Artificial Neural Networks

Machine Learning Techniques

Dr. Ashish Tendulkar

IIT Madras

Recall: Logistic regression

Logistic regression model

1. Linear combination

$$\boldsymbol{x} \rightarrow \boldsymbol{w}^T \boldsymbol{x}$$

2. Non-linear activation

$$\boldsymbol{w}^T \boldsymbol{x} \rightarrow \sigma(\boldsymbol{w}^T \boldsymbol{x})$$

Recall: Logistic regression

Basic unit of computation:

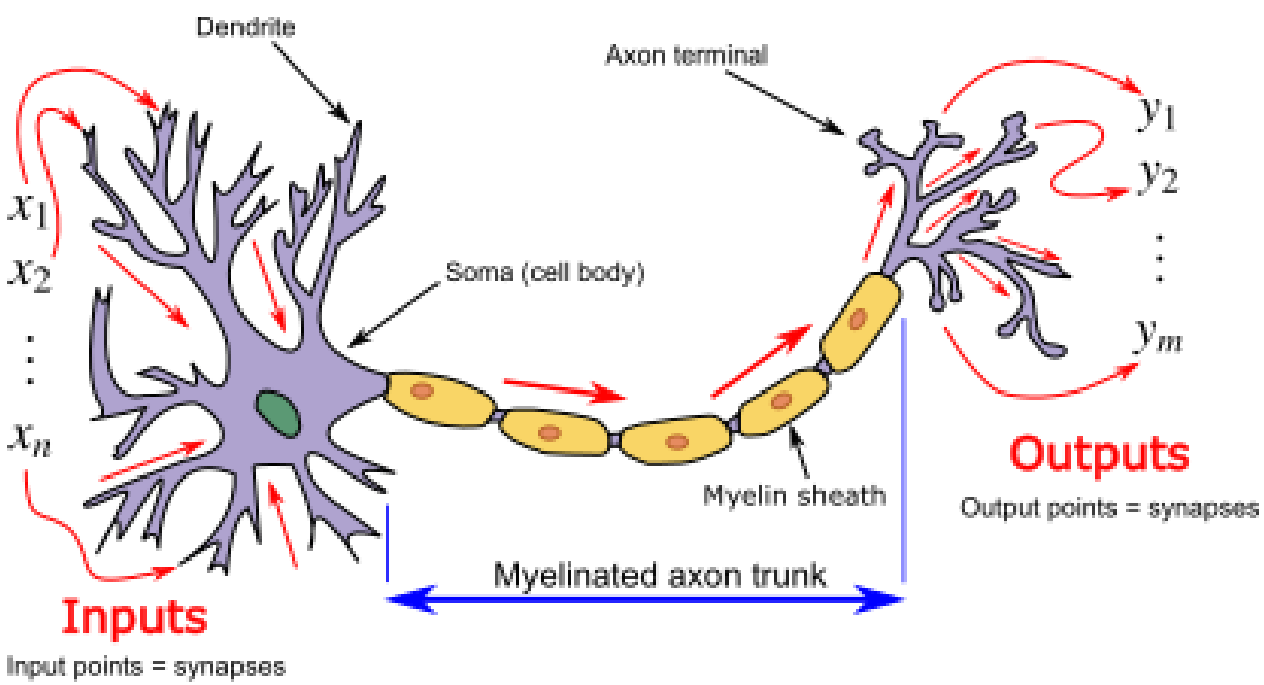
(1) Linear combination

(2) Non-linear activation

Where do neurons enter the picture?

Accept a number of **inputs**. Combine them linearly and simulate neuronal **activity**. See if they **fire**.

Biological Neuron

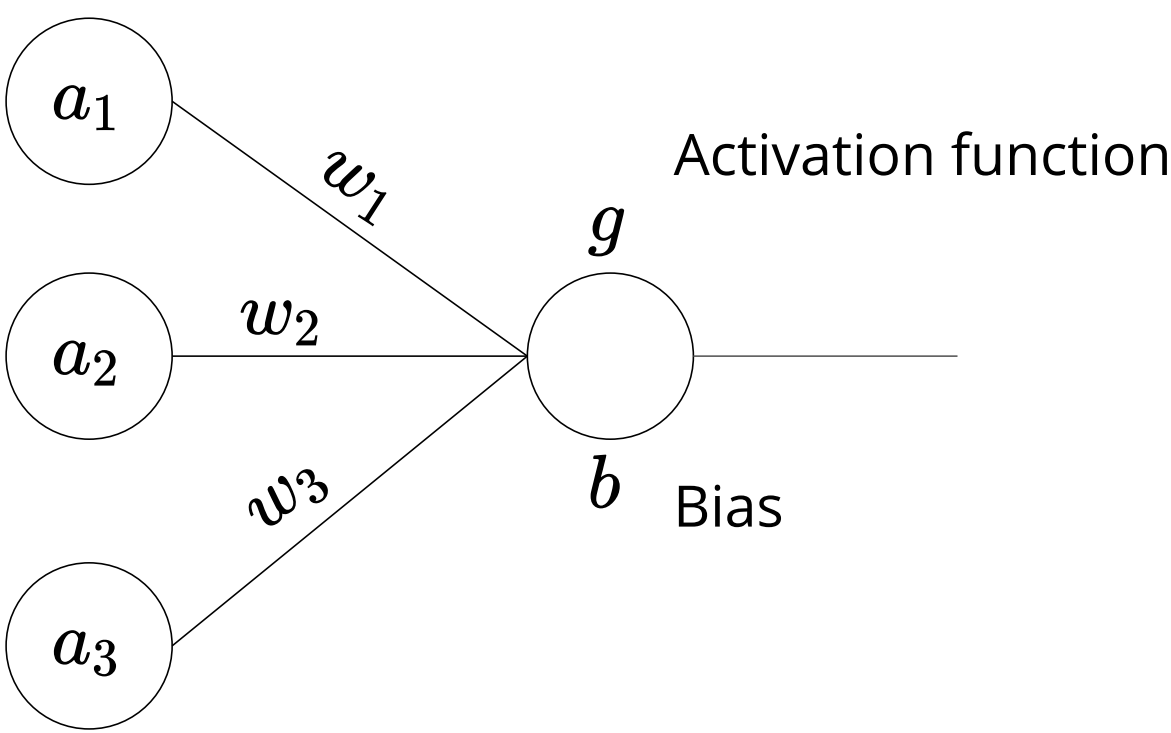


Source: https://commons.wikimedia.org/wiki/File:Neuron3.png

Artificial Neuron

Pre-activation $$z = w_1 a_1 + w_2 a_2 + w_3 a_3 + b$$

Activation $$a = g(z)$$



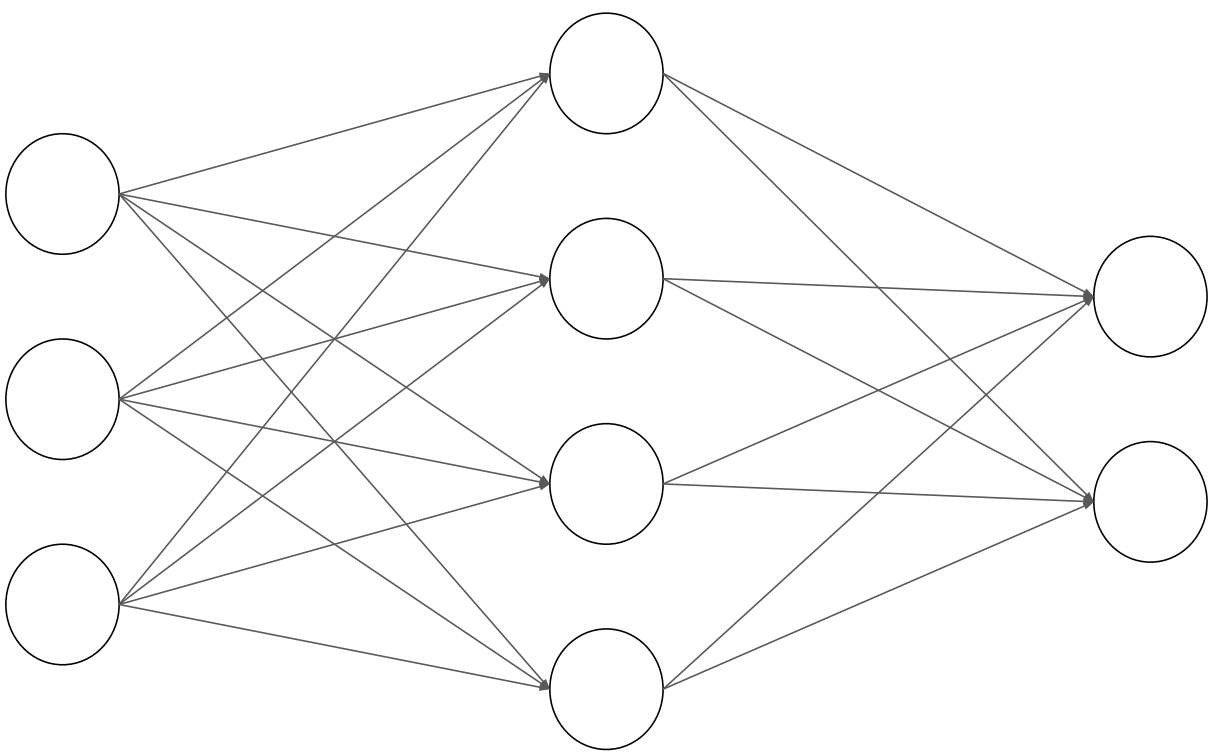
Network of Neurons

Source: https://youtu.be/HnIeAP--vWc

A single neuron doesn't have much capacity. When multiple neurons are connected together in a network, we have a powerful model.

Feedforward Networks

We will only look at feedforward networks where information flows in one direction: left to right, from inputs to outputs. The arrows will be dropped in subsequent images.



Problems

We will explore neural networks in the context of the following supervised learning problems:

(1) Regression

(2) Multi-class classification

Notation

Scalar: a

Vector: $\boldsymbol{a}$ $\begin{bmatrix} \cdots & a_i & \cdots \end{bmatrix}^T$

Matrix: $\boldsymbol{A}$ $\begin{bmatrix} \cdots & \cdots & \cdots \\ \vdots & A_{ij} & \vdots \\ \cdots & \cdots & \cdots \end{bmatrix}$

Data (regression)

Feature-matrix: $\boldsymbol{X}$

Labels: $\boldsymbol{y}$

Size: $n \times m$

Size: n

Number of data-points: n

Number of features: m

Data (classification)

Feature-matrix: $\boldsymbol{X}$

Size: $n \times m$

Number of data-points: n

Number of features: m

Labels: (one-hot) $\boldsymbol{Y}$

Size: $n \times k$

Number of classes: k

Neural Network

Black-box

$$h : \mathbb{R}^m \to \mathbb{R}^k$$

$$h(\boldsymbol{x}) = \hat{\boldsymbol{y}}$$

Neural Network

Black-box

$$h(\boldsymbol{x}) = \hat{\boldsymbol{y}}$$

Regression

$\hat{y}$ Predicted label
Some real number

Classification

$\hat{\boldsymbol{y}}$ Probability distribution over k classes

Neural Network

Black-box

$$h(\boldsymbol{x}) = \hat{\boldsymbol{y}}$$

Classification

Two-step process

(1) Probability distribution $\hat{\boldsymbol{y}}$

$$\downarrow$$

(2) Inference (predicted label) $\arg\max_{c} \quad \hat{y}_c$

Neural Network

Under the hood

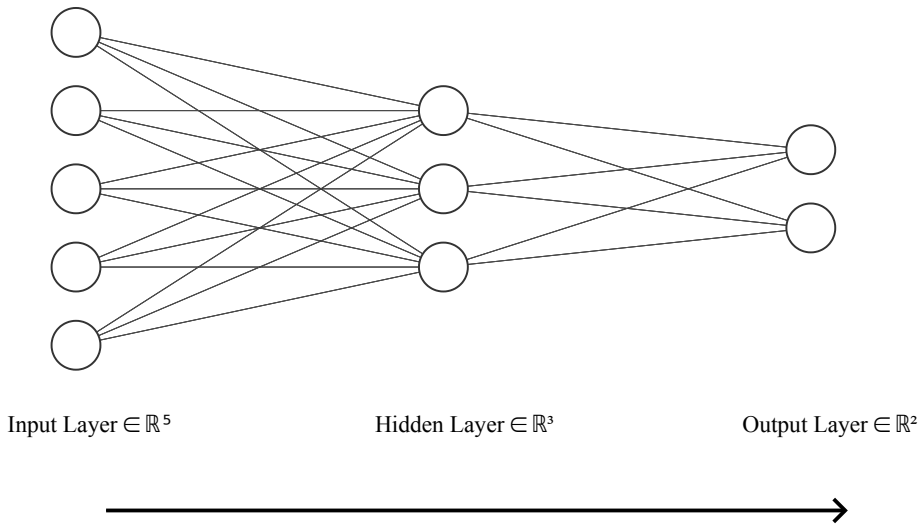
$$h(\boldsymbol{x}) = \hat{\boldsymbol{y}}$$

How does a neural network transform the input to the output?

