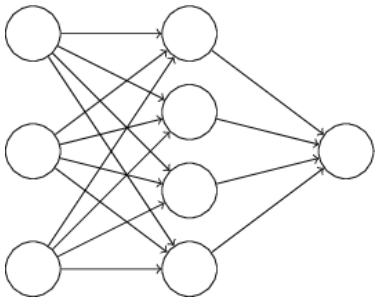
# Introduction to Artificial Neural Networks

*Prepared for PyData Denver*

by

Ryan Arredondo

03/08/2017



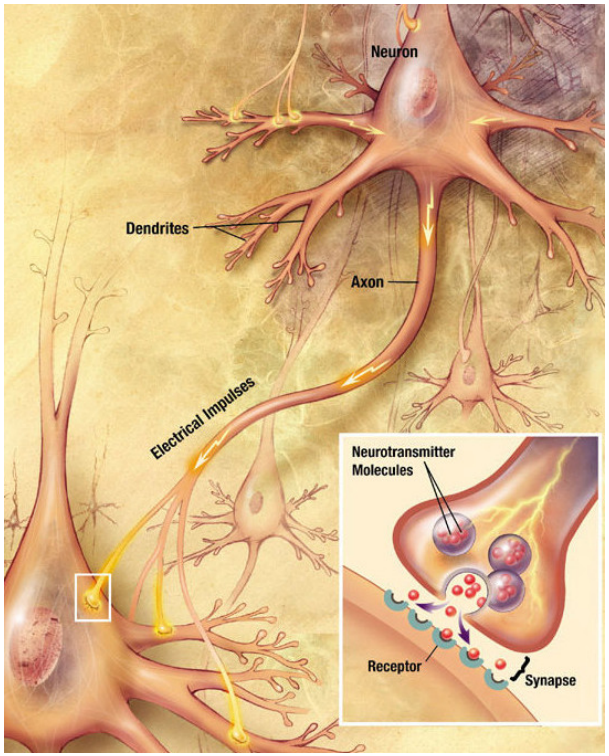
+

=

# Table of Contents

# Biologically Motivated

- Artificial neural networks were motivated by models of how learning could happen in the brain.




Source: http://www.nia.nih.gov/alzheimers/publication/alzheimers-disease-unraveling-mystery/preface

# First Wave (1940s – 1960s)

## Linear Models

Models computed a linear function

$$f(x, w) = x_1 w_1 + x_2 w_2 + \cdots + x_n w_n$$

from an input $x$ and a set of weights $w$.

# First Wave (1940s – 1960s)

## Linear Models

Models computed a linear function

$$f(x, w) = x_1 w_1 + x_2 w_2 + \cdots + x_n w_n$$

from an input $x$ and a set of weights $w$.

Some examples

# First Wave (1940s – 1960s)

## Linear Models

Models computed a linear function

$$f(x, w) = x_1 w_1 + x_2 w_2 + \cdots + x_n w_n$$

from an input $x$ and a set of weights $w$.

Some examples

- **(1943) McCulloch-Pitts neuron:** served as a computational model; weights are set by operator

# First Wave (1940s – 1960s)

## Linear Models

Models computed a linear function

$$f(x, w) = x_1 w_1 + x_2 w_2 + \cdots + x_n w_n$$

from an input $x$ and a set of weights $w$.

Some examples

- **(1943) McCulloch-Pitts neuron:** served as a computational model; weights are set by operator
- **(1957) Rosenblatt's percepton:** algorithm to learn the weights from input; was used in a machine intended for image recognition

# First Wave (1940s – 1960s)

## Linear Models

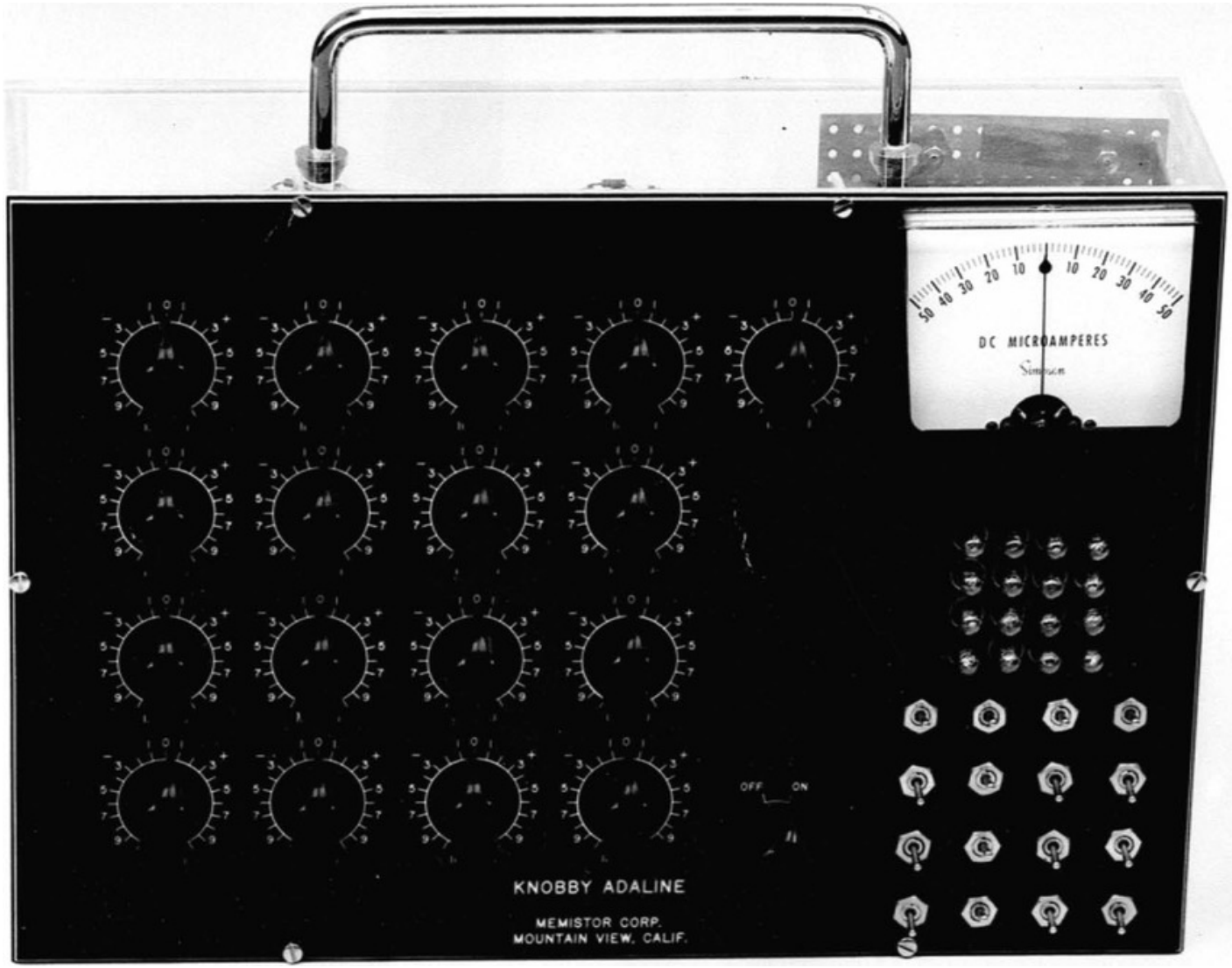
Models computed a linear function

$$f(x, w) = x_1 w_1 + x_2 w_2 + \cdots + x_n w_n$$

from an input $x$ and a set of weights $w$.

Some examples

- **(1943) McCulloch-Pitts neuron:** served as a computational model; weights are set by operator
- **(1957) Rosenblatt's percepton:** algorithm to learn the weights from input; was used in a machine intended for image recognition
- **(1960) Widrow and Hoff's adaptive linear neuron:** trained using a special case of stochastic gradient descent




ADALINE. An adaptive linear neuron.

# Second Wave (1980s – 1990s)

## Important progress

New research arose in the context of cognitive science.

# Second Wave (1980s – 1990s)

## Important progress

New research arose in the context of cognitive science.

Some key concepts

# Second Wave (1980s – 1990s)

## Important progress

New research arose in the context of cognitive science.

Some key concepts

- **Distributed representation:** the inputs should have many features, and the features should be representative of many inputs

# Second Wave (1980s – 1990s)

## Important progress

New research arose in the context of cognitive science.

Some key concepts

- **Distributed representation:** the inputs should have many features, and the features should be representative of many inputs
- **Backpropagation:** Algorithm to train neural networks. Is still the dominant approach today.

# Third Wave (2006 – Present)

## Reasons for its resurgence

Key factors in the success of deep neural networks

# Third Wave (2006 – Present)

## Reasons for its resurgence

Key factors in the success of deep neural networks

- **Increasing data set sizes.**

# Third Wave (2006 – Present)

## Reasons for its resurgence

Key factors in the success of deep neural networks

- **Increasing data set sizes.**
  "Deep learning will generally... match or exceed human performance.. with 10 million labelled examples" (Goodfellow et al)

# Third Wave (2006 – Present)

## Reasons for its resurgence

Key factors in the success of deep neural networks

- **Increasing data set sizes.**
  "Deep learning will generally... match or exceed human performance.. with 10 million labelled examples" (Goodfellow et al)
- **Increasing model sizes.** Models double in size every $\sim$2.4 years.

# Third Wave (2006 – Present)

## Reasons for its resurgence

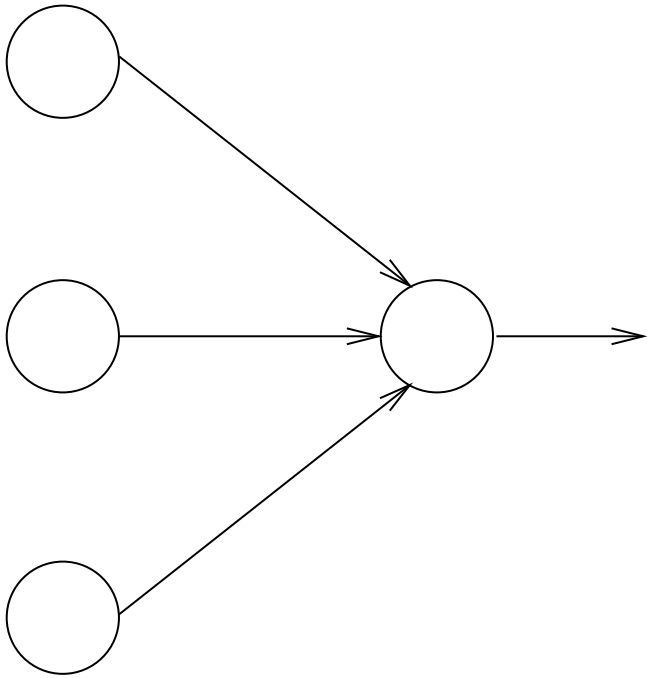
Key factors in the success of deep neural networks

- **Increasing data set sizes.**
  "Deep learning will generally... match or exceed human performance.. with 10 million labelled examples" (Goodfellow et al)
- **Increasing model sizes.** Models double in size every $\sim$2.4 years.
- **Increasing accuracy and complexity.** Significant drops in the error rate for object and speech recognition problems using NNs.

# Table of Contents

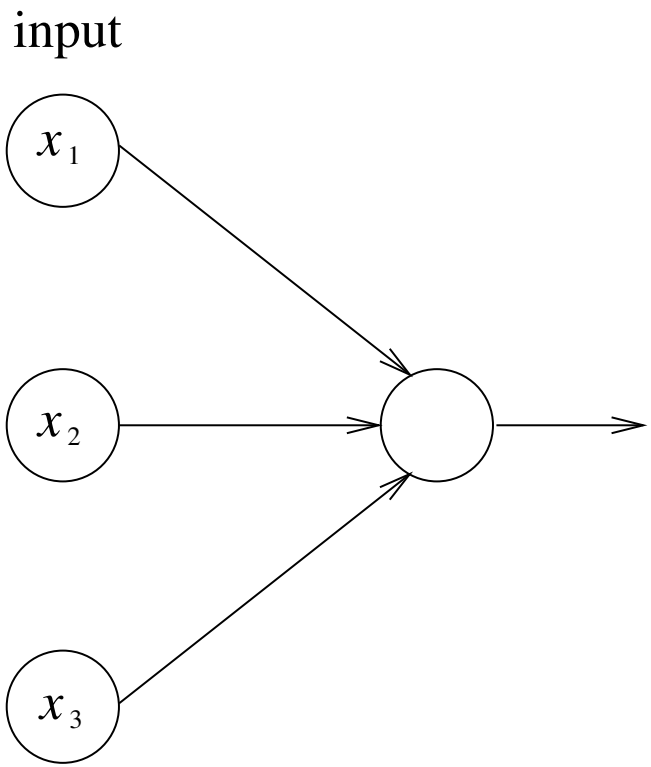
# The Simplest Neural Network

Sinlge-layer perceptron:



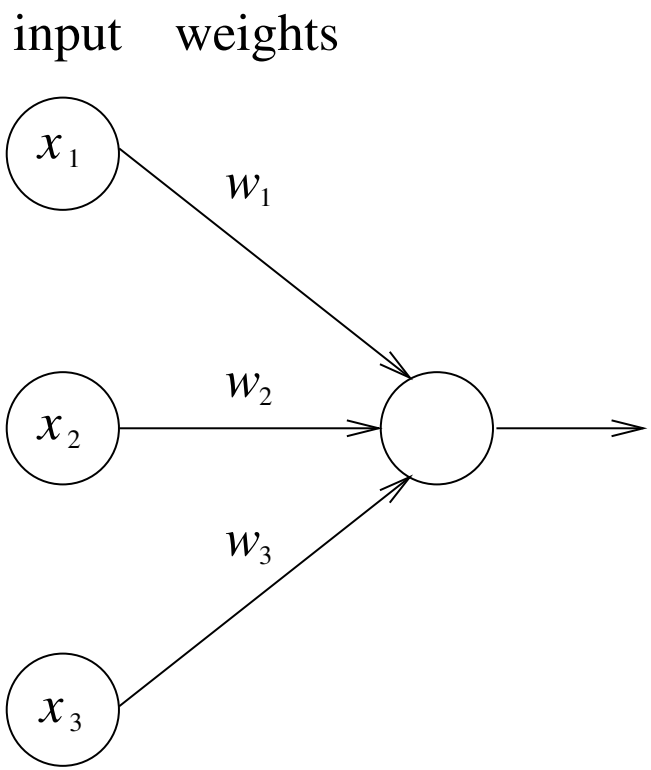
# The Simplest Neural Network

Sinlge-layer perceptron:



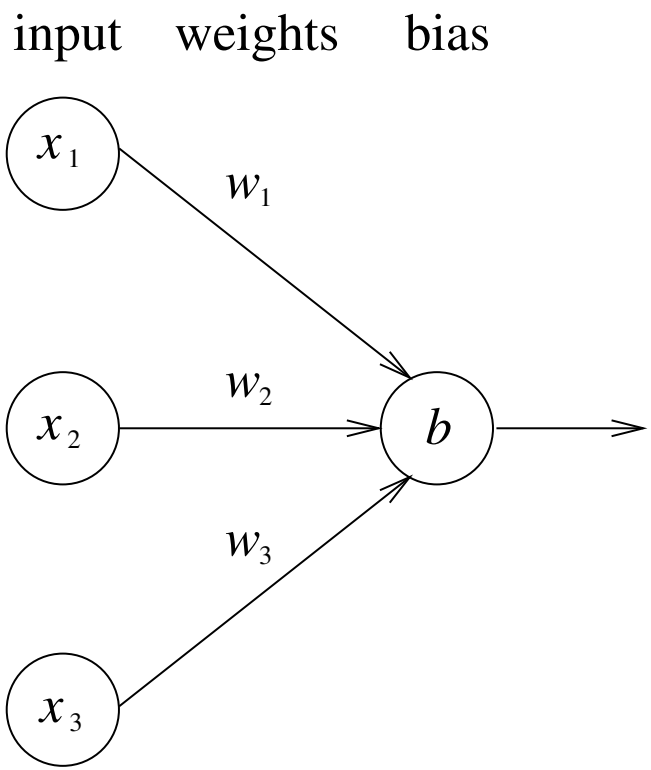
# The Simplest Neural Network

Sinlge-layer perceptron:



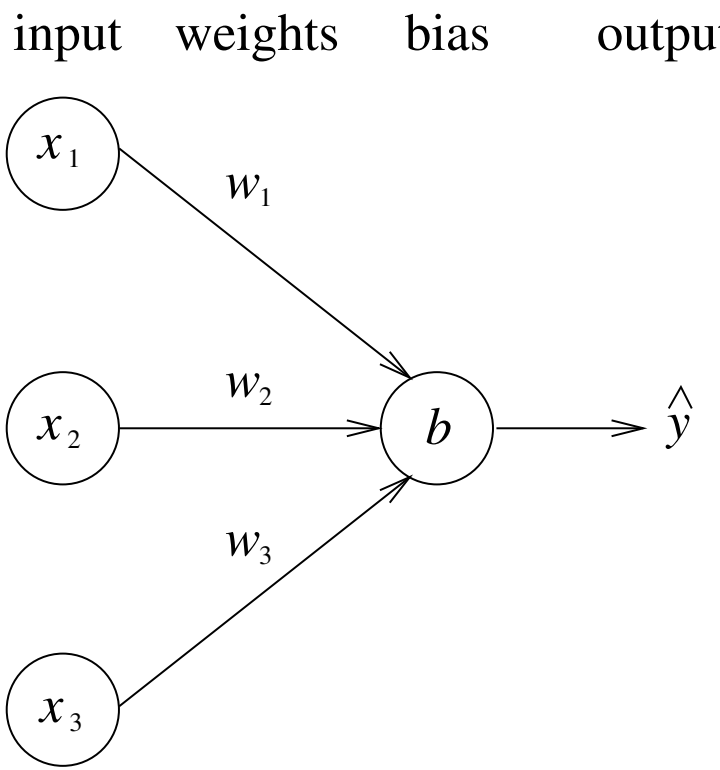
# The Simplest Neural Network

Sinlge-layer perceptron:



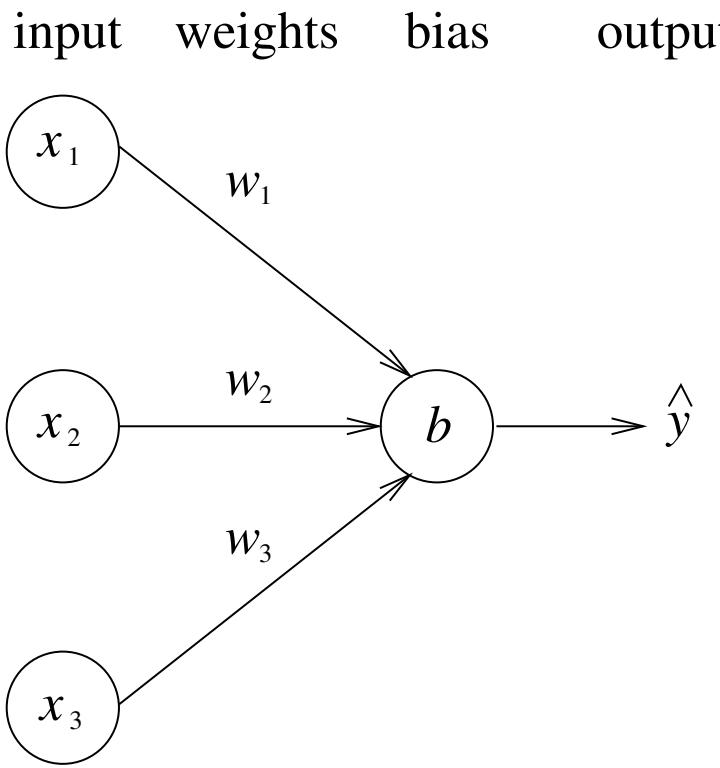
# The Simplest Neural Network

Sinlge-layer perceptron:



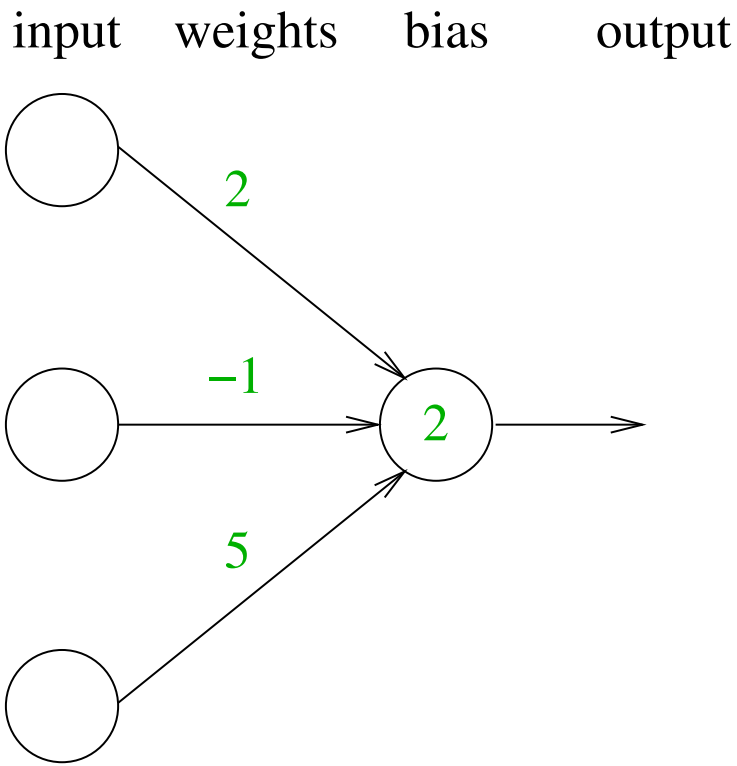
# The Simplest Neural Network

Sinlge-layer perceptron:

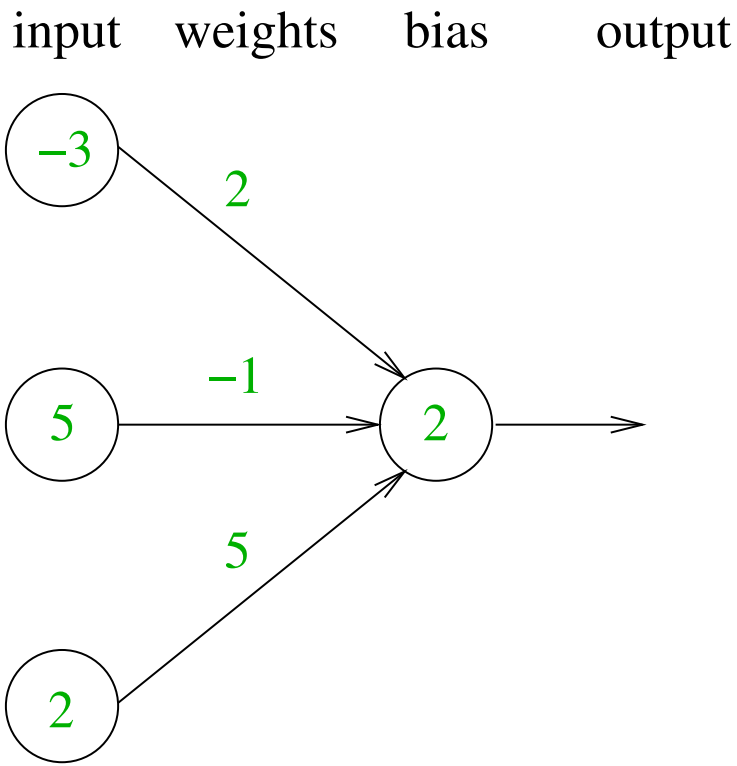



$$\hat{y} = \begin{cases} 0 & \text{if } \sum x_i w_i + b \leq 0 \\ 1 & \text{if } \sum x_i w_i + b > 0 \end{cases}$$

