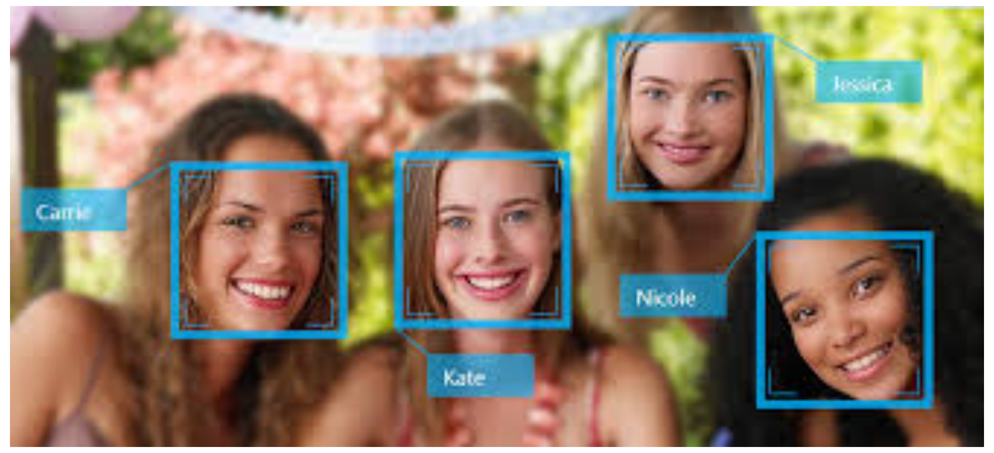
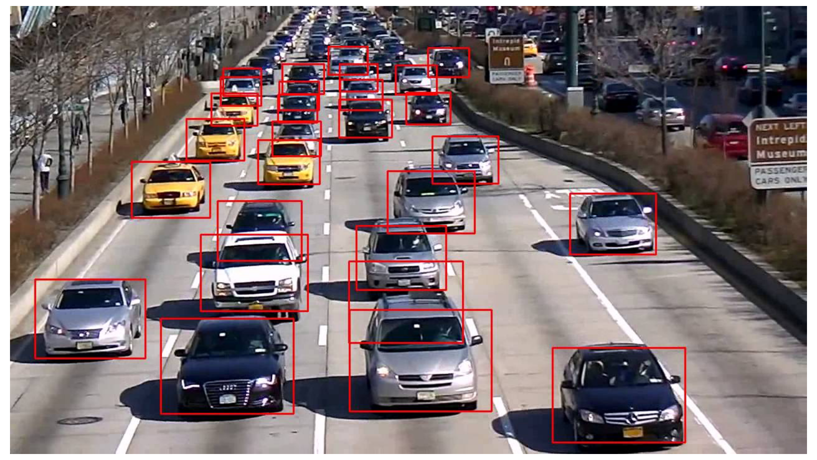
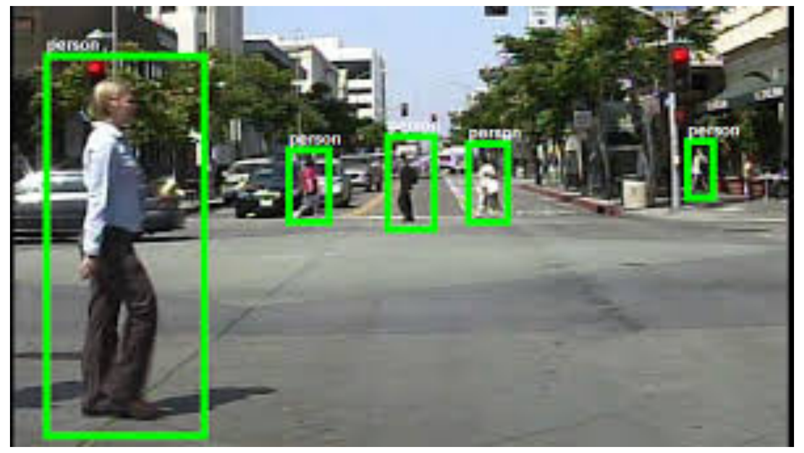
THE UNIVERSITY OF BRITISH COLUMBIA

# CPSC 425: Computer Vision



**Lecture 29:** Object Detection

# **Menu** for Today (**November 20th, 2020**)

**Topics:**

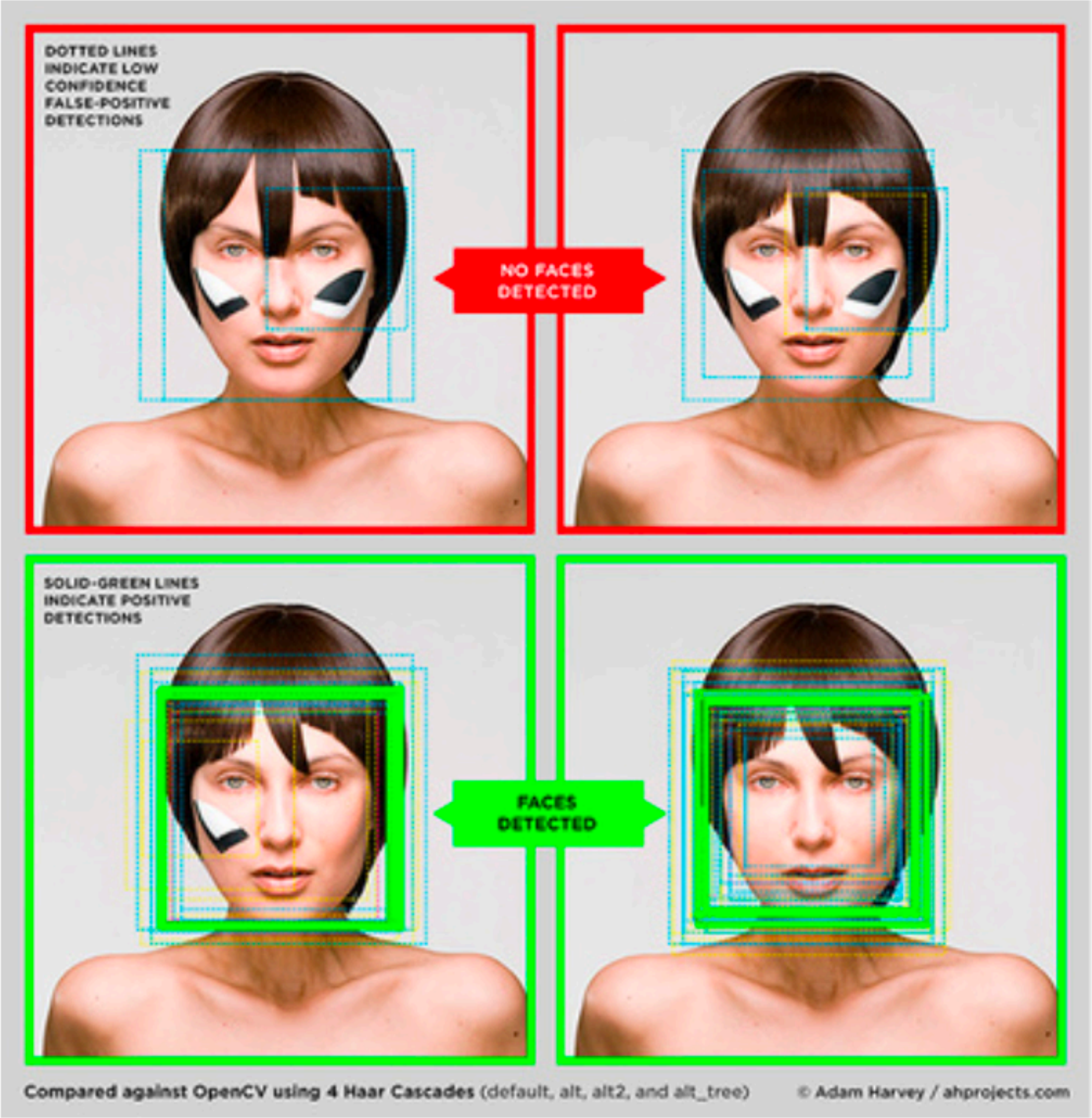
- Neuron
- Neural Networks
- Layers and activation functions
- Backpropagation

**Redings:**

- **Today's** Lecture: N/A
- **Next** Lecture: N/A

# Today's "**fun**" Example: Fooling Face Detection

Just for fun:




"CV Dazzle, a project focused on finding fashionable ways to thwart facial-recognition technology"

**Figure source**: Wired, 2015

# Today’s “**fun**” Example: Fooling Face Detection

Fools Viola-Jones detector

# **Lecture 28**: Re-cap — Sliding Window

Train an image classifier as described previously. 'Slide' a fixed-sized detection window across the image and evaluate the classifier on each window.

Is there a car?

**Image credit**: KITTI Vision Benchmark

# **Lecture 28**: Re-cap — Sliding Window

Train an image classifier as described previously. 'Slide' a fixed-sized detection window across the image and evaluate the classifier on each window.

**Is there a car?**

**Image credit**: KITTI Vision Benchmark

# **Lecture 28**: Re-cap — Sliding Window

Train an image classifier as described previously. ‘Slide’ a fixed-sized detection window across the image and evaluate the classifier on each window.

**Is there a car?**

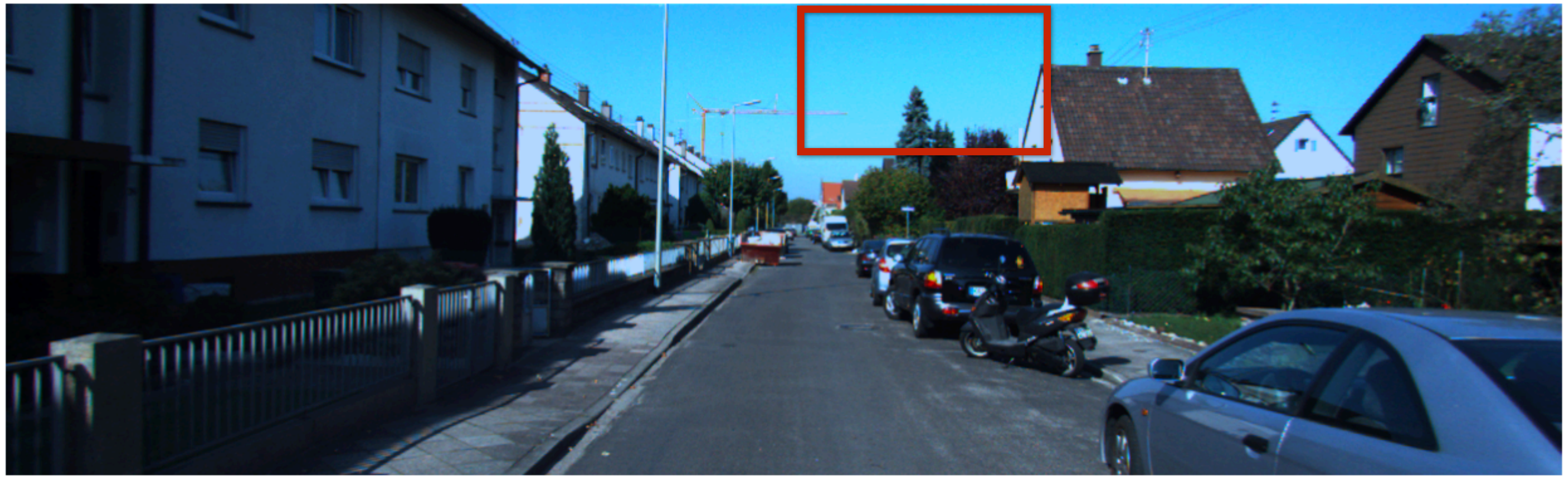


**Image credit**: KITTI Vision Benchmark

# **Lecture 28**: Re-cap — Sliding Window

Train an image classifier as described previously. 'Slide' a fixed-sized detection window across the image and evaluate the classifier on each window.

**Is there a car?**

**Image credit**: KITTI Vision Benchmark

# **Lecture 28**: Re-cap — Sliding Window

Train an image classifier as described previously. 'Slide' a fixed-sized detection window across the image and evaluate the classifier on each window.

**Is there a car?**

**Image credit**: KITTI Vision Benchmark

# **Lecture 28**: Re-cap — Sliding Window

Train an image classifier as described previously. 'Slide' a fixed-sized detection window across the image and evaluate the classifier on each window.

Is there a car?

**Image credit**: KITTI Vision Benchmark

# **Lecture 28**: Re-cap — Sliding Window

Train an image classifier as described previously. 'Slide' a fixed-sized detection window across the image and evaluate the classifier on each window.

**Is there a car?**

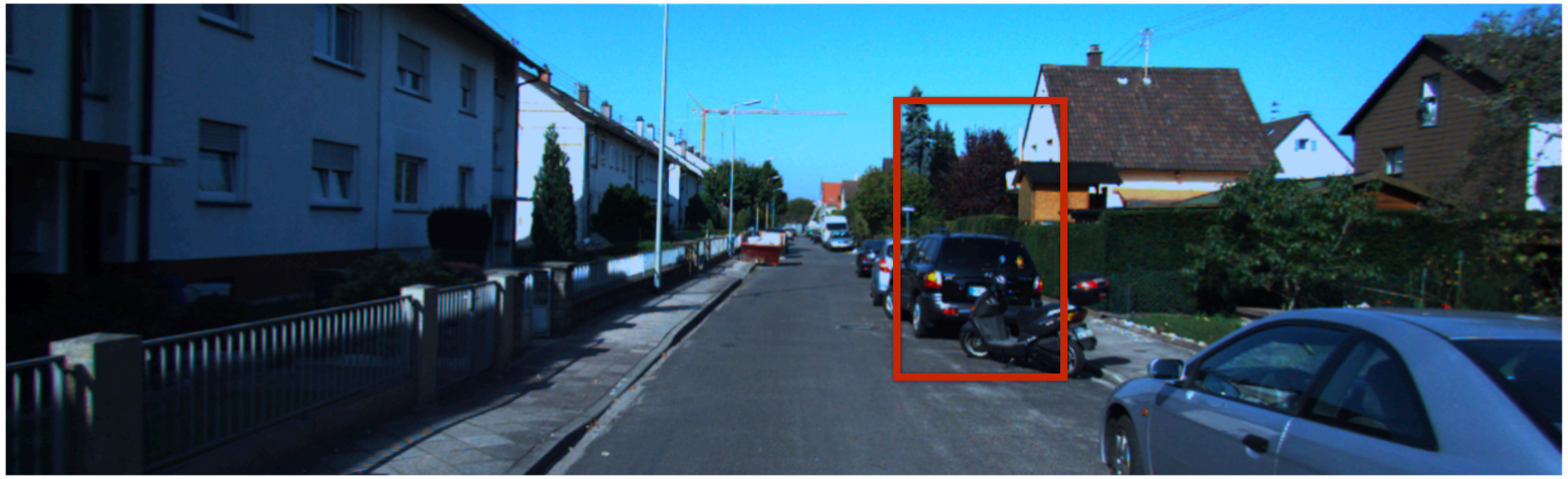


**Image credit**: KITTI Vision Benchmark

# **Lecture 28**: Re-cap — Sliding Window

Train an image classifier as described previously. 'Slide' a fixed-sized detection window across the image and evaluate the classifier on each window.

**Is there a car?**

**Image credit**: KITTI Vision Benchmark

# **Lecture 28**: Re-cap — Sliding Window

Train an image classifier as described previously. ‘Slide’ a fixed-sized detection window across the image and evaluate the classifier on each window.

**Image credit**: KITTI Vision Benchmark

This is a search over location
— We have to search over scale as well
— We may also have to search over aspect ratios

# **Lecture 28**: Re-cap — Viola-Jones Face Detection

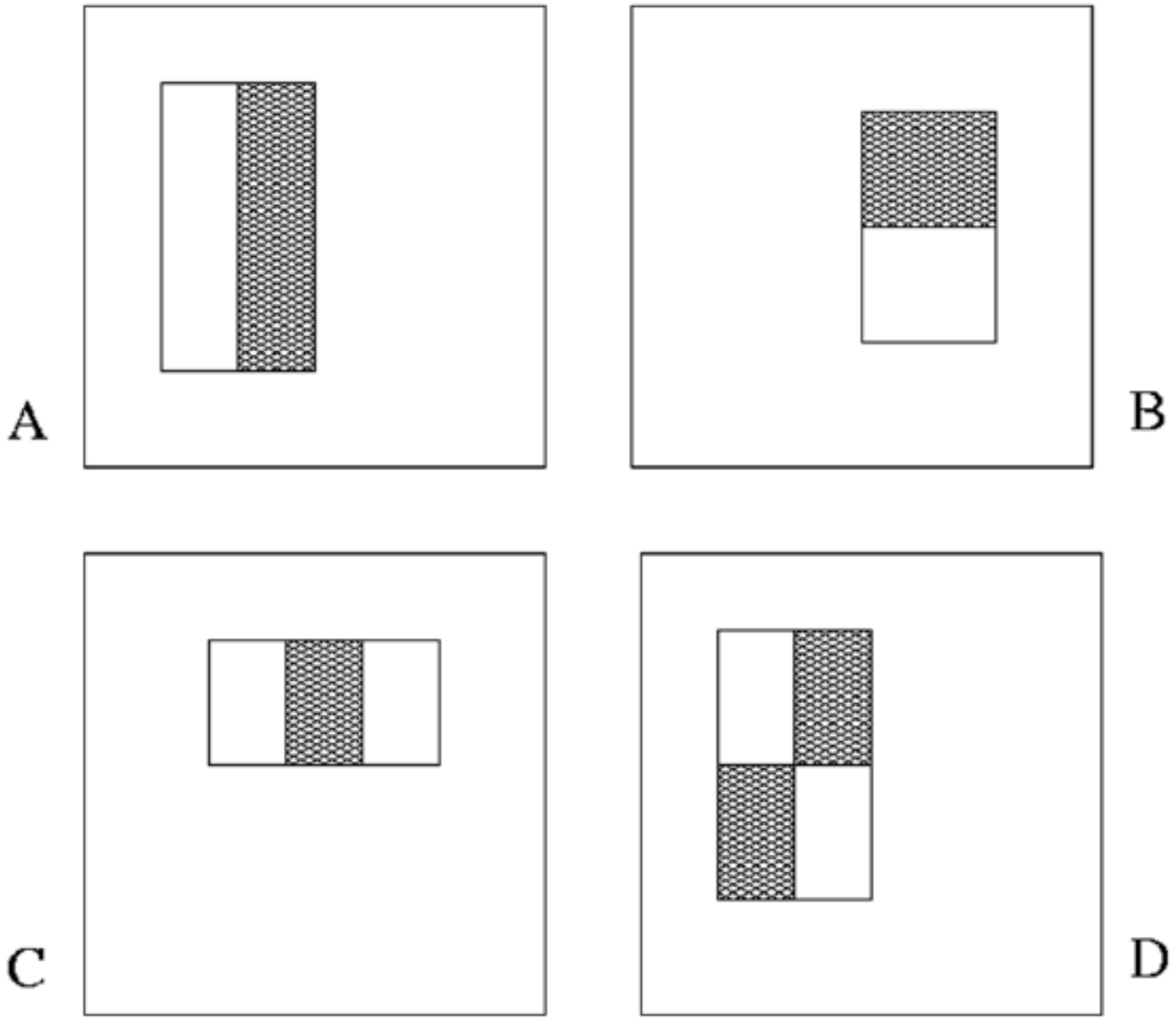
The **Viola-Jones** face detector is a classic sliding window detector that learns both efficient features and a classifier

A key strategy is to use features that are fast to evaluate to reject most windows early

The Viola-Jones detector computes ‘rectangular’ features within each window

# **Lecture 28**: Re-cap — Viola-Jones Face Detection

A 'rectangular' feature is computed by summing up pixel values within rectangular regions and then differencing those region sums




a.k.a. **Harr** Wavelets

**Figure credit**: P. Viola and M. Jones, 2001

# **Lecture 28**: Re-cap — Viola-Jones Face Detection

1. Select best filter/threshold combination

   a. Normalize the weights $w_{t,i} \leftarrow \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$

   $$h_j(x) = \begin{cases} 1 & \text{if } f_j(x) > \theta_j \\ 0 & \text{otherwise} \end{cases}$$

   b. For each feature, *j* $\varepsilon_j = \sum_i w_i \left| h_j(x_i) - y_i \right|$

   c. Choose the classifier, ***h**$_t$* with the lowest error $\varepsilon_t$

2. Re-weight examples

$$w_{t+1,i} = w_{t,i} \beta_t^{1-\left|h_t(x_i)-y_i\right|} \qquad \beta_t = \frac{\varepsilon_t}{1-\varepsilon_t}$$



**Image Credit**: Ioannis (Yannis) Gkioulekas (CMU)

# **Lecture 28**: Re-cap — Viola-Jones Face Detection

## Viola & Jones algorithm

3. The final strong classifier is

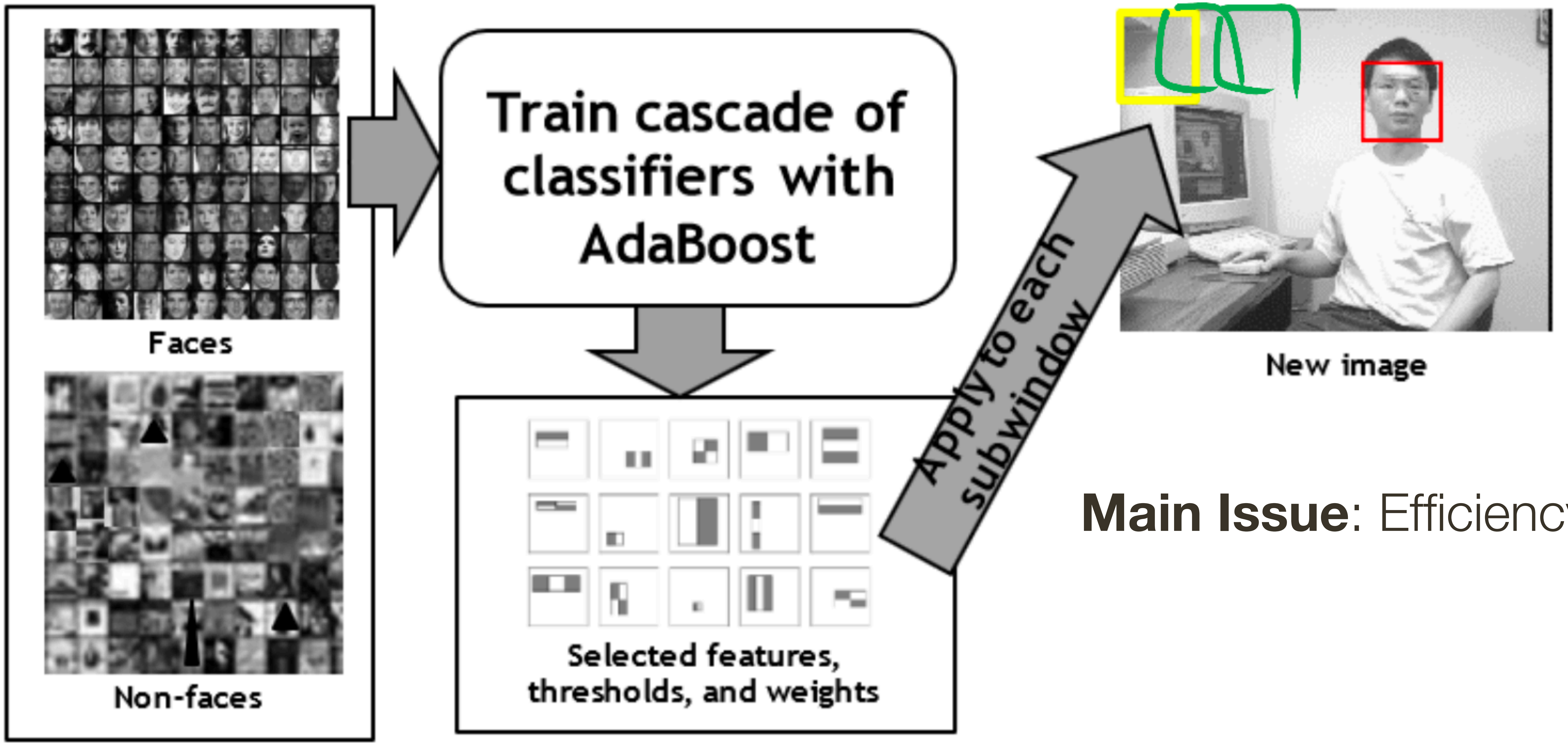
$$h(x) = \begin{cases} 1 & \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & \text{otherwise} \end{cases}$$

$$\alpha_t = \log \frac{1}{\beta_t}$$

The final strong classifier is a weighted linear combination of the T weak classifiers where the weights are inversely proportional to the training errors

**Image Credit**: Ioannis (Yannis) Gkioulekas (CMU)

# **Lecture 28**: Re-cap — Viola-Jones Face Detection




**Main Issue**: Efficiency

**Figure credit**: K. Grauman

# **Cascading** Classifiers

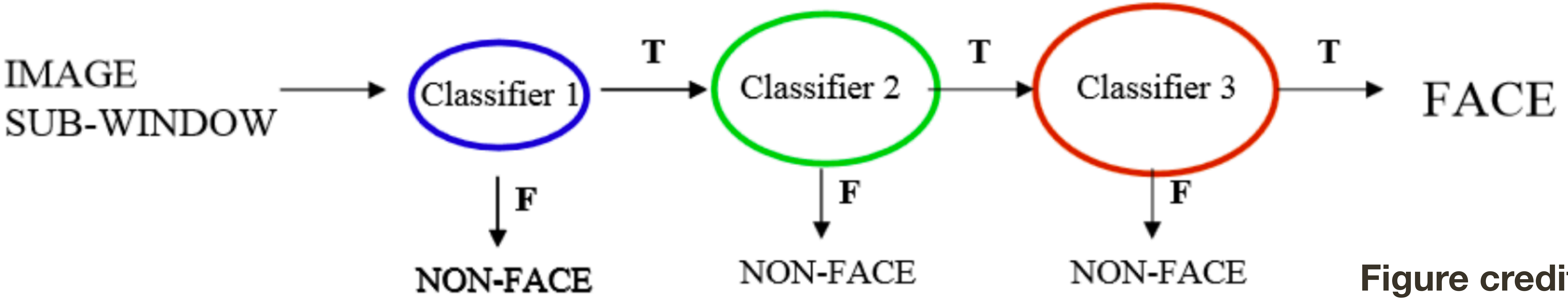



**Figure credit**: P. Viola

To make detection **faster**, features can be reordered by increasing complexity of evaluation and the thresholds adjusted so that the early (simpler) tests have few or no false negatives

Any window that is rejected by early tests can be discarded quickly without computing the other features

This is referred to as a **cascade** architecture

# **Hard** Negative Mining

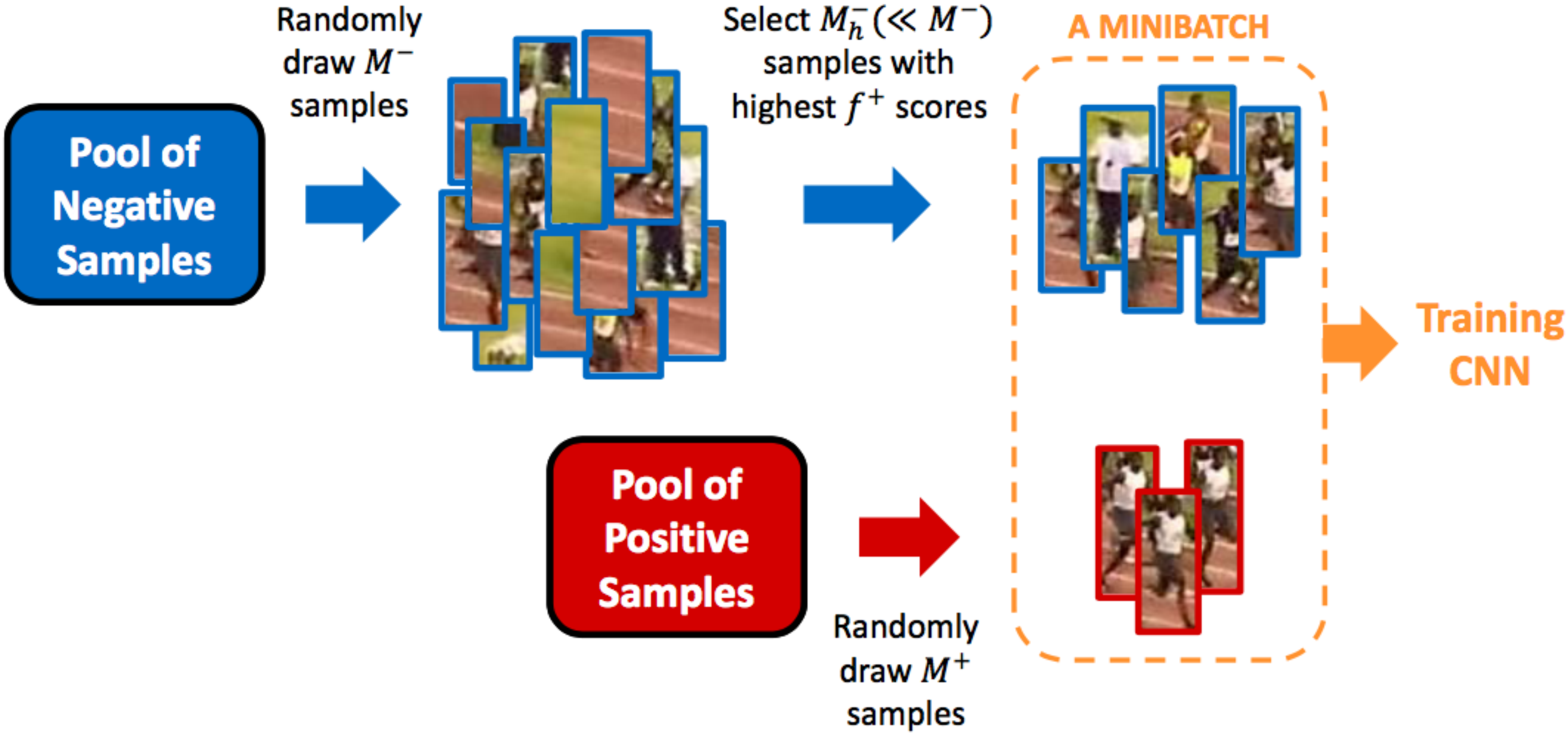



**Image From**: Jamie Kang

# **Viola-Jones** in Action

https://vimeo.com/12774628

# **Viola-Jones** in Action

https://vimeo.com/12774628

# **Recall**: Sliding Window

Train an image classifier as described previously. ‘Slide’ a fixed-sized detection window across the image and evaluate the classifier on each window.

