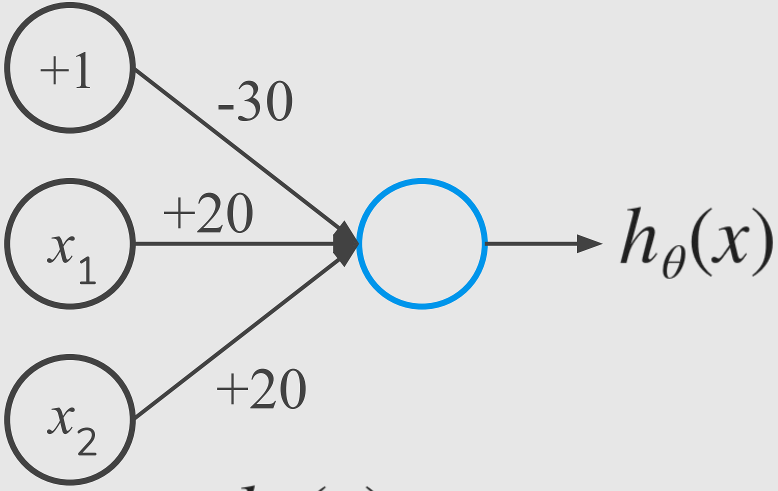


$$h_\theta(x) = g(-30 + 20x_1 + 20x_2)$$

Simple Example: AND

$x_1, x_2 \in \{0,1\}$ $\quad$ $y = x_1 \text{ AND } x_2$



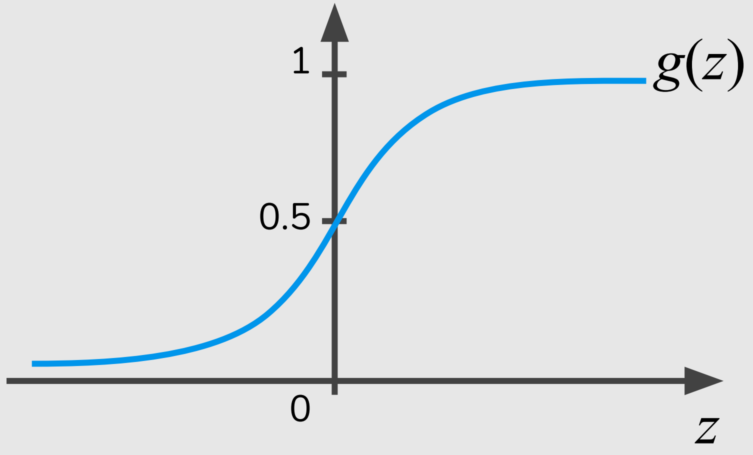


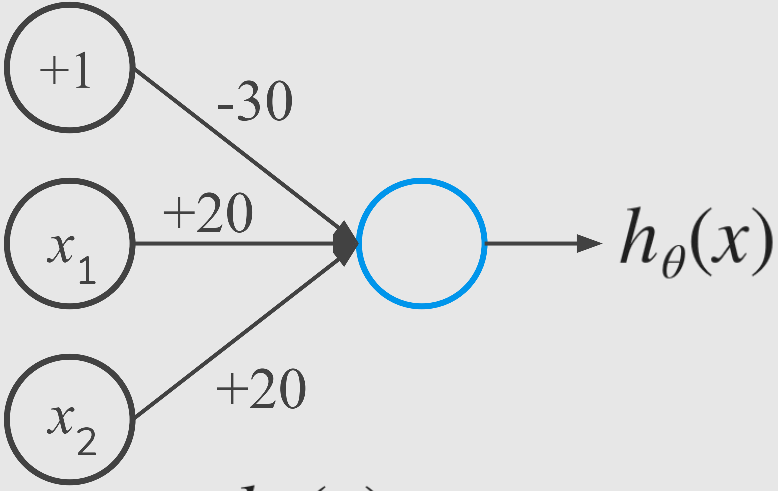
$$h_\theta(x) = g(-30 + 20x_1 + 20x_2)$$

x_1	x_2	$h_\theta(x)$
0	0	
0	1	
1	0	
1	1	

Simple Example: AND

$x_1, x_2 \in \{0,1\}$ $\quad y = x_1 \text{ AND } x_2$



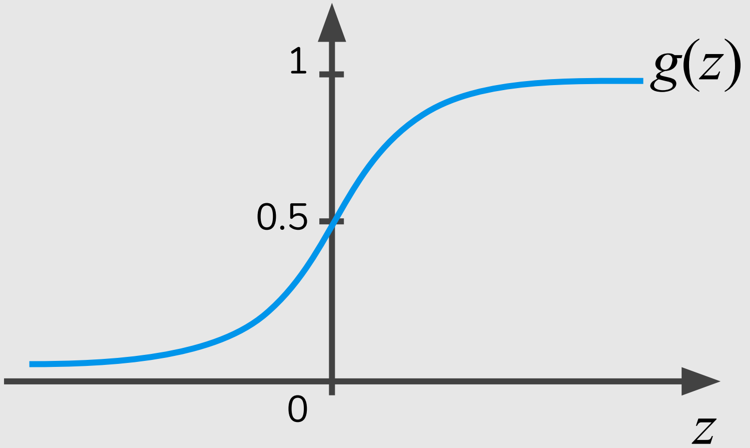


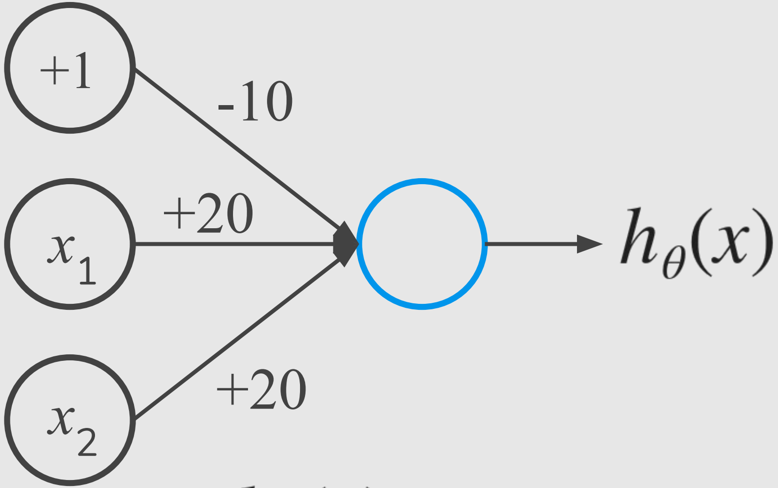
x_1	x_2	$h_\theta(x)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

$$h_\theta(x) = g(-30 + 20x_1 + 20x_2)$$

Simple Example: OR

$x_1, x_2 \in \{0,1\}$ $\quad y = x_1 \text{ OR } x_2$





x_1	x_2	$h_\theta(x)$
0	0	$g(-10) \approx 0$
0	1	$g(10) \approx 1$
1	0	$g(10) \approx 1$
1	1	$g(30) \approx 1$

$$h_\theta(x) = g(-10 + 20x_1 + 20x_2)$$

What does an artificial neuron do?

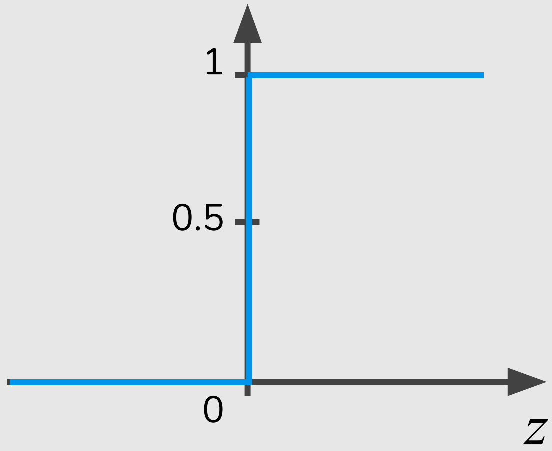
It calculates a “weighted sum” of its input, adds a bias and then decides whether it should be “fired” or not.

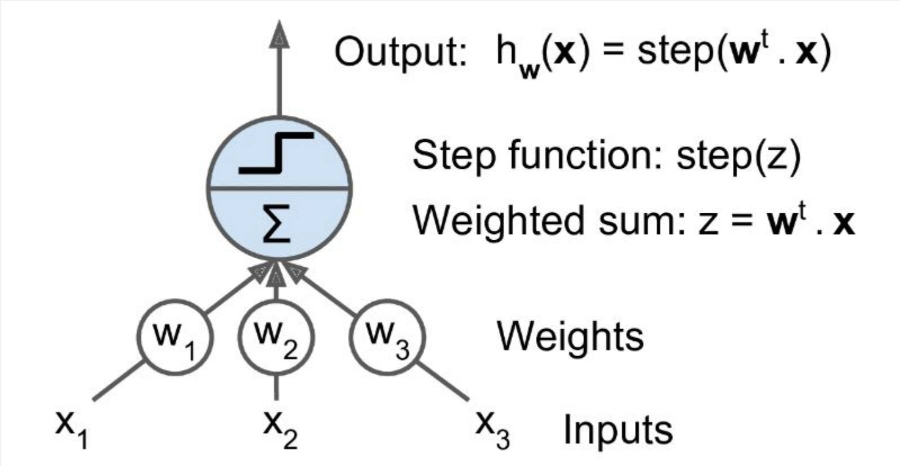
How do we decide whether the neuron should fire or not?

We decided to add “activation functions” for this purpose.

Step Function

Its output is **1 (activated)** when value > 0 (threshold) and outputs a **0 (not activated)** otherwise.



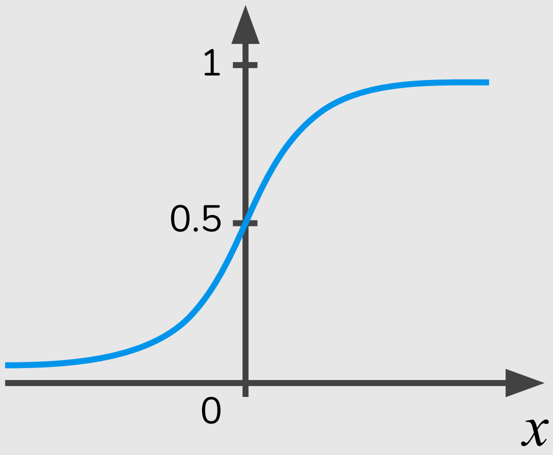


Step Function: **Problem?**

- Binary classifier ("yes" or "no", activate or not activate). A Step function could do that for you!
- Multi classifier (class1, class2, class3, etc). What will happen if more than 1 neuron is "activated"?

Sigmoid Function

- The output of the activation function is always going to be in range **(0,1)**.
- It is nonlinear in nature.
- Combinations of this function are also nonlinear! Great!!



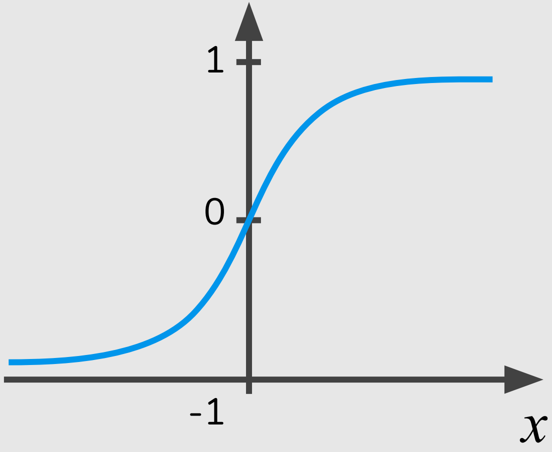
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Sigmoid Function: **Problem?**

- Towards either end of the sigmoid function, the $\sigma(x)$ values tend to respond very less to changes in x.
- The problem of "**vanishing gradients**".
  - Cannot make significant change because of the extremely small value.

Tanh Function

- The output of the activation function is always going to be in range **(-1,1)**.
- It is nonlinear in nature.
- Combinations of this function are also nonlinear! Great!!