To understand this, we have to peep under the hood of the model.

Forward Pass

$$h(\boldsymbol{x}) = \hat{\boldsymbol{y}}$$



(1) The network has a sequence of **layers**. The input to the network passes through these successive layers.

(2) Each layer transforms the input from the previous layer with the help of **weights** and **activation functions**.

(3) The entire process is termed a **forward pass**.

Neural Network

Components

To understand how a forward pass works, we need to study three components of a network:

(1) Layers

(2) Weights and biases

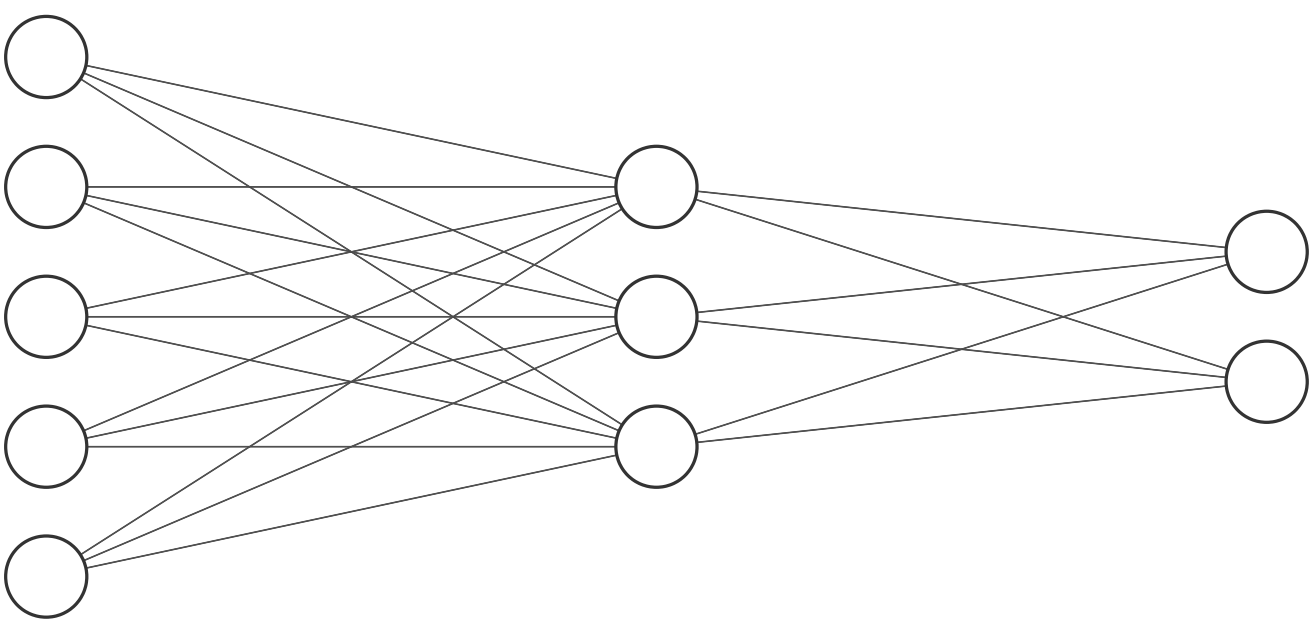
(3) Activation functions

Layers

Reference: http://alexlenail.me/NN-SVG/index.html

There are three kinds of layers that all networks have:

Input **Hidden** **Output**

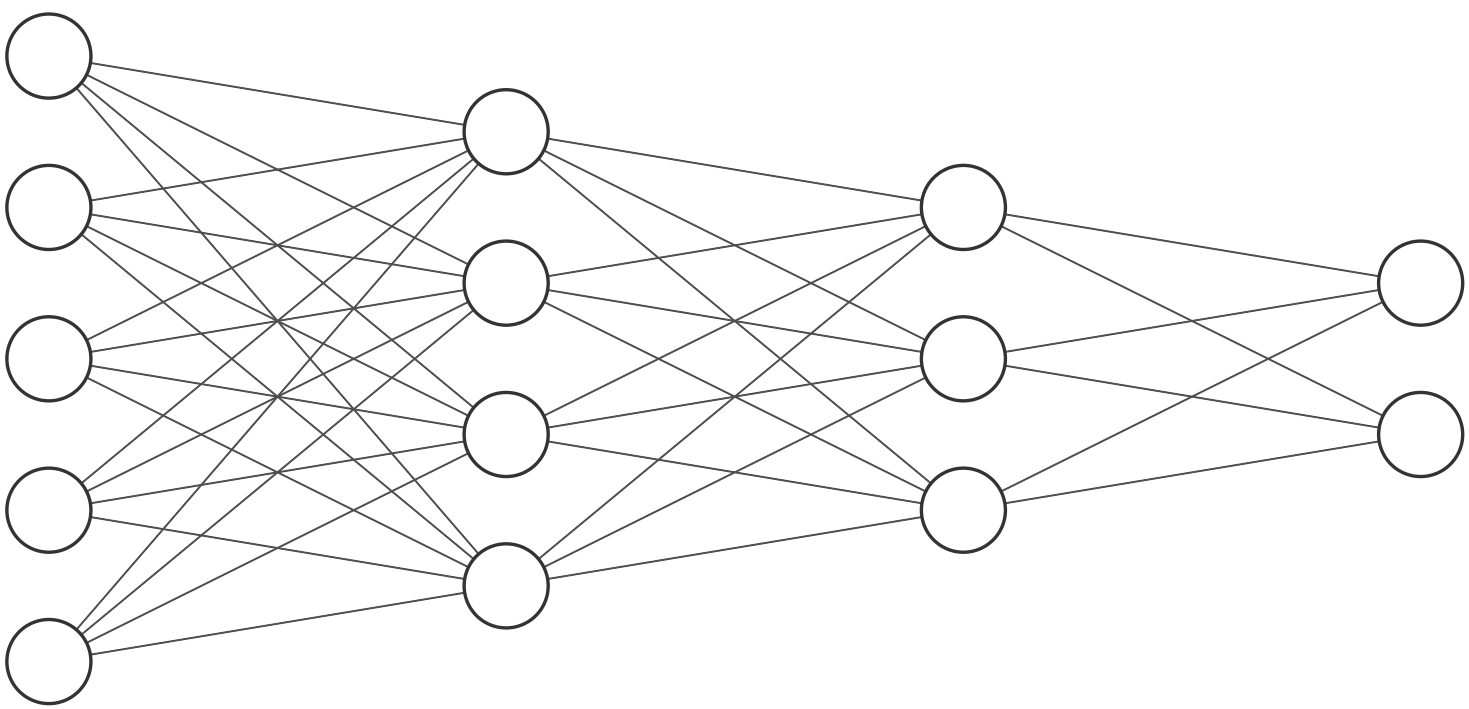


Input Layer $\in \mathbb{R}^5$ Hidden Layer $\in \mathbb{R}^3$ Output Layer $\in \mathbb{R}^2$

Layers

Reference: http://alexlenail.me/NN-SVG/index.html

Input and output layers are fixed. The number of hidden layers can vary. This network has two hidden layers.



Input Layer $\in \mathbb{R}^5$ Hidden Layer $\in \mathbb{R}^4$ Hidden Layer $\in \mathbb{R}^3$ Output Layer $\in \mathbb{R}^2$

Layers

The layers are indexed using l where, $0 \leq l \leq L$, and L is a hyperparameter.

$$
\begin{aligned}
&\text{Layer-0} : \text{Input layer} \\
&\text{Layer-1} : \text{First hidden layer} \\
&\qquad \vdots \\
&\text{Layer-}l \ : l^{\text{th}} \text{ hidden layer} \\
&\qquad \vdots \\
&\text{Layer-}L : \text{Output layer}
\end{aligned}
$$

Layers

The sizes of the layers are determined by the number of neurons in each layer and are denoted by:

$$S_l, \quad 0 \leq l \leq L$$

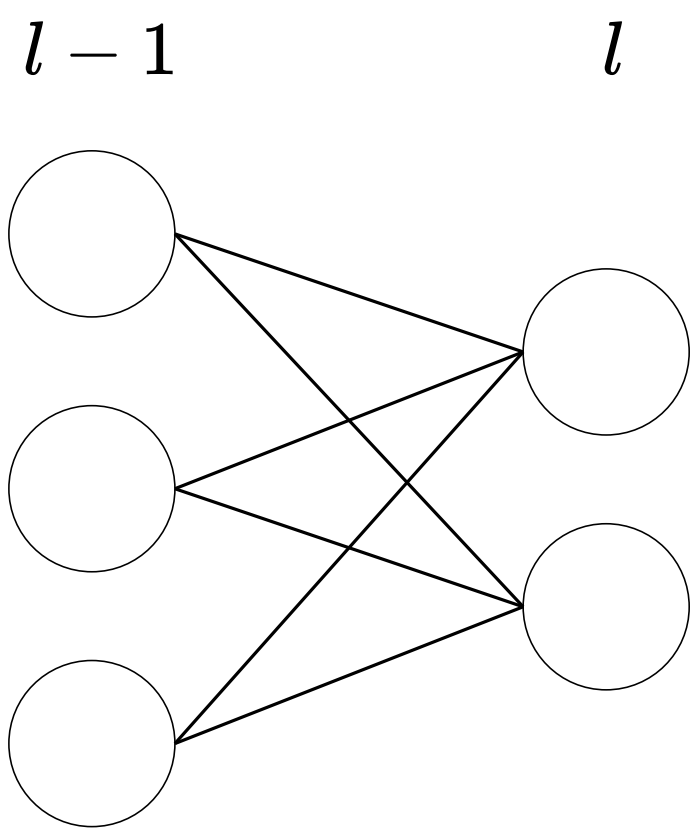
$S_0 = m$ Number of input- features

$S_1, \cdots, S_{L-1}$ Number of neurons in each hidden layer - hyperparameters

$$S_L = \begin{cases} 1, & \text{Regression} \\ k, & \text{Classification} \end{cases}$$

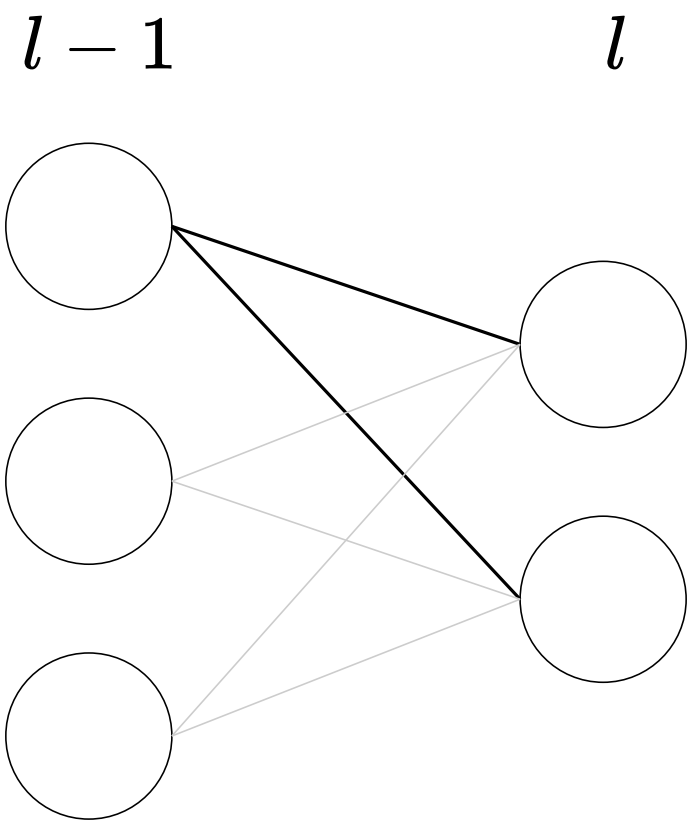
Weights

Weights determine the **importance** of the connections or **edges** between neurons.