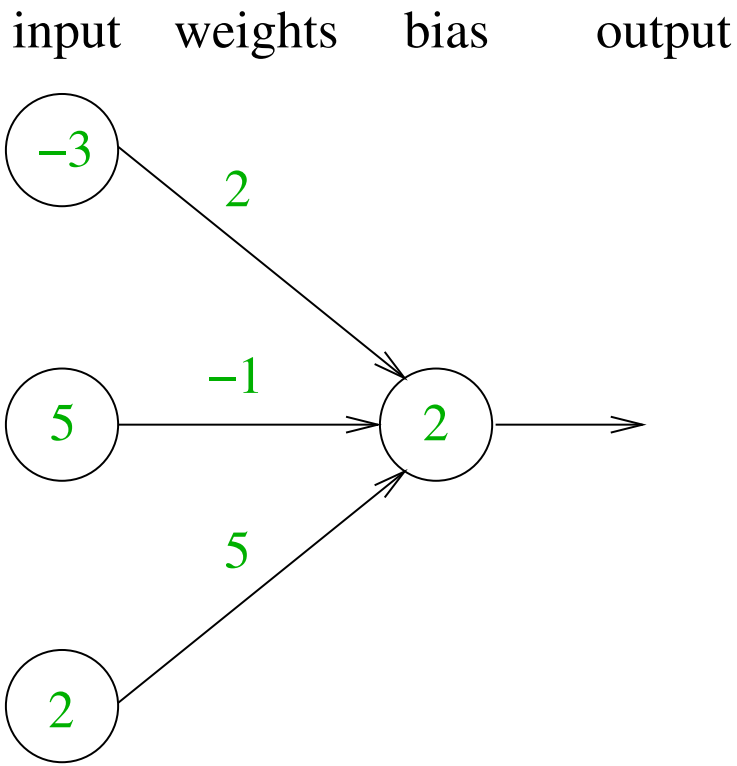
# The Simplest Neural Network Example



# The Simplest Neural Network Example

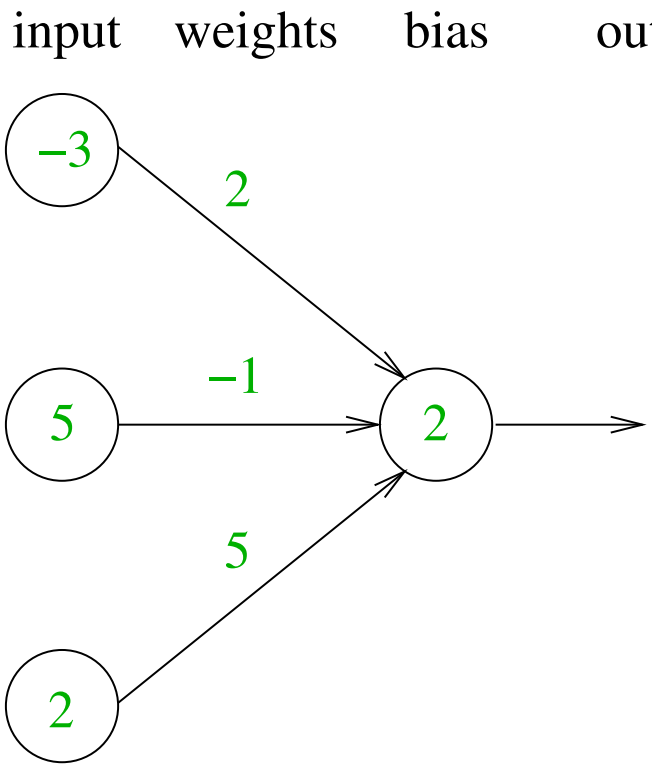


# The Simplest Neural Network Example



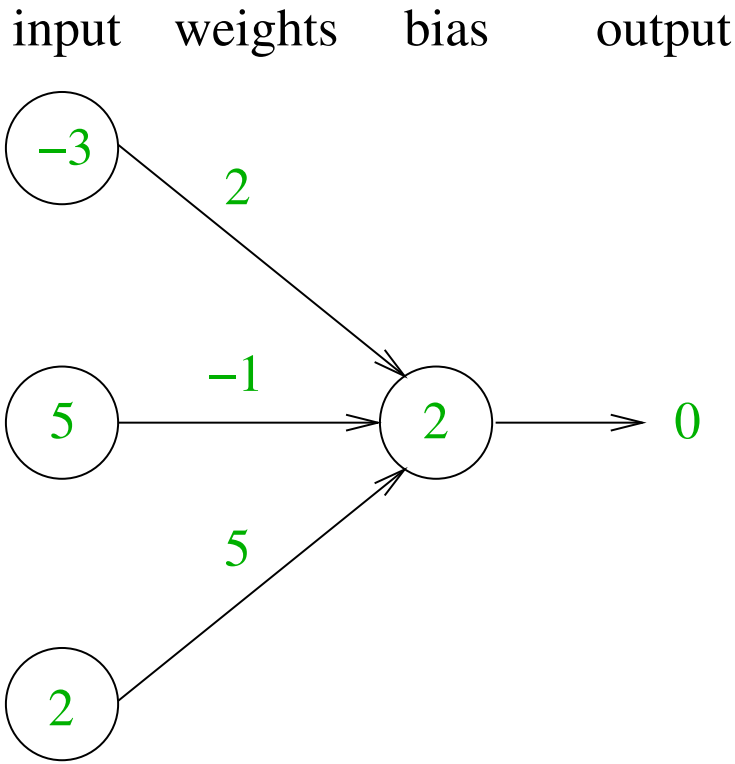

$$\sum x_i w_i + b = (-3) \times 2 + 5 \times (-1) + 2 \times 5 + 2$$

# The Simplest Neural Network Example



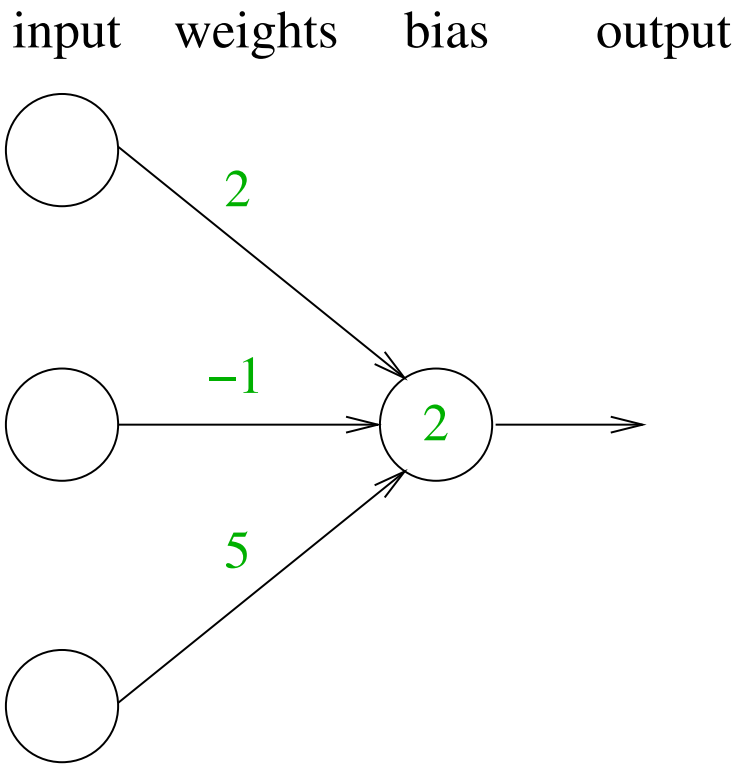

$$\sum x_i w_i + b = (-3) \times 2 + 5 \times (-1) + 2 \times 5 + 2 = -1$$

# The Simplest Neural Network Example

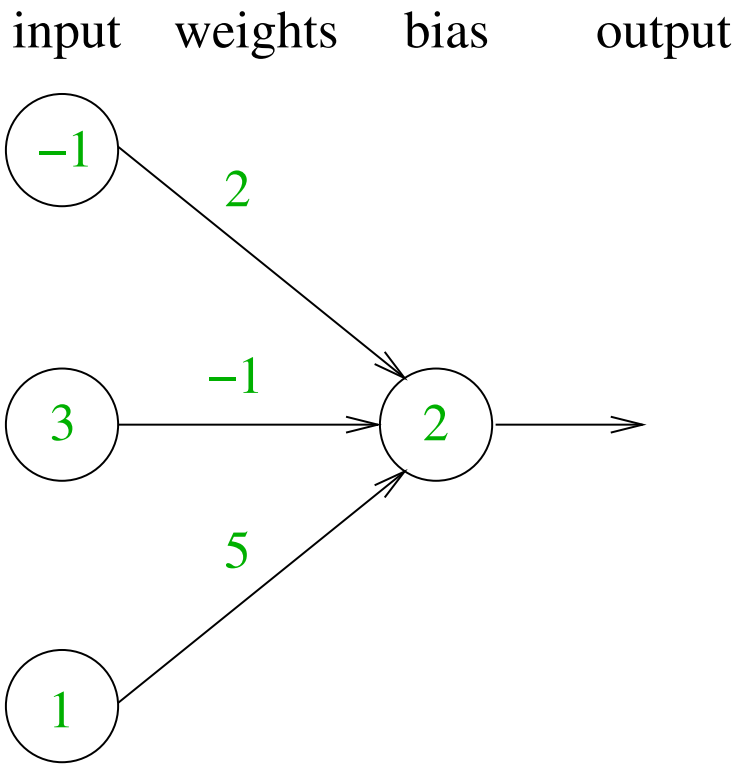



$$\sum x_i w_i + b = (-3) \times 2 + 5 \times (-1) + 2 \times 5 + 2 = -1 \quad \Longrightarrow \quad y = 0$$