**Image credit**: KITTI Vision Benchmark

# **Recall**: Sliding Window

Train an image classifier as described previously. ‘Slide’ a fixed-sized detection window across the image and evaluate the classifier on each window.

**Image credit**: KITTI Vision Benchmark

This is a lot of possible windows! And most will not contain the object we are looking for.

# **Object** Proposals

**Object proposal** algorithms generate a short list of regions that have generic object-like properties

— These regions are likely to contain some kind of foreground object instead of background texture

The object detector then considers these candidate regions only, instead of exhaustive sliding window search

# **Object** Proposals

First introduced by Alexe et al., who asked 'what is an object?' and defined an 'objectness' score based on several visual cues

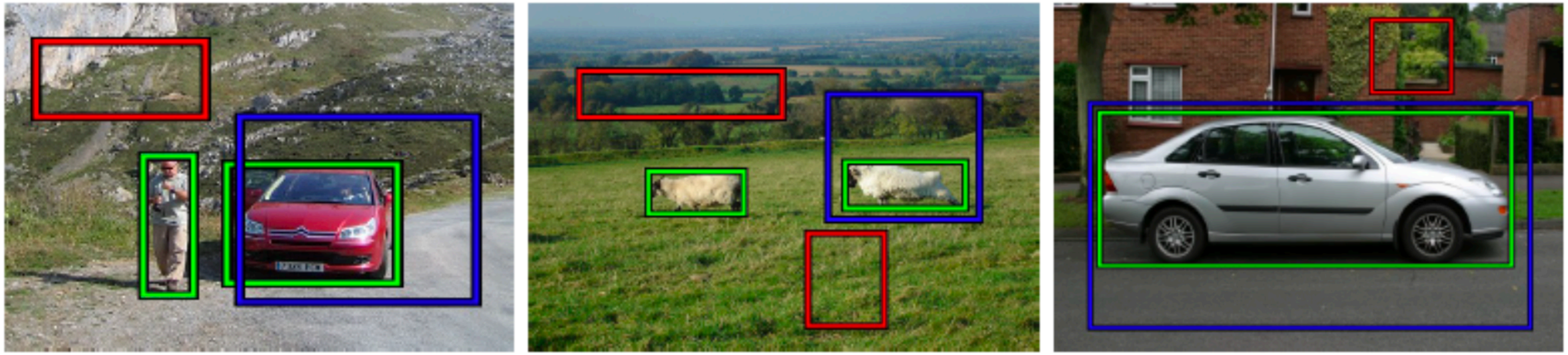


**Figure credit**: Alexe et al., 2012

# **Object** Proposals

First introduced by Alexe et al., who asked ‘what is an object?’ and defined an ‘objectness’ score based on several visual cues

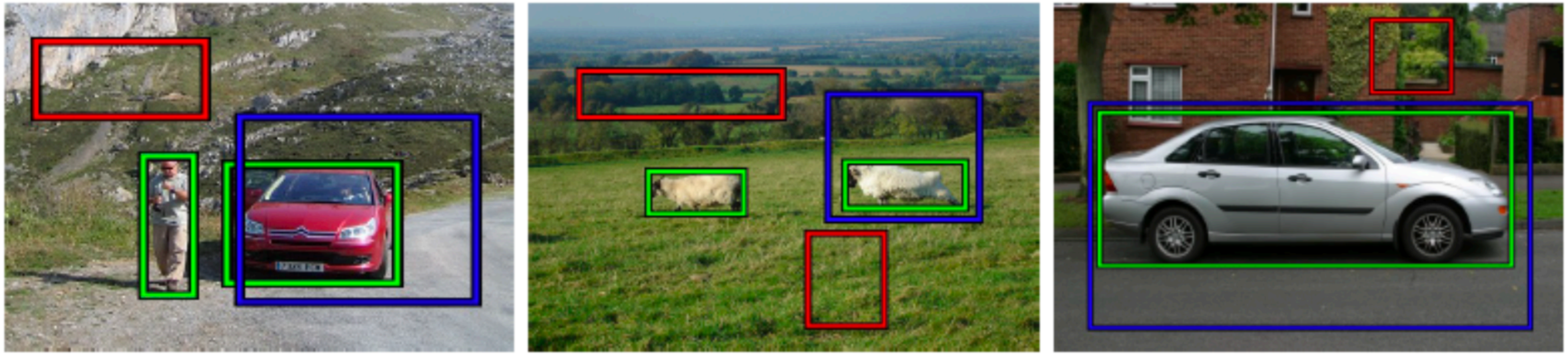


**Figure credit**: Alexe et al., 2012

This work argued that objects typically

— are unique within the image and stand out as salient

— have a contrasting appearance from surroundings and/or

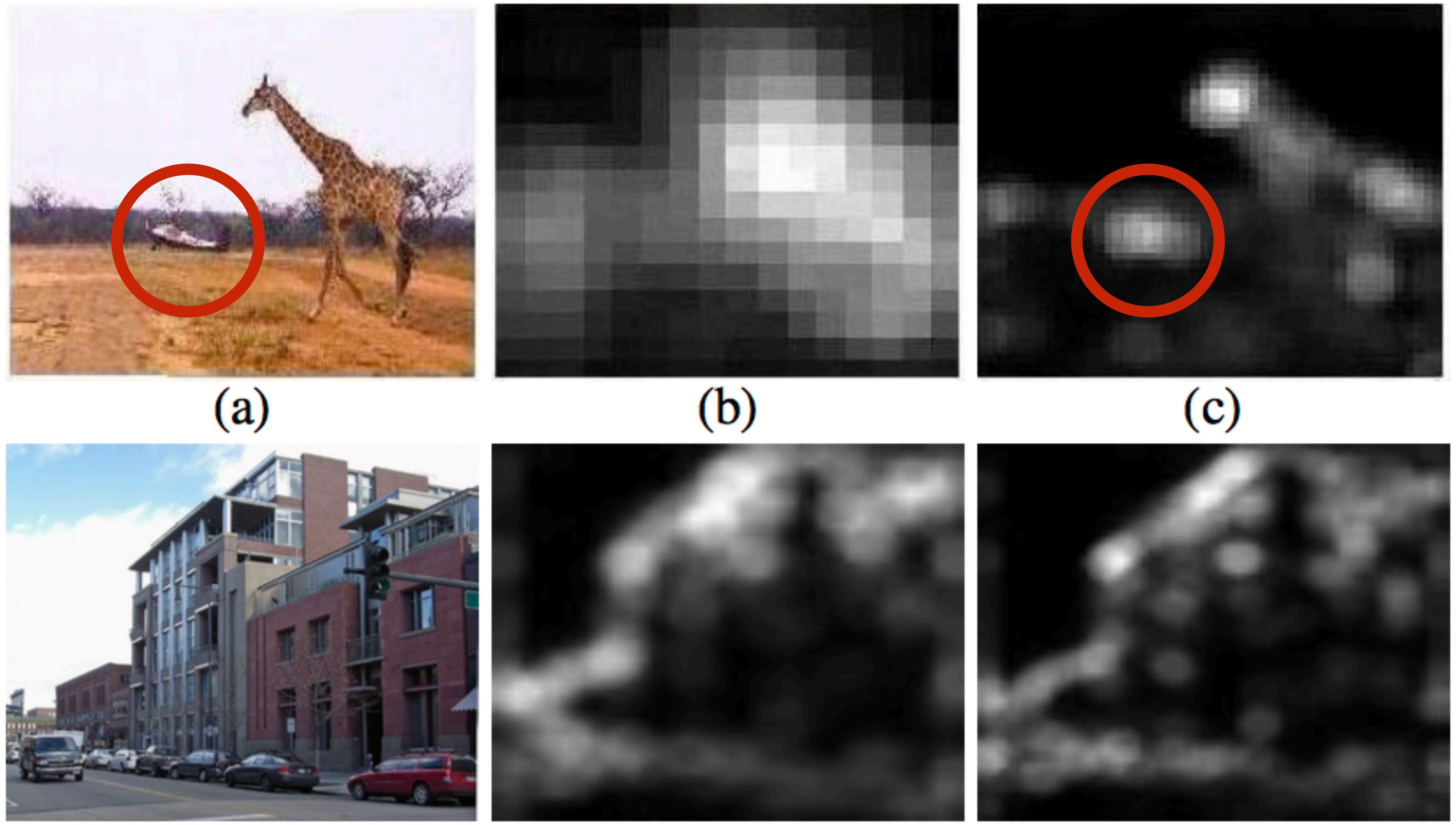
— have a well-defined closed boundary in space

# **Object** Proposals

Multiscale **Saliency**

— Favors regions with a unique appearance within the image

**High** scale **Low** scale



(a) (b) (c)

**Successful** Case

(d) (e) (f)

**Failure** Case

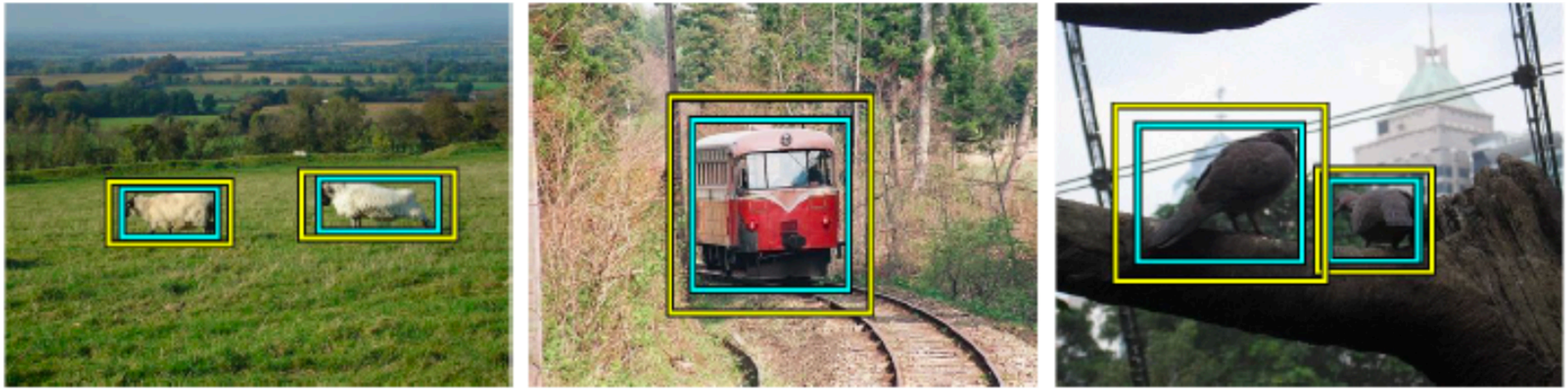


**Figure credit**: Alexe et al., 2012

# **Object** Proposals

**Colour Contrast**

— Favors regions with a contrasting colour appearance from immediate surroundings



**Successful** Cases

**Failure** Case



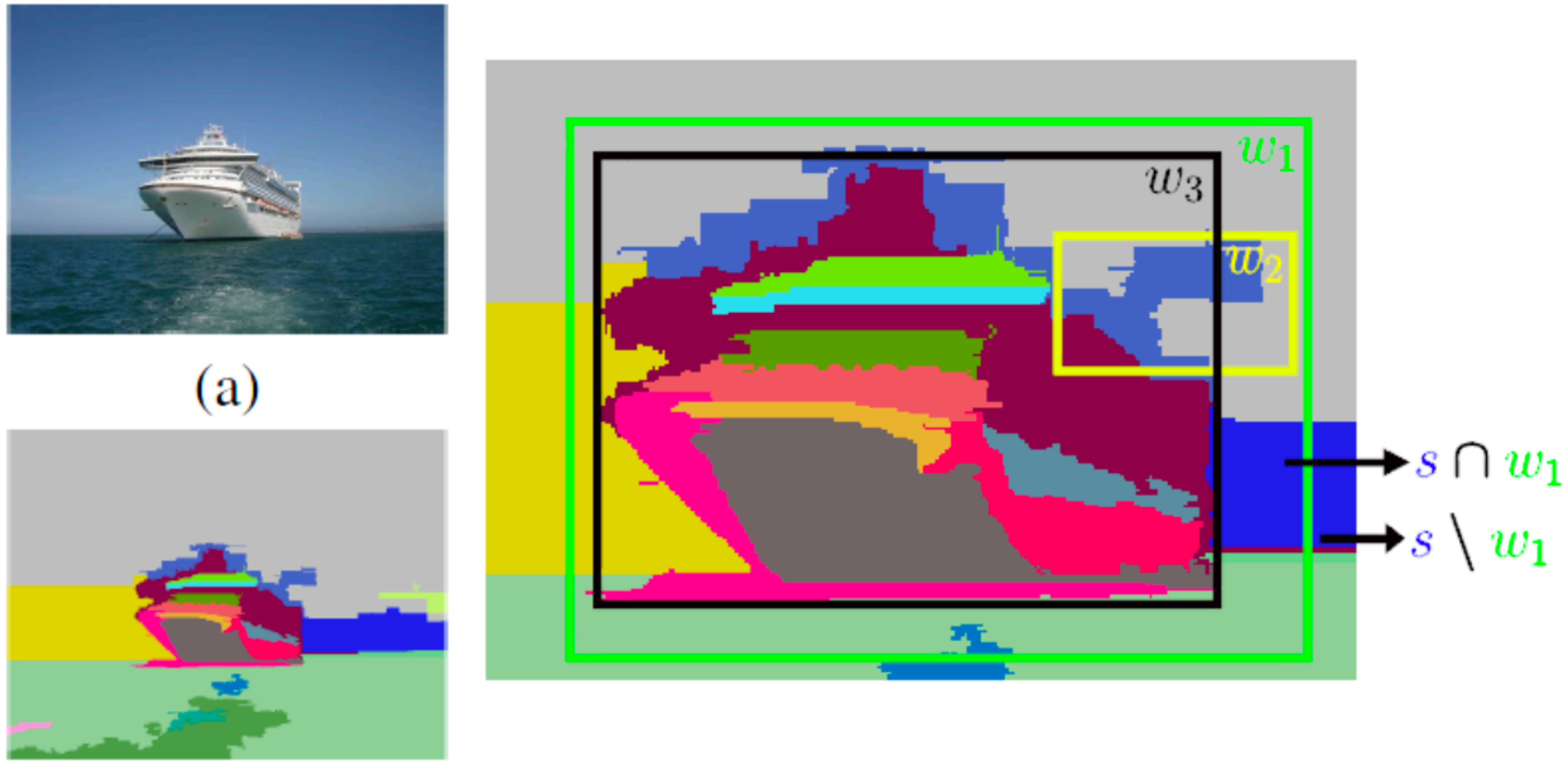
**Figure credit**: Alexe et al., 2012

# **Object** Proposals

**Superpixels** Straddling

— Favors regions with a well-defined closed boundary

— Measures the extent to which superpixels (obtained by image segmentation) contain pixels both inside and outside of the window




(a) (b) (c)



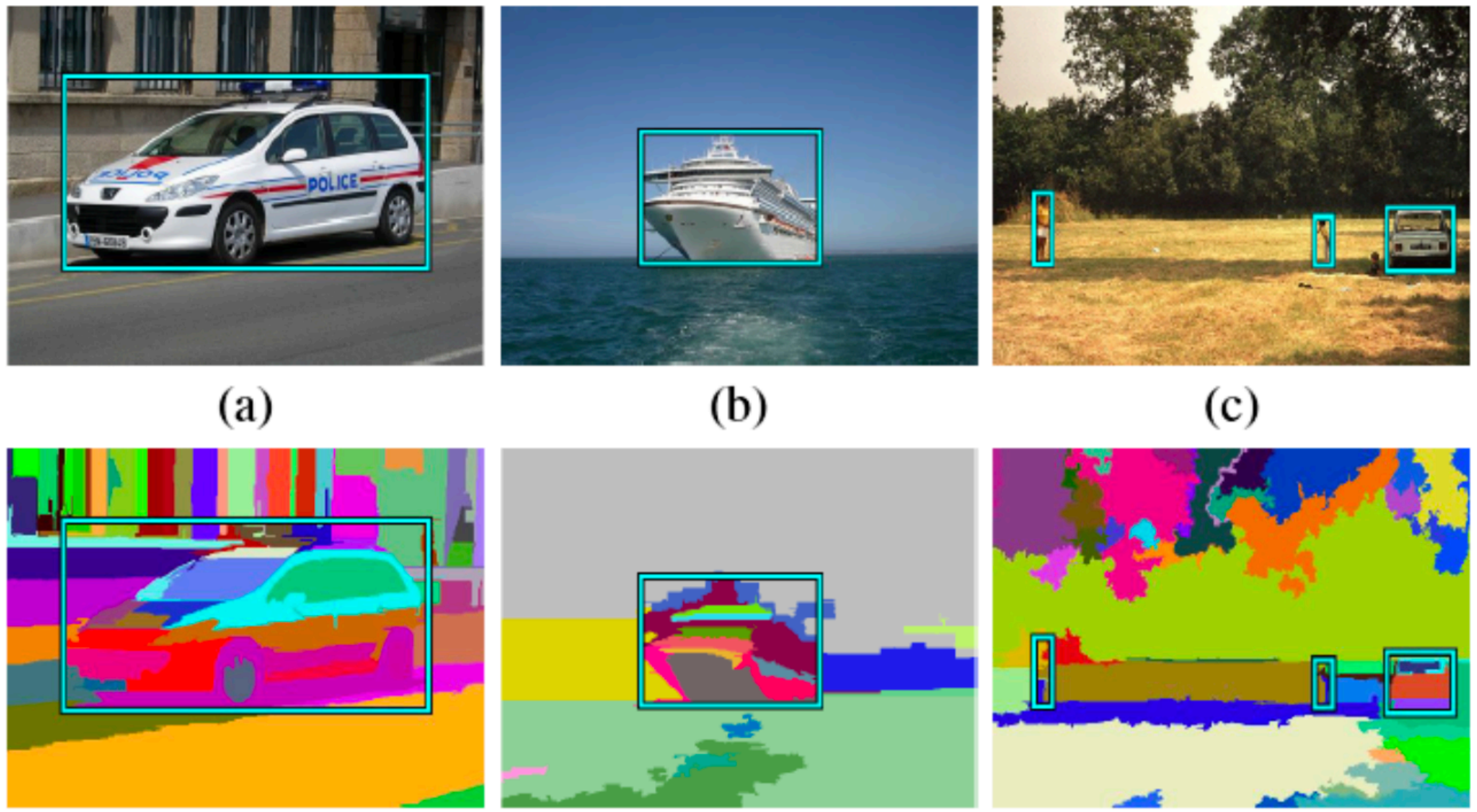
**Figure credit**: Alexe et al., 2012

# **Object** Proposals

**Superpixels** Straddling

— Favors regions with a well-defined closed boundary

— Measures the extent to which superpixels (obtained by image segmentation) contain pixels both inside and outside of the window




**Successful** Cases

**Failure** Case



**Figure credit**: Alexe et al., 2012

# **Object** Proposals

TABLE 2: For each detector [11, 18, 33] we report its performance (left column) and that of our algorithm 1 using the same window scoring function (right column). We show the average number of windows evaluated per image #win and the detection performance as the mean average precision (mAP) over all 20 classes.

| | [11] | OBJ- [11] | [18] | OBJ- [18] | ESS-BOW[33] | OBJ-BOW |
|---|---|---|---|---|---|---|
| mAP | 0.186 | 0.162 | 0.268 | 0.225 | 0.127 | 0.125 |
| #win | 79945 | 1349 | 18562 | 1358 | 183501 | 2997 |

**Table credit**: Alexe et al., 2012

Speeding up [11] HOG pedestrian detector [18] Deformable part model detector [33] Bag of words detector

# Summary

Detection scores in the deformable part model are based on both appearance and location

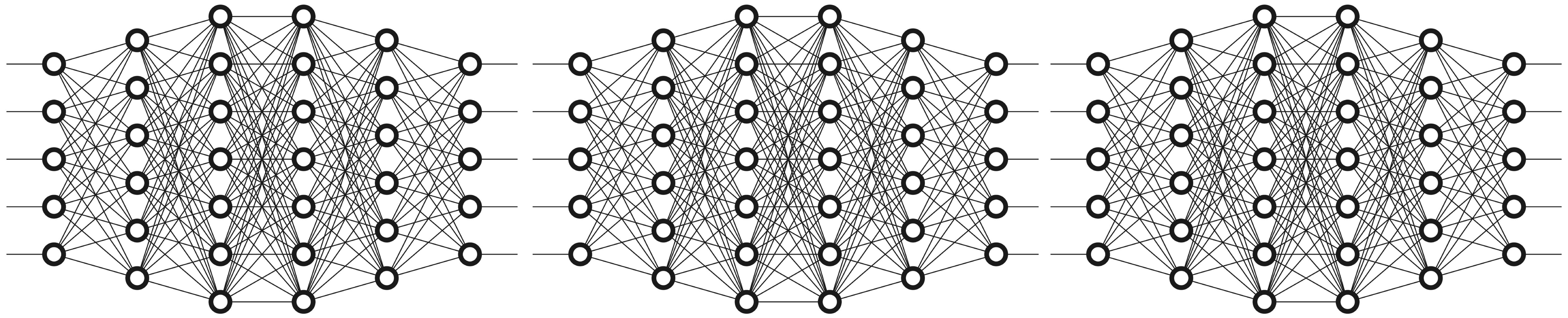
The deformable part model is trained iteratively by alternating the steps

1. Assume components and part locations given; compute appearance and offset models
2. Assume appearance and offset models given; compute components and part locations

An object **proposal** algorithm generates a short list of regions with generic object-like properties that can be evaluated by an object detector in place of an exhaustive sliding window search

THE UNIVERSITY OF BRITISH COLUMBIA

# CPSC 425: Computer Vision



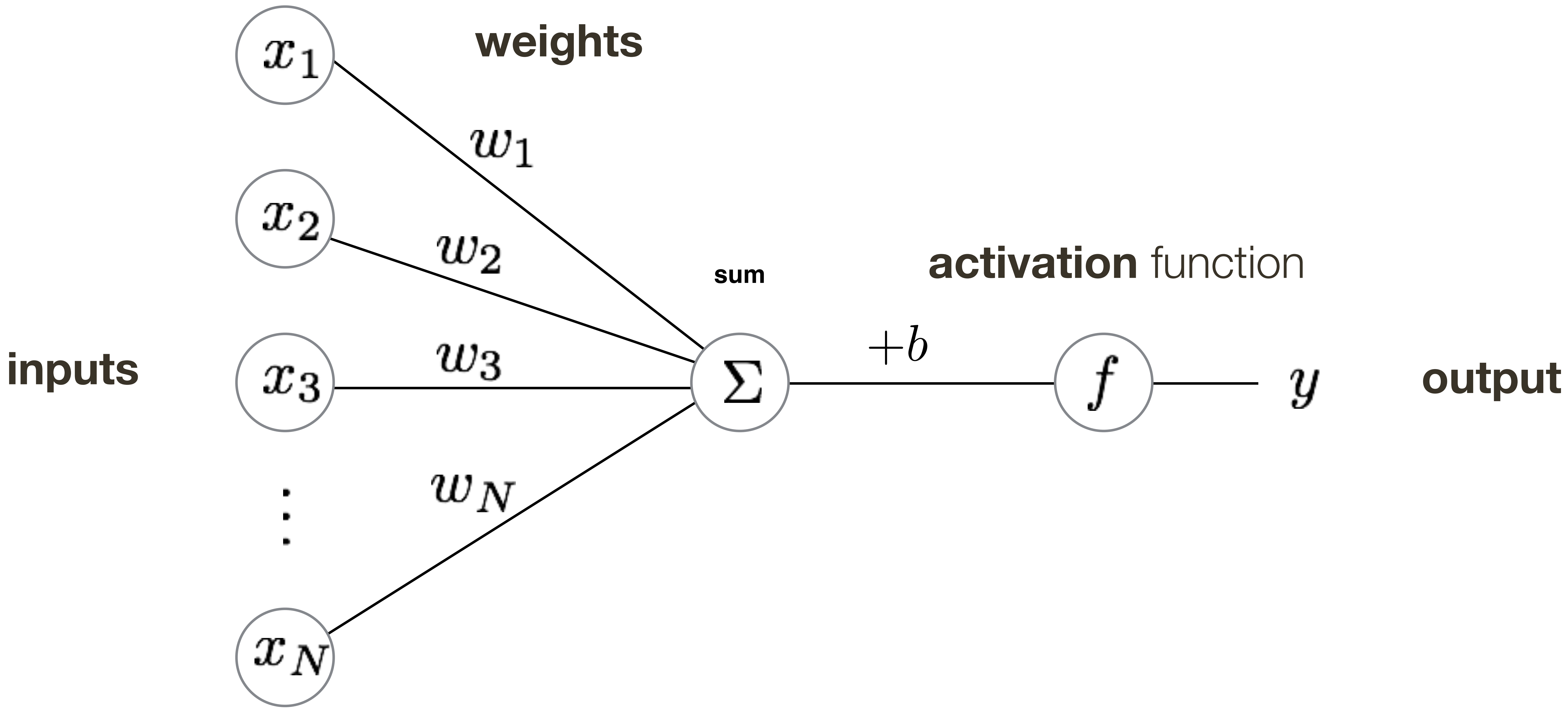
**Lecture 22:** Neural Networks

# **Warning**:

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

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

# A **Neuron**

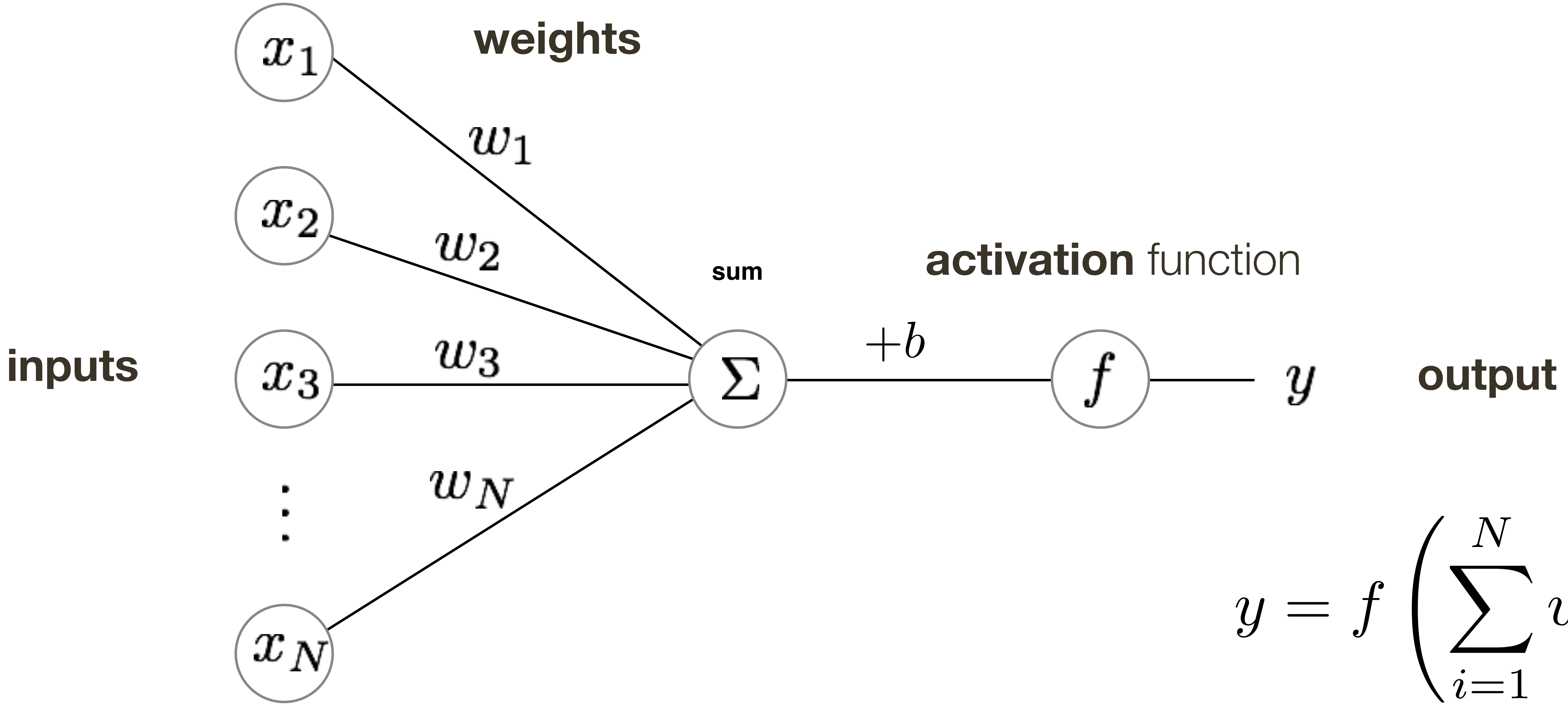



— The basic unit of computation in a neural network is a neuron.