Weights

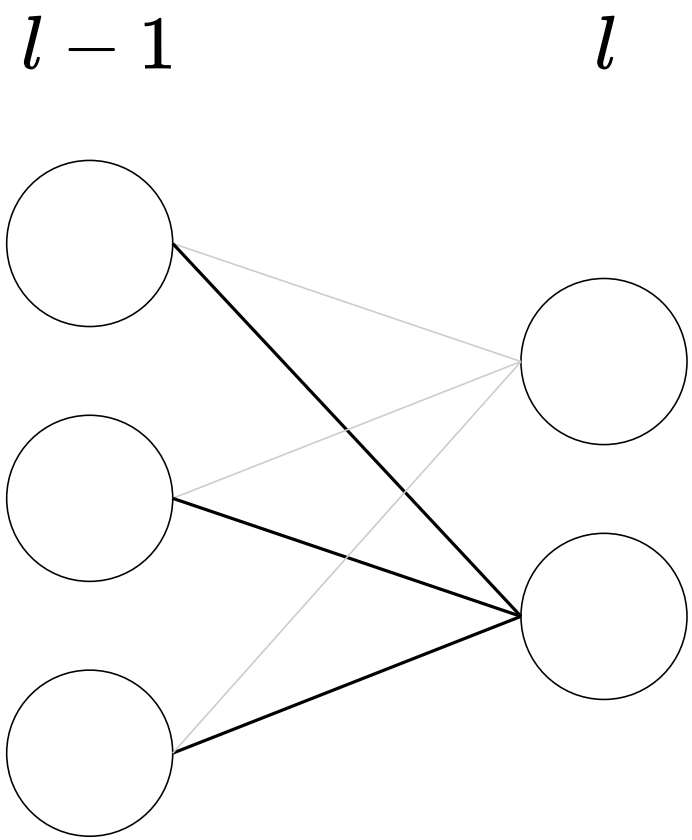
Outgoing edges from layer $l-1$



Each neuron in layer $l-1$ has S_l outgoing edges

Weights

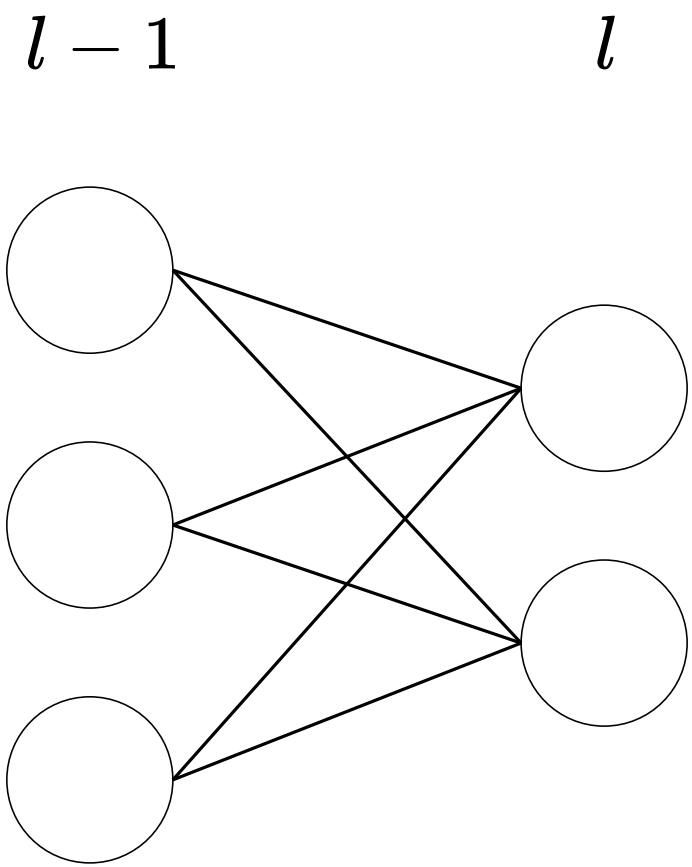
Incoming edges to layer l



Each neuron in layer l
has S_{l-1} incoming edges

Weights

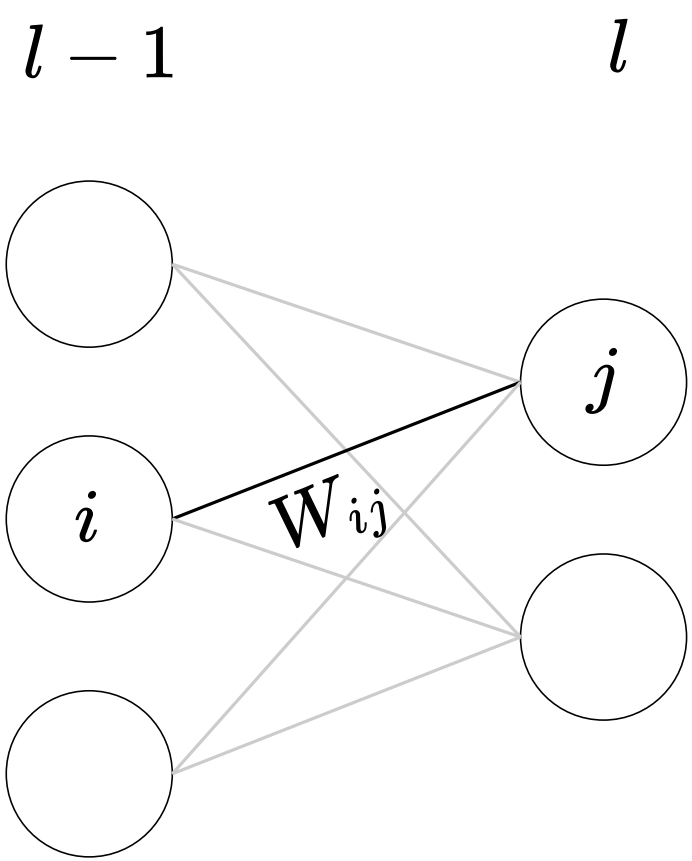
Each edge has a weight associated with it.



At layer l the total number of weights is equal to $S_{l-1} \times S_l$

Weights

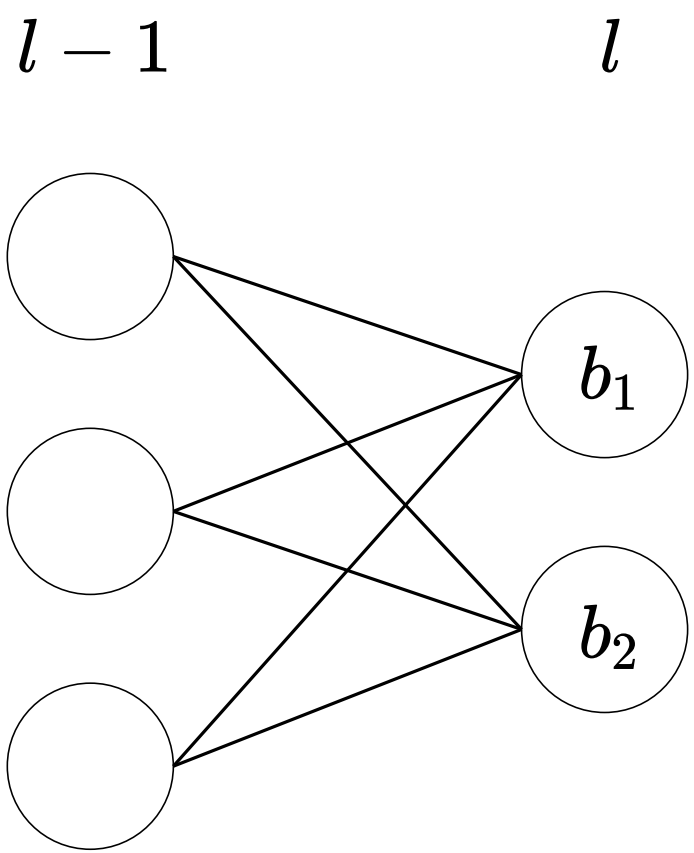
These weights can be neatly packed into a matrix $\boldsymbol{W_l}$ of size $S_{l-1} \times S_l$



$$\boldsymbol{W_l} = \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \\ W_{31} & W_{32} \end{bmatrix}$$

Biases

Each neuron at layer l has a bias associated with it.
This results in a vector $\boldsymbol{b_l}$ of biases at layer l.



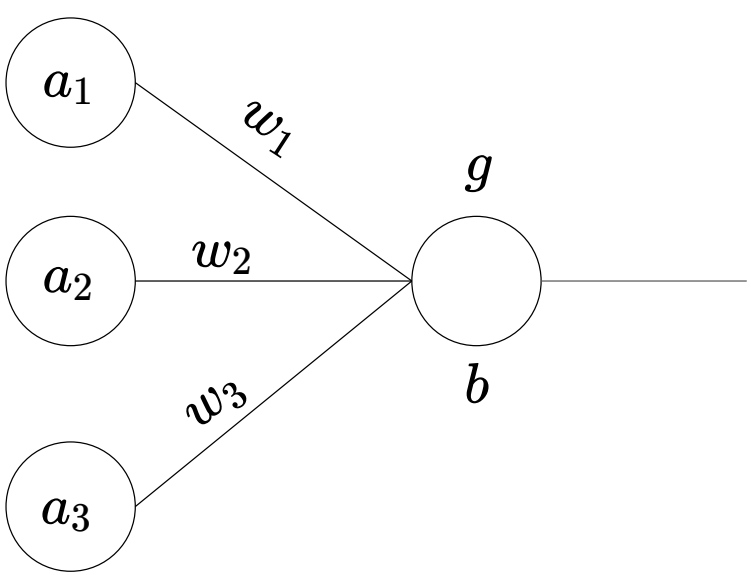
$$\boldsymbol{b_l} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Single neuron

Recall what happens in the case of a single neuron

Pre-activation $z = w_1 a_1 + w_2 a_2 + w_3 a_3 + b$

Activation $a = g(z)$



Pre-activations

(1) Linear combination of inputs

Activations

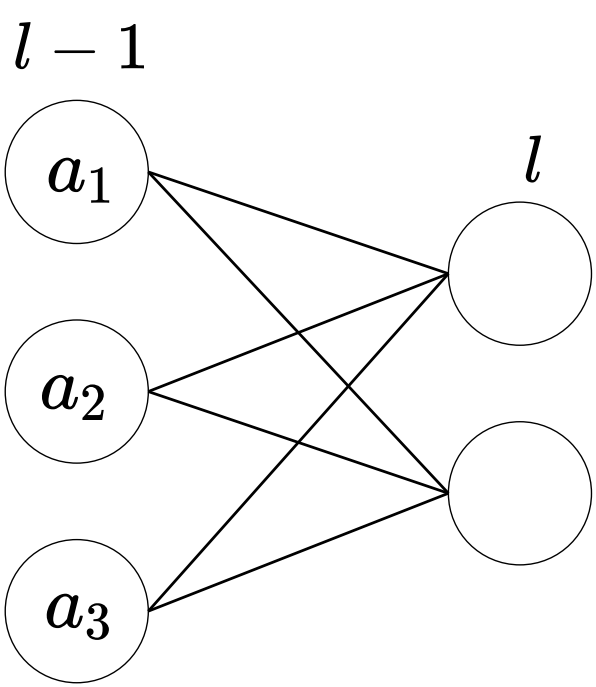
(2) Non-linear transformation

Pre-activations (vector form)

Vector of input activations at layer l: $\boldsymbol{a_{l-1}}$

Vector of pre-activations at layer l: $\boldsymbol{z_l}$

$$\begin{bmatrix} z_1 & z_2 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix} \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \\ W_{31} & W_{32} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 \end{bmatrix}$$