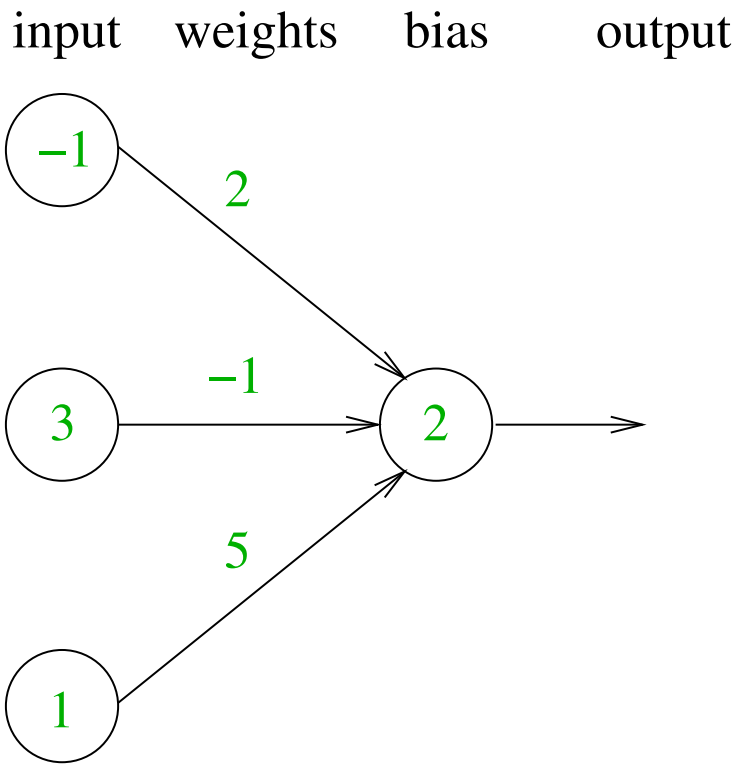
# The Simplest Neural Network Example



# The Simplest Neural Network Example

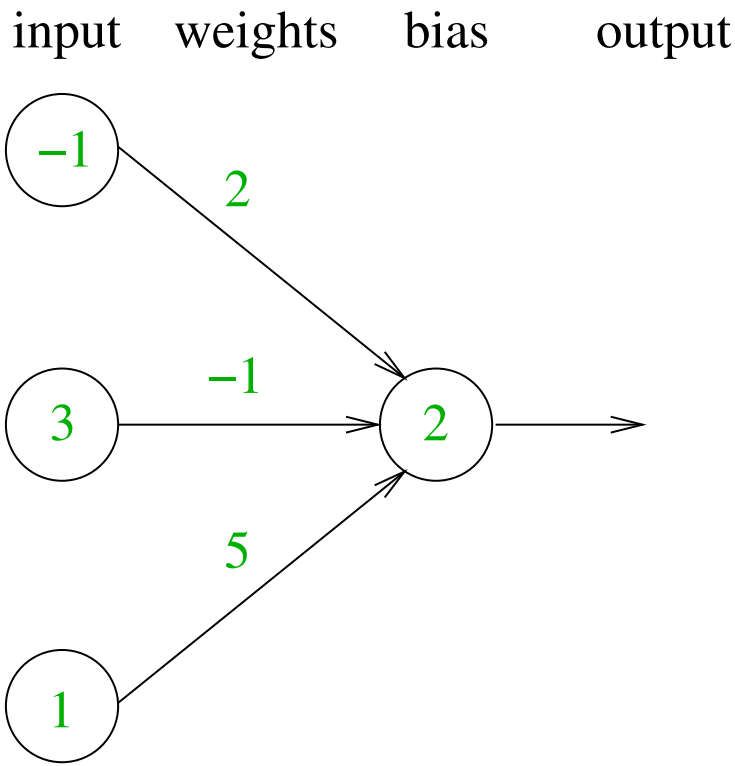


# The Simplest Neural Network Example



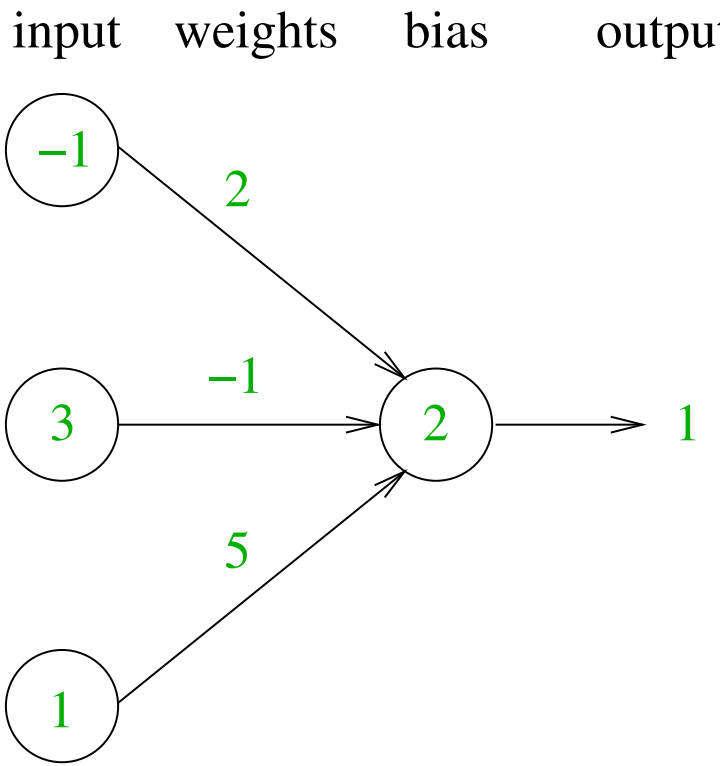

$$\sum x_i w_i + b = (-1) \times 2 + 3 \times (-1) + 1 \times 5 + 2$$

# The Simplest Neural Network Example



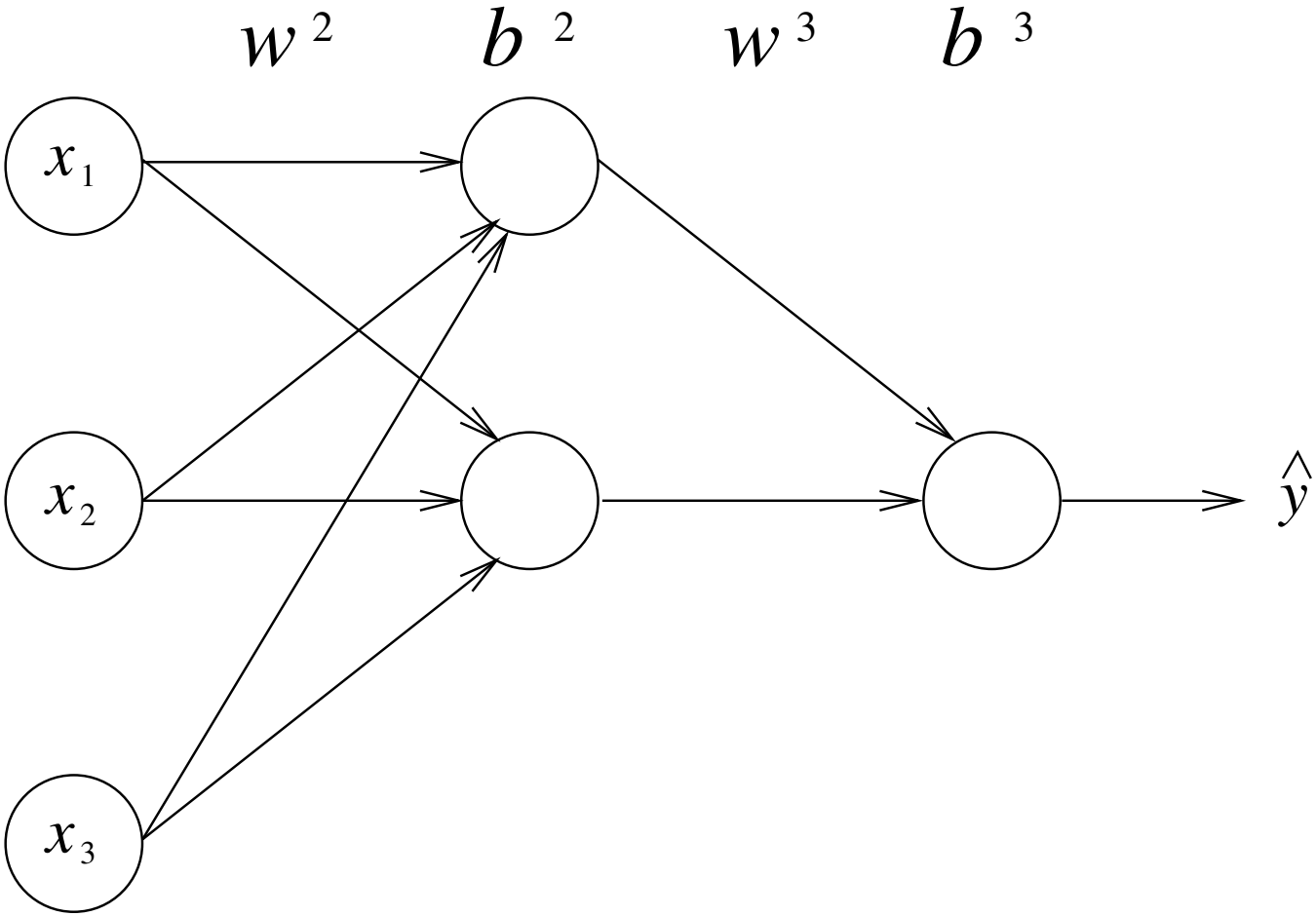

$$\sum x_i w_i + b = (-1) \times 2 + 3 \times (-1) + 1 \times 5 + 2 = 2$$

# The Simplest Neural Network Example

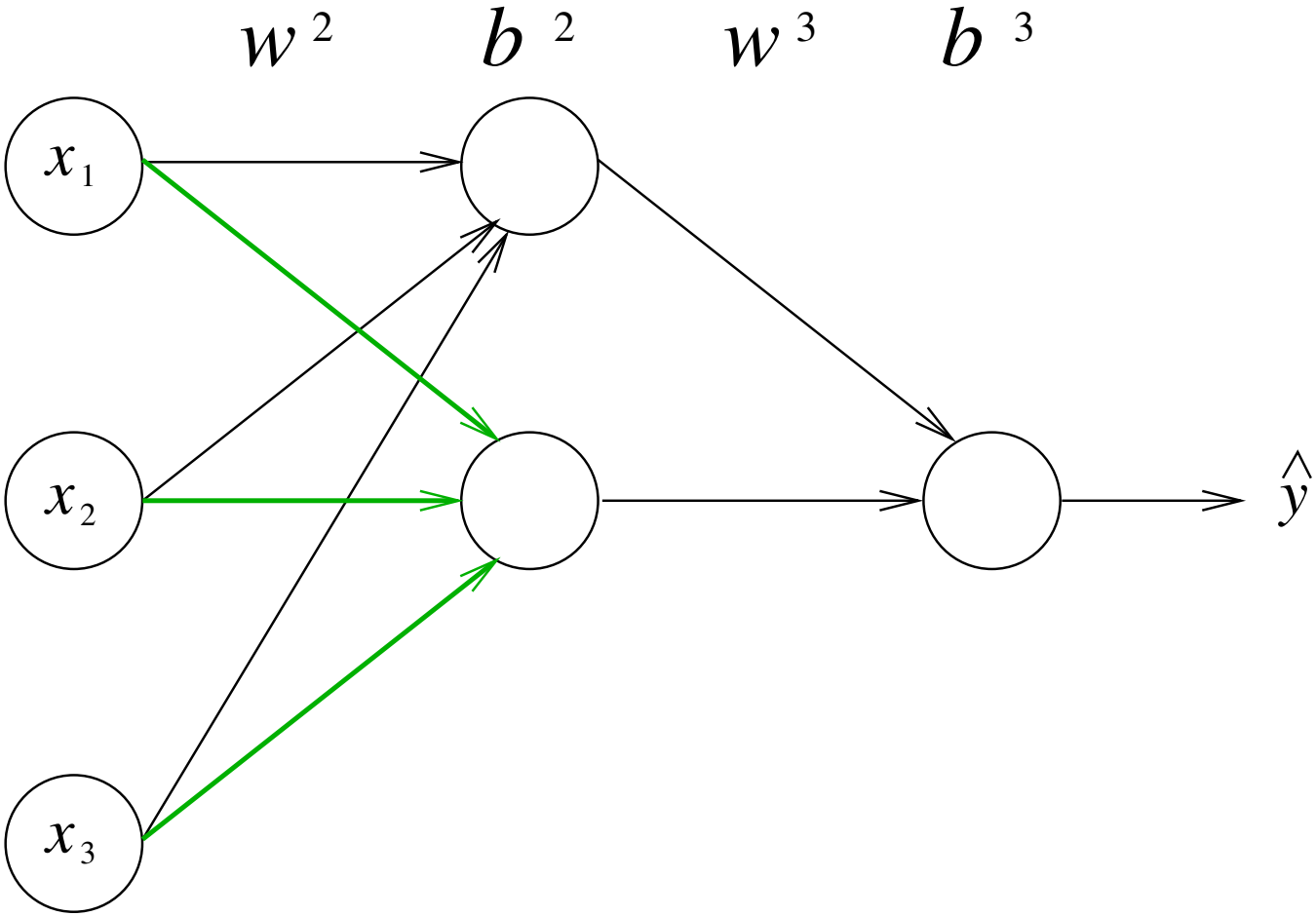



$$\sum x_i w_i + b = (-1) \times 2 + 3 \times (-1) + 1 \times 5 + 2 = 2 \implies y = 1$$