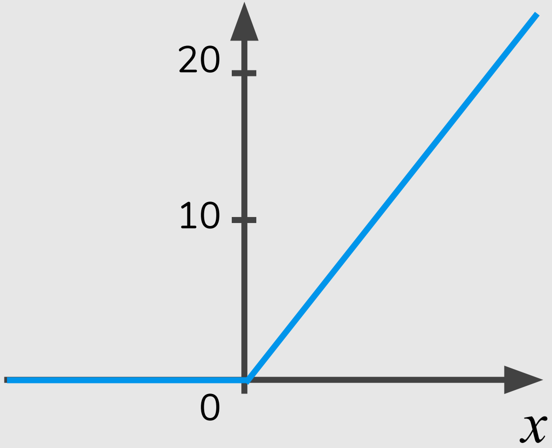
$$tanh(x) = \frac{2}{1+e^{-2x}} - 1$$

Tanh Function: **Problem?**

- Like sigmoid, tanh also has the vanishing gradient problem.

ReLU (Rectified Linear Unit) Function

- It gives an output x if x is positive and 0 otherwise. The range is **[0, inf)**.
- It is nonlinear in nature. Combinations of this function are also nonlinear!
- Sparsity of the activation!



$$\mathrm{ReLU}(x) = max(0,x)$$

ReLU Function: **Problem?**

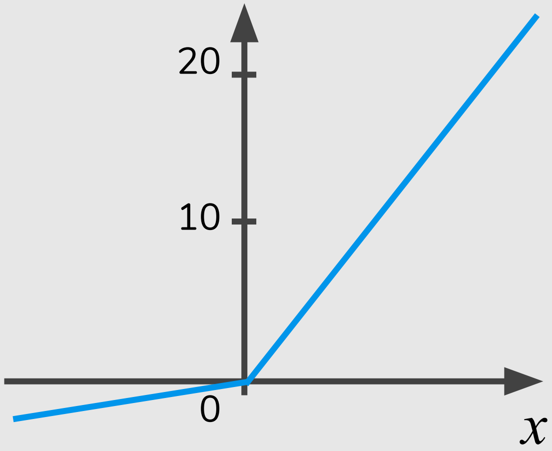
- Because of the horizontal line in ReLU(for negative x), the gradient can go towards 0.

- “Dying ReLU problem”: several neurons can just die and not respond making a substantial part of the network passive.

Leaky ReLU Function

- It gives an output x if x is positive and 0 otherwise. The range is **[0, inf)**.

- (Leaky) ReLU is less computationally expensive than *tanh* and *sigmoid* because it involves simpler mathematical operations.



$$\text{Leaky ReLU}(x) = \\ = \begin{cases} x \text{ if } x > 0 \\ 0.01x \text{ otherwise} \end{cases}$$

Ok! Which One Do We Use?

- If you don't know the nature of the function you are trying to learn, start with ReLU.
- You can use your own custom functions too!

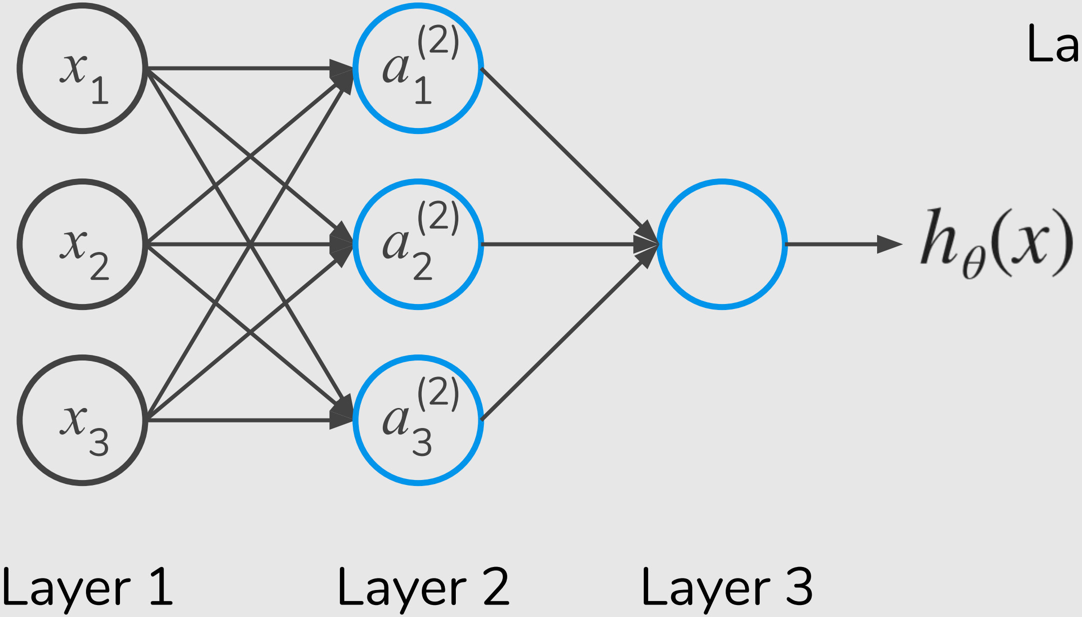
Neural Network

Neural Network

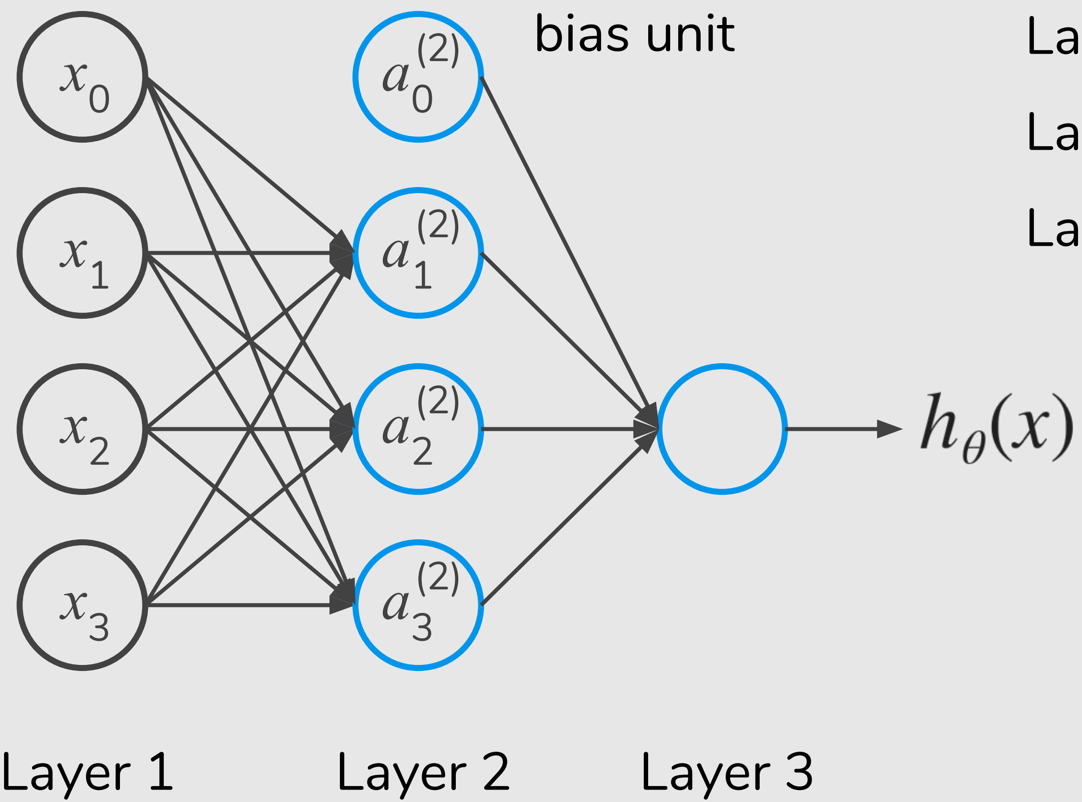
Layer 1 = Input layer

Layer 2 = Hidden layer

Layer 3 = Output layer



Neural Network



Layer 1 = Input layer

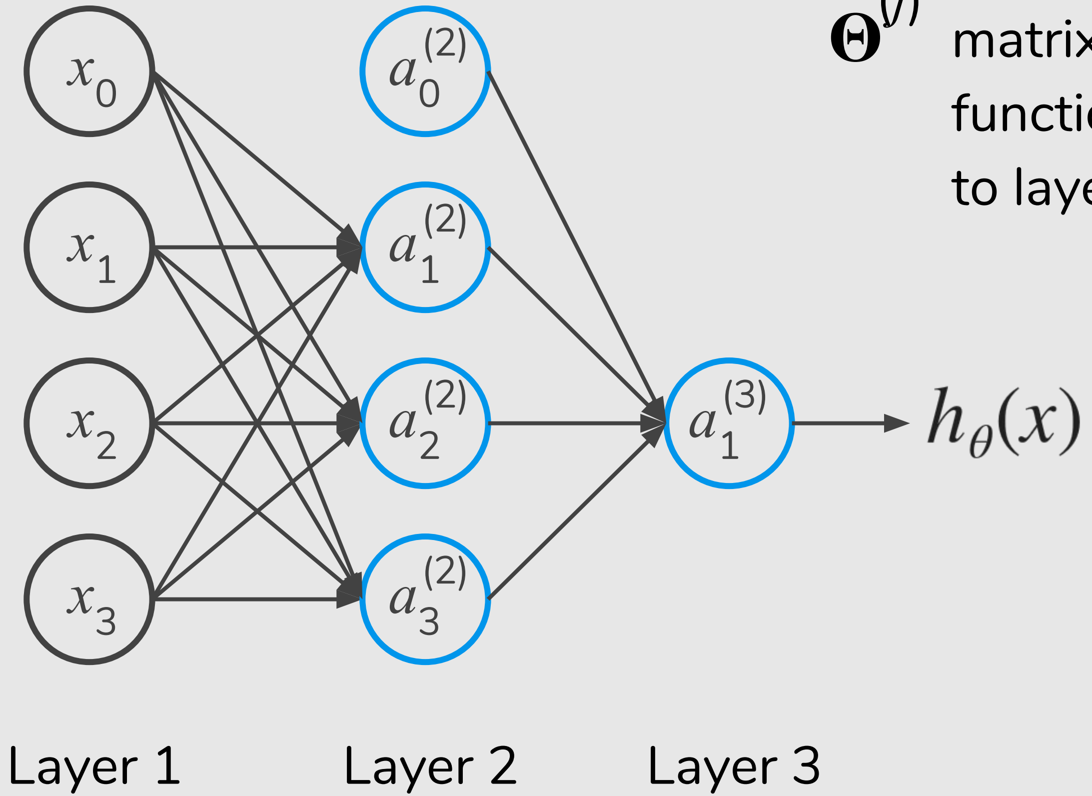
Layer 2 = Hidden layer

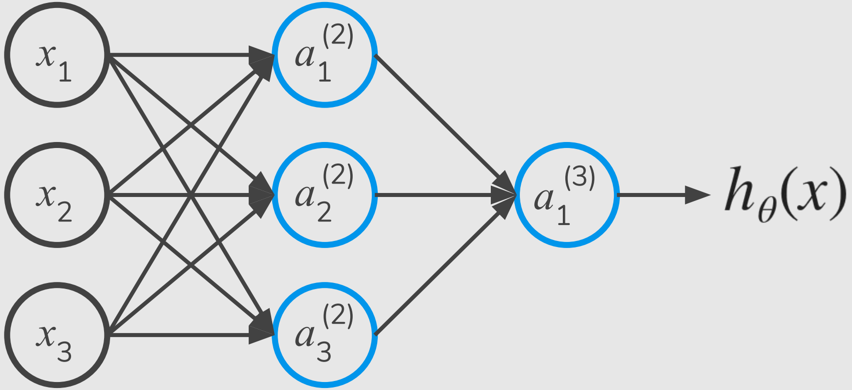
Layer 3 = Output layer

Neural Network

$a_i^{(j)}$ "activation" of unit i in layer j

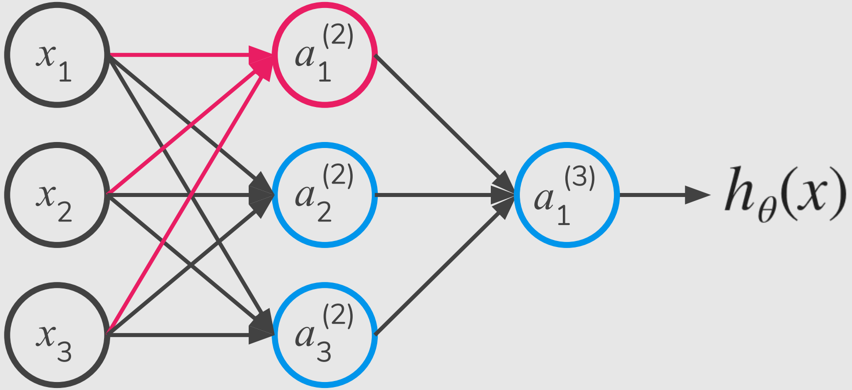
$\mathbf{\Theta}^{(j)}$ matrix of weights controlling function mapping from layer j to layer $j+1$