— 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:

image features

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

weights (parameters)

bias vector (parameters)



**Image Credit**: Ioannis (Yannis) Gkioulekas (CMU)

# **Recall**: Linear Classifier

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

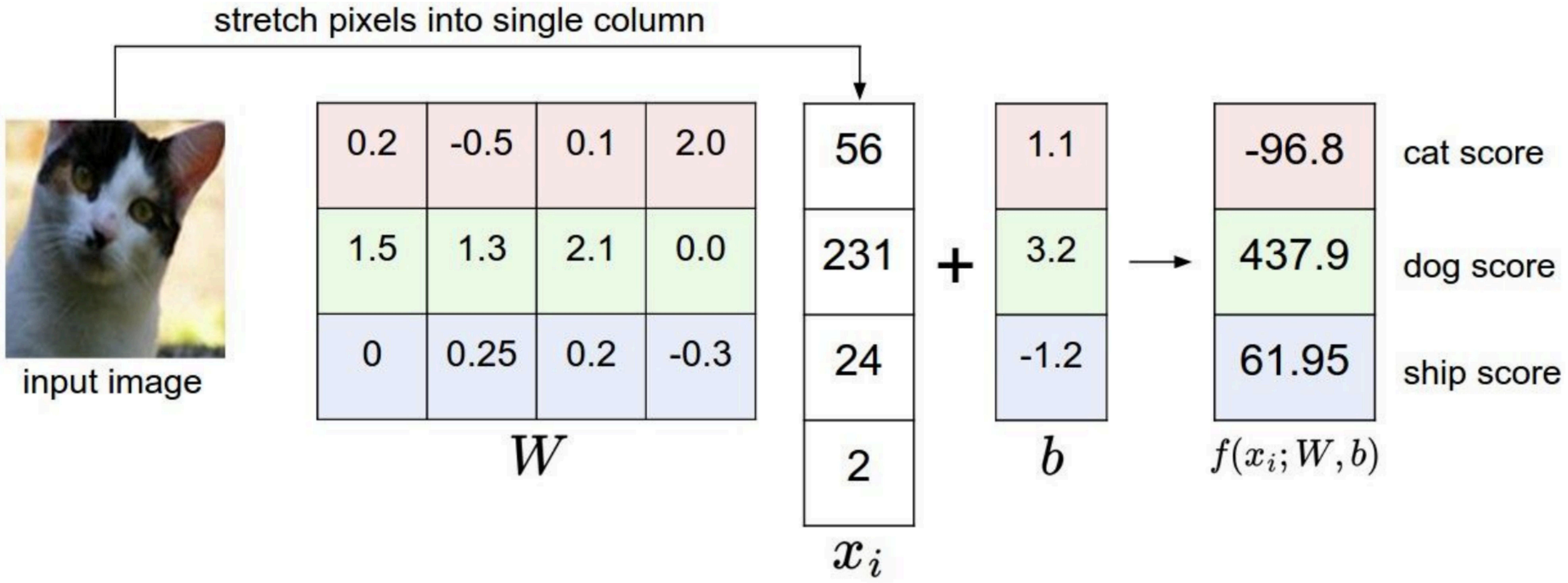
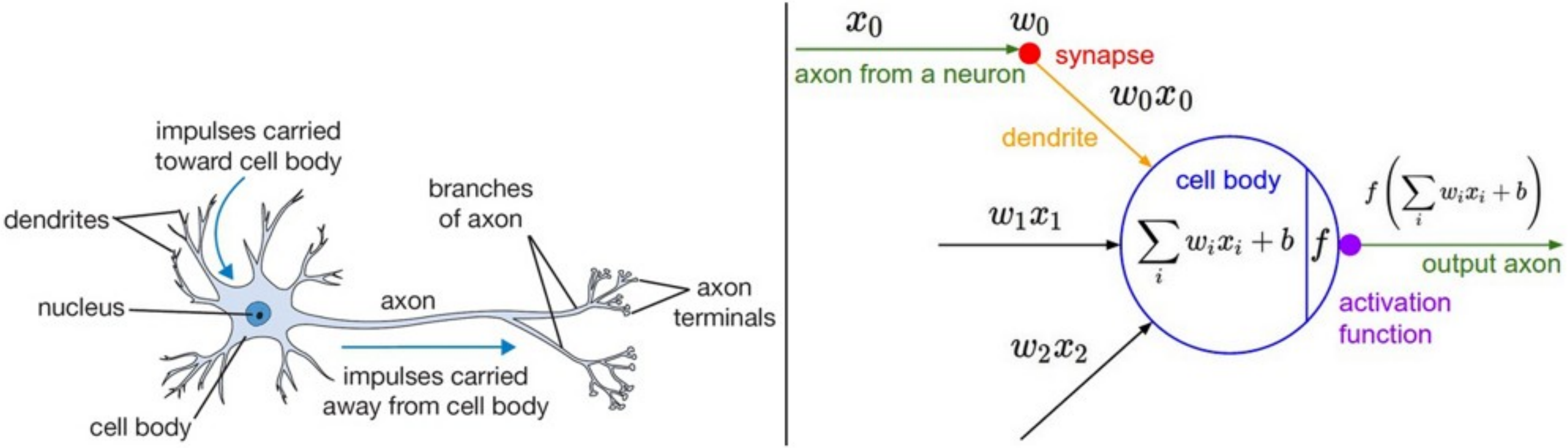





**Image Credit**: Ioannis (Yannis) Gkioulekas (CMU)

# **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**

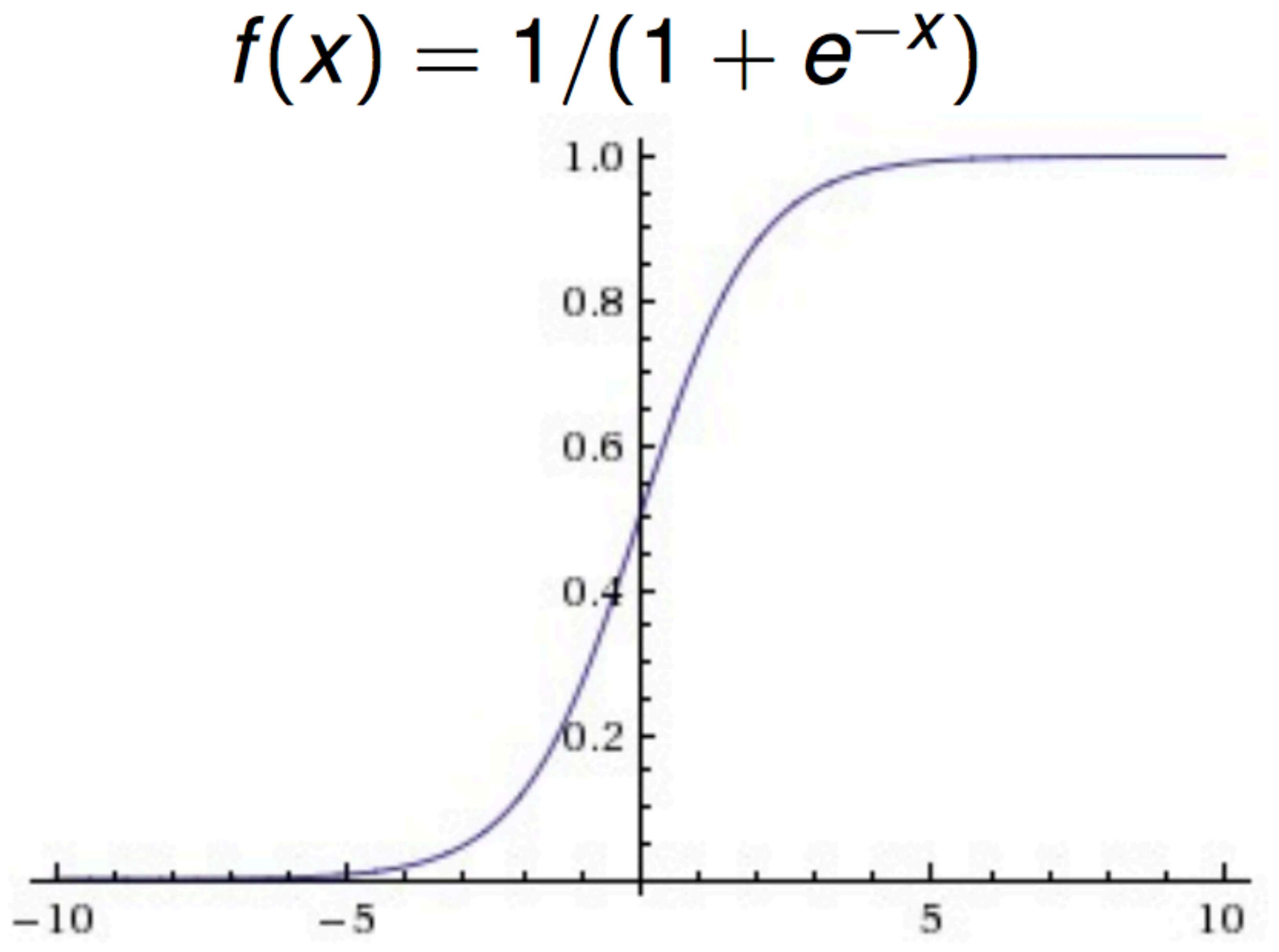



**Figure credit**: Fei-Fei and Karpathy

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)

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

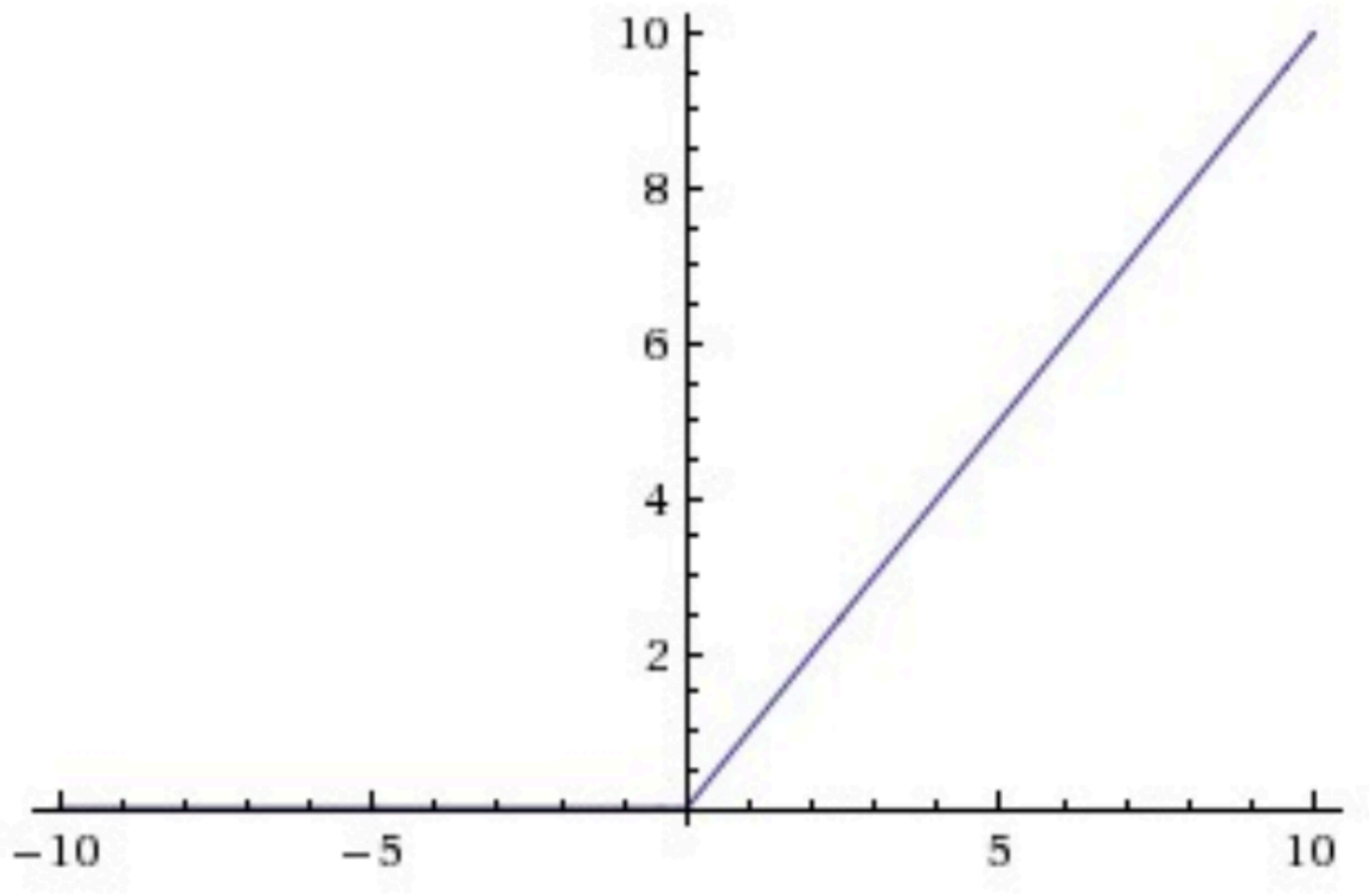
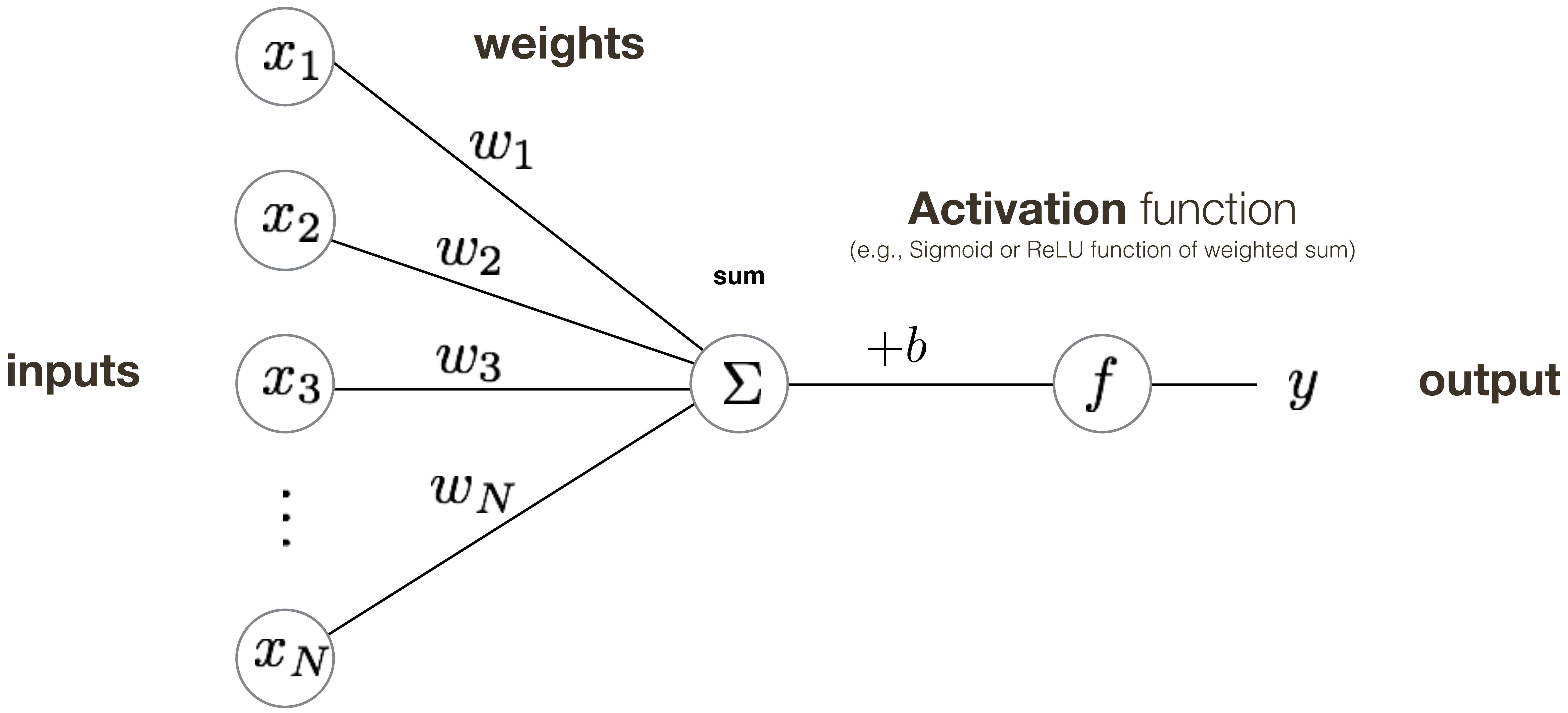



**Figure credit**: Fei-Fei and Karpathy

Found to accelerate convergence during learning

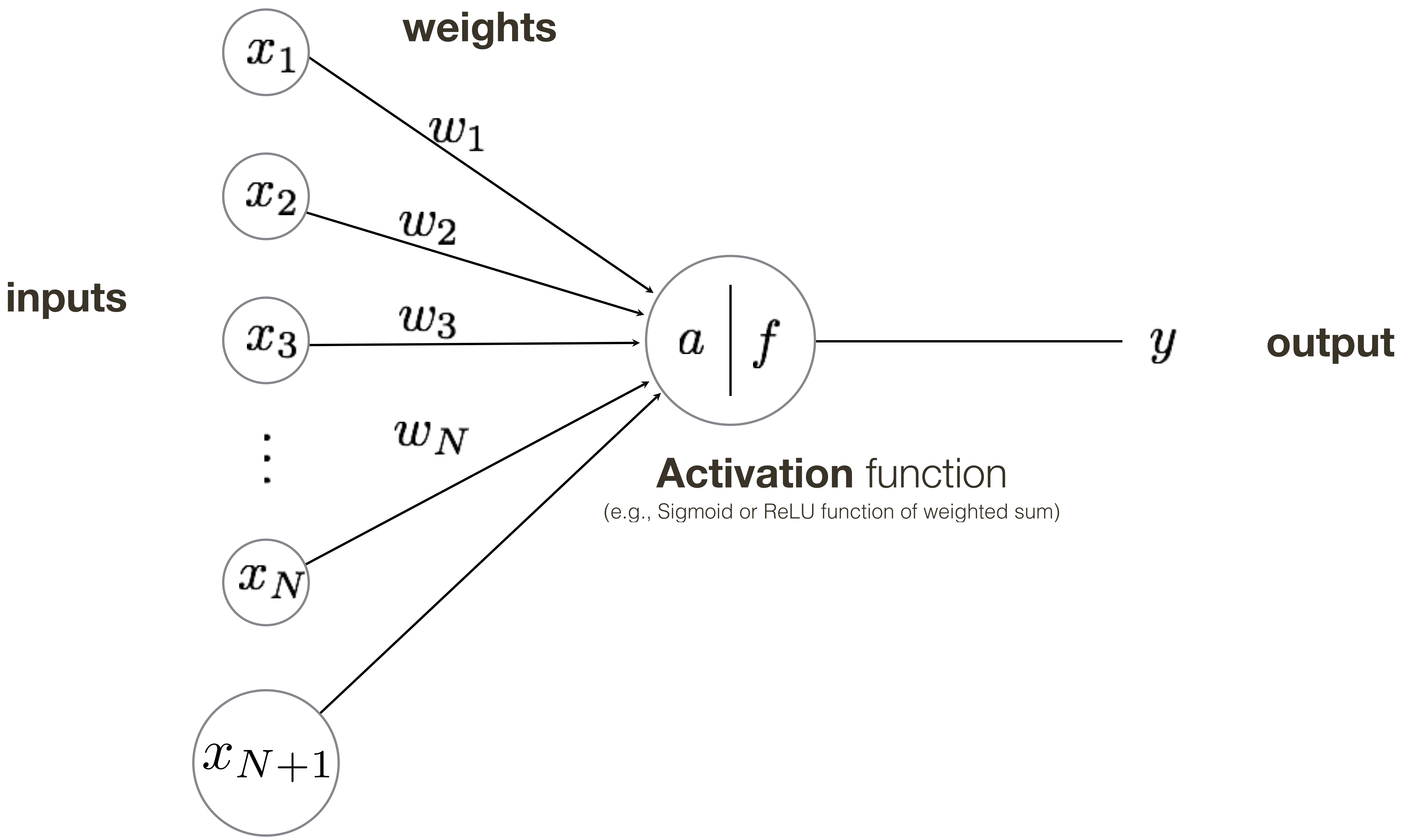
Used in the most recent neural networks

# A **Neuron**



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

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

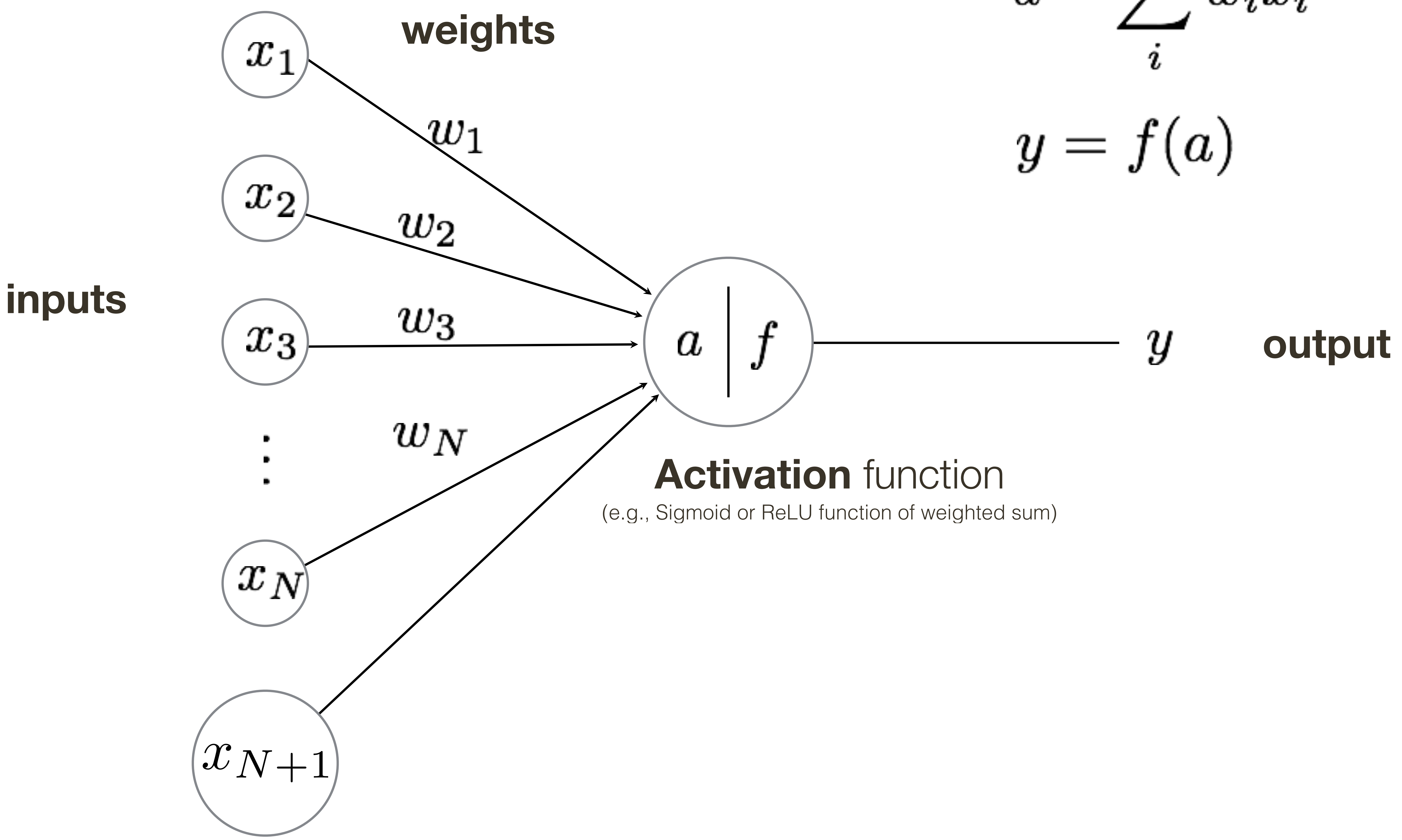


# A **Neuron** … another way to draw it …

(1) Combine the sum and activation function

$$a = \sum_i w_i x_i$$

$$y = f(a)$$




**Activation** function

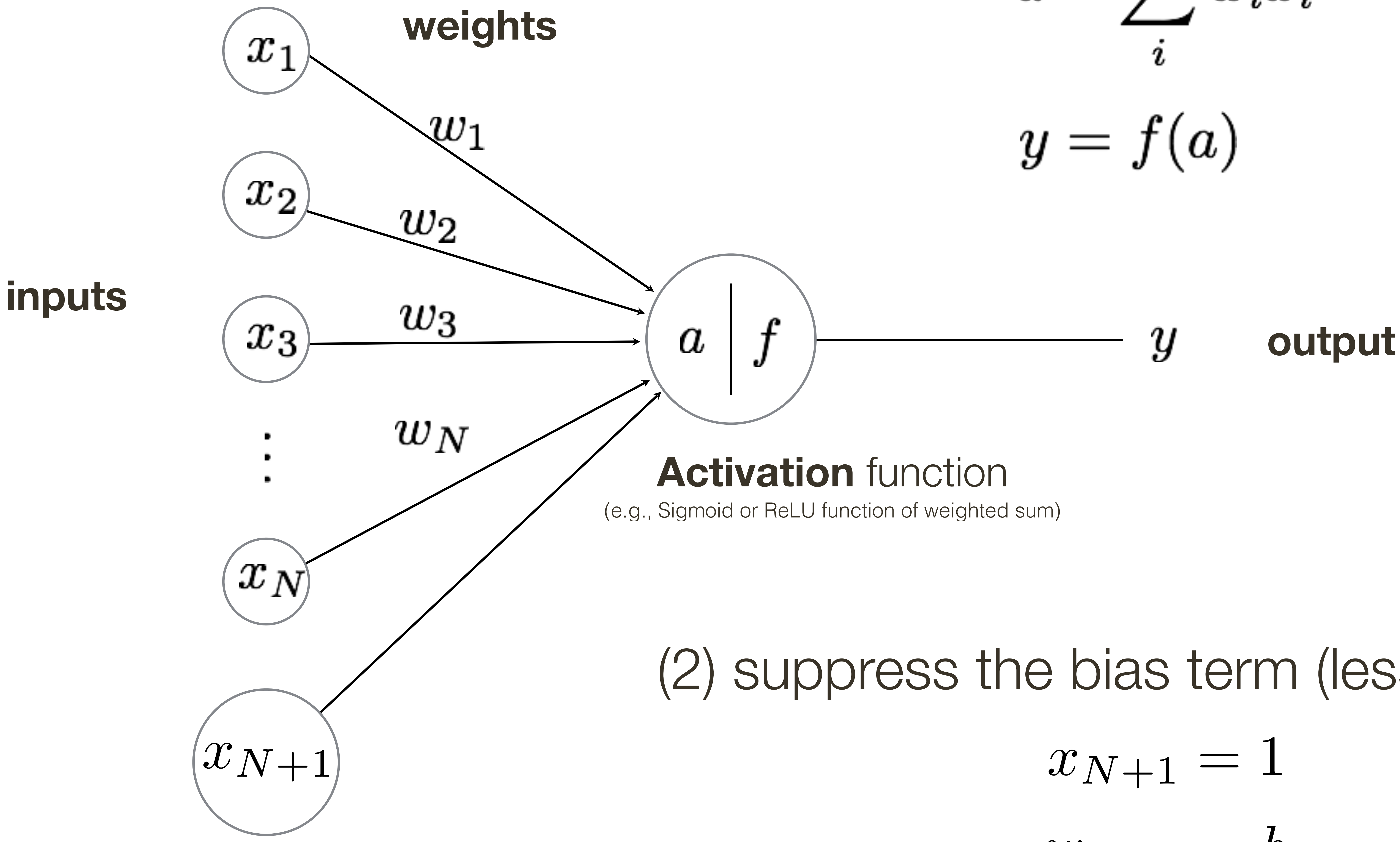
(e.g., Sigmoid or ReLU function of weighted sum)

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

(1) Combine the sum and activation function

$$a = \sum_i w_i x_i$$

$$y = f(a)$$




**Activation** function

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

(2) suppress the bias term (less clutter)