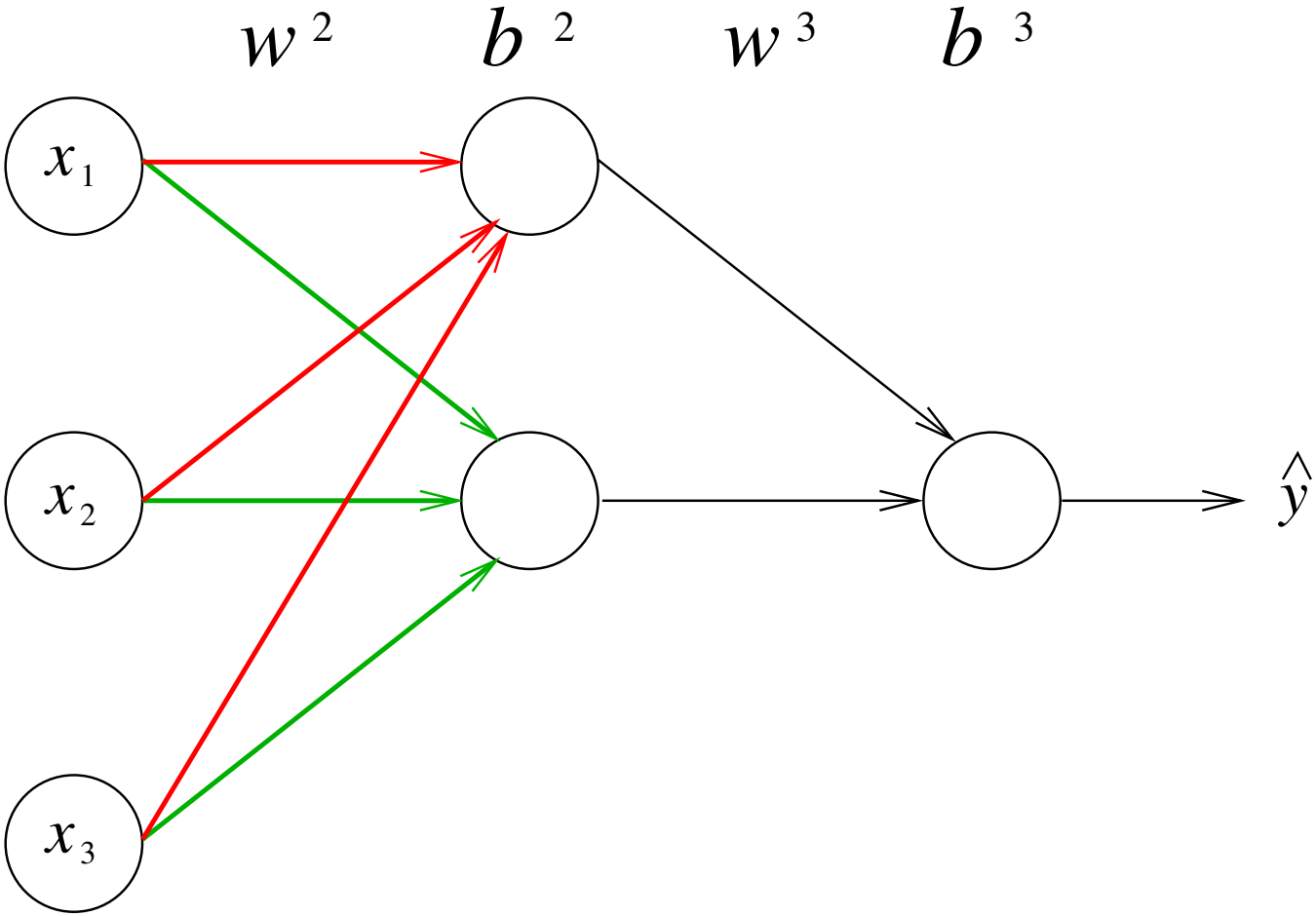
# More Complex Networks



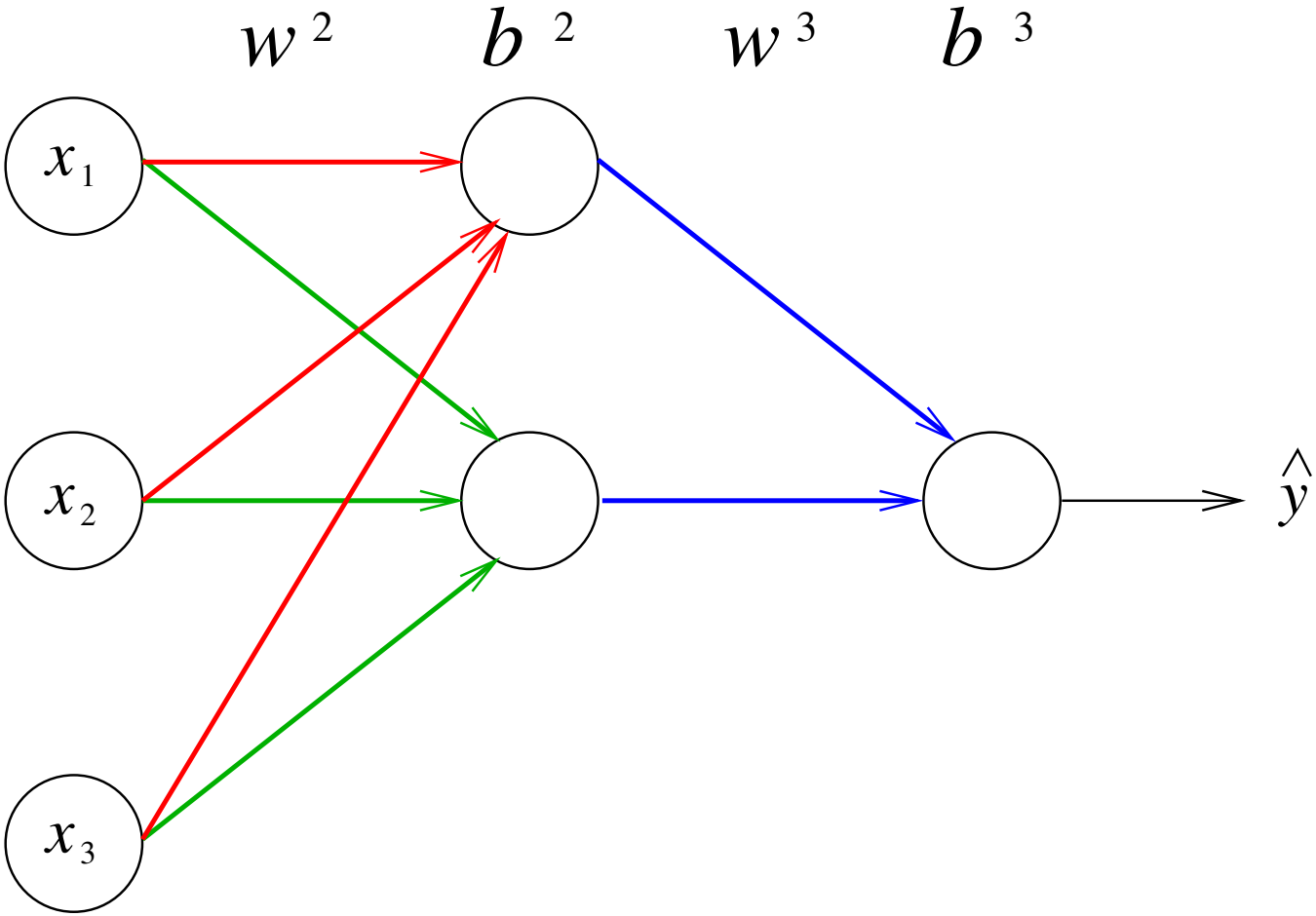
# More Complex Networks



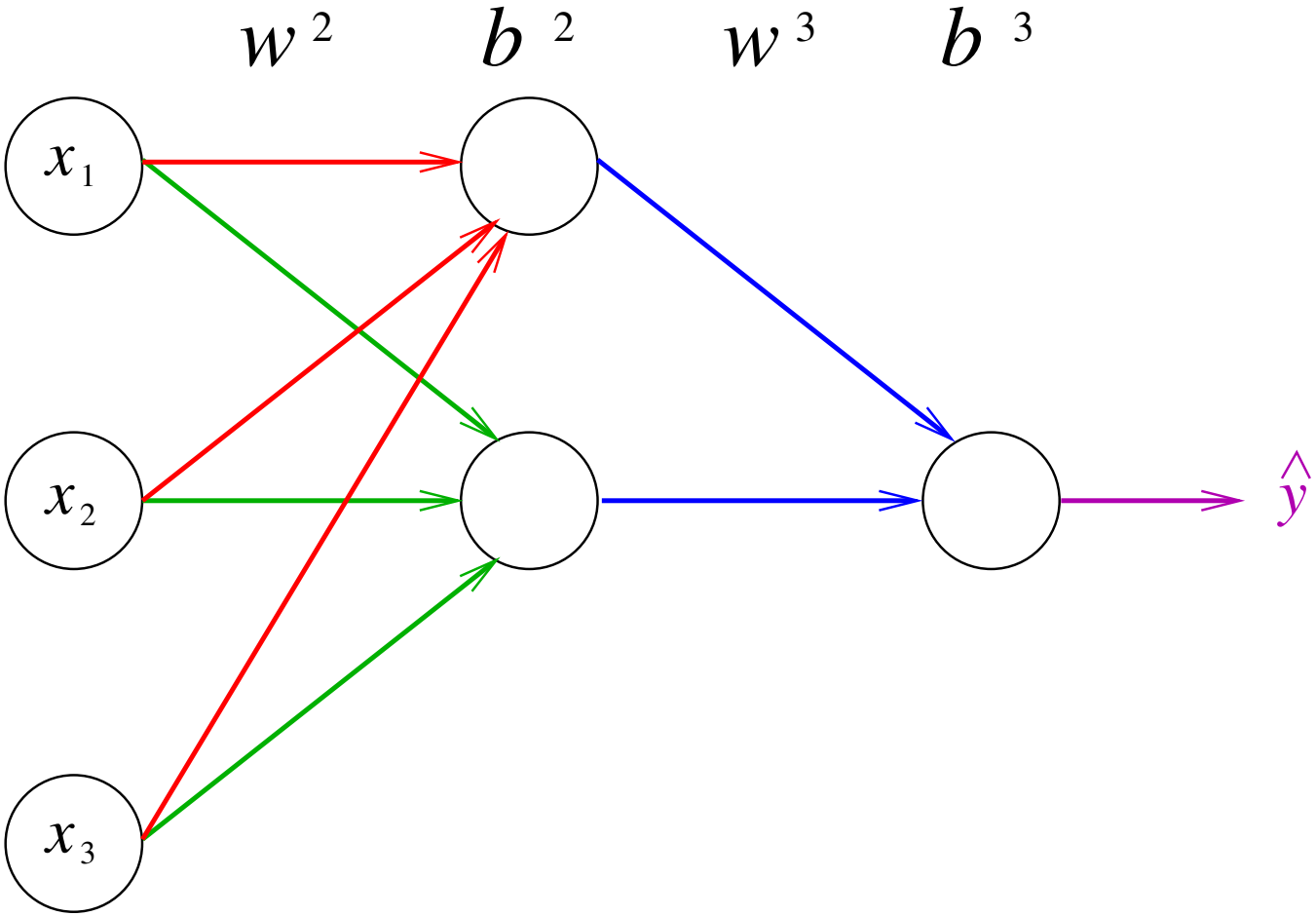
# More Complex Networks



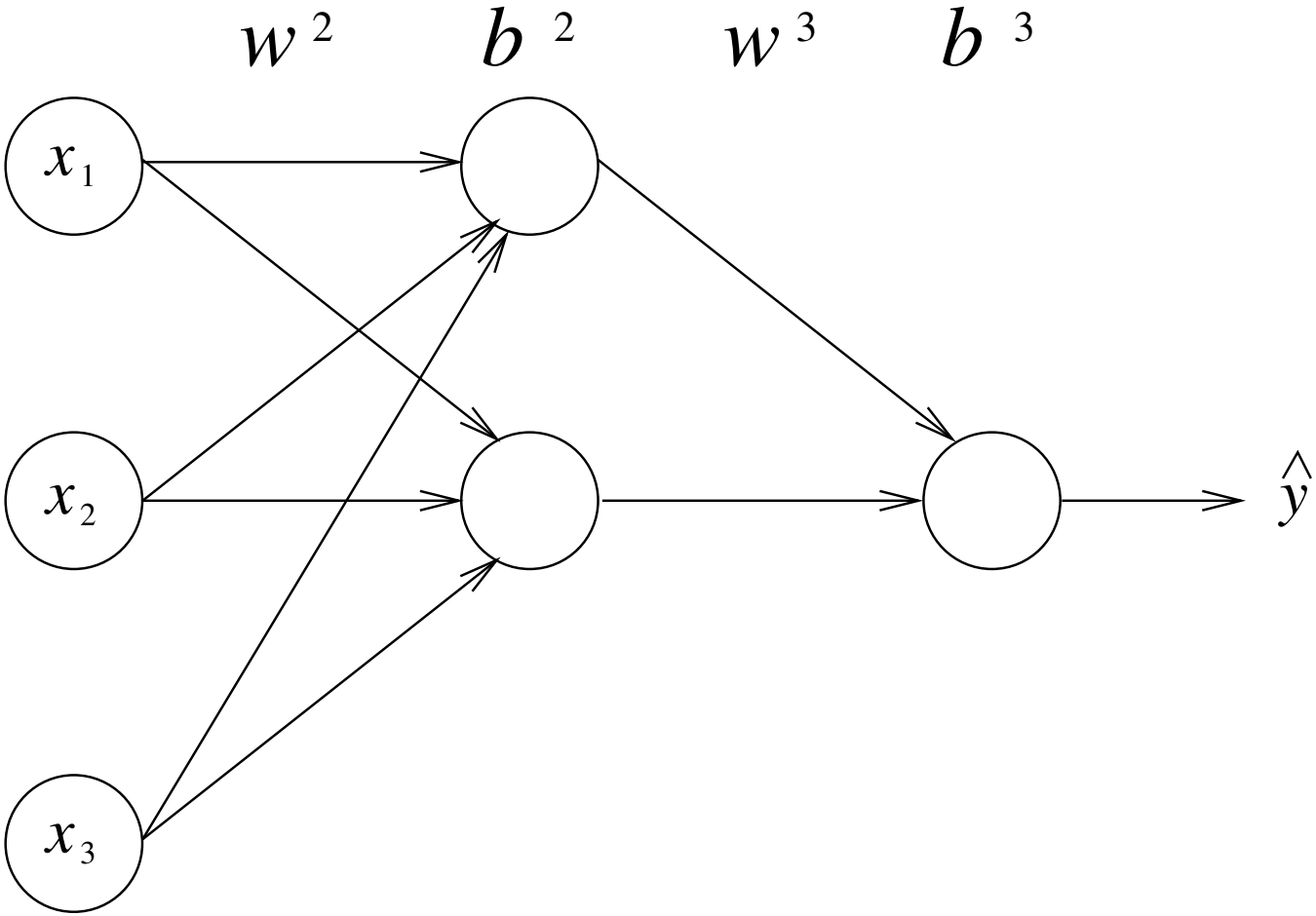
# More Complex Networks



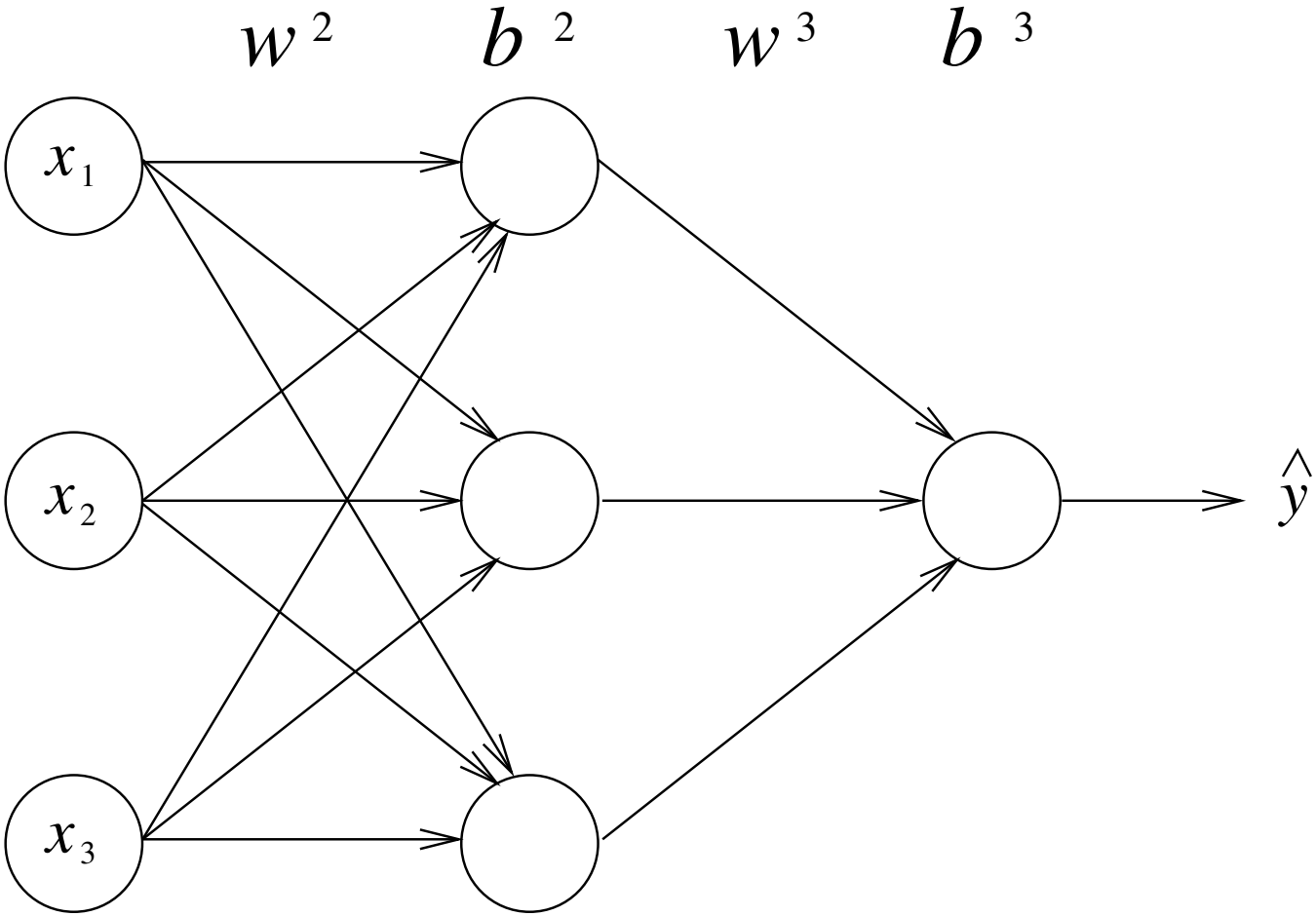
# More Complex Networks



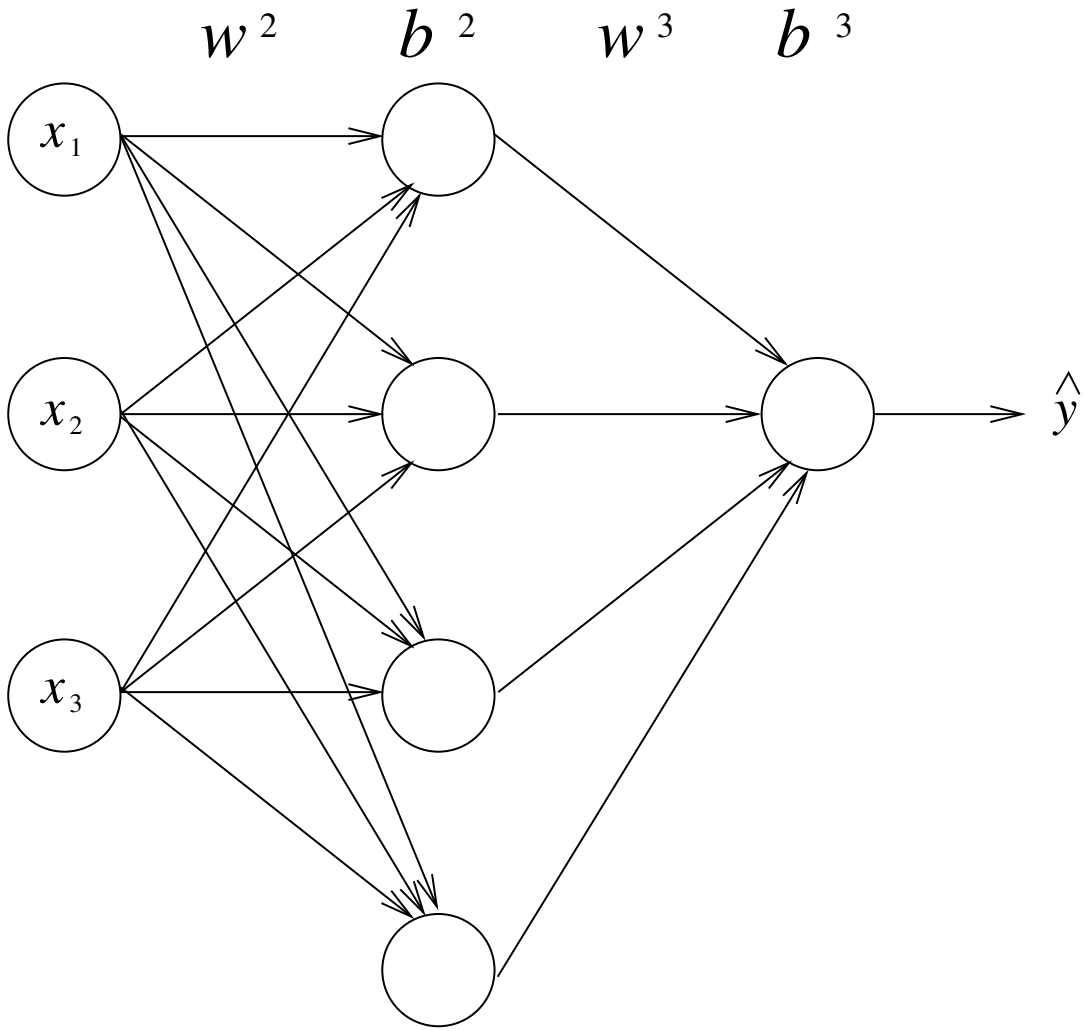
# More Complex Networks



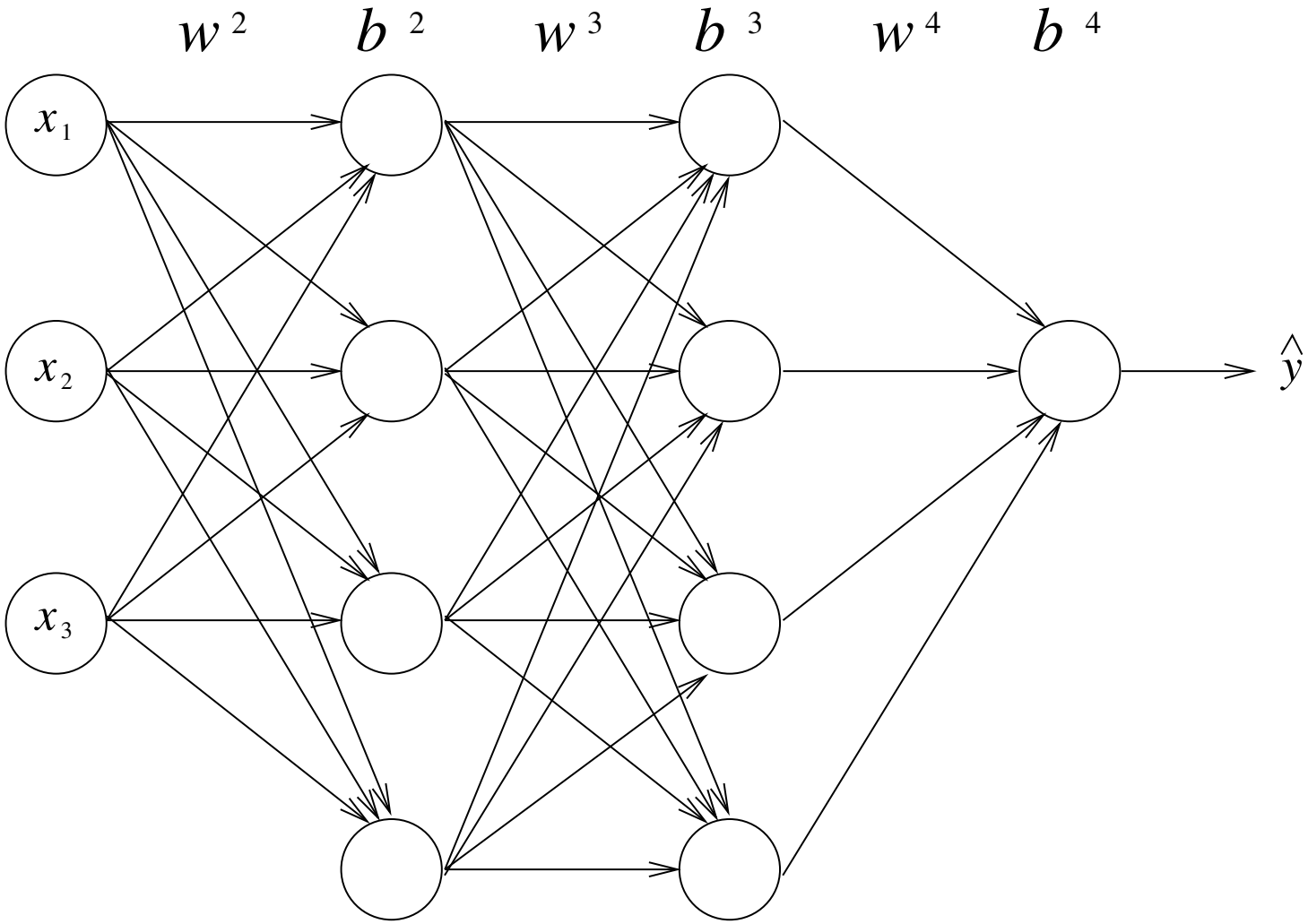
# More Complex Networks



# More Complex Networks



# More Complex Networks



# Weights in a Complex Network

$$w^{\ell} = \left[ w_{ij}^{\ell} \right]$$

where

# Weights in a Complex Network

$$w^{\ell} = \left[ w_{ij}^{\ell} \right]$$

where

- Number of rows in $w^{\ell}$ is the number of neurons in layer $\ell$

# Weights in a Complex Network

$$w^{\ell} = \left[ w_{ij}^{\ell} \right]$$

where

- Number of rows in $w^{\ell}$ is the number of neurons in layer $\ell$
- Number of cols in $w^{\ell}$ is the number of neurons in layer $\ell - 1$

