Recall from last time ...

What does an artificial neuron do?

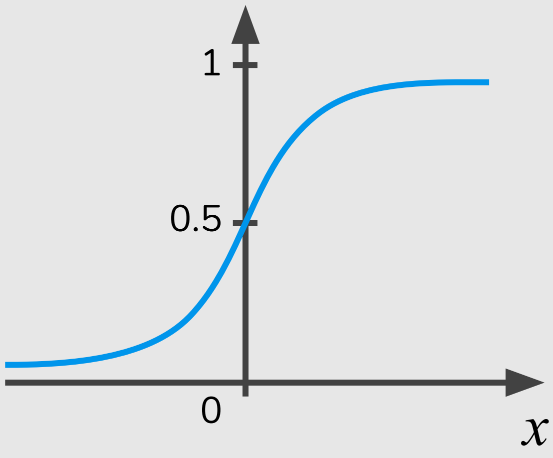
It calculates a “weighted sum” of its input, adds a bias and then decides whether it should be “fired” or not.

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

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

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?**

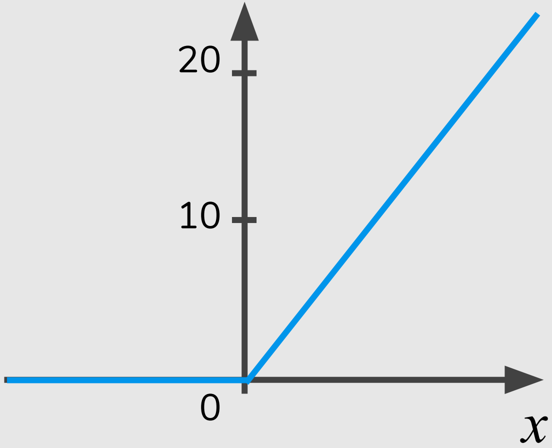
- Towards either end of the sigmoid function, the $\sigma(x)$ values tend to respond very less to changes in x.
- The problem of "**vanishing gradients**".
  - Cannot make significant change because of the extremely small value.

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!



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

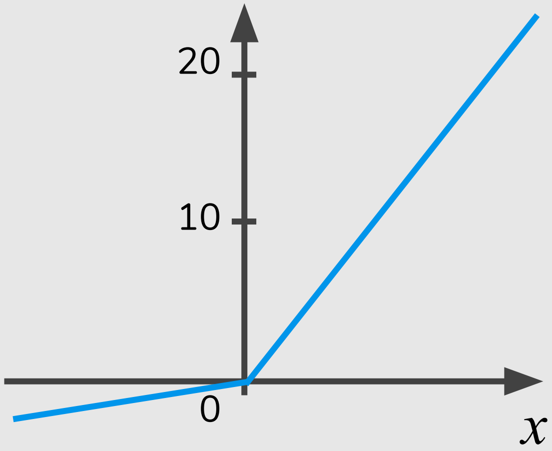
ReLU Function: **Problem?**

- 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!