$a_i^{(j)}$ "activation" of unit i in layer j

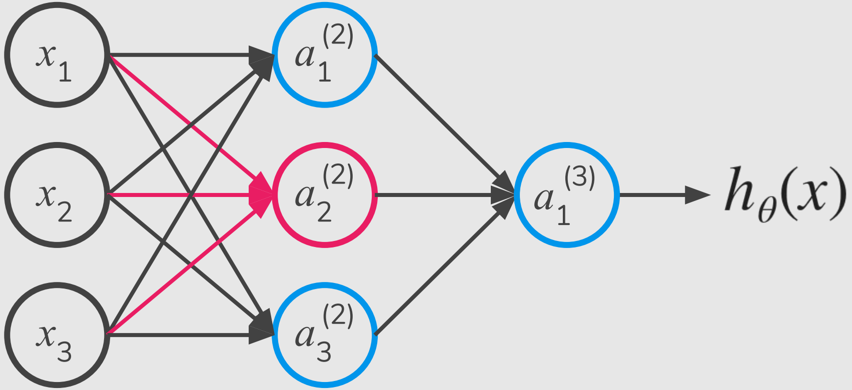
$\mathbf{\Theta}^{(j)}$ matrix of weights controlling function mapping from layer j to layer $j+1$



$a_i^{(j)}$ "activation" of unit i in layer j

$\mathbf{\Theta}^{(j)}$ matrix of weights controlling function mapping from layer j to layer $j+1$

$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$

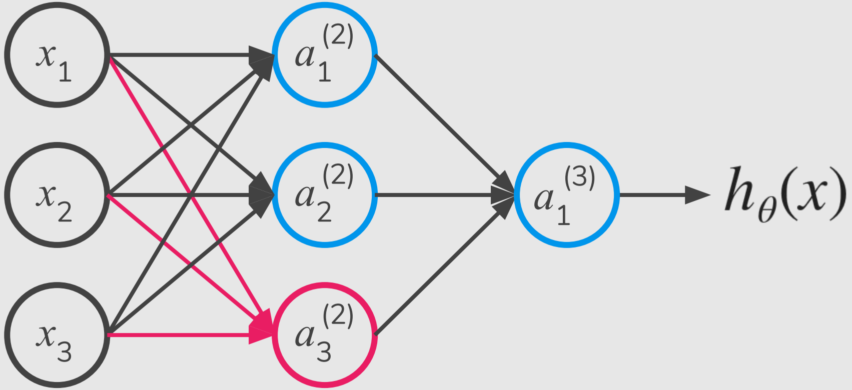


$a_i^{(j)}$ "activation" of unit i in layer j

$\Theta^{(j)}$ matrix of weights controlling function mapping from layer j to layer $j+1$

$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$

$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$



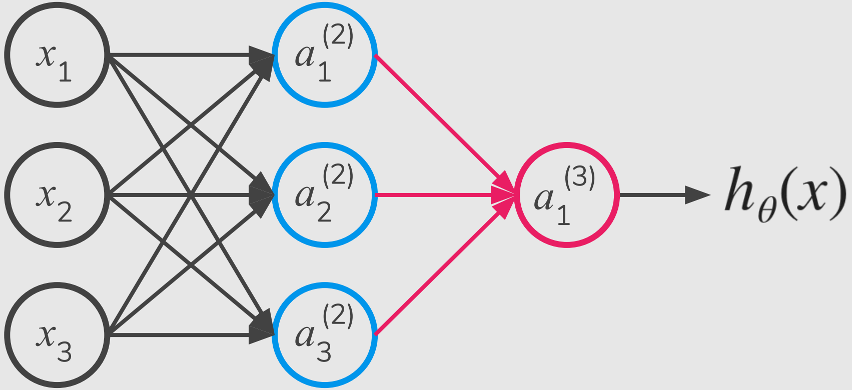
$a_i^{(j)}$ "activation" of unit i in layer j

$\mathbf{\Theta}^{(j)}$ matrix of weights controlling function mapping from layer j to layer $j+1$

$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$

$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$

$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$



$a_i^{(j)}$ "activation" of unit i in layer j

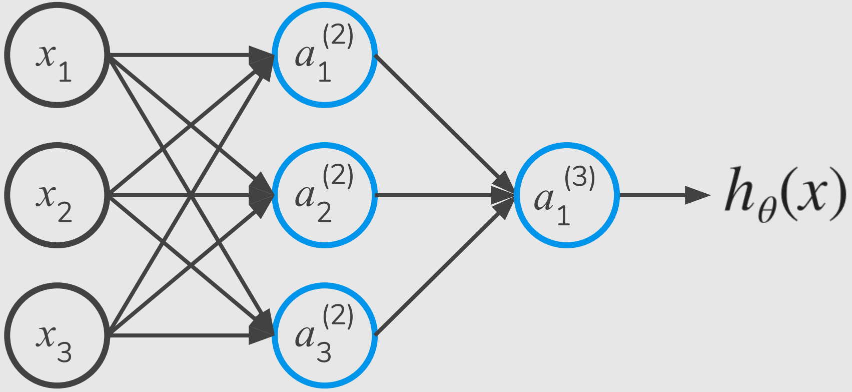
$\mathbf{\Theta}^{(j)}$ matrix of weights controlling function mapping from layer j to layer $j+1$

$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$

$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$

$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$

$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$



$a_i^{(j)}$ "activation" of unit i in layer j

$\Theta^{(j)}$ matrix of weights controlling function mapping from layer j to layer $j+1$

Feedforward Neural Network (forward propagating)

$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$

Training a Neural Network

Training a Neural Network

- The first thing we need to do is to select an architecture.
- **Input units:** dimensionality of the problem (features x)

Training a Neural Network

- The first thing we need to do is to select an architecture.
- **Input units:** dimensionality of the problem (features x)
- **Output units:** Number of classes

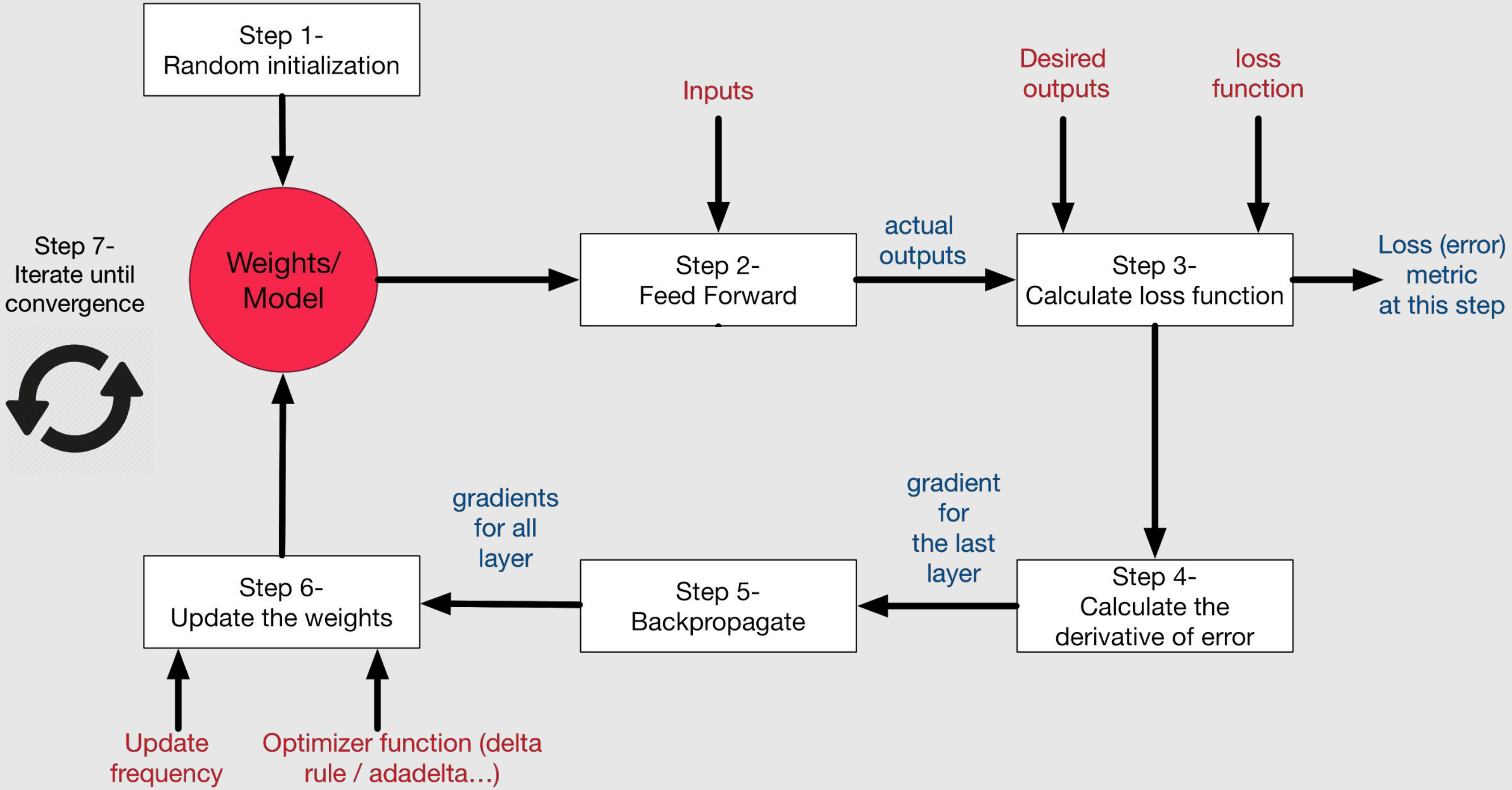
Training a Neural Network

- The first thing we need to do is to select an architecture.
- **Input units:** dimensionality of the problem (features x)
- **Output units:** Number of classes
- **Hidden units** (per layer)

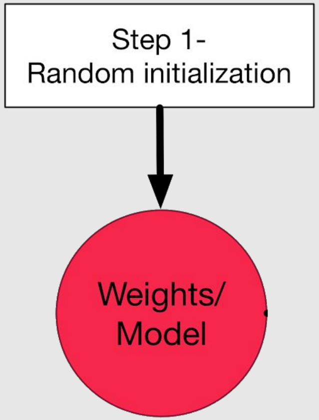
Training a Neural Network

- **Hidden units** (per layer):
    - Usually, the more the better
    - Good start: a number close to the number of input
    - Default: 1 hidden layer. If you have >1 hidden layer, then it is interesting that you have the **same number of units in every hidden layer**.

Training a Neural Network

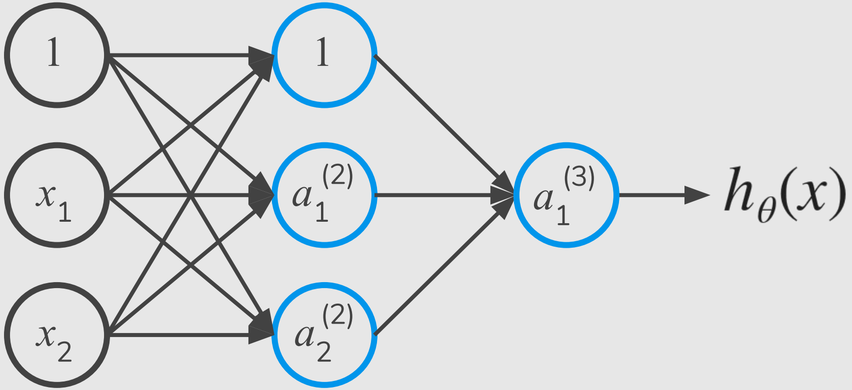


Training a Neural Network



Zero Initialization

Symmetric Weights



$$a_1^{(2)} = a_2^{(2)}$$

After each update, parameters corresponding to inputs going into each of two hidden units are identical.

