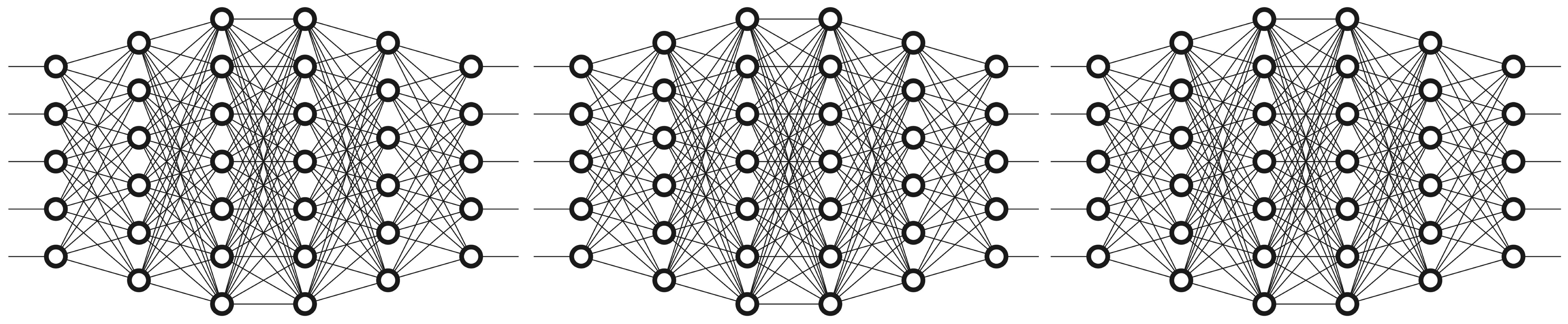
THE UNIVERSITY OF BRITISH COLUMBIA

CPSC 425: Computer Vision



Lecture 21: Neural Networks Intro

Recall: Linear Classifier

Defines a score function:

$$f(\mathbf{x}_i, \mathbf{W}, \mathbf{b}) = \mathbf{W}\mathbf{x}_i + \mathbf{b}$$

image features

weights (parameters)

bias vector

Image Credit: Ioannis (Yannis) Gkioulekas (CMU)

Recall: Linear Classifier

Example with an image with 4 pixels, and 3 classes (cat/dog/ship)

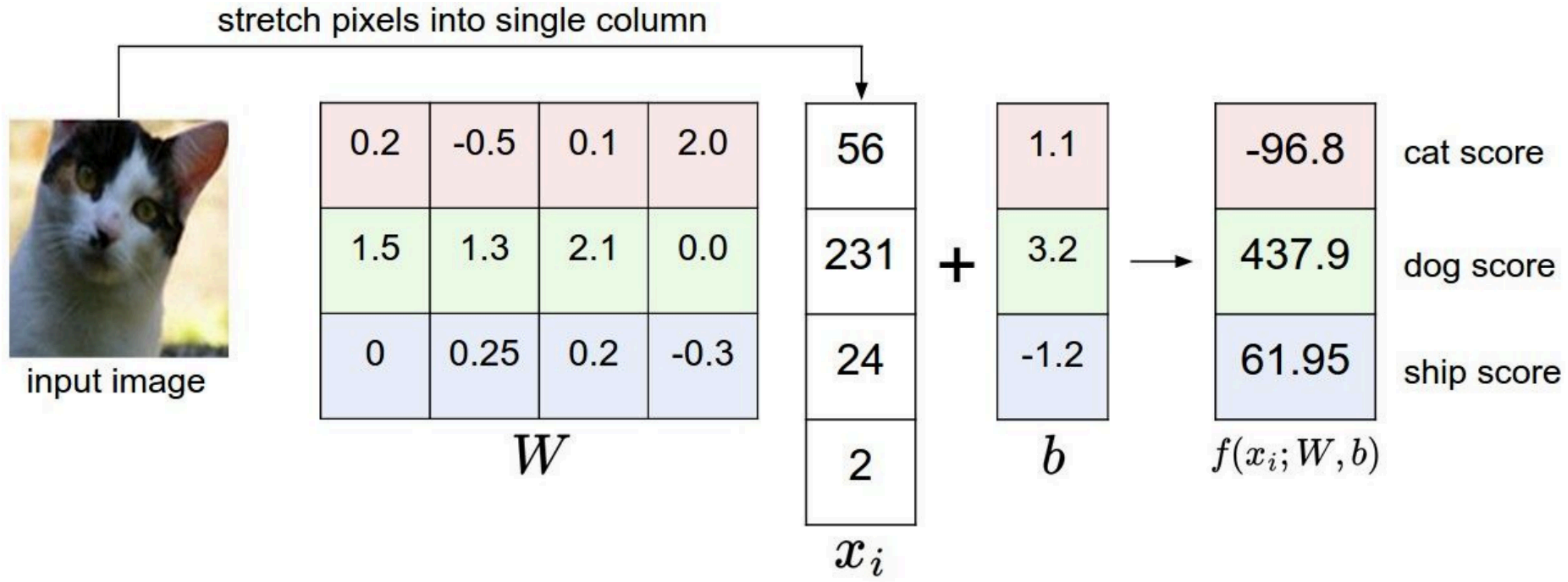
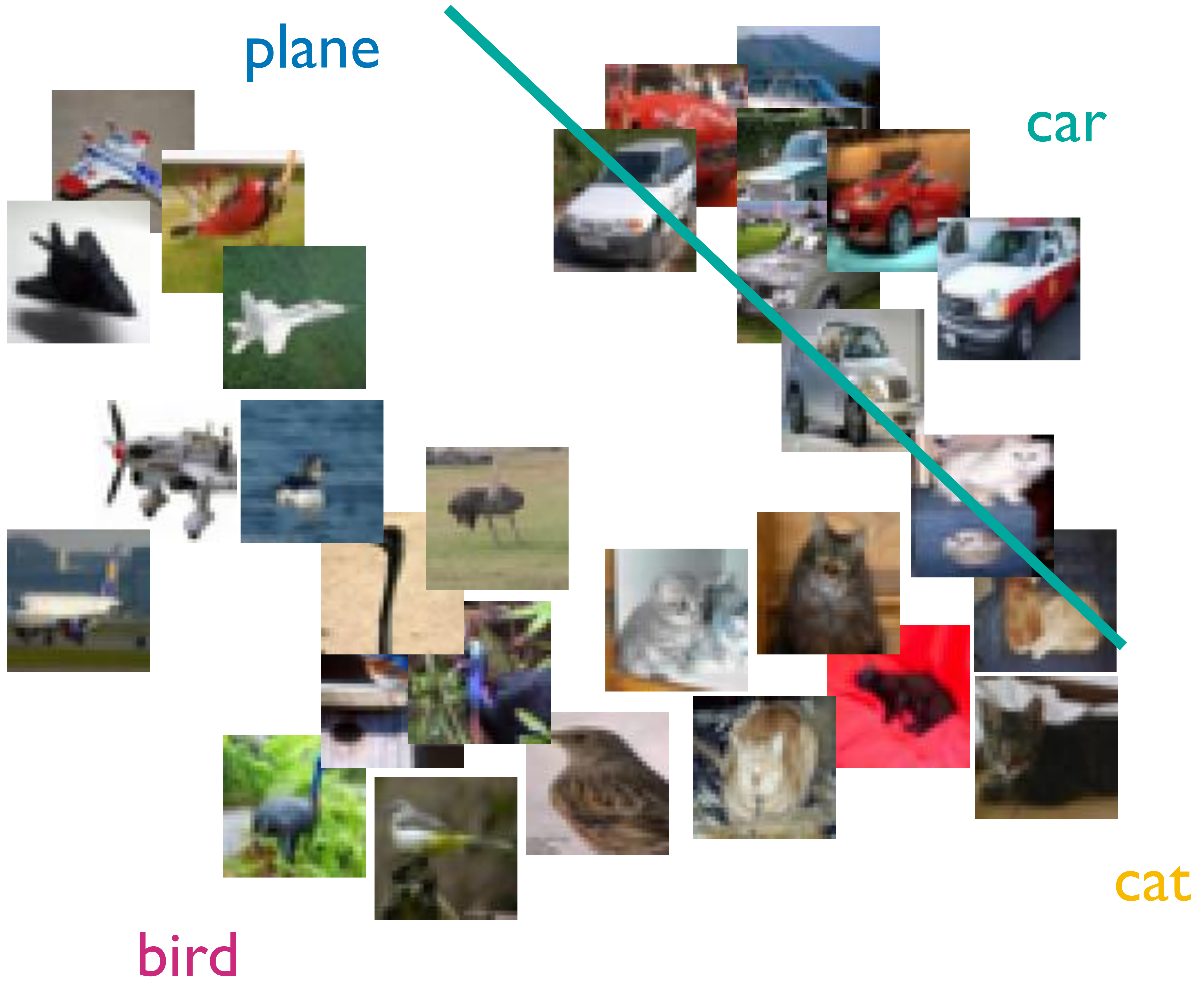


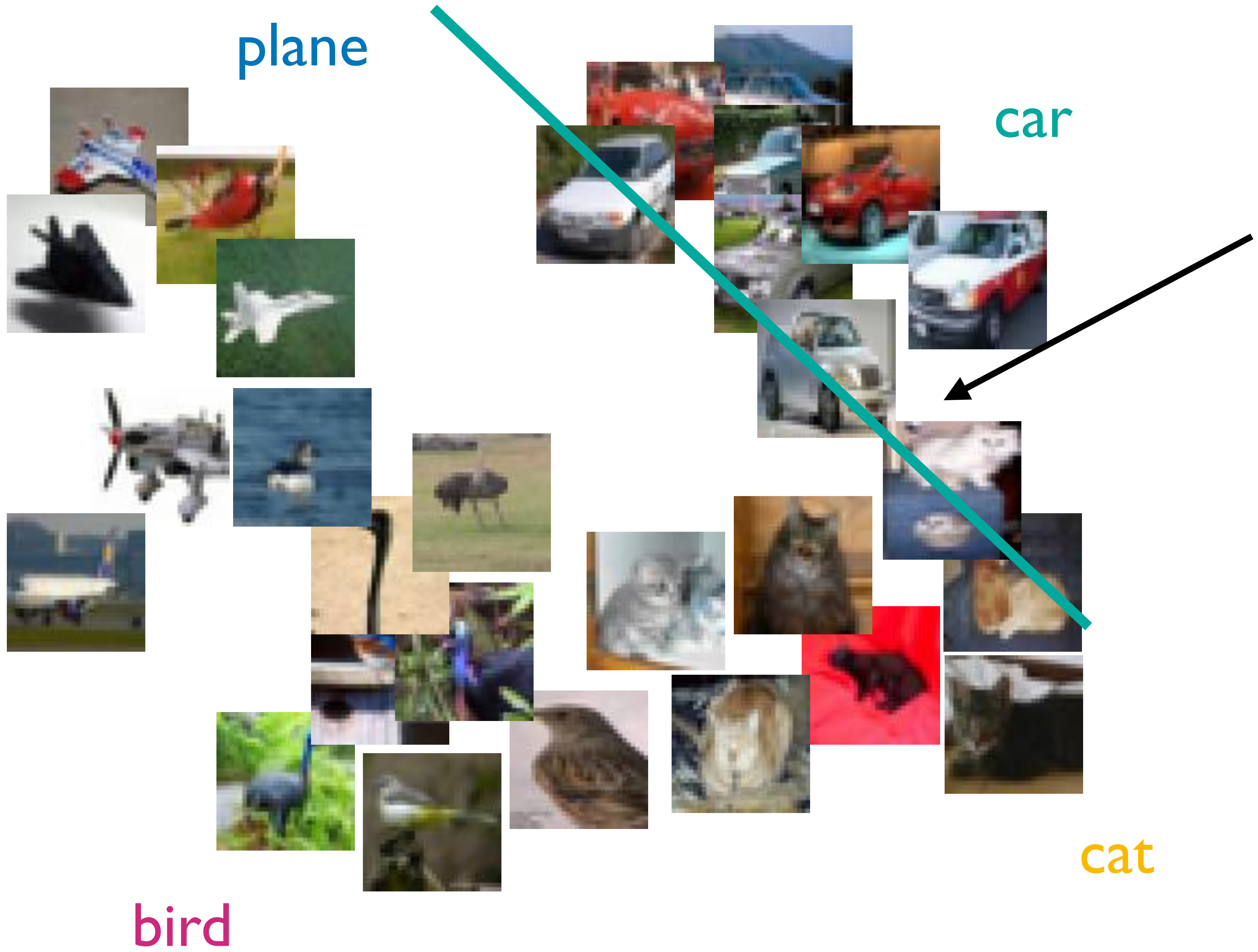
Image Credit: Ioannis (Yannis) Gkioulekas (CMU)

1-vs-All **Linear SVM**

1-vs-All **Linear SVM**

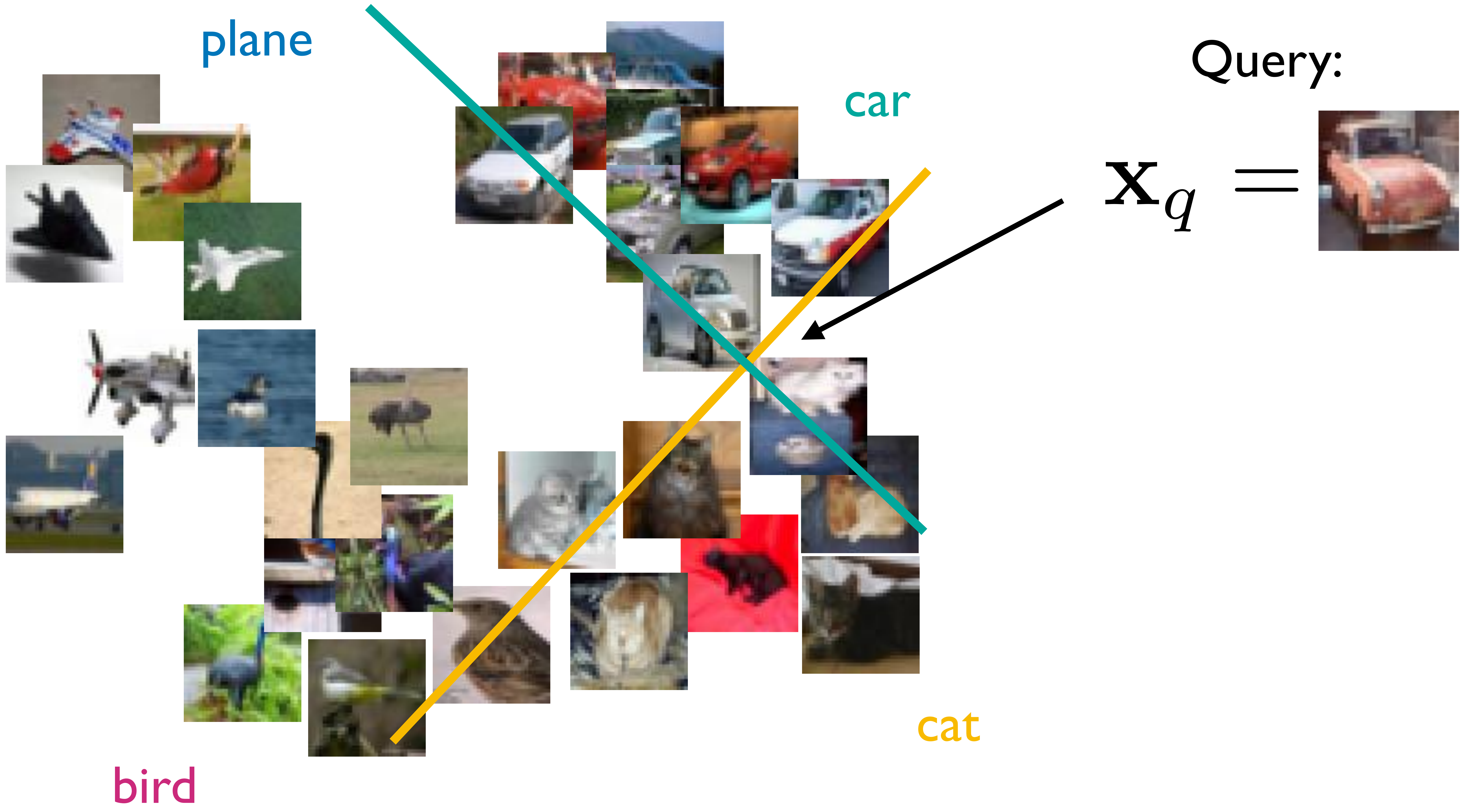


1-vs-All **Linear SVM**

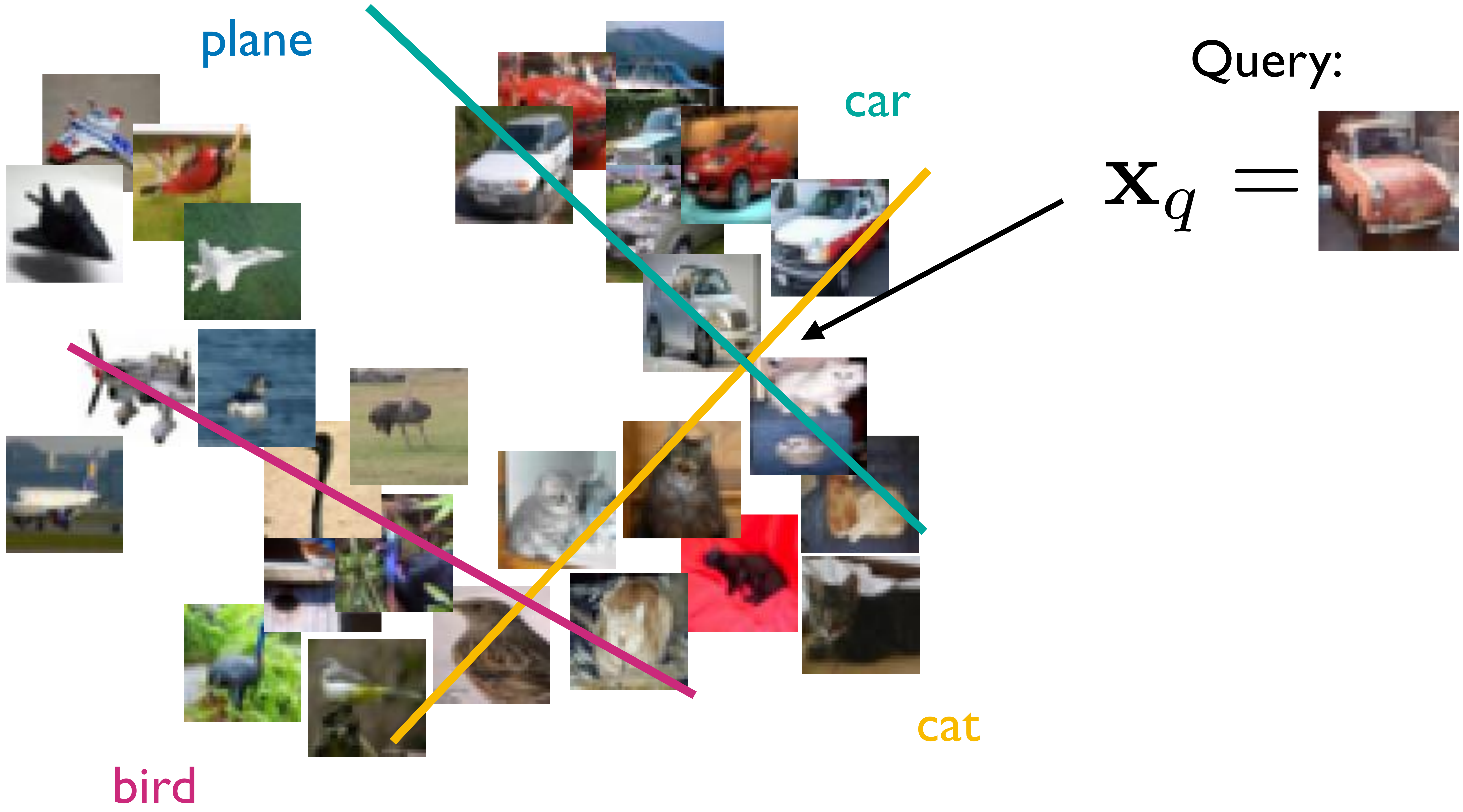


$$\mathbf{x}_q =$$

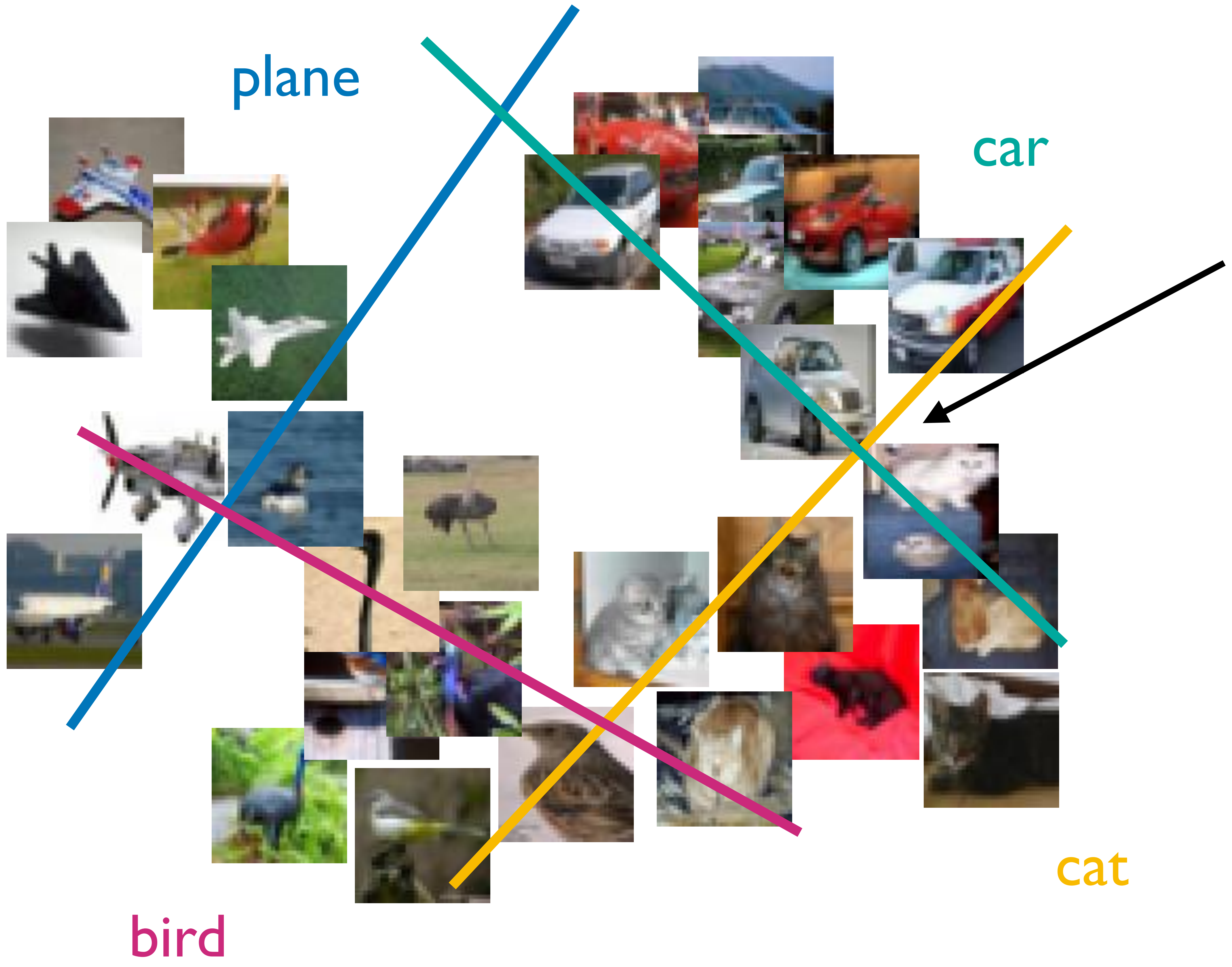
1-vs-All **Linear SVM**



1-vs-All **Linear SVM**



1-vs-All **Linear SVM**



$\mathbf{x}_q =$ 

One-hot Regression

An alternative solution is to regress to one-hot targets = 1 vs all classifiers

$$\left[\quad \mathbf{W}^T \quad \right] \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \cdots \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ \cdots \end{bmatrix}$$

← class 2 = 'automobile'

$$\left[\quad \mathbf{W}^T \quad \right] \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \cdots \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ \cdots \end{bmatrix}$$

← class 4 = 'cat'

One-hot Regression

Transpose


$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots \\ x_{21} & x_{22} & x_{23} & \dots \\ x_{31} & x_{32} & x_{33} & \dots \\ & \dots & & \end{bmatrix} \begin{bmatrix} \\ \mathbf{W} \\ \\ \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & \dots \\ 0 & 0 & 0 & 1 & \dots \\ & .. & .. & & \end{bmatrix} \begin{matrix} \text{auto} \\ \text{cat} \\ \\ \end{matrix}$$

$$\mathbf{XW} = \mathbf{T}$$

One-hot Regression

Transpose

32 x 32 x 3 = 3072

training images

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots \\ x_{21} & x_{22} & x_{23} & \dots \\ x_{31} & x_{32} & x_{33} & \dots \\ & \dots & & \end{bmatrix} \begin{bmatrix} \\ \quad \mathbf{W} \quad \\ \\ \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & \dots \\ 0 & 0 & 0 & 1 & \dots \\ & .. & .. & & \end{bmatrix} \begin{matrix} \text{auto} \\ \text{cat} \\ \\ \end{matrix}$$

$$\mathbf{X}\mathbf{W} = \mathbf{T}$$

One-hot Regression

Transpose

training images

32 x 32 x 3 = 3072

classes

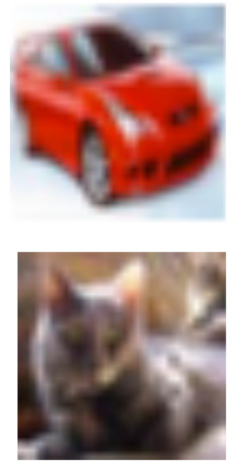
3072

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots \\ x_{21} & x_{22} & x_{23} & \dots \\ x_{31} & x_{32} & x_{33} & \dots \\ & \dots & & \end{bmatrix} \begin{bmatrix} \mathbf{W} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & \dots \\ 0 & 0 & 0 & 1 & \dots \\ & .. & .. & & \end{bmatrix} \begin{matrix} \text{auto} \\ \text{cat} \\ \ \end{matrix}$$

$$\mathbf{XW} = \mathbf{T}$$

One-hot Regression

Transpose



training images

32 x 32 x 3 = 3072

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & \cdots \\ x_{21} & x_{22} & x_{23} & \cdots \\ x_{31} & x_{32} & x_{33} & \cdots \\ & \cdots & & \end{bmatrix} \begin{bmatrix} \\ \mathbf{W} \\ \\ \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & \ldots \\ 0 & 0 & 0 & 1 & \ldots \\ & .. & .. & & \end{bmatrix}$$

classes

3072

auto
cat

$$\mathbf{XW} = \mathbf{T}$$

Solve regression problem by Least Squares

$$\mathcal{L} = |\mathbf{XW} - \mathbf{T}|^2$$

One-hot Regression

Transpose

32 x 32 x 3 = 3072

training images

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots \\ x_{21} & x_{22} & x_{23} & \dots \\ x_{31} & x_{32} & x_{33} & \dots \\ & \dots & & \end{bmatrix} \begin{bmatrix} \mathbf{W} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & \dots \\ 0 & 0 & 0 & 1 & \dots \\ & .. & .. & & \end{bmatrix} \begin{matrix} \text{auto} \\ \text{cat} \\ \end{matrix}$$

classes

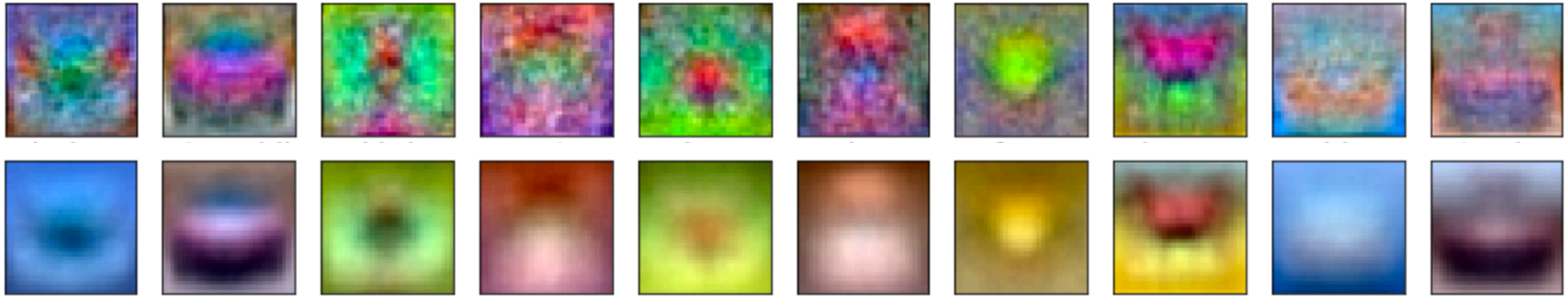
3072

$$\mathbf{XW} = \mathbf{T}$$

Solve regression problem by Least Squares

$$\mathcal{L} = |\mathbf{XW} - \mathbf{T}|^2 + \lambda|\mathbf{W}|^2$$

One-hot Regression



Solve regression problem by Least Squares

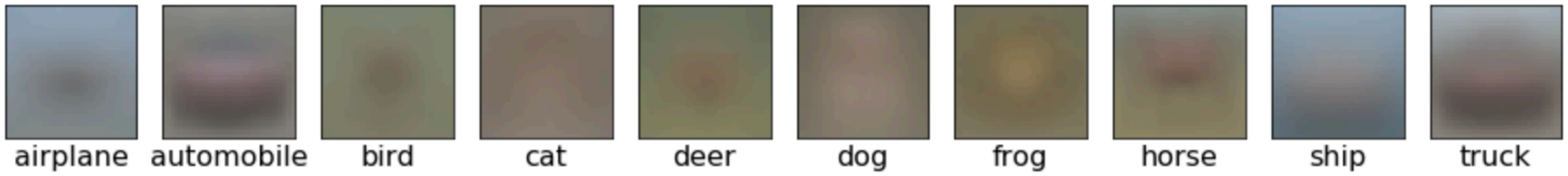
$$\mathcal{L} = |\mathbf{XW} - \mathbf{T}|^2 + \lambda|\mathbf{W}|^2$$