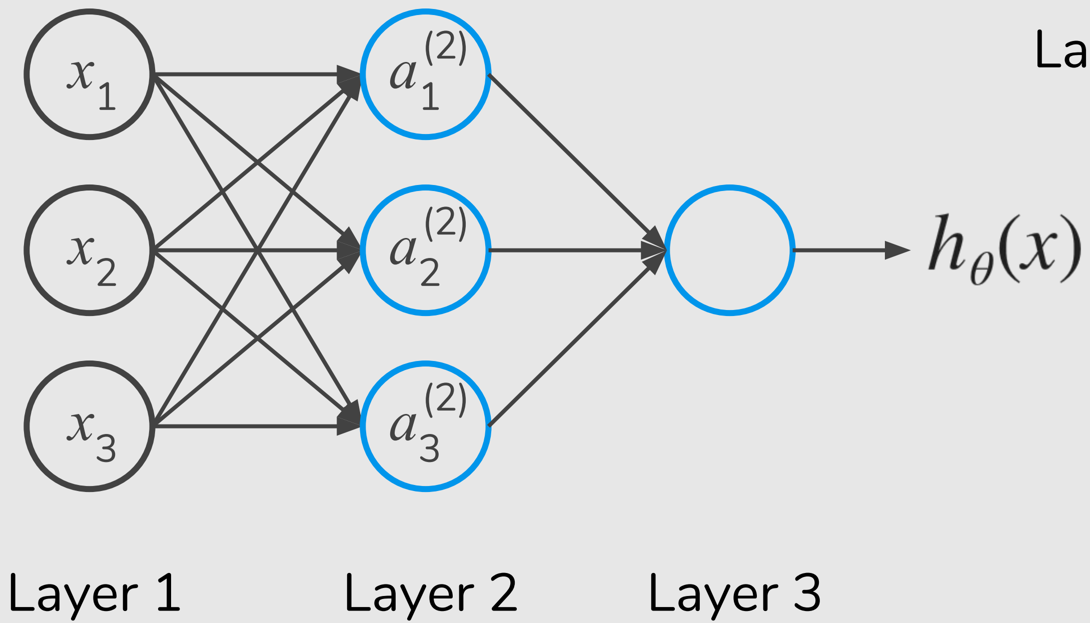
Neural Network

Neural Network

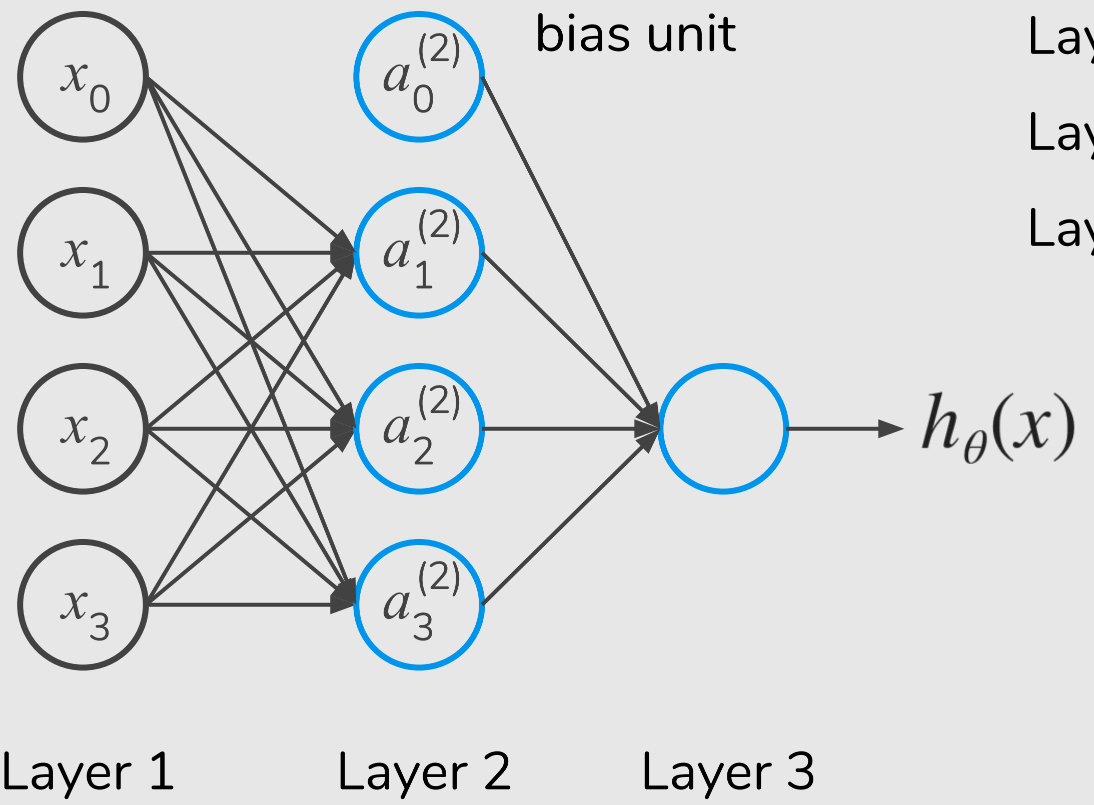
Layer 1 = Input layer

Layer 2 = Hidden layer

Layer 3 = Output layer



Neural Network

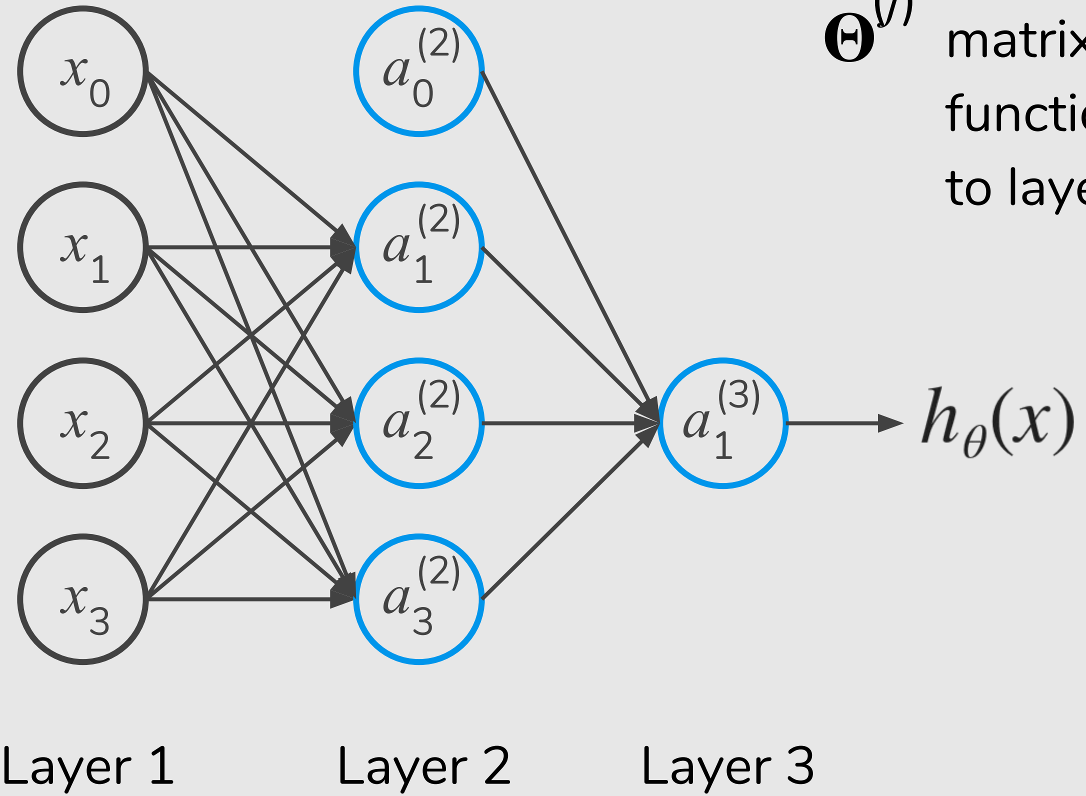


Layer 1 = Input layer

Layer 2 = Hidden layer

Layer 3 = Output layer

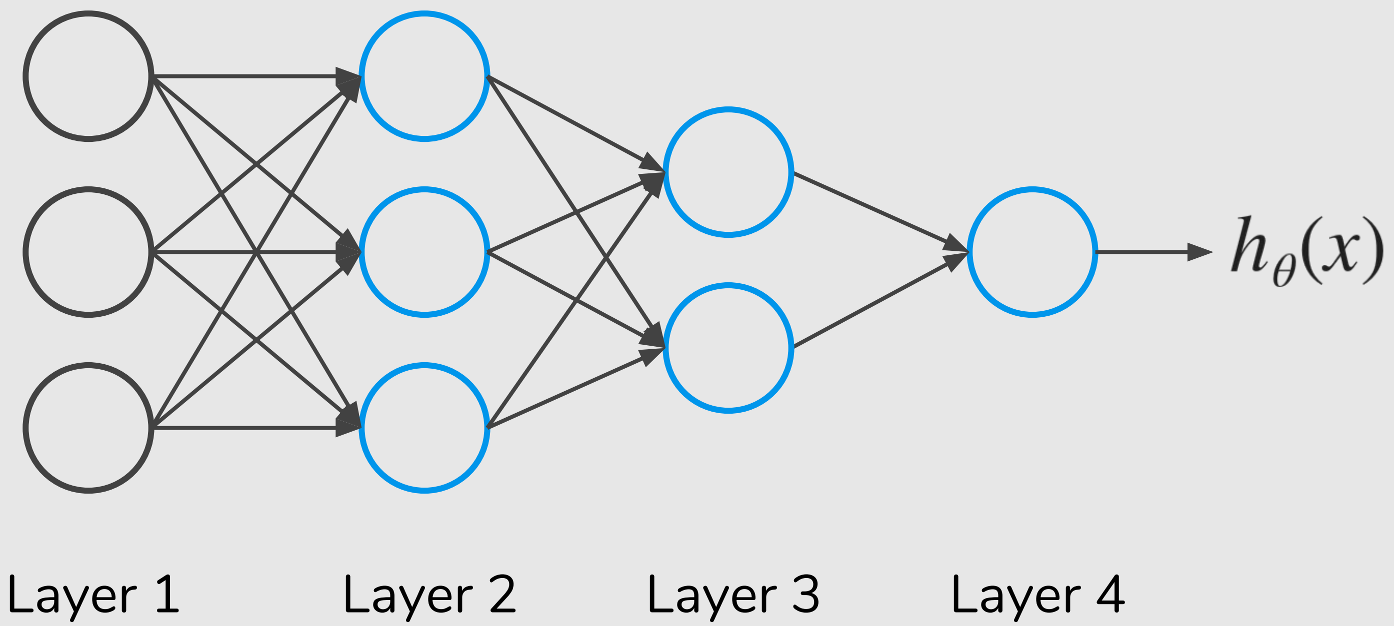
Neural Network



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

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

Other Network Architectures



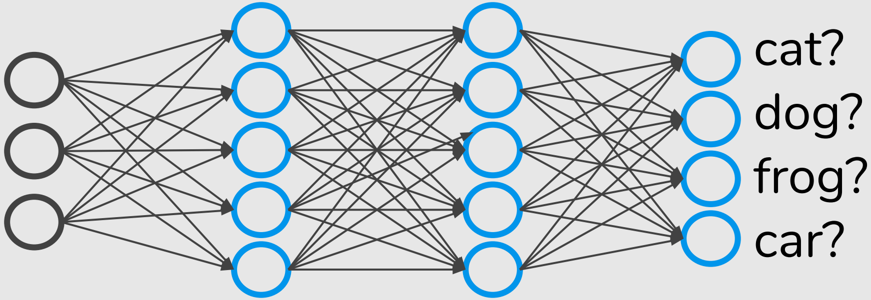
Multi-class Classification


Cat


Dog


Frog


Car

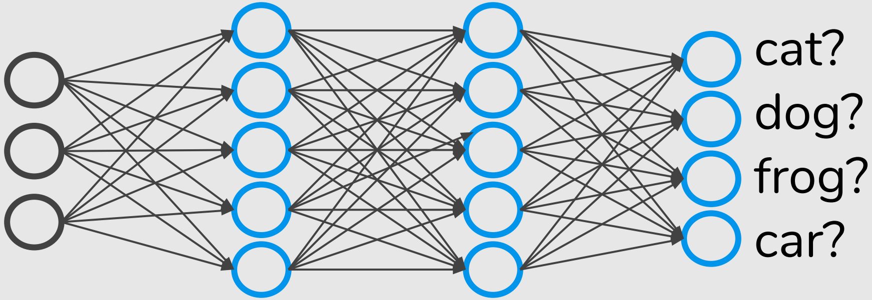


Cat

Dog

Frog

Car

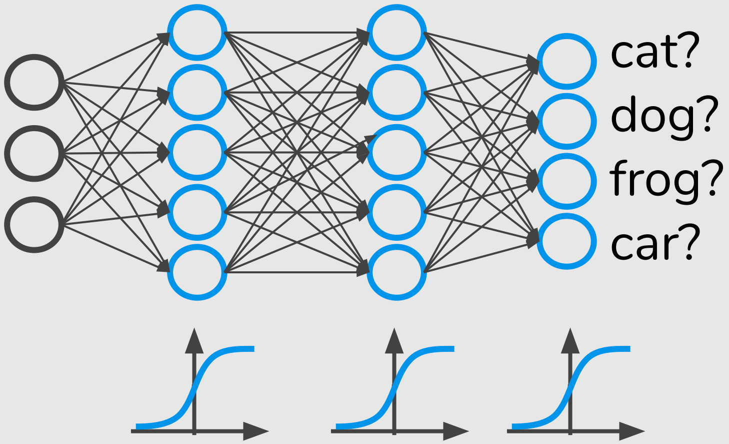


Want $h_\Theta(x) \approx \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$, $h_\Theta(x) \approx \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$, $h_\Theta(x) \approx \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$, $h_\Theta(x) \approx \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

when cat, when dog, when frog, when car

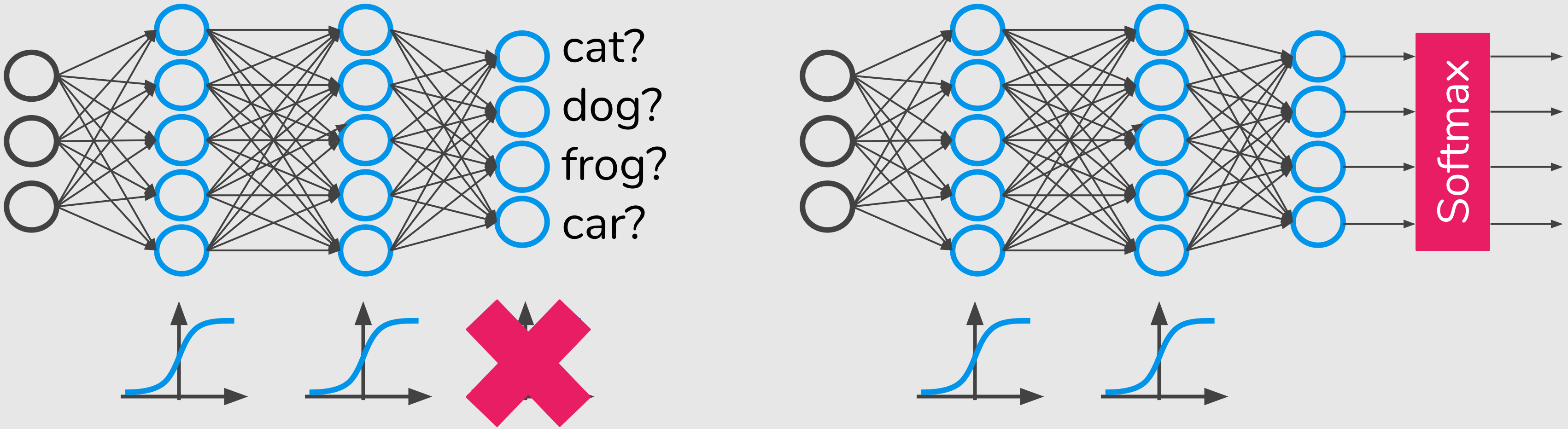
Softmax Classification

The **output layer** is typically modified **by replacing** the individual activation functions **by a shared softmax** function.



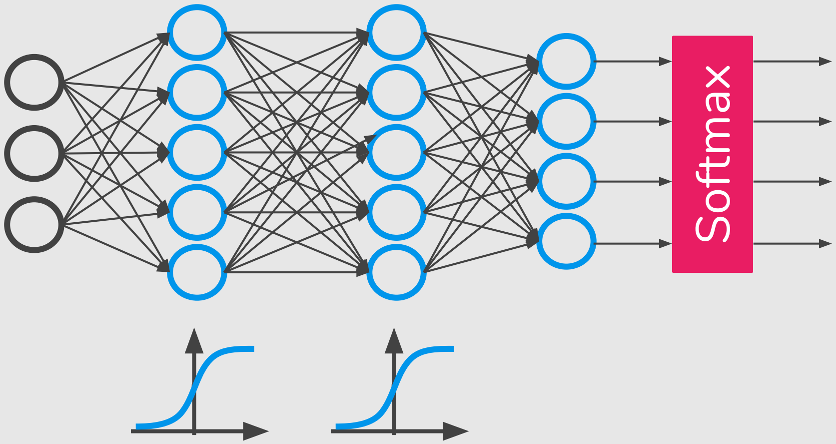
Softmax Classification

The **output layer** is typically modified **by replacing** the individual activation functions **by a shared softmax** function.



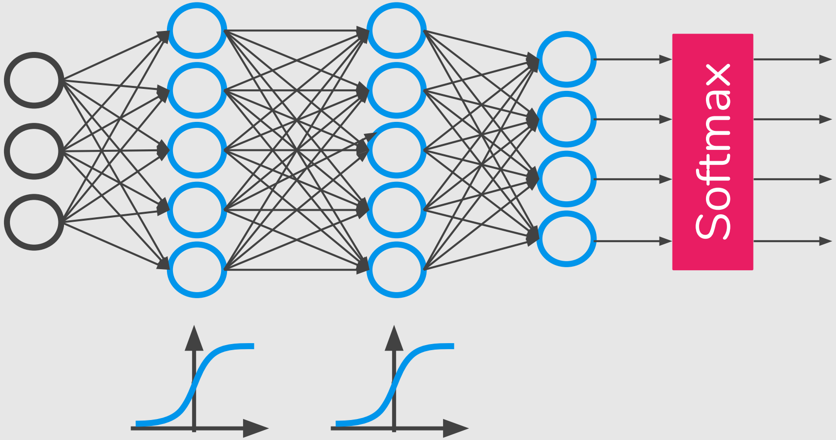
Softmax Classification

The **output layer** is typically modified **by replacing** the individual activation functions **by a shared softmax** function.



$$f(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}$$

Softmax Classification



Cat	5.1	164.0	0.87
Dog	3.2	24.5	0.13
Frog	-1.7	0.18	0.00
Car	-2.0	0.13	0.00

$$\boxed{f(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}}$$

Artificial Neural Networks

Machine Learning and Pattern Recognition

Prof. Sandra Avila
Institute of Computing (IC/Unicamp)

MC886/MO444, September 13, 2018

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

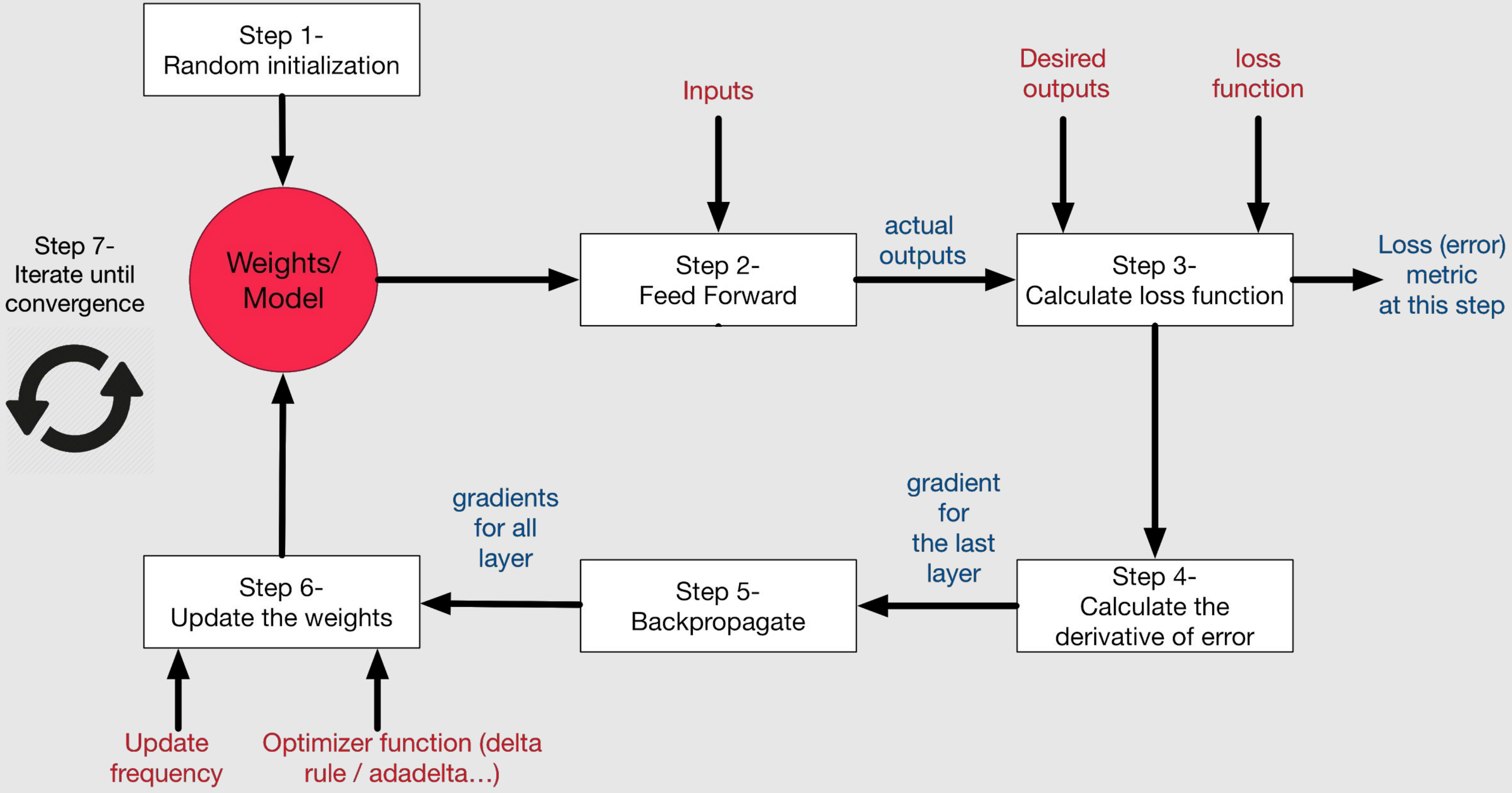
Training a Neural Network

- 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)