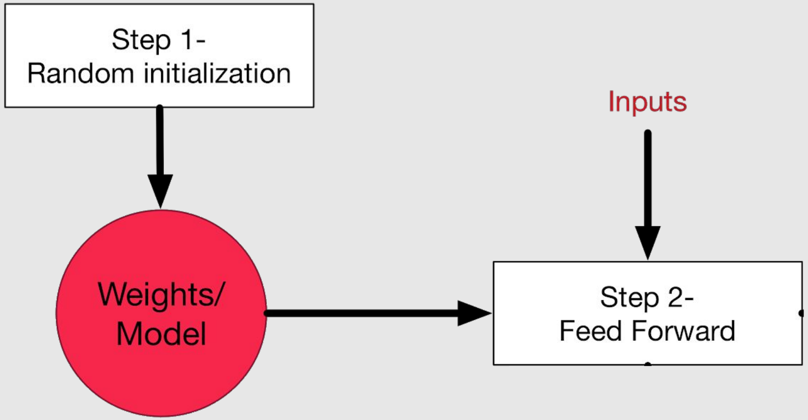
Symmetric Breaking

- We must initialize $\mathbf{\Theta}$ to a **random value** in $[-\varepsilon, \varepsilon]$ (i.e. $[-\varepsilon \leq \mathbf{\Theta} \leq \varepsilon]$)

- If the dimensions of `Theta1` is 3x4, `Theta2` is 3x4 and `Theta3` is 1x4.

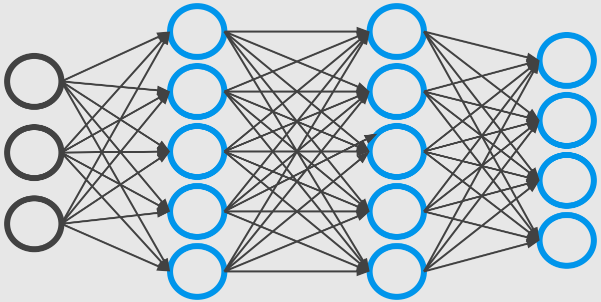
```
Theta1 = random(3,4) * (2 * EPSILON) - EPSILON;
Theta2 = random(3,4) * (2 * EPSILON) - EPSILON;
Theta3 = random(1,4) * (2 * EPSILON) - EPSILON;
```

Training a Neural Network



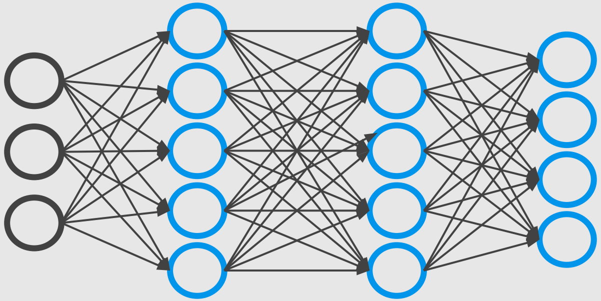
Forward Propagation

$$
\begin{aligned}
a^{(1)} &= x \\
z^{(2)} &= \Theta^{(1)} a^{(1)} \\
a^{(2)} &= g(z^{(2)}) \quad \text{(add } a_0^{(2)}\text{)} \\
z^{(3)} &= \Theta^{(2)} a^{(2)} \\
a^{(3)} &= g(z^{(3)}) \quad \text{(add } a_0^{(3)}\text{)} \\
z^{(4)} &= \Theta^{(3)} a^{(3)} \\
a^{(4)} &= h_\Theta(x) = g(z^{(4)})
\end{aligned}
$$



Forward Propagation

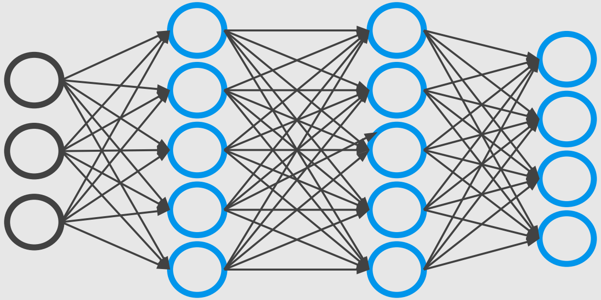
Given one training example (x, y):



Forward Propagation

Given one training example (x, y):

$$a^{(1)} = x$$

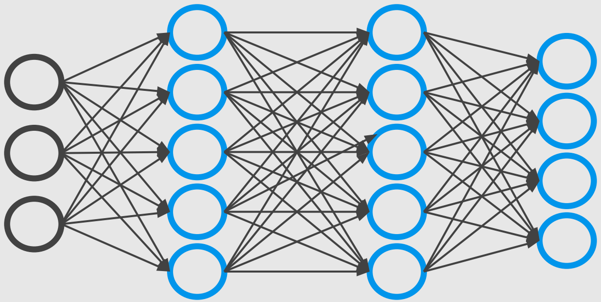


Forward Propagation

Given one training example (x, y):

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$



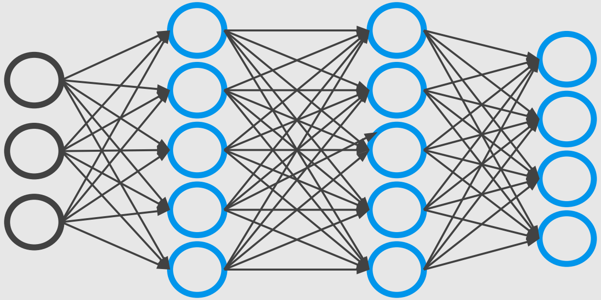
Forward Propagation

Given one training example (x, y):

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$



Forward Propagation

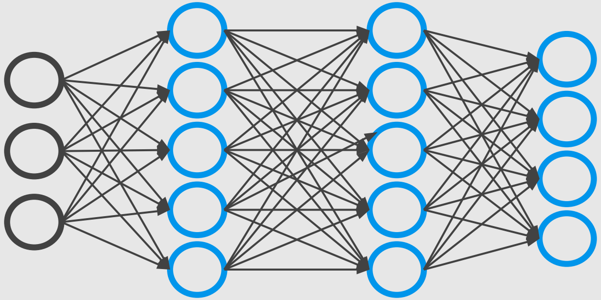
Given one training example (x, y):

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$

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



Forward Propagation

Given one training example (x, y):

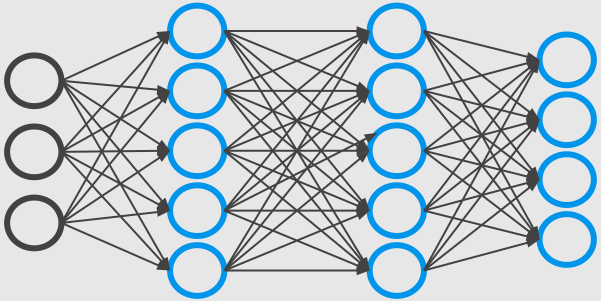
$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$

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

$$a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$



Forward Propagation

Given one training example (x, y):

$$a^{(1)} = x$$

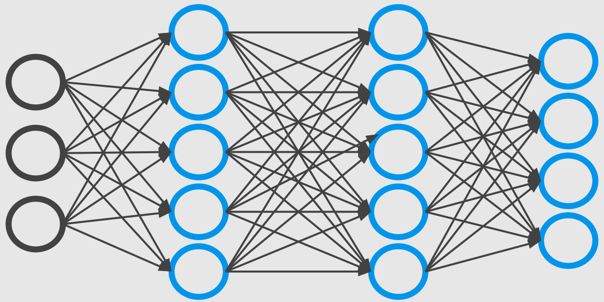
$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$

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

$$a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$

$$z^{(4)} = \Theta^{(3)} a^{(3)}$$



Forward Propagation

Given one training example (x, y):

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

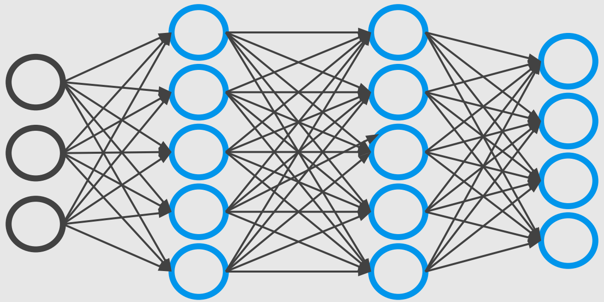
$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$

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

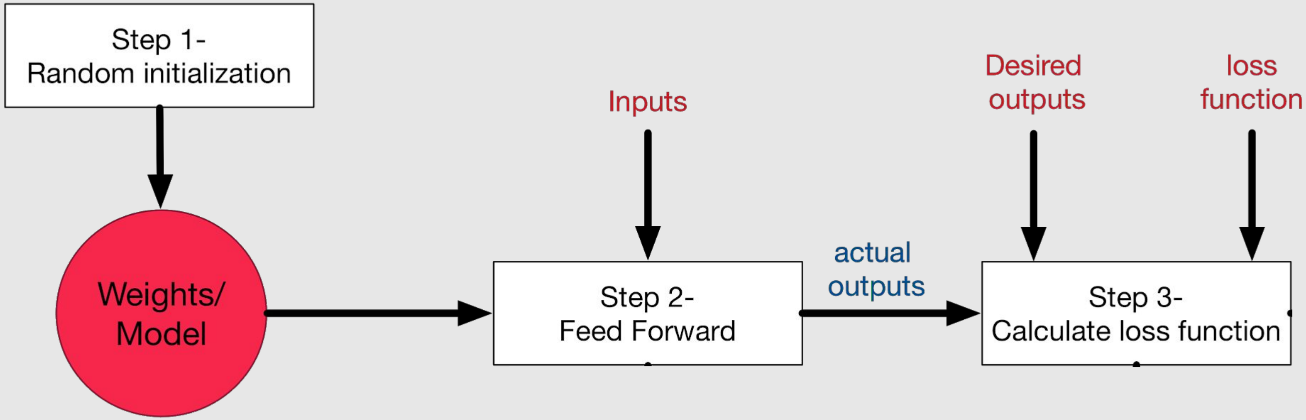
$$a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$