Recall: **Nearest Mean** Classifier

Find the nearest mean and assign class:

$$c_q = \arg\min_i |\mathbf{x}_q - \mathbf{m}_i|^2$$

CIFAR10 class means:

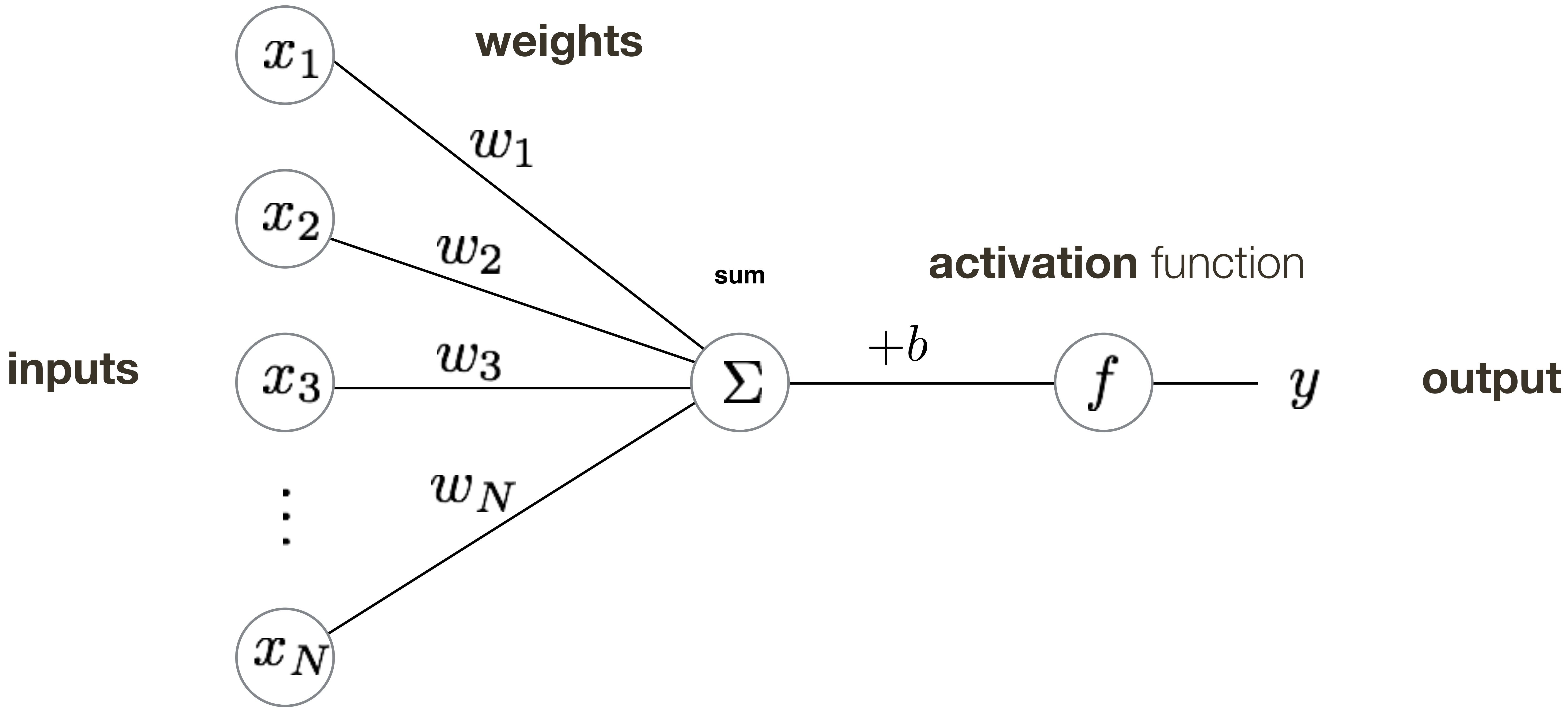


Warning:

Our intro to **Neural Networks** will be very light weight …

… if you want to know more, take my **CPSC 532S**

A **Neuron**

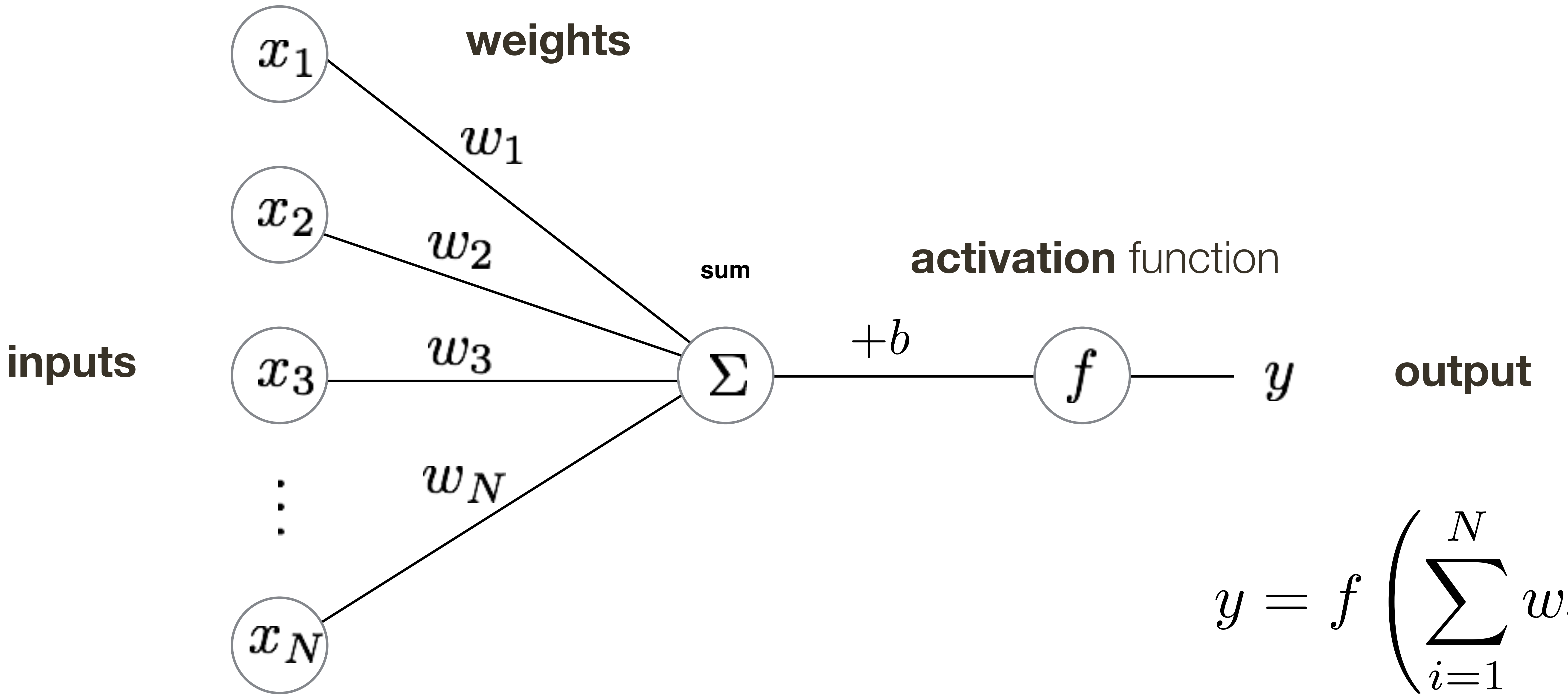


— The basic unit of computation in a neural network is a neuron.

— A neuron accepts some number of input signals, computes their weighted sum, and applies an **activation function** (or **non-linearity**) to the sum.

— Common activation functions include sigmoid and rectified linear unit (ReLU)

A **Neuron**



$$y = f\left(\sum_{i=1}^{N} w_i x_i + b\right)$$

— The basic unit of computation in a neural network is a neuron.

— A neuron accepts some number of input signals, computes their weighted sum, and applies an **activation function** (or **non-linearity**) to the sum.

— Common activation functions include sigmoid and rectified linear unit (ReLU)

Recall: Linear Classifier

Defines a score function:

$$f(\mathbf{x}_i, \mathbf{W}, \mathbf{b}) = \mathbf{W}\mathbf{x}_i + \mathbf{b}$$

image features

weights (parameters)

bias vector

Image Credit: Ioannis (Yannis) Gkioulekas (CMU)

Recall: Linear Classifier

Example with an image with 4 pixels, and 3 classes (cat/dog/ship)

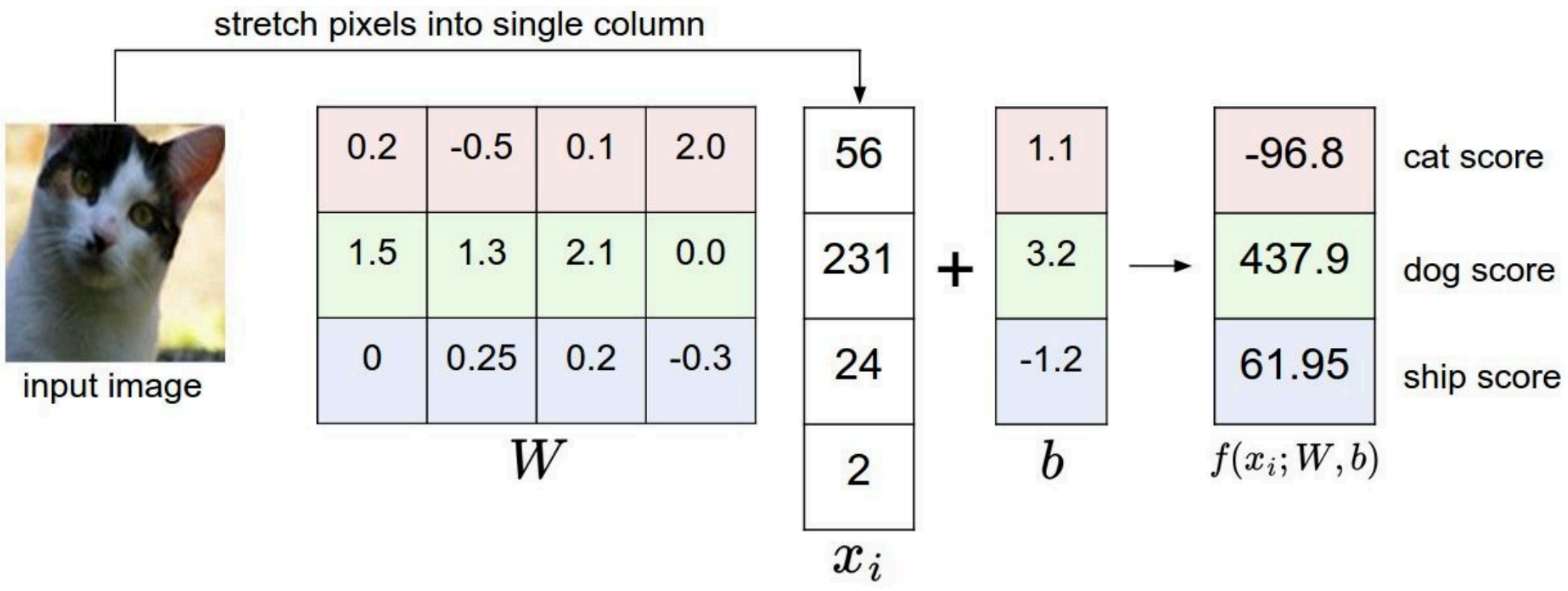
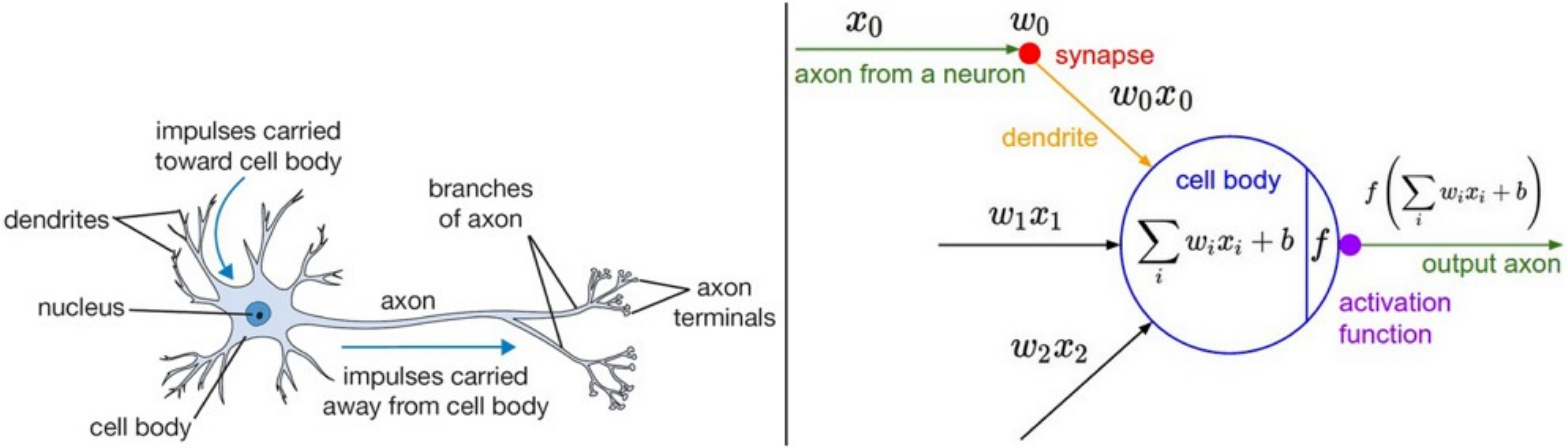


Image Credit: Ioannis (Yannis) Gkioulekas (CMU)

Aside: Inspiration from Biology

Figure credit: Fei-Fei and Karpathy



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

Neural nets/perceptrons are loosely inspired by biology.

But they certainly are not a model of how the brain works, or even how neurons work.

Activation Function: **Sigmoid**

$$f(x) = 1/(1 + e^{-x})$$

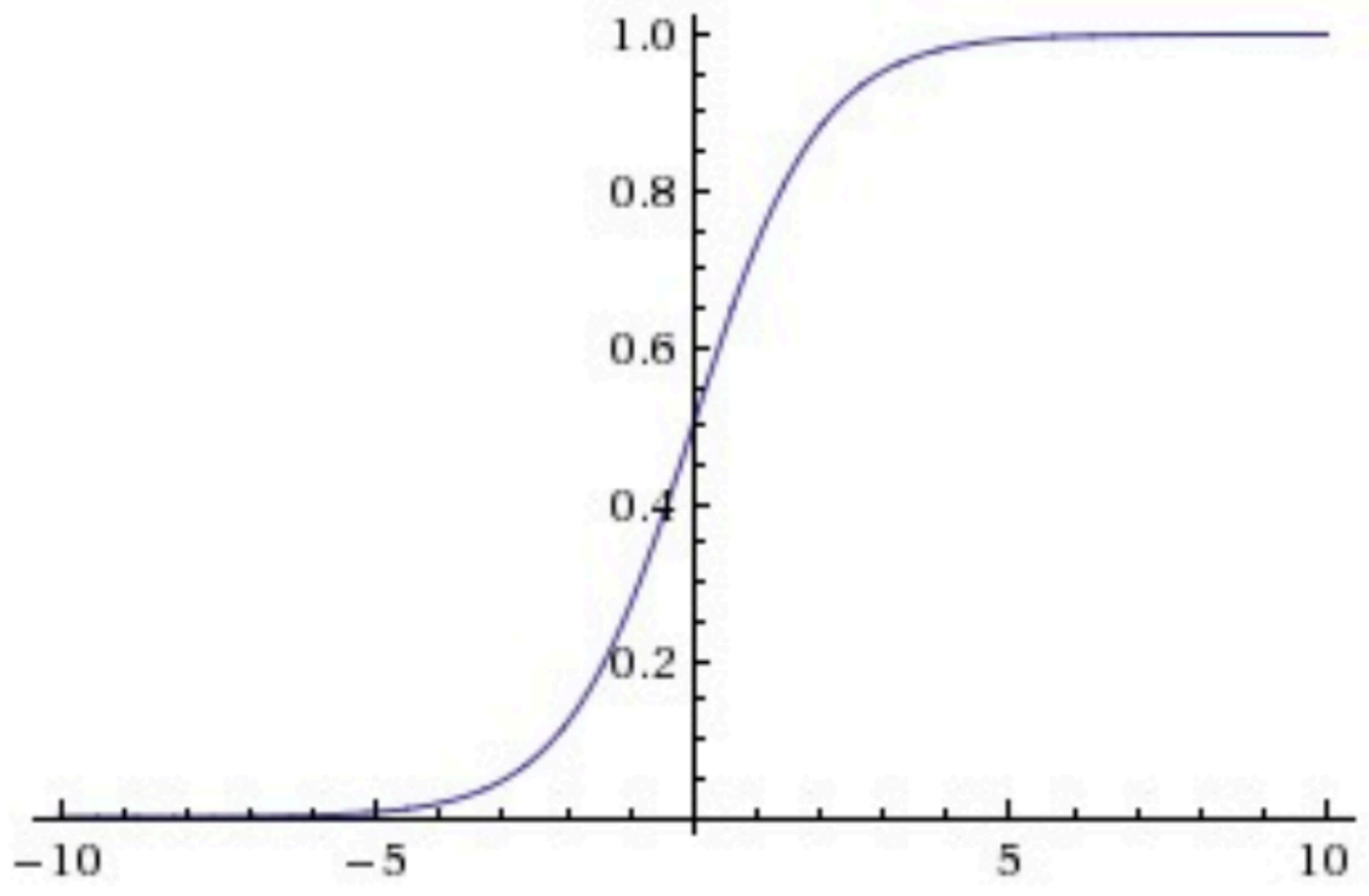


Figure credit: Fei-Fei and Karpathy

Common in many early neural networks

Biological analogy to saturated firing rate of neurons

Maps the input to the range [0,1]

Activation Function: **ReLU** (Rectified Linear Unit)

$$f(x) = \max(0, x)$$

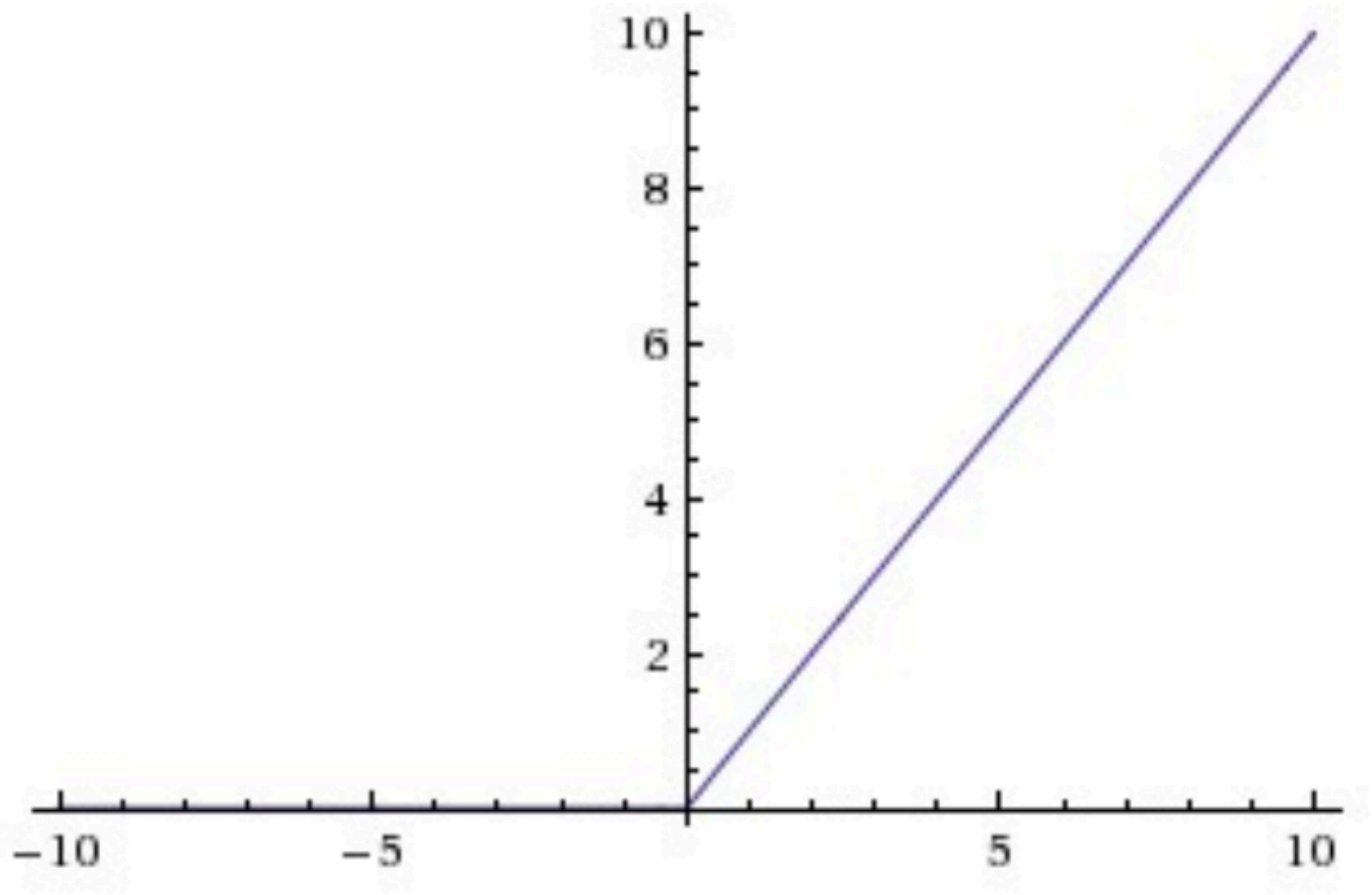
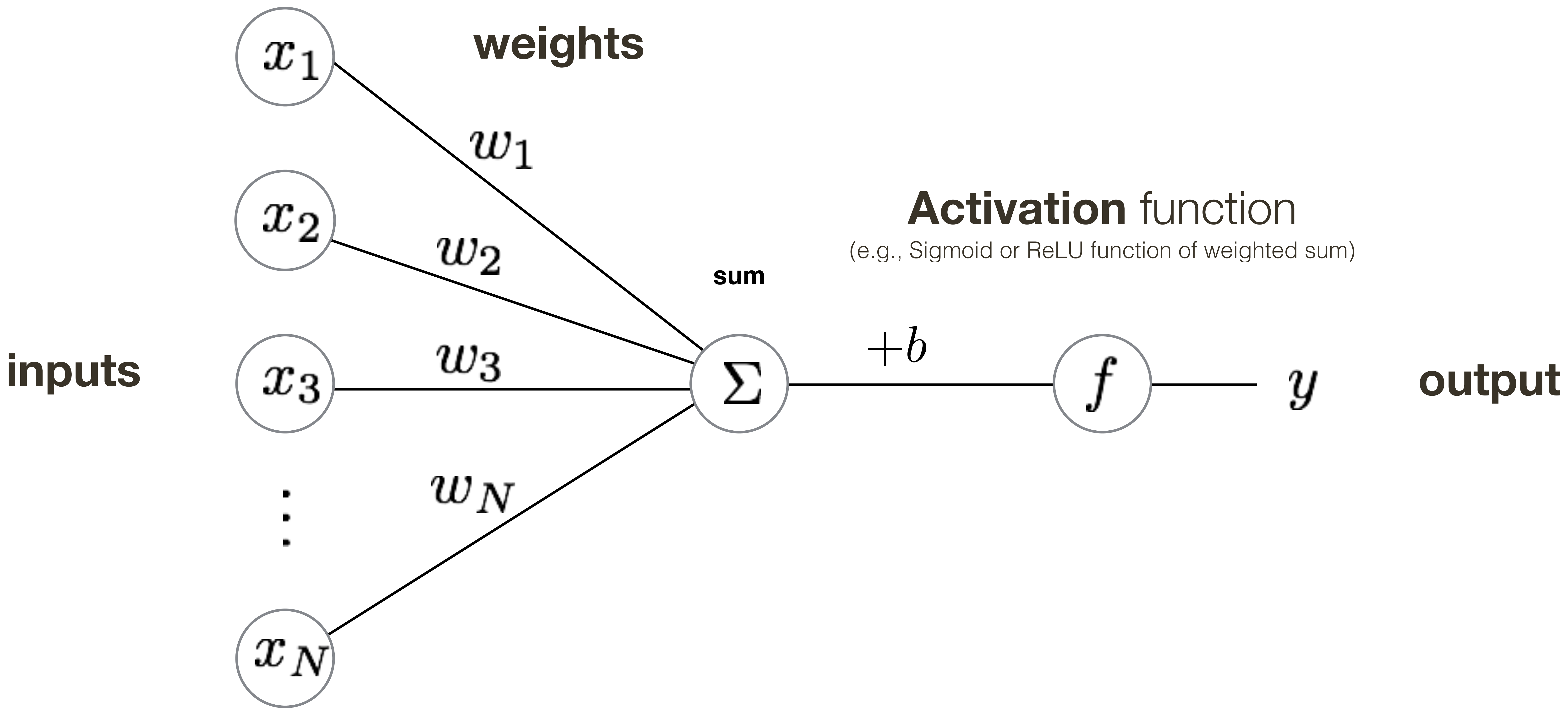