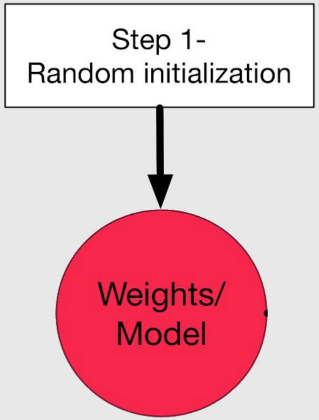
Training a Neural Network

- **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

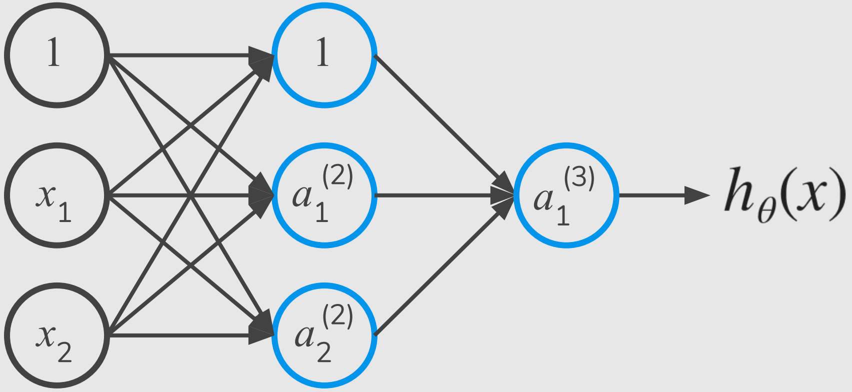


Training a Neural Network



Zero Initialization

Symmetric Weights



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

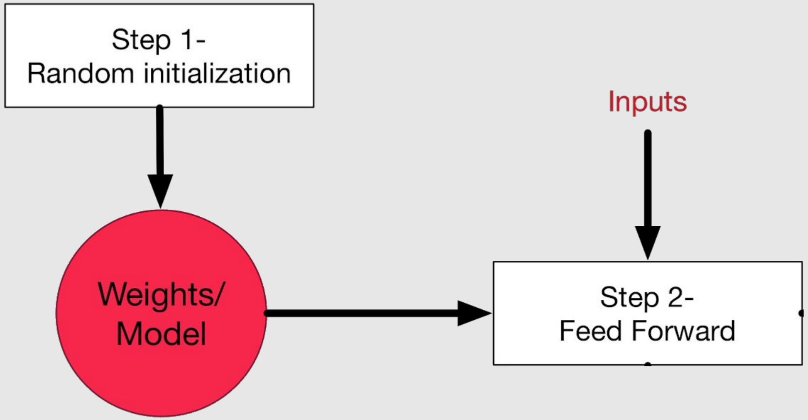
After each update, parameters corresponding to inputs going into each of two hidden units are identical.

Symmetric Breaking

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

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

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

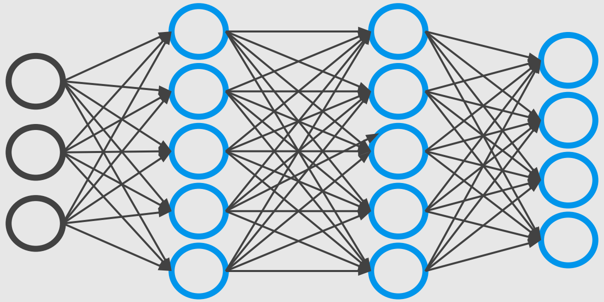
$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$

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

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

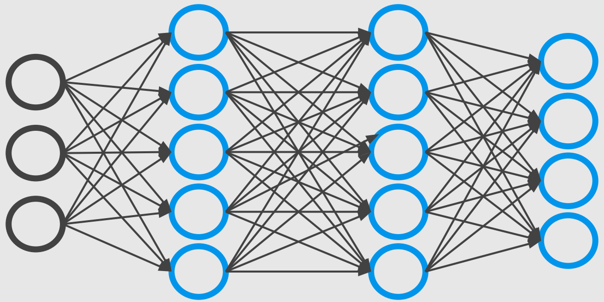
$$z^{(4)} = \Theta^{(3)} a^{(3)}$$

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



Forward Propagation

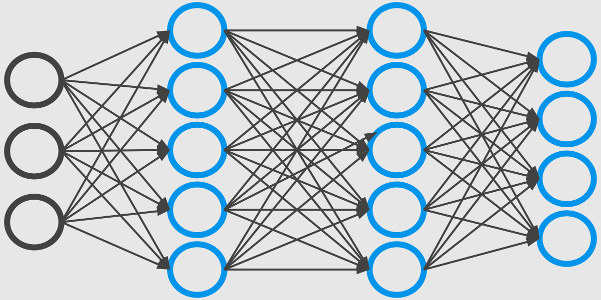
Given one training example (x, y):



Forward Propagation

Given one training example (x, y):

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

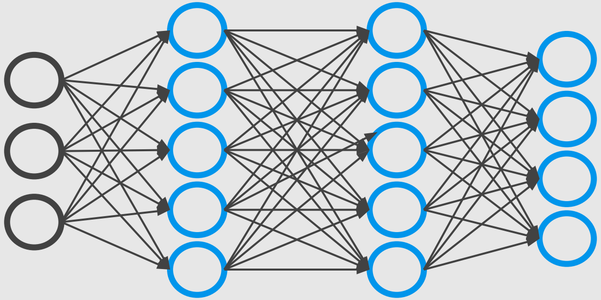


Forward Propagation

Given one training example (x, y):

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

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



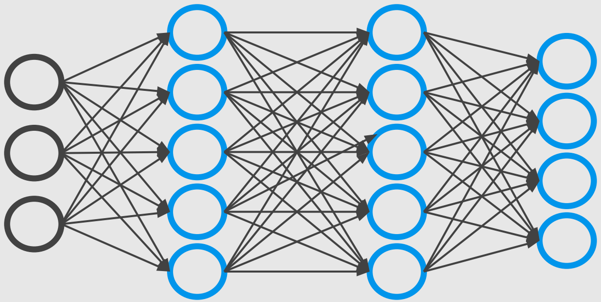
Forward Propagation

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

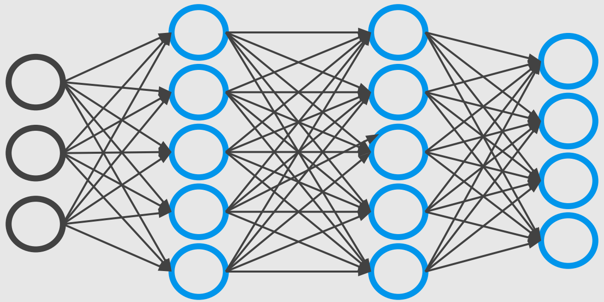
Given one training example (x, y):

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

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

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

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



Forward Propagation

Given one training example (x, y):

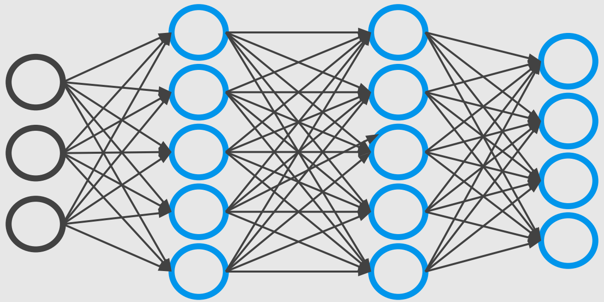
$$a^{(1)} = x$$

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

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

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

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



Forward Propagation

Given one training example (x, y):

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

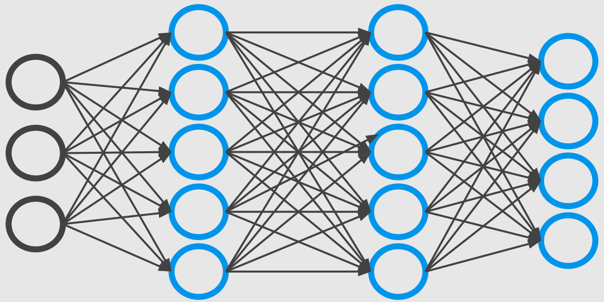
$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

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

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

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

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



Forward Propagation

Given one training example (x, y):

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

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

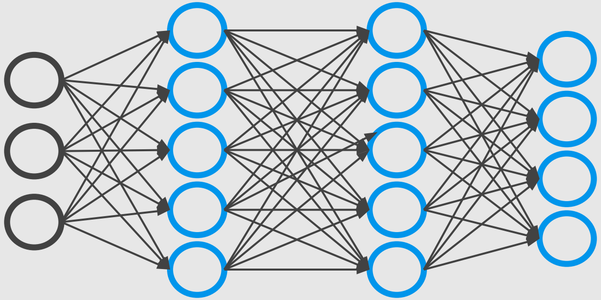
$$a^{(2)} = g(z^{(2)}) \quad (\text{add } a_0^{(2)})$$

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

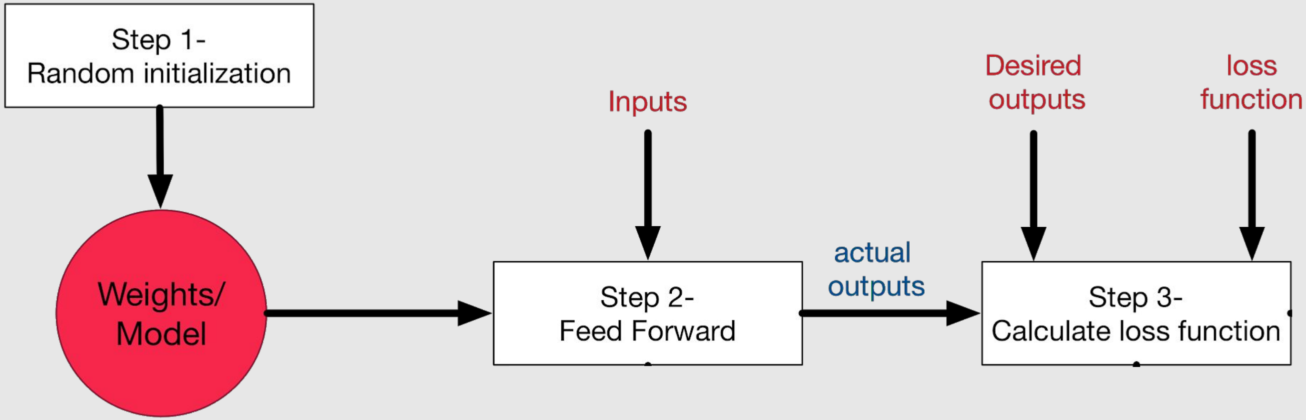
$$a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$