$$z^{(4)} = \Theta^{(3)} a^{(3)}$$

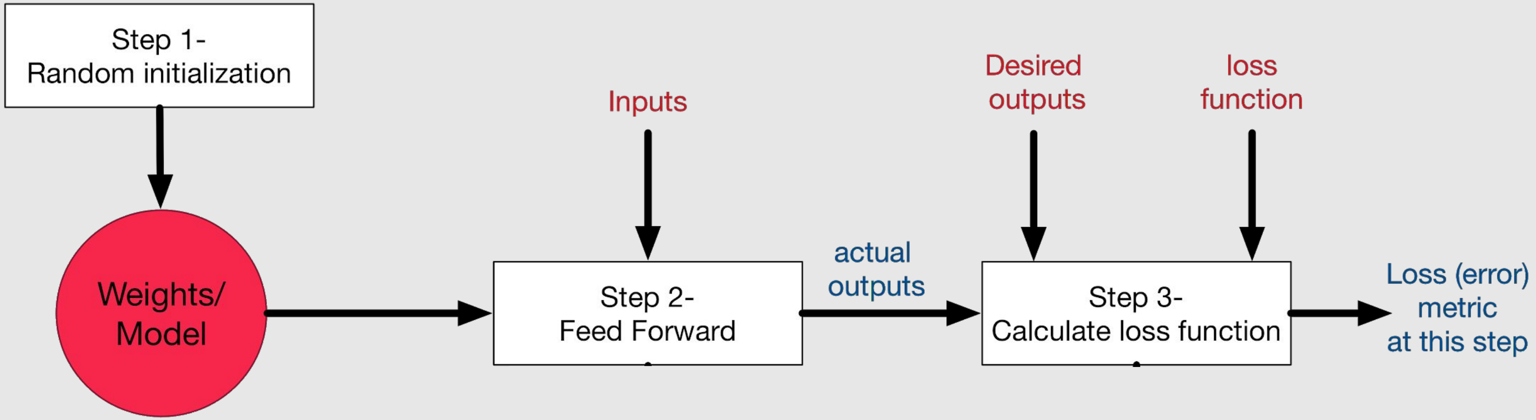
$$a^{(4)} = h_\Theta(x) = g(z^{(4)})$$



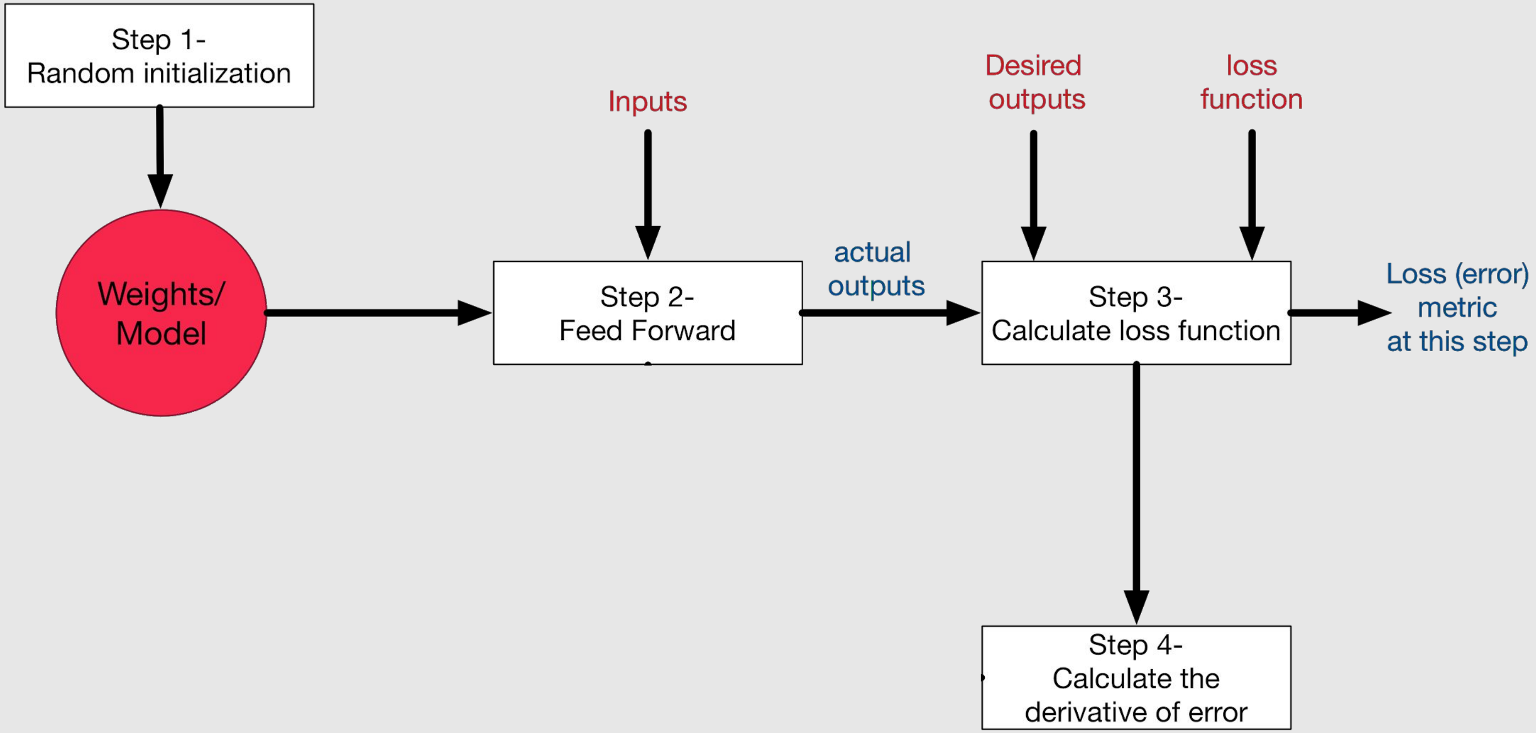
Training a Neural Network



Training a Neural Network

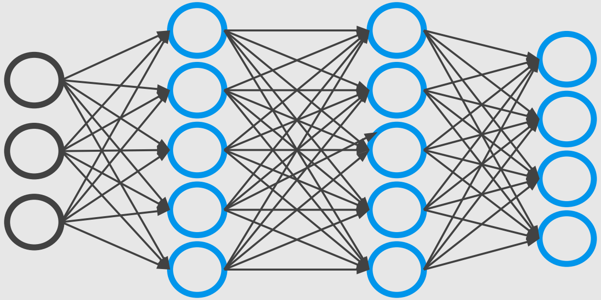


Training a Neural Network



Gradient Computation: Backpropagation Algorithm

Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l.

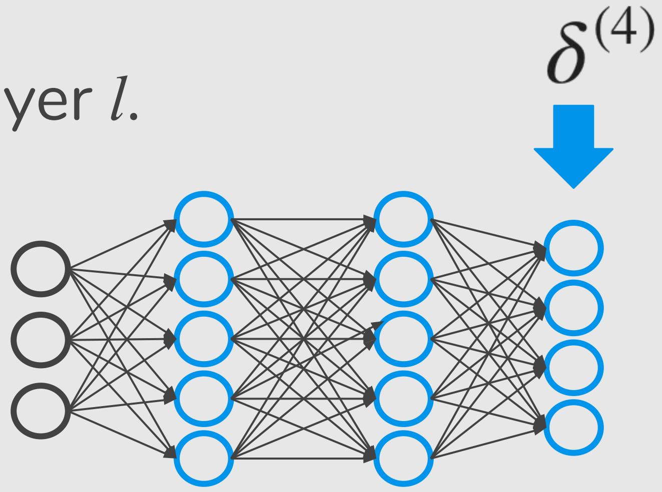


Gradient Computation: Backpropagation Algorithm

Intuition: $\delta_j^{(l)} =$ "error" of node j in layer l.

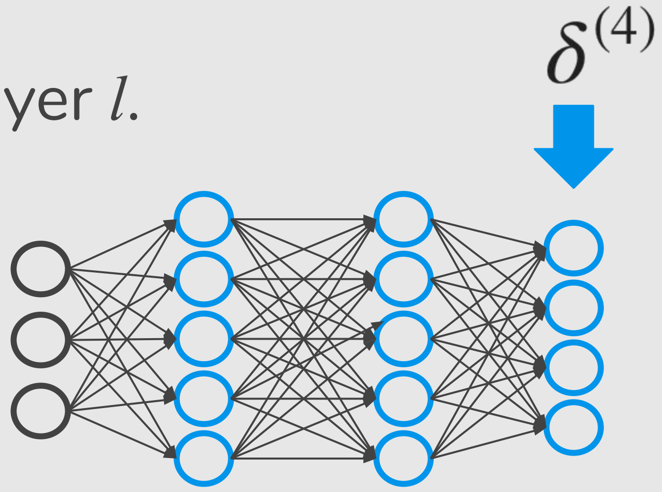
For each output unit (layer $L = 4$)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$



Gradient Computation: Backpropagation Algorithm

Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l.



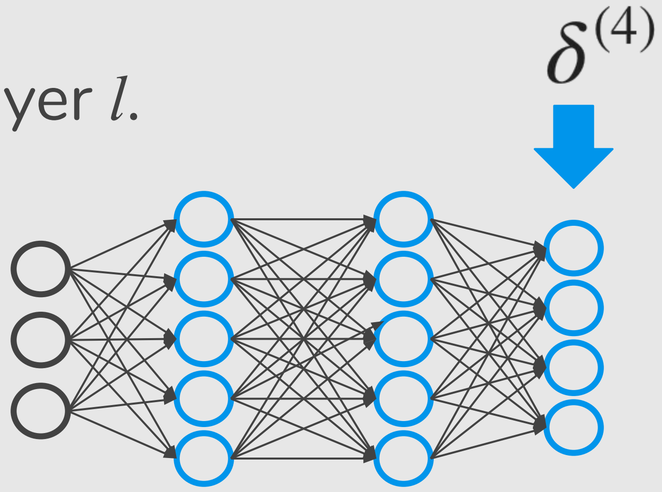
For each output unit (layer $L = 4$)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

$$(h_\Theta(x))_j$$

Gradient Computation: Backpropagation Algorithm

Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l.



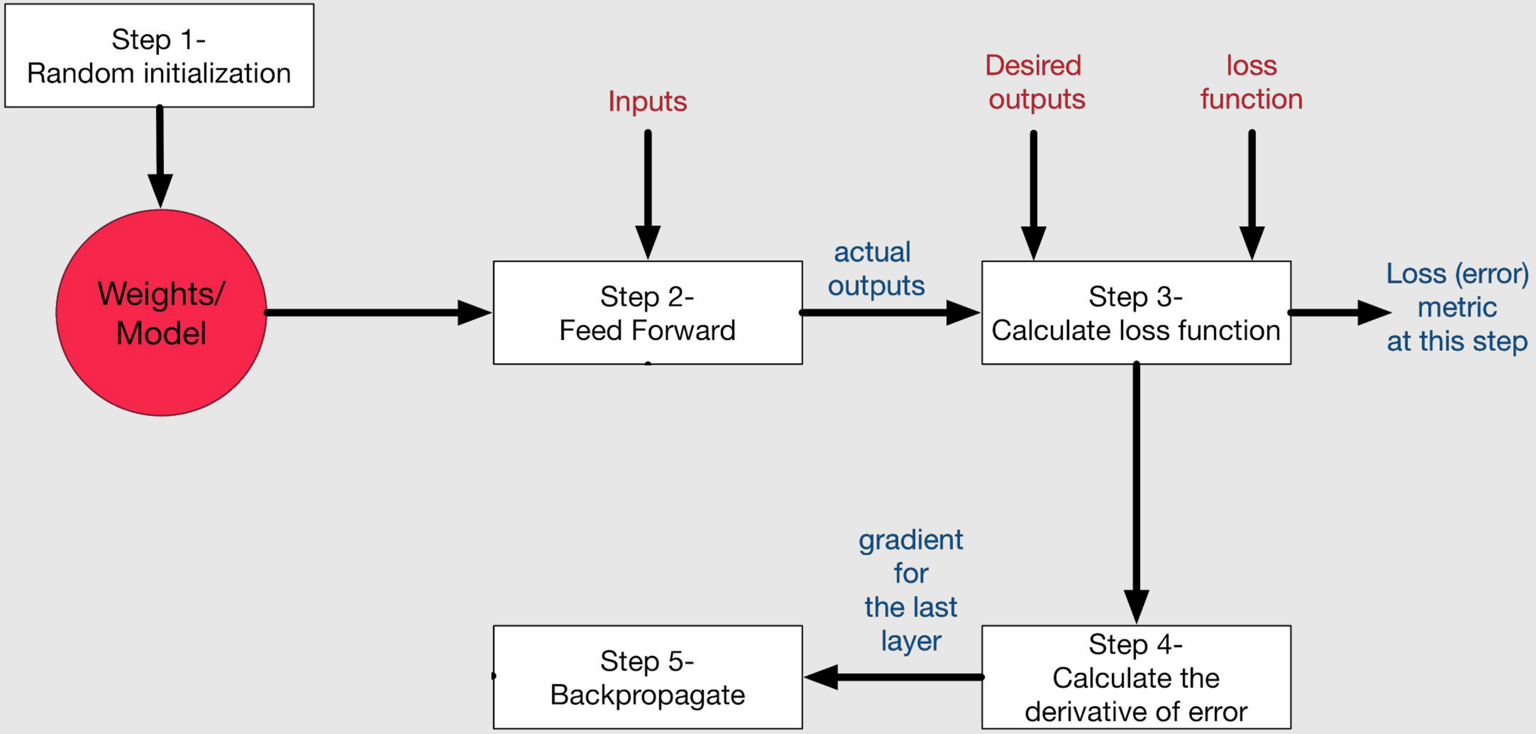
For each output unit (layer L = 4)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

Vectorizing it, we have:

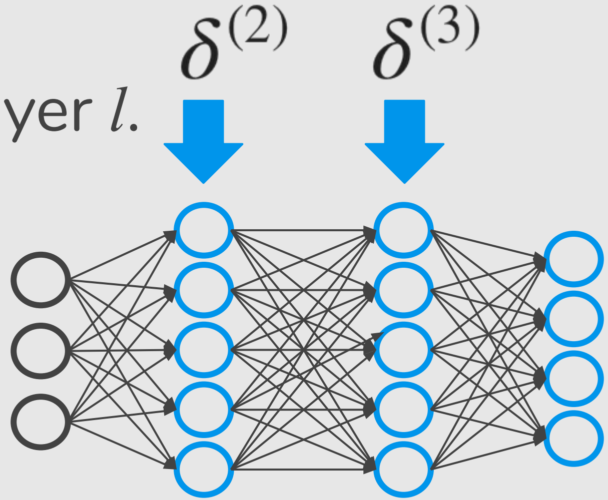
$$\delta^{(4)} = a^{(4)} - y$$

Training a Neural Network



Gradient Computation: Backpropagation Algorithm

Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l.