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

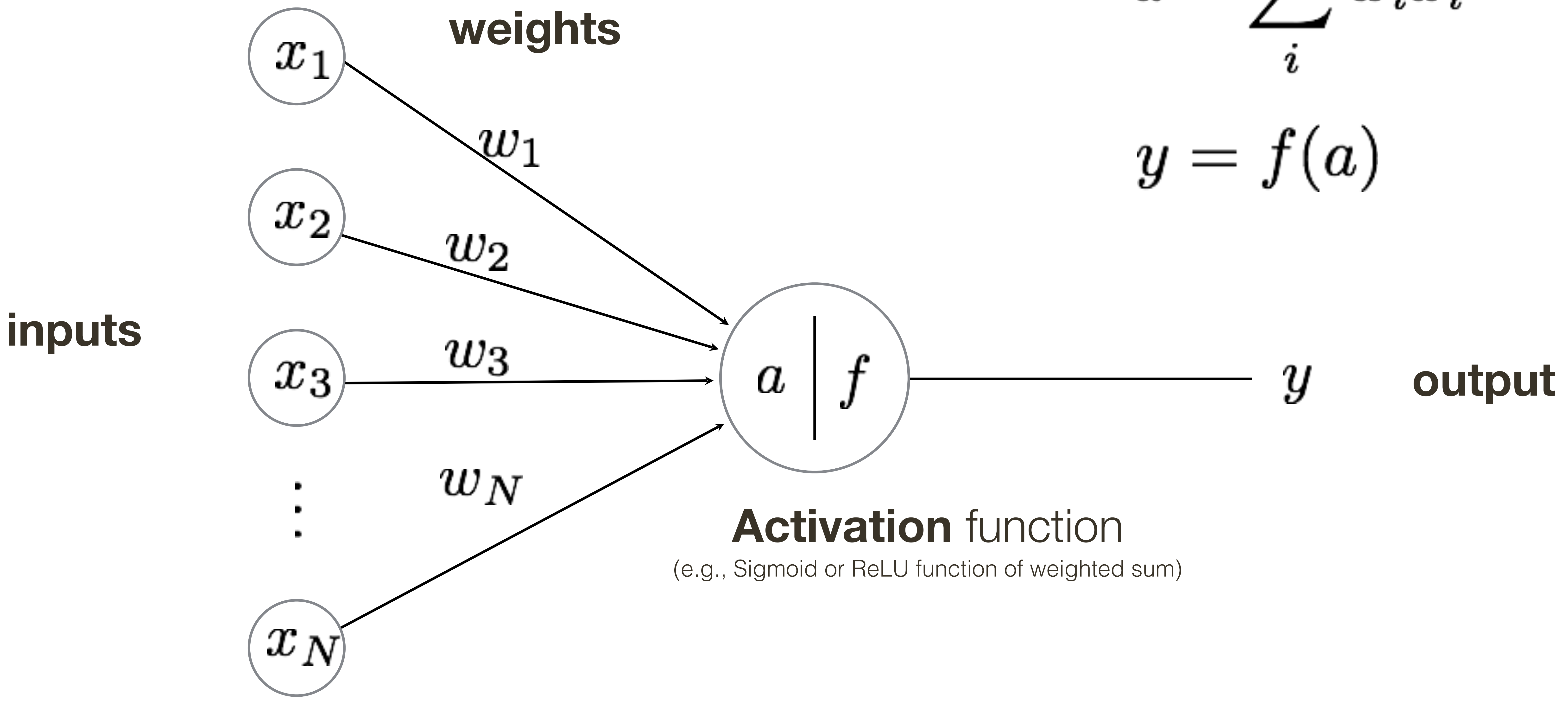
$$w_{N+1} = b$$

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

(1) Combine the sum and activation function

$$a = \sum_i w_i x_i$$

$$y = f(a)$$



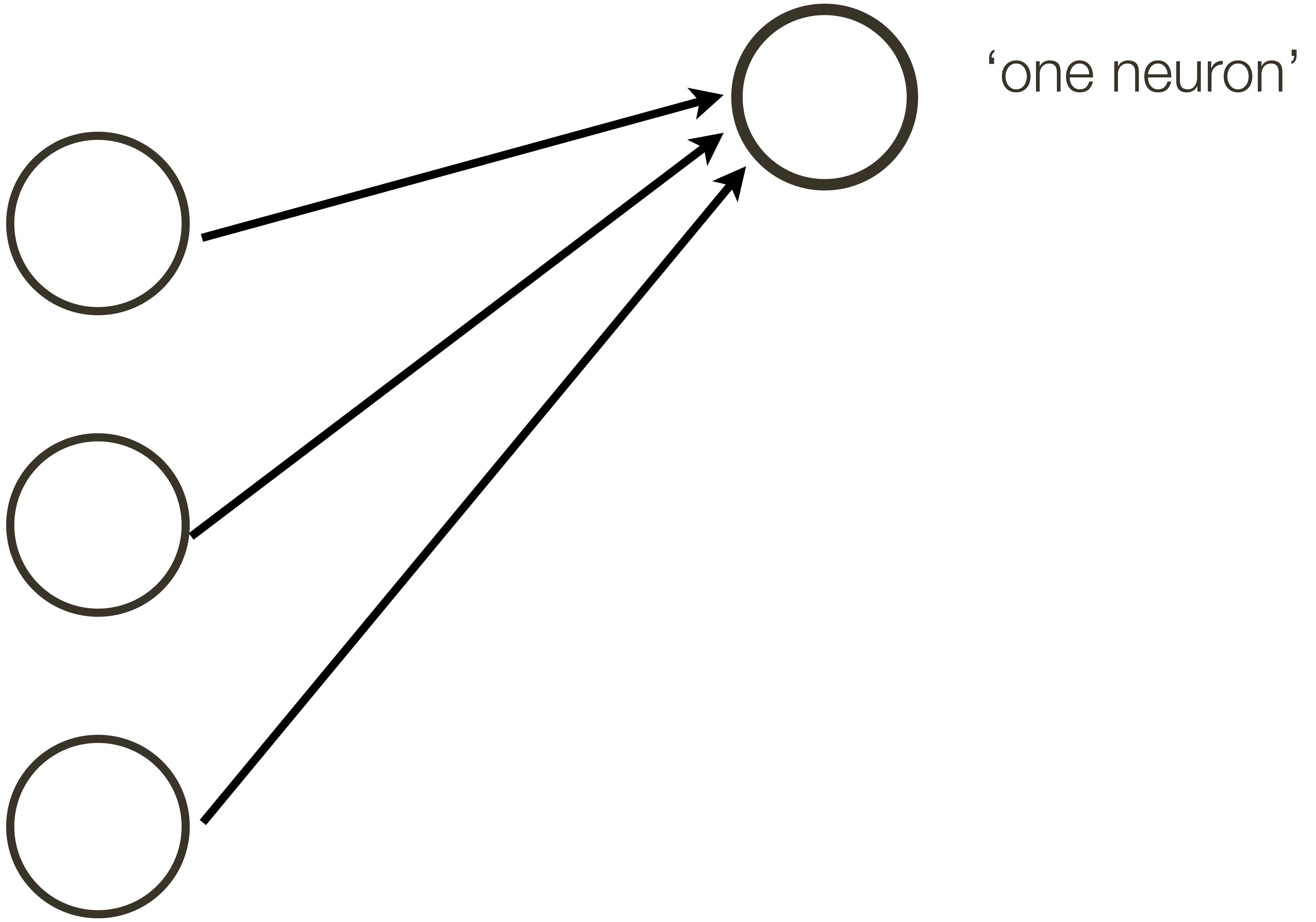

(2) suppress the bias term (less clutter)

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

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

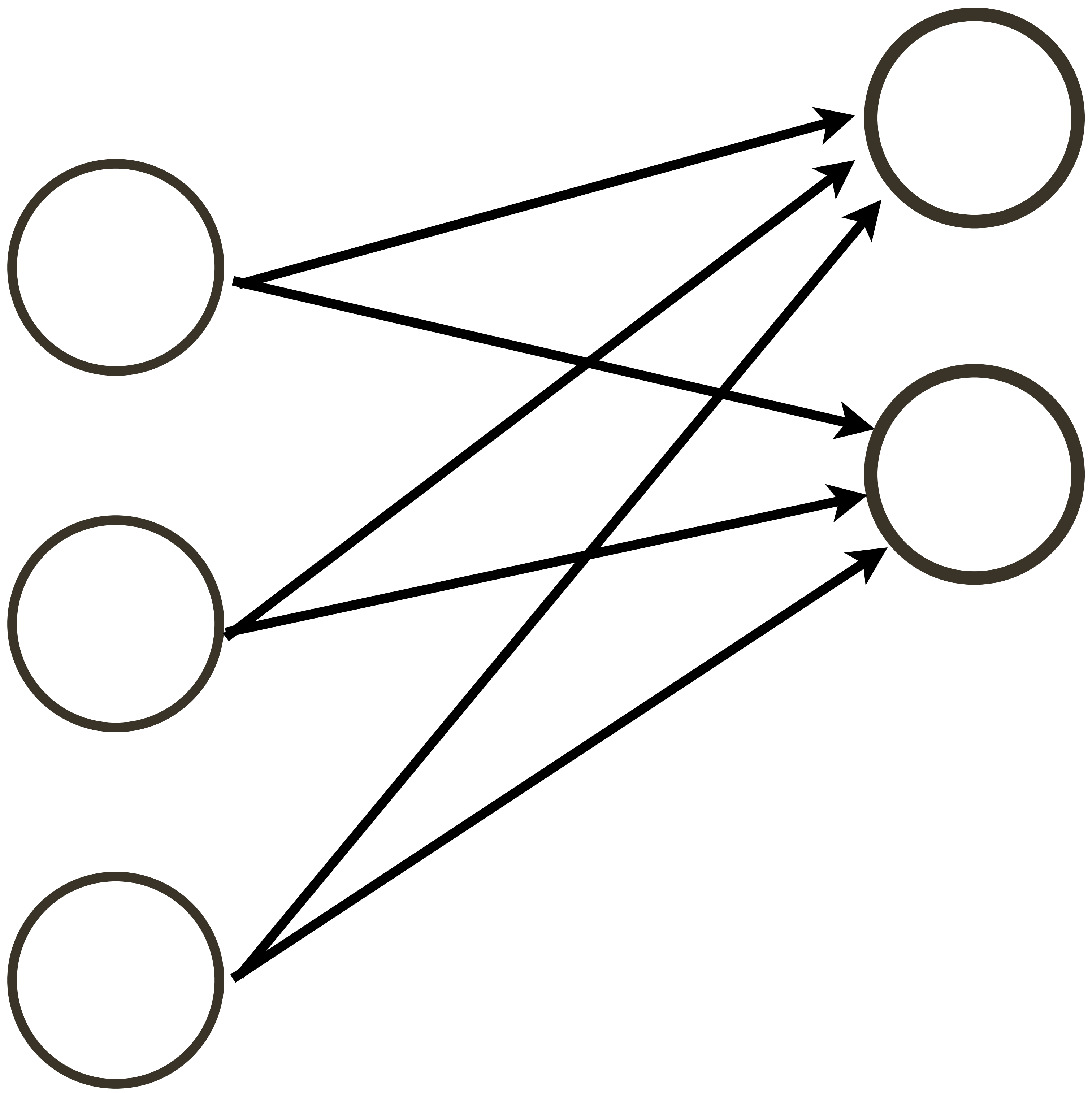
# **Neural** Network

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



# **Neural** Network

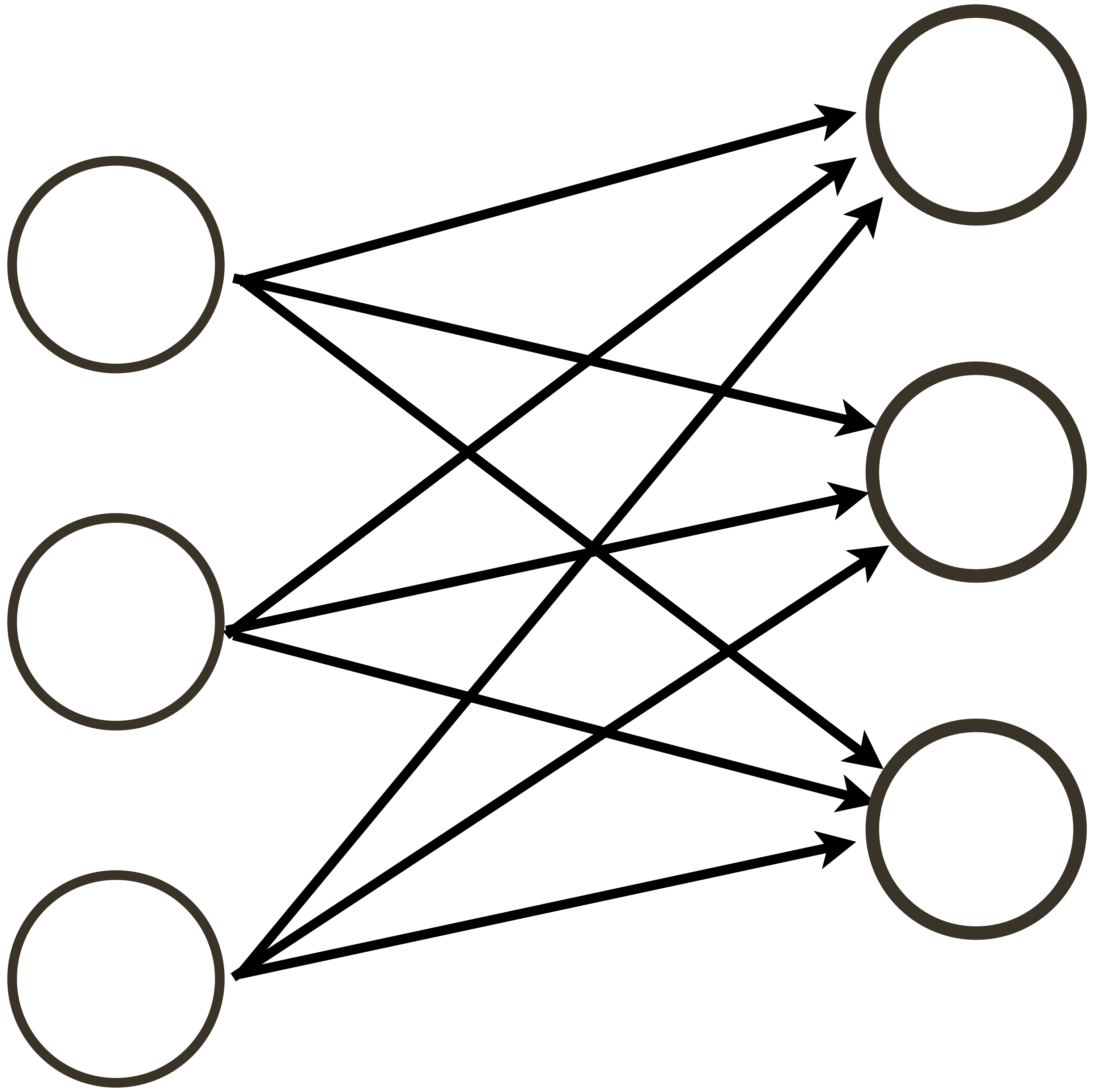
Connect a bunch of neurons together — a collection of connected neurons



‘two neurons’

# **Neural** Network

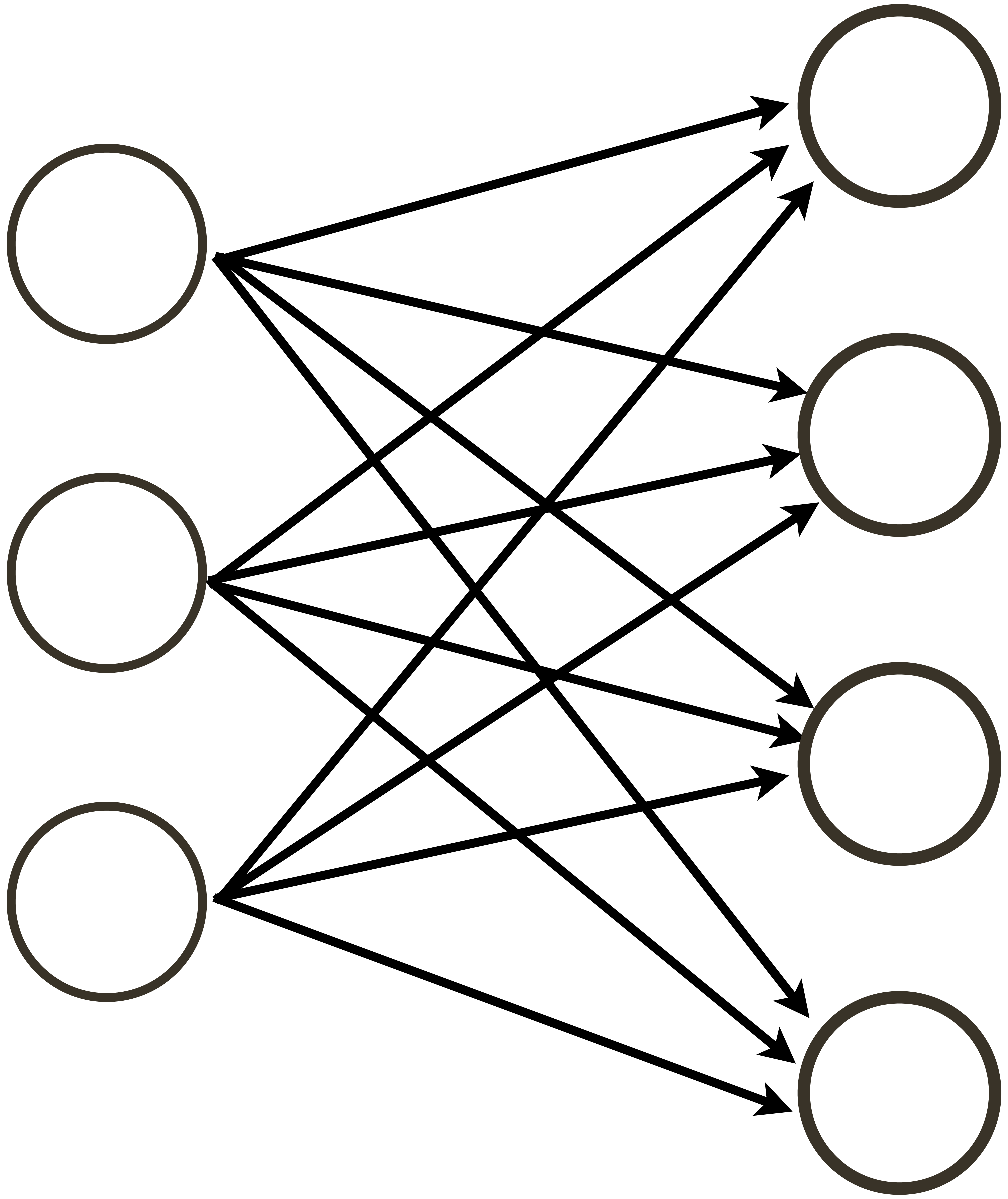
Connect a bunch of neurons together — a collection of connected neurons



‘three neurons’

# **Neural** Network

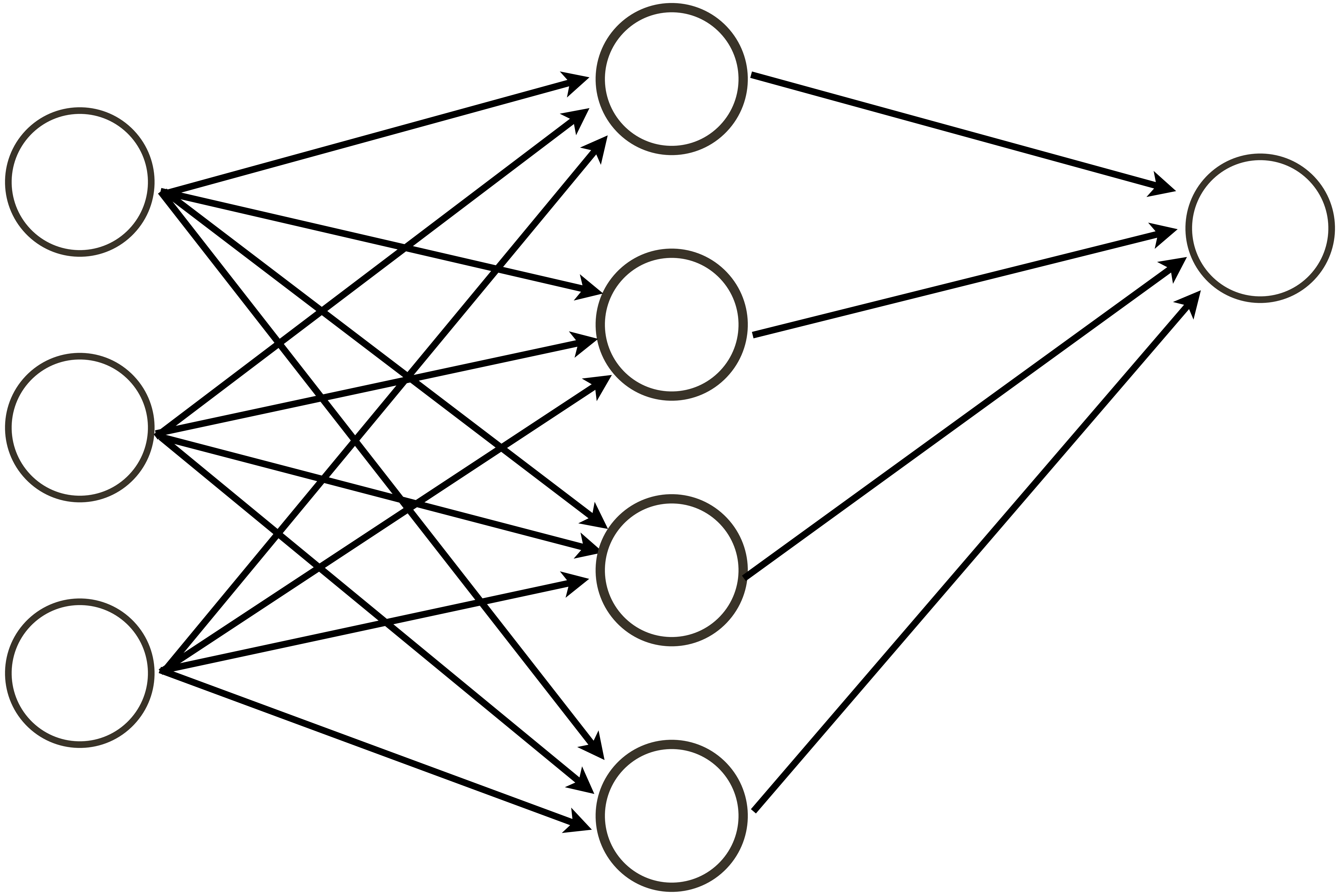
Connect a bunch of neurons together — a collection of connected neurons



‘four neurons’

# **Neural** Network

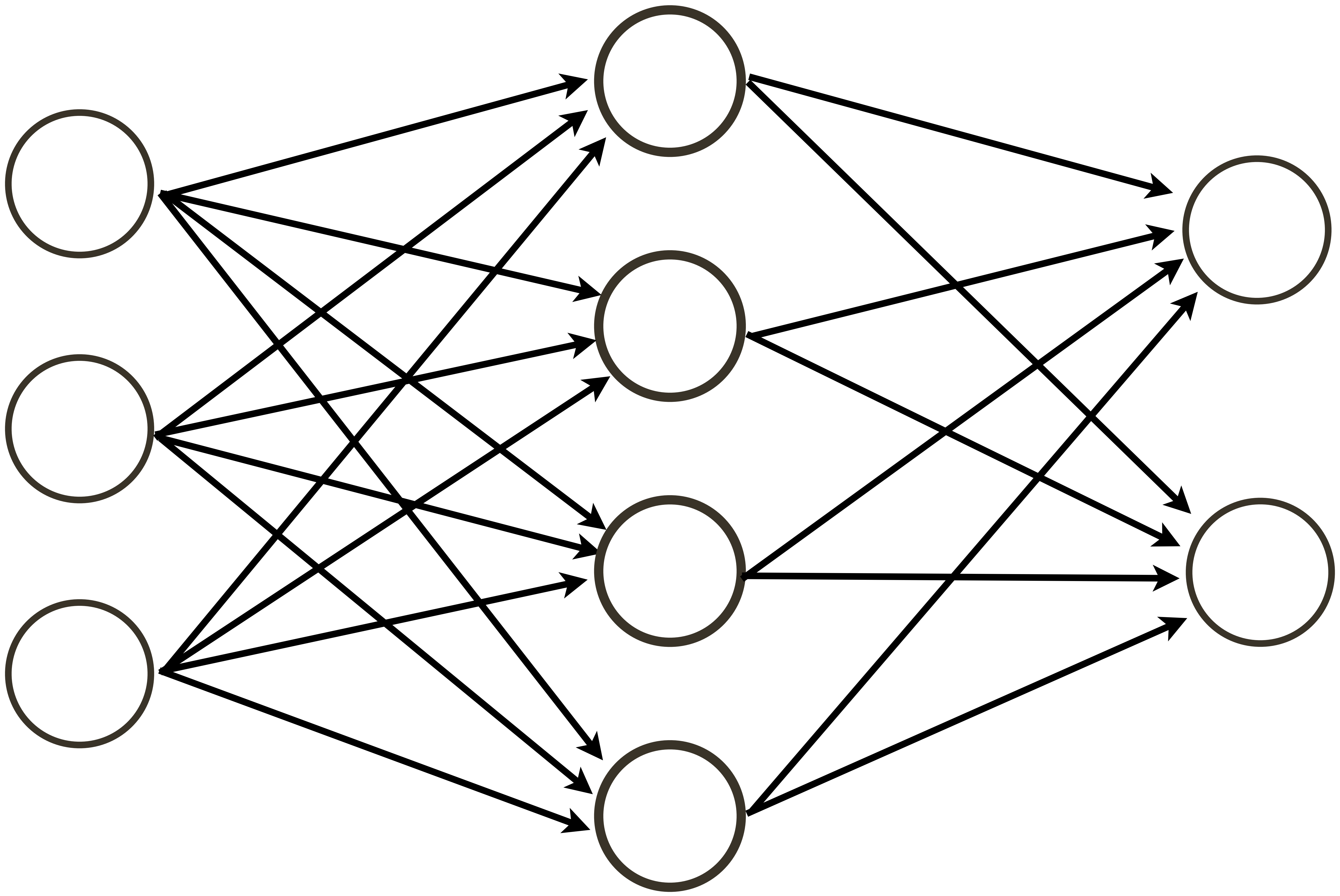
Connect a bunch of neurons together — a collection of connected neurons



'five neurons'

# **Neural** Network

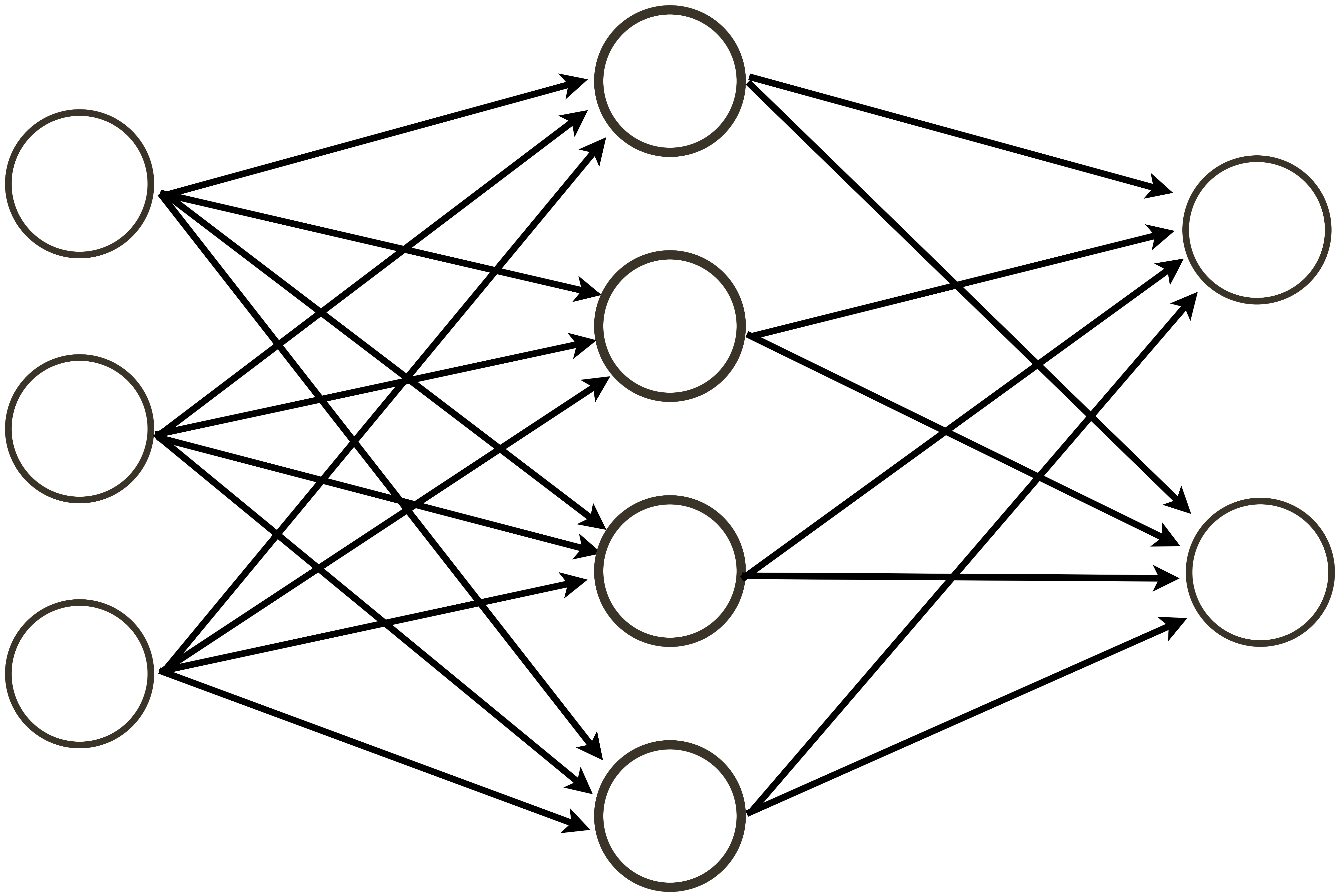
Connect a bunch of neurons together — a collection of connected neurons