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

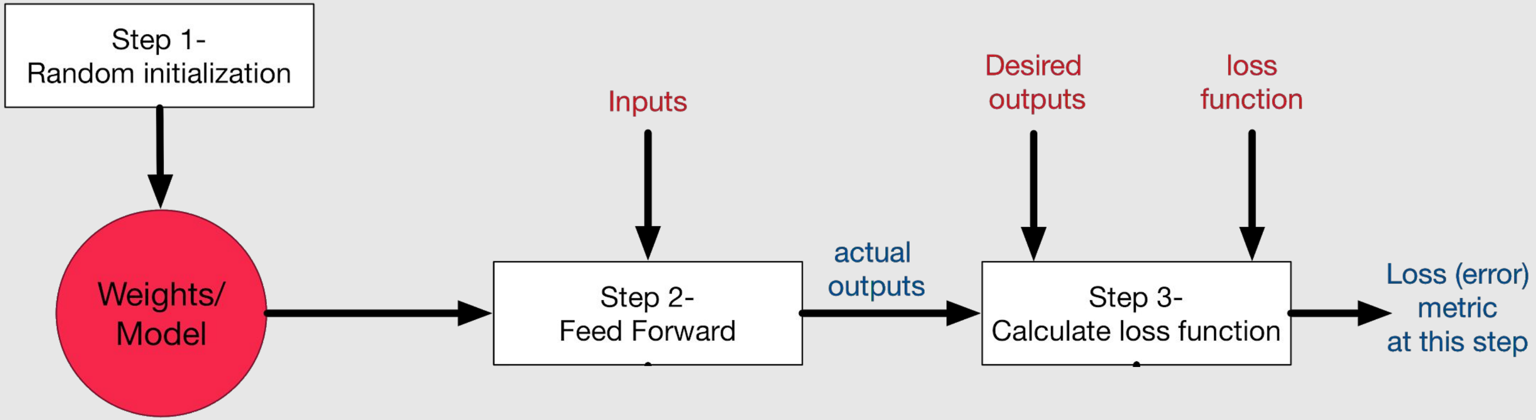
$$a^{(4)} = h_\Theta(x) = g(z^{(4)})$$



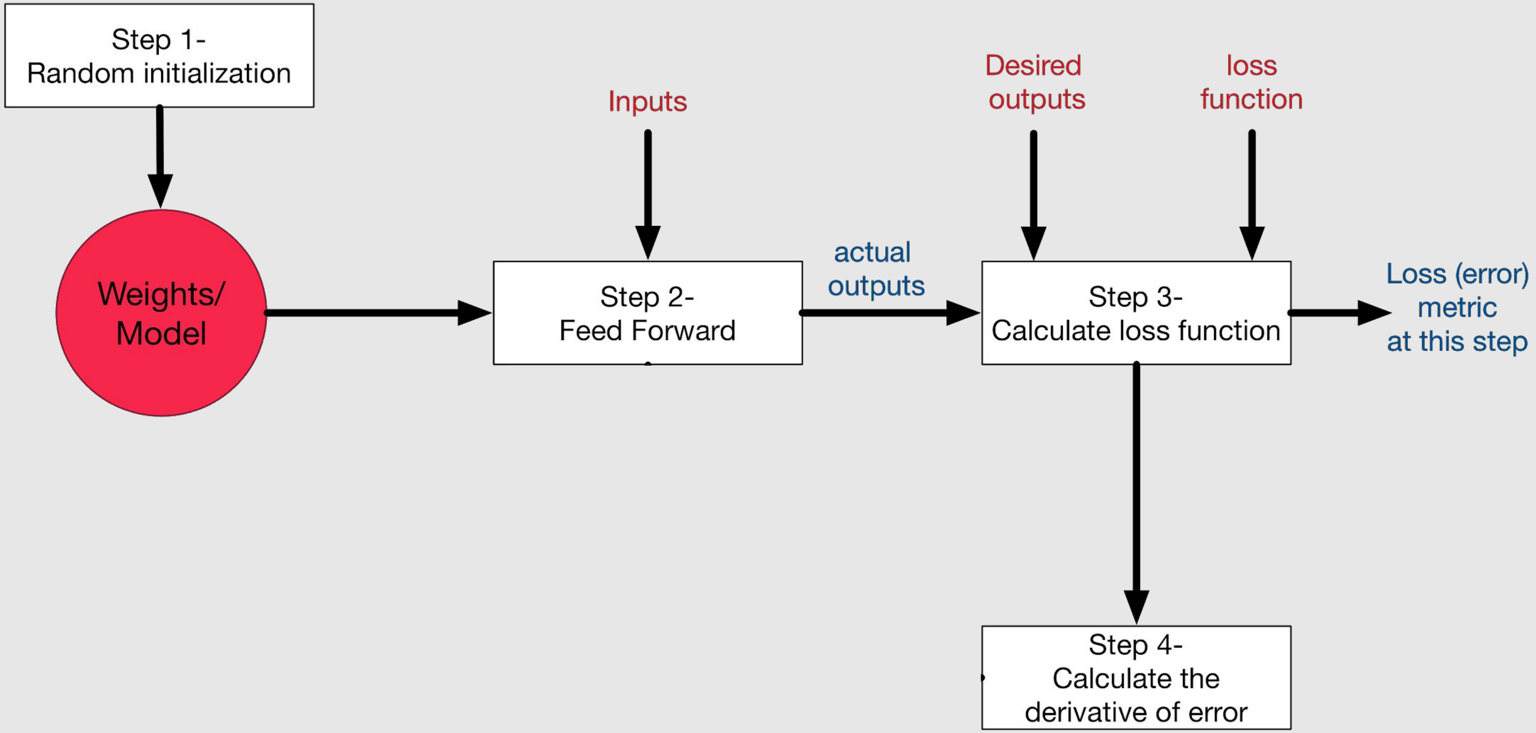
Training a Neural Network



Training a Neural Network

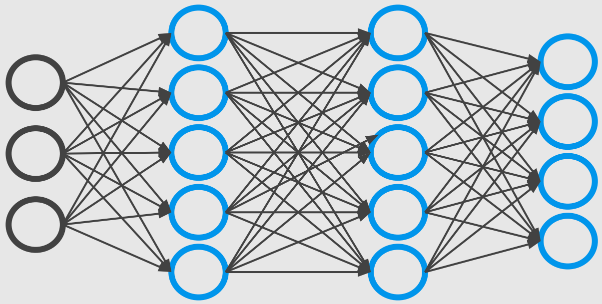


Training a Neural Network



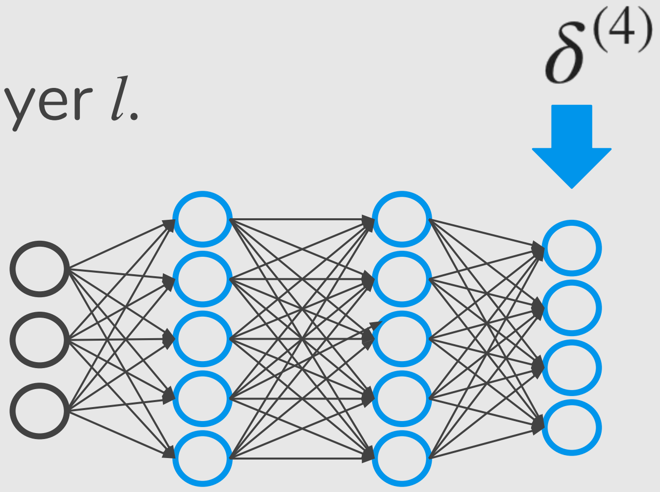
Gradient Computation: Backpropagation Algorithm

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.



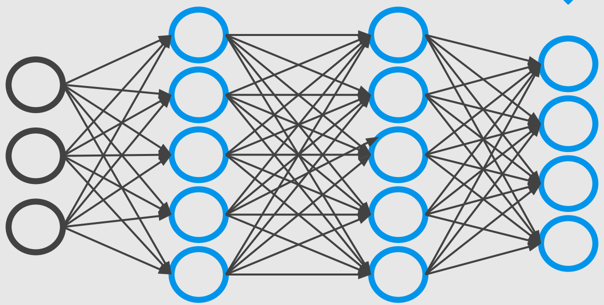
For each output unit (layer $L = 4$)

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

Gradient Computation: Backpropagation Algorithm

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

$\delta^{(4)}$



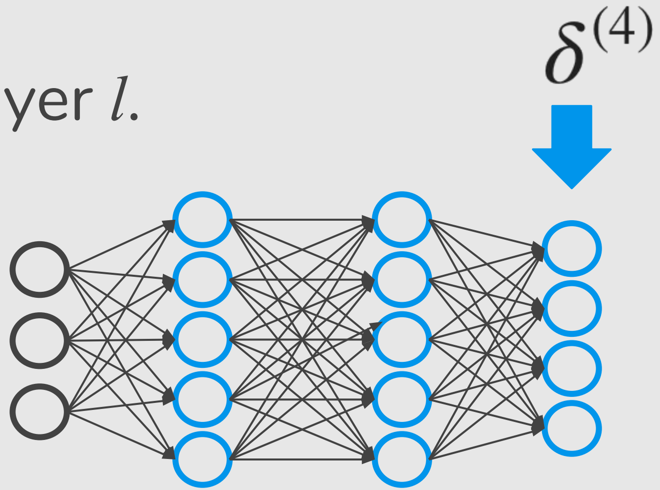
For each output unit (layer L = 4)

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

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

Gradient Computation: Backpropagation Algorithm

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



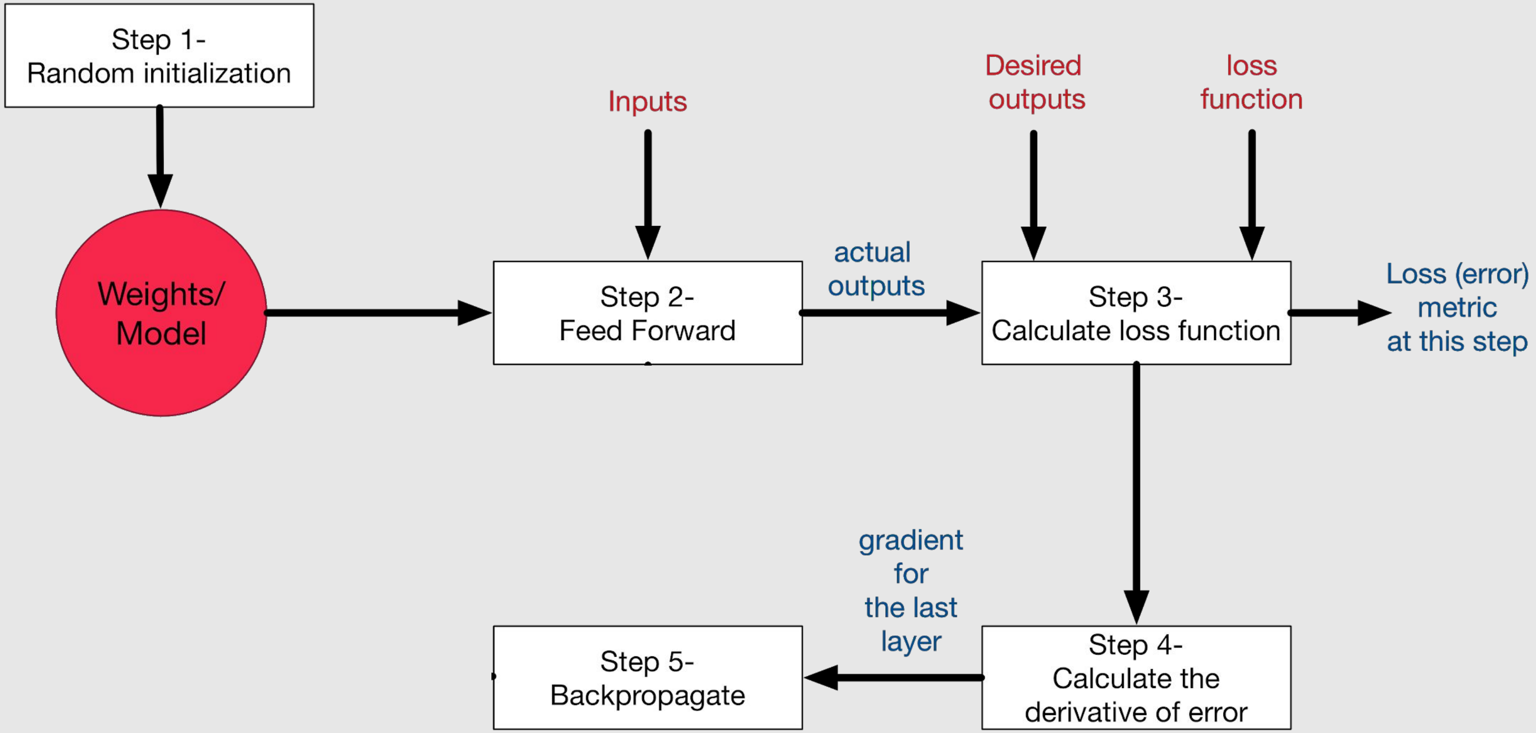
For each output unit (layer $L = 4$)

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

Vectorizing it, we have:

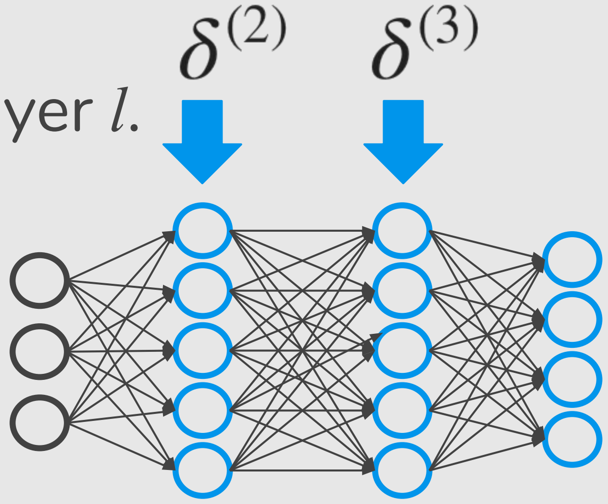
$$\delta^{(4)} = a^{(4)} - y$$

Training a Neural Network



Gradient Computation: Backpropagation Algorithm

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



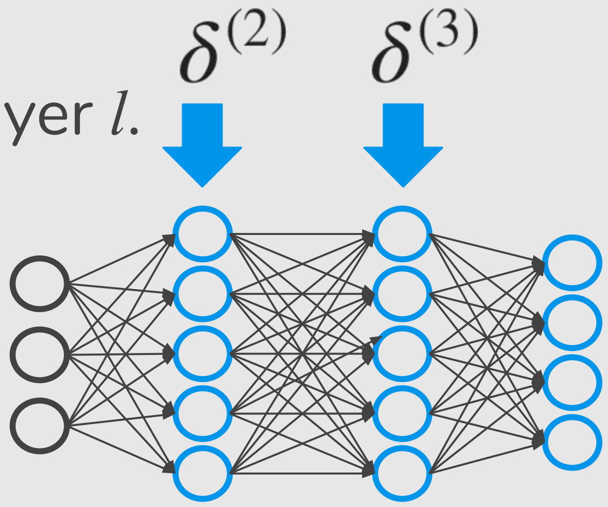
For each output unit (layer $L = 4$)

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

For each hidden unit

Gradient Computation: Backpropagation Algorithm

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



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

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

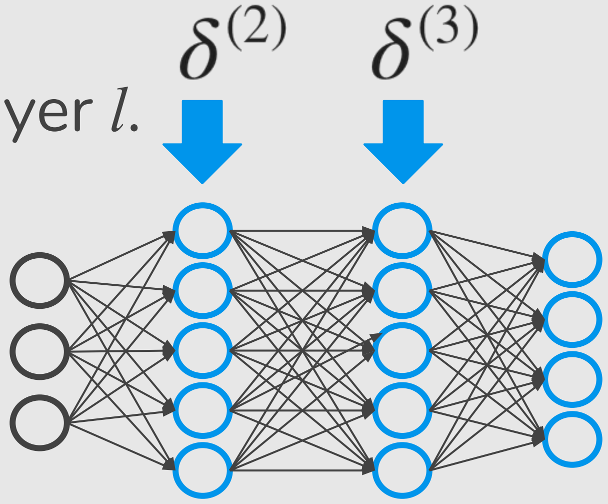
For each hidden unit

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

.* element-wise multiplication

Gradient Computation: Backpropagation Algorithm

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



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

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

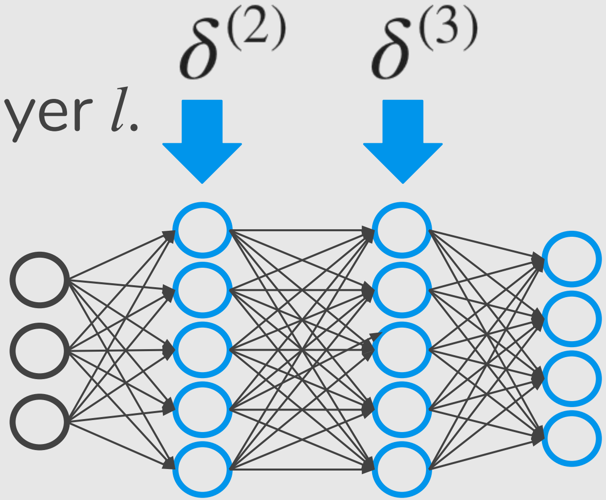
For each hidden unit

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

.* 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

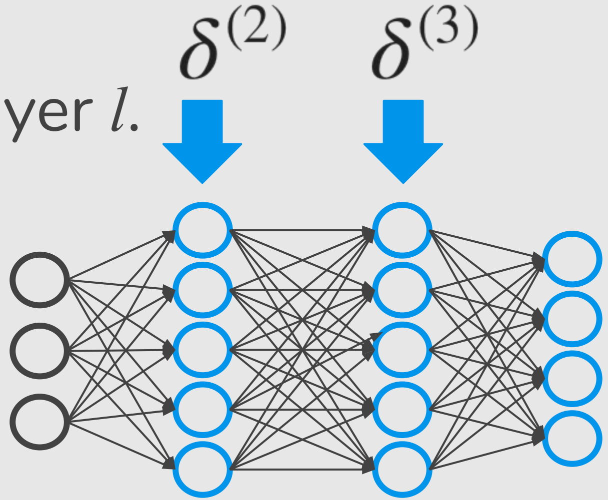
$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$$

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

$.*$ 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

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

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

Derivative of Logistic Function

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

$$g'(z) = \frac{d}{dz}\frac{1}{1+e^{-z}}$$

$$= \frac{0\cdot(1+e^{-z}) - 1\cdot(-e^{-z})}{(1+e^{-z})^2}$$ (quotient rule)

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

$$= \left(\frac{1}{1+e^{-z}}\right)\left(1-\frac{1}{1+e^{-z}}\right)$$

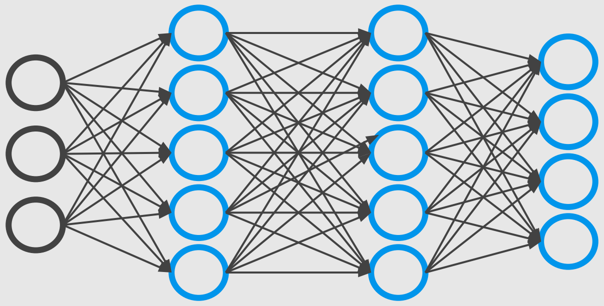
$$= g(z)(1-g(z))$$

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

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

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

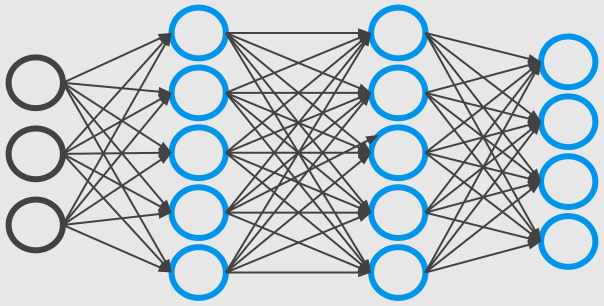
No $\delta^{(1)}$.

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

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

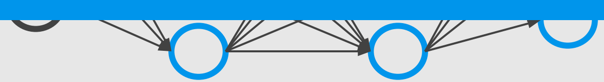
$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$

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

Gradient Computation: Backpropagation Algorithm

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

Proof: https://theclevermachine.wordpress.com/2014/09/06/derivation-error-backpropagation-gradient-descent-for-neural-networks