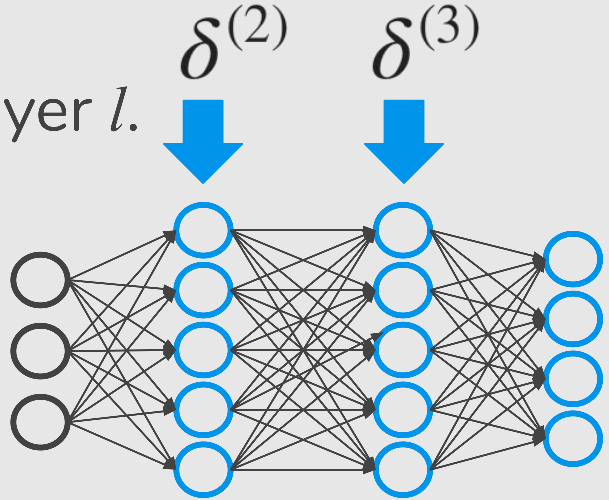
For each output unit (layer $L = 4$)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

For each hidden unit

Gradient Computation: Backpropagation Algorithm

Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l.



For each output unit (layer $L = 4$)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

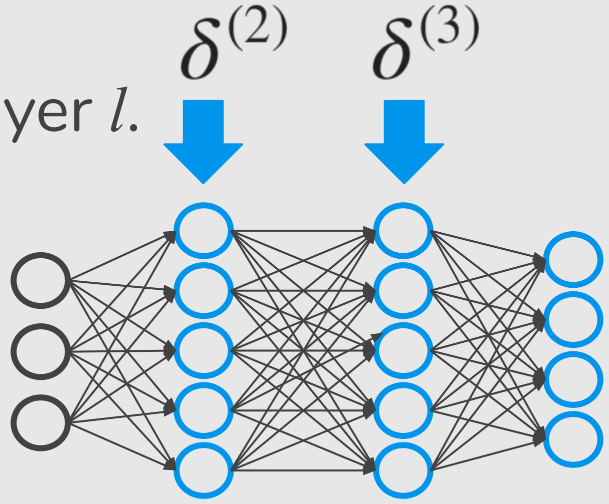
For each hidden unit

$$\delta^{(3)} = (\Theta^{(3)})^{T}\delta^{(4)}$$

$.*$ element-wise multiplication

Gradient Computation: Backpropagation Algorithm

Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l.



For each output unit (layer $L = 4$)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

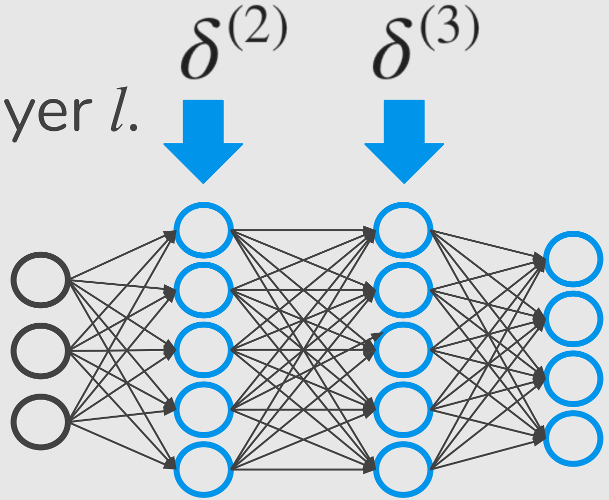
For each hidden unit

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$$

$.*$ element-wise multiplication

Gradient Computation: Backpropagation Algorithm

Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l.



For each output unit (layer $L = 4$)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

For each hidden unit

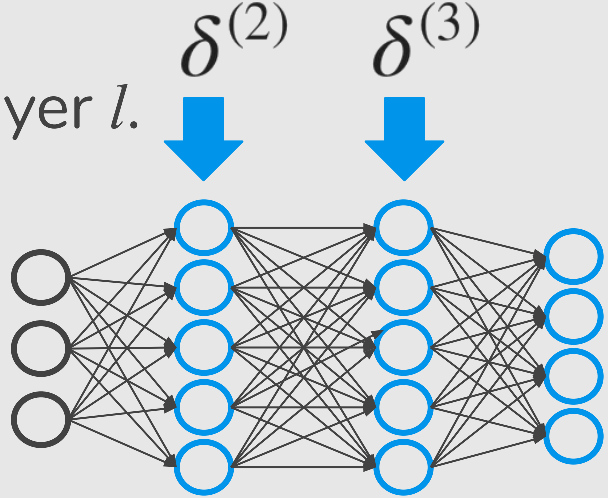
$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$$

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$

$.*$ element-wise multiplication

Gradient Computation: Backpropagation Algorithm

Intuition: $\delta_j^{(l)} =$ "error" of node j in layer l.



For each output unit (layer $L = 4$)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

For each hidden unit

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* \boxed{g'(z^{(3)})} \quad a^{(3)}(1 - a^{(3)})$$

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$

Derivative of Logistic Function

$$\boxed{g(z) = \frac{1}{1+e^{-z}}}$$

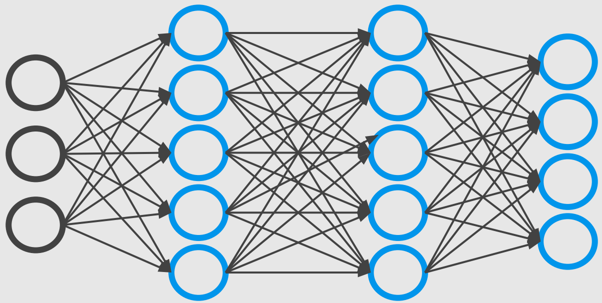
$$\begin{aligned}
g'(z) &= \frac{d}{dz}\,\frac{1}{1+e^{-z}} \\
&= \frac{0\cdot(1+e^{-z}) - 1\cdot(-e^{-z})}{(1+e^{-z})^2} \qquad \text{(quotient rule)} \\
&= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \left(\frac{1}{1+e^{-z}}\right)\left(1 - \frac{1}{1+e^{-z}}\right) \\
&= g(z)(1-g(z))
\end{aligned}$$

Gradient Computation: Backpropagation Algorithm

Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l.

For each output unit (layer $L = 4$)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$



For each hidden unit

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$$

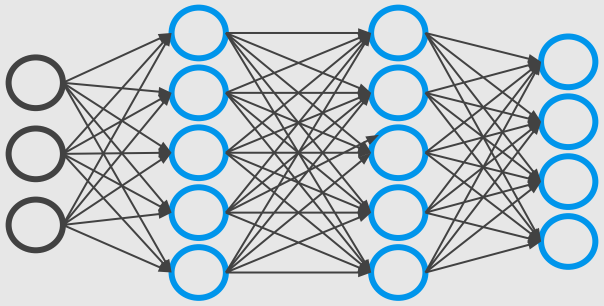
$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$

No $\delta^{(1)}$.

Gradient Computation: Backpropagation Algorithm

Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l.

For each output unit (layer $L = 4$)



$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

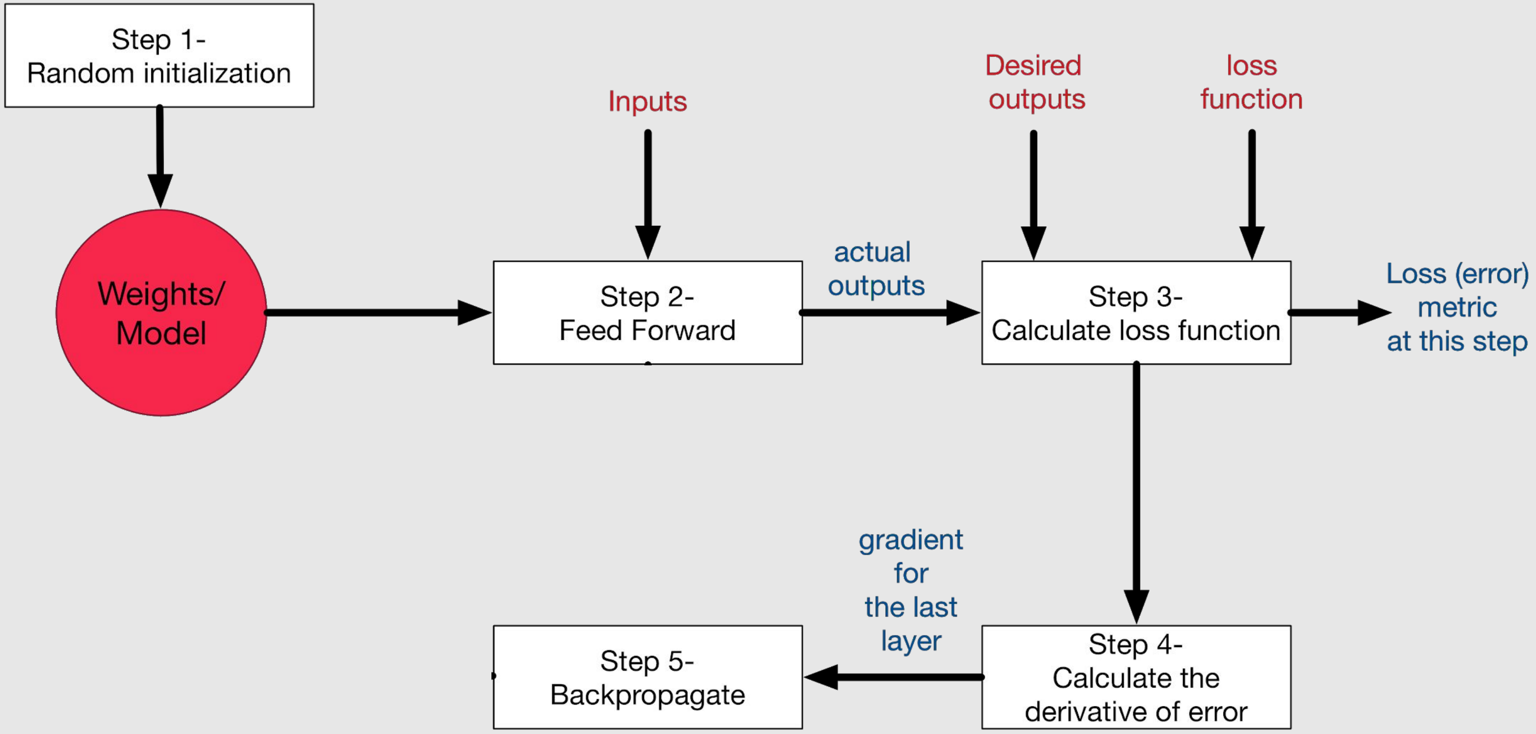
For each hidden unit

$$\delta^{(3)} = (\Theta^{(3)})^{T}\delta^{(4)} .* g'(z^{(3)})$$

$$\delta^{(2)} = (\Theta^{(2)})^{T}\delta^{(3)} .* g'(z^{(2)})$$

$$\boxed{\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)}\delta_i^{(l+1)}}$$

Training a Neural Network



Backpropagation Algorithm

Backpropagation Algorithm

Training Set: $(x^{(1)}, y^{(1)}),\ (x^{(2)}, y^{(2)}),\ \ldots,\ (x^{(m)}, y^{(m)})$

Backpropagation Algorithm

Training Set: $(x^{(1)}, y^{(1)}),\ (x^{(2)}, y^{(2)}),\ \ldots,\ (x^{(m)}, y^{(m)})$

Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j)

will be used as accumulators for computing $\dfrac{\partial}{\partial \Theta_{i,j}^{(l)}} J(\Theta)$

Backpropagation Algorithm

Training Set: $(x^{(1)}, y^{(1)}),\ (x^{(2)}, y^{(2)}),\ \ldots,\ (x^{(m)}, y^{(m)})$

Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j)

For i = 1 to m

Set $a^{(1)} = x^{(i)}$

Backpropagation Algorithm

Training Set: $(x^{(1)}, y^{(1)}),\ (x^{(2)}, y^{(2)}),\ \ldots,\ (x^{(m)}, y^{(m)})$

Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j)

For i = 1 to m

Set $a^{(1)} = x^{(i)}$

Performed forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$

Backpropagation Algorithm

Training Set: $(x^{(1)}, y^{(1)}),\ (x^{(2)}, y^{(2)}),\ \ldots,\ (x^{(m)}, y^{(m)})$

Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j)

For i = 1 to m

Set $a^{(1)} = x^{(i)}$

Performed forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$

Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$

Backpropagation Algorithm

Training Set: $(x^{(1)},y^{(1)}),\ (x^{(2)},y^{(2)}),\ \ldots,\ (x^{(m)},y^{(m)})$

Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j)

For $i = 1$ to m

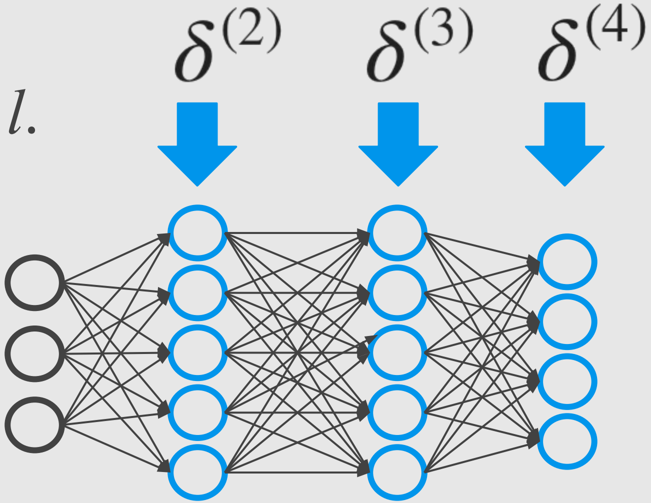
Set $a^{(1)} = x^{(i)}$

Performed forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$

Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$

Compute $\delta^{(L-1)},\delta^{(L-2)},\ldots,\delta^{(2)}$

Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l.