Figure credit: Fei-Fei and Karpathy

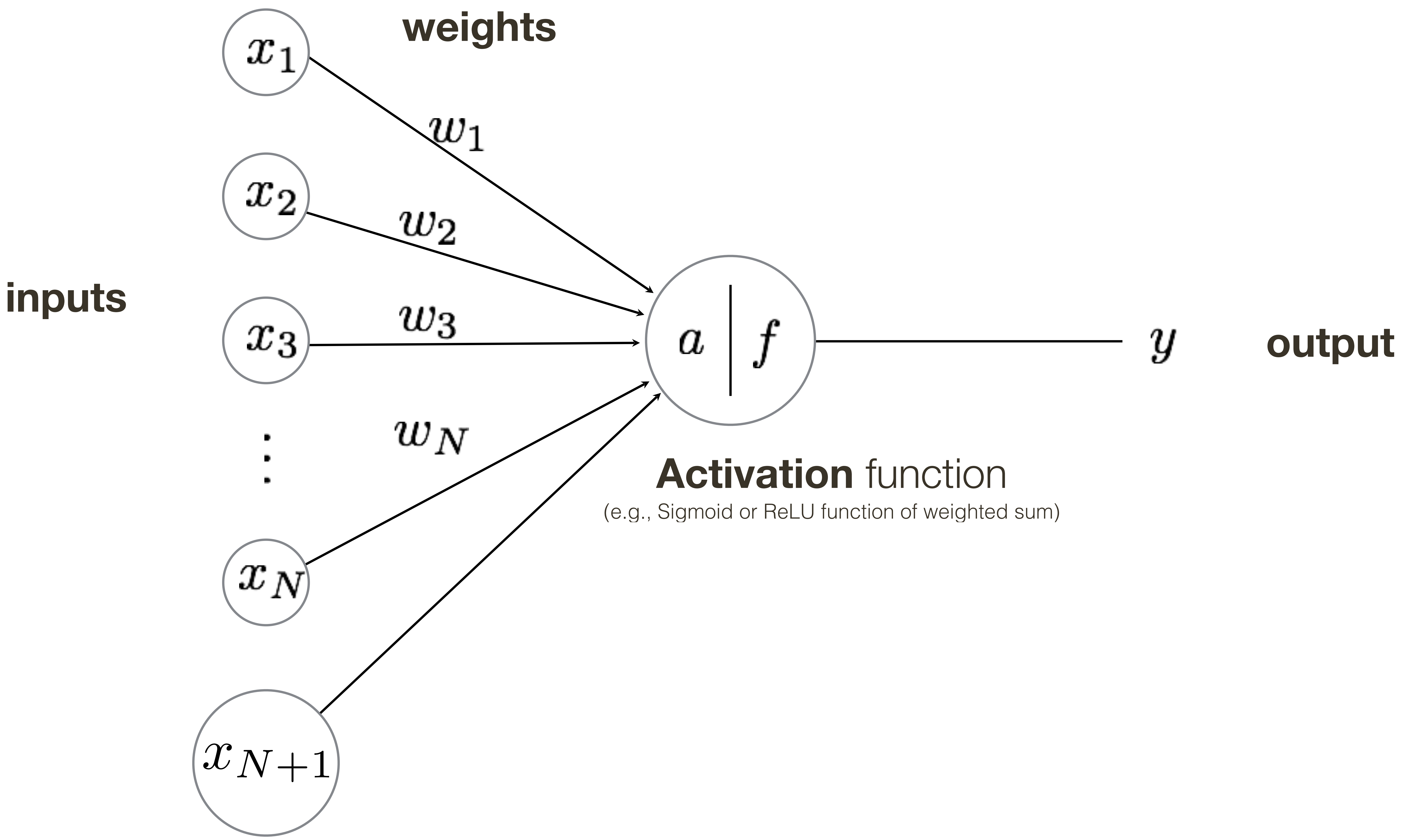
Found to accelerate convergence during learning

Used in the most recent neural networks

A **Neuron**


weights
w_1
w_2
w_3
w_N
inputs
x_1
x_2
x_3
x_N
sum
Σ
$+b$
Activation function
(e.g., Sigmoid or ReLU function of weighted sum)
f
y
output

A **Neuron** … another way to draw it …

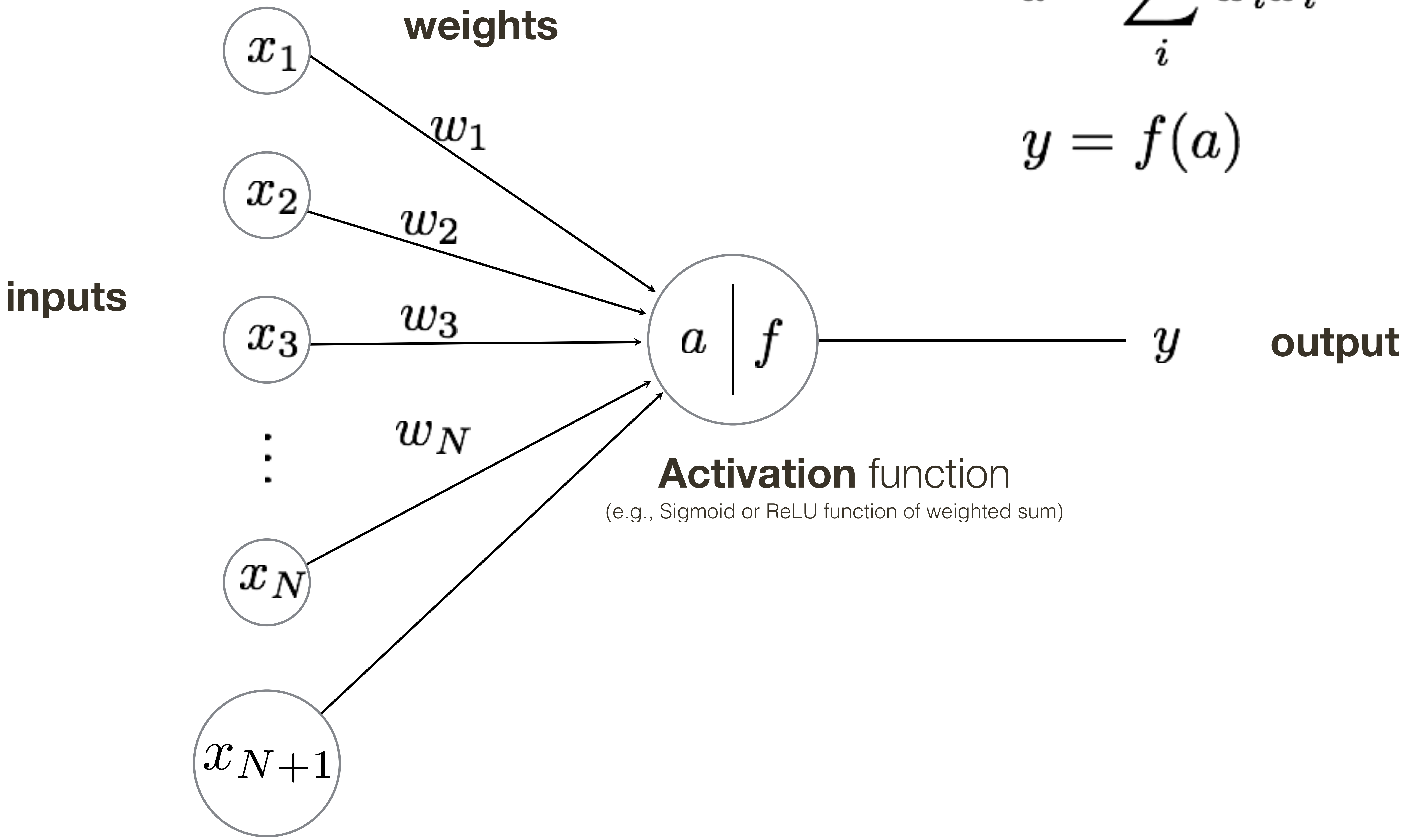


A **Neuron** … another way to draw it …

(1) Combine the sum and activation function

$$a = \sum_i w_i x_i$$

$$y = f(a)$$



Activation function

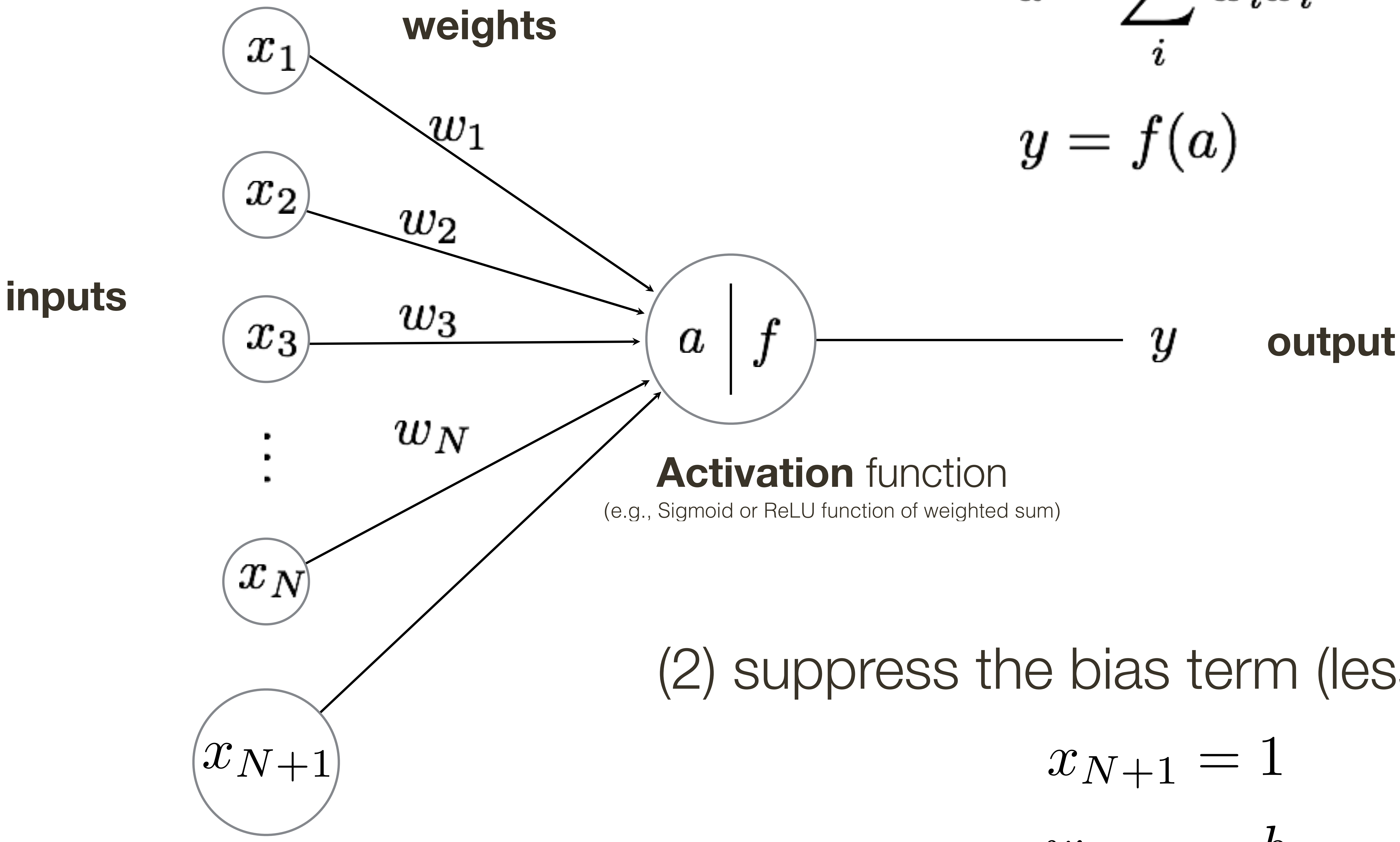
(e.g., Sigmoid or ReLU function of weighted sum)

A **Neuron** … another way to draw it …

(1) Combine the sum and activation function

$$a = \sum_i w_i x_i$$

$$y = f(a)$$



(2) suppress the bias term (less clutter)

$$x_{N+1} = 1$$

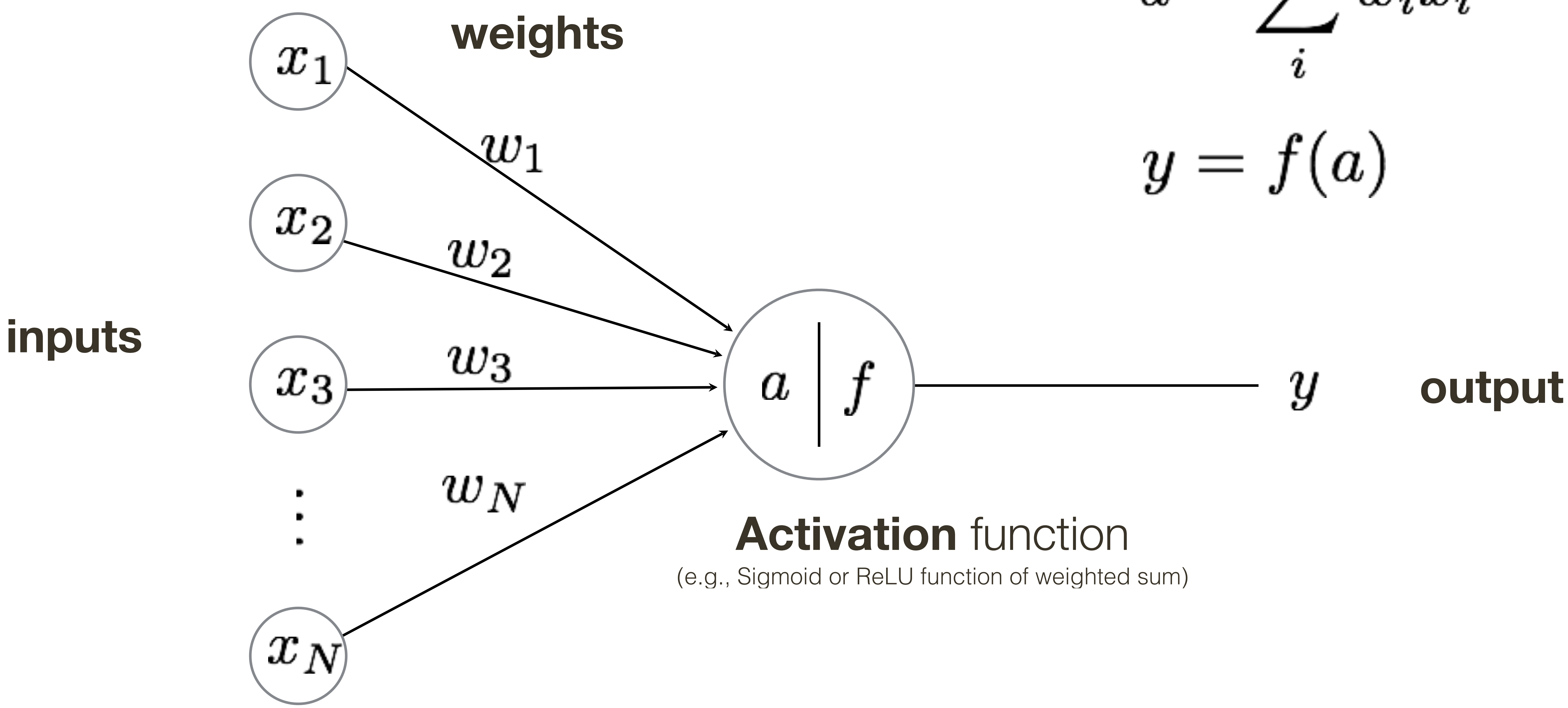
$$w_{N+1} = b$$

A **Neuron** … another way to draw it …

(1) Combine the sum and activation function

$$a = \sum_i w_i x_i$$

$$y = f(a)$$



Activation function

(e.g., Sigmoid or ReLU function of weighted sum)

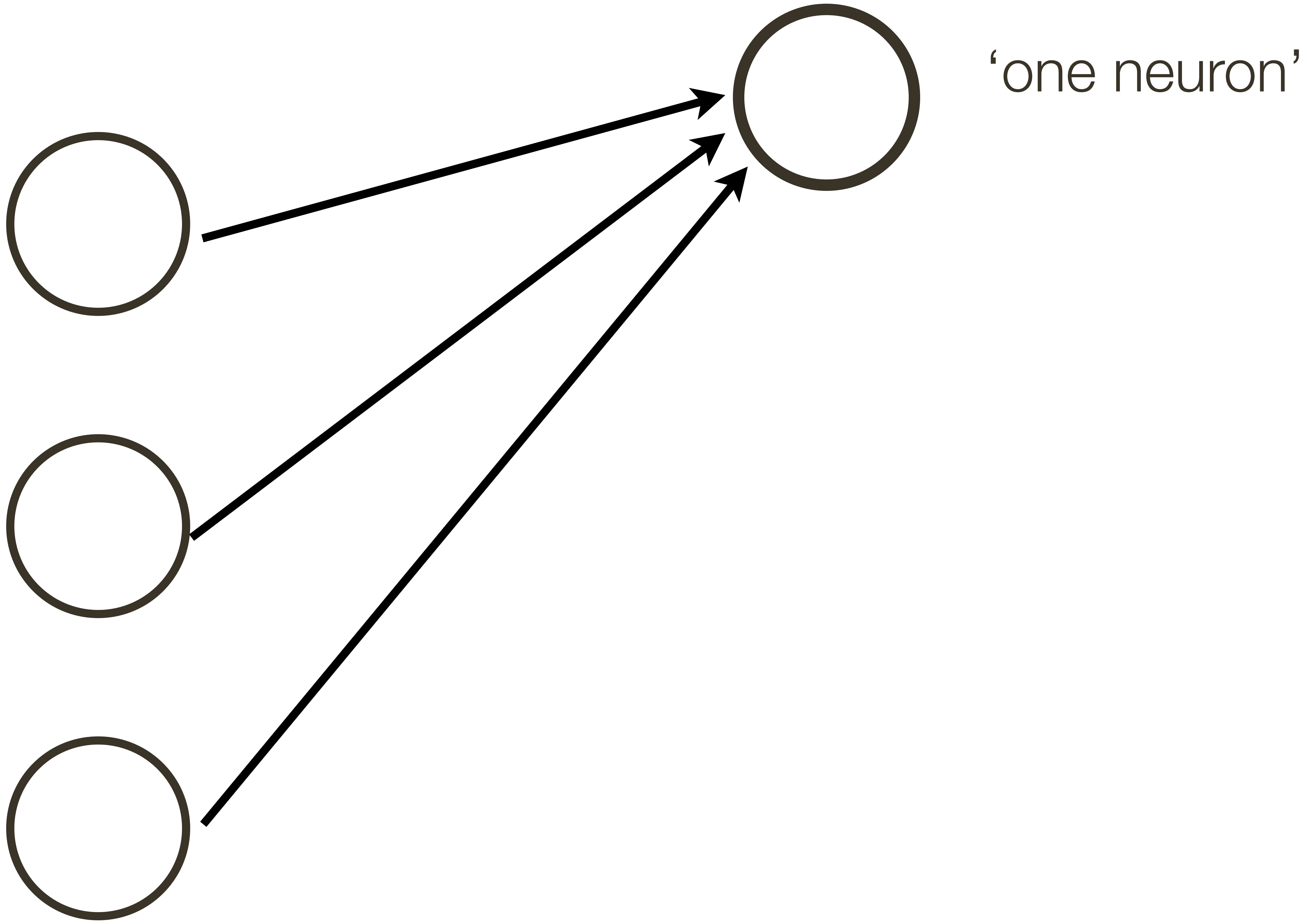
(2) suppress the bias term (less clutter)

$$x_{N+1} = 1$$

$$w_{N+1} = b$$

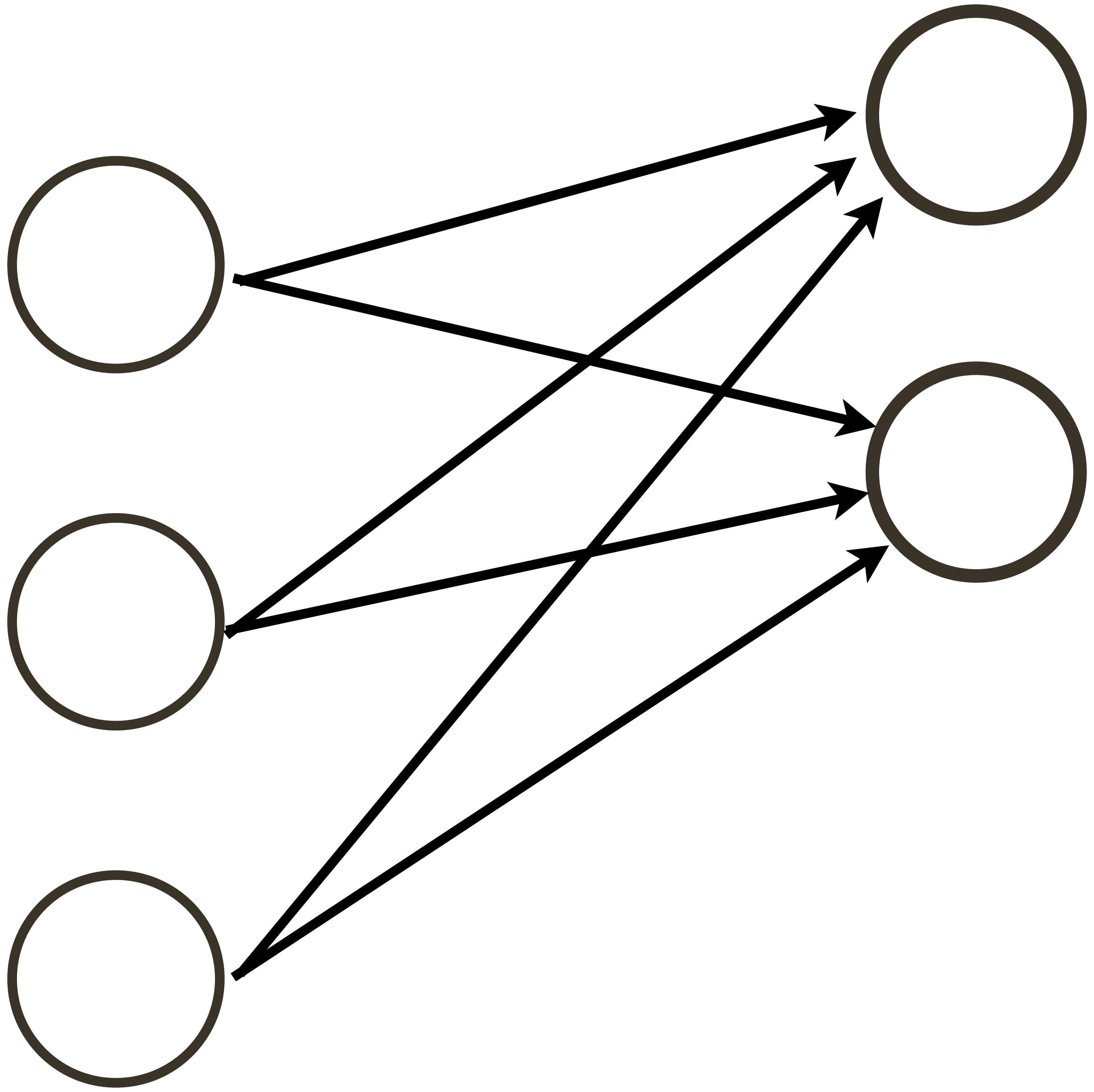
Neural Network

Connect a bunch of neurons together — a collection of connected neurons



Neural Network

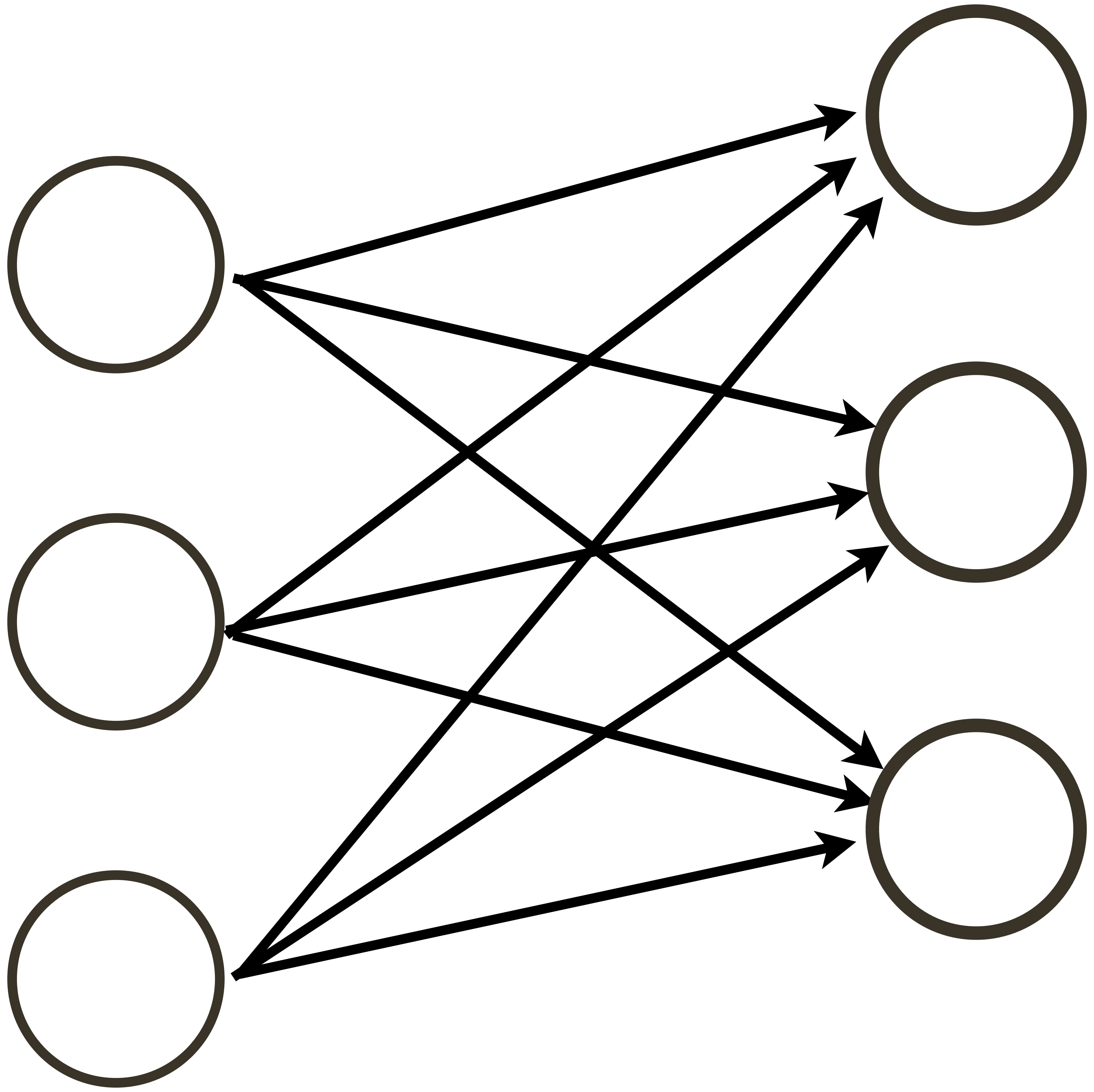
Connect a bunch of neurons together — a collection of connected neurons



‘two neurons’

Neural Network

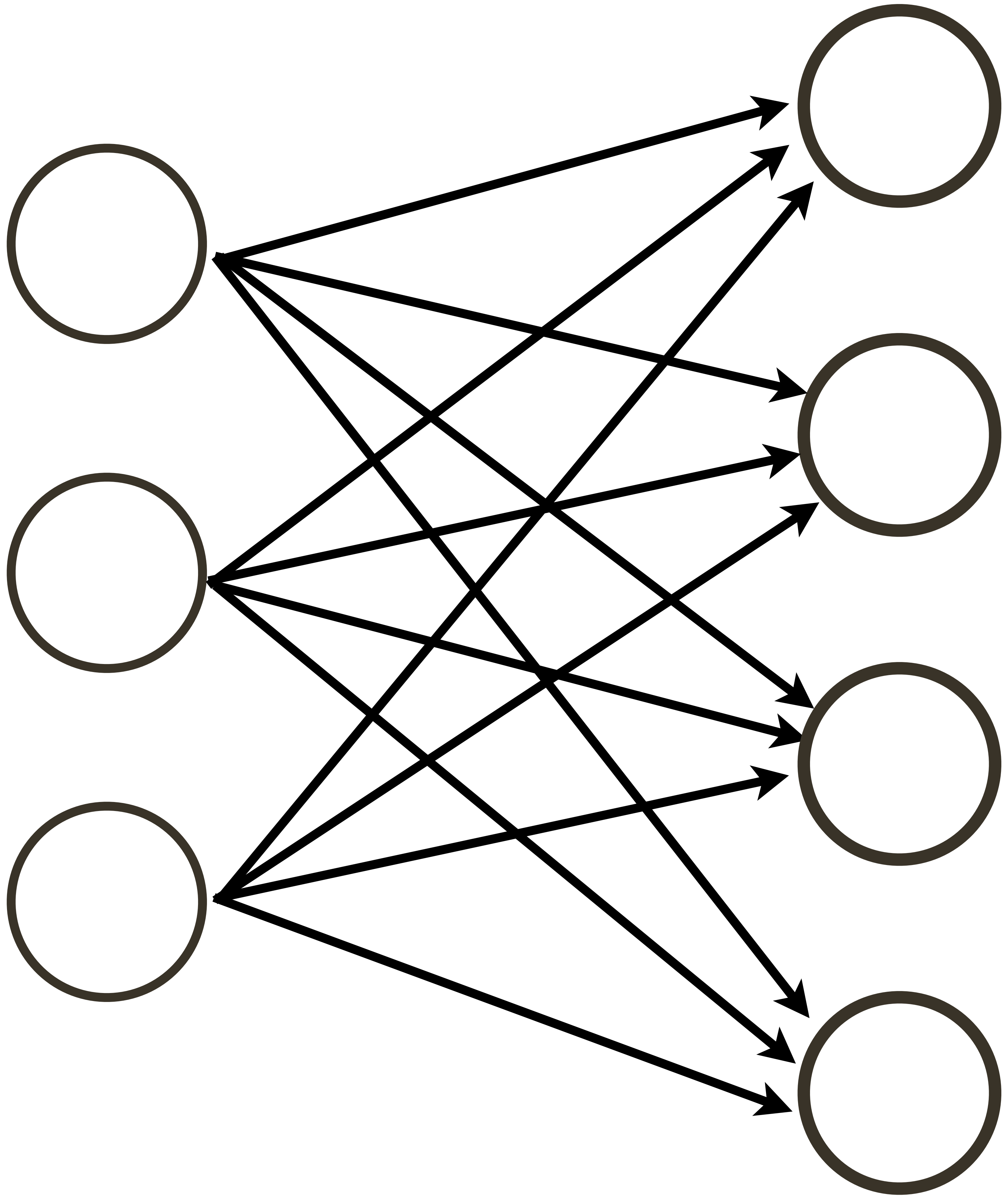
Connect a bunch of neurons together — a collection of connected neurons