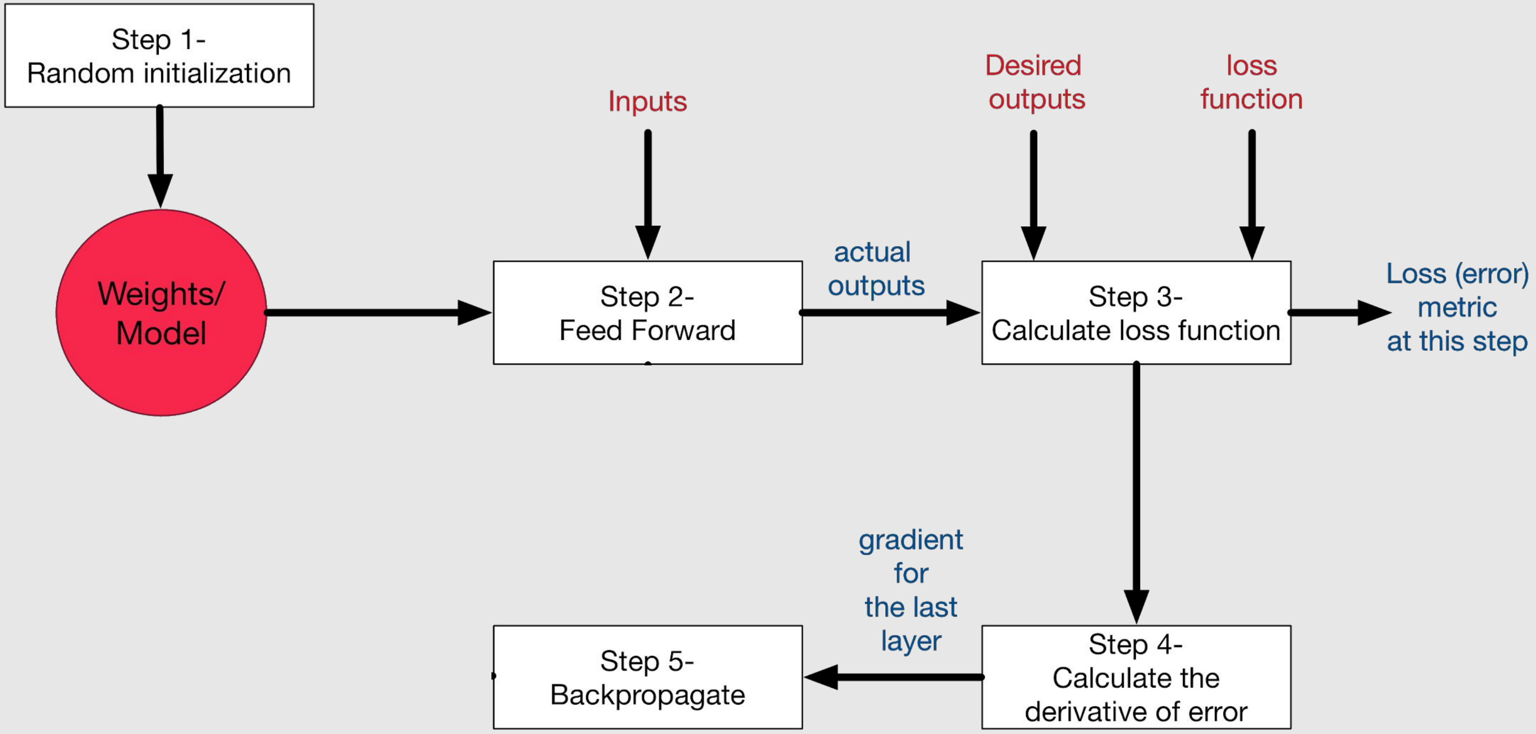
For each hidden unit

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

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

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

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

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

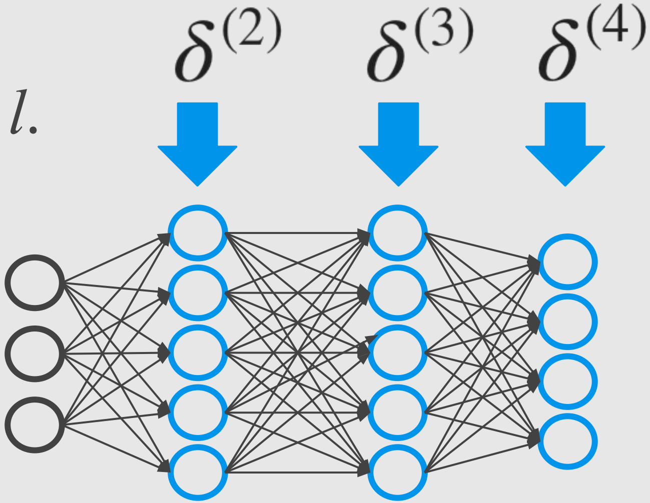
Set $a^{(1)} = x^{(i)}$

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

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

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

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

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

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

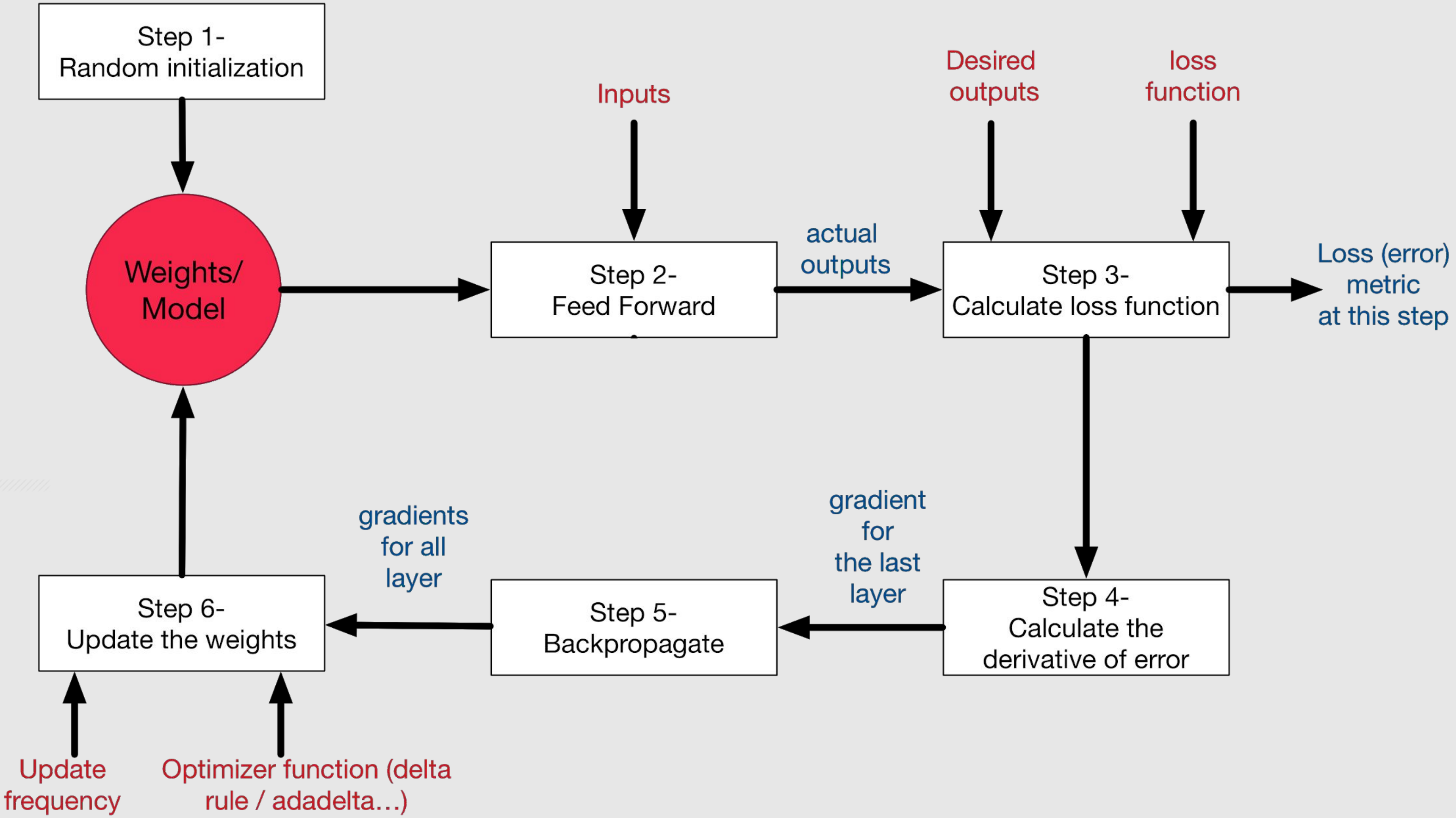
Compute $\delta^{(L-1)},\delta^{(L-2)},\ldots,\delta^{(2)}$

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

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

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

Training a Neural Network



Gradient Descent

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

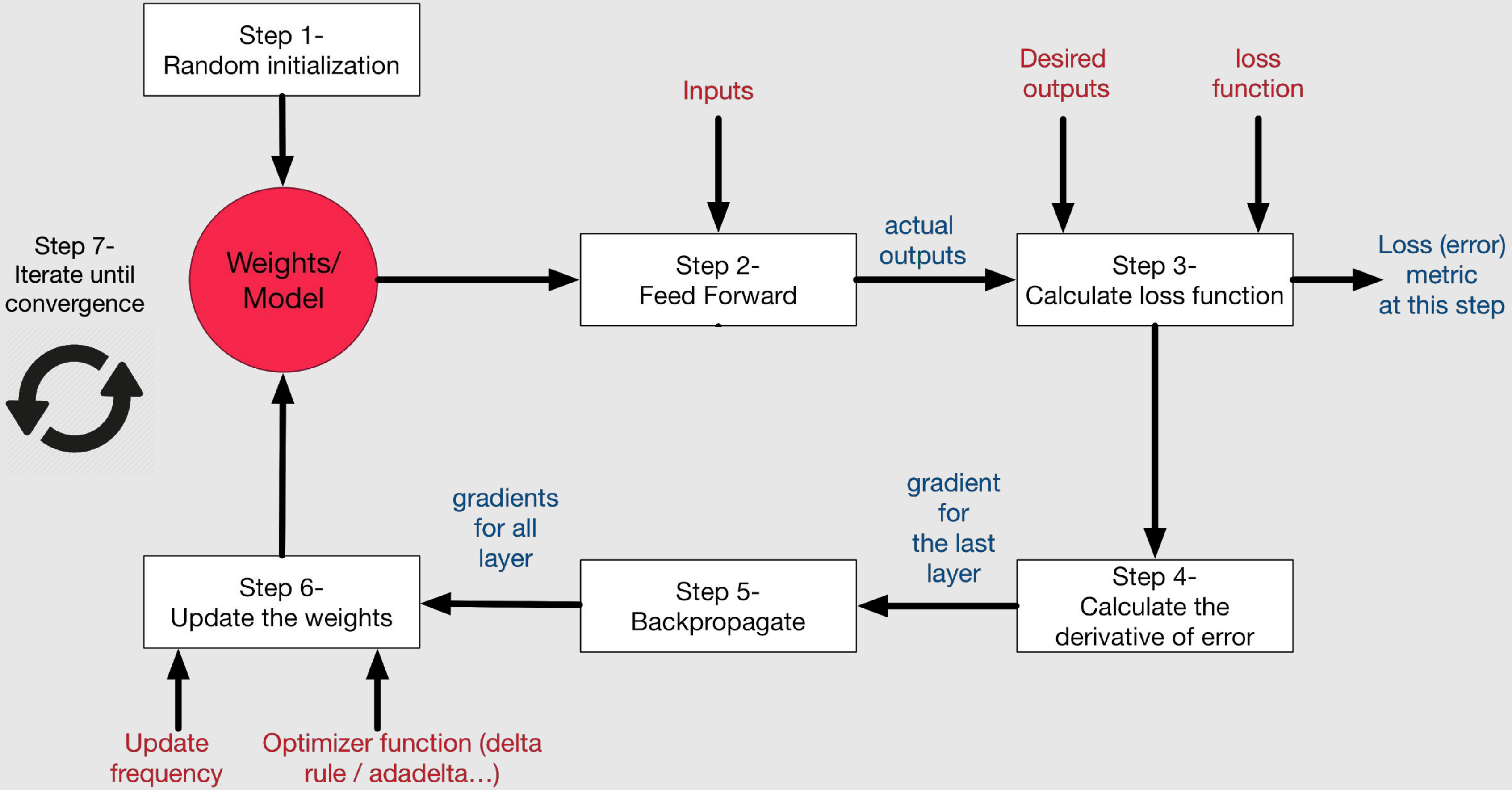
Want $\min_\Theta J(\Theta)$:

repeat {

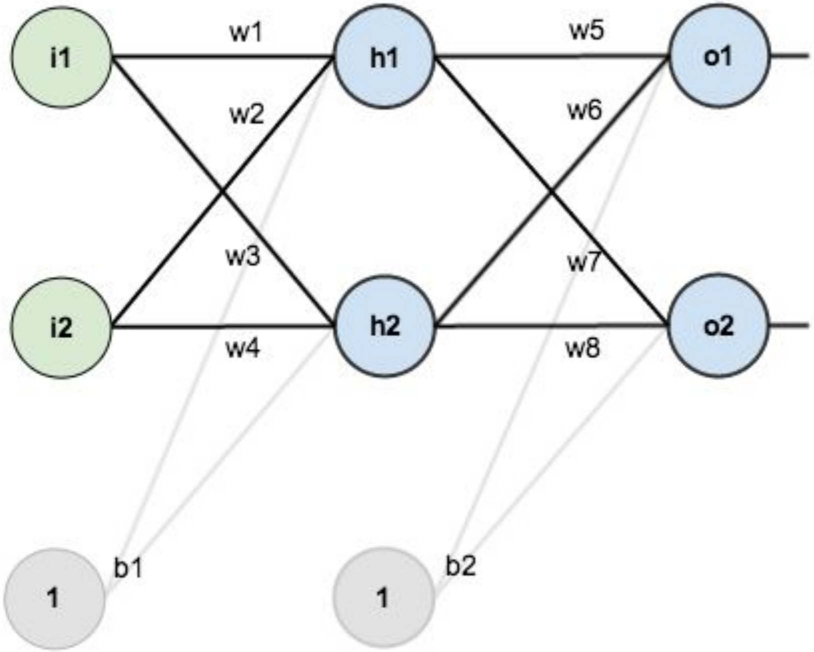
$$\Theta_{ij}^{(l)} := \Theta_{ij}^{(l)} - \alpha \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$$

}

Training a Neural Network

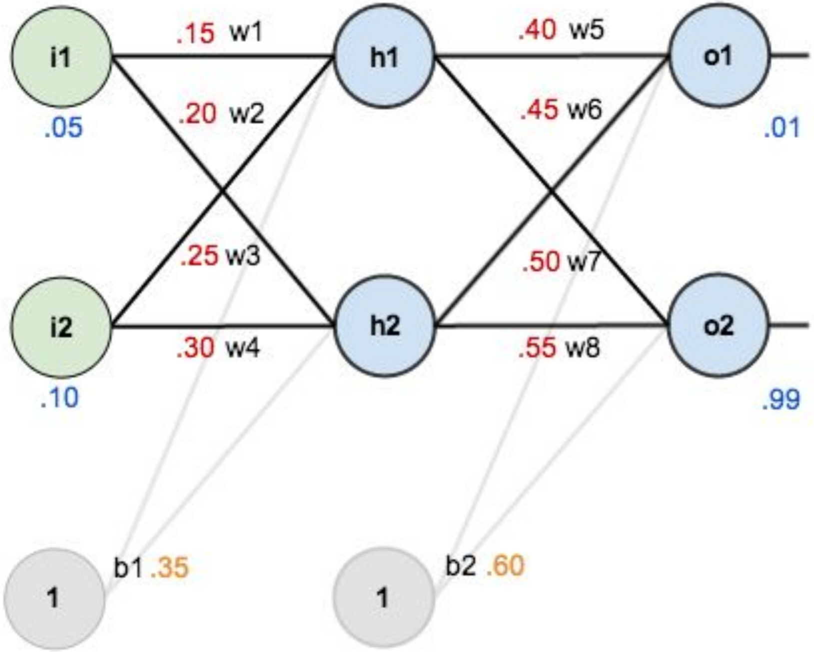


A Step by Step Backpropagation Example



https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Given inputs 0.05 and 0.10,
we want the neural network to output 0.01 and 0.99.



Initial weights, the biases, and training inputs/outputs.

The Forward Pass

Here's how we calculate the total net input for h_1:

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

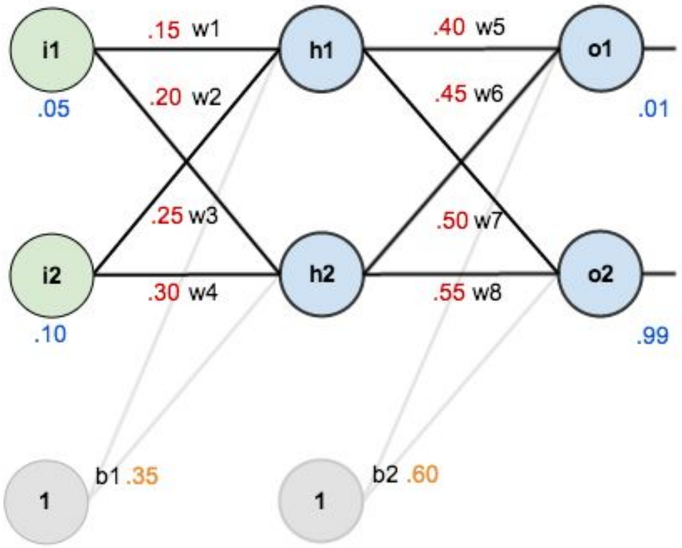
$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

We then squash it using the logistic function to get the output of h_1:

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

Carrying out the same process for h_2 we get:

$$out_{h2} = 0.596884378$$



The Forward Pass

We repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs.