For each output unit (layer $L = 4$)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

For each hidden unit

$$\delta^{(3)} = (\Theta^{(3)})^{\mathrm{T}} \delta^{(4)} .* \boxed{g'(z^{(3)})} \qquad a^{(3)}(1 - a^{(3)})$$

$$\delta^{(2)} = (\Theta^{(2)})^{\mathrm{T}} \delta^{(3)} .* g'(z^{(2)})$$

Backpropagation Algorithm

Training Set: $(x^{(1)}, y^{(1)}),\ (x^{(2)}, y^{(2)}),\ \dots,\ (x^{(m)}, y^{(m)})$

Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j)

For i = 1 to m

Set $a^{(1)} = x^{(i)}$

Performed forward propagation to compute $a^{(l)}$ for $l = 2, 3, \dots, L$

Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$

Compute $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$

$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

Backpropagation Algorithm

Training Set: $(x^{(1)},y^{(1)}),\ (x^{(2)},y^{(2)}),\ \ldots,\ (x^{(m)},y^{(m)})$

Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j)

For i = 1 to m

Set $a^{(1)} = x^{(i)}$

Performed forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$

Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$

Compute $\delta^{(L-1)},\delta^{(L-2)},\ldots,\delta^{(2)}$

$$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)}\delta_i^{(l+1)}$$

$$D_{ij}^{(l)} := \frac{1}{m}\Delta_{ij}^{(l)}$$

Backpropagation Algorithm

Training Set: $(x^{(1)}, y^{(1)}),\ (x^{(2)}, y^{(2)}),\ \ldots,\ (x^{(m)}, y^{(m)})$

Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j)

For $i = 1$ to m

Set $a^{(1)} = x^{(i)}$

Performed forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$

Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$

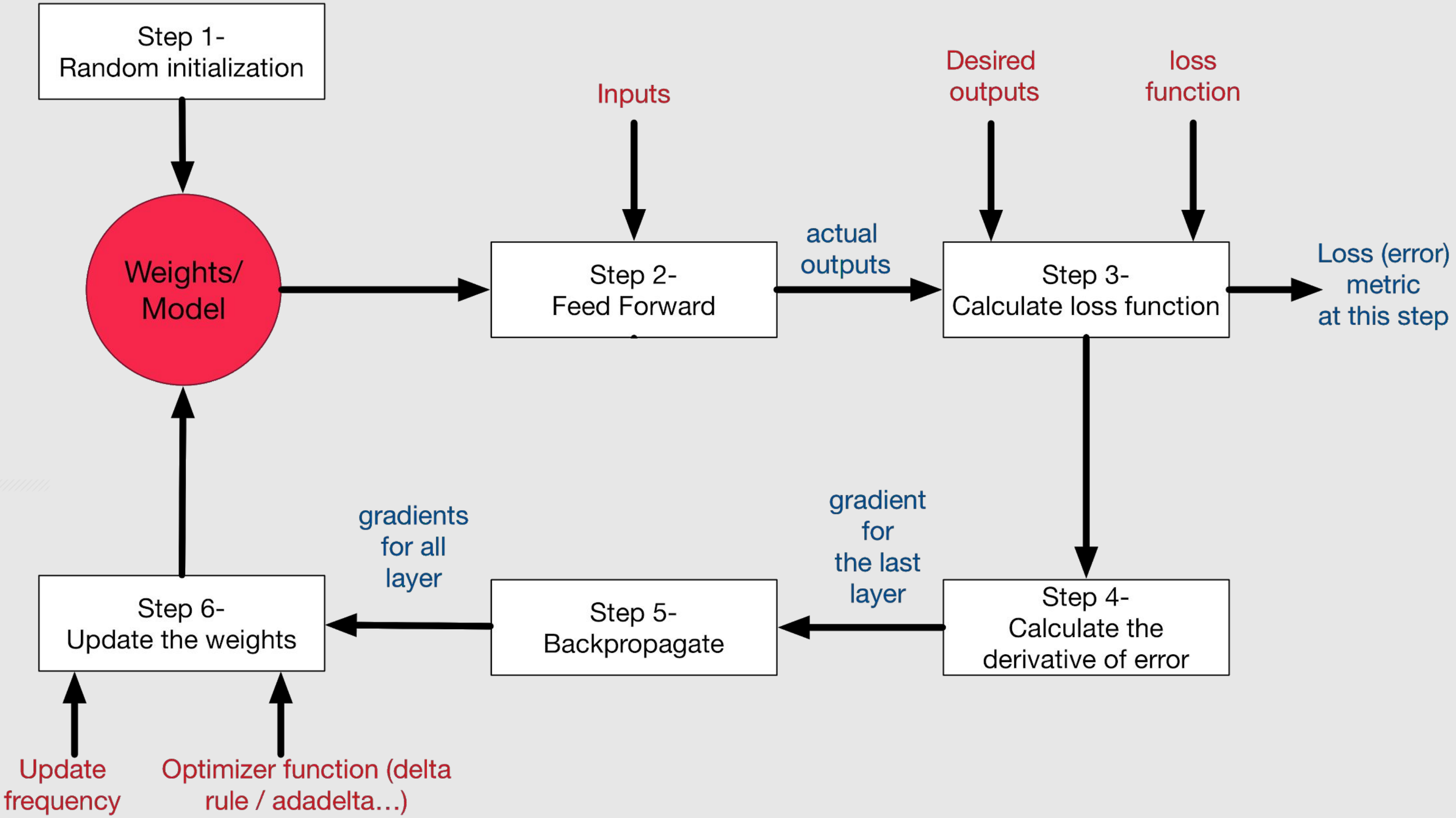
Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$

$$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$$

$$\frac{\partial}{\partial \Theta_{i,j}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

$$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)}$$

Training a Neural Network



Gradient Descent

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)}\log(h_\Theta(x^{(i)}))_k + (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)\right]$$

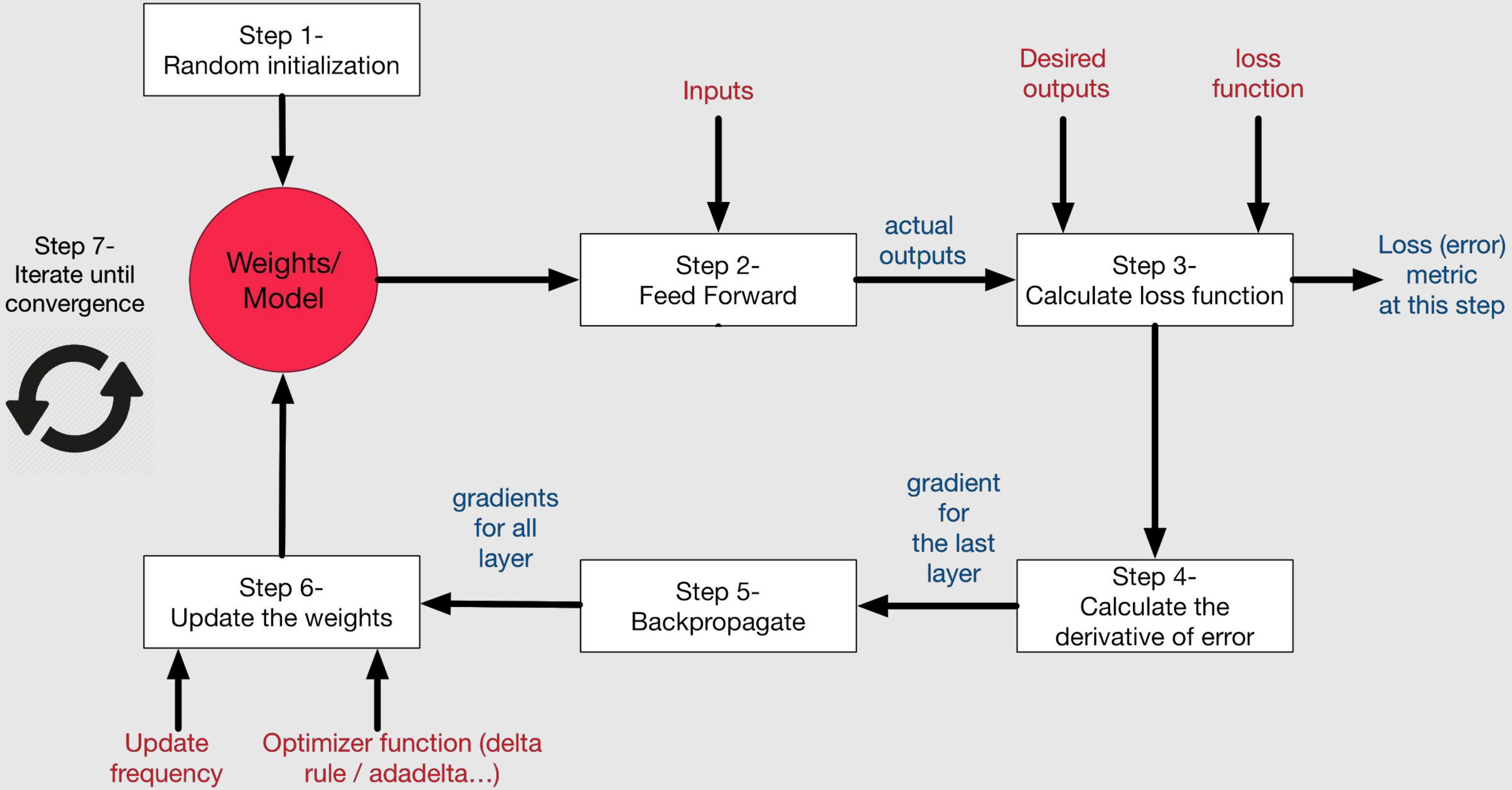
Want $\min_{\Theta} J(\Theta)$:

repeat {

$$\Theta_{ij}^{(l)} := \Theta_{ij}^{(l)} - \alpha\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta)$$

}

Training a Neural Network



How many iterations are needed to converge?

How many iterations are needed to converge?

1. It depends on the meta-parameters of the network (how many layers, how complex the nonlinear functions are).

How many iterations are needed to converge?

1. It depends on the meta-parameters of the network (how many layers, how complex the nonlinear functions are).
2. It depends on the learning rate.

How many iterations are needed to converge?