# Weights in a Complex Network

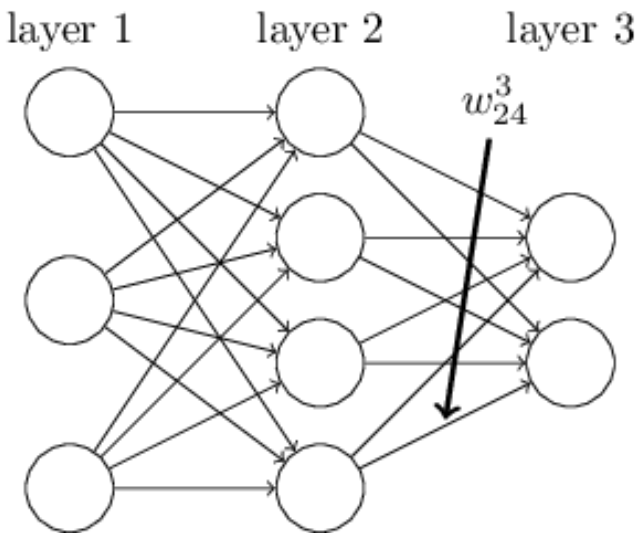
$$w^{\ell} = \left[ w_{ij}^{\ell} \right]$$

where

- Number of rows in $w^{\ell}$ is the number of neurons in layer $\ell$
- Number of cols in $w^{\ell}$ is the number of neurons in layer $\ell - 1$
- Entry $w_{ij}^{\ell}$ is the weight applied to the output from the $j$th neuron in layer $\ell - 1$ going into the $i$th neuron in layer $\ell$

# Weights in a Complex Network

$$w^\ell = \left[ w_{ij}^\ell \right]$$




where

- Number of rows in $w^\ell$ is the number of neurons in layer $\ell$
- Number of cols in $w^\ell$ is the number of neurons in layer $\ell - 1$
- Entry $w_{ij}^\ell$ is the weight applied to the output from the $j$th neuron in layer $\ell - 1$ going into the $i$th neuron in layer $\ell$

# Implementing the Network Object

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
```

# Implementing the Network Object

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
```

## Example

`Network([3, 4, 4, 1])` initializes a neural network with 3 input neurons, two hidden layers with 4 neurons each, and 1 output neuron.

# Objective of Neural Networks

## Cost function

Let $(x, y)$ be a record from our training set of size $m$ and let $\hat{y}(x)$ be the predicted output for the input $x$.

# Objective of Neural Networks

## Cost function

Let $(x, y)$ be a record from our training set of size $m$ and let $\hat{y}(x)$ be the predicted output for the input $x$.

$$E = \frac{1}{2m} \sum_{x} \|y - \hat{y}(x)\|^2$$

(Mean Squared Error *or* Quadratic Cost Function)

# Objective of Neural Networks

## Cost function

Let $(x, y)$ be a record from our training set of size $m$ and let $\hat{y}(x)$ be the predicted output for the input $x$.

$$E = \frac{1}{2m} \sum_{x} \|y - \hat{y}(x)\|^2$$

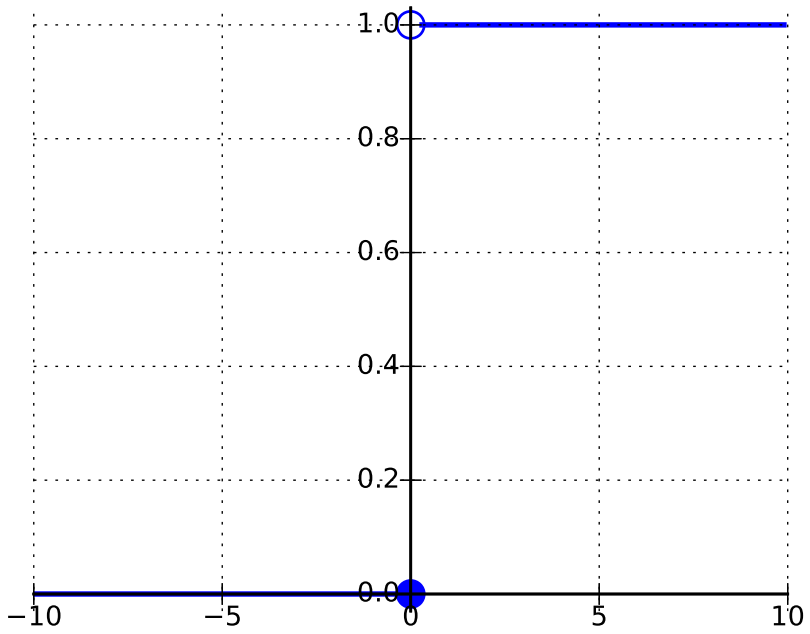
(Mean Squared Error *or* Quadratic Cost Function)

## Objective

Find weights $w^2, \ldots, w^L$ and biases $b^2, \ldots, b^L$ that minimize the Cost.

# A Better Activation Function

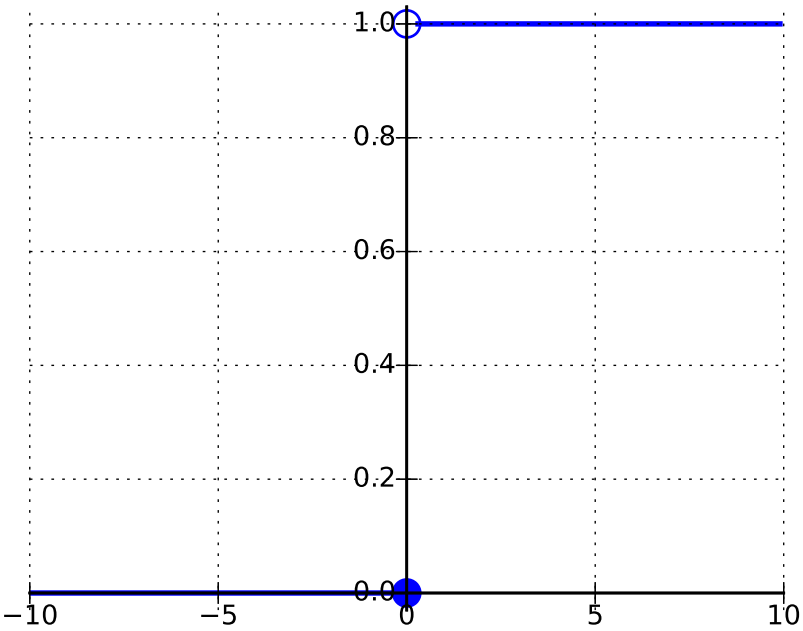
Consider the perceptron function:




Perceptron

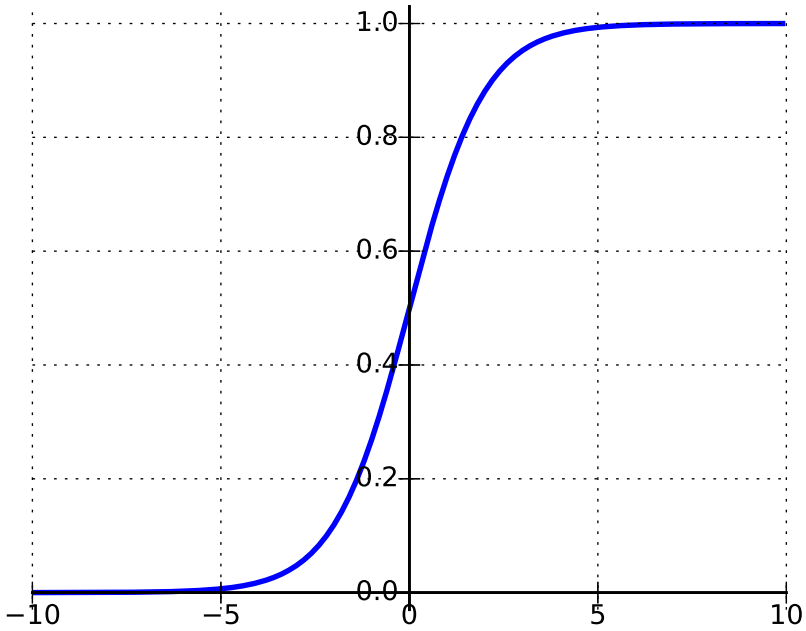
# A Better Activation Function

Consider the perceptron function:



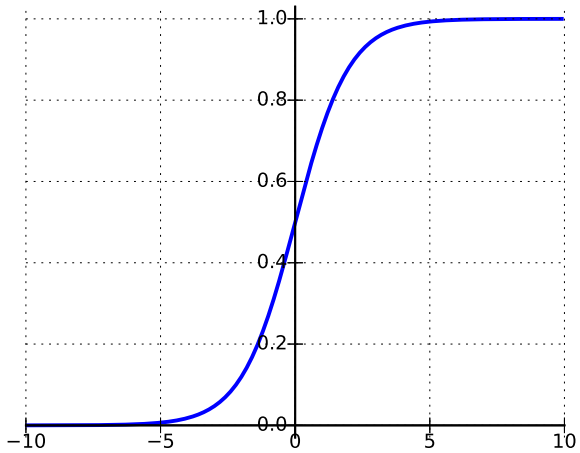

Perceptron

A continuous function is better:




Sigmoid

# Sigmoid Activation Function



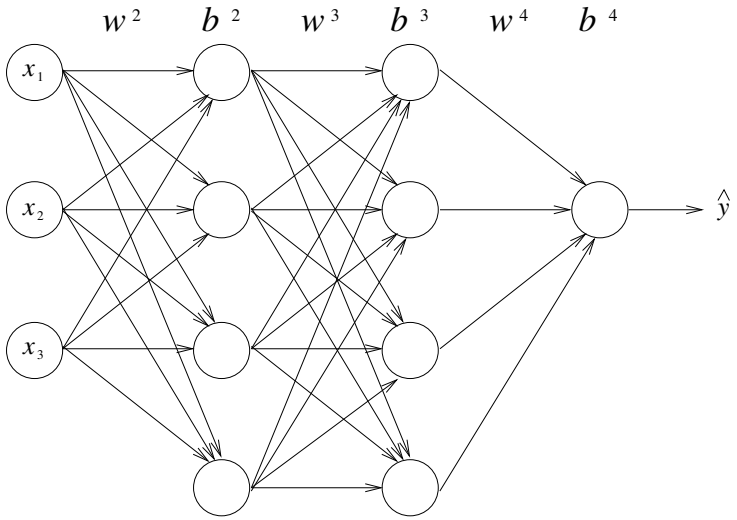

## Sigmoid

The **sigmoid (logistic) function** is defined to be

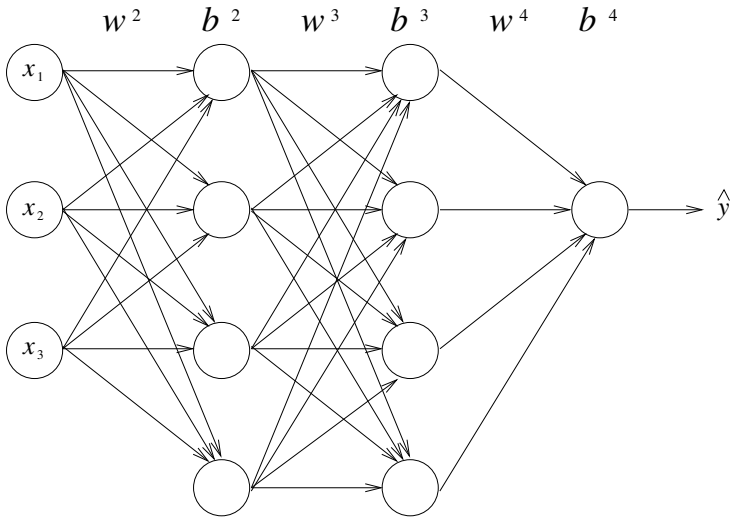
$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

The function can output any number in the range 0-1 (not inclusive).