‘three neurons’

Neural Network

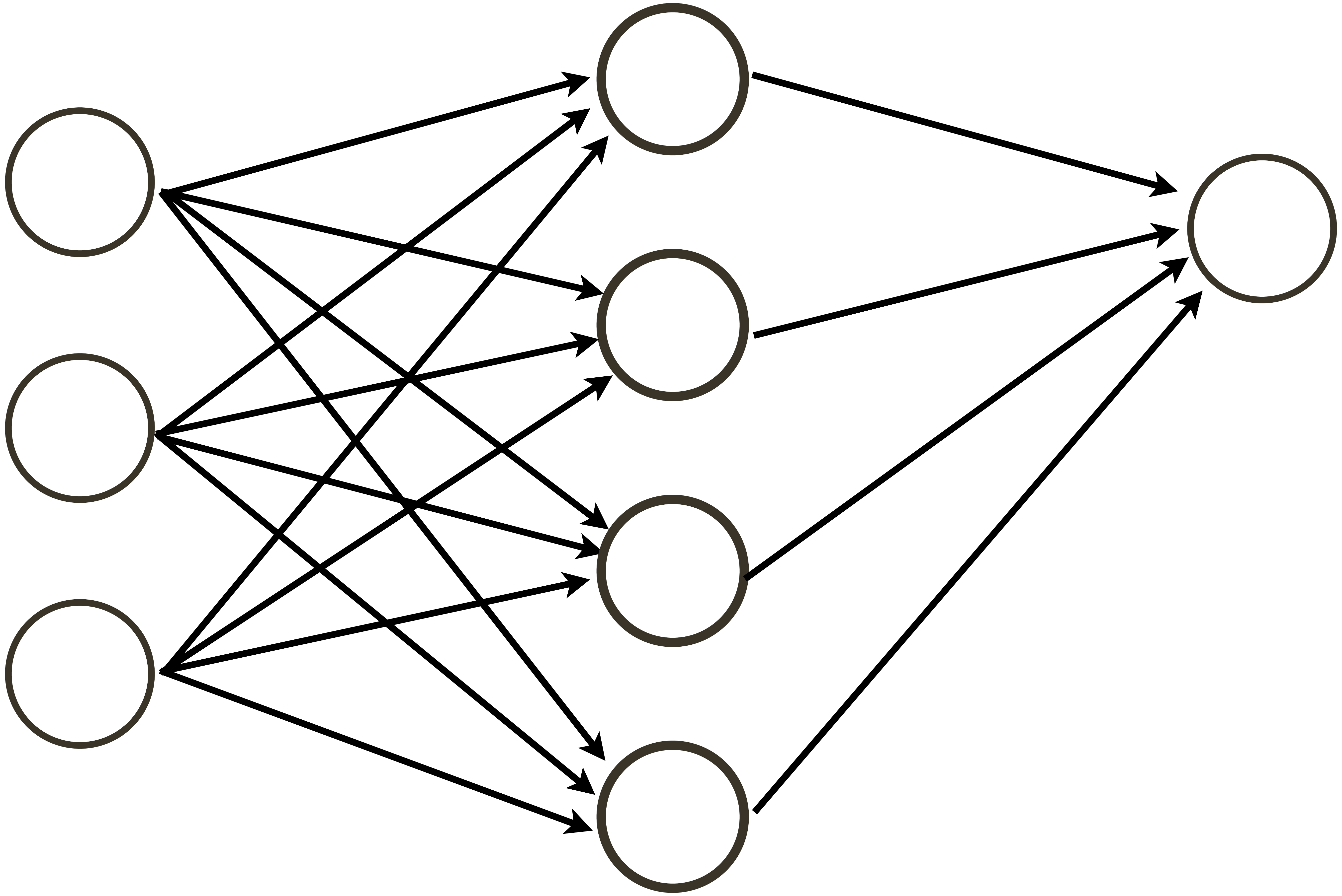
Connect a bunch of neurons together — a collection of connected neurons



‘four neurons’

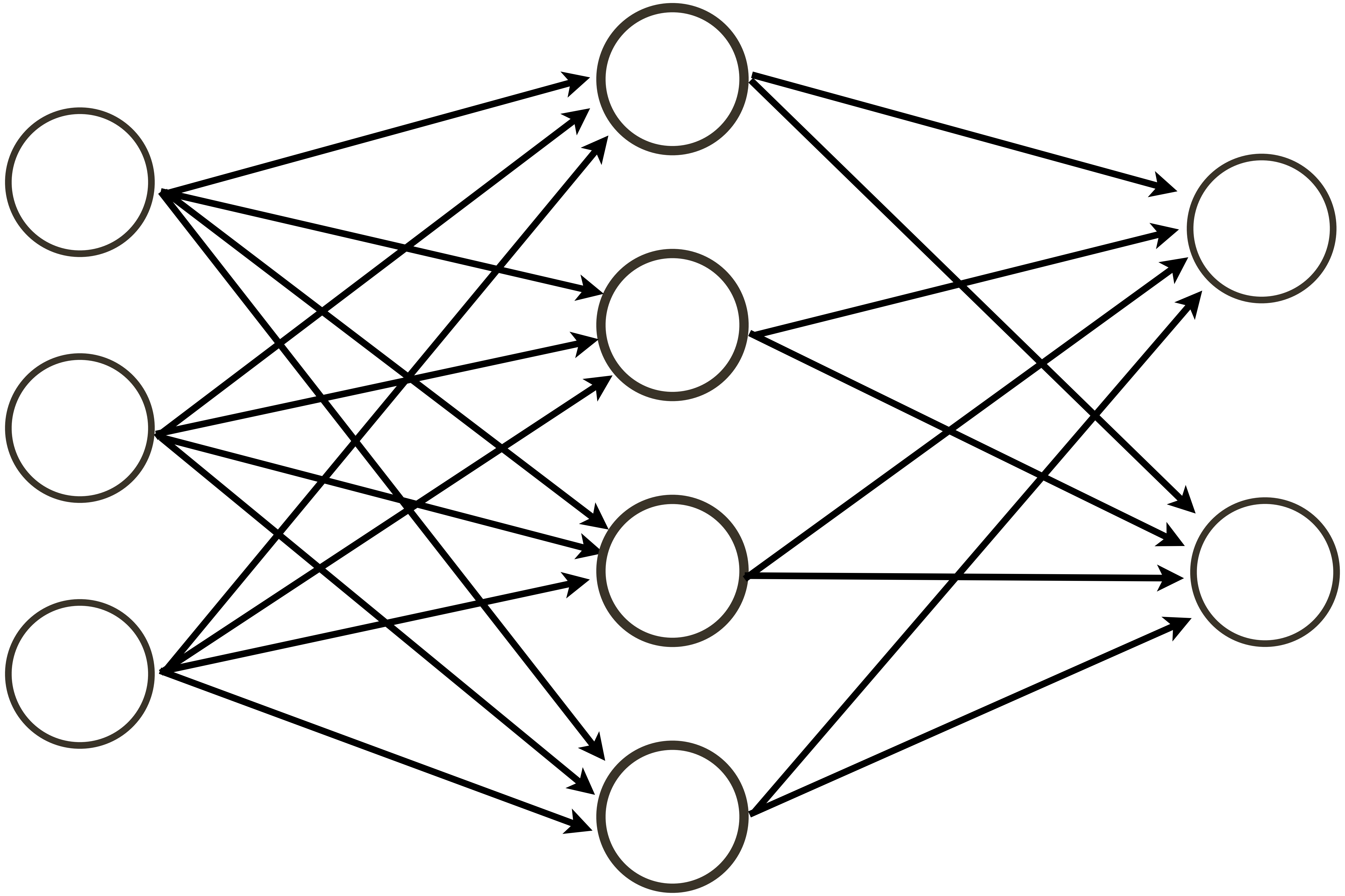
Neural Network

Connect a bunch of neurons together — a collection of connected neurons



Neural Network

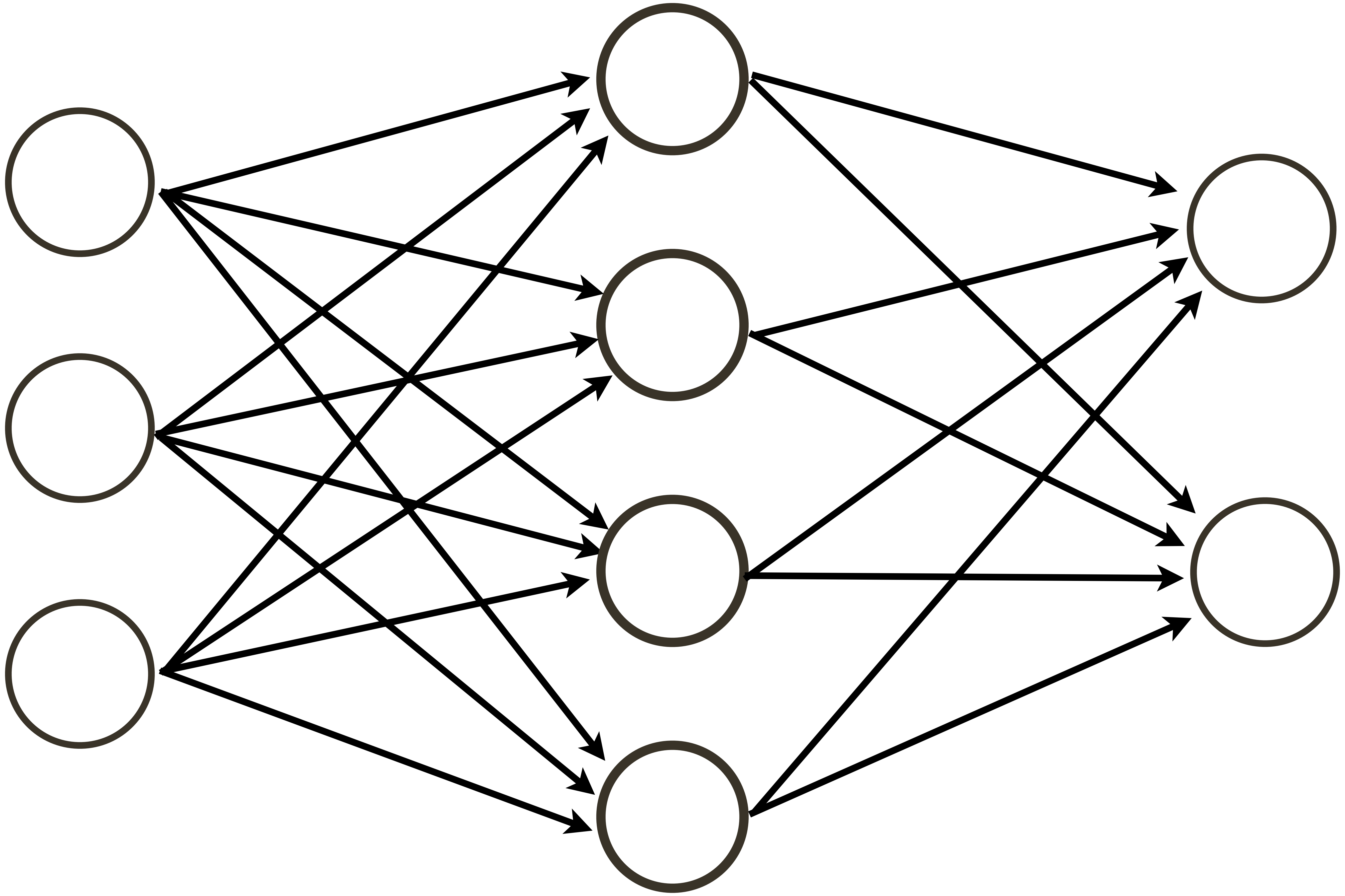
Connect a bunch of neurons together — a collection of connected neurons



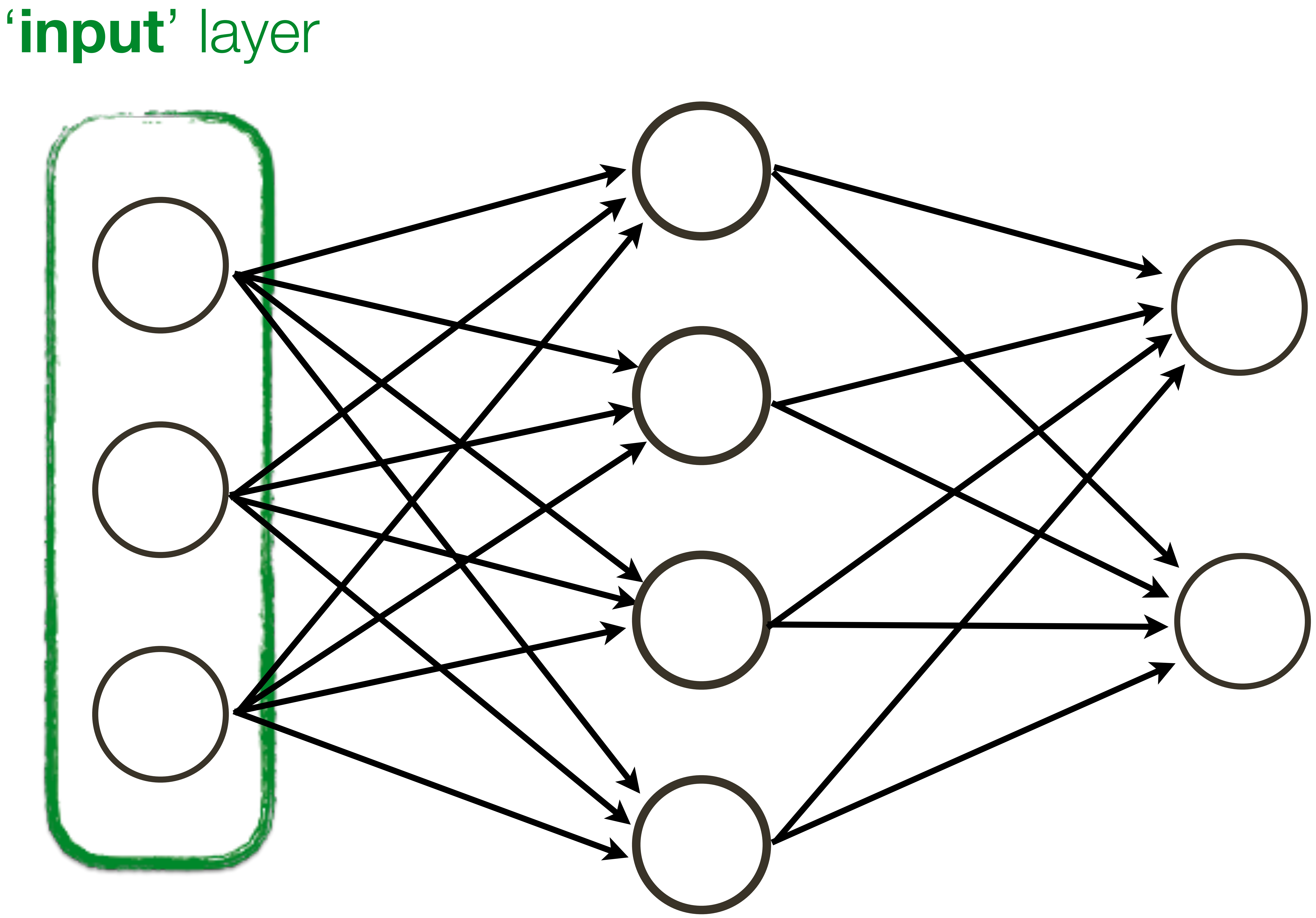
‘six neurons’

Neural Network

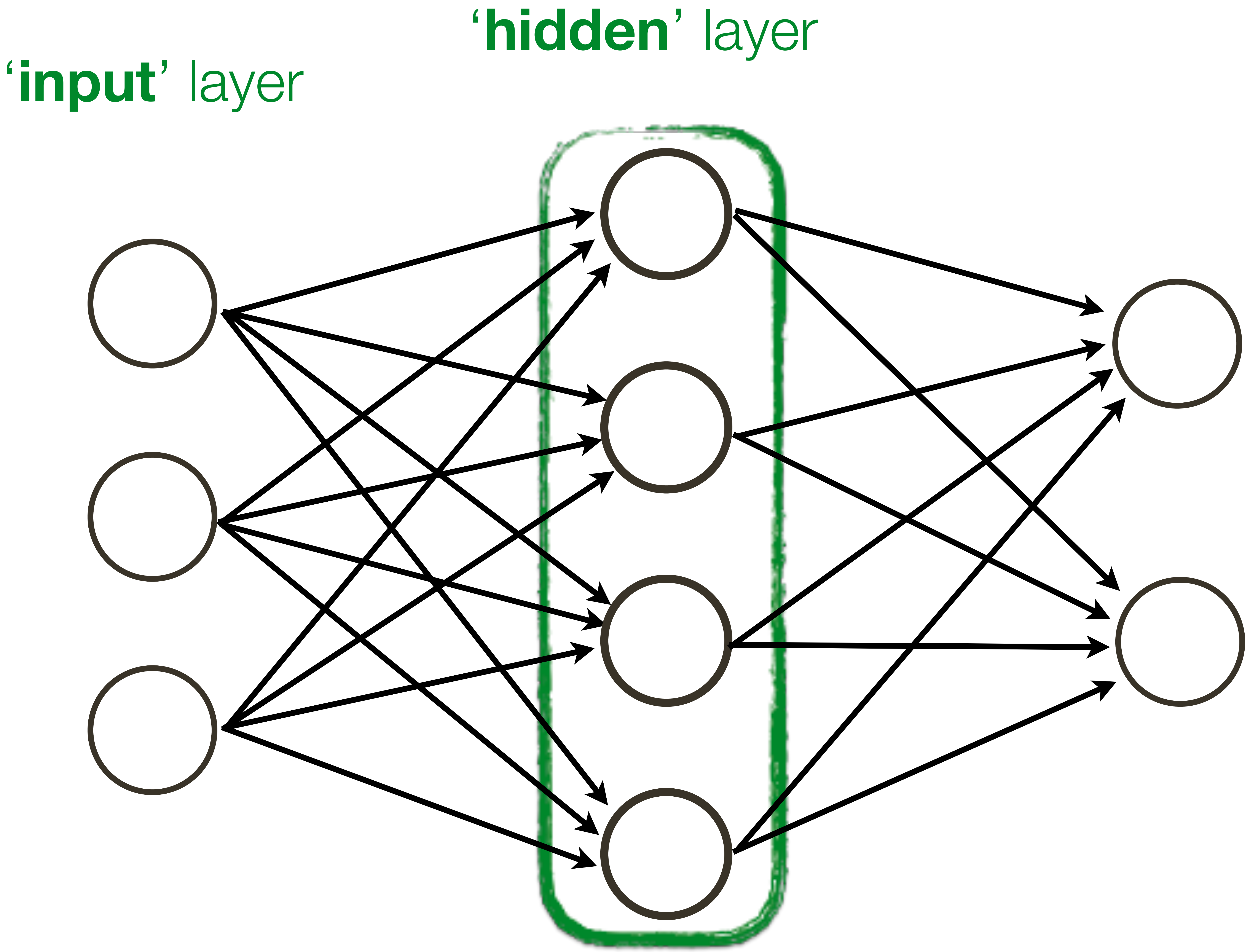
This network is also called a **Multi-layer Perceptron** (MLP)