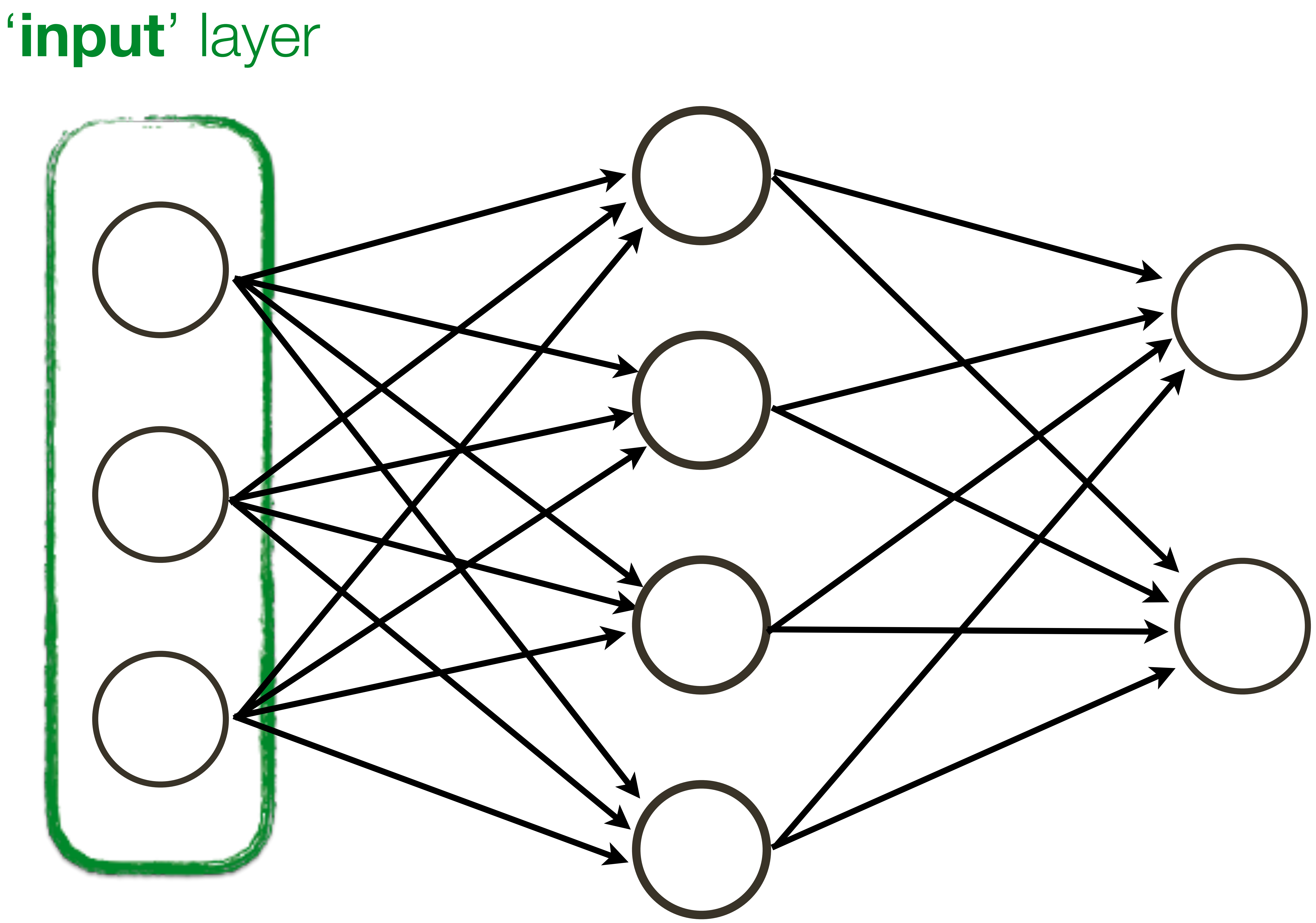
‘six neurons’

# **Neural** Network

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



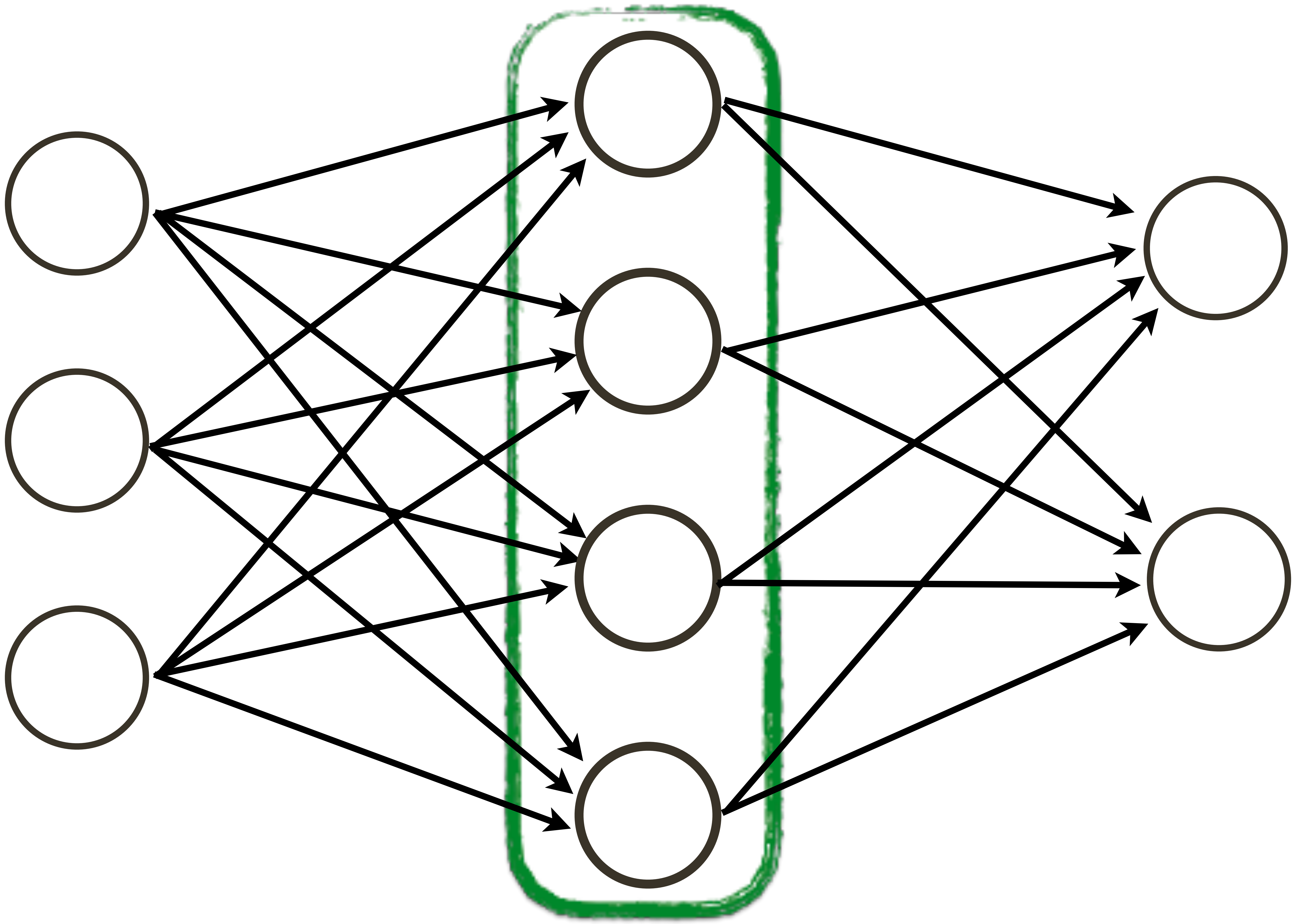
# Neural Network: **Terminology**



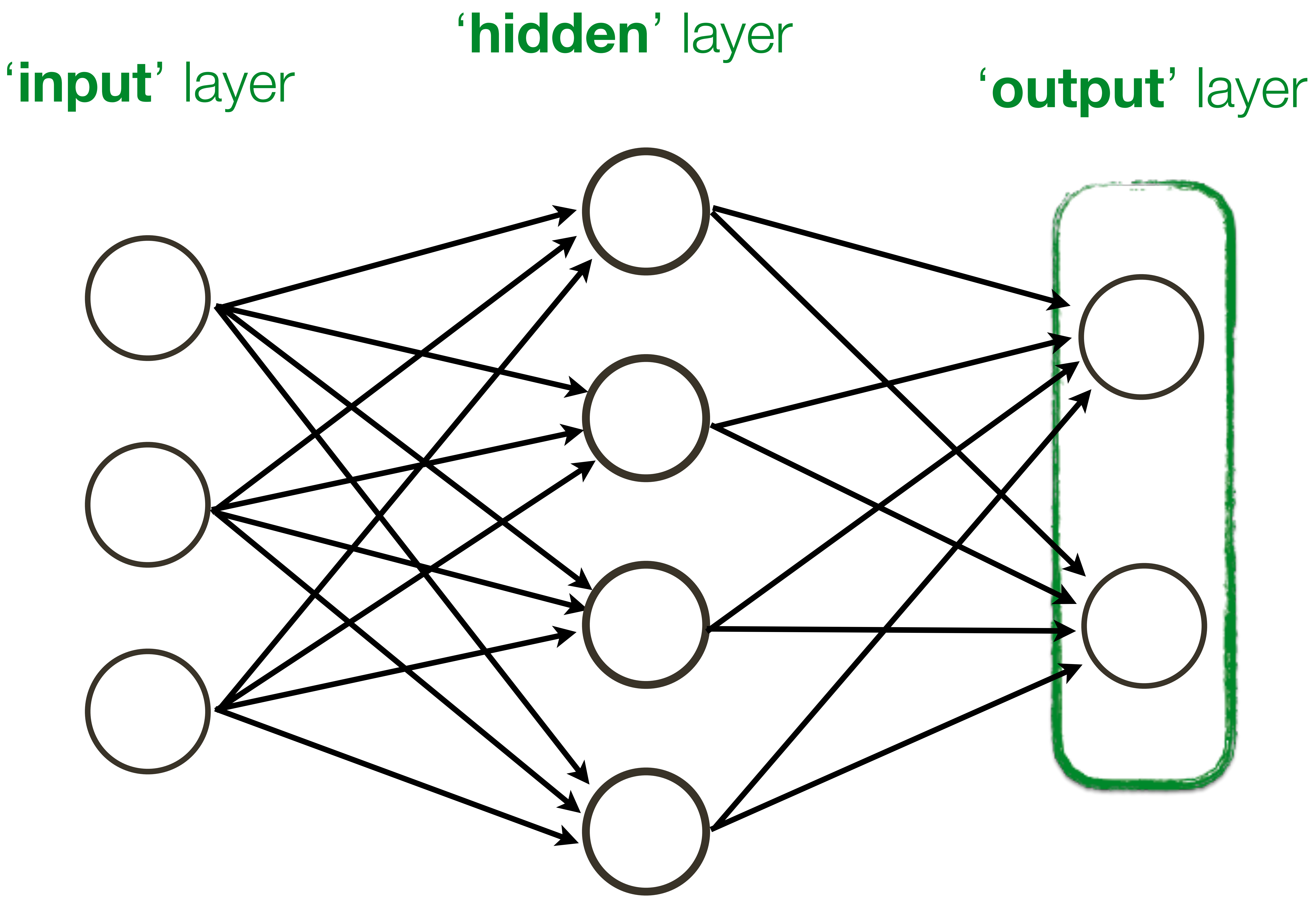
# Neural Network: **Terminology**

‘**hidden**’ layer

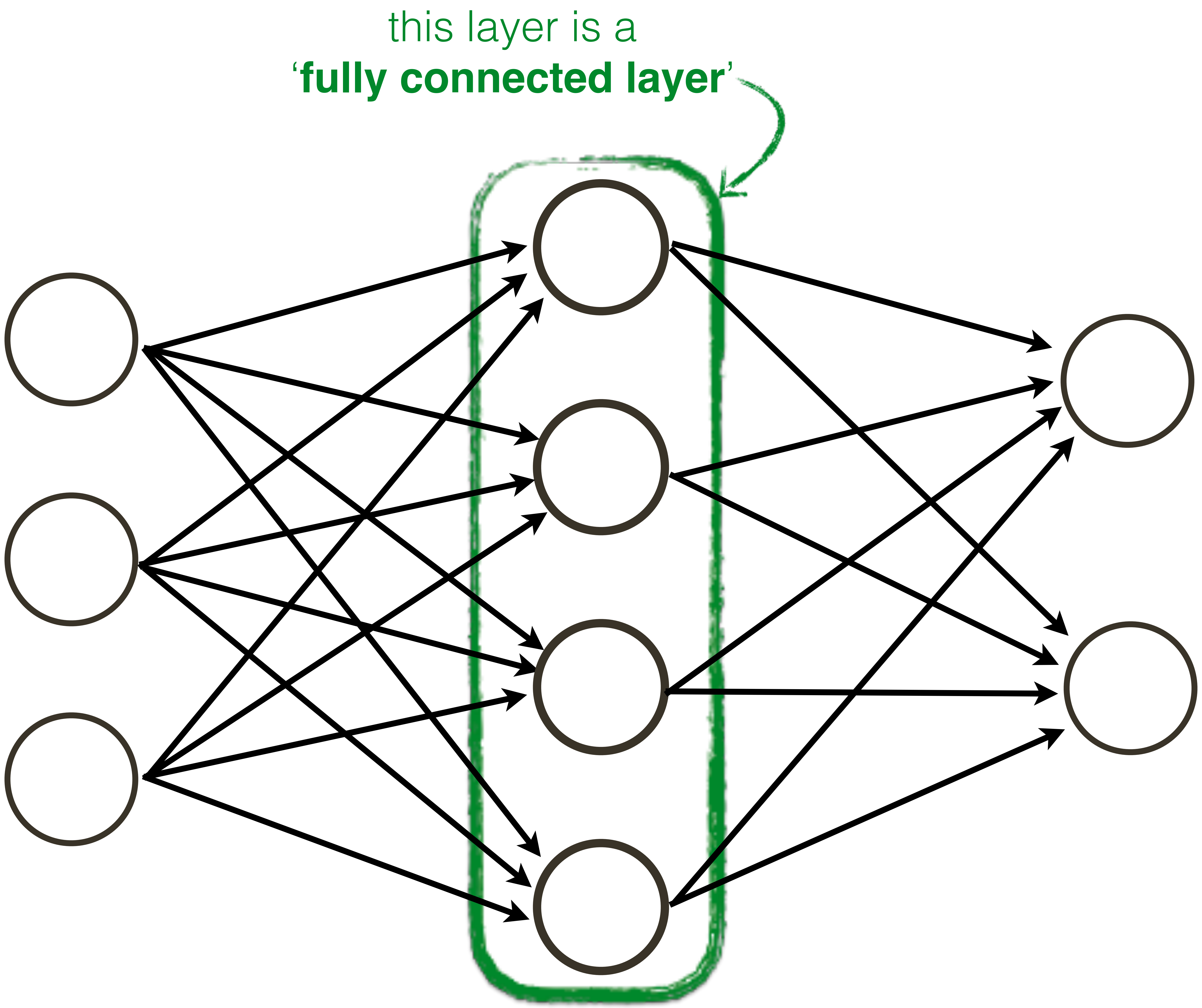
‘**input**’ layer



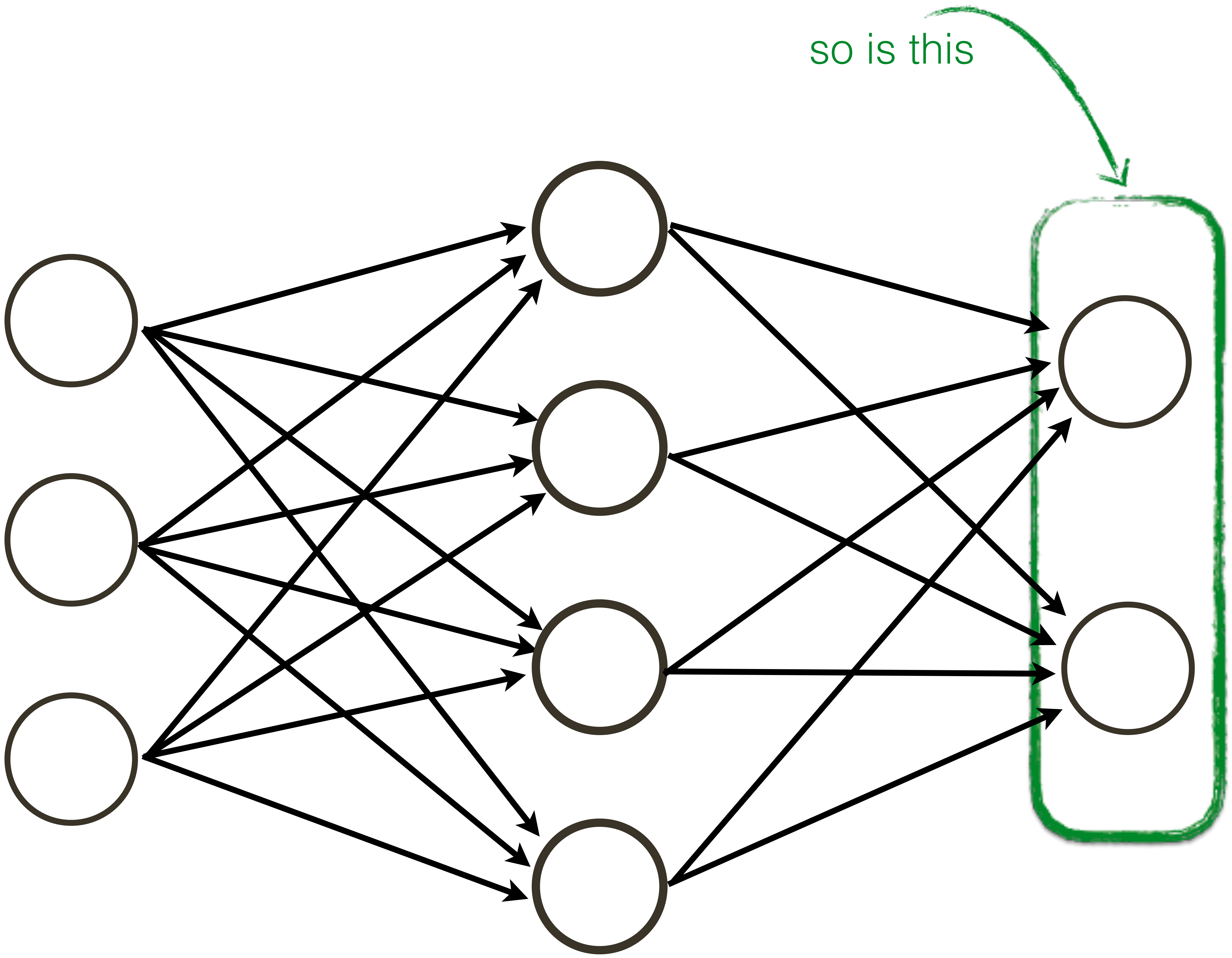
# Neural Network: **Terminology**



# 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

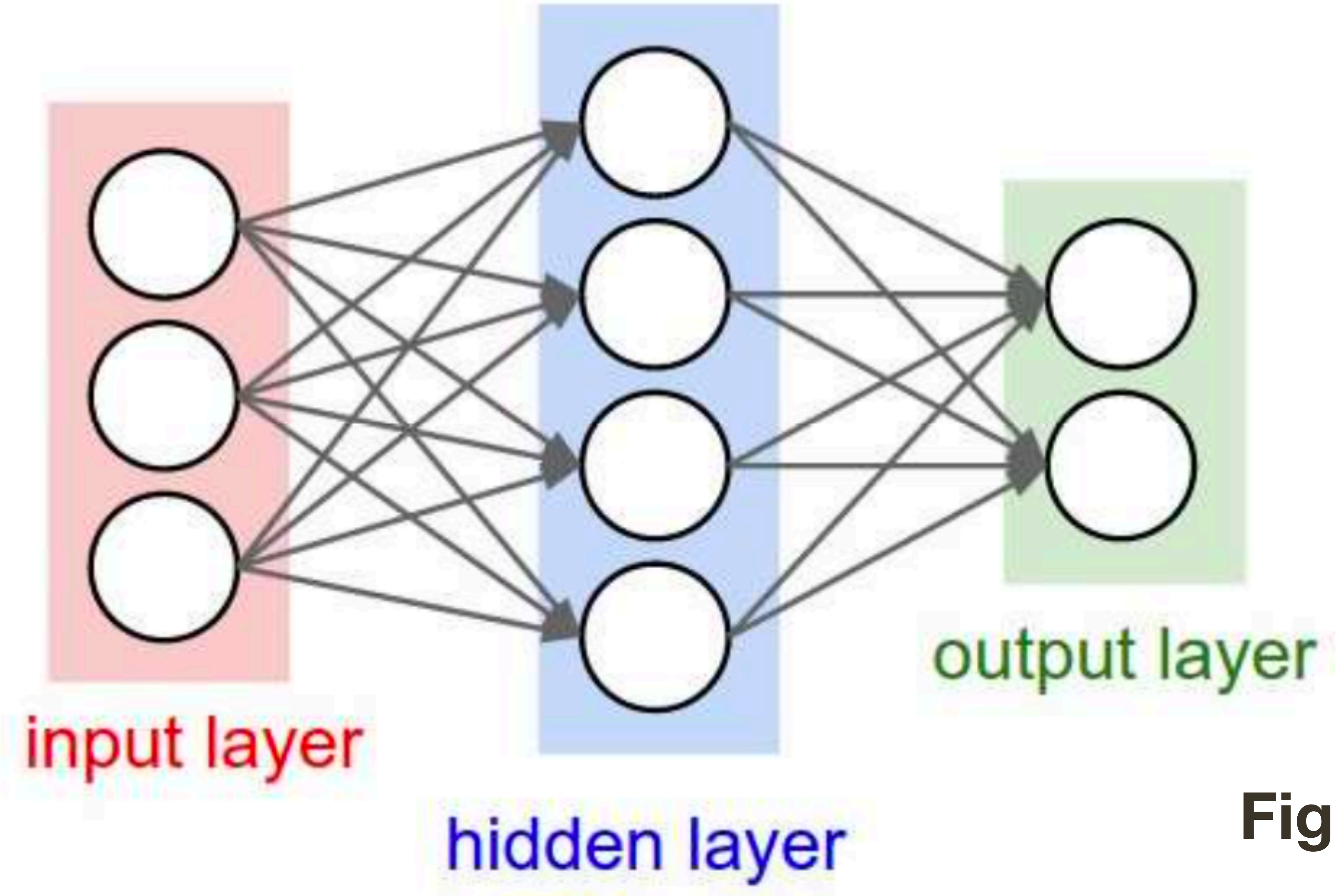



**Figure credit**: Fei-Fei and Karpathy

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

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

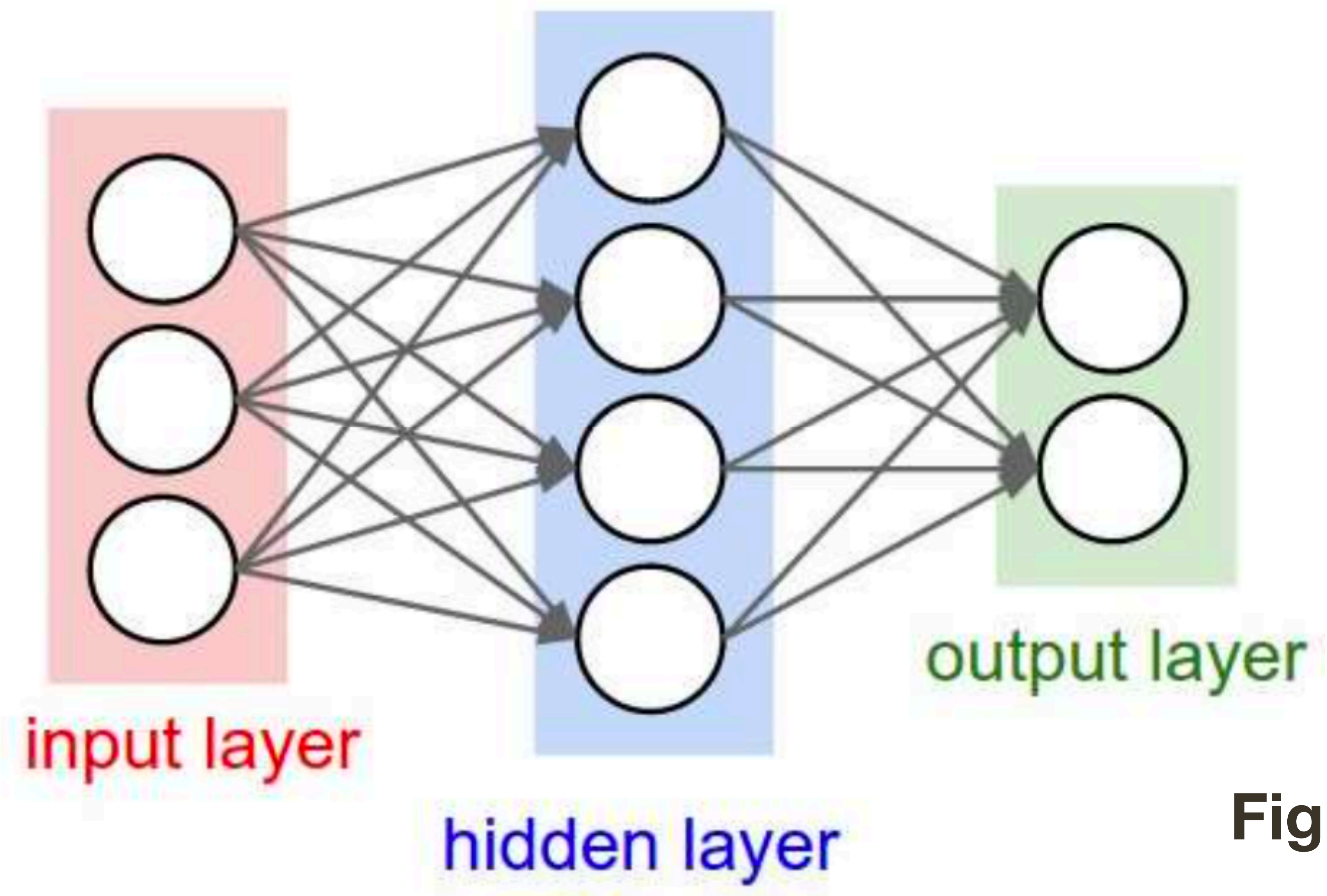



**Figure credit**: Fei-Fei and Karpathy

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)