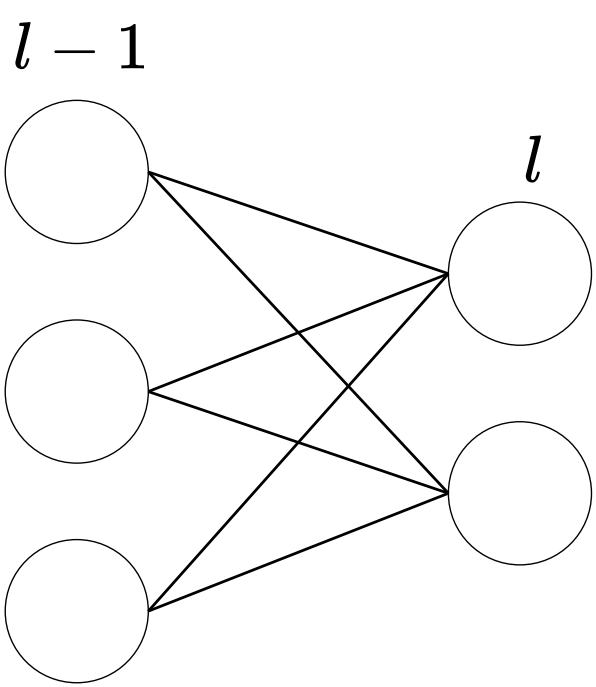


$$\boldsymbol{z_l}^T = \boldsymbol{a_{l-1}}^T \boldsymbol{W_l} + \boldsymbol{b_l}^T$$

Pre-activations (matrix form)

Matrix of input activations at layer l: $\boldsymbol{A_{l-1}}$

Matrix of pre-activations at layer l: $\boldsymbol{Z_l}$



$$\boldsymbol{Z_l} = \boldsymbol{A_{l-1}}\boldsymbol{W_l} + \boldsymbol{b_l}$$

$n \times S_l$

S_l

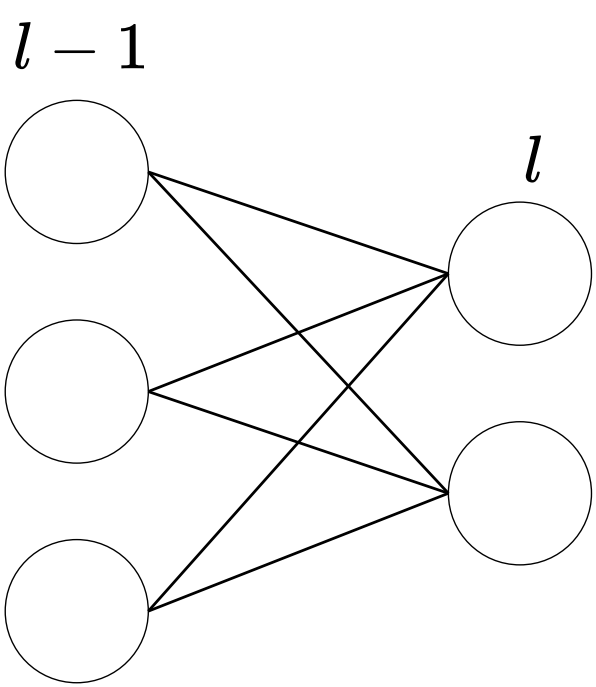
$n \times S_{l-1}$

$S_{l-1} \times S_l$

Activations (matrix form)

Matrix of pre-activations at layer l : $\boldsymbol{Z_l}$

Matrix of activations at layer l : $\boldsymbol{A_l}$



$$\boldsymbol{A_l} = g(\boldsymbol{Z_l})$$

$$n \times S_l$$

In the hidden layers, g is applied element-wise

Activation functions

Why is an activation function required?

Why should it be non-linear?

Activation functions

Assume that there is no activation function for any layer, then the output will be a sequence of matrix products (ignore the bias for now):

$$\hat{\boldsymbol{Y}} = \boldsymbol{X}\boldsymbol{W}_{\mathbf{1}} \cdots \boldsymbol{W}_{\boldsymbol{L}} = \boldsymbol{X}\boldsymbol{W}$$

The network degenerates into a simple linear model!

Activation functions (Hidden layers)

We will look at three activation functions that are commonly used in the hidden layers:

Sigmoid

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

Tanh

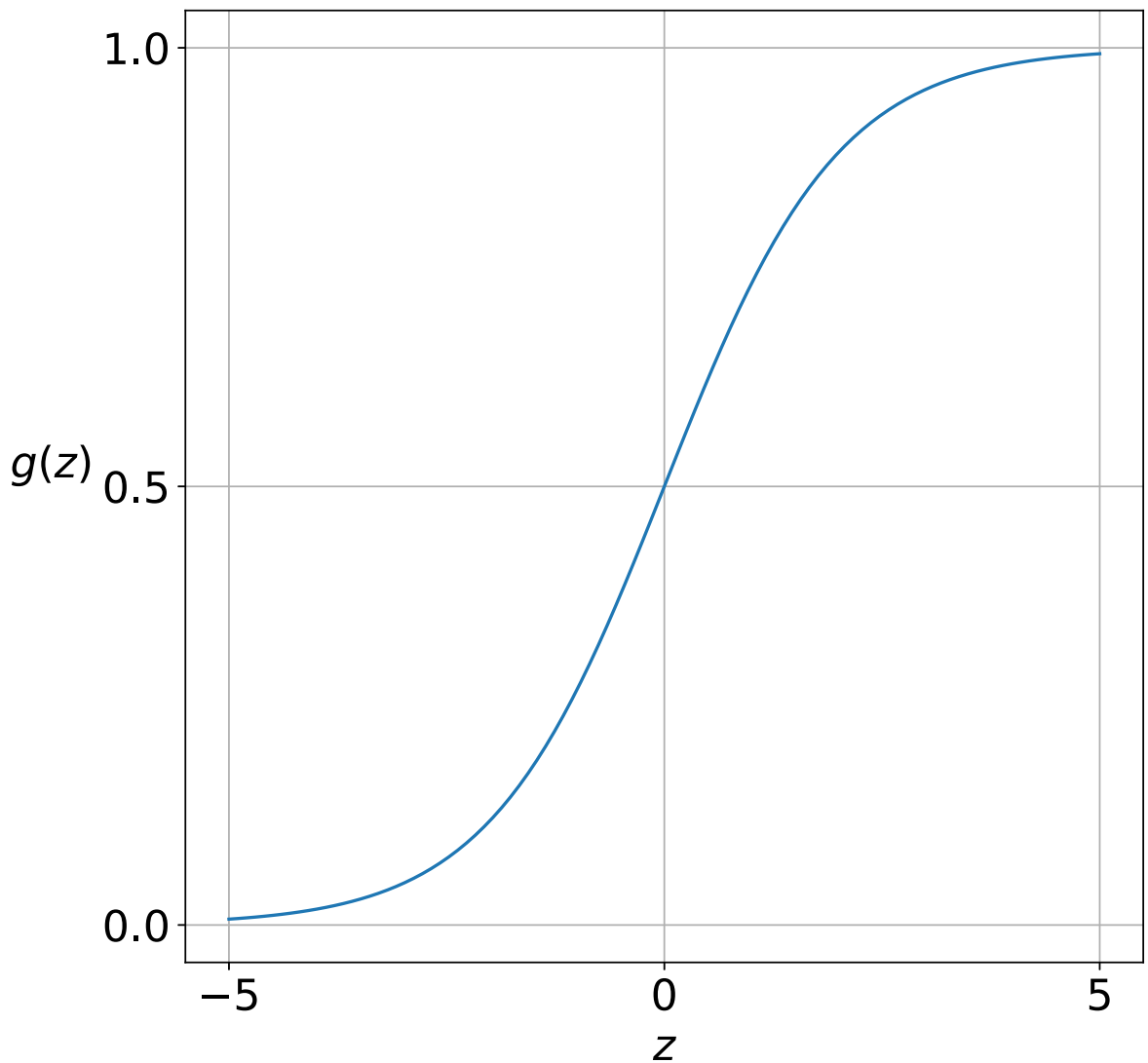
$$g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