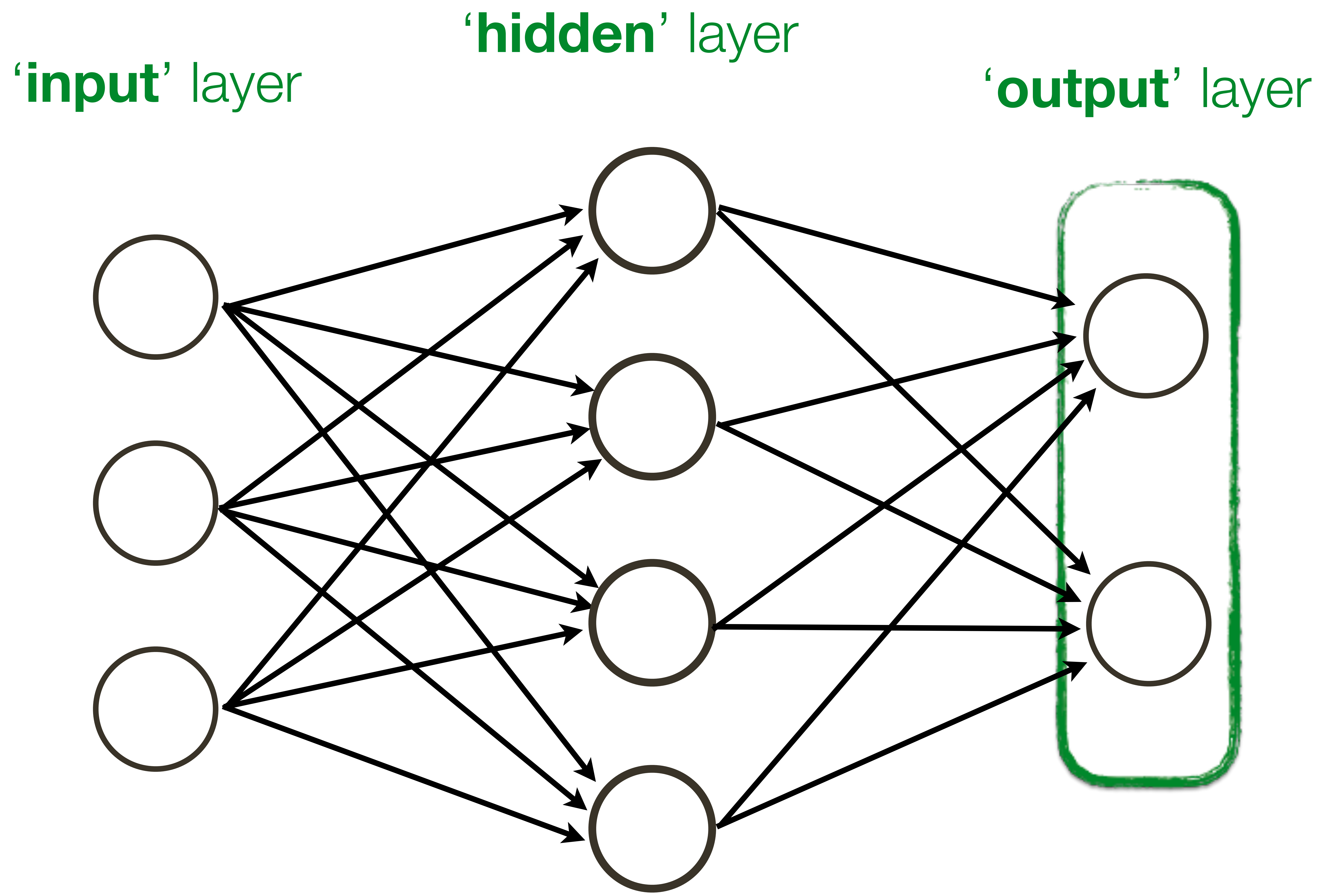
Neural Network: **Terminology**



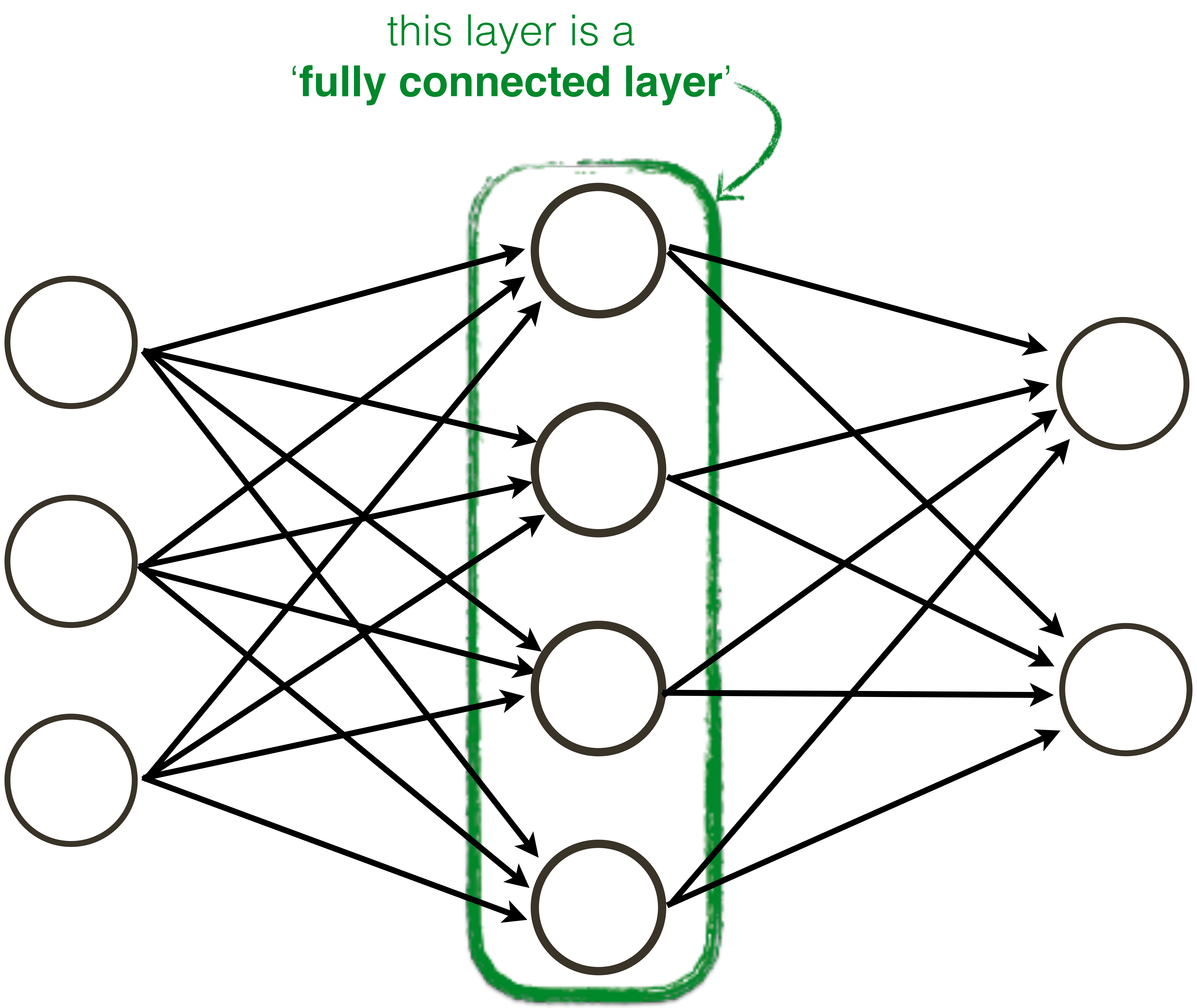
Neural Network: **Terminology**



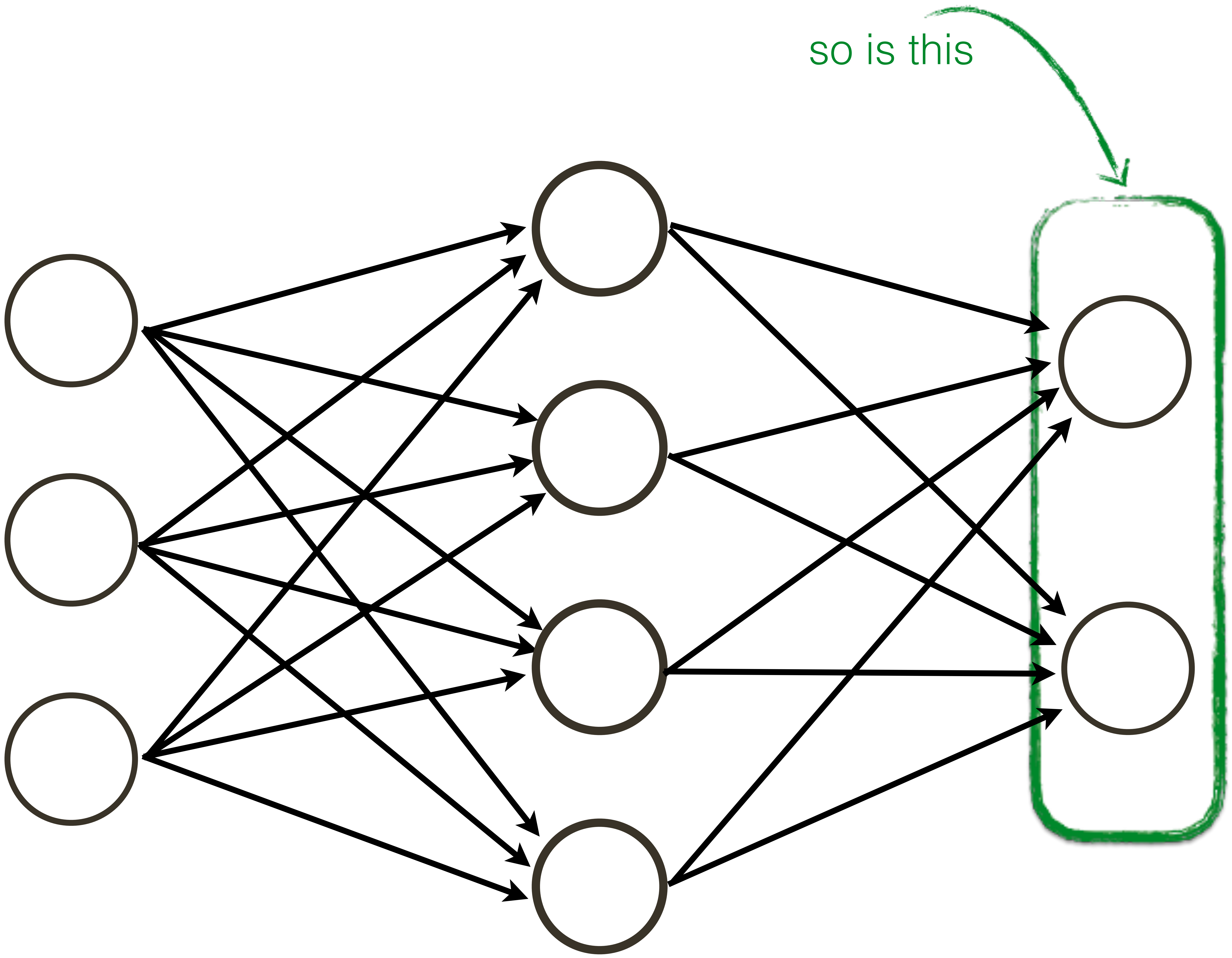
Neural Network: **Terminology**



Neural Network: **Terminology**



Neural Network: **Terminology**



Neural Network

A neural network comprises neurons connected in an acyclic graph

The outputs of neurons can become inputs to other neurons

Neural networks typically contain multiple layers of neurons

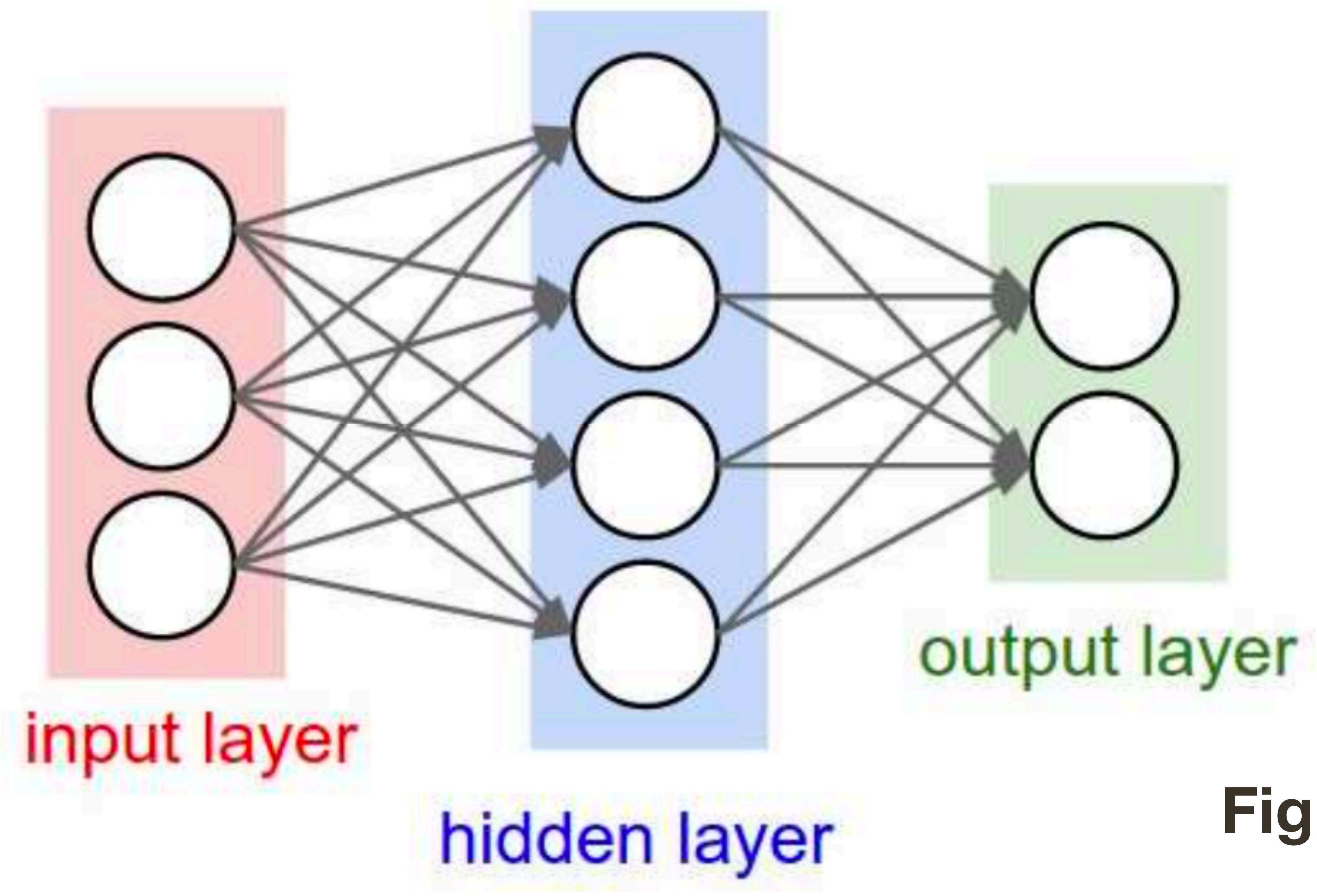


Figure credit: Fei-Fei and Karpathy

Example of a neural network with three inputs, a single hidden layer of four neurons, and an output layer of two neurons

Neural Network **Intuition**

Question: What is a Neural Network?

Answer: Complex mapping from an input (vector) to an output (vector)

* slide from Marc'Aurelio Renzato

Neural Network **Intuition**

Question: What is a Neural Network?

Answer: Complex mapping from an input (vector) to an output (vector)

Question: What class of functions should be considered for this mapping?

Answer: Compositions of simpler functions (a.k.a. layers)? We will talk more about what specific functions next …

* slide from Marc'Aurelio Renzato

Neural Network **Intuition**

Question: What is a Neural Network?

Answer: Complex mapping from an input (vector) to an output (vector)

Question: What class of functions should be considered for this mapping?

Answer: Compositions of simpler functions (a.k.a. layers)? We will talk more about what specific functions next …

Question: What does a hidden unit do?

Answer: It can be thought of as classifier or a feature.

* slide from Marc'Aurelio Renzato

Neural Network **Intuition**

Question: What is a Neural Network?

Answer: Complex mapping from an input (vector) to an output (vector)

Question: What class of functions should be considered for this mapping?

Answer: Compositions of simpler functions (a.k.a. layers)? We will talk more about what specific functions next …

Question: What does a hidden unit do?

Answer: It can be thought of as classifier or a feature.

Question: Why have many layers?

Answer: 1) More layers = more complex functional mapping

2) More efficient due to distributed representation

* slide from Marc'Aurelio Renzato

Neural Network

A neural network comprises neurons connected in an acyclic graph

The outputs of neurons can become inputs to other neurons

Neural networks typically contain multiple layers of neurons

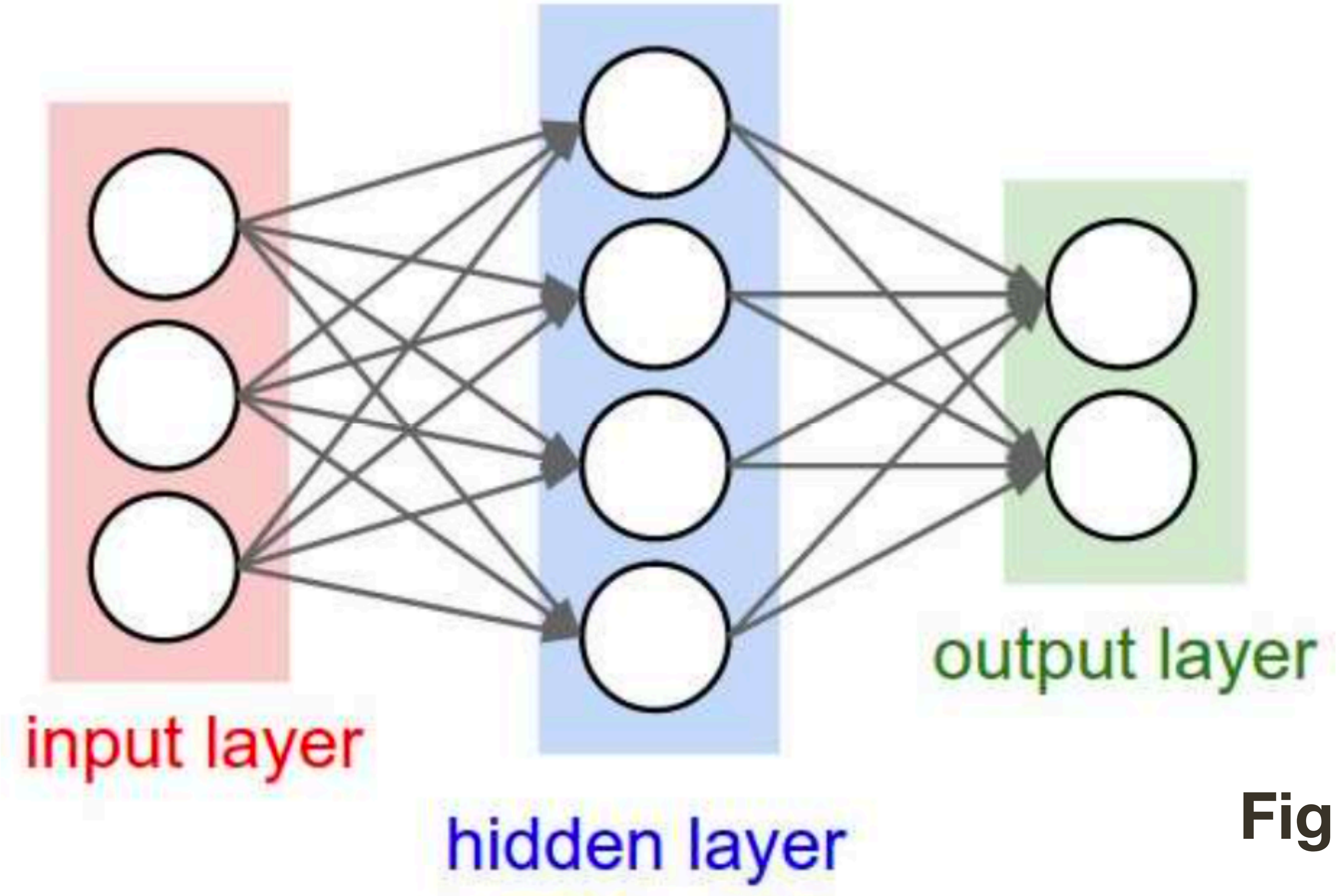


Figure credit: Fei-Fei and Karpathy

Example of a neural network with three inputs, a single hidden layer of four neurons, and an output layer of two neurons

Neural Network

Note: each neuron will have its own vector of weights and a bias, its easier to think of all neurons in a layer as a single entity with a matrix of weights (size = number of inputs x number of neurons) and a vector of biases (size = number of neurons)

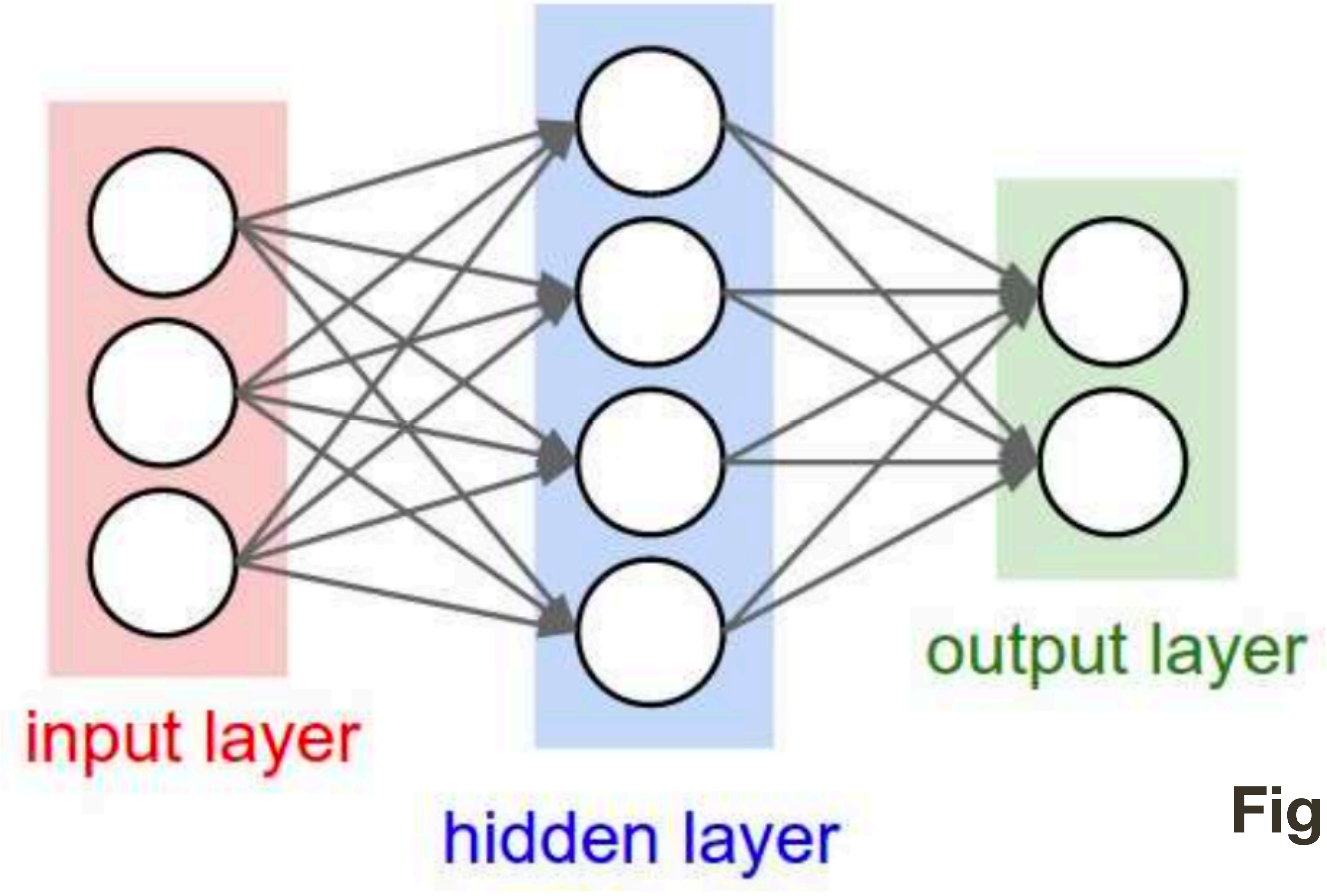


Figure credit: Fei-Fei and Karpathy

Neural Network

Note: each neuron will have its own vector of weights and a bias, its easier to think of all neurons in a layer as a single entity with a matrix of weights (size = number of inputs x number of neurons) and a vector of biases (size = number of neurons)

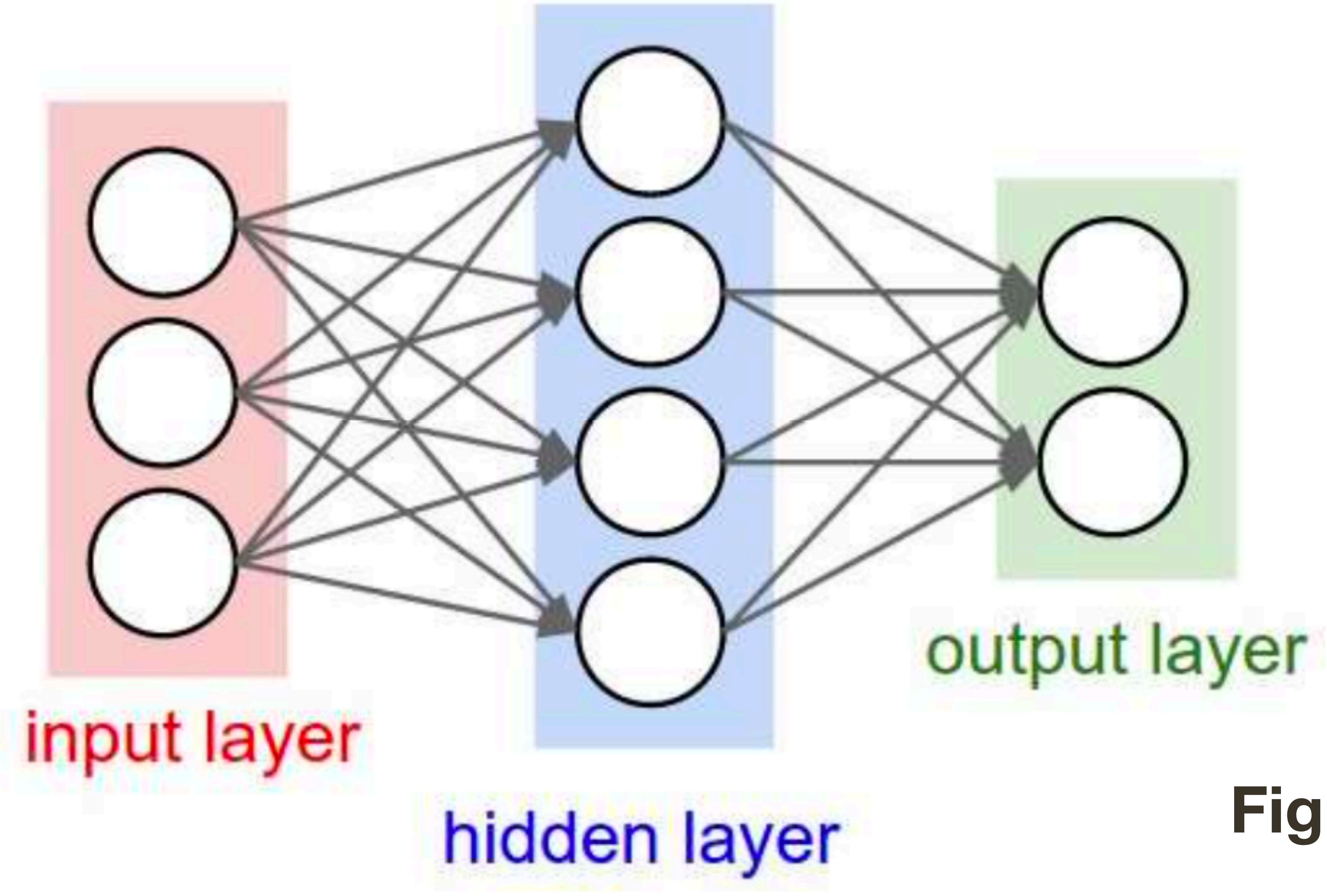
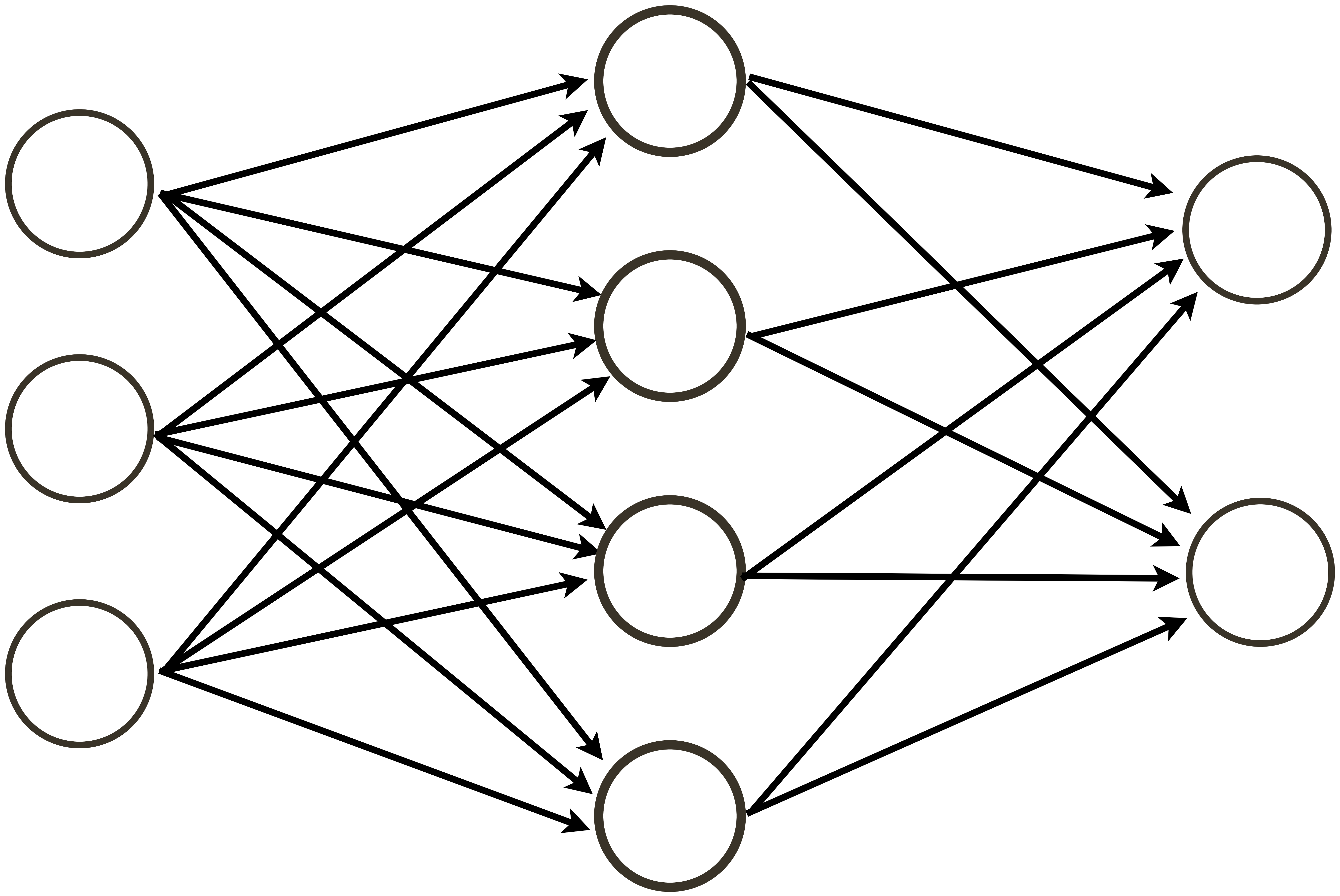


Figure credit: Fei-Fei and Karpathy

$$\hat{\mathbf{y}} = f(\mathbf{x}, \mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2) = \sigma\left(\mathbf{W}_2^{(2\times4)} \sigma\left(\mathbf{W}_1^{(4\times3)} \mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)}\right)$$

Activation Function

Why can't we have **linear** activation functions? Why have non-linear activations?