Here's the output for o_1:

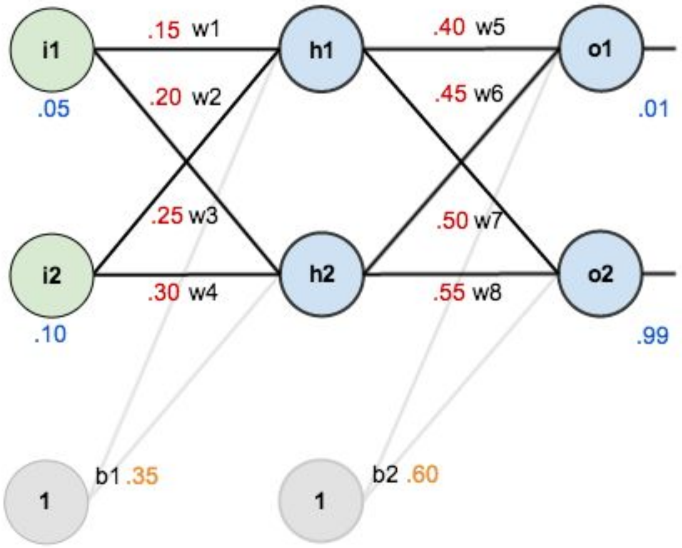
$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

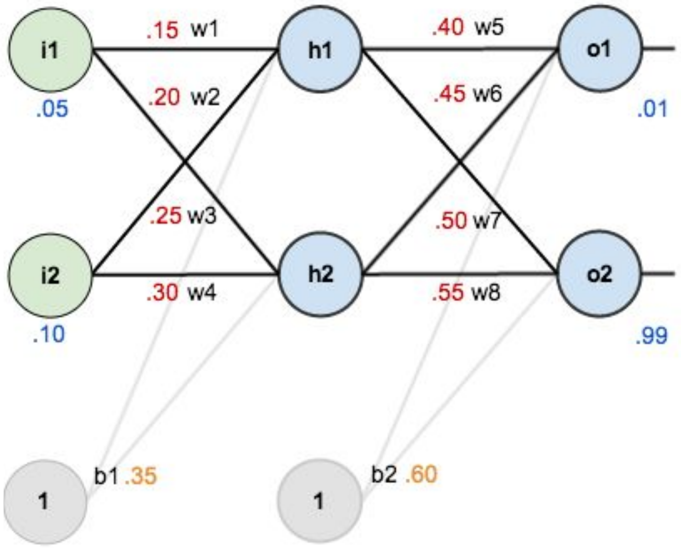
$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$

And carrying out the same process for o_2 we get:

$$out_{o2} = 0.772928465$$



The Error



We can now calculate the error for each output neuron using the squared error function and sum them to get the total error:

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

For example, the target output for o_1 is 0.01 but the neural network output 0.75136507, therefore its error is:

$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

Repeating this process for o_2 (remembering that the target is 0.99) we get:

$$E_{o2} = 0.023560026$$

The total error for the neural network is the sum of these errors:

$$E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$$

The Backprogation Pass

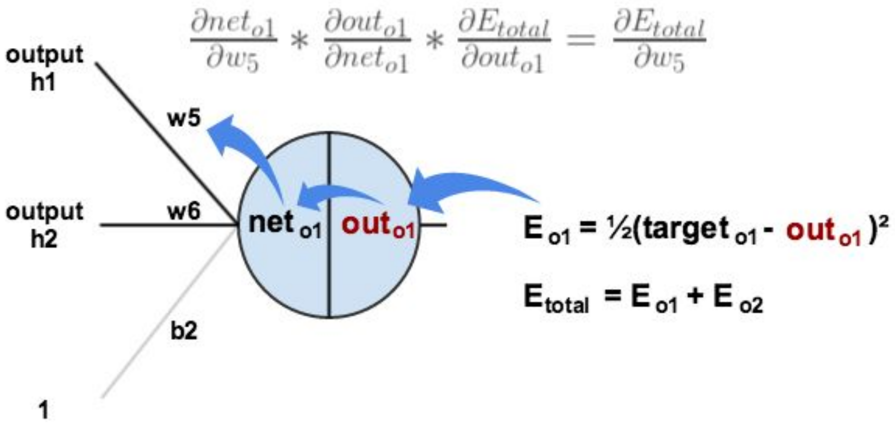
Output Layer

Consider w_5. We want to know how much a change in w_5 affects the total error, aka $\frac{\partial E_{total}}{\partial w_5}$.

> $\frac{\partial E_{total}}{\partial w_5}$ is read as "the partial derivative of E_{total} with respect to w_5". You can also say "the gradient with respect to w_5".

By applying the chain rule we know that:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$



The Backprogation Pass

We need to figure out each piece in this equation.

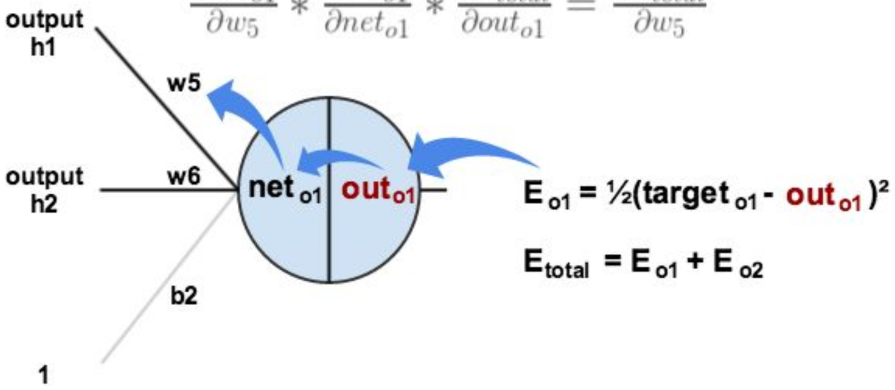
First, how much does the total error change with respect to the output?

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507$$

$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



The Backprogation Pass

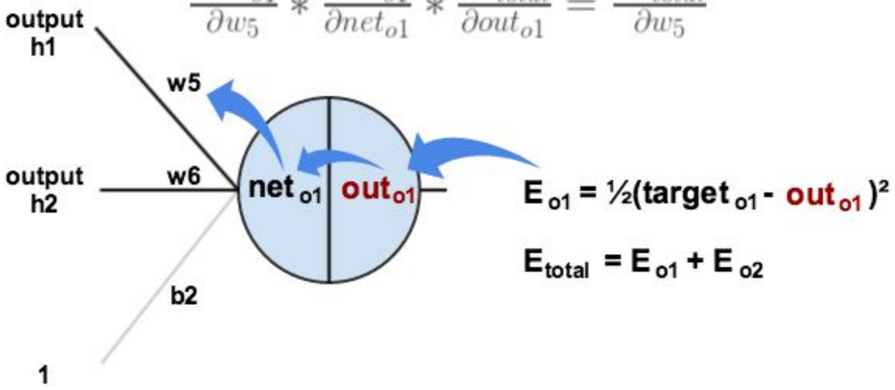
Next, how much does the output of o_1 change with respect to its total net input?

The partial derivative of the logistic function is the output multiplied by 1 minus the output:

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$$

$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



The Backprogation Pass

Finally, how much does the total net input of $o1$ change with respect to w_5?

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

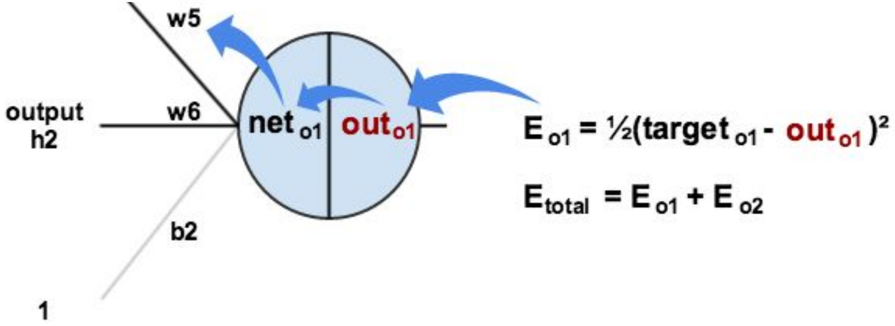
$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1} = 0.593269992$$

Putting it all together:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial w_5} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



The Backprogation Pass

You'll often see this calculation combined in the form of the delta rule:

$$\frac{\partial E_{total}}{\partial w_5} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1}) * out_{h1}$$

Alternatively, we have $\frac{\partial E_{total}}{\partial out_{o1}}$ and $\frac{\partial out_{o1}}{\partial net_{o1}}$ which can be written as $\frac{\partial E_{total}}{\partial net_{o1}}$, aka δ_{o1} (the Greek letter delta) aka the *node delta*. We can use this to rewrite the calculation above:

$$\delta_{o1} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = \frac{\partial E_{total}}{\partial net_{o1}}$$

$$\delta_{o1} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1})$$

Therefore:

$$\frac{\partial E_{total}}{\partial w_5} = \delta_{o1} out_{h1}$$

The Backprogation Pass

To decrease the error, we then subtract this value from the current weight (optionally multiplied by some learning rate, eta, which we'll set to 0.5):

$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

Some sources use α (alpha) to represent the learning rate, others use η (eta), and others even use ϵ (epsilon).

We can repeat this process to get the new weights w_6, w_7, and w_8:

$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

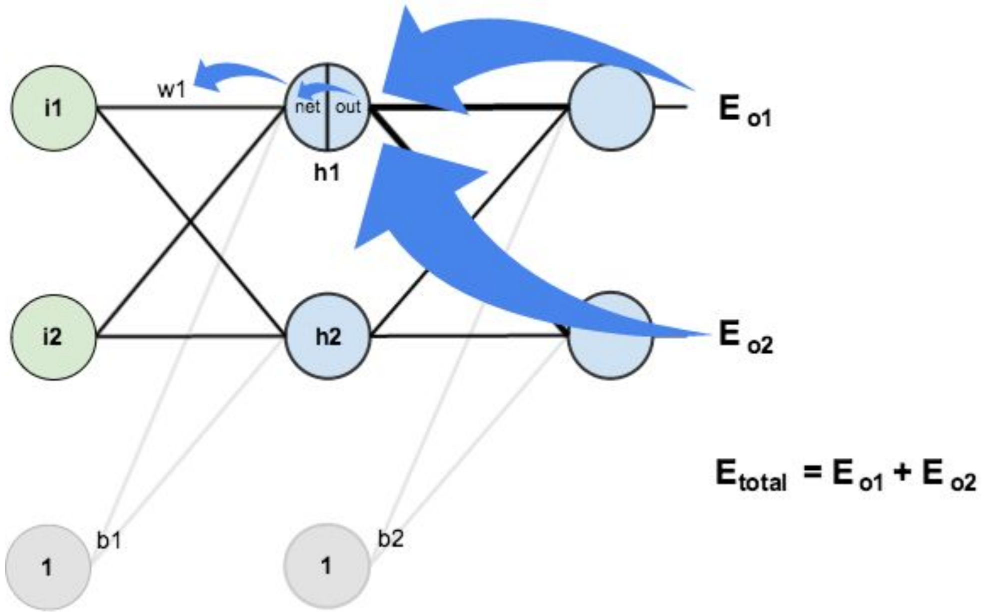
The Backprogation Pass

Hidden Layer

Next, we'll continue the backwards pass by calculating new values for w_1, w_2, w_3, and w_4.

Big picture, here's what we need to figure out:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$



The Backprogation Pass

We're going to use a similar process as we did for the output layer, but slightly different to account for the fact that the output of each hidden layer neuron contributes to the output (and therefore error) of multiple output neurons. We know that out_{h1} affects both out_{o1} and out_{o2} therefore the $\frac{\partial E_{total}}{\partial out_{h1}}$ needs to take into consideration its effect on the both output neurons:

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

Starting with $\frac{\partial E_{o1}}{\partial out_{h1}}$:

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}}$$

We can calculate $\frac{\partial E_{o1}}{\partial net_{o1}}$ using values we calculated earlier:

$$\frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = 0.74136507 * 0.186815602 = 0.138498562$$

The Backprogation Pass

And $\frac{\partial net_{o1}}{\partial out_{h1}}$ is equal to w_5:

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.40$$

Plugging them in:

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}} = 0.138498562 * 0.40 = 0.055399425$$

The Backprogation Pass

Following the same process for $\frac{\partial E_{o2}}{\partial out_{h1}}$, we get:

$$\frac{\partial E_{o2}}{\partial out_{h1}} = -0.019049119$$

Therefore:

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.055399425 + -0.019049119 = 0.036350306$$

Now that we have $\frac{\partial E_{total}}{\partial out_{h1}}$, we need to figure out $\frac{\partial out_{h1}}{\partial net_{h1}}$ and then $\frac{\partial net_{h1}}{\partial w}$ for each weight:

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}}$$

$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.59326999(1 - 0.59326999) = 0.241300709$$

The Backprogation Pass

We calculate the partial derivative of the total net input to h_1 with respect to w_1 the same as we did for the output neuron:

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$\frac{\partial net_{h1}}{\partial w_1} = i_1 = 0.05$$

Putting it all together:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

The Backprogation Pass

We can now update w_1:

$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.000438568 = 0.149780716$$

Repeating this for w_2, w_3, and w_4

$$w_2^+ = 0.19956143$$

$$w_3^+ = 0.24975114$$

$$w_4^+ = 0.29950229$$

Finally, we've updated all of our weights! When we fed forward the 0.05 and 0.1 inputs originally, the error on the network was 0.298371109. After this first round of backpropagation, the total error is now down to 0.291027924. It might not seem like much, but after repeating this process 10,000 times, for example, the error plummets to 0.0000351085. At this point, when we feed forward 0.05 and 0.1, the two outputs neurons generate 0.015912196 (vs 0.01 target) and 0.984065734 (vs 0.99 target).