# Feedforward in a Neural Network



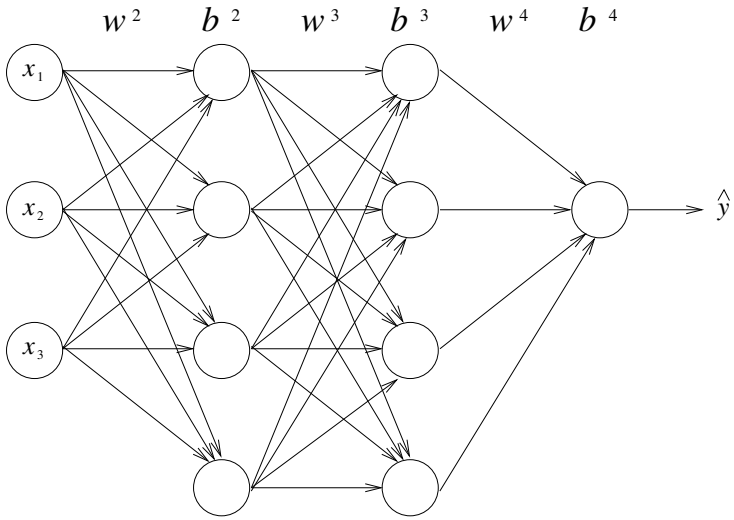
# Feedforward in a Neural Network




## Feed forward process

For an input $x$, the network predicts an output $\hat{y}(x)$

# Feedforward in a Neural Network

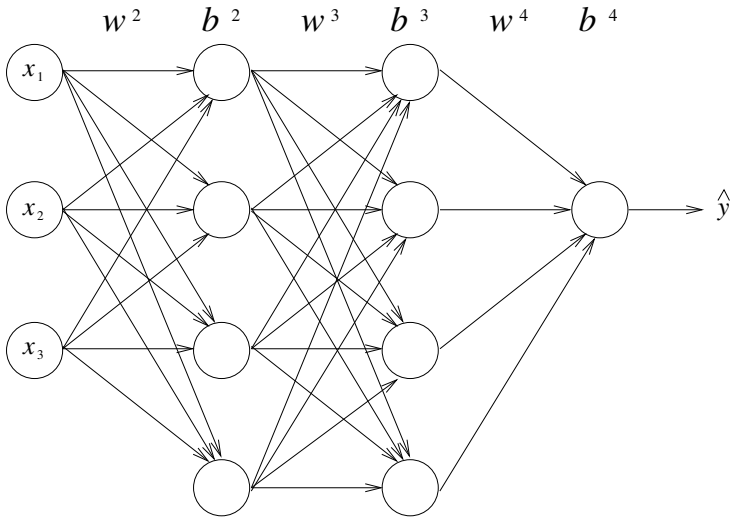



## Feed forward process

For an input $x$, the network predicts an output $\hat{y}(x)$

1. Set $a = x$

# Feedforward in a Neural Network

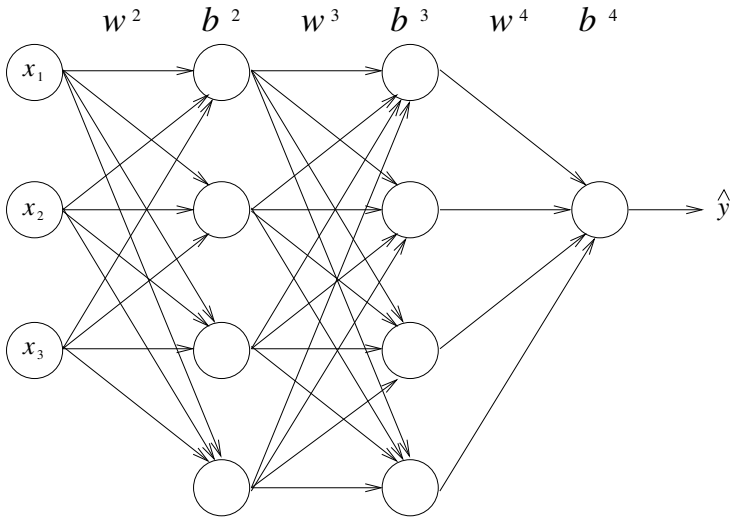



## Feed forward process

For an input $x$, the network predicts an output $\hat{y}(x)$

1. Set $a = x$
2. Set $a \leftarrow a' = \sigma(w^\ell a + b^\ell)$ for $\ell = 2 \ldots 4$

# Feedforward in a Neural Network

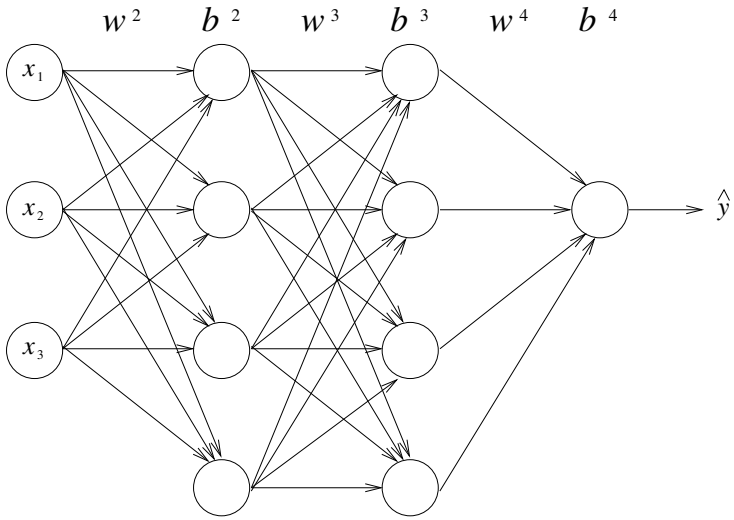



## Feed forward process

For an input $x$, the network predicts an output $\hat{y}(x)$

1. Set $a = x$
2. Set $a \leftarrow a' = \sigma(w^\ell a + b^\ell)$ for $\ell = 2 \ldots 4$ or, $\ell = 2 \ldots L$

# Feedforward in a Neural Network




## Feed forward process

For an input $x$, the network predicts an output $\hat{y}(x)$

1. Set $a = x$
2. Set $a \leftarrow a' = \sigma(w^\ell a + b^\ell)$ for $\ell = 2 \ldots 4$ or, $\ell = 2 \ldots L$
3. Output $\hat{y}(x) = a$

# Implementing Feed Forward and Sigmoid

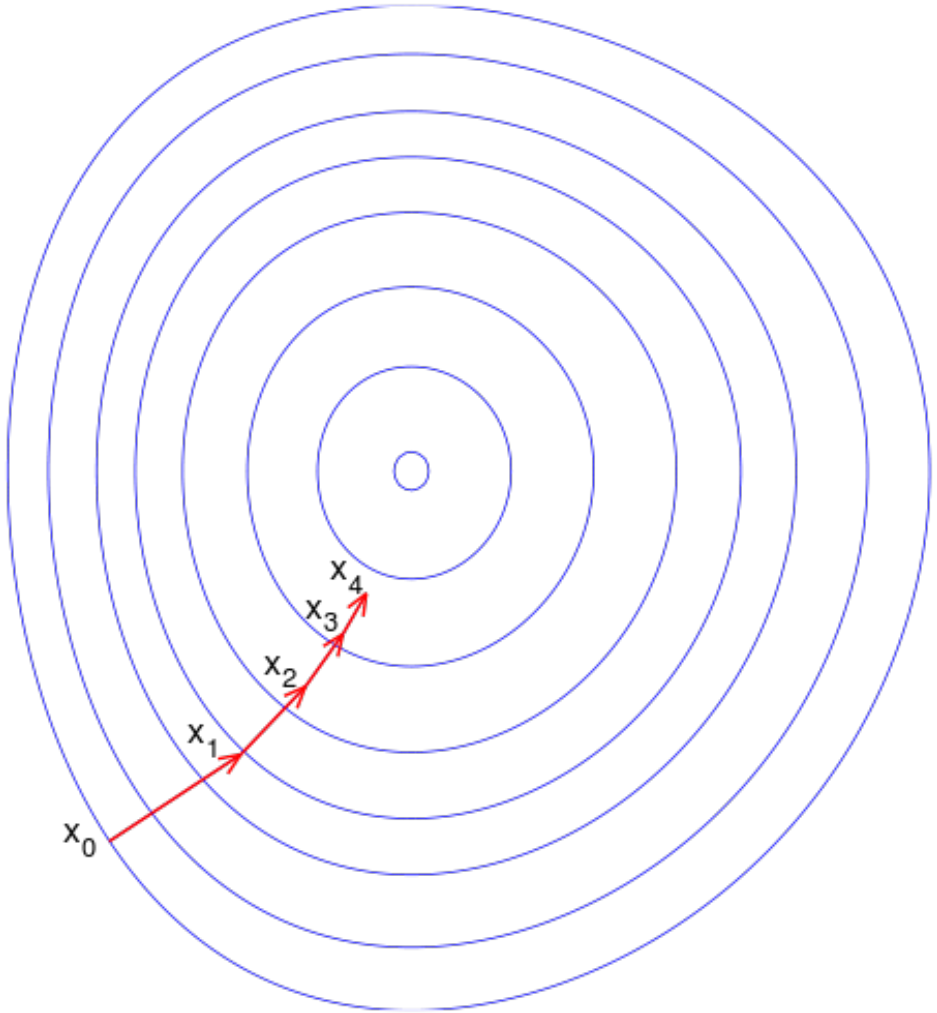
```
class Network(object):
...
    def feedforward(self, x):
        a = x
        for w, b in zip(self.weights, self.biases):
            a = sigmoid(np.dot(w, a)+b)
        return a

...
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    return sigmoid(z)*(1-sigmoid(z))
```

# Table of Contents

# Intuition for gradient descent



# General algorithm

## How does a Neural Network Learn it's weights?

For every training example $(x, y)$, we find how a change in the weights and biases effect the error:

# General algorithm

## How does a Neural Network Learn it's weights?

For every training example $(x, y)$, we find how a change in the weights and biases effect the error:

- $\frac{\partial E}{\partial w}$ for all weights $w$ in the network.

# General algorithm

## How does a Neural Network Learn it's weights?

For every training example $(x, y)$, we find how a change in the weights and biases effect the error:

- $\frac{\partial E}{\partial w}$ for all weights $w$ in the network.
- $\frac{\partial E}{\partial b}$ for all biases $b$ in the network.

# General algorithm

## How does a Neural Network Learn it's weights?

For every training example $(x, y)$, we find how a change in the weights and biases effect the error:

- $\frac{\partial E}{\partial w}$ for all weights $w$ in the network.
- $\frac{\partial E}{\partial b}$ for all biases $b$ in the network.

We update the weights and biases by subtracting the changes:

# General algorithm

## How does a Neural Network Learn it's weights?

For every training example $(x, y)$, we find how a change in the weights and biases effect the error:

- $\frac{\partial E}{\partial w}$ for all weights $w$ in the network.
- $\frac{\partial E}{\partial b}$ for all biases $b$ in the network.

We update the weights and biases by subtracting the changes:

- $w \leftarrow w' = w - \eta \frac{\partial E}{\partial w}$

# General algorithm

## How does a Neural Network Learn it's weights?

For every training example $(x, y)$, we find how a change in the weights and biases effect the error:

- $\frac{\partial E}{\partial w}$ for all weights $w$ in the network.
- $\frac{\partial E}{\partial b}$ for all biases $b$ in the network.

We update the weights and biases by subtracting the changes:

- $w \leftarrow w' = w - \eta \frac{\partial E}{\partial w}$
- $b \leftarrow b' = b - \eta \frac{\partial E}{\partial b}$

# General algorithm

## How does a Neural Network Learn it's weights?

For every training example $(x, y)$, we find how a change in the weights and biases effect the error:

- $\frac{\partial E}{\partial w}$ for all weights $w$ in the network.
- $\frac{\partial E}{\partial b}$ for all biases $b$ in the network.

We update the weights and biases by subtracting the changes:

- $w \leftarrow w' = w - \eta \frac{\partial E}{\partial w}$
- $b \leftarrow b' = b - \eta \frac{\partial E}{\partial b}$

Eta ($\eta$) is the **learning rate**

# Implementing Gradient Descent

```
class Network(object):
...
    def SGD(self, training_data, iters, eta):
        for j in range(iters):
            for x, y in training_data:
                delta_w, delta_b = self.backprop(x, y)
                self.weights = [w-eta*dw
                        for w, dw in zip(self.weights, delta_w)]
                self.biases = [b-eta*db
                        for b, db in zip(self.biases, delta_b)]
```

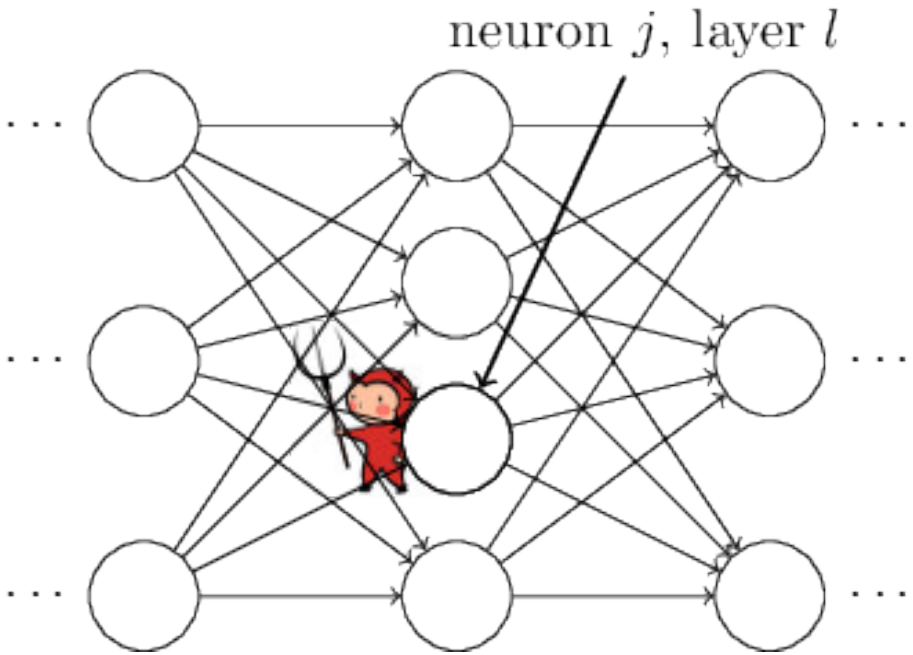
# Implementing Gradient Descent

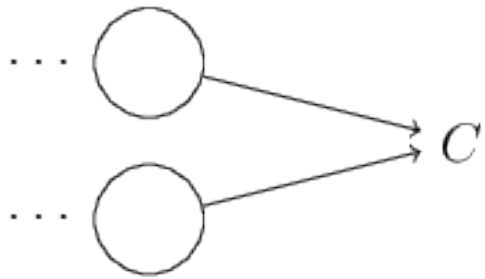
```
class Network(object):
...
    def SGD(self, training_data, iters, eta):
        for j in range(iters):
            for x, y in training_data:
                delta_w, delta_b = self.backprop(x, y)
                self.weights = [w-eta*dw
                        for w, dw in zip(self.weights, delta_w)]
                self.biases = [b-eta*db
                        for b, db in zip(self.biases, delta_b)]
```

## Stochastic Gradient Descent

The above implementation is called **Stochastic Gradient Descent**. We get the `delta_w, delta_b` from a single training example $(x, y)$.