Activation Function

$$\hat{\mathbf{y}} = f(\mathbf{x}, \mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2) = \sigma\left(\mathbf{W}_2^{(2\times4)}\sigma\left(\mathbf{W}_1^{(4\times3)}\mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)}\right)$$

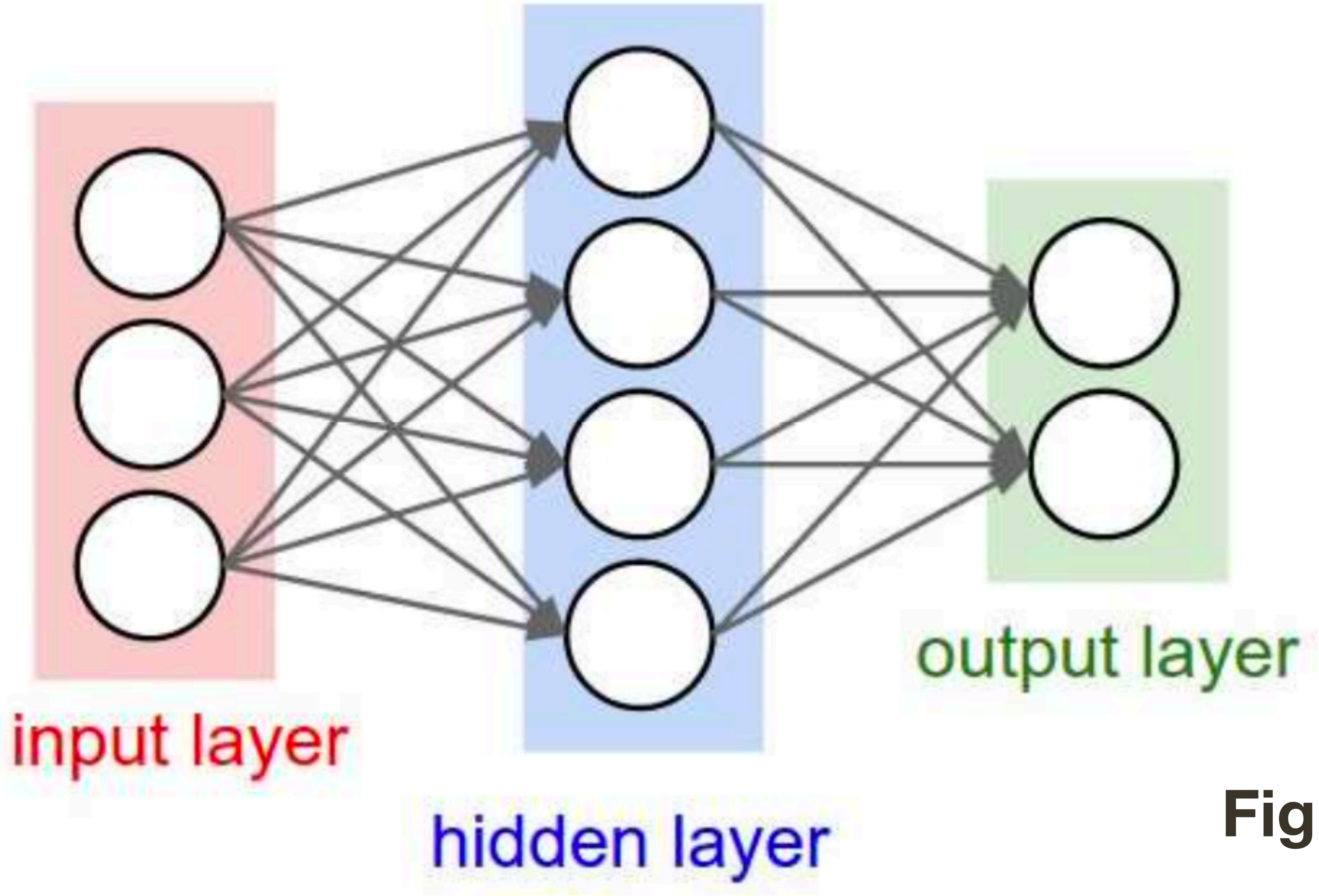


Figure credit: Fei-Fei and Karpathy

Activation Function

$$\hat{\mathbf{y}} = f(\mathbf{x}, \mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2) = \sigma\left(\mathbf{W}_2^{(2\times4)}\sigma\left(\mathbf{W}_1^{(4\times3)}\mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)}\right)$$
$$= \mathbf{W}_2^{(2\times4)}\left(\mathbf{W}_1^{(4\times3)}\mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)}$$

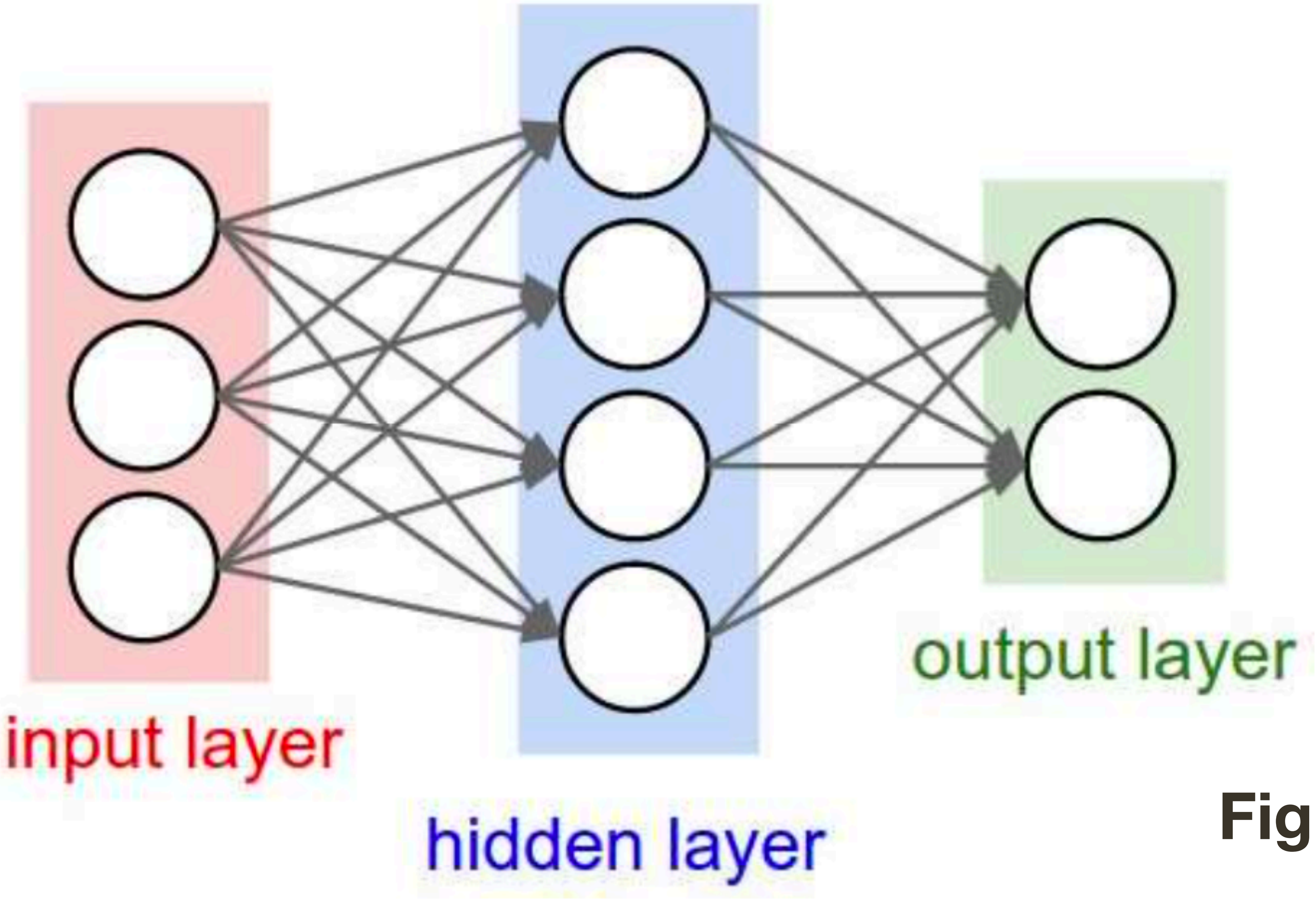


Figure credit: Fei-Fei and Karpathy

Activation Function

$$\hat{\mathbf{y}} = f(\mathbf{x}, \mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2) = \sigma\left(\mathbf{W}_2^{(2\times4)}\sigma\left(\mathbf{W}_1^{(4\times3)}\mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)}\right)$$
$$= \mathbf{W}_2^{(2\times4)}\left(\mathbf{W}_1^{(4\times3)}\mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)}$$
$$= \mathbf{W}_2^{(2\times4)}\mathbf{W}_1^{(4\times3)}\mathbf{x} + \mathbf{W}_2^{(2\times4)}\mathbf{b}_1^{(4)} + \mathbf{b}_2^{(2)}$$

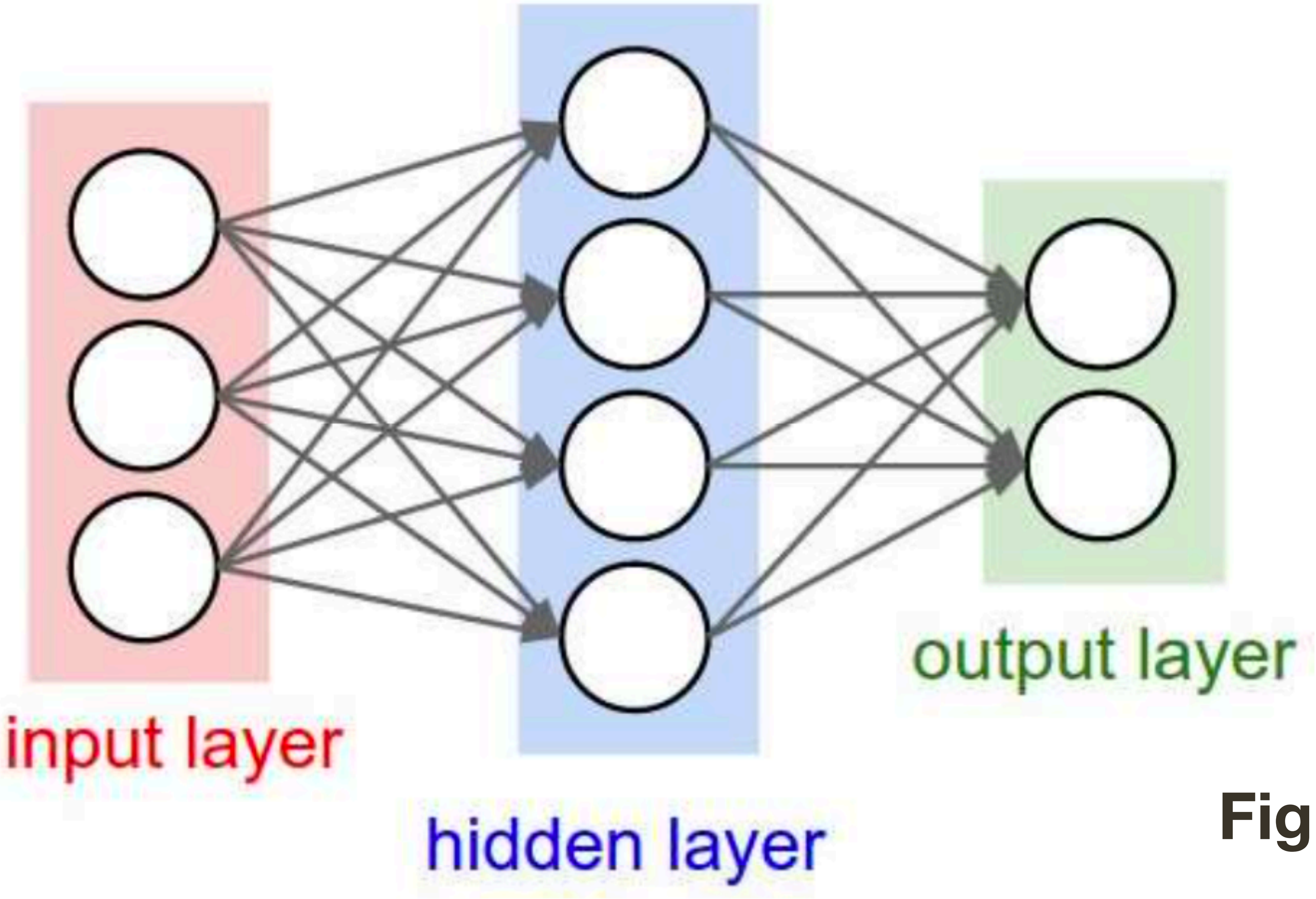


Figure credit: Fei-Fei and Karpathy

Activation Function

$$\hat{\mathbf{y}} = f(\mathbf{x}, \mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2) = \sigma\left(\mathbf{W}_2^{(2\times4)}\sigma\left(\mathbf{W}_1^{(4\times3)}\mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)}\right)$$

$$= \mathbf{W}_2^{(2\times4)}\left(\mathbf{W}_1^{(4\times3)}\mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)}$$

$$= \underbrace{\mathbf{W}_2^{(2\times4)}\mathbf{W}_1^{(4\times3)}}_{\mathbf{W}_*^{(2\times3)}}\mathbf{x} + \underbrace{\mathbf{W}_2^{(2\times4)}\mathbf{b}_1^{(4)} + \mathbf{b}_2^{(2)}}_{\mathbf{b}^{(2)}}$$

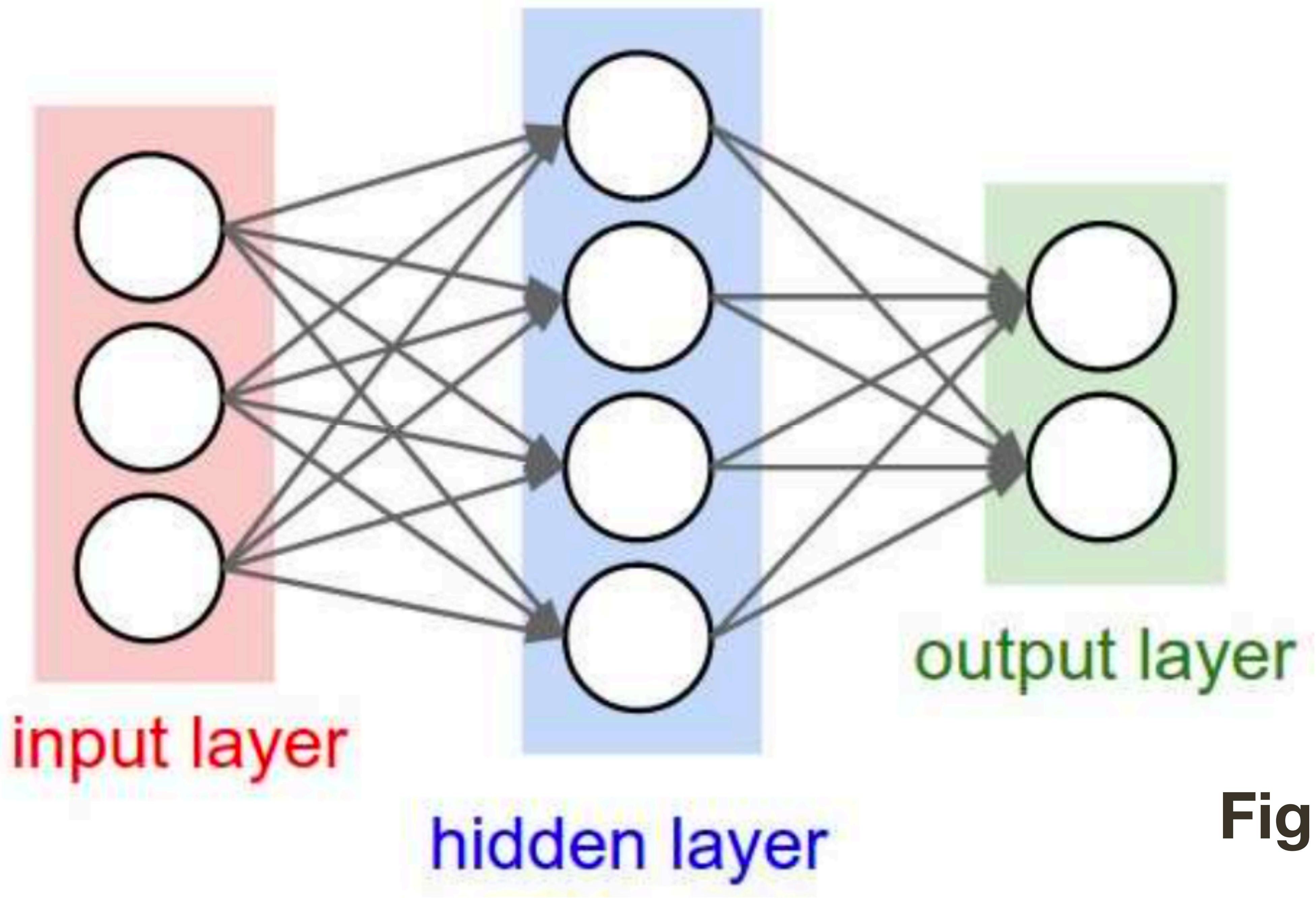


Figure credit: Fei-Fei and Karpathy

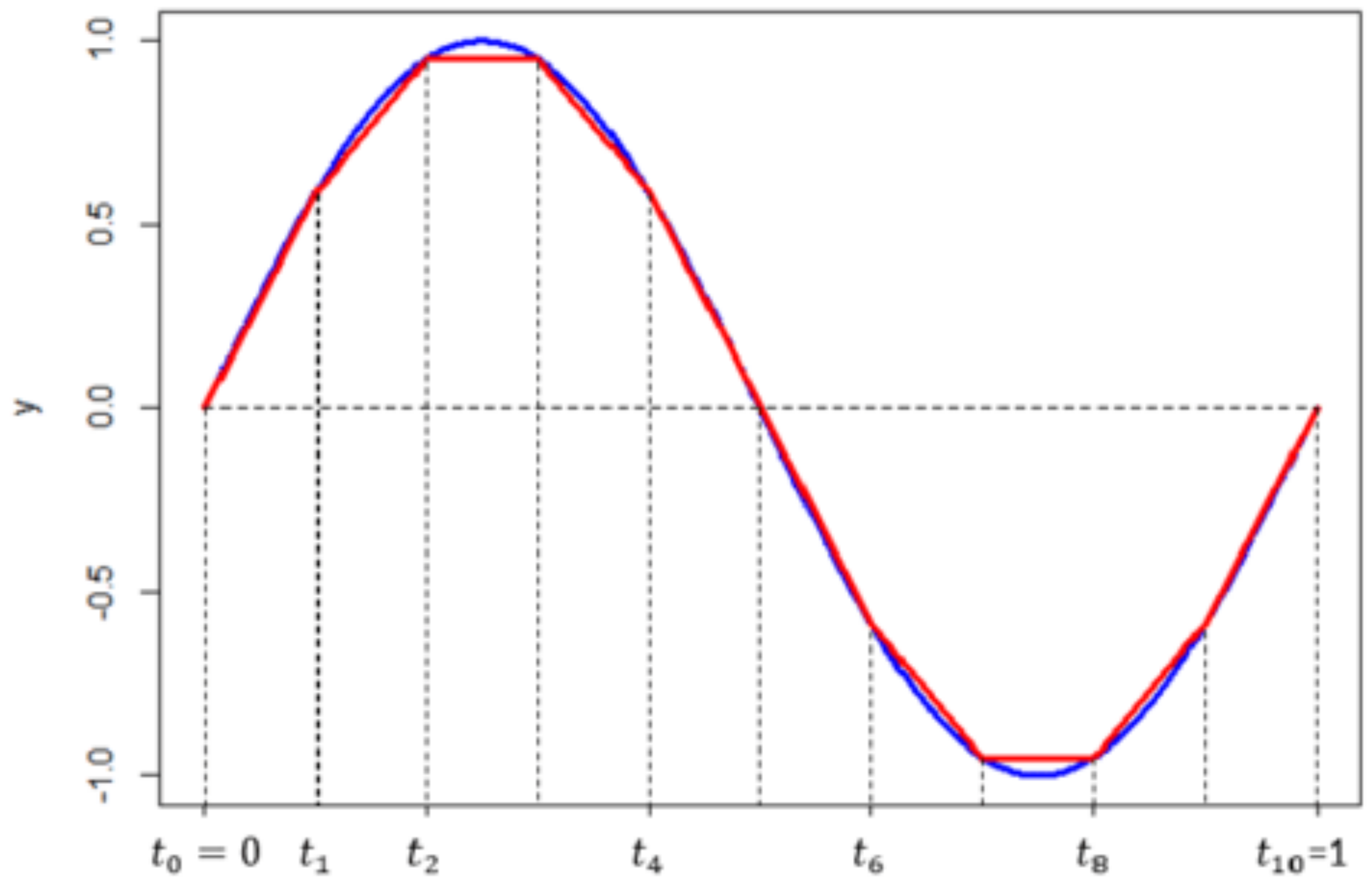
Activation Function

Non-linear activation is required to provably make the Neural Net a **universal function approximator**

Intuition: with ReLU activation, we effectively get a linear spline approximation to any function.

Optimization of neural net parameters = finding slops and transitions of linear pieces

The quality of approximation depends on the number of linear segments



Number of linear segments for large input dimension: $\Omega(2^{\frac{2}{3}Ln})$

Light Theory: Neural Network as Universal Approximator

Universal Approximation Theorem: Single hidden layer can approximate any continuous function with compact support to arbitrary accuracy, when the width goes to infinity.

[Hornik *et al*., 1989]

Universal Approximation Theorem (revised): A network of infinite depth with a hidden layer of size $d+1$ neurons, where d is the dimension of the input space, can approximate any continuous function.

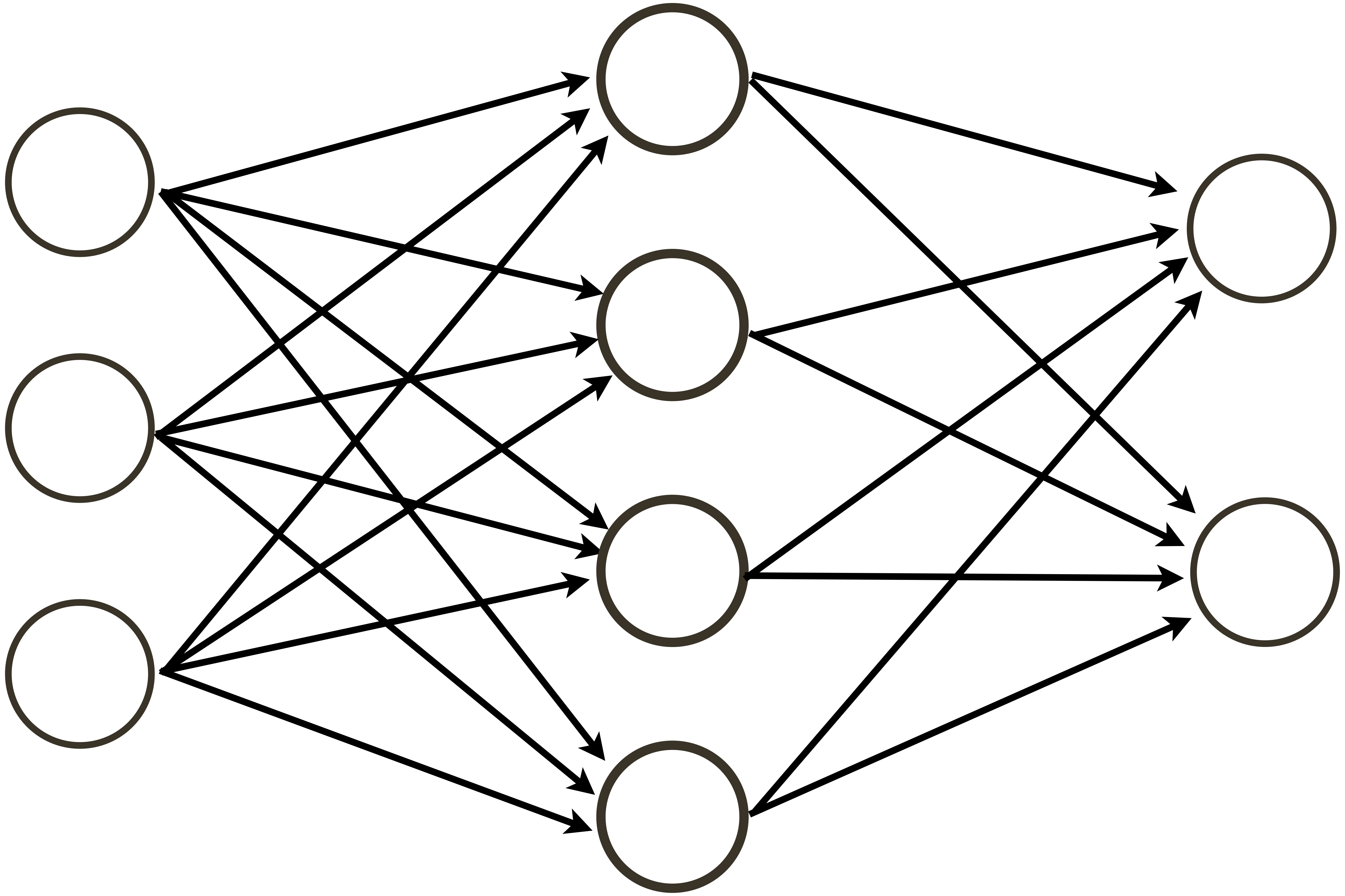
[Lu *et al*., NIPS 2017]

Universal Approximation Theorem (further revised): ResNet with a single hidden unit and infinite depth can approximate any continuous function.

[Lin and Jegelka, NIPS 2018]

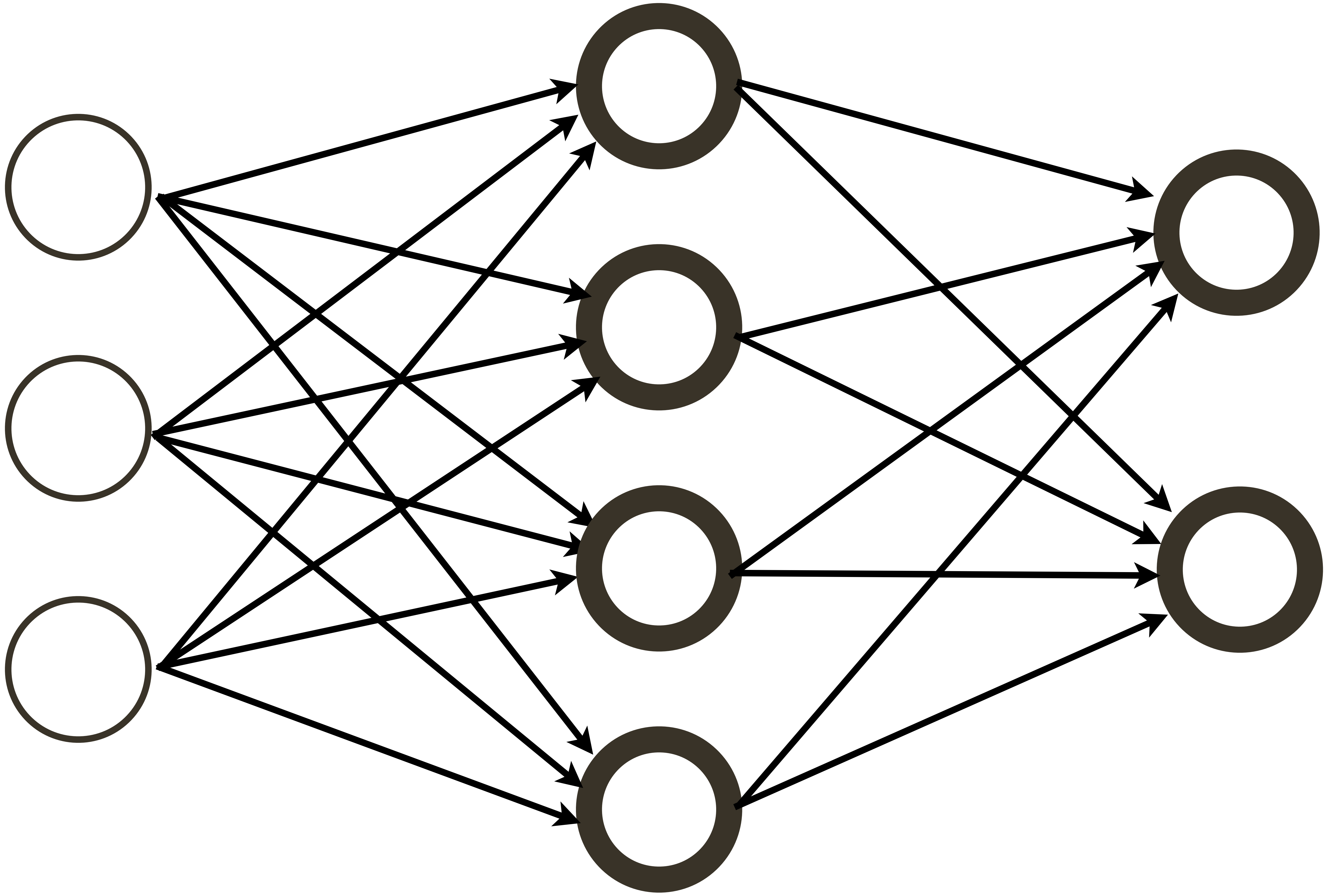
Neural Network

How many neurons?



Neural Network

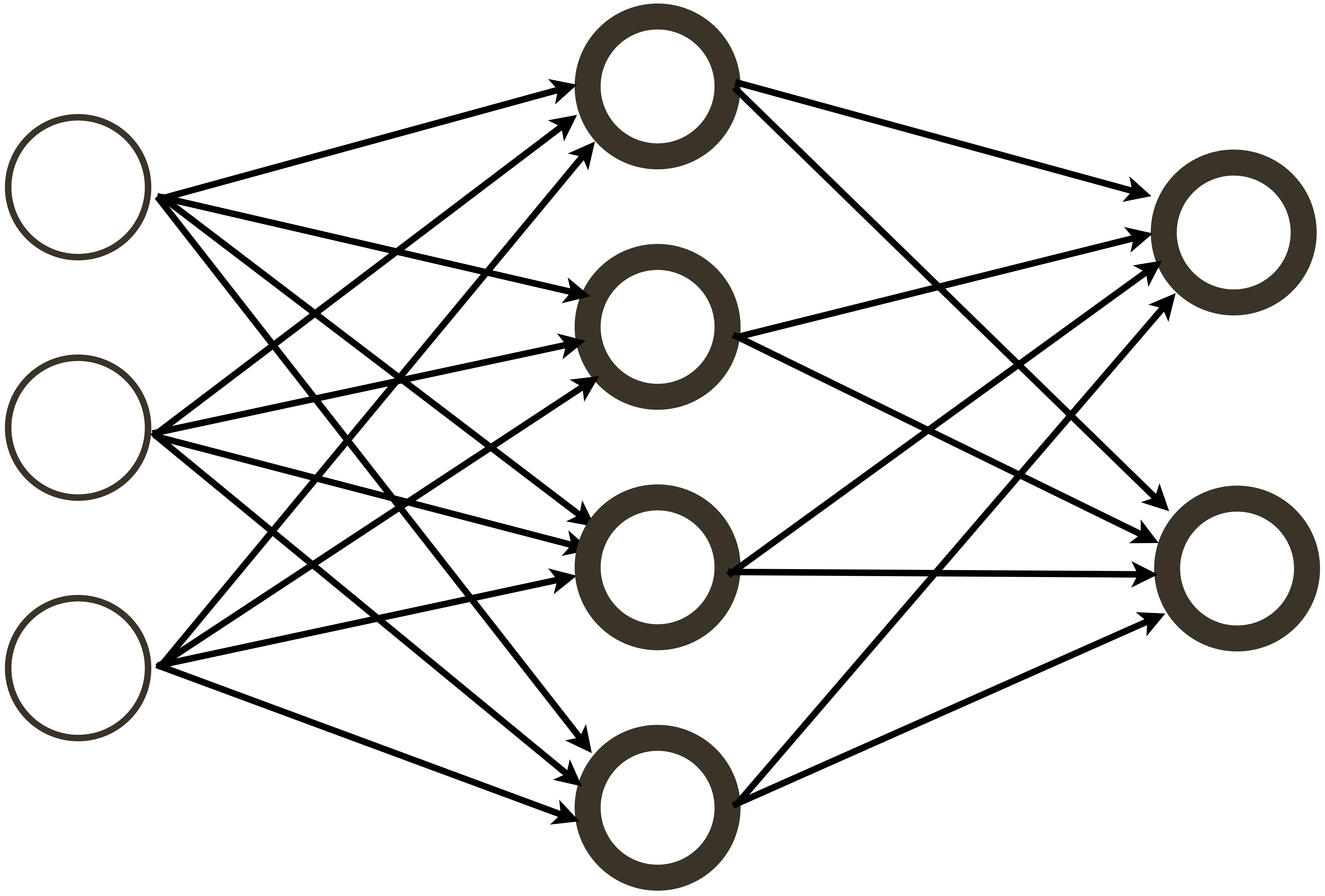
How many neurons?   4+2 = 6



Neural Network

How many neurons? 4+2 = 6

How many weights?