ReLU

$$g(z) = \begin{cases} z & z \geq 0 \\ 0 & z < 0 \end{cases}$$

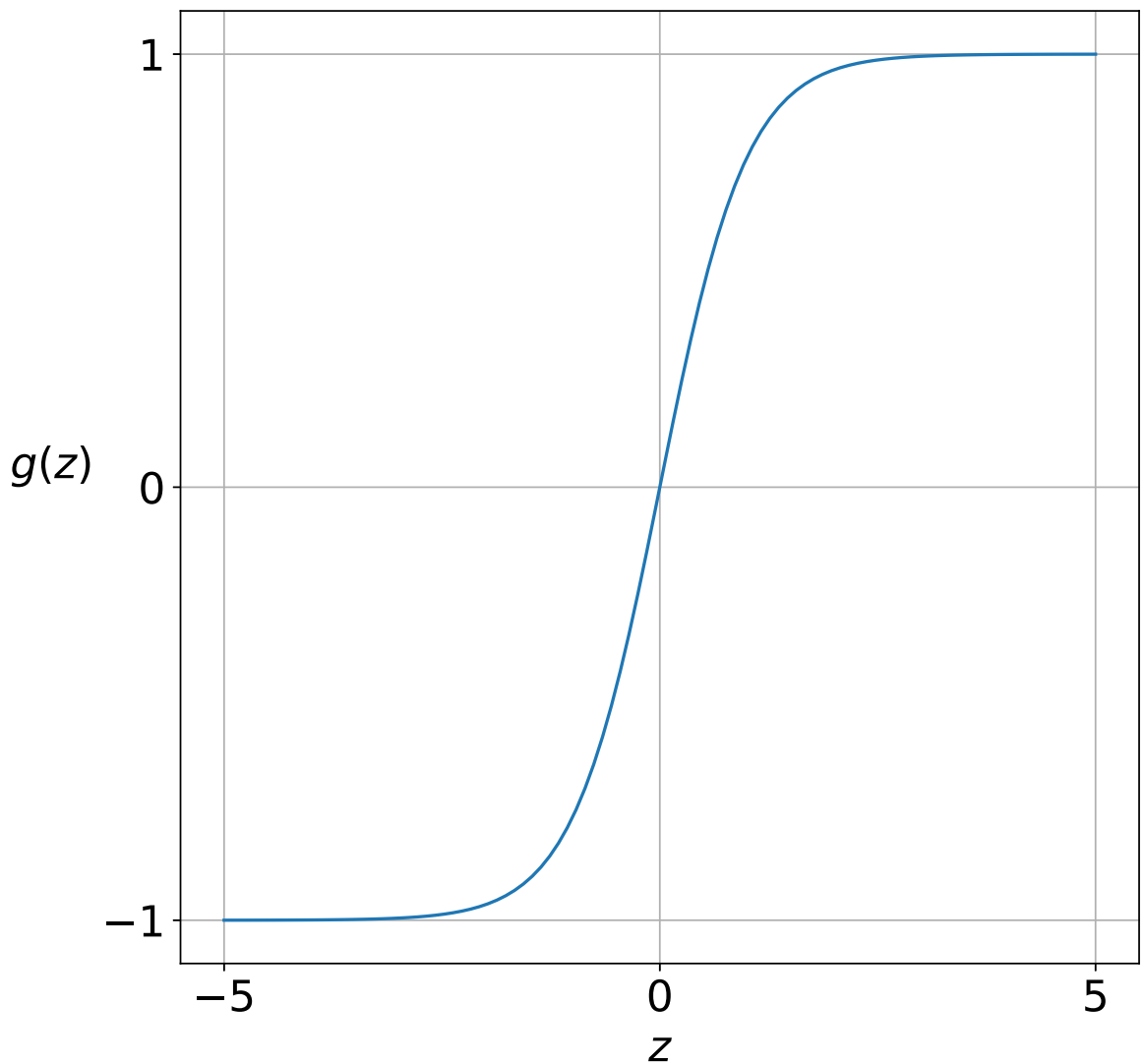
Sigmoid

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



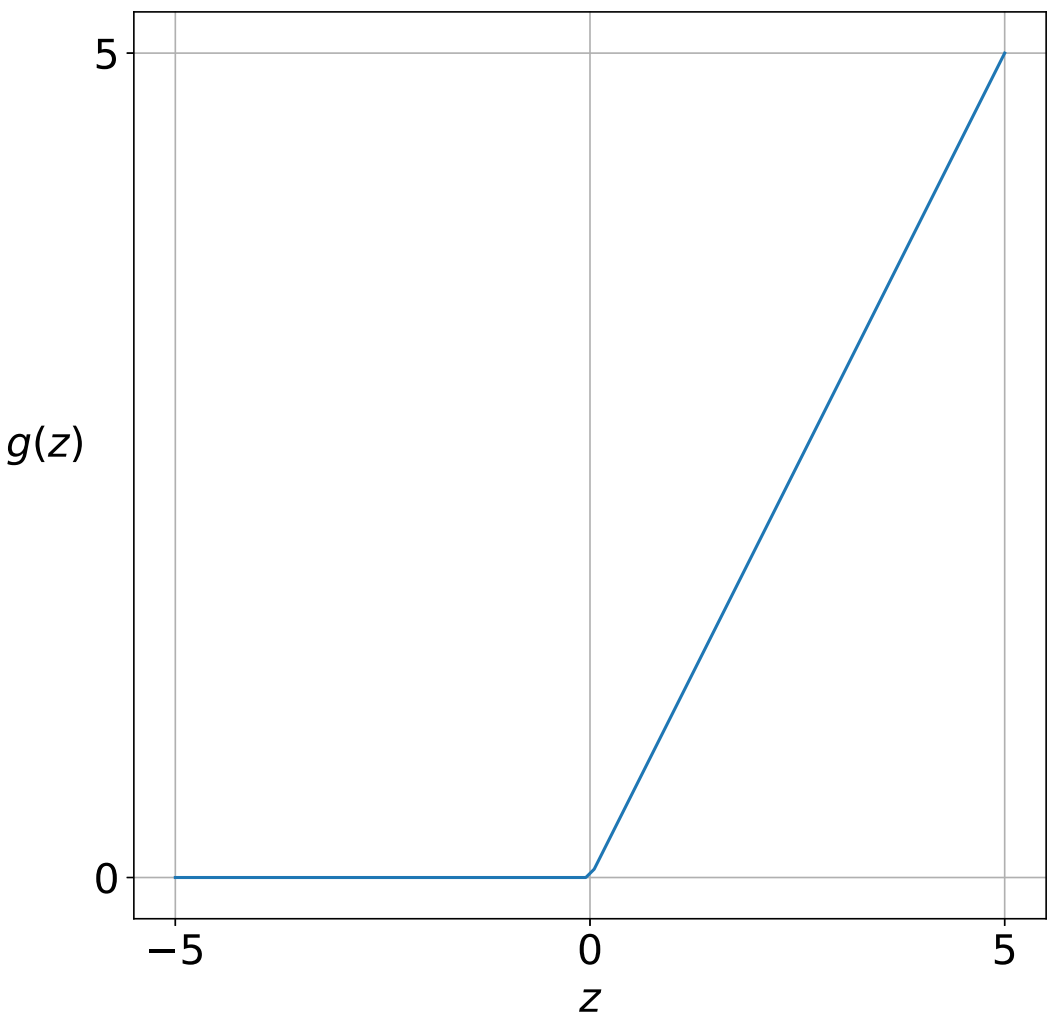
Tanh

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



ReLU

$$g(z) = \begin{cases} z & z \geq 0 \\ 0 & z < 0 \end{cases}$$



ReLU:
Rectified
Linear Unit

Activation Functions (Hidden layers)

Choice of activation function for the hidden layers is a hyperparameter.

In practice:

(1) Both Sigmoid and Tanh are found to suffer from the problem of vanishing gradients.

(2) ReLU is a good choice for networks with a large number of hidden layers.

Activation Functions (Output layer)

The activation function at the output layer depends on the problem being solved.

Regression

$$g(z) = z$$

Identity

Classification

$$g(\boldsymbol{z}) = \begin{bmatrix} \cdots & \dfrac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} & \cdots \end{bmatrix}$$

Softmax

Forward Pass (regression)

Iterative algorithm to compute $\hat{\boldsymbol{y}}$ given $\boldsymbol{X}$

```
Initialize:
```
$$\boldsymbol{A}_0 = \boldsymbol{X}$$

```
for l = 1 to l = L:
```
$$\boldsymbol{Z}_l = \boldsymbol{A}_{l-1}\boldsymbol{W}_l + \boldsymbol{b}_l$$

$$\boldsymbol{A}_l = g(\boldsymbol{Z}_l)$$

```
Assign:
```
$$\hat{\boldsymbol{y}} = \boldsymbol{A}_L$$

```
Return:
```
$$\hat{\boldsymbol{y}}$$

Forward Pass (regression)

Iterative algorithm to compute $\hat{\boldsymbol{y}}$ given $\boldsymbol{X}$

```
Initialize:
```
$$\boldsymbol{A}_0 = \boldsymbol{X}$$

```
for l = 1 to l = L:
```
$$\boldsymbol{Z}_l = \boldsymbol{A}_{l-1}\boldsymbol{W}_l + \boldsymbol{b}_l$$

$$\boldsymbol{A}_l = g(\boldsymbol{Z}_l)$$

```
Assign:
```
$$\hat{\boldsymbol{y}} = \boldsymbol{A}_L$$

```
Return:
```
$$\hat{\boldsymbol{y}}$$

Notes

(1) Output layer has 1 neuron

(2) g is identity for the output layer

(3) $\boldsymbol{X}$ has size $n \times m$

(4) $\hat{\boldsymbol{y}}$ has size n

(5) $\boldsymbol{A}_L$ is a matrix of size $n \times 1$

Forward Pass (classification)

Iterative algorithm to compute $\hat{\boldsymbol{Y}}$ given $\boldsymbol{X}$

```
Initialize:
```
$$\boldsymbol{A}_0 = \boldsymbol{X}$$

```
for l = 1 to l = L:
```
$$\boldsymbol{Z}_l = \boldsymbol{A}_{l-1}\boldsymbol{W}_l + \boldsymbol{b}_l$$

$$\boldsymbol{A}_l = g(\boldsymbol{Z}_l)$$

```
Assign:
```
$$\hat{\boldsymbol{Y}} = \boldsymbol{A}_L$$

```
Return:
```
$$\hat{\boldsymbol{Y}}$$

Forward Pass (classification)

Iterative algorithm to compute $\hat{\boldsymbol{Y}}$ given $\boldsymbol{X}$

```
Initialize:
```
$$\boldsymbol{A_0} = \boldsymbol{X}$$

```
for l = 1 to l = L:
```
$$\boldsymbol{Z_l} = \boldsymbol{A_{l-1}}\boldsymbol{W_l} + \boldsymbol{b_l}$$

$$\boldsymbol{A_l} = g(\boldsymbol{Z_l})$$

```
Assign:
```
$$\hat{\boldsymbol{Y}} = \boldsymbol{A_L}$$

```
Return:
```
$$\hat{\boldsymbol{Y}}$$

Notes

(1) Output layer has k neurons

(2) g is Softmax for the output layer

(3) $\boldsymbol{X}$ has size $n \times m$

(4) $\hat{\boldsymbol{Y}}$ has size $n \times k$

Loss (regression)

- $\boldsymbol{y}$ is a vector of target label for n data-points
- $\hat{\boldsymbol{y}}$ is the output of the network and corresponds to the predicted labels.

$$L(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{2} \cdot (\hat{\boldsymbol{y}} - \boldsymbol{y})^T (\hat{\boldsymbol{y}} - \boldsymbol{y})$$

Squared-Error Loss

Loss (regression)

- $\boldsymbol{y}$ is a vector of target label for n data-points
- $\hat{\boldsymbol{y}}$ is the output of the network and corresponds to the predicted labels.

$$L(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{2} \cdot (\hat{\boldsymbol{y}} - \boldsymbol{y})^T (\hat{\boldsymbol{y}} - \boldsymbol{y})$$

Squared-Error Loss

In Numpy: `L = 0.5 * np.sum((y_hat - y) * (y_hat - y))`

Loss (classification)

- $\boldsymbol{Y}$ is a matrix of one-hot labels for n data-points.
- $\hat{\boldsymbol{Y}}$ is the output of the network and corresponds to the predicted probabilities.

$$L(\boldsymbol{Y}, \hat{\boldsymbol{Y}}) = -\mathbf{1}_n^T \left(\boldsymbol{Y} \odot \log \hat{\boldsymbol{Y}} \right) \mathbf{1}_k$$

Categorical Cross-Entropy Loss

Loss (classification)

$$L(\boldsymbol{Y}, \hat{\boldsymbol{Y}}) = -\mathbf{1}_n^T \left(\boldsymbol{Y} \odot \log \hat{\boldsymbol{Y}} \right) \mathbf{1}_k$$

This equation can be understood as follows:

- $\mathbf{1}_n$ and $\mathbf{1}_k$ are vectors of ones of sizes n and k
- If $\boldsymbol{M}$ is a matrix of size $n \times k$ then $\mathbf{1}_n^T M \mathbf{1}_k$ is the sum of all elements in the matrix.
- $\odot$ is element-wise product

In Numpy:

```
L = -np.sum(Y * np.log(Y_hat))
```

Optimization (Gradient Descent)

- We will turn to gradient descent for minimizing the loss.
- We need to compute the gradients of the loss with respect to the weights and biases. Specifically:

$$\frac{\partial L}{\partial W_{lij}} \qquad \frac{\partial L}{\partial b_{lj}}$$

W_{lij}	Weight connecting neuron i in layer $l-1$ to neuron j in layer l
b_{lj}	Bias of neuron j in layer l

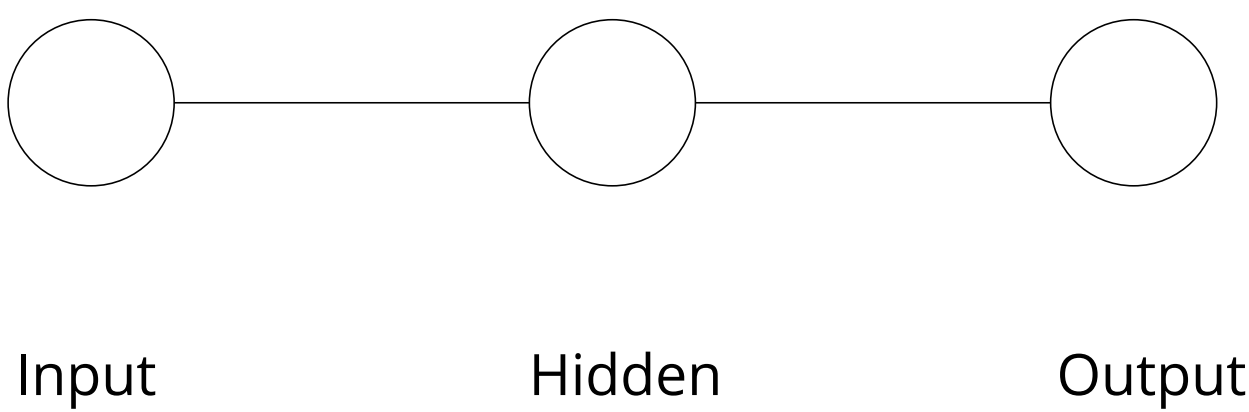
Backpropagation

- Backpropagation is an efficient algorithm to compute the gradients of the loss w.r.t the weights.
- At the heart of the algorithm is the chain rule for derivatives.

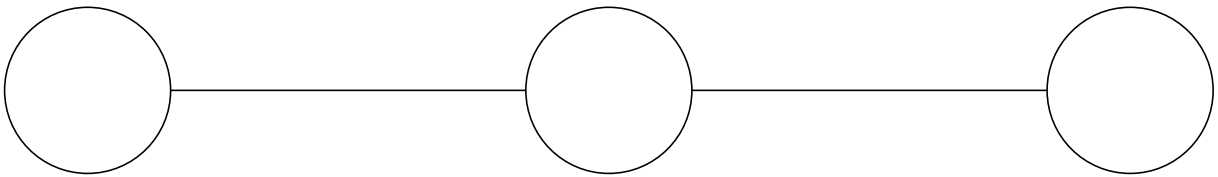
$$\frac{\partial L}{\partial W_{lij}} \qquad \frac{\partial L}{\partial b_{lj}}$$

Backpropagation

To understand how to compute gradients in a network, let us consider this absurdly simple network for a regression problem.