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.

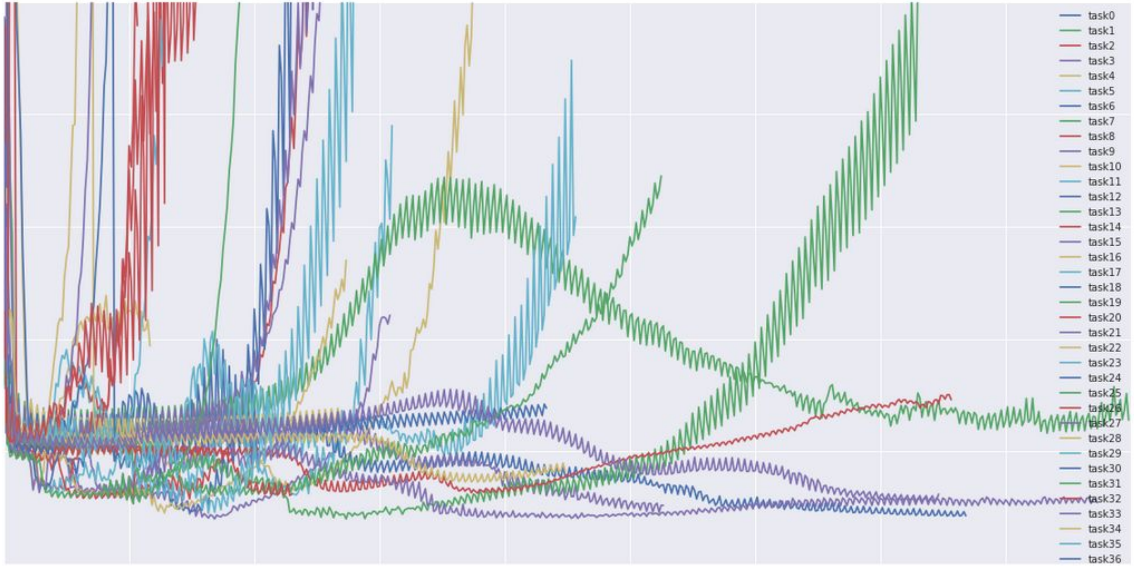
http://ruder.io/optimizing-gradient-descent/

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.

Blog
About
Papers
News
Newsletter

Deep Learning, NLP, ...



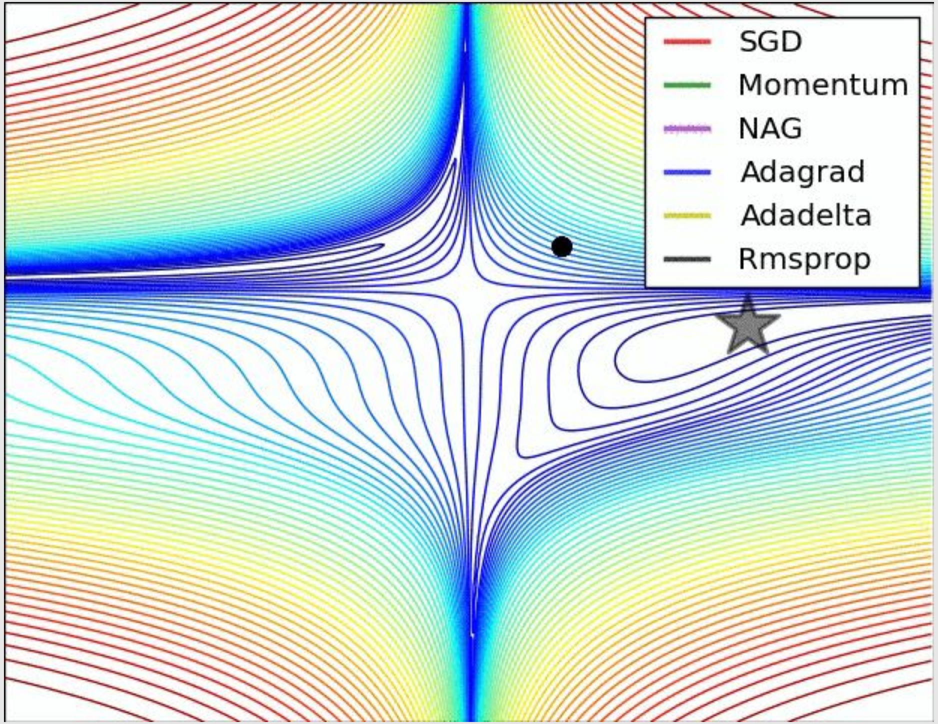
An overview of gradient descent optimization algorithms

- Momentum
- Nesterov
- Adagrad

- Adadelta
- RMSprop
- Adam

- AdaMax
- Nadam

- Batch gradient descent



Credit: Alec Radford.

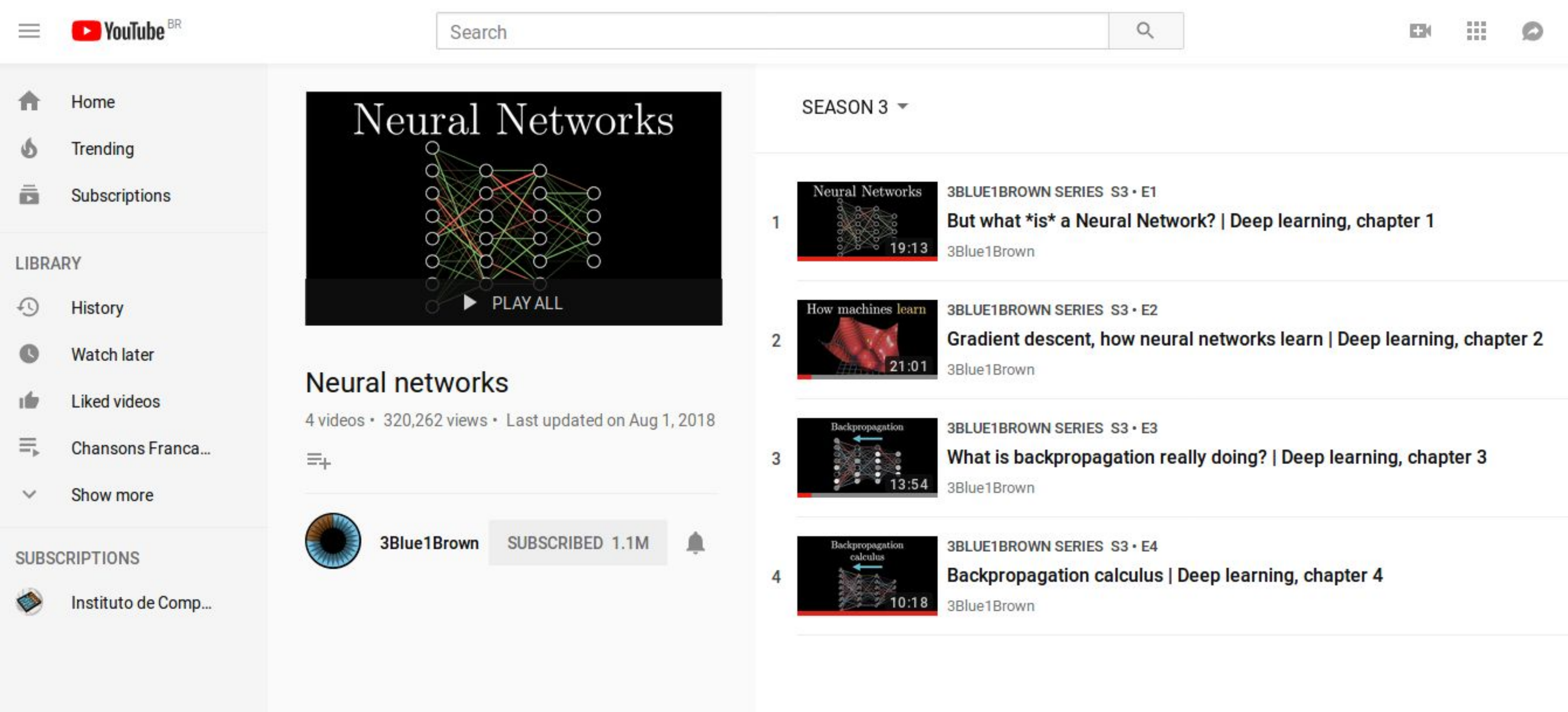
How many iterations are needed to converge?

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.

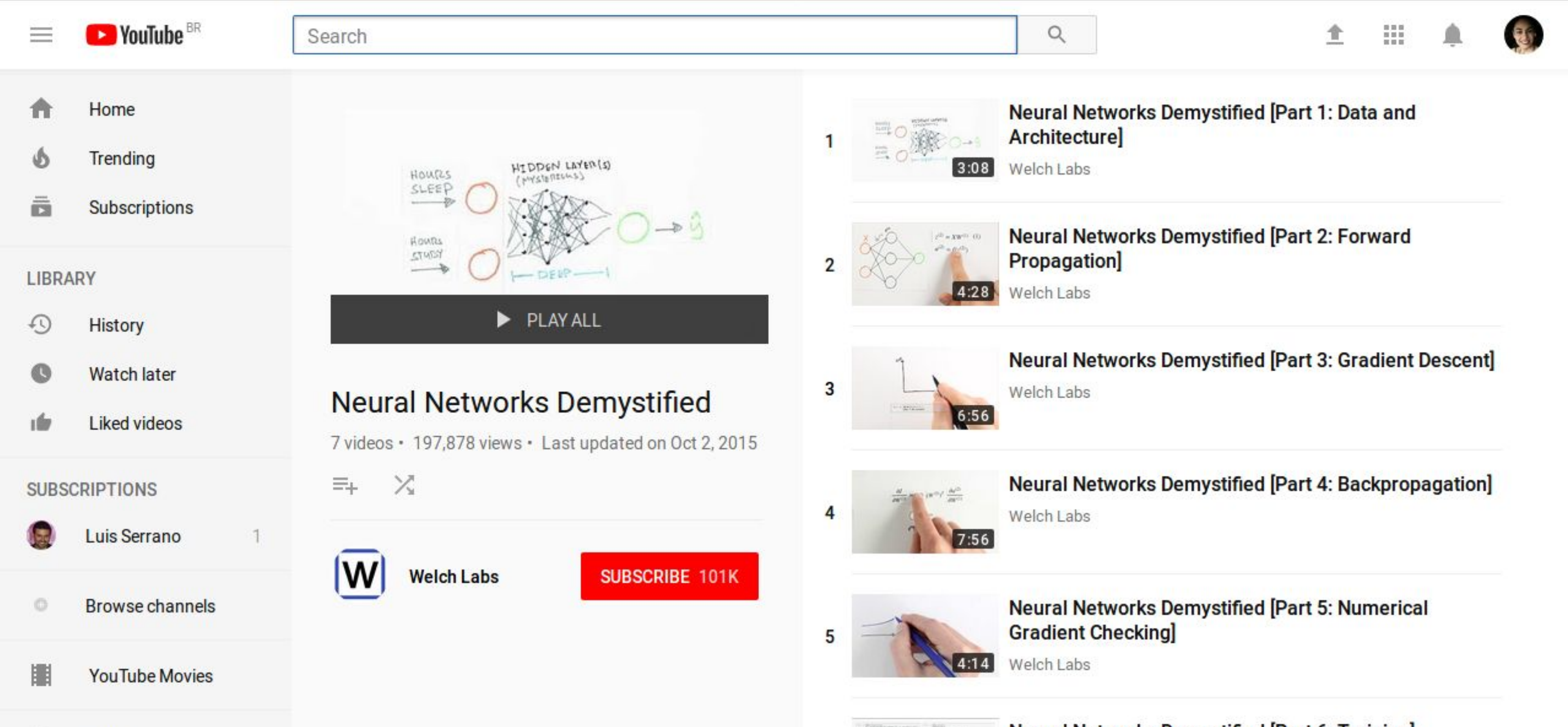
How many iterations are needed to converge?

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