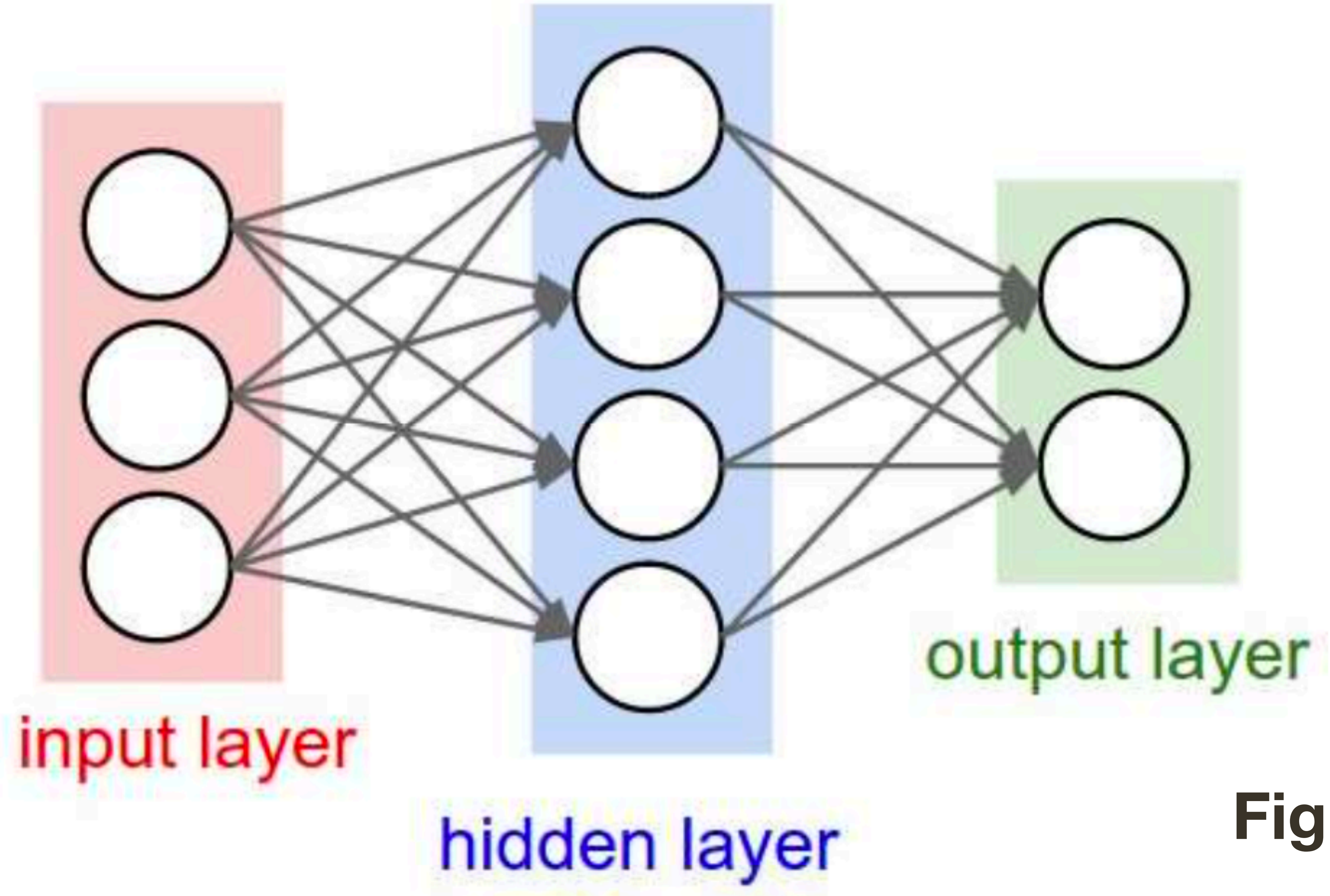



**Figure credit**: Fei-Fei and Karpathy

# **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)

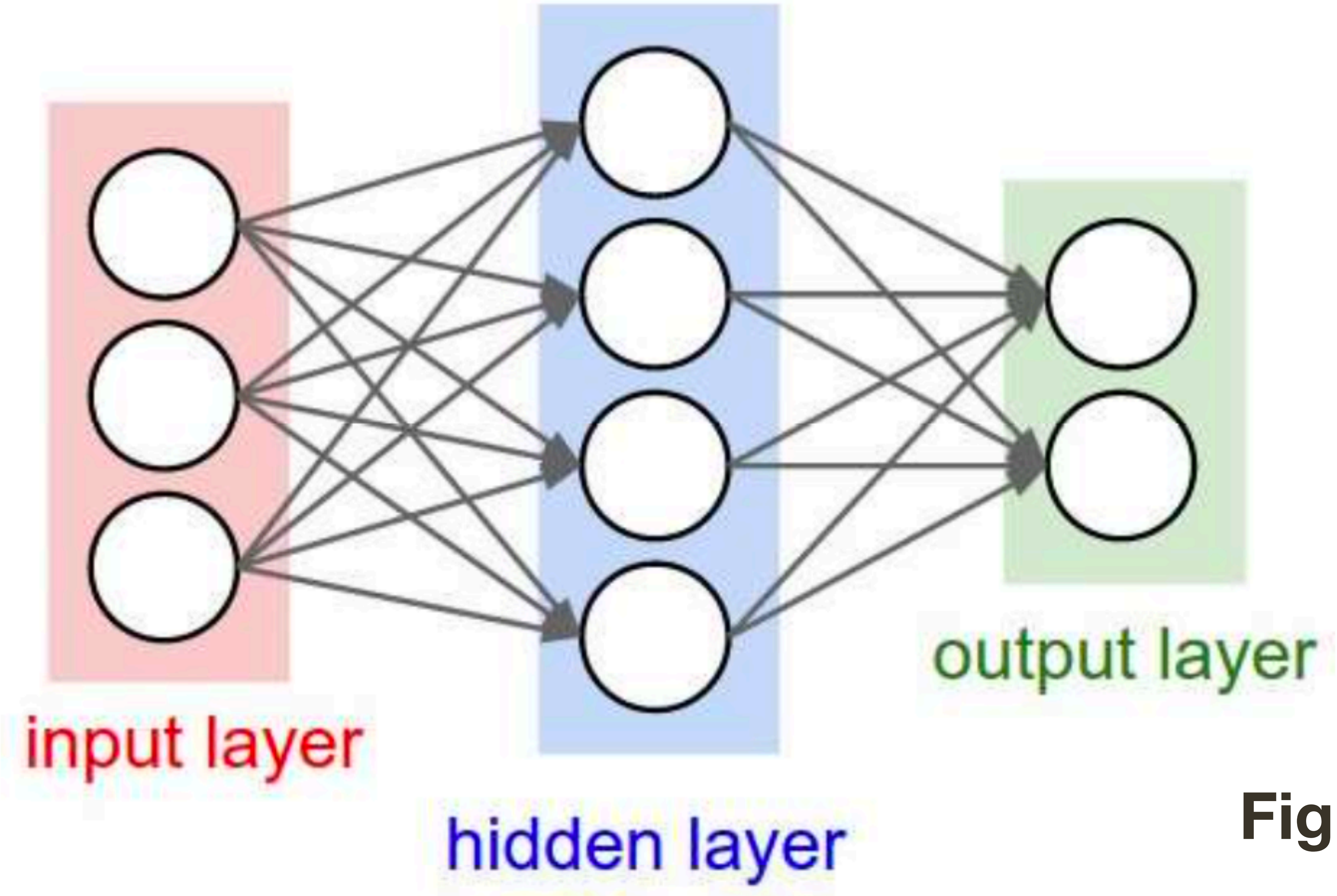



**Figure credit**: Fei-Fei and Karpathy

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

# 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?

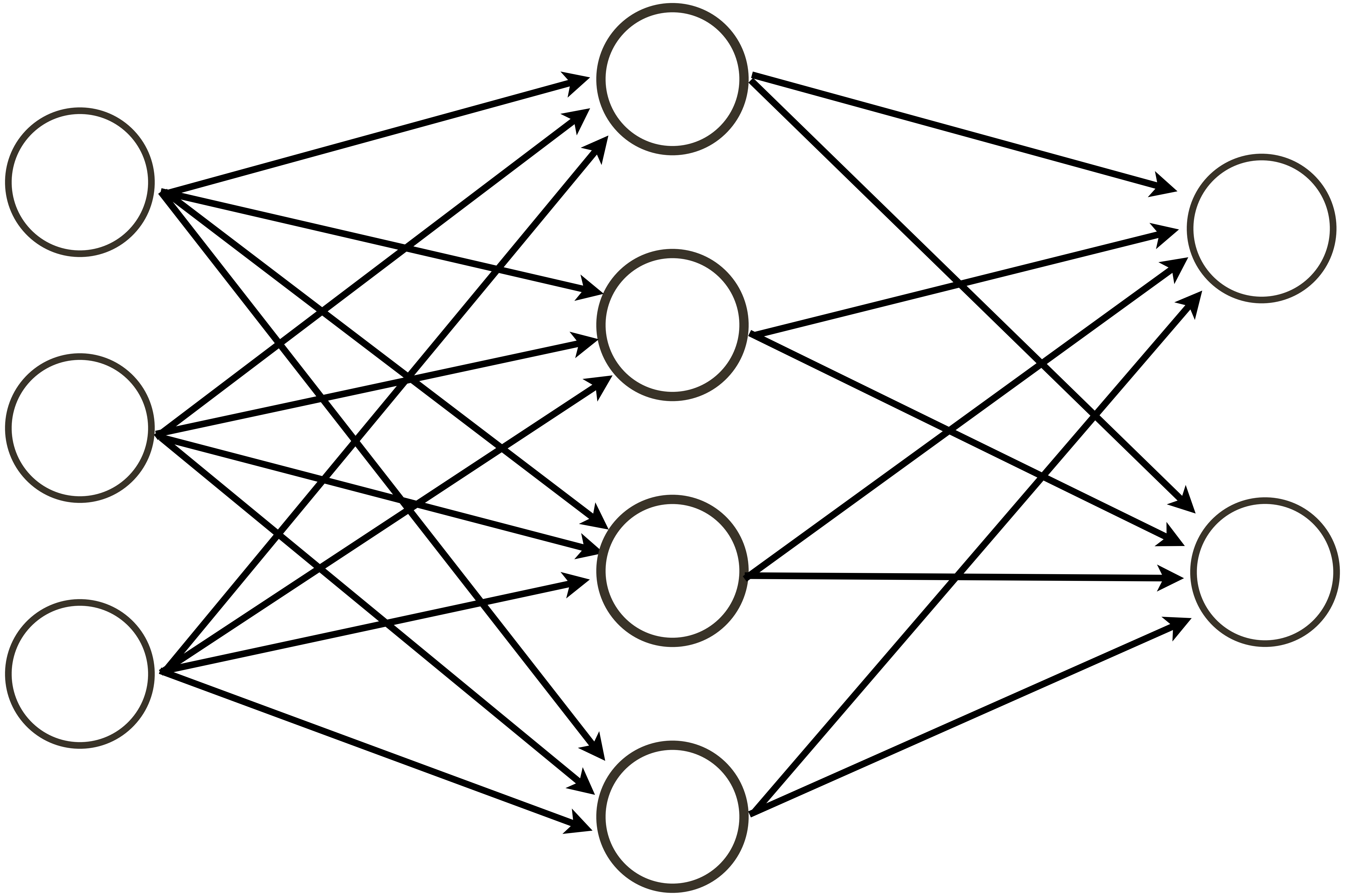
**Answer:** 1) More layers = more complex functional mapping

2) More efficient due to distributed representation

* slide from Marc'Aurelio Renzato

# **Activation** Function

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



# **Activation** Function

$$\hat{\mathbf{y}} = f(\mathbf{x}, \mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2) = \sigma\left(\mathbf{W}_2^{(2\times 4)}\sigma\left(\mathbf{W}_1^{(4\times 3)}\mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)}\right)$$

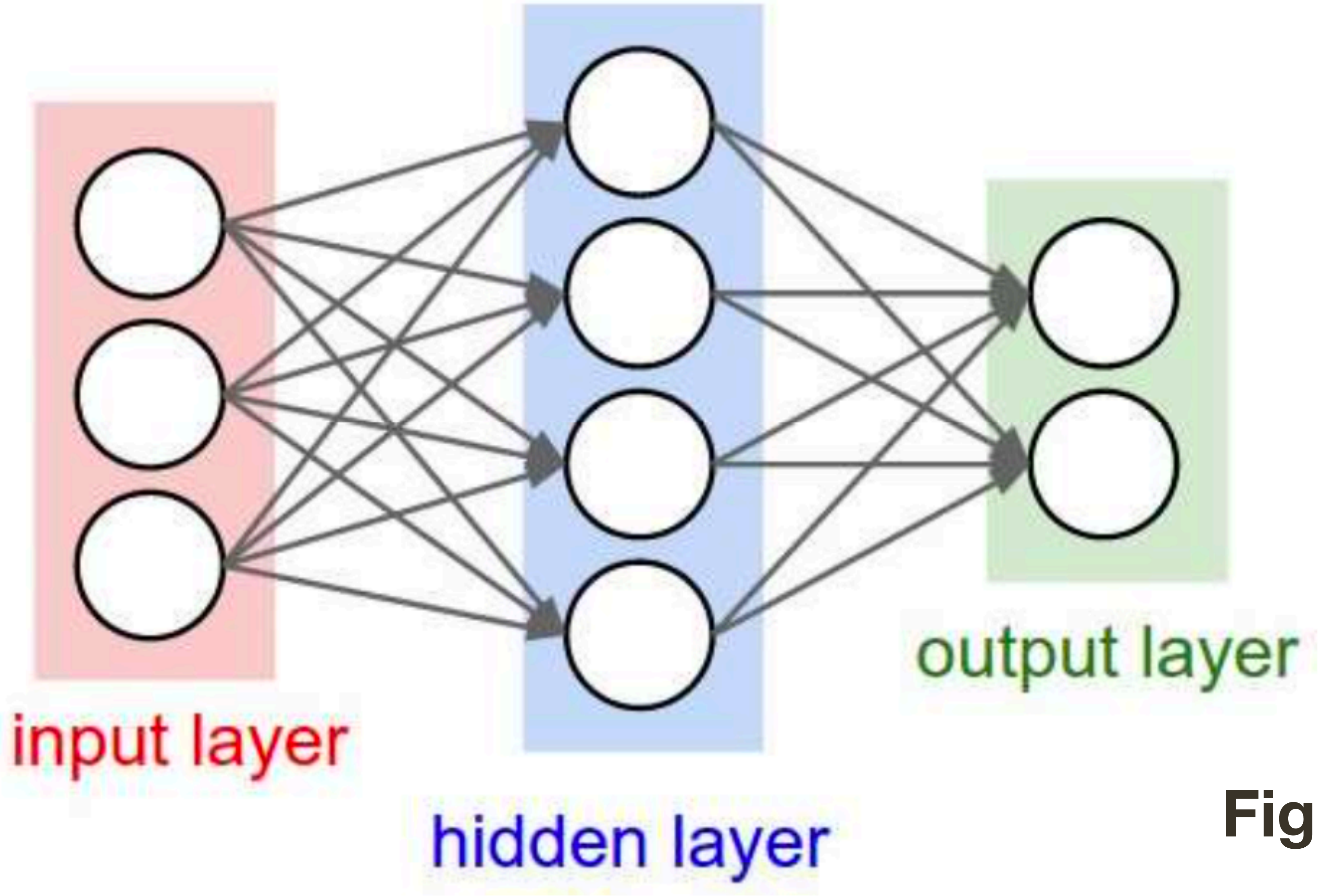



**Figure credit**: Fei-Fei and Karpathy

# **Activation** Function

$$\hat{\mathbf{y}} = f(\mathbf{x}, \mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2) = \sigma\left(\mathbf{W}_2^{(2\times 4)}\sigma\left(\mathbf{W}_1^{(4\times 3)}\mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)}\right)$$

$$= \mathbf{W}_2^{(2\times 4)}\left(\mathbf{W}_1^{(4\times 3)}\mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)}$$

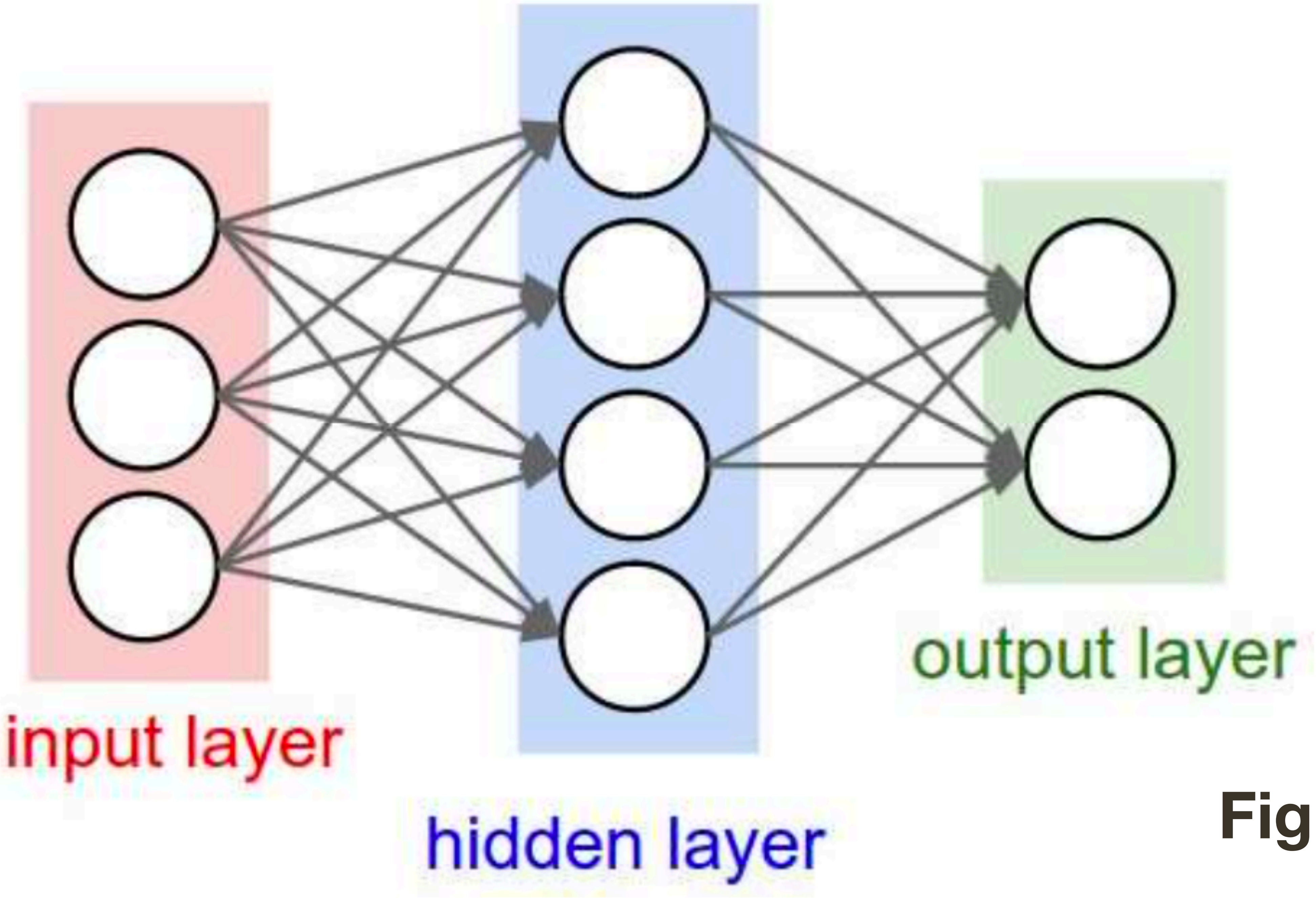



**Figure credit**: Fei-Fei and Karpathy

# **Activation** Function

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

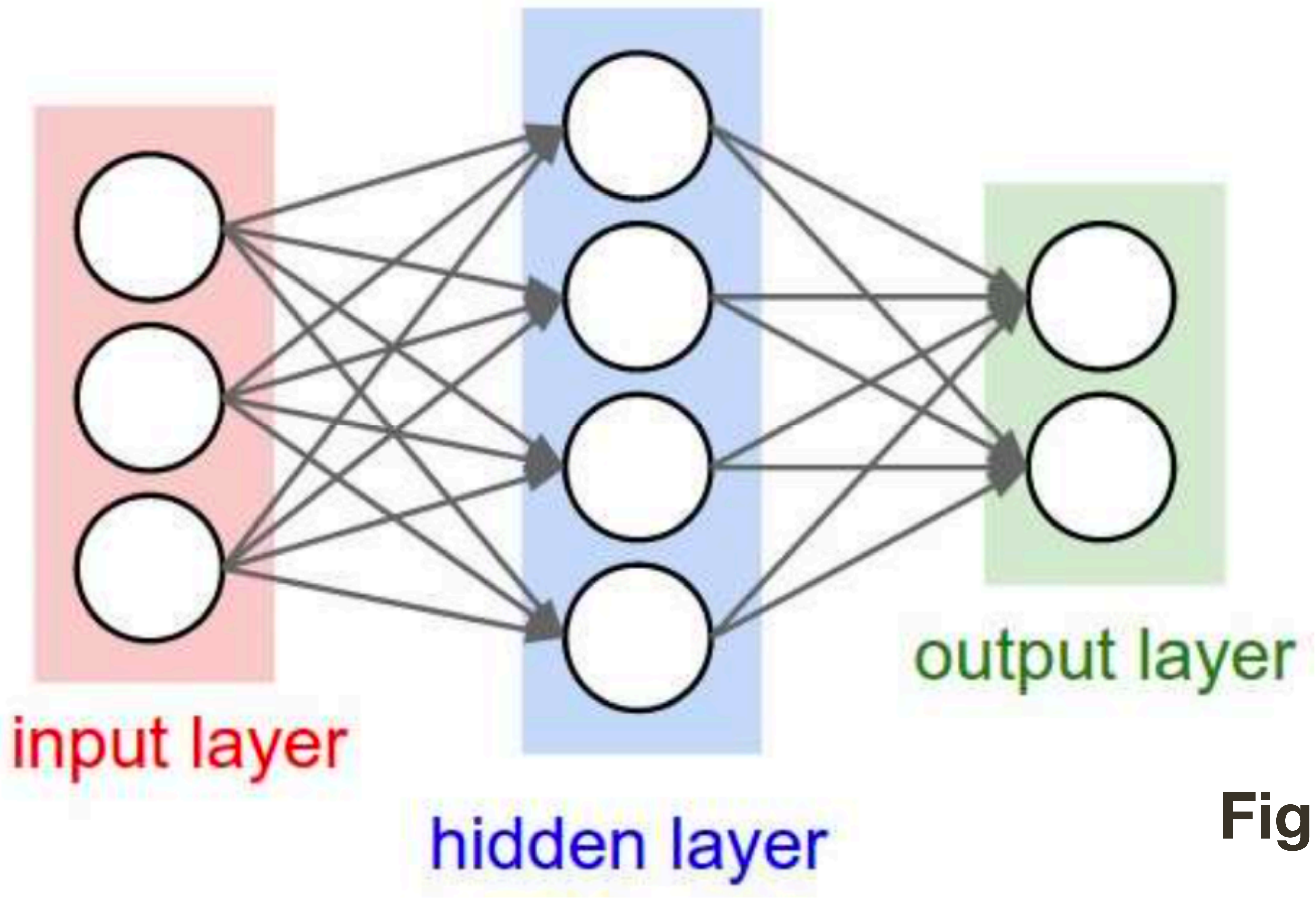



**Figure credit**: Fei-Fei and Karpathy

# **Activation** Function

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

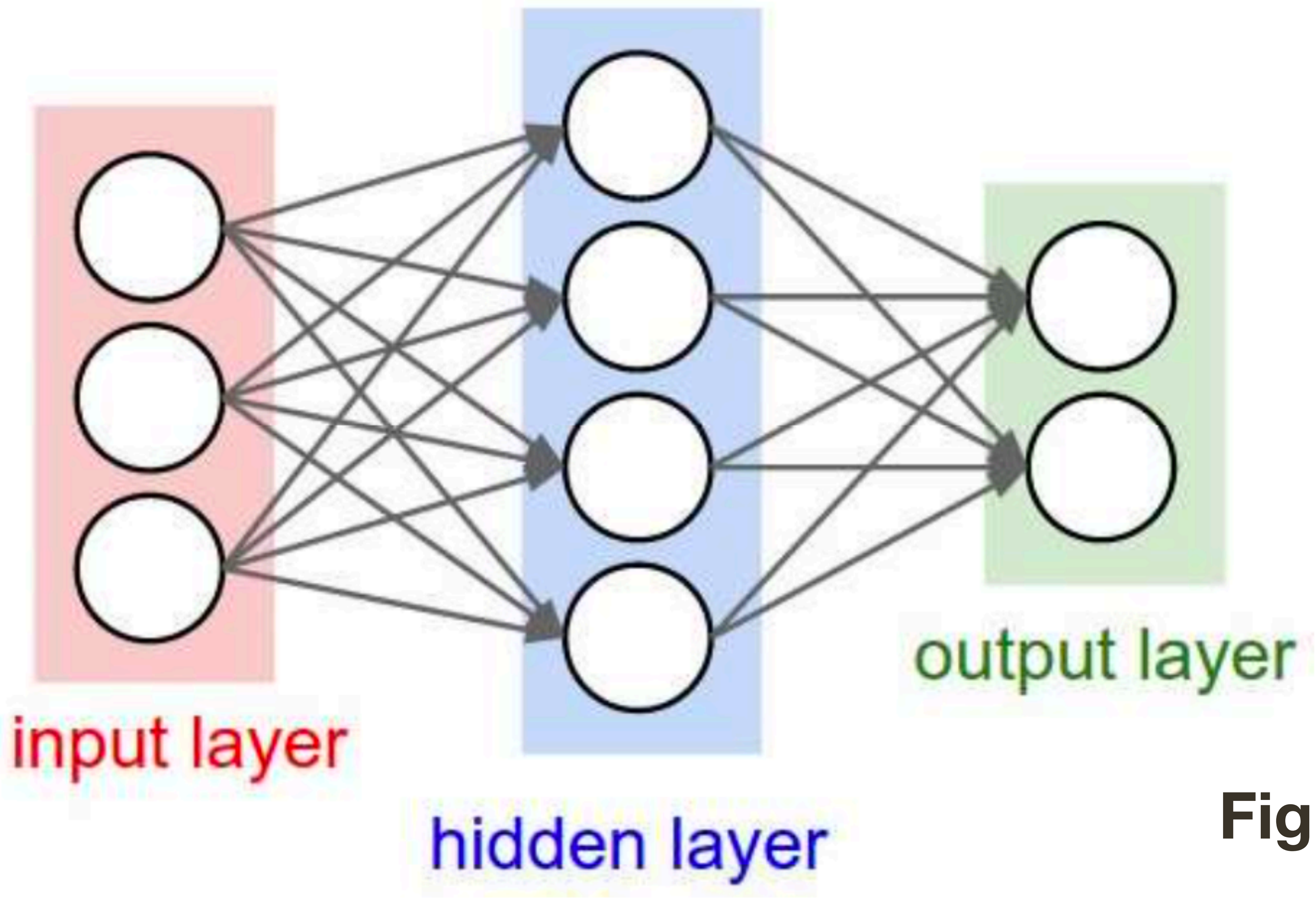



**Figure credit**: Fei-Fei and Karpathy

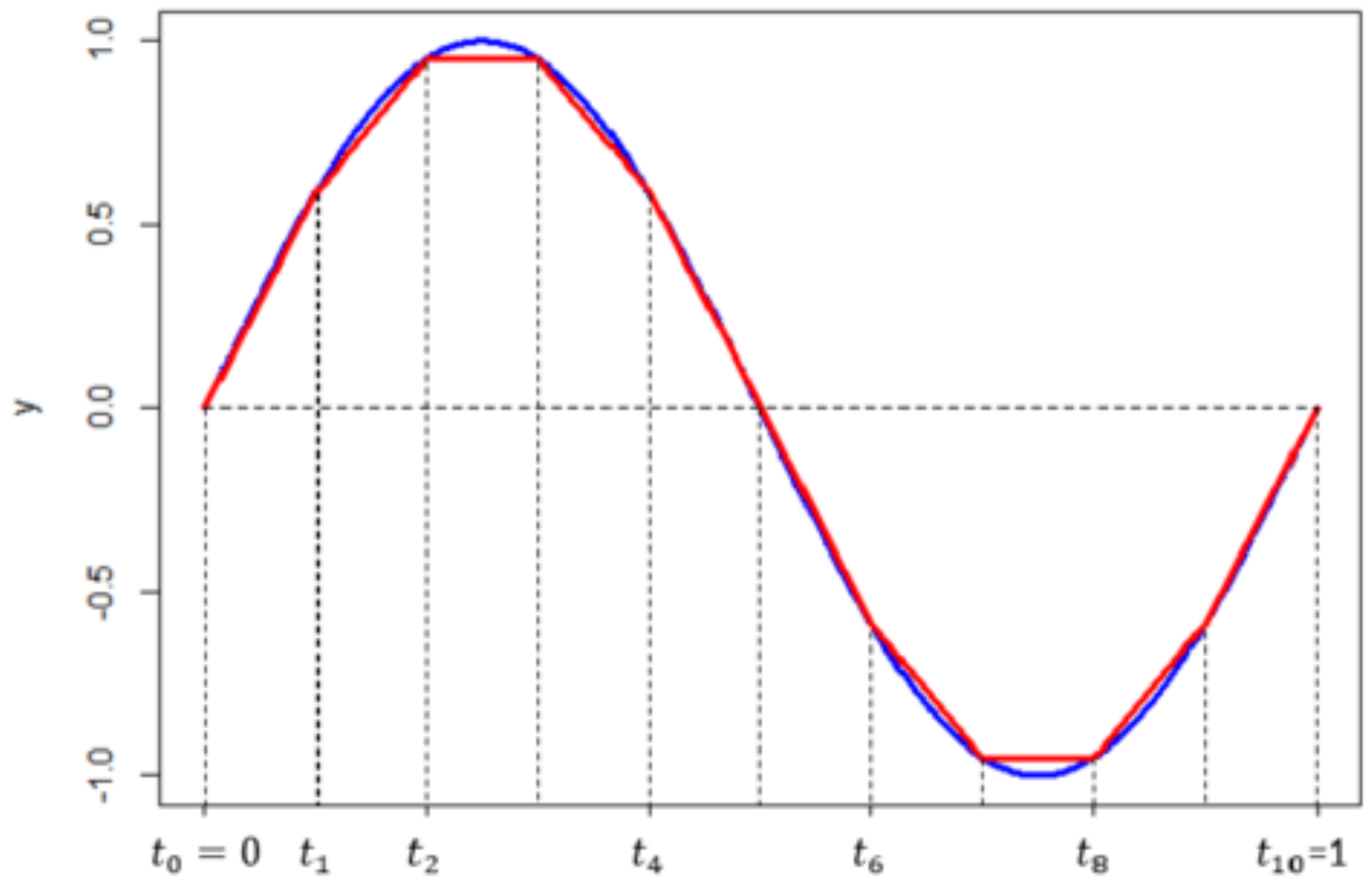
# **Activation** Function

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.