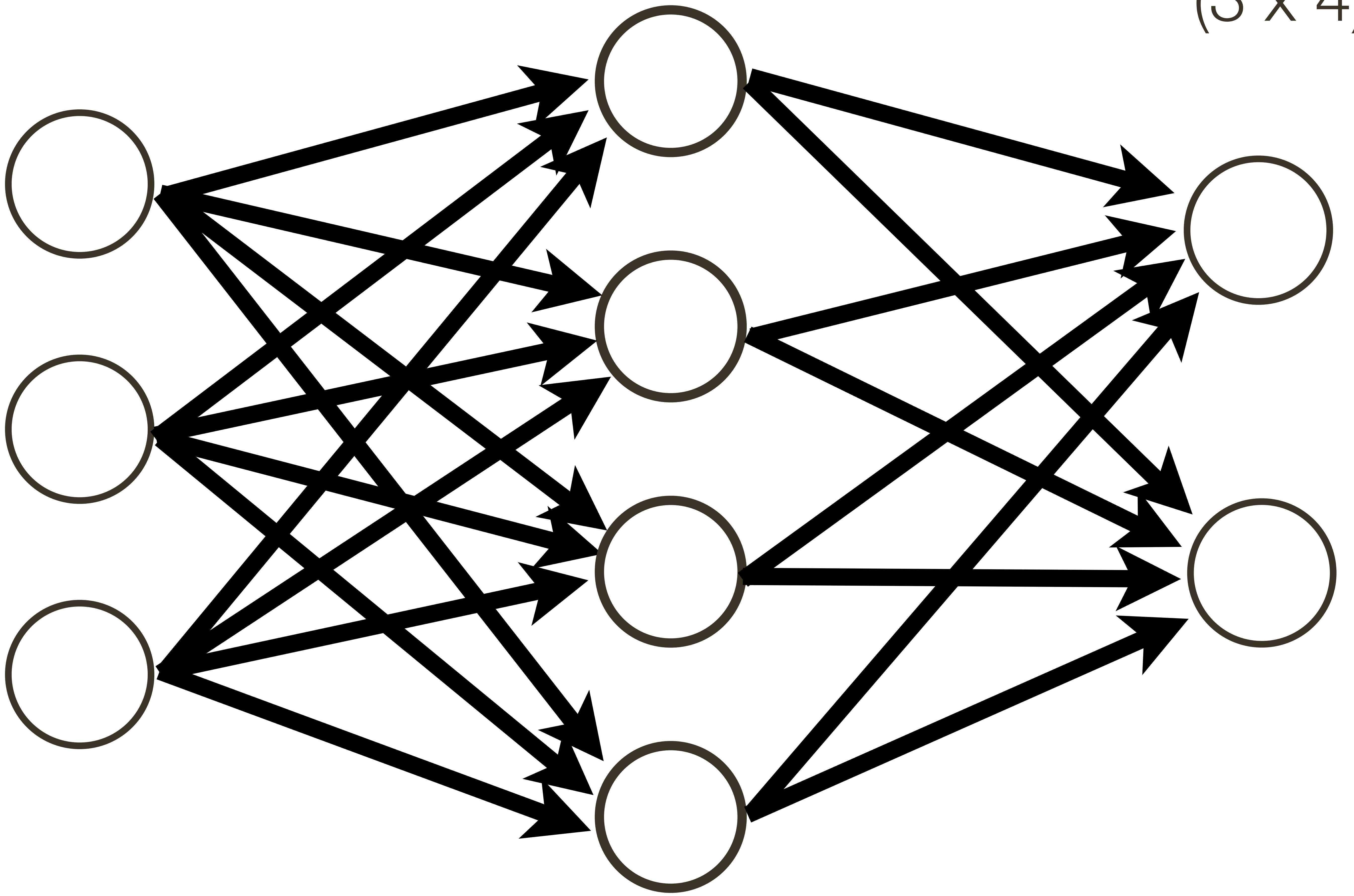


Neural Network

How many neurons? 4+2 = 6

How many weights?

(3 x 4) + (4 x 2) = 20

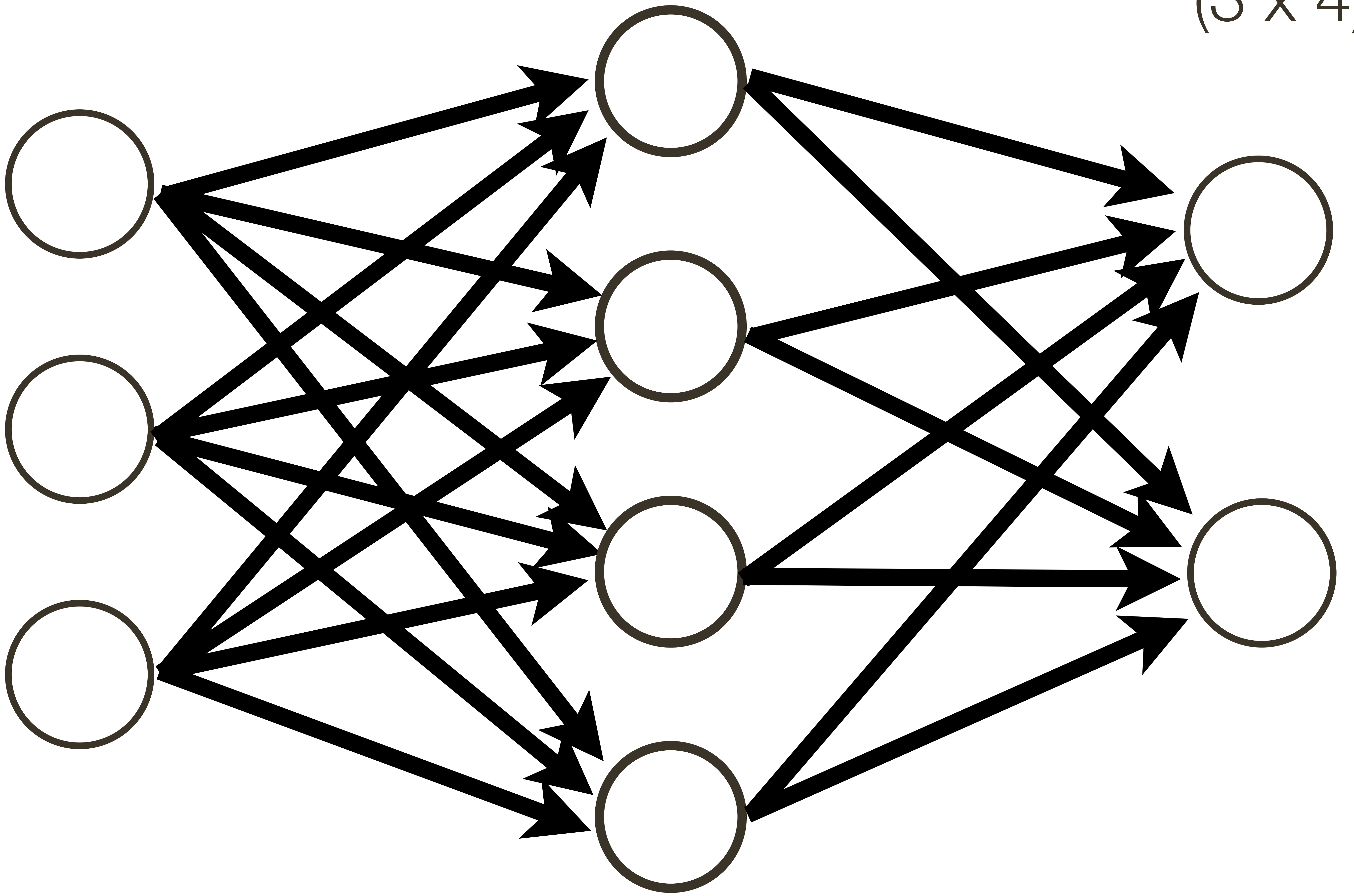


Neural Network

How many neurons? 4+2 = 6

How many weights?

(3 x 4) + (4 x 2) = 20



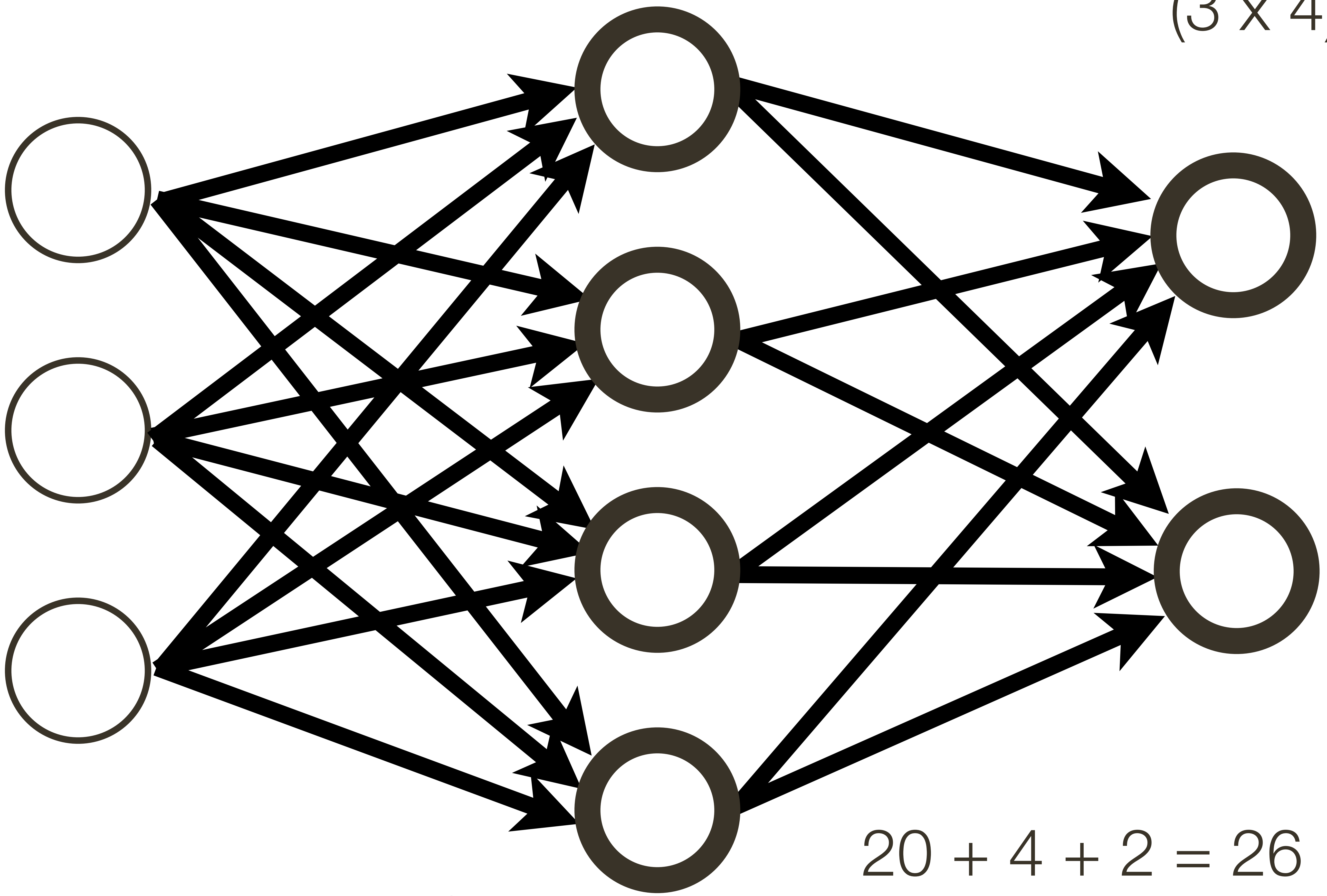
How many learnable parameters?

Neural Network

How many neurons? 4+2 = 6

How many weights?

(3 x 4) + (4 x 2) = 20



How many learnable parameters?

20 + 4 + 2 = 26

bias terms

Neural Networks

Modern **convolutional neural networks** contain 10-20 layers and on the order of 100 million parameters

Training a neural network requires estimating a large number of parameters

Backpropagation

When training a neural network, the final output will be some loss (error) function

— e.g. cross-entropy loss: $\mathcal{L} = -\sum_i y_i \log(\hat{y}_i) \quad \hat{y}_i = \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}$

which defines loss for i-th training example with true class index y_i; and f_j is the j-th element of the vector of class scores coming from neural net.

Backpropagation

When training a neural network, the final output will be some loss (error) function

— e.g. cross-entropy loss: $\mathcal{L} = -\sum_i y_i \log(\hat{y}_i) \quad \hat{y}_i = \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}$

which defines loss for i-th training example with true class index y_i; and f_j is the j-th element of the vector of class scores coming from neural net.

Consider neural net which takes input vector $\mathbf{x}_i$ and predicts scores for 3 classes, with true class being class 3:

Backpropagation

When training a neural network, the final output will be some loss (error) function

— e.g. cross-entropy loss: $\mathcal{L} = -\sum_i y_i \log(\hat{y}_i) \quad \hat{y}_i = \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}$

which defines loss for i-th training example with true class index y_i; and f_j is the j-th element of the vector of class scores coming from neural net.

Consider neural net which takes input vector $\mathbf{x}_i$ and predicts scores for 3 classes, with true class being class 3:

$$f$$

$$c_1 = -2.85$$

$$c_2 = 0.86$$

$$c_3 = 0.28$$

Backpropagation

When training a neural network, the final output will be some loss (error) function

— e.g. cross-entropy loss: $\mathcal{L} = -\sum_i y_i \log(\hat{y}_i) \quad \hat{y}_i = \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}$

which defines loss for i-th training example with true class index y_i; and f_j is the j-th element of the vector of class scores coming from neural net.

Consider neural net which takes input vector $\mathbf{x}_i$ and predicts scores for 3 classes, with true class being class 3:

$$f$$

$$\begin{aligned} c_1 &= -2.85 \\ c_2 &= 0.86 \\ c_3 &= 0.28 \end{aligned} \xrightarrow{\text{exp}} \begin{aligned} &0.058 \\ &2.36 \\ &1.32 \end{aligned}$$

Backpropagation

When training a neural network, the final output will be some loss (error) function

— e.g. cross-entropy loss: $\mathcal{L} = -\sum_i y_i \log(\hat{y}_i) \quad \hat{y}_i = \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}$

which defines loss for i-th training example with true class index y_i; and f_j is the j-th element of the vector of class scores coming from neural net.

Consider neural net which takes input vector $\mathbf{x}_i$ and predicts scores for 3 classes, with true class being class 3:

f	exp	Normalize to sum to 1
$c_1 = -2.85$	0.058	0.016
$c_2 = 0.86$	2.36	0.631
$c_3 = 0.28$	1.32	0.353

Backpropagation

When training a neural network, the final output will be some loss (error) function

— e.g. cross-entropy loss: $\mathcal{L} = -\sum_i y_i \log(\hat{y}_i) \quad \hat{y}_i = \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}$

which defines loss for i-th training example with true class index y_i; and f_j is the j-th element of the vector of class scores coming from neural net.

Consider neural net which takes input vector $\mathbf{x}_i$ and predicts scores for 3 classes, with true class being class 3:

f			Normalize to sum to 1	probability of a class
$c_1 = -2.85$	exp $\longrightarrow$	0.058	$\longrightarrow$	0.016
$c_2 = 0.86$		2.36		0.631
$c_3 = 0.28$		1.32		0.353

Backpropagation

When training a neural network, the final output will be some loss (error) function

— e.g. cross-entropy loss: $\mathcal{L} = -\sum_i y_i \log(\hat{y}_i)$ $\hat{y}_i = \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}$

softmax function
multi-class classifier

which defines loss for i-th training example with true class index y_i; and f_j is the j-th element of the vector of class scores coming from neural net.

Consider neural net which takes input vector $\mathbf{x}_i$ and predicts scores for 3 classes, with true class being class 3:

probability of a class

f

$c_1 = -2.85$ → exp → 0.058 → Normalize to sum to 1 → 0.016

$c_2 = 0.86$ → exp → 2.36 → Normalize to sum to 1 → 0.631

$c_3 = 0.28$ → exp → 1.32 → Normalize to sum to 1 → 0.353

Backpropagation

When training a neural network, the final output will be some loss (error) function

— e.g. cross-entropy loss: $\mathcal{L} = -\sum_i y_i \log(\hat{y}_i) \quad \hat{y}_i = \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}$

which defines loss for i-th training example with true class index y_i; and f_j is the j-th element of the vector of class scores coming from neural net.

Consider neural net which takes input vector $\mathbf{x}_i$ and predicts scores for 3 classes, with true class being class 3:

f				probability of a class	
$c_1 = -2.85$		0.058	Normalize to sum to 1	0.016	
$c_2 = 0.86$	$\xrightarrow{\exp}$	2.36	$\longrightarrow$	0.631	$\mathcal{L} = -\log(0.353) = 1.04$
$c_3 = 0.28$		1.32		0.353	

Backpropagation

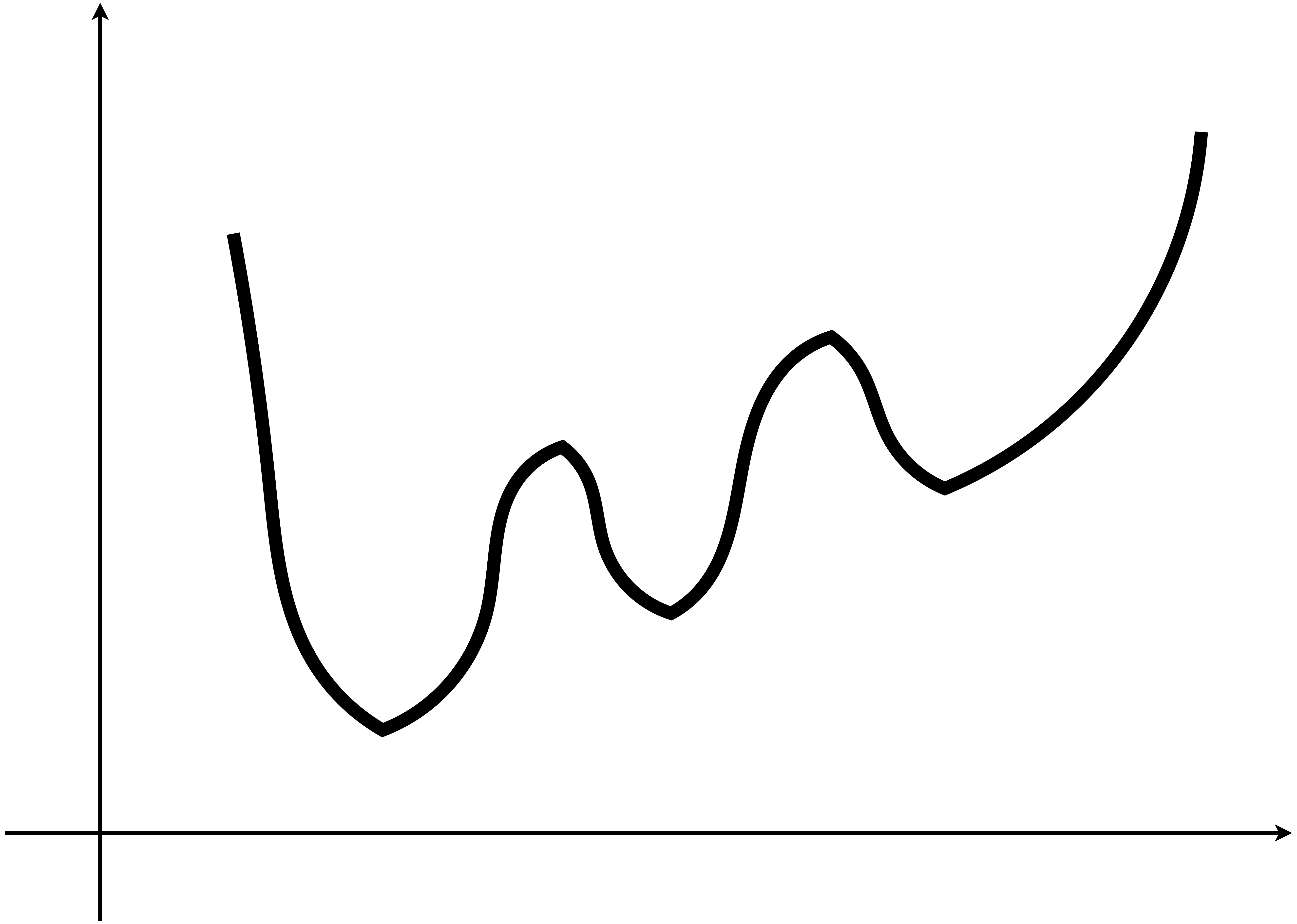
When training a neural network, the final output will be some loss (error) function

— e.g. cross-entropy loss: $\mathcal{L} = -\sum_i y_i \log(\hat{y}_i) \quad \hat{y}_i = \frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}$

which defines loss for i-th training example with true class index y_i; and f_j is the j-th element of the vector of class scores coming from neural net.

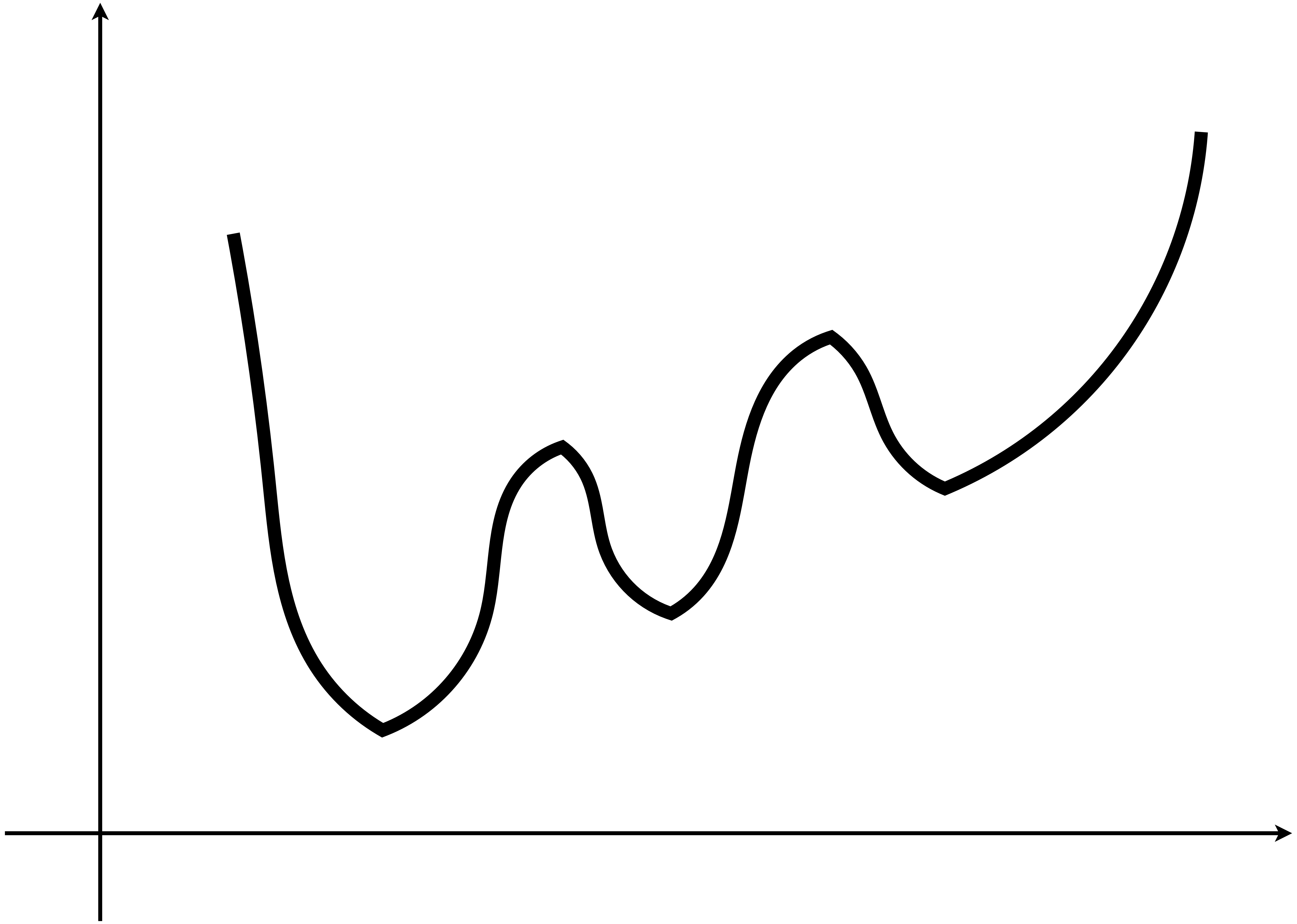
We want to compute the **gradient** of the loss with respect to the network parameters so that we can incrementally adjust the network parameters