# Backpropagation Intuition




Source: http://neuralnetworksanddeeplearning.com/chap2.html

# Backpropagation

## Equations

The backpropagation algorithm depends on the following equations:

# Backpropagation

## Equations

The backpropagation algorithm depends on the following equations:

1. $$\frac{\partial E}{\partial(\text{input; final layer})} = (\hat{y}(x) - y) \times \sigma'(\text{input; final layer})$$

# Backpropagation

## Equations

The backpropagation algorithm depends on the following equations:

1. $$\frac{\partial E}{\partial(\text{input; final layer})} = (\hat{y}(x) - y) \times \sigma'(\text{input; final layer})$$
2. $$\frac{\partial E}{\partial(\text{input; layer } \ell)} = w^{\ell+1}\left(\frac{\partial E}{\partial(\text{input; layer } \ell+1)}\right) \odot \sigma'(\text{input; layer } \ell)$$

# Backpropagation

## Equations

The backpropagation algorithm depends on the following equations:

1. $$\frac{\partial E}{\partial(\text{input; final layer})} = (\hat{y}(x) - y) \times \sigma'(\text{input; final layer})$$
2. $$\frac{\partial E}{\partial(\text{input; layer } \ell)} = w^{\ell+1}\left(\frac{\partial E}{\partial(\text{input; layer } \ell+1)}\right) \odot \sigma'(\text{input; layer } \ell)$$
3. $$\frac{\partial E}{\partial w^{\ell}_{jk}} = \frac{\partial E}{\partial(\text{input; layer } \ell)} \times (\text{output; layer } \ell - 1)$$

# Backpropagation

## Equations

The backpropagation algorithm depends on the following equations:

1. $$\frac{\partial E}{\partial(\text{input; final layer})} = (\hat{y}(x) - y) \times \sigma'(\text{input; final layer})$$
2. $$\frac{\partial E}{\partial(\text{input; layer } \ell)} = w^{\ell+1}\left(\frac{\partial E}{\partial(\text{input; layer } \ell+1)}\right) \odot \sigma'(\text{input; layer } \ell)$$
3. $$\frac{\partial E}{\partial w^{\ell}_{jk}} = \frac{\partial E}{\partial(\text{input; layer } \ell)} \times (\text{output; layer } \ell - 1)$$
4. $$\frac{\partial E}{\partial b^{\ell}_{j}} = \frac{\partial E}{\partial(\text{input; layer } \ell)}$$

# Backpropagation Implementation

```
class Network(object):
...
    def backprop(self, x, y):
        delta_w = [np.zeros(w.shape) for w in self.weights]
        delta_b = [np.zeros(b.shape) for b in self.biases]
        # Feed it forward
        layer_out = [x] # Stores output from all layers
        layer_in_prime = [] # Stores sigmoid' of input to all layers
        for w, b in zip(self.weights, self.biases):
            z = np.dot(w, layer_out[-1])+b # Get input to next layer
            layer_out.append(sigmoid(z))
            layer_in_prime.append(sigmoid_prime(z))
        # Take it back
        delta = (layer_out[-1]-y) * layer_in_prime[-1]
        delta_w[-1] = np.dot(delta, layer_out[-2].transpose())
        delta_b[-1] = delta
        for l in range(2, self.num_layers):
            sp = layer_in_prime[-l]
            delta = np.dot(self.weights[-l+1].transpose(), delta)*sp
            delta_w[-l] = np.dot(delta, layer_out[-l-1].transpose())
            delta_b[-l] = delta
        return delta_w, delta_b
```

# First Equation

1. $$\frac{\partial E}{\partial(\text{input; final layer})} = (\hat{y}(x) - y) \times \sigma'(\text{input; final layer})$$

```
def backprop(self, x, y):
...
    layer_out = [x] # Stores output from all layers
    layer_in_prime = [] # Stores sigmoid' of input to all layers
...
    delta = (layer_out[-1]-y) * layer_in_prime[-1]
```

# Second Equation

2. $$\frac{\partial E}{\partial(\text{input; layer } \ell)} = w^{\ell+1}\left(\frac{\partial E}{\partial(\text{input; layer } \ell+1)}\right) \odot \sigma'(\text{input; layer } \ell)$$

```
def backprop(self, x, y):
...
    layer_in_prime = [] # Stores sigmoid' of input to all layers
...
    for l in range(2, self.num_layers):
        sp = layer_in_prime[-l]
        delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
```

# Third Equation

3. $$\frac{\partial E}{\partial w_{jk}^{\ell}} = \frac{\partial E}{\partial (\text{input; layer } \ell)} \times (\text{output; layer } \ell - 1)$$

```
def backprop(self, x, y):
...
    layer_out = [x] # Stores output from all layers
...
    delta_w[-1] = np.dot(delta, layer_out[-2].transpose())
    for l in range(2, self.num_layers):
    ...
        delta_w[-l] = np.dot(delta, layer_out[-l-1].transpose())
```

# Fourth Equation

4. $$\frac{\partial E}{\partial b_j^\ell} = \frac{\partial E}{\partial(\text{input; layer } \ell)}$$

```
def backprop(self, x, y):
...
    delta_b[-1] = delta
    for l in range(2, self.num_layers):
    ...
        delta_b[-l] = delta
```

# Code

Let's run the code:

https://github.com/rarredon/PyDataTalk/blob/master/src/Intro%20to%20Neural%20Networks%20Talk.ipynb

# Table of Contents

# A New Error Function

## Cross-Entropy Function

Let $(x, y)$ be a record from our training set of size $m$ and let $\hat{y}(x)$ be the predicted output for the input $x$.

# A New Error Function

## Cross-Entropy Function

Let $(x, y)$ be a record from our training set of size $m$ and let $\hat{y}(x)$ be the predicted output for the input $x$.

$$E = -\frac{1}{m}\sum_{x}[y \ln(\hat{y}) + (1-y)\ln(1-\hat{y})]$$

# Cross-entropy effects on Backpropagation

Cross entropy has an effect on the backpropagation algorithm.

# Cross-entropy effects on Backpropagation

Cross entropy has an effect on the backpropagation algorithm. In particular,

1. $$\frac{\partial E}{\partial(\text{input; final layer})} = (\hat{y}(x) - y) \times \sigma'(\text{input; final layer})$$

# Cross-entropy effects on Backpropagation

Cross entropy has an effect on the backpropagation algorithm. In particular,

1. $$\frac{\partial E}{\partial(\text{input; final layer})} = (\hat{y}(x) - y) \times \sigma'(\text{input; final layer})$$

becomes

1. $$\frac{\partial E}{\partial(\text{input; final layer})} = (\hat{y}(x) - y)$$

# Cross-Entropy Implementation

In the code we change

```
def backprop(self, x, y):
...
    delta = (layer_out[-1]-y) * layer_in_prime[-1]
```

to

```
def backprop(self, x, y):
...
    delta = (layer_out[-1]-y)
```

# Regularization

## $L_2$ Regularization

To keep weights small, while minimizing the cost, we alter the cost:

- $E = -\frac{1}{m}\sum_x [y \ln(\hat{y}(x)) + (1-y)\ln(1-\hat{y}(x))] + \frac{\lambda}{2m}\sum_w w^2$

# Effect of $L_2$ Regularization Gradient Descent

$L_2$ regularization has an effect on gradient descent.

# Effect of $L_2$ Regularization Gradient Descent

$L_2$ regularization has an effect on gradient descent. In particular,

1. $w \leftarrow w' = w - \eta \frac{\partial E}{\partial w}$

# Effect of $L_2$ Regularization Gradient Descent

$L_2$ regularization has an effect on gradient descent. In particular,

1. $w \leftarrow w' = w - \eta \frac{\partial E}{\partial w}$

becomes

# Effect of $L_2$ Regularization Gradient Descent

$L_2$ regularization has an effect on gradient descent. In particular,

1. $w \leftarrow w' = w - \eta \frac{\partial E}{\partial w}$

becomes

1. $w \leftarrow w' = \left(1 - \frac{\eta\lambda}{m}\right) w - \eta \frac{\partial E}{\partial w}$

# Cross-Entropy Implementation

In the code we change

```
def SGD(self, training_data, iters, eta):
...
            self.weights = [w-eta*dw
                    for w, dw in zip(self.weights, delta_w)]
            self.biases = [b-eta*db
                    for b, db in zip(self.biases, delta_b)]
```

to

```
def SGD(self, training_data, iters, eta):
...
            self.weights = [(1-eta*(lmbda/m))w-eta*dw
                    for w, dw in zip(self.weights, delta_w)]
            self.biases = [b-eta*db
                    for b, db in zip(self.biases, delta_b)]
```

# Other ways to avoid overfitting

Some additional ways to avoid overfitting:

# Other ways to avoid overfitting

Some additional ways to avoid overfitting:

- **$L_1$ regularization:** like $L_2$ but uses $\sum_w |w|$ instead of $\sum_w w^2$.

# Other ways to avoid overfitting

Some additional ways to avoid overfitting:

- **$L_1$ regularization:** like $L_2$ but uses $\sum_w |w|$ instead of $\sum_w w^2$.
- **Dropout method:** train several networks with some of the nerurons missing and then combine these into one network.

# Other ways to avoid overfitting

Some additional ways to avoid overfitting:

- **$L_1$ regularization:** like $L_2$ but uses $\sum_w |w|$ instead of $\sum_w w^2$.
- **Dropout method:** train several networks with some of the nerurons missing and then combine these into one network.
- **More data:** Obtain more data or artificially create more data.

# Faster computational speeds

## Mini-batch Stochastic Gradient Descent

## Weight initialization

# Faster computational speeds

## Mini-batch Stochastic Gradient Descent

Update the weights and biases for a mini-batch:

## Weight initialization

# Faster computational speeds

## Mini-batch Stochastic Gradient Descent

Update the weights and biases for a mini-batch:

- $w \leftarrow w' = w - \eta \sum_x \frac{\partial E_x}{\partial w}$

## Weight initialization

# Faster computational speeds

## Mini-batch Stochastic Gradient Descent

Update the weights and biases for a mini-batch:

- $w \leftarrow w' = w - \eta \sum_x \frac{\partial E_x}{\partial w}$

## Weight initialization

# Faster computational speeds

## Mini-batch Stochastic Gradient Descent

Update the weights and biases for a mini-batch:

- $w \leftarrow w' = w - \eta \sum_x \frac{\partial E_x}{\partial w}$
- $b \leftarrow b' = b - \eta \sum_x \frac{\partial E_x}{\partial b}$

## Weight initialization

# Faster computational speeds

## Mini-batch Stochastic Gradient Descent

Update the weights and biases for a mini-batch:

- $w \leftarrow w' = w - \eta \sum_x \frac{\partial E_x}{\partial w}$
- $b \leftarrow b' = b - \eta \sum_x \frac{\partial E_x}{\partial b}$

## Weight initialization

- Weights initialized using standard normal distribution.

# Faster computational speeds

## Mini-batch Stochastic Gradient Descent

Update the weights and biases for a mini-batch:

- $w \leftarrow w' = w - \eta \sum_x \frac{\partial E_x}{\partial w}$
- $b \leftarrow b' = b - \eta \sum_x \frac{\partial E_x}{\partial b}$

## Weight initialization

- Weights initialized using standard normal distribution.
- Better results if variance is inversely propotional to the number input neurons, i.e., $\sigma^2 = 1/n_{\text{in}}$.

# Other Optimization Methods

## Variations on Gradient Descent Algorithm

# Other Optimization Methods

## Variations on Gradient Descent Algorithm

- Hessian Technique

# Other Optimization Methods

## Variations on Gradient Descent Algorithm

- Hessian Technique
- Momentum-based gradient descent

# Other Optimization Methods

## Variations on Gradient Descent Algorithm

- Hessian Technique
- Momentum-based gradient descent
- Conjugate gradient descent

# Other Optimization Methods

## Variations on Gradient Descent Algorithm

- Hessian Technique
- Momentum-based gradient descent
- Conjugate gradient descent
- BFGS Method

# Other Optimization Methods

## Variations on Gradient Descent Algorithm

- Hessian Technique
- Momentum-based gradient descent
- Conjugate gradient descent
- BFGS Method

For a reference that introduces and compares these methods see "Efficient BackProp" by LeCun et. al.

# Table of Contents

# Summary of Hyper-parameters

- **Iterations:** Number of iterations to train the network on.

# Summary of Hyper-parameters

- **Iterations:** Number of iterations to train the network on.
- **Eta ($\eta$):** The learning rate; sets step size for gradient descent.

# Summary of Hyper-parameters

- **Iterations:** Number of iterations to train the network on.
- **Eta ($\eta$):** The learning rate; sets step size for gradient descent.
- **Lambda ($\lambda$):** Regularization parameter; a larger value favors smaller weights.

# Summary of Hyper-parameters

- **Iterations:** Number of iterations to train the network on.
- **Eta ($\eta$):** The learning rate; sets step size for gradient descent.
- **Lambda ($\lambda$):** Regularization parameter; a larger value favors smaller weights.
- **mini-batch-size:** Number of training examples used to perform gradient descent update.

# Summary of Hyper-parameters

- **Iterations:** Number of iterations to train the network on.
- **Eta ($\eta$):** The learning rate; sets step size for gradient descent.
- **Lambda ($\lambda$):** Regularization parameter; a larger value favors smaller weights.
- **mini-batch-size:** Number of training examples used to perform gradient descent update.
- **network size:** Number of layers and number of neurons in each.

# MNIST Best Results

## Deep, yet Simple

In a 2010 paper, "Deep, Big, Simple Neural Networks Excel on Handwritten Digit Recognition", the authors trained a network whose hidden layers were $2500, 2000, 1500, 1000,$ and 500 neurons, respectively. Their classification accuracy was a resounding 99.65%.

# MNIST Best Results

## Deep, yet Simple

In a 2010 paper, "Deep, Big, Simple Neural Networks Excel on Handwritten Digit Recognition", the authors trained a network whose hidden layers were $2500, 2000, 1500, 1000,$ and 500 neurons, respectively. Their classification accuracy was a resounding 99.65%.

## Deep Convolutional

In 2012, the same authors trained a convolutional neural network with a classification accuracy of 99.77%. [a]

[a] http://repository.supsi.ch/5145/1/IDSIA-04-12.pdf

# Convolutional Neural Networks (CNNs)

- CNNs are Neural Networks with a special structure.

# Convolutional Neural Networks (CNNs)

- CNNs are Neural Networks with a special structure.
- Their structure exploits spatial invariance.

# Convolutional Neural Networks (CNNs)

- CNNs are Neural Networks with a special structure.
- Their structure exploits spatial invariance.
- They typically require less parameters than ordinary deep Neural Networks; not every neuron in one layer is connected is connected to every neuron in the next layer; also, many of the neurons use shared weights.

# Convolutional Neural Networks (CNNs)

- CNNs are Neural Networks with a special structure.
- Their structure exploits spatial invariance.
- They typically require less parameters than ordinary deep Neural Networks; not every neuron in one layer is connected is connected to every neuron in the next layer; also, many of the neurons use shared weights.
- Certain techniques have become standard in their use without a theoretical backing or understanding.

# Convolutional Neural Networks (CNNs)

- CNNs are Neural Networks with a special structure.
- Their structure exploits spatial invariance.
- They typically require less parameters than ordinary deep Neural Networks; not every neuron in one layer is connected is connected to every neuron in the next layer; also, many of the neurons use shared weights.
- Certain techniques have become standard in their use without a theoretical backing or understanding.
- They consistently win the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC).

# Future of Neural Networks

# Future of Neural Networks

?

# References

The following references were a huge help in preparing this talk:


1. Michael A. Nielsen, “Neural Networks and Deep Learning”, Determination Press, 2015. Retrieved from http://neuralnetworksanddeeplearning.com/
2. Ian Goodfellow and Yoshua Bengio and Aaron Courville, “Deep Learning”, MIT Press, 2016. Retrieved from http://www.deeplearningbook.org/

# Thanks!

Many thanks to Chris Luiz and Bob Mickus for organizing the meetup.

Thank you for coming to watch me talk.

Materials available on my github account:

https://github.com/rarredon/PyDataTalk

## Questions?