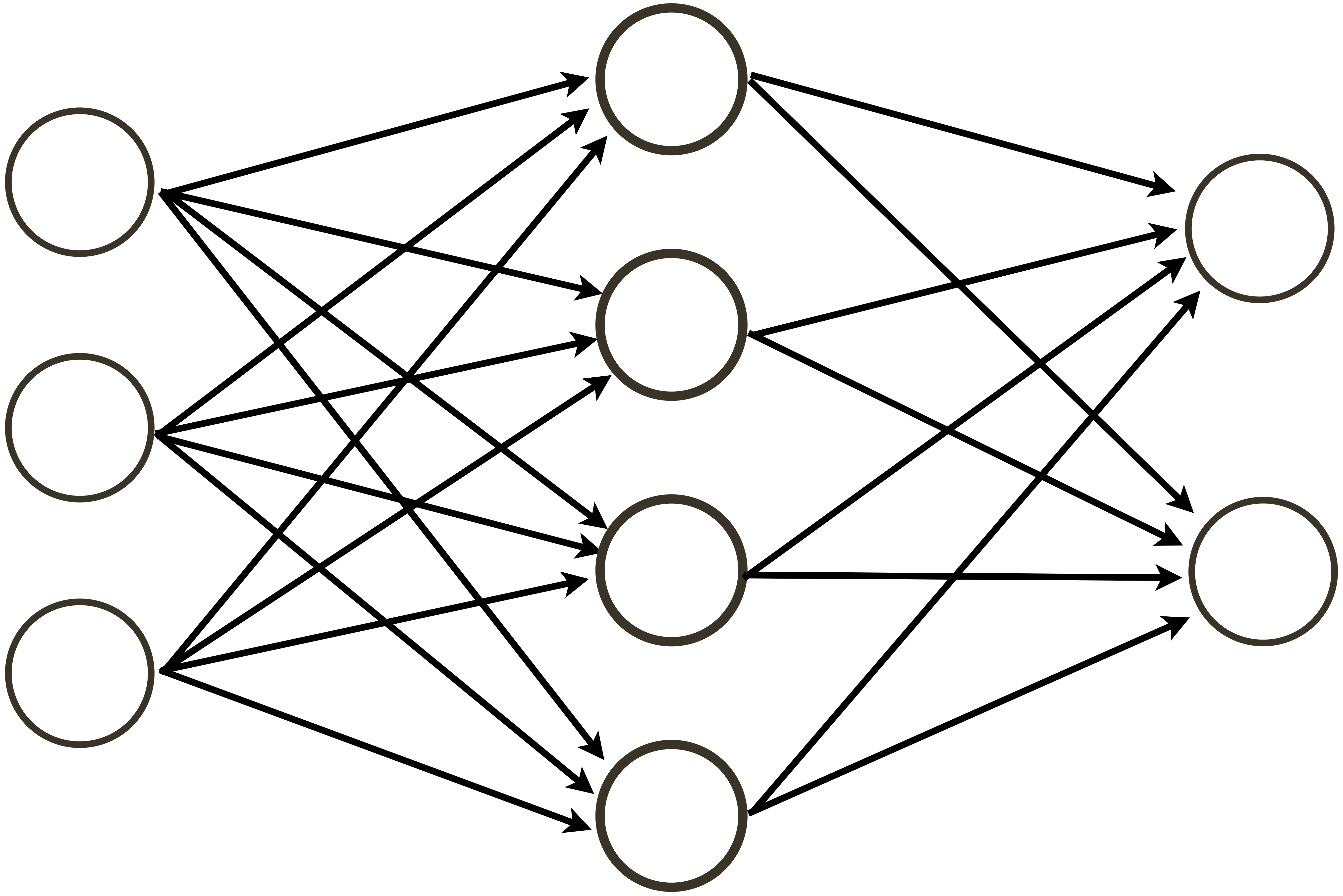
[ Lu *et al*., NIPS 2017 ]

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

[ Lin and Jegelka, NIPS 2018 ]

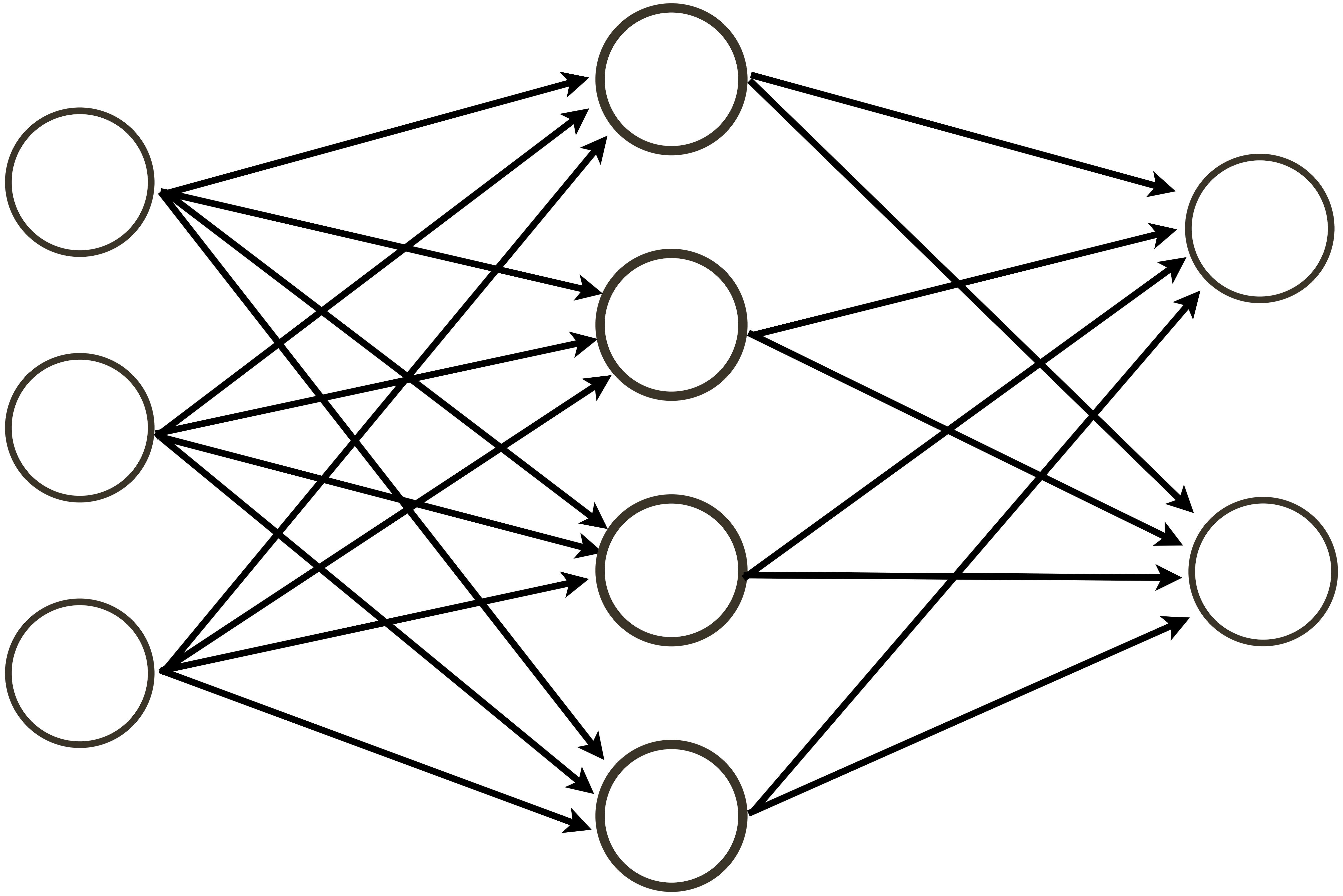
# **Activation** Function

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



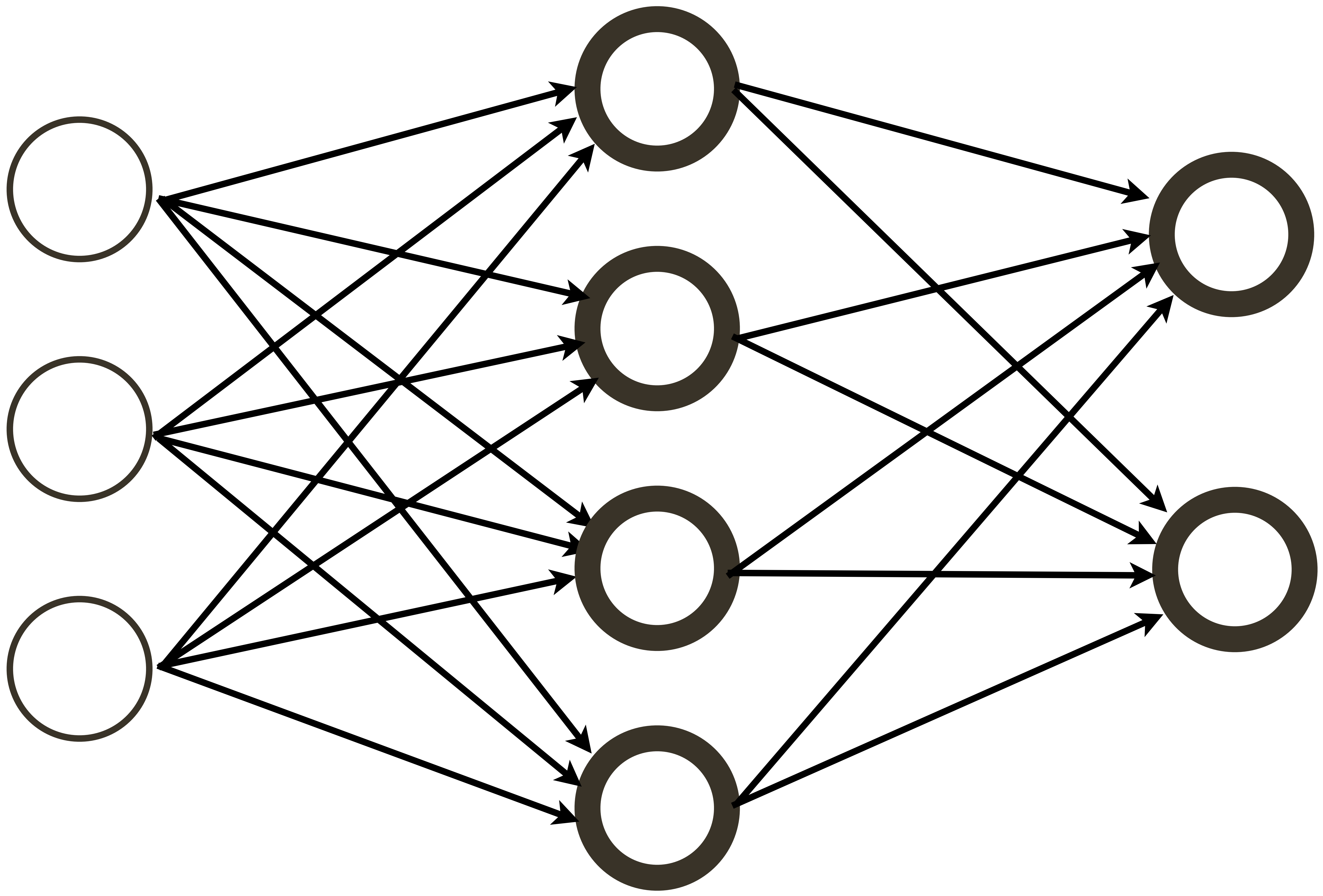
# **Neural** Network

How many neurons?



# **Neural** Network

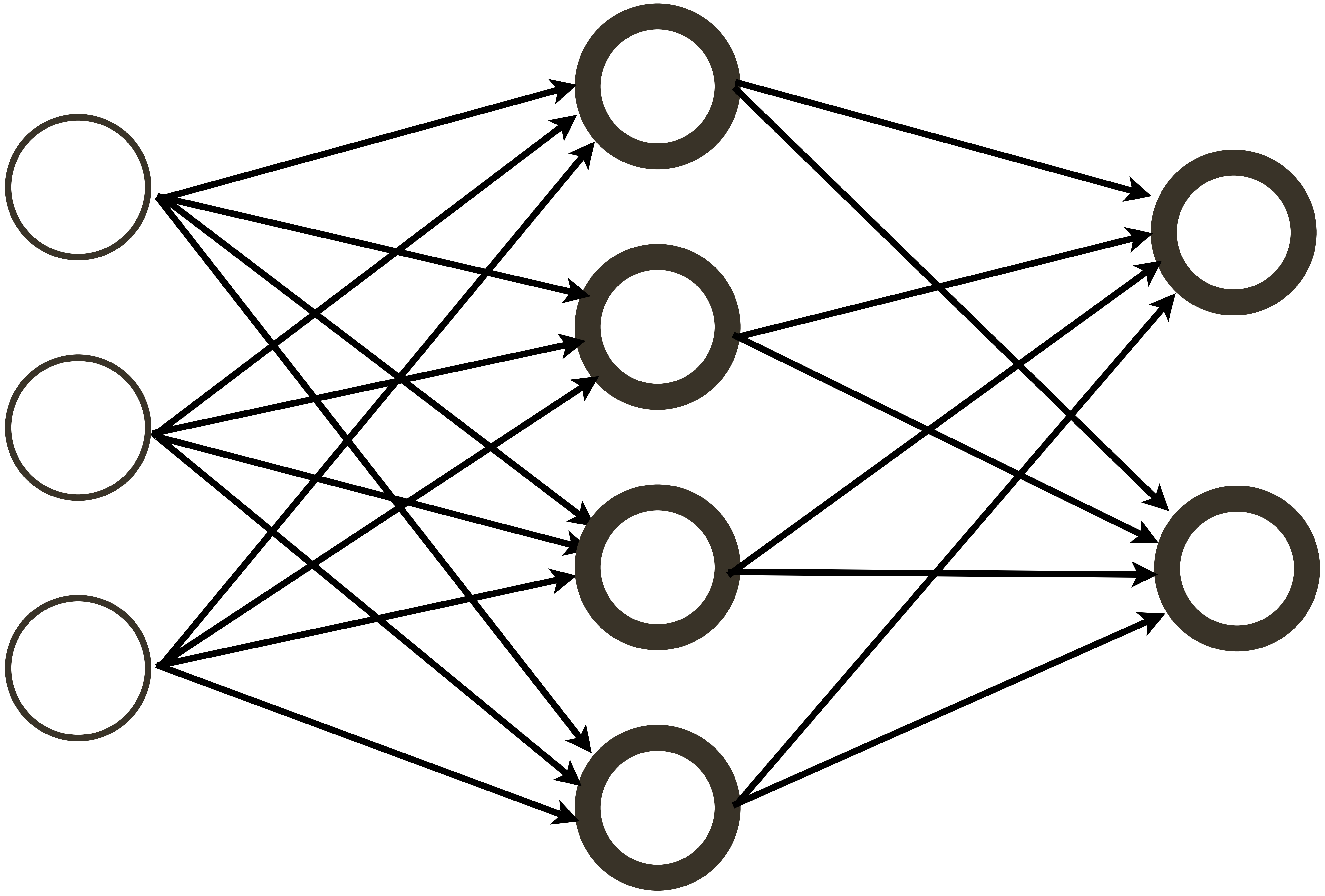
How many neurons? 4+2 = 6



# **Neural** Network

How many neurons? 4+2 = 6

How many weights?

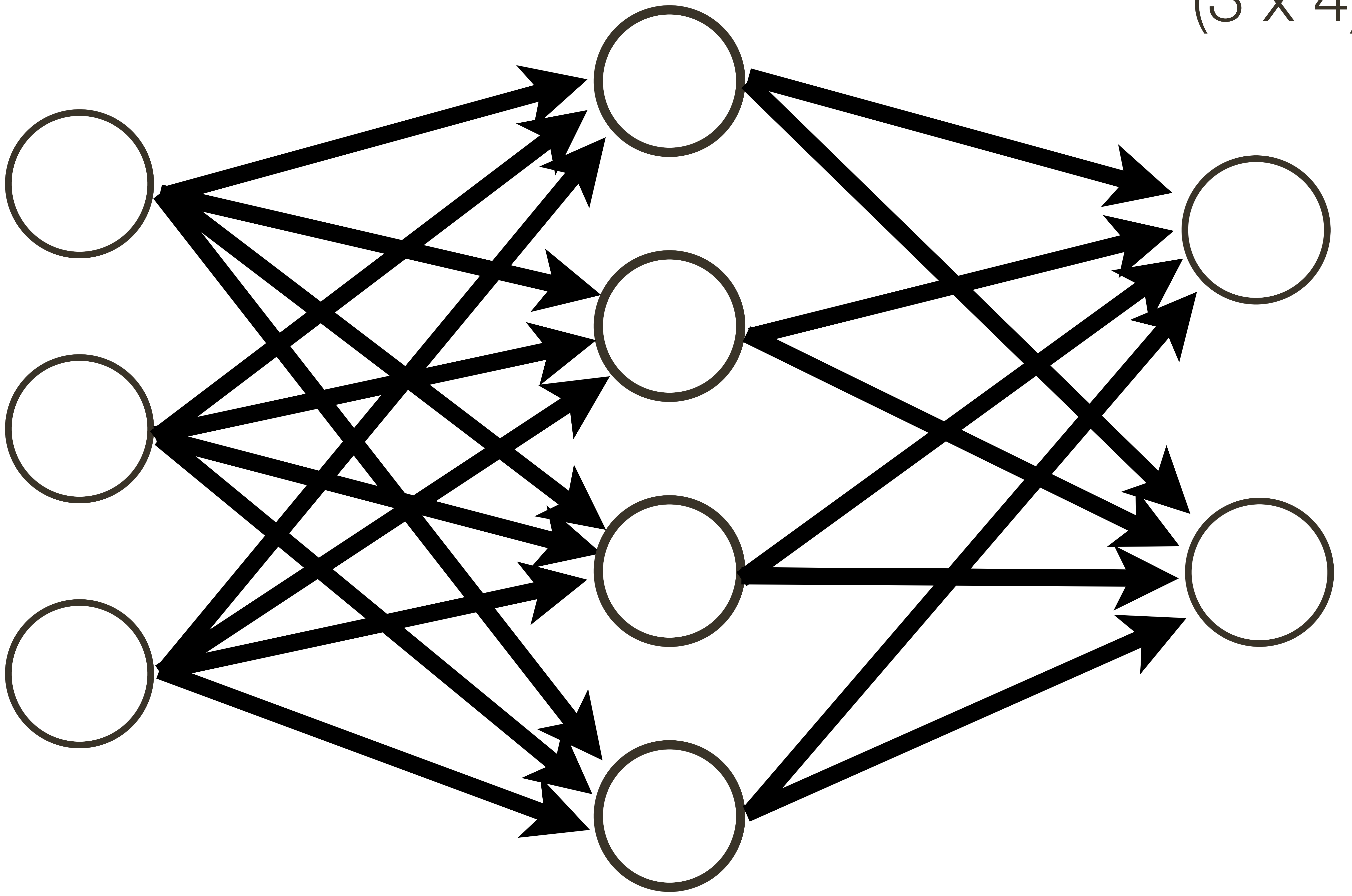


# **Neural** Network

How many neurons? 4+2 = 6

How many weights?

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

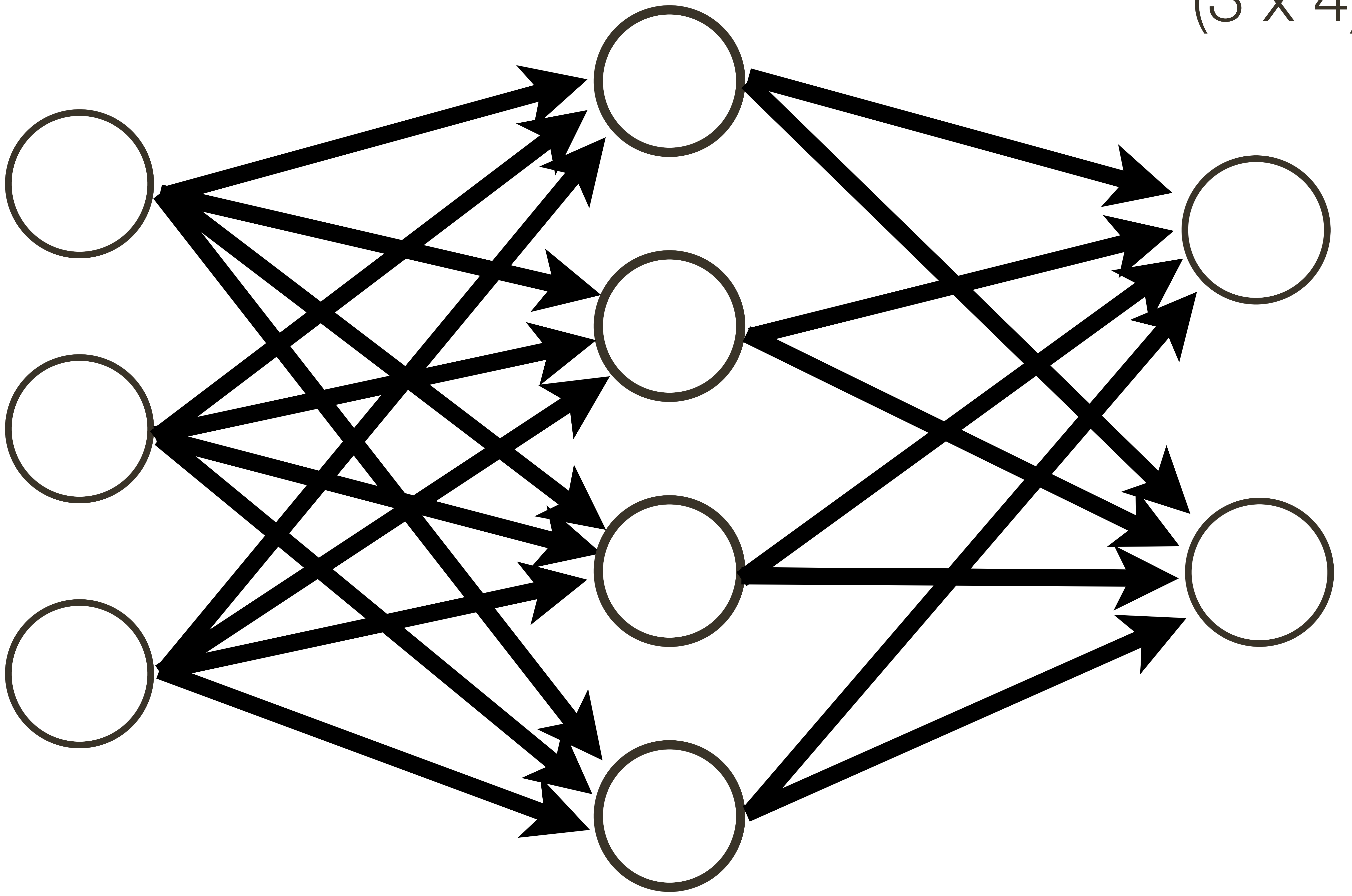


# **Neural** Network

How many neurons? 4+2 = 6

How many weights?

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



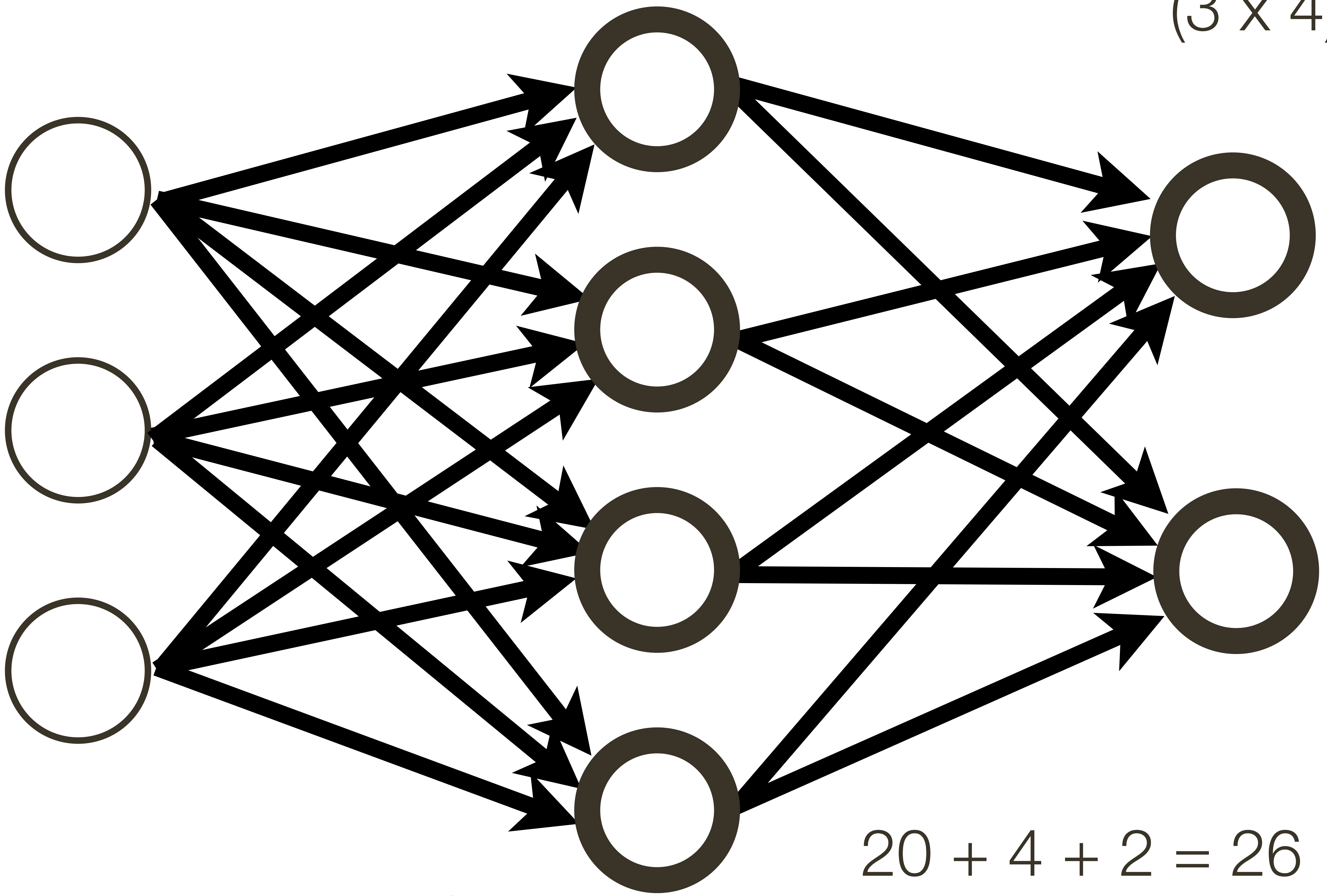
How many learnable parameters?

# **Neural** Network

How many neurons? 4+2 = 6

How many weights?

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



20 + 4 + 2 = 26

bias terms

How many learnable parameters?

# **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: $$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}\right)$$

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: $$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}\right)$$

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: $L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}\right)$

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: $$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}\right)$$

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:

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

# Backpropagation

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

— e.g. cross-entropy loss: $$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}\right)$$

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:

$$\begin{array}{lcccl} f & & & & \\ c_1 = -2.85 & & 0.058 & & 0.016 \\ c_2 = 0.86 & \xrightarrow{\text{exp}} & 2.36 & \xrightarrow{\text{Normalize to sum to 1}} & 0.631 \\ c_3 = 0.28 & & 1.32 & & 0.353 \end{array}$$

# Backpropagation

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

— e.g. cross-entropy loss: $$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}\right)$$

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

$$
\begin{array}{lcccc}
f & & & & \\
c_1 = -2.85 & & 0.058 & & 0.016 \\
c_2 = 0.86 & \xrightarrow{\exp} & 2.36 & \xrightarrow{\text{Normalize to sum to 1}} & 0.631 \\
c_3 = 0.28 & & 1.32 & & 0.353
\end{array}
$$

# Backpropagation

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

— e.g. cross-entropy loss: $L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}\right)$

**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

$$\begin{array}{ccccc} f & & & & \\ c_1 = -2.85 & & 0.058 & \text{Normalize to sum to 1} & 0.016 \\ c_2 = 0.86 & \xrightarrow{\exp} & 2.36 & \longrightarrow & 0.631 \\ c_3 = 0.28 & & 1.32 & & 0.353 \end{array}$$

# Backpropagation

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

— e.g. cross-entropy loss: $$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}\right)$$

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

$$\begin{array}{c} f \\ c_1 = -2.85 \\ c_2 = 0.86 \\ c_3 = 0.28 \end{array} \xrightarrow{\text{exp}} \begin{array}{c} 0.058 \\ 2.36 \\ 1.32 \end{array} \xrightarrow{\text{Normalize to sum to 1}} \begin{array}{c} 0.016 \\ 0.631 \\ 0.353 \end{array} \quad L_i = -\log(0.353) = 1.04$$

# Backpropagation

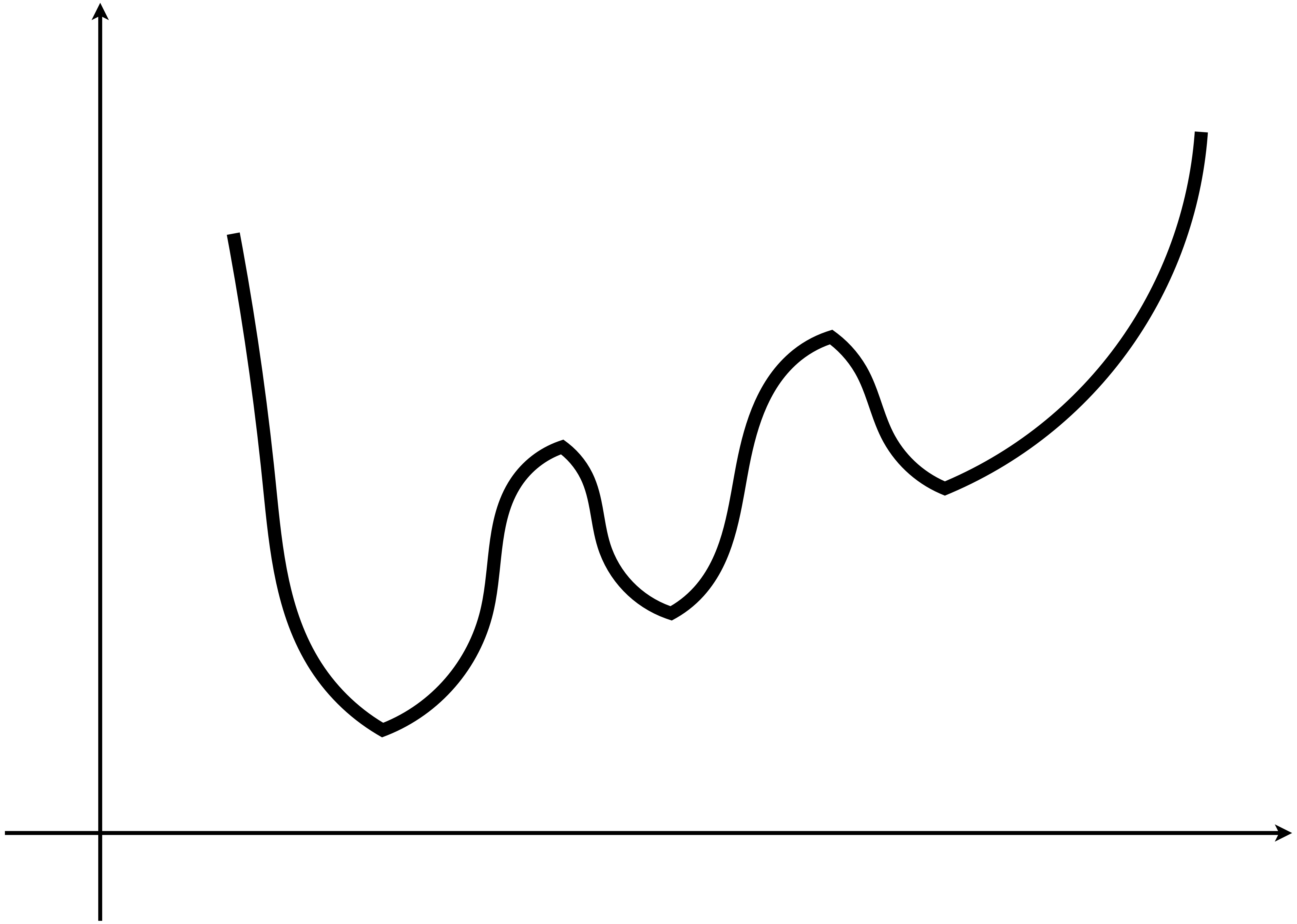
When training a neural network, the final output will be some loss (error) function

— e.g. cross-entropy loss: $L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}\right)$

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.