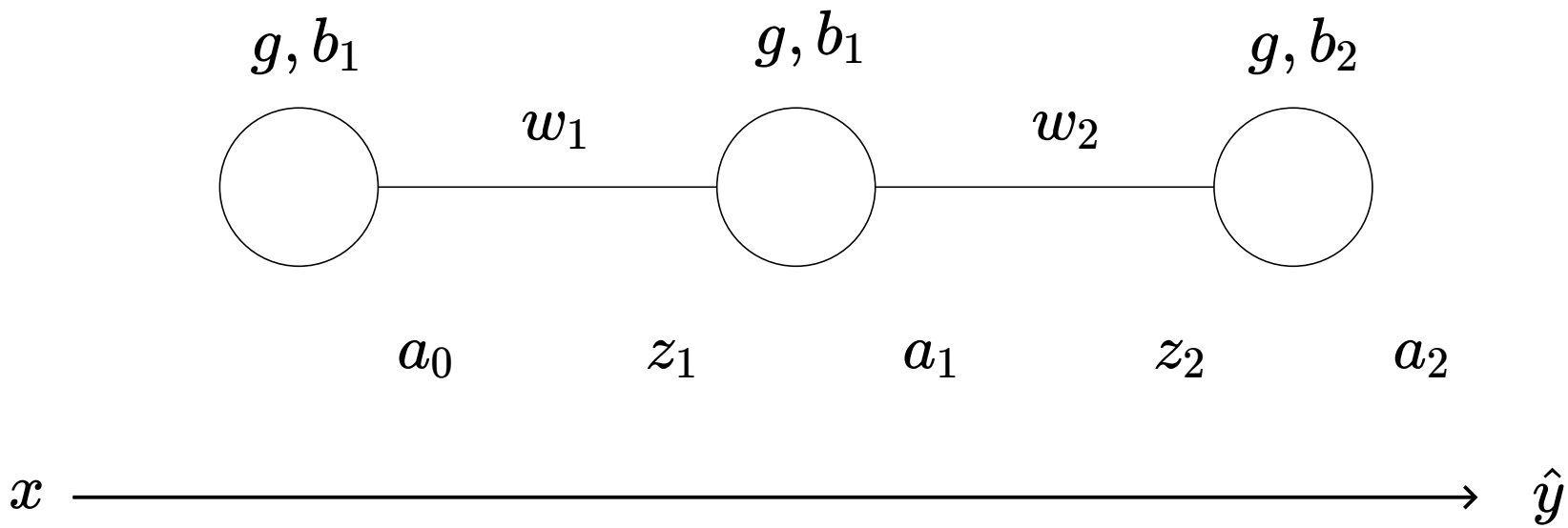


Backpropagation



In order to compute the gradients, we first need to compute the loss on a training example.

Forward Pass



Layer-0

$$a_0 = x$$

Layer-1

$$z_1 = w_1 a_0 + b_1$$

$$a_1 = g(z_1)$$

Layer-2

$$z_2 = w_2 a_1 + b_2$$

$$a_2 = g(z_2)$$

$$\hat{y} = a_2$$

$$L(y, \hat{y}) = \frac{1}{2}(\hat{y} - y)^2$$

Gradients

$$x \longrightarrow \hat{y}$$

Layer-0	Layer-1	Layer-2
$a_0 = x$	$z_1 = w_1 a_0 + b_1$	$z_2 = w_2 a_1 + b_2$
	$a_1 = g(z_1)$	$a_2 = g(z_2)$
		$\hat{y} = a_2$
	$\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial b_1}$	$\frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial b_2}$

$$L(y, \hat{y}) = \frac{1}{2}(\hat{y} - y)^2$$

Chain rule (layer-2)

$$x \longrightarrow \hat{y}$$

Layer-0	Layer-1	Layer-2
$a_0 = x$	$z_1 = w_1 a_0 + b_1$	$z_2 = w_2 a_1 + b_2$
	$a_1 = g(z_1)$	$a_2 = g(z_2)$
		$\hat{y} = a_2$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_2}$$

$$L(y, \hat{y}) = \frac{1}{2}(\hat{y} - y)^2$$

Chain rule (layer-2)

$$x \longrightarrow \hat{y}$$

Layer-0	Layer-1	Layer-2
$a_0 = x$	$z_1 = w_1 a_0 + b_1$	$z_2 = w_2 a_1 + b_2$
	$a_1 = g(z_1)$	$a_2 = g(z_2)$
		$\hat{y} = a_2$

$$\frac{\partial L}{\partial w_2} = (\hat{y} - y) \cdot g'(z_2) \cdot a_1$$

$$L(y, \hat{y}) = \frac{1}{2}(\hat{y} - y)^2$$

Chain rule (layer-1)

$$x \longrightarrow \hat{y}$$

Layer-0	Layer-1	Layer-2
$a_0 = x$	$z_1 = w_1 a_0 + b_1$	$z_2 = w_2 a_1 + b_2$
	$a_1 = g(z_1)$	$a_2 = g(z_2)$
		$\hat{y} = a_2$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$$

$$\frac{\partial L}{\partial w_2} = (\hat{y} - y) \cdot g'(z_2) \cdot a_1$$

Chain rule (layer-1)

$$x \longrightarrow \hat{y}$$

Layer-0	Layer-1	Layer-2
$a_0 = x$	$z_1 = w_1 a_0 + b_1$	$z_2 = w_2 a_1 + b_2$
	$a_1 = g(z_1)$	$a_2 = g(z_2)$
		$\hat{y} = a_2$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$$

$$\frac{\partial L}{\partial w_2} = (\hat{y} - y) \cdot g'(z_2) \cdot a_1$$

$$\frac{\partial L}{\partial a_1} = (\hat{y} - y) \cdot g'(z_2) \cdot w_2$$

Backward Pass

$$x \longrightarrow \hat{y}$$

Layer-0	Layer-1	Layer-2
$a_0 = x$	$z_1 = w_1 a_0 + b_1$	$z_2 = w_2 a_1 + b_2$
	$a_1 = g(z_1)$	$a_2 = g(z_2)$
		$\hat{y} = a_2$

$$\longleftarrow$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial a_1} \cdot g'(z_1) \cdot a_0$$

$$\frac{\partial L}{\partial w_2} = (\hat{y} - y) \cdot g'(z_2) \cdot a_1$$

$$\frac{\partial L}{\partial a_1} = (\hat{y} - y) \cdot g'(z_2) \cdot w_2$$

Gradients (hidden layers)

```
Activations -> Pre-activations-> Weights
```

$\boldsymbol{A}_l^{(g)}$ gradient of the loss w.r.t the activations

$\boldsymbol{Z}_l^{(g)}$ gradient of the loss w.r.t the pre-activations

$\boldsymbol{W}_l^{(g)}$ gradient of the loss w.r.t the weights

Gradients (hidden layers)

If $\boldsymbol{A}_l^{(g)}$ is already known, we can compute the rest of the gradients:

$$\boldsymbol{Z}_l^{(g)} = \boldsymbol{A}_l^{(g)} \odot g'(\boldsymbol{Z}_l)$$

$$\boldsymbol{W}_l^{(g)} = \boldsymbol{A}_{l-1}^T \boldsymbol{Z}_l^{(g)}$$

$$\boldsymbol{A}_{l-1}^{(g)} = \boldsymbol{Z}_l^{(g)} \boldsymbol{W}_l{}^T$$

Gradients (hidden layers)

These equations will make more sense if we compare them with their forward-pass counterparts:

$$\boldsymbol{Z_l} = \boldsymbol{A_{l-1}}\boldsymbol{W_l} + \boldsymbol{b_l}$$

$$\boldsymbol{A_l} = g(\boldsymbol{Z_l})$$

$$\boldsymbol{Z_l^{(g)}} = \boldsymbol{A_l^{(g)}} \odot g'(\boldsymbol{Z_l})$$

$$\boldsymbol{W_l^{(g)}} = \boldsymbol{A_{l-1}^T}\boldsymbol{Z_l^{(g)}}$$

$$\boldsymbol{A_{l-1}^{(g)}} = \boldsymbol{Z_l^{(g)}}\boldsymbol{W_l}^T$$

For a moment think of all of them as scalars. We can show that these equations are the matrix equivalents of the chain-rule.

Gradients (output layer)

All gradients in the hidden layers depend on the gradient of the loss with respect to the output activations: $\boldsymbol{A}_{\boldsymbol{L}}^{(\boldsymbol{g})}$. This depends on the nature of the loss.

Regression

$$\boldsymbol{A}_{\boldsymbol{L}}^{(\boldsymbol{g})} = \hat{\boldsymbol{y}} - \boldsymbol{y}$$

Classification

$$\boldsymbol{A}_{\boldsymbol{L}}^{(\boldsymbol{g})} = -\boldsymbol{Y} \odot \hat{\boldsymbol{Y}}^{\odot -1}$$

Gradients (regression, output layer)

The activation function in the final layer for regression is just the identity function. There is a slight abuse of notation: $\boldsymbol{A}_{\boldsymbol{L}}^{(\boldsymbol{g})}$ and $\boldsymbol{Z}_{\boldsymbol{L}}^{(\boldsymbol{g})}$ are vectors.

$$\boldsymbol{A}_{\boldsymbol{L}}^{(\boldsymbol{g})} = \hat{\boldsymbol{y}} - \boldsymbol{y}$$

$$\boldsymbol{Z}_{\boldsymbol{L}}^{(\boldsymbol{g})} = g'(\boldsymbol{Z}_{\boldsymbol{l}}) \odot \boldsymbol{A}_{\boldsymbol{L}}^{(\boldsymbol{g})}$$

$$= \boldsymbol{A}_{\boldsymbol{L}}^{(\boldsymbol{g})}$$

Gradients (classification, output layer)

The activation function in the final layer for classification is Softmax.

$$\boldsymbol{A}_{\boldsymbol{L}}^{(\boldsymbol{g})} = -\boldsymbol{Y} \odot \hat{\boldsymbol{Y}}^{\odot -1}$$

$$\boldsymbol{Z}_{\boldsymbol{L}}^{(\boldsymbol{g})} = \hat{\boldsymbol{Y}} - \boldsymbol{Y}$$

The derivation for Softmax is a bit involved. The details can be found here:

https://bsc-iitm.github.io/machine-learning-techniques/neural-networks/07-appendix/#gradient-for-softmax-layer

Backward pass (regression)

Iterative algorithm to compute the gradients given $(\boldsymbol{y}, \hat{\boldsymbol{y}})$

```
Initialize:
```
$$\boldsymbol{Z}_{\boldsymbol{L}}^{(g)} = \hat{\boldsymbol{y}} - \boldsymbol{y}$$

```
for l = L to l = 1:
```

$$\boldsymbol{W}_{\boldsymbol{l}}^{(g)} = \boldsymbol{A}_{\boldsymbol{l-1}}^{T} \boldsymbol{Z}_{\boldsymbol{l}}^{(g)}$$

$$\boldsymbol{b}_{\boldsymbol{l}}^{(g)} = \boldsymbol{Z}_{\boldsymbol{l}}^{(g)^T} \mathbf{1}_{\boldsymbol{n}}$$

$$\boldsymbol{A}_{\boldsymbol{l-1}}^{(g)} = \boldsymbol{Z}_{\boldsymbol{l}}^{(g)} \boldsymbol{W}_{\boldsymbol{l}}^{T}$$

$$\boldsymbol{Z}_{\boldsymbol{l-1}}^{(g)} = \boldsymbol{A}_{\boldsymbol{l-1}}^{(g)} \odot g'(\boldsymbol{Z}_{\boldsymbol{l-1}})$$

Notes

(1) g is identity for the output layer

(2) Therefore, $\boldsymbol{Z}_{\boldsymbol{L}}^{(g)} = \boldsymbol{A}_{\boldsymbol{L}}^{(g)}$

Backward pass (classification)

Iterative algorithm to compute the gradients given $(\boldsymbol{Y}, \hat{\boldsymbol{Y}})$

```
Initialize:
```
$$\boldsymbol{Z}_{\boldsymbol{L}}^{(g)} = \hat{\boldsymbol{Y}} - \boldsymbol{Y}$$

```
for l = L to l = 1:
```

$$\boldsymbol{W}_{\boldsymbol{l}}^{(g)} = \boldsymbol{A}_{\boldsymbol{l-1}}^{T} \boldsymbol{Z}_{\boldsymbol{l}}^{(g)}$$

$$\boldsymbol{b}_{\boldsymbol{l}}^{(g)} = \boldsymbol{Z}_{\boldsymbol{l}}^{(g)^{T}} \mathbf{1}_{\boldsymbol{n}}$$

$$\boldsymbol{A}_{\boldsymbol{l-1}}^{(g)} = \boldsymbol{Z}_{\boldsymbol{l}}^{(g)} \boldsymbol{W}_{\boldsymbol{l}}^{T}$$

$$\boldsymbol{Z}_{\boldsymbol{l-1}}^{(g)} = \boldsymbol{A}_{\boldsymbol{l-1}}^{(g)} \odot g'(\boldsymbol{Z}_{\boldsymbol{l-1}})$$

Notes

(1) g is softmax for the output layer

(2) Therefore, $\boldsymbol{Z}_{\boldsymbol{L}}^{(g)} = \hat{\boldsymbol{Y}} - \boldsymbol{Y}$

Gradient Descent (Neural Networks)

We now have all the ingredients to completely specify the learning algorithm.