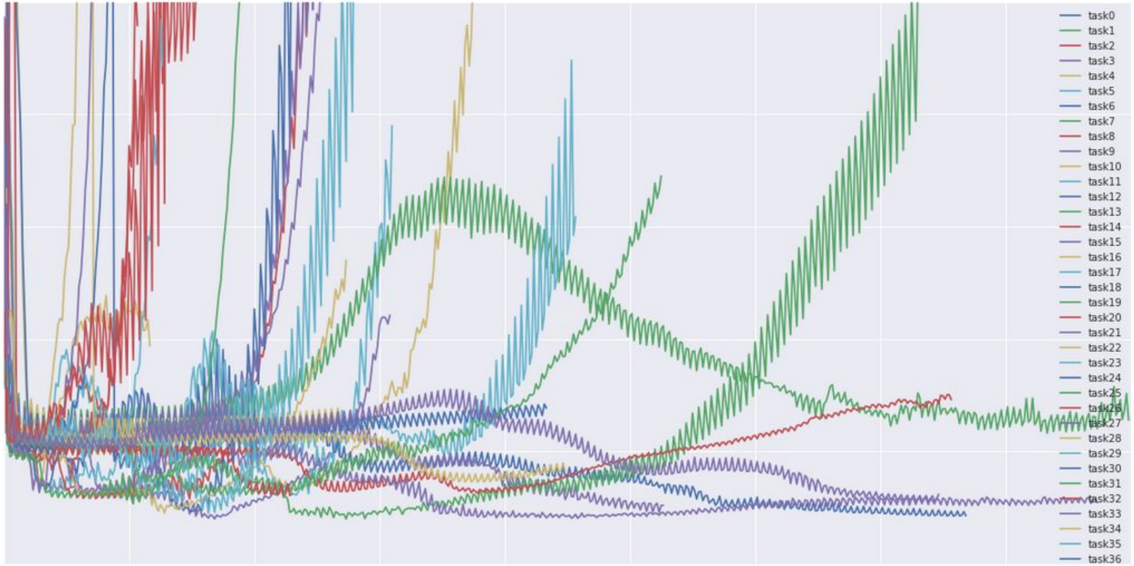
1. It depends on the meta-parameters of the network (how many layers, how complex the nonlinear functions are).
2. It depends on the learning rate.
3. It depends on the optimization method.

http://ruder.io/optimizing-gradient-descent/

Sebastian Ruder

I'm a PhD student in Natural Language Processing and a research scientist at AYLIEN. I blog about Machine Learning, Deep Learning, NLP, and startups.

Deep Learning, NLP, ...



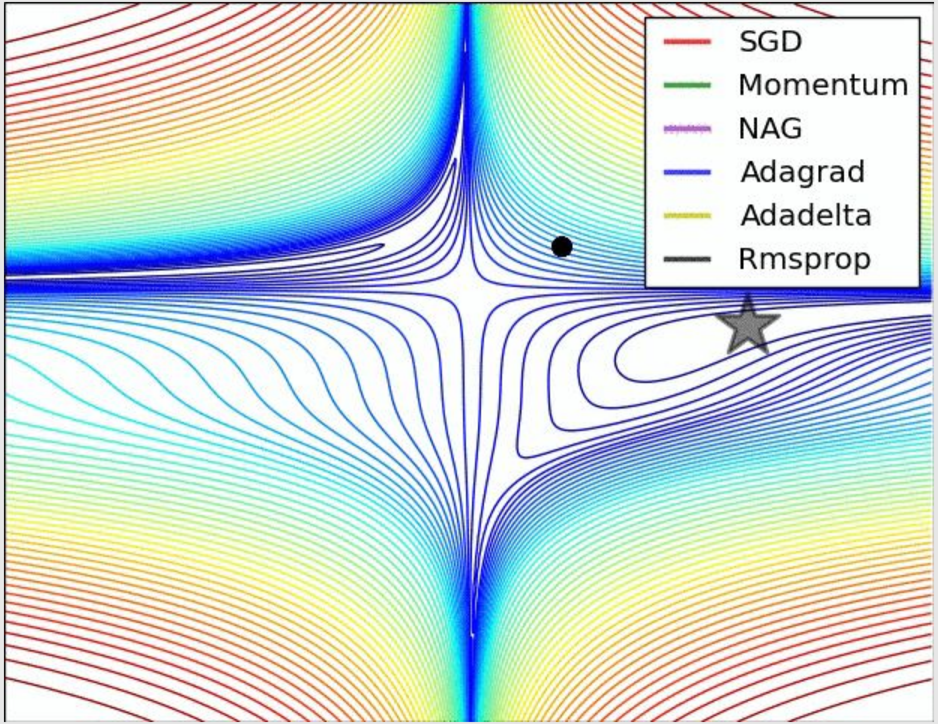
An overview of gradient descent optimization algorithms

- Momentum
- Nesterov
- Adagrad

- Adadelta
- RMSprop
- Adam

- AdaMax
- Nadam

- Batch gradient descent



Credit: Alec Radford.

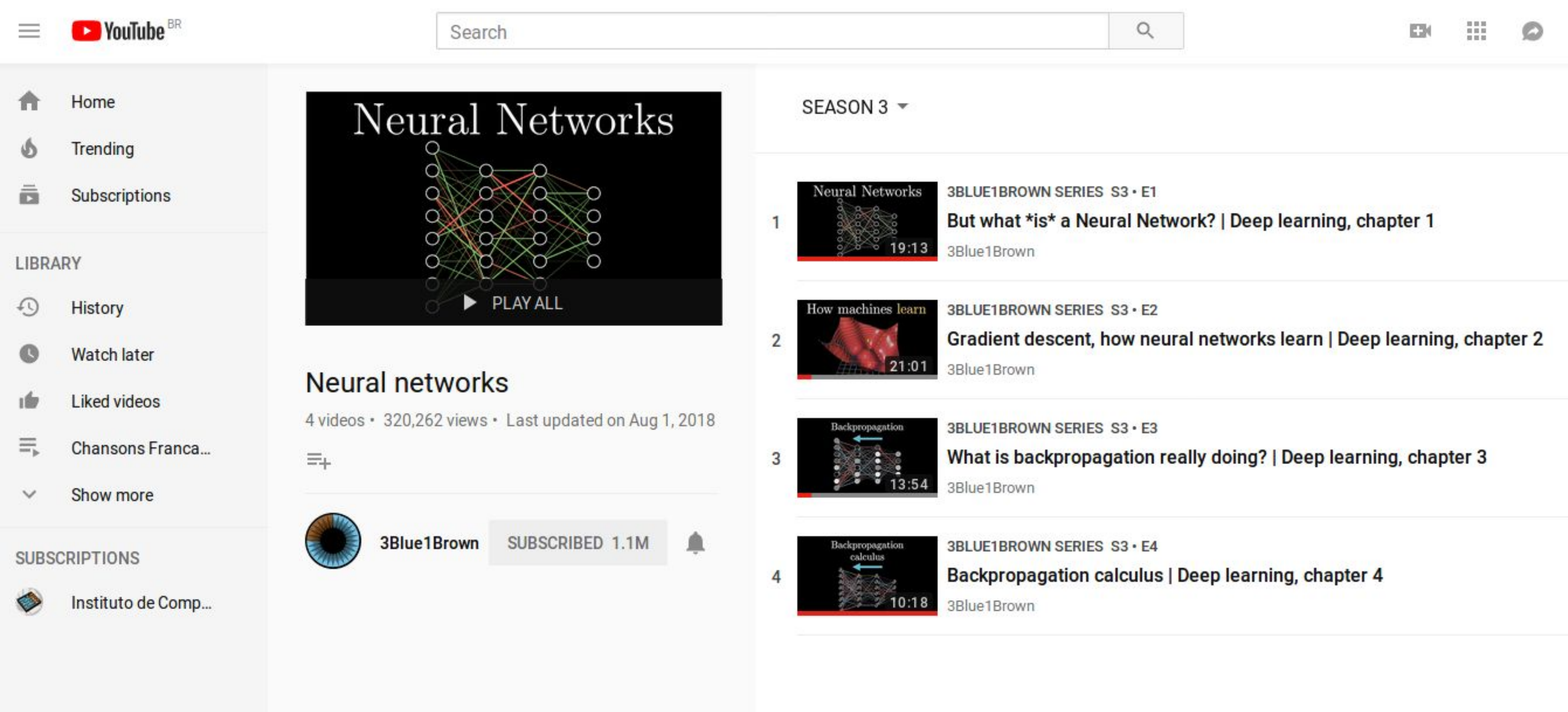
How many iterations are needed to converge?

1. It depends on the meta-parameters of the network (how many layers, how complex the nonlinear functions are).
2. It depends on the learning rate.
3. It depends on the optimization method.
4. It depends on the random initialization of the network.

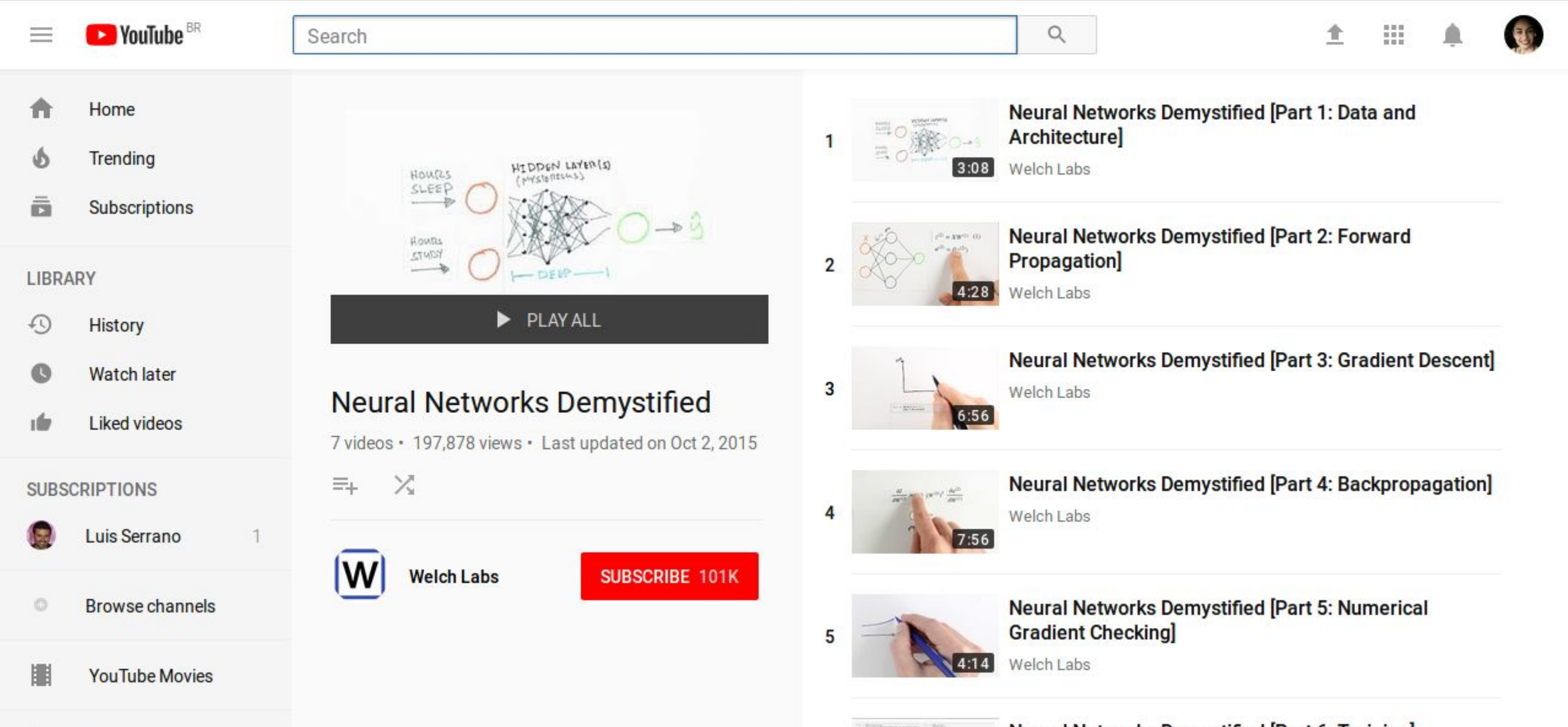
How many iterations are needed to converge?

1. It depends on the meta-parameters of the network (how many layers, how complex the nonlinear functions are).
2. It depends on the learning rate.
3. It depends on the optimization method.
4. It depends on the random initialization of the network.
5. It depends on the quality of the training set.

Neural Networks (3Blue1Brown)



Neural Networks Demystified (in Python)



References

Machine Learning Books

- Hands-On Machine Learning with Scikit-Learn and TensorFlow, Chap. 10
- Pattern Recognition and Machine Learning, Chap. 5
- Pattern Classification, Chap. 6
- Free online book: **http://neuralnetworksanddeeplearning.com**

Machine Learning Courses

- https://www.coursera.org/learn/machine-learning, Week 4 & 5
- https://www.coursera.org/learn/neural-networks