Gradient Descent



*slide adopted from V. Ordonex

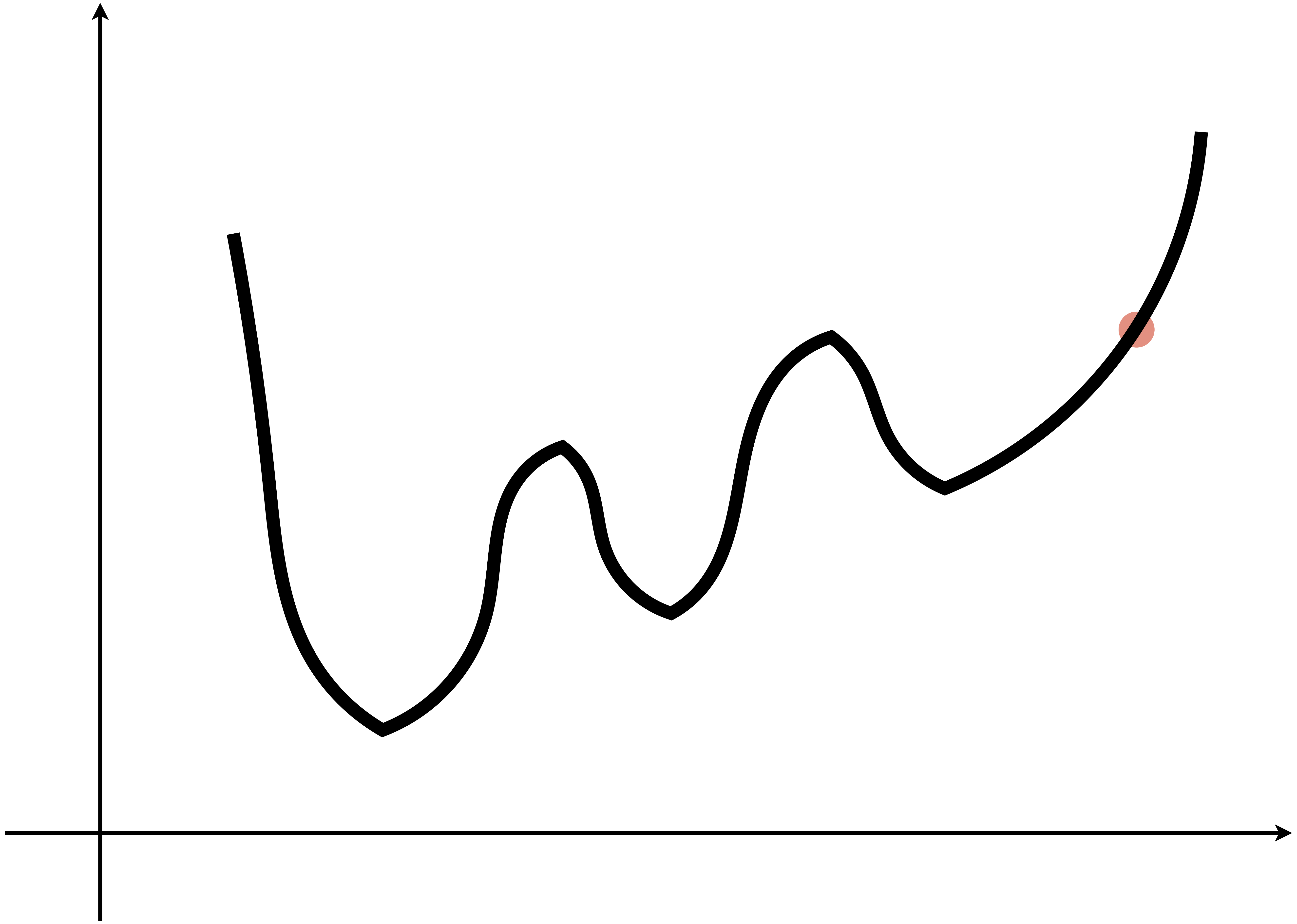
Gradient Descent



1. Start from random value of $\mathbf{W}_0, \mathbf{b}_0$

*slide adopted from V. Ordonex

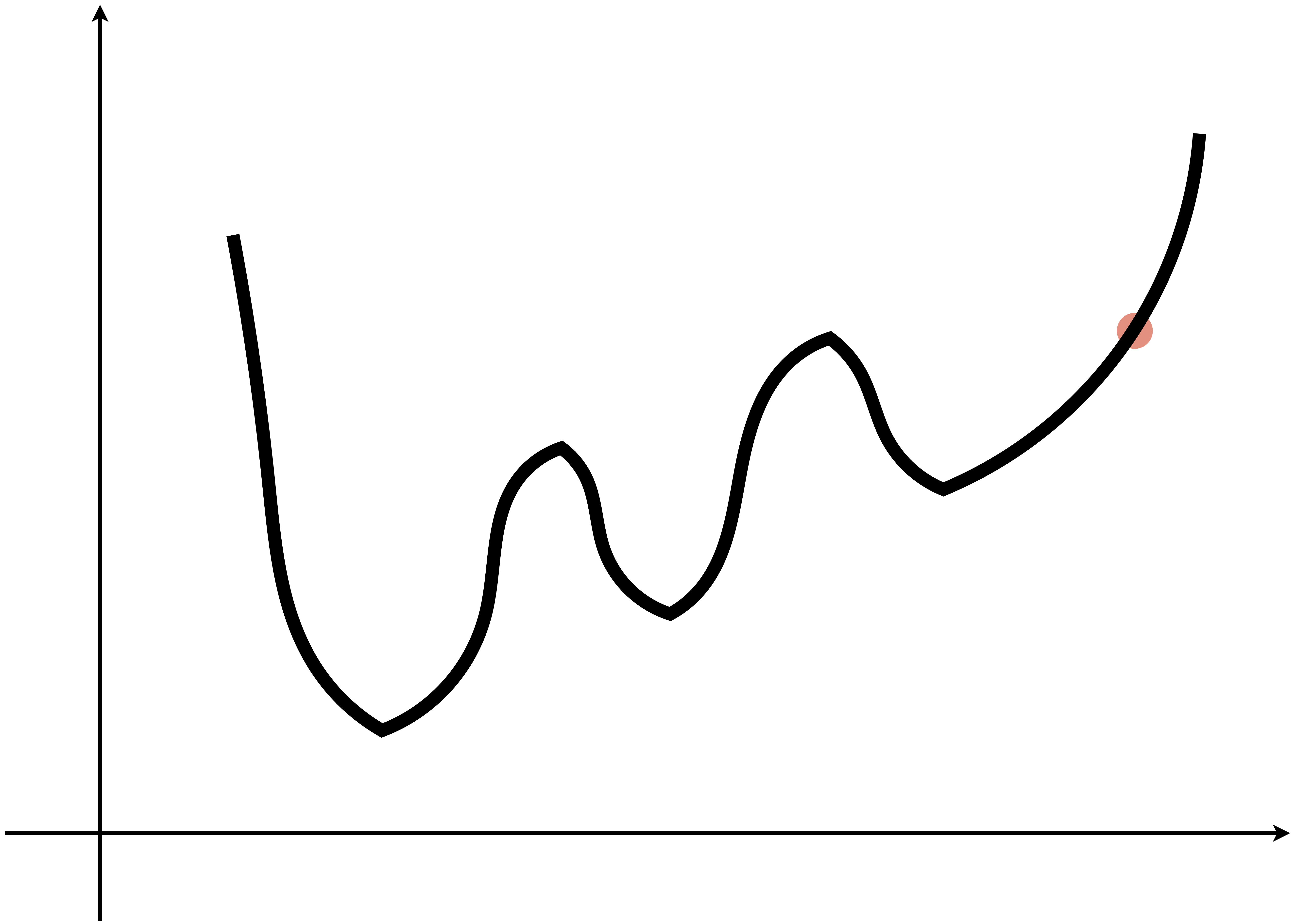
Gradient Descent



1. Start from random value of $\mathbf{W}_0, \mathbf{b}_0$

*slide adopted from V. Ordonex

Gradient Descent



1. Start from random value of $\mathbf{W}_0, \mathbf{b}_0$

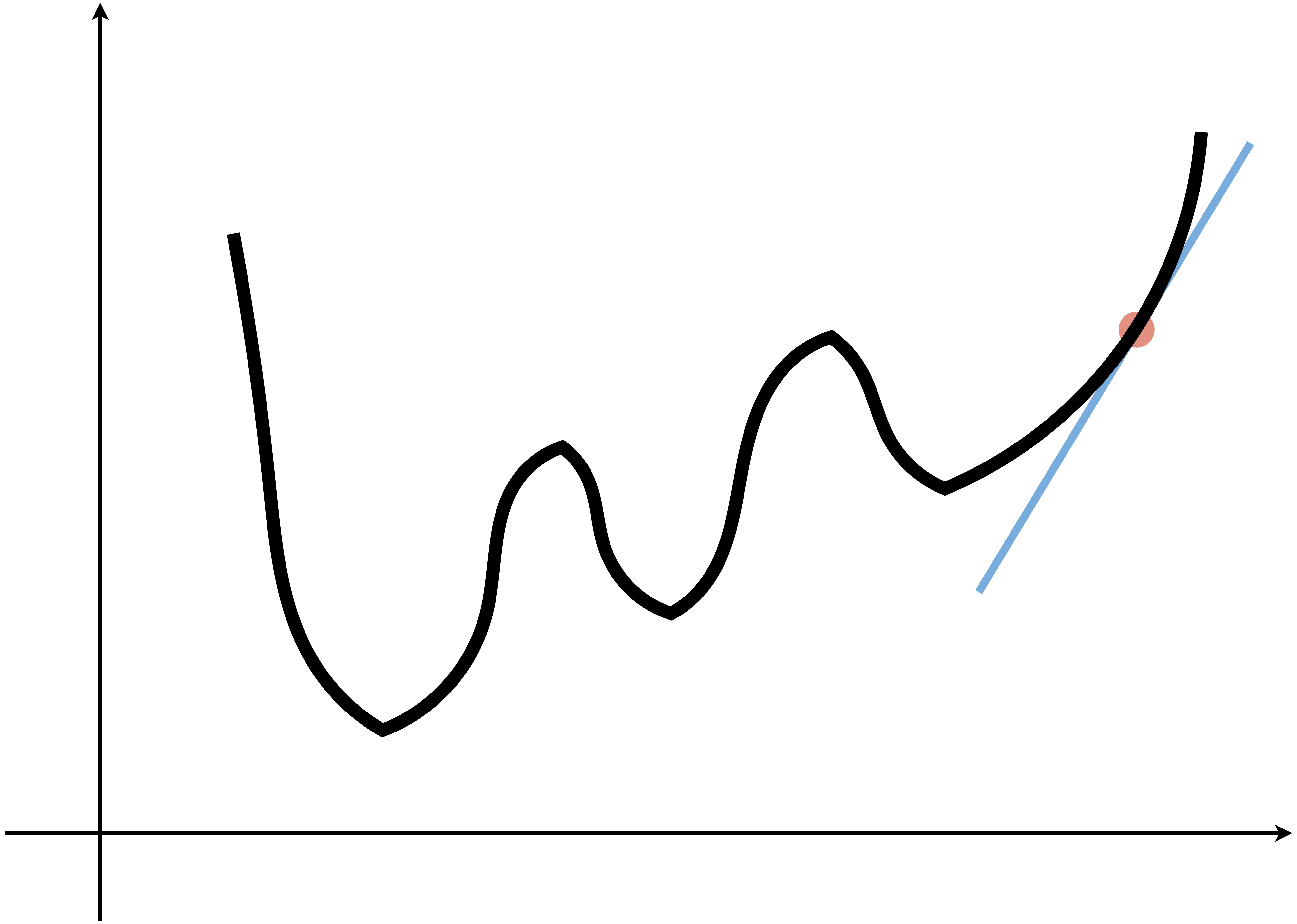
For $k = 0$ to max number of iterations

2. Compute gradient of the loss with respect to previous (initial) parameters:

$$\nabla \mathcal{L}(\mathbf{W}, \mathbf{b})|_{\mathbf{W}=\mathbf{W}_k, \mathbf{b}=\mathbf{b}_k}$$

*slide adopted from V. Ordonex

Gradient Descent



1. Start from random value of $\mathbf{W}_0, \mathbf{b}_0$

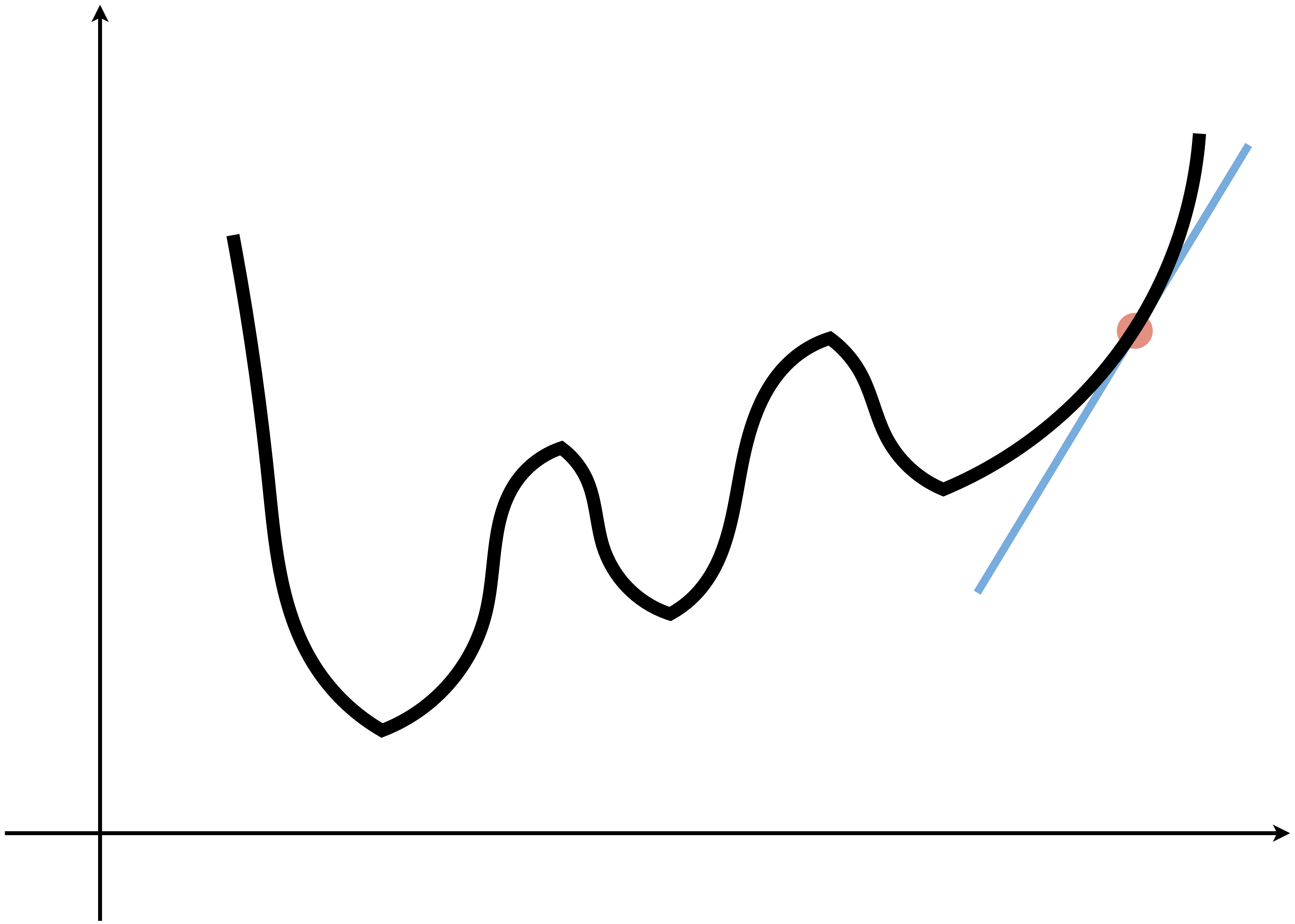
For $k = 0$ to max number of iterations

2. Compute gradient of the loss with respect to previous (initial) parameters:

$$\nabla \mathcal{L}(\mathbf{W}, \mathbf{b})|_{\mathbf{W}=\mathbf{W}_k, \mathbf{b}=\mathbf{b}_k}$$

*slide adopted from V. Ordonex

Gradient Descent



1. Start from random value of $\mathbf{W}_0, \mathbf{b}_0$

For $k = 0$ to max number of iterations

2. Compute gradient of the loss with respect to previous (initial) parameters:

$$\nabla \mathcal{L}(\mathbf{W}, \mathbf{b})\big|_{\mathbf{W}=\mathbf{W}_k, \mathbf{b}=\mathbf{b}_k}$$

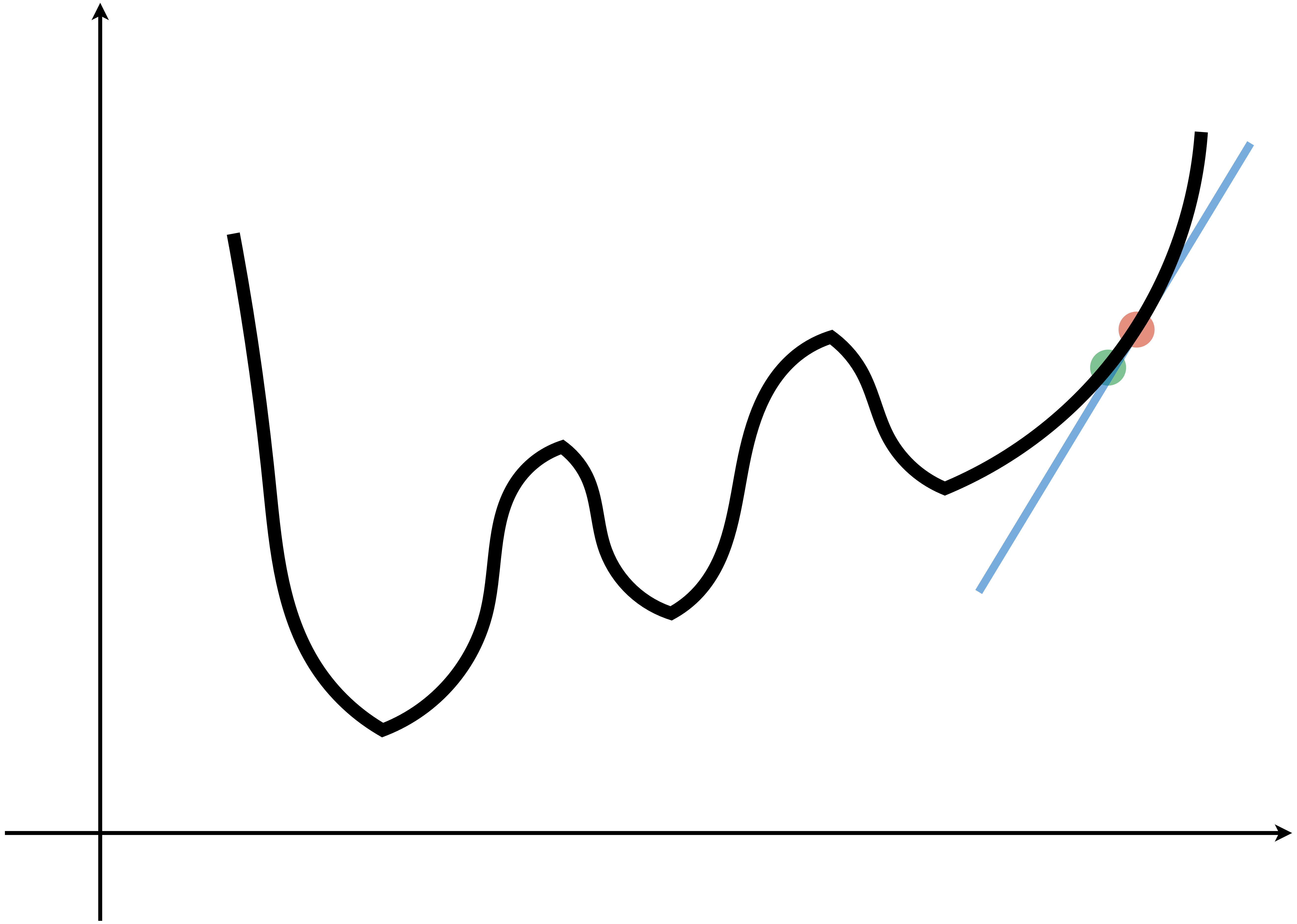
3. Re-estimate the parameters

$$\mathbf{W}_{k+1} = \mathbf{W}_k - \lambda \frac{\partial \mathcal{L}(\mathbf{W}, \mathbf{b})}{\partial \mathbf{W}}\bigg|_{\mathbf{W}=\mathbf{W}_k, \mathbf{b}=\mathbf{b}_k}$$

$$\mathbf{b}_{k+1} = \mathbf{b}_k - \lambda \frac{\partial \mathcal{L}(\mathbf{W}, \mathbf{b})}{\partial \mathbf{b}}\bigg|_{\mathbf{W}=\mathbf{W}_k, \mathbf{b}=\mathbf{b}_k}$$

*slide adopted from V. Ordonex

Gradient Descent



1. Start from random value of $\mathbf{W}_0, \mathbf{b}_0$

For $k = 0$ to max number of iterations

2. Compute gradient of the loss with respect to previous (initial) parameters:

$$\nabla \mathcal{L}(\mathbf{W}, \mathbf{b})\big|_{\mathbf{W}=\mathbf{W}_k, \mathbf{b}=\mathbf{b}_k}$$

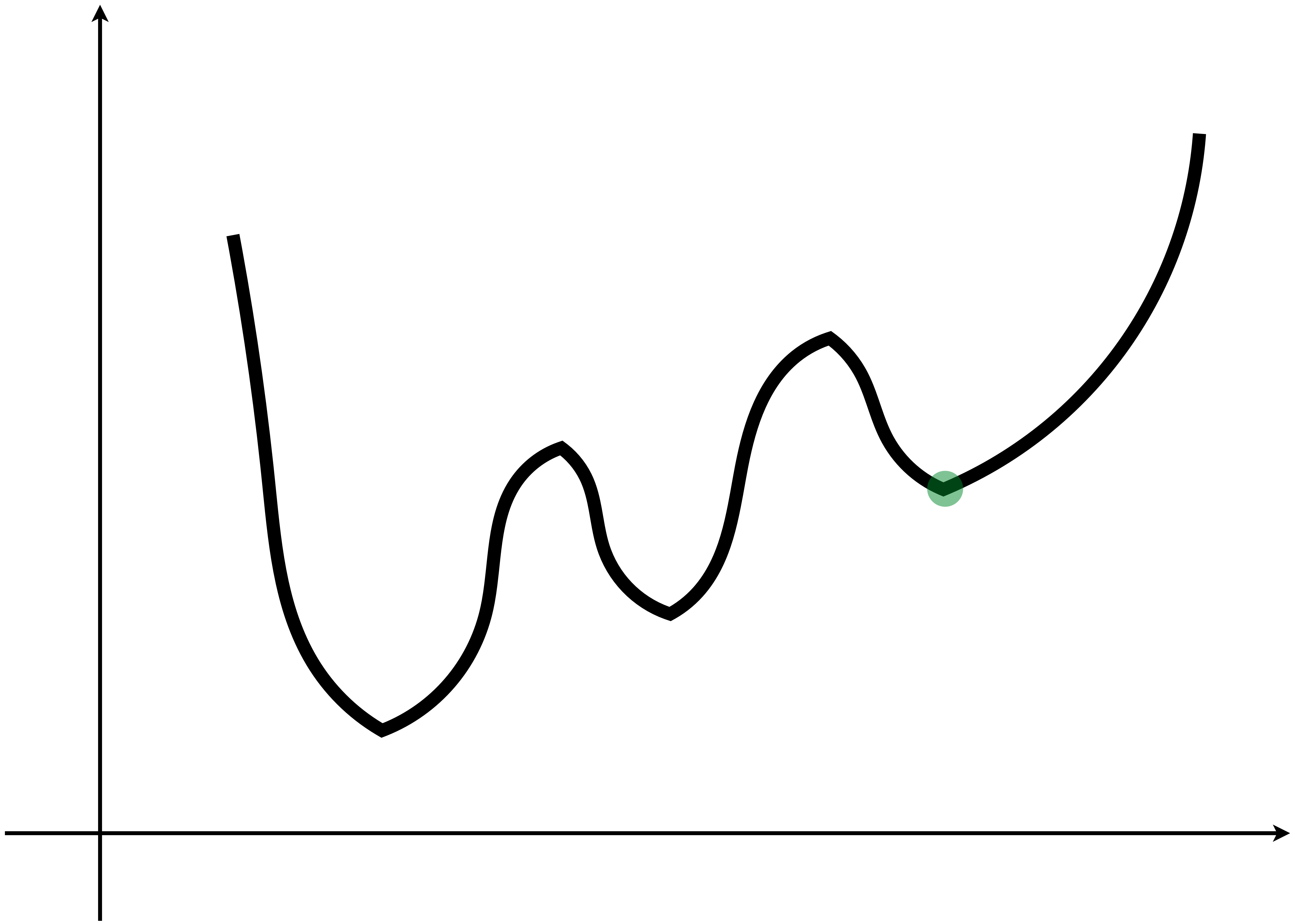
3. Re-estimate the parameters

$$\mathbf{W}_{k+1} = \mathbf{W}_k - \lambda \left.\frac{\partial \mathcal{L}(\mathbf{W}, \mathbf{b})}{\partial \mathbf{W}}\right|_{\mathbf{W}=\mathbf{W}_k, \mathbf{b}=\mathbf{b}_k}$$

$$\mathbf{b}_{k+1} = \mathbf{b}_k - \lambda \left.\frac{\partial \mathcal{L}(\mathbf{W}, \mathbf{b})}{\partial \mathbf{b}}\right|_{\mathbf{W}=\mathbf{W}_k, \mathbf{b}=\mathbf{b}_k}$$

*slide adopted from V. Ordonex

Gradient Descent



1. Start from random value of $\mathbf{W}_0, \mathbf{b}_0$

For $k = 0$ to max number of iterations

2. Compute gradient of the loss with respect to previous (initial) parameters:

$$\nabla \mathcal{L}(\mathbf{W}, \mathbf{b})|_{\mathbf{W}=\mathbf{W}_k, \mathbf{b}=\mathbf{b}_k}$$

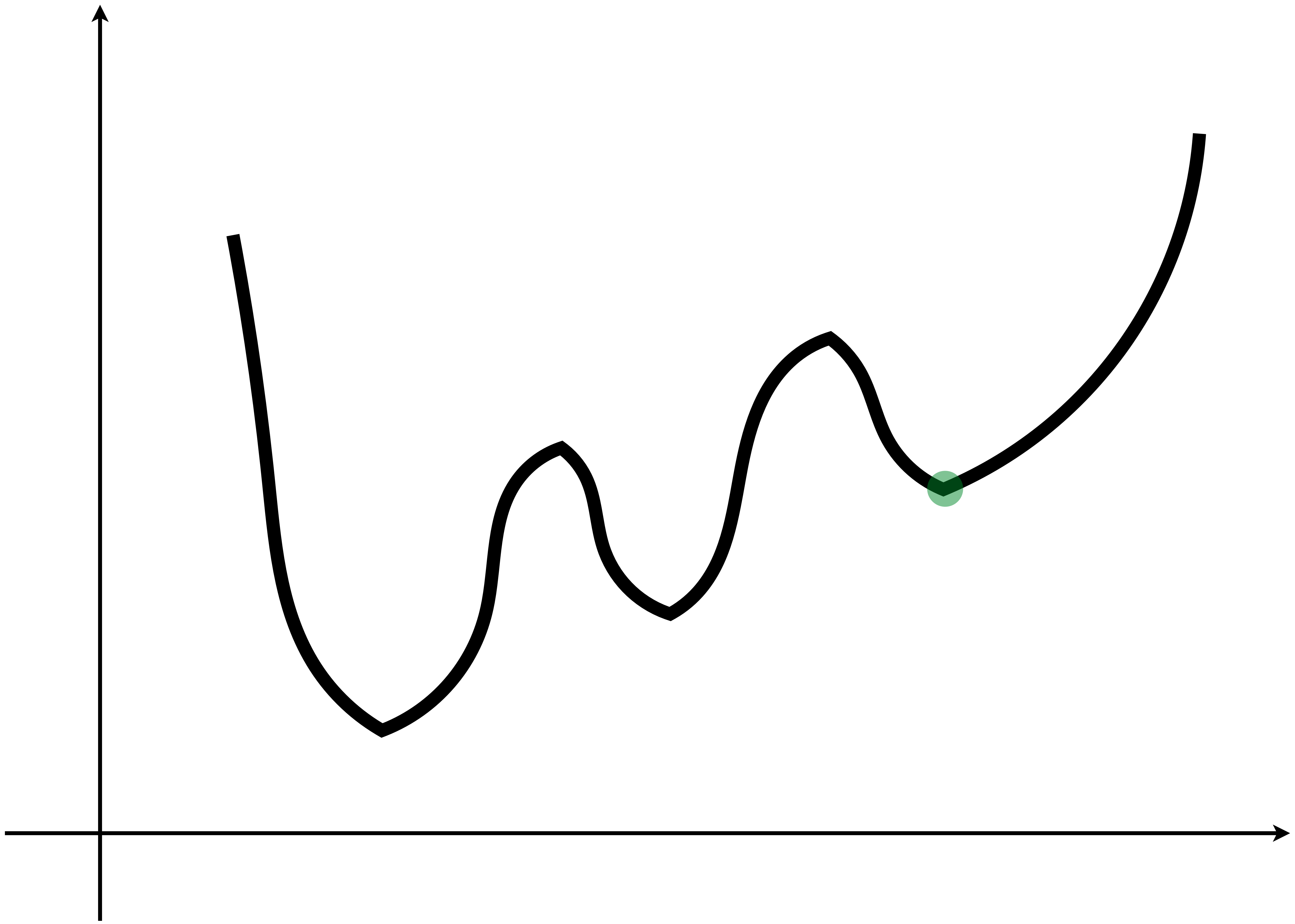
3. Re-estimate the parameters

$$\mathbf{W}_{k+1} = \mathbf{W}_k - \lambda \left. \frac{\partial \mathcal{L}(\mathbf{W}, \mathbf{b})}{\partial \mathbf{W}} \right|_{\mathbf{W}=\mathbf{W}_k, \mathbf{b}=\mathbf{b}_k}$$

$$\mathbf{b}_{k+1} = \mathbf{b}_k - \lambda \left. \frac{\partial \mathcal{L}(\mathbf{W}, \mathbf{b})}{\partial \mathbf{b}} \right|_{\mathbf{W}=\mathbf{W}_k, \mathbf{b}=\mathbf{b}_k}$$

*slide adopted from V. Ordonex

Gradient Descent



λ - is the learning rate

1. Start from random value of $\mathbf{W}_0, \mathbf{b}_0$

For $k = 0$ to max number of iterations

2. Compute gradient of the loss with respect to previous (initial) parameters:

$$\nabla \mathcal{L}(\mathbf{W}, \mathbf{b})|_{\mathbf{W}=\mathbf{W}_k, \mathbf{b}=\mathbf{b}_k}$$

3. Re-estimate the parameters

$$\mathbf{W}_{k+1} = \mathbf{W}_k - \lambda \left. \frac{\partial \mathcal{L}(\mathbf{W}, \mathbf{b})}{\partial \mathbf{W}} \right|_{\mathbf{W}=\mathbf{W}_k, \mathbf{b}=\mathbf{b}_k}$$

$$\mathbf{b}_{k+1} = \mathbf{b}_k - \lambda \left. \frac{\partial \mathcal{L}(\mathbf{W}, \mathbf{b})}{\partial \mathbf{b}} \right|_{\mathbf{W}=\mathbf{W}_k, \mathbf{b}=\mathbf{b}_k}$$

*slide adopted from V. Ordonex