- Let $\boldsymbol{\theta}$ refer to all the parameters in the model.
- Let us define the following functions:

$$\hat{\boldsymbol{Y}} = \text{forward-pass}(\boldsymbol{X})$$

$$L = \text{loss}(\boldsymbol{Y}, \hat{\boldsymbol{Y}})$$

$$\boldsymbol{\theta}^{(g)} = \text{backward-pass}(\boldsymbol{Y}, \hat{\boldsymbol{Y}})$$

Only the most important arguments are displayed here

Gradient Descent (Neural Networks)

If we have E epochs, then the algorithm is as follows:

```
Initialize: θ ~ N(0,1)

for e = 1 to e = E:
```

$$\hat{\boldsymbol{Y}} = \text{forward-pass}(\boldsymbol{X})$$

$$L = \text{loss}(\boldsymbol{Y}, \hat{\boldsymbol{Y}})$$

$$\boldsymbol{\theta}^{(g)} = \text{backward-pass}(\boldsymbol{Y}, \hat{\boldsymbol{Y}})$$

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \boldsymbol{\theta}^{(g)}$$

Initialization

Why did we not initialize all parameters to zero?

```
Initialize: θ ~ N(0, 1)

for e = 1 to e = E:
```

$$\hat{\boldsymbol{Y}} = \text{forward-pass}(\boldsymbol{X})$$

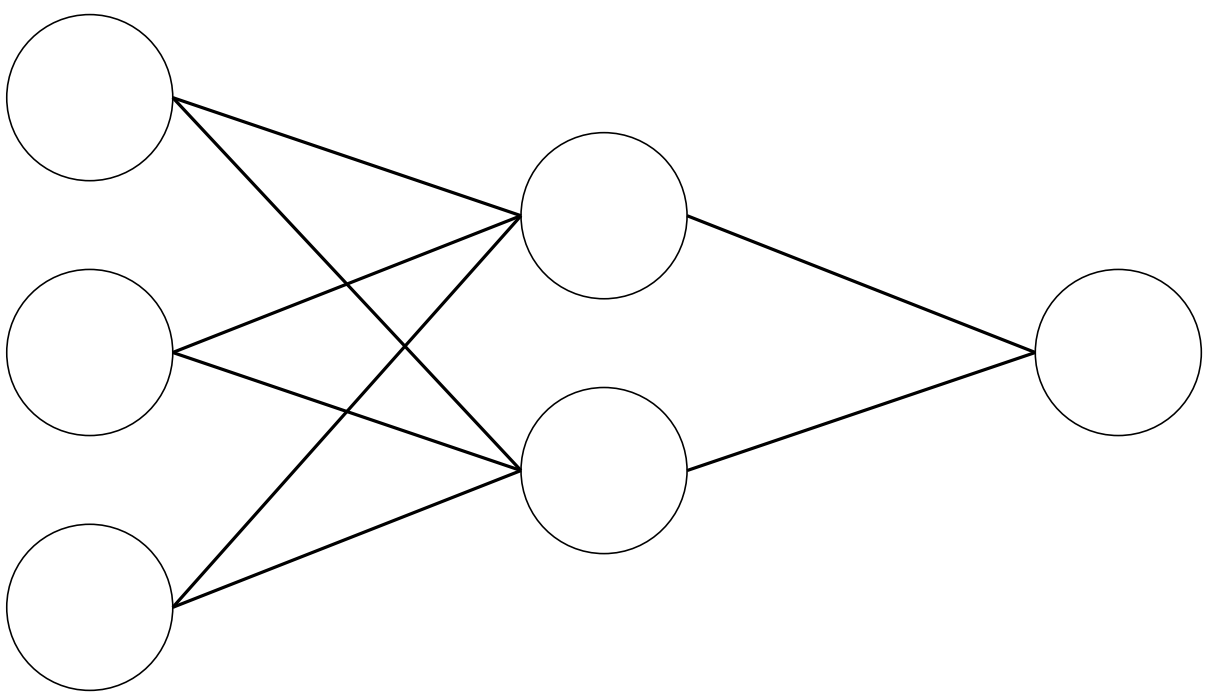
$$L = \text{loss}(\boldsymbol{Y}, \hat{\boldsymbol{Y}})$$

$$\boldsymbol{\theta}^{(g)} = \text{backward-pass}(\boldsymbol{Y}, \hat{\boldsymbol{Y}})$$

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \boldsymbol{\theta}^{(g)}$$

Initialization

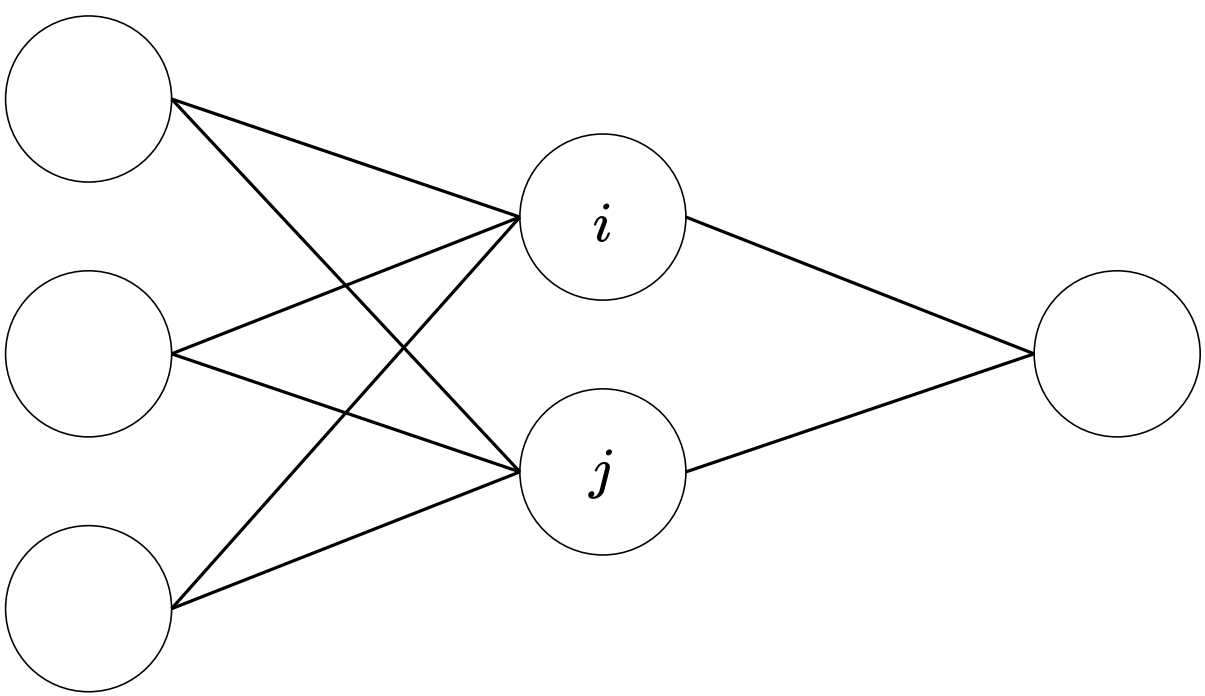
What happens if we initialize all the parameters to some constant value?



Initialization

Reference: https://www.deeplearning.ai/ai-notes/initialization/

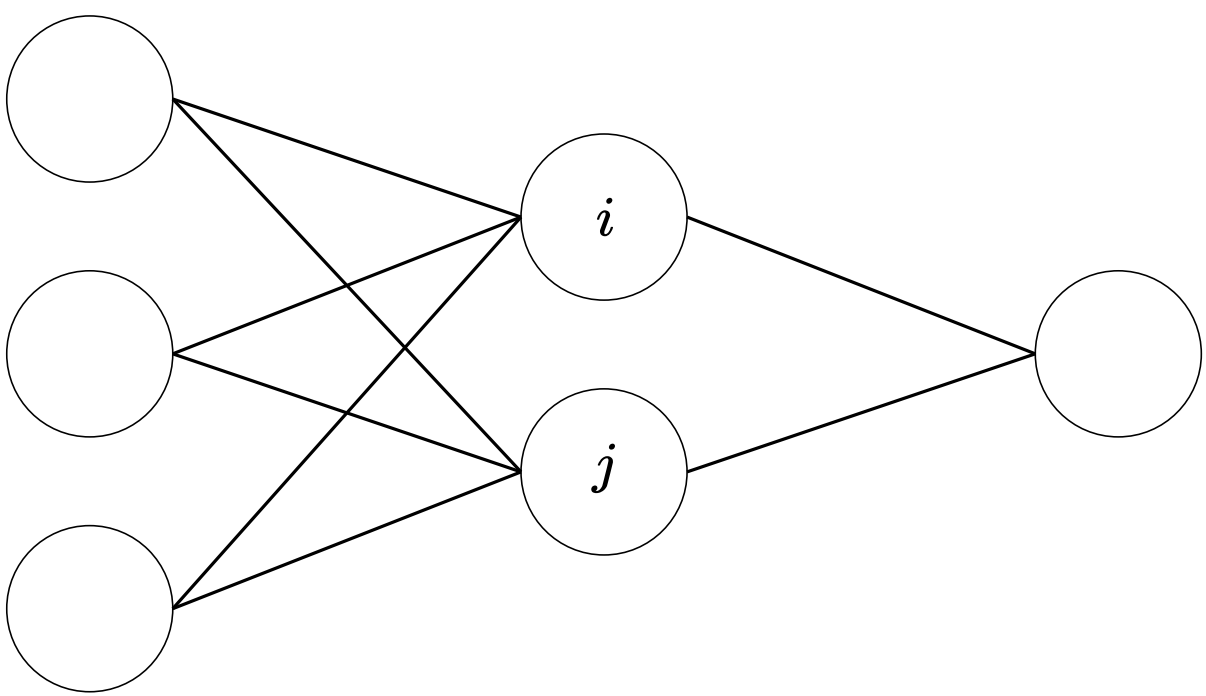
- Since the incoming weights are the same for all neurons in a layer, there is nothing to differentiate between two neurons.
- This symmetry means that they will evolve identically and will not learn different things.



Initialization

Reference: https://www.deeplearning.ai/ai-notes/initialization/

- So, setting all weights to a constant value, especially 0, is a bad idea for neural networks.
- In our implementation, we have sampled the weights from a standard normal distribution.



Overfitting

Let us take a simple three layer neural network for the MNIST classification problem with the following architecture:

Layer	**Number of neurons**
Input	784
Hidden layer	50
Output layer	10

Overfitting

The number of parameters (weights + biases) for this model is as follows:

Layer	**Number of neurons**	**Weights**
Input	784	0
Hidden layer	50	784 * 50 = 39200
Output layer	10	50 * 10 = 500

Number of parameters = (39200 + 50) + (500 + 10) = 39,760

Overfitting

- With just 1 hidden layer, we have a model with nearly 40,000 parameters.
- This number will quickly blow up as more hidden layers are added.
- Neural networks are thus prone to overfitting.

Regularization

There are two methods to regularize the model:

(1) L1/L2 regularization

(2) Dropout

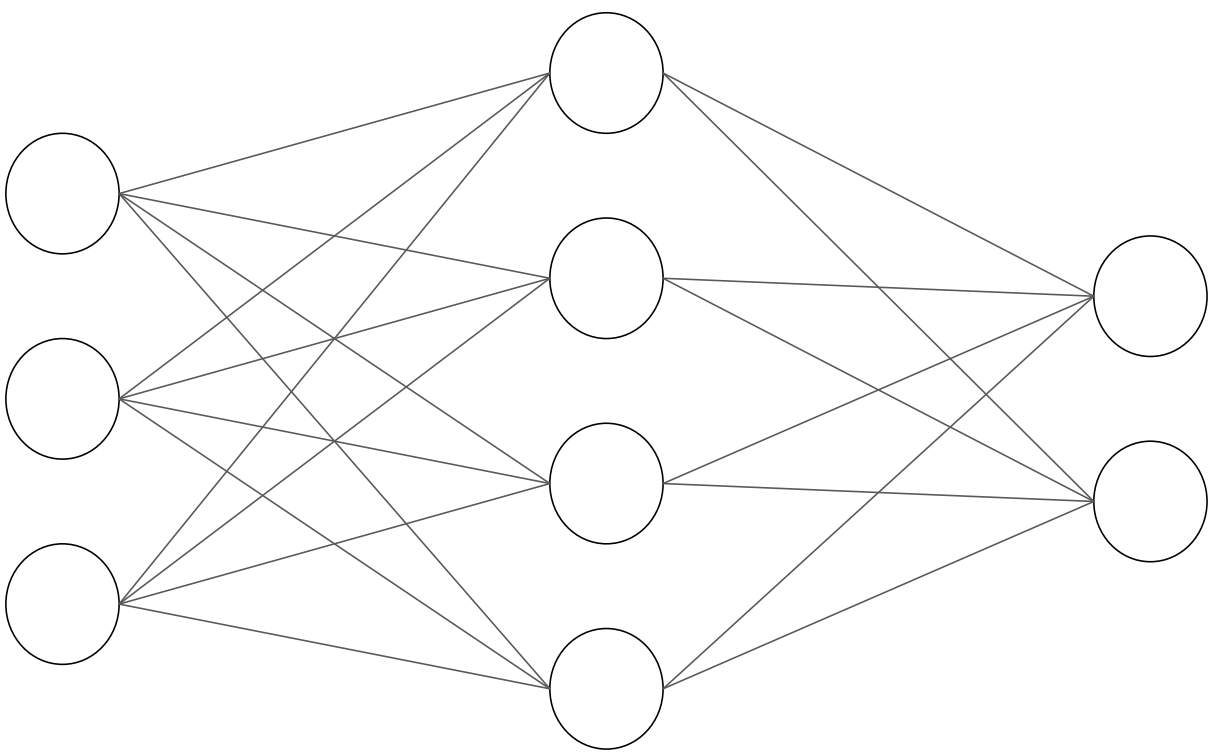
We are familiar with method (1).

Method (2) is particular to neural networks.

Dropout

Reference: http://neuralnetworksanddeeplearning.com/chap3.html

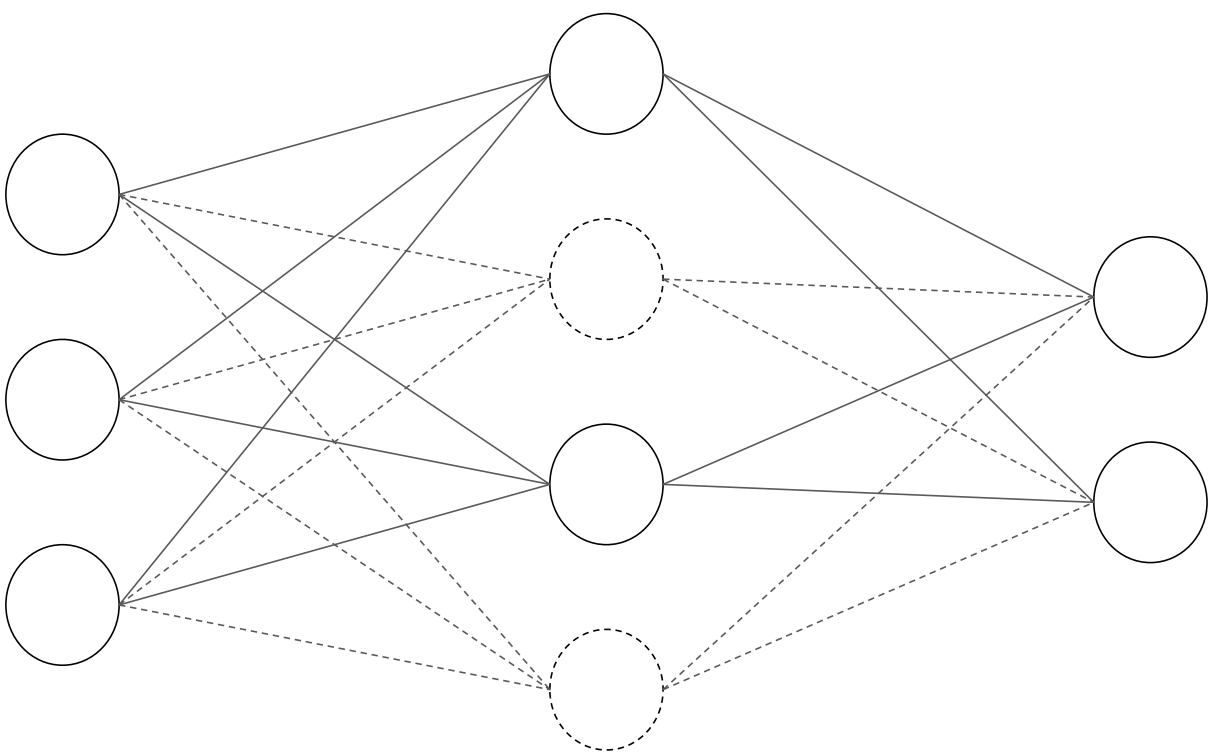
Consider the following network with one hidden layer:



Dropout

Reference: http://neuralnetworksanddeeplearning.com/chap3.html

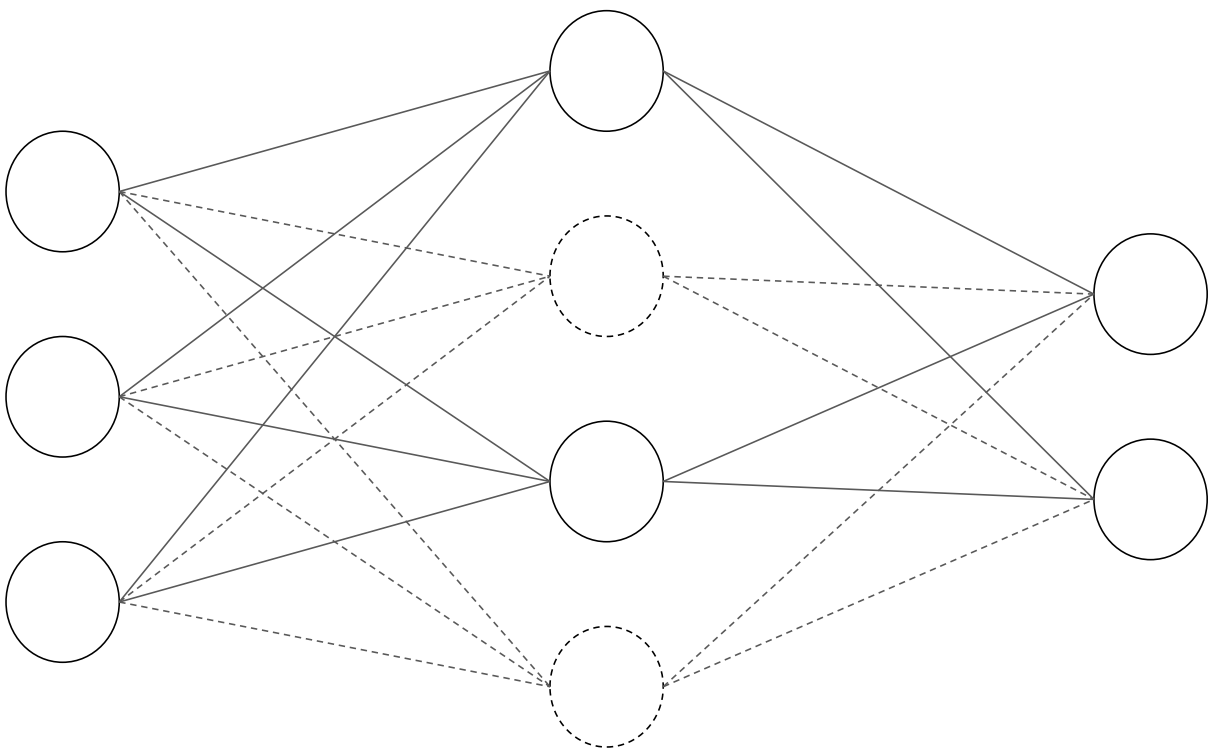
In each iteration of GD, randomly choose half the neurons in the hidden layer and "drop" them "out" of the network.



Dropout

Reference: http://neuralnetworksanddeeplearning.com/chap3.html

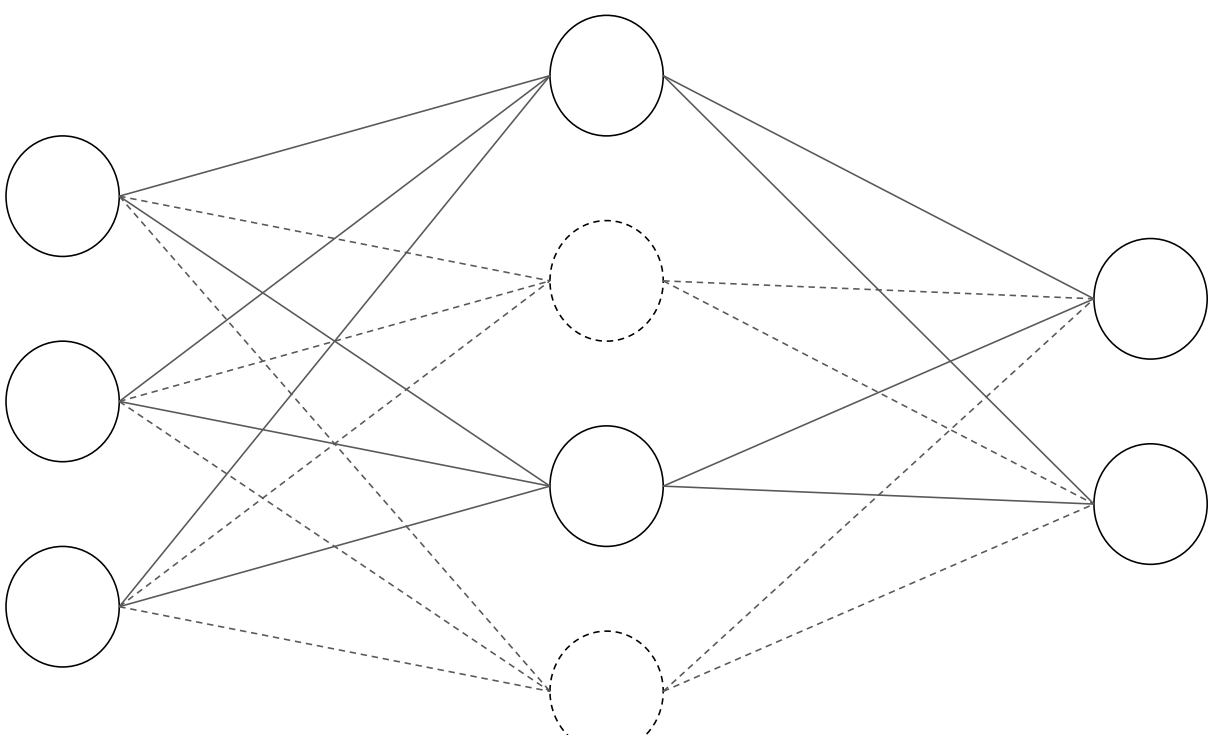
Do a forward pass and backward pass on this modified network. Update the weights.



Dropout

Reference: http://neuralnetworksanddeeplearning.com/chap3.html

At inference, use the full network, but halve the outgoing weights from the hidden layer. This is because only half of the neurons were active during training.



Dropout

Reference: http://neuralnetworksanddeeplearning.com/chap3.html

Why does dropout work?

- During the training phase, dropout can be seen as producing multiple networks, each having a different number of neurons in the hidden layers.
- At test time, we can think of the output of a network as averaging over the results of all these networks.

Regularization

Reference: http://neuralnetworksanddeeplearning.com/chap3.html

There is a key difference between these two methods of regularization:

(1) L1/L2 regularization: modifies the loss function

(2) Dropout: modifies the network