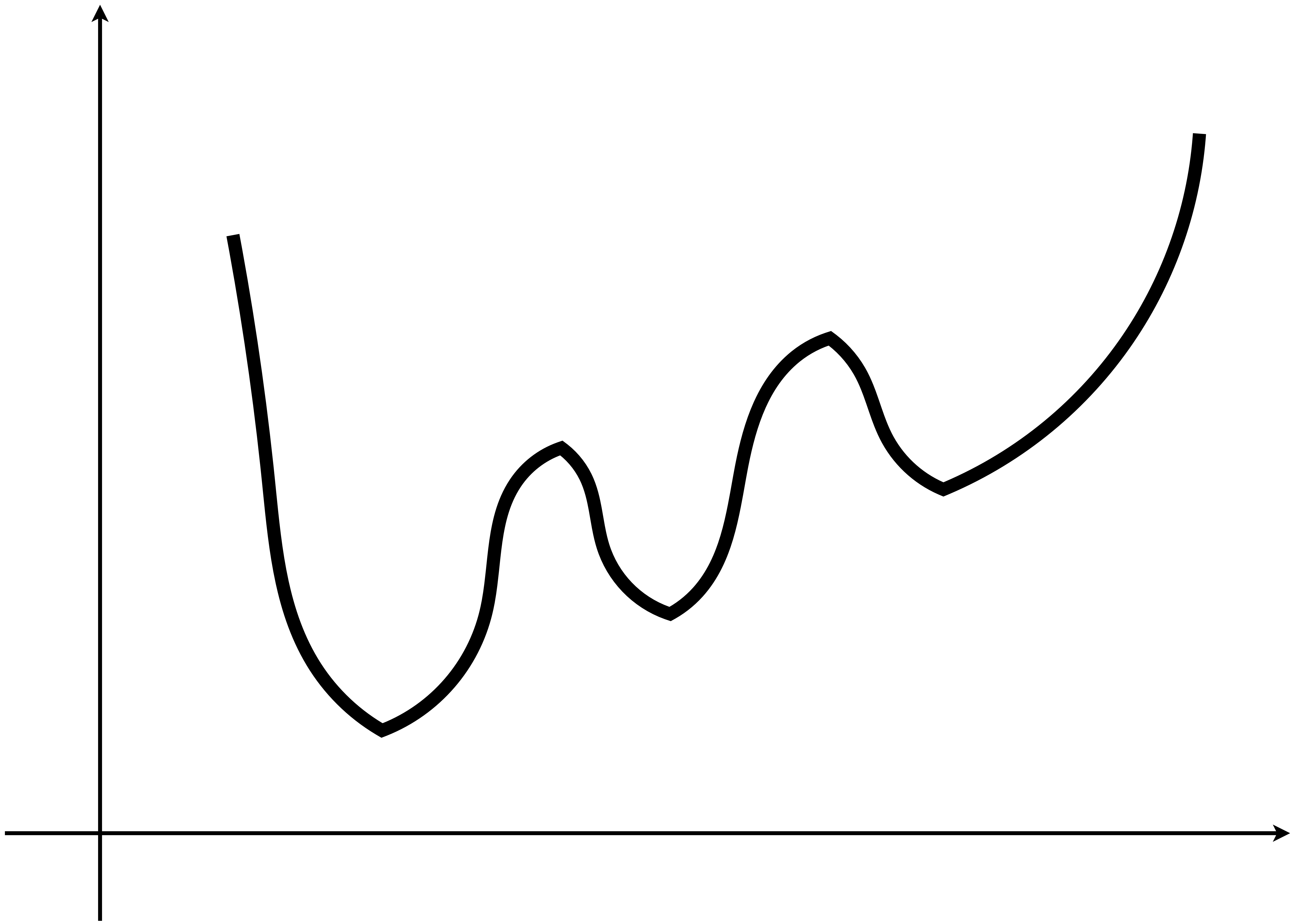
We want to compute the **gradient** of the loss with respect to the network parameters so that we can incrementally adjust the network parameters

# **Gradient** Descent



*slide adopted from V. Ordonex

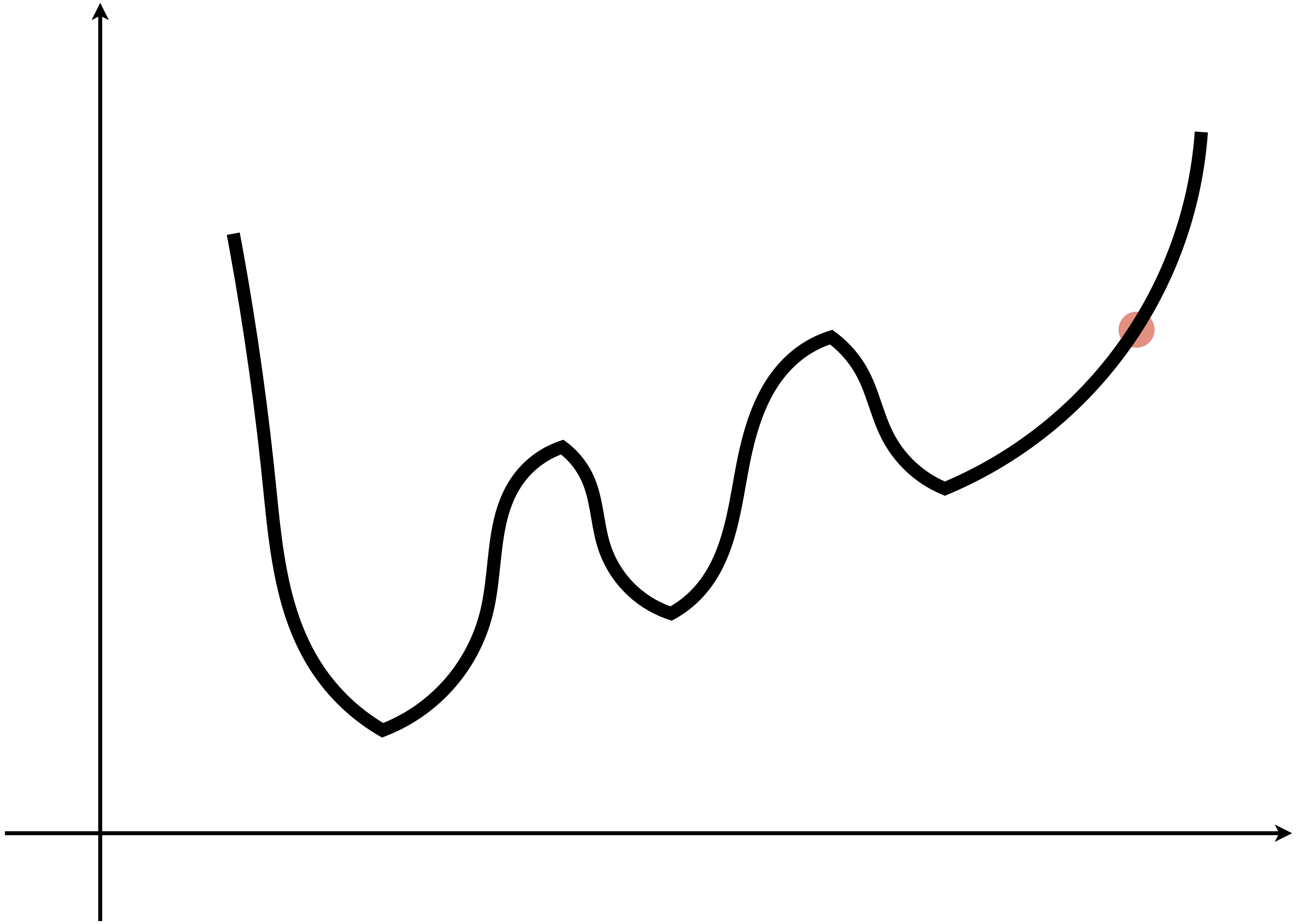
# **Gradient** Descent



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

*slide adopted from V. Ordonex

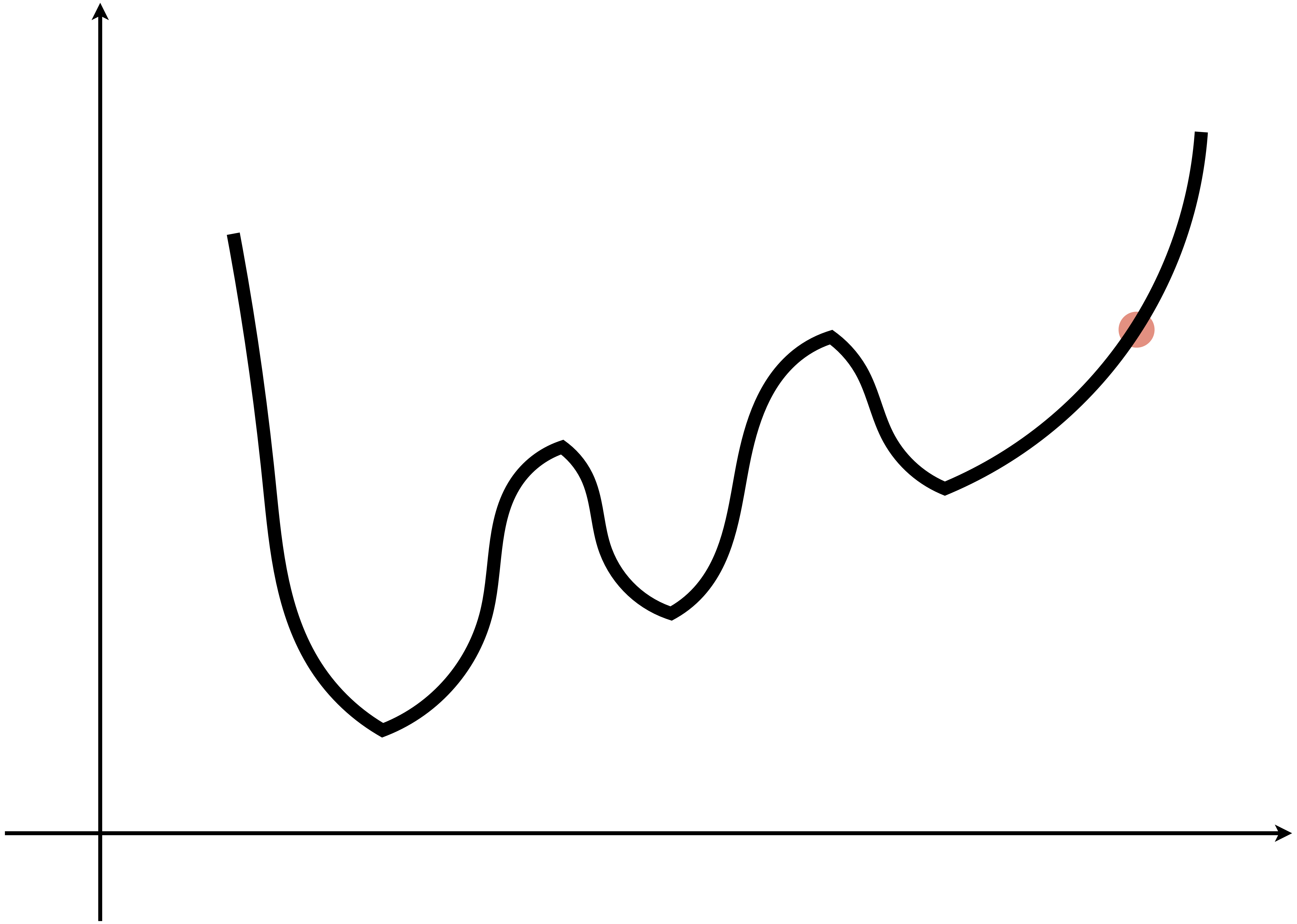
# **Gradient** Descent



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 $\boxed{\mathbf{W}_0, \mathbf{b}_0}$

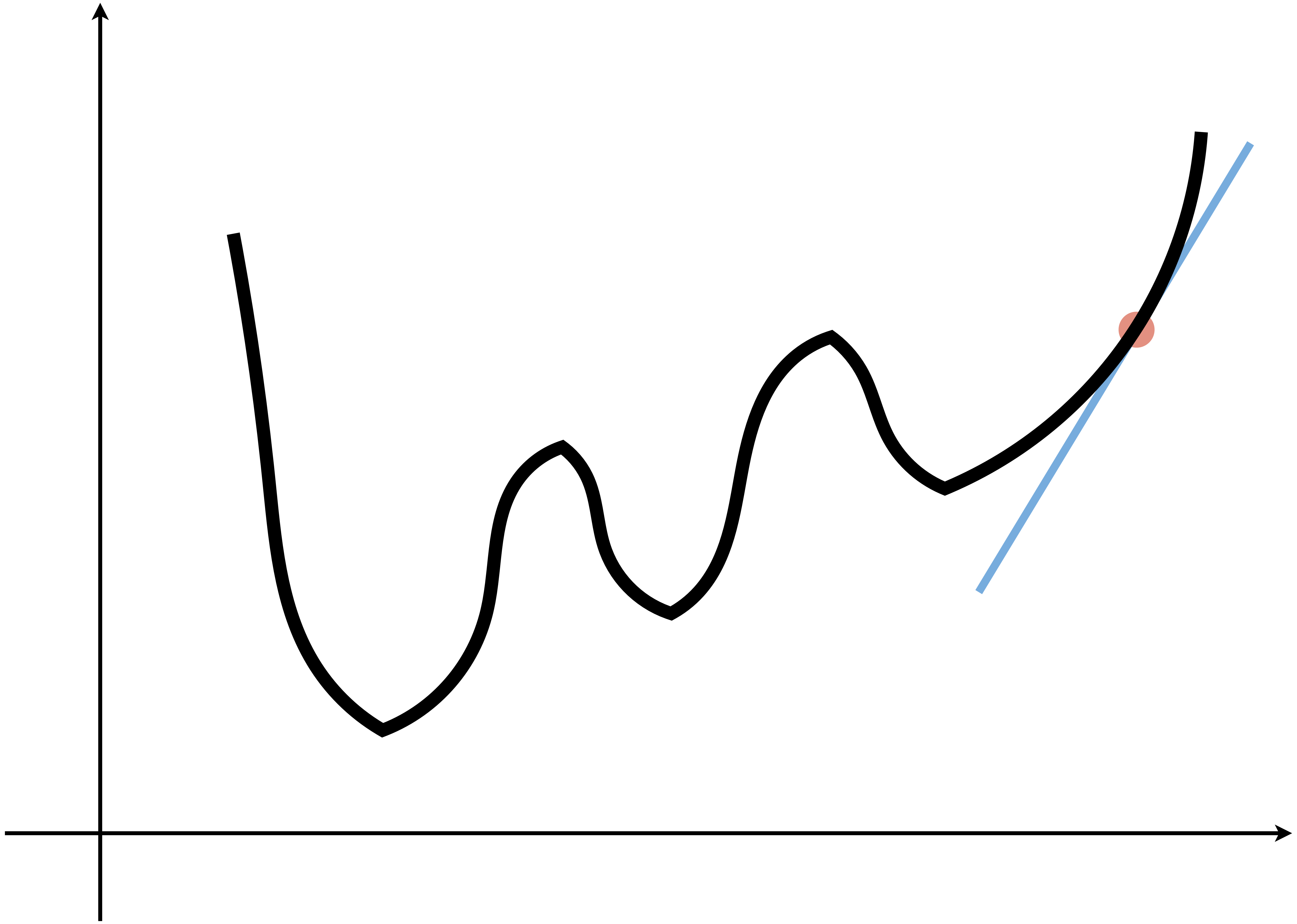
For $k = 0$ to max number of iterations

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 $\mathbf{W}_0, \mathbf{b}_0$

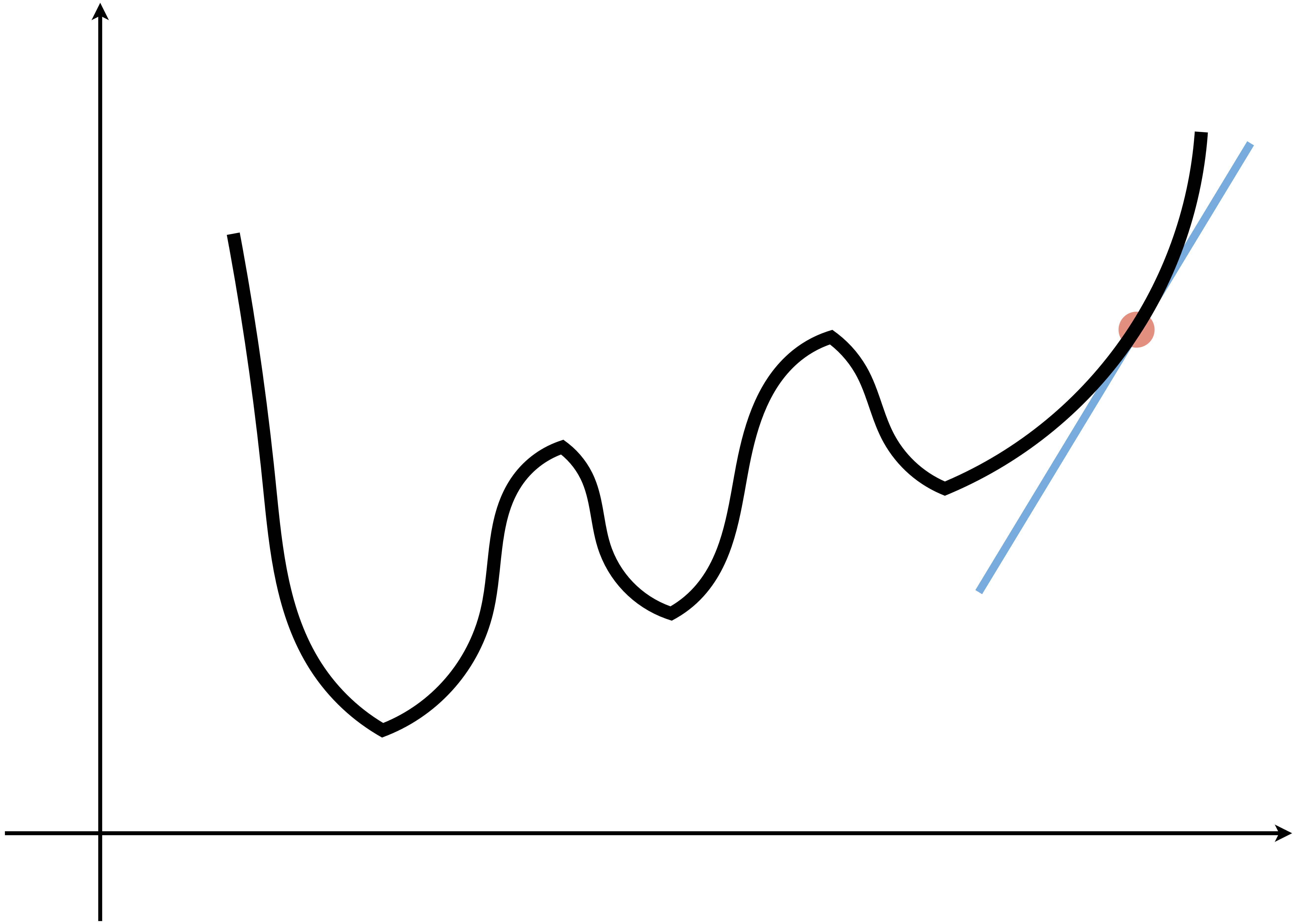
For $k = 0$ to max number of iterations

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 $\boxed{\mathbf{W}_0, \mathbf{b}_0}$

For $k = 0$ to max number of iterations

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

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

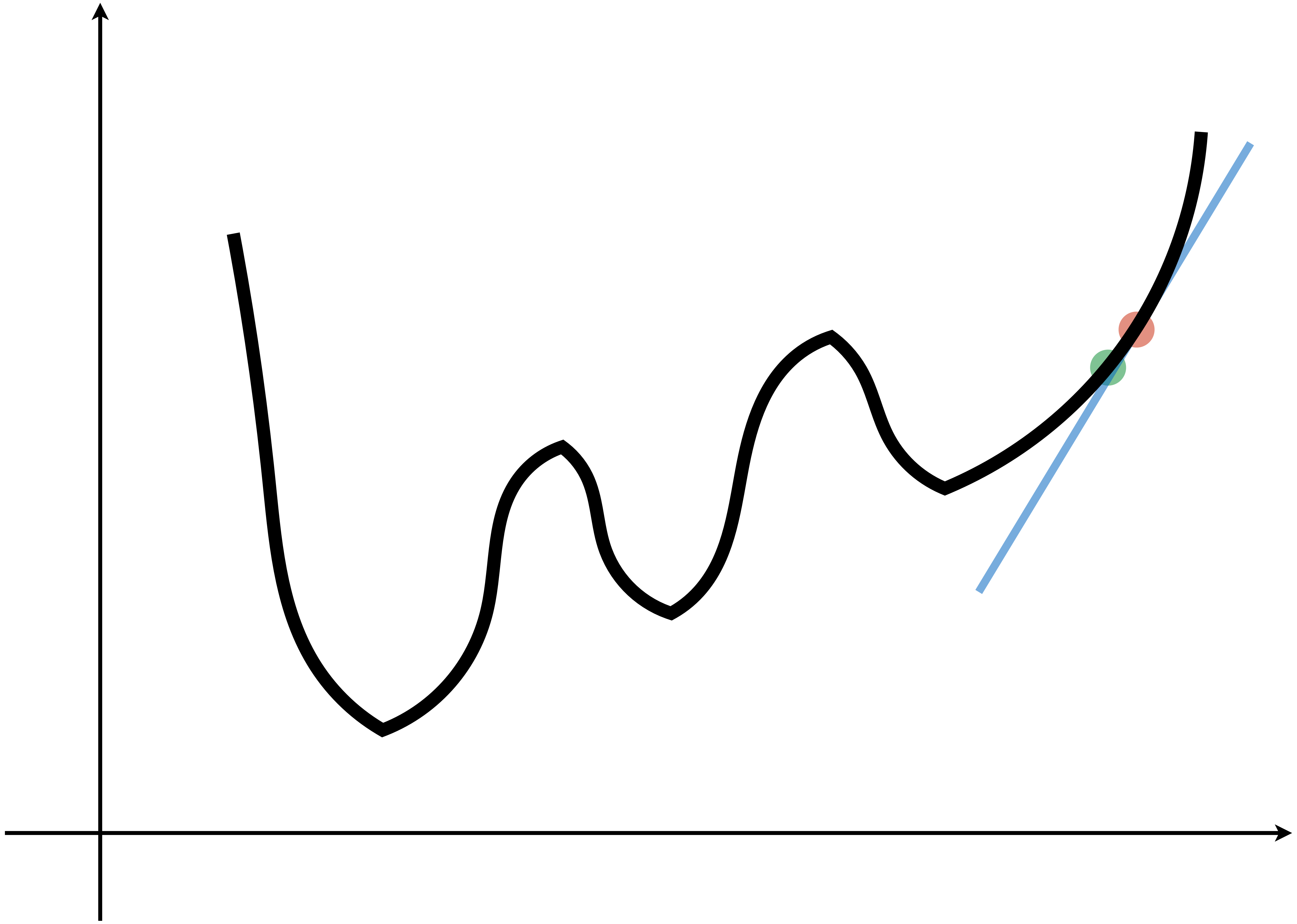
3. Re-estimate the parameters

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

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

*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:

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

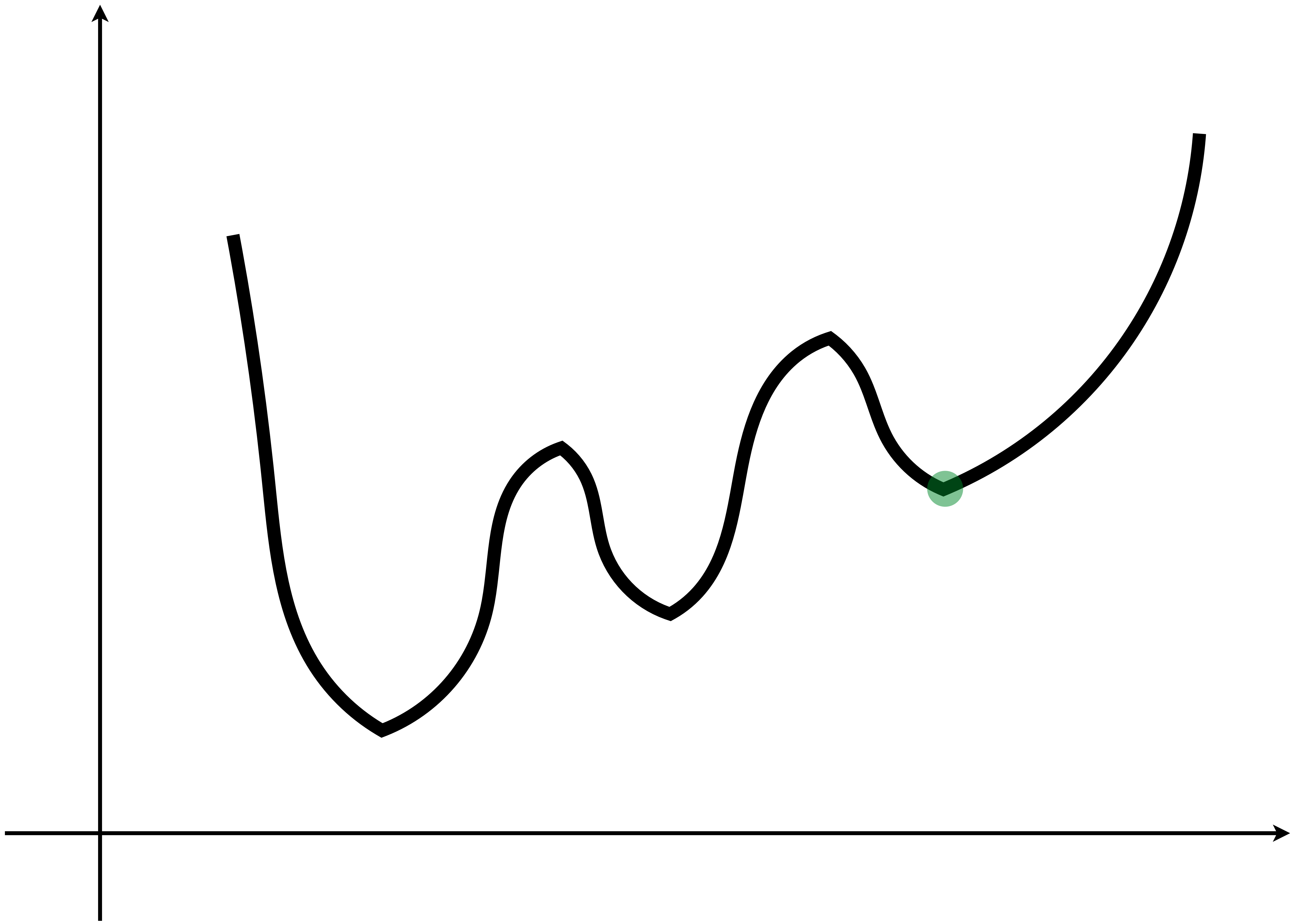
3. Re-estimate the parameters

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

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

*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:

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

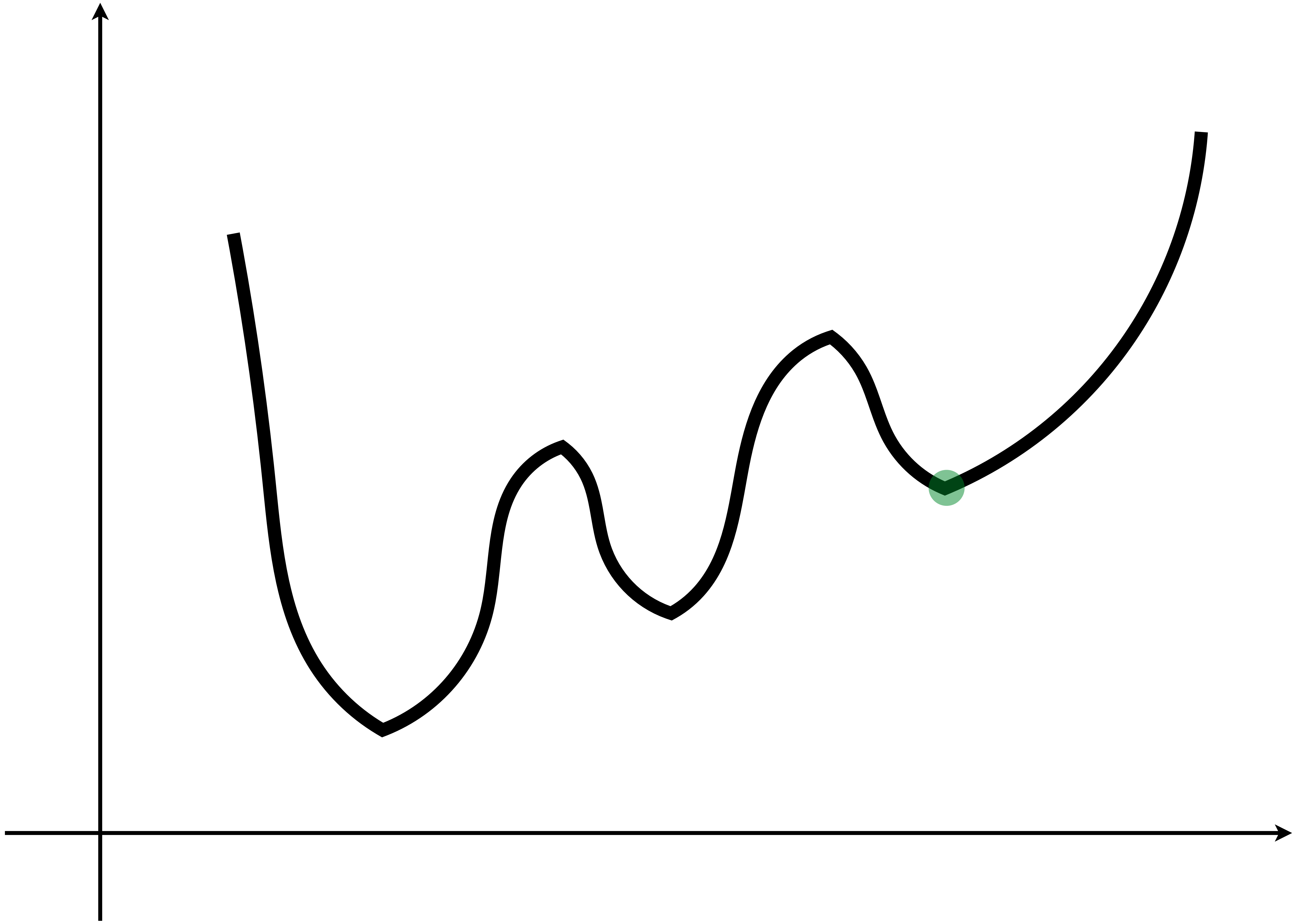
3. Re-estimate the parameters

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

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

*slide adopted from V. Ordonex

# **Gradient** Descent



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

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

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

$\lambda$ - is the learning rate

*slide adopted from V. Ordonex

# **Gradient** Descent

**Loss:** $$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = ||\mathbf{y} - \hat{\mathbf{y}}|| = ||\mathbf{y} - f(\mathbf{x}, \mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2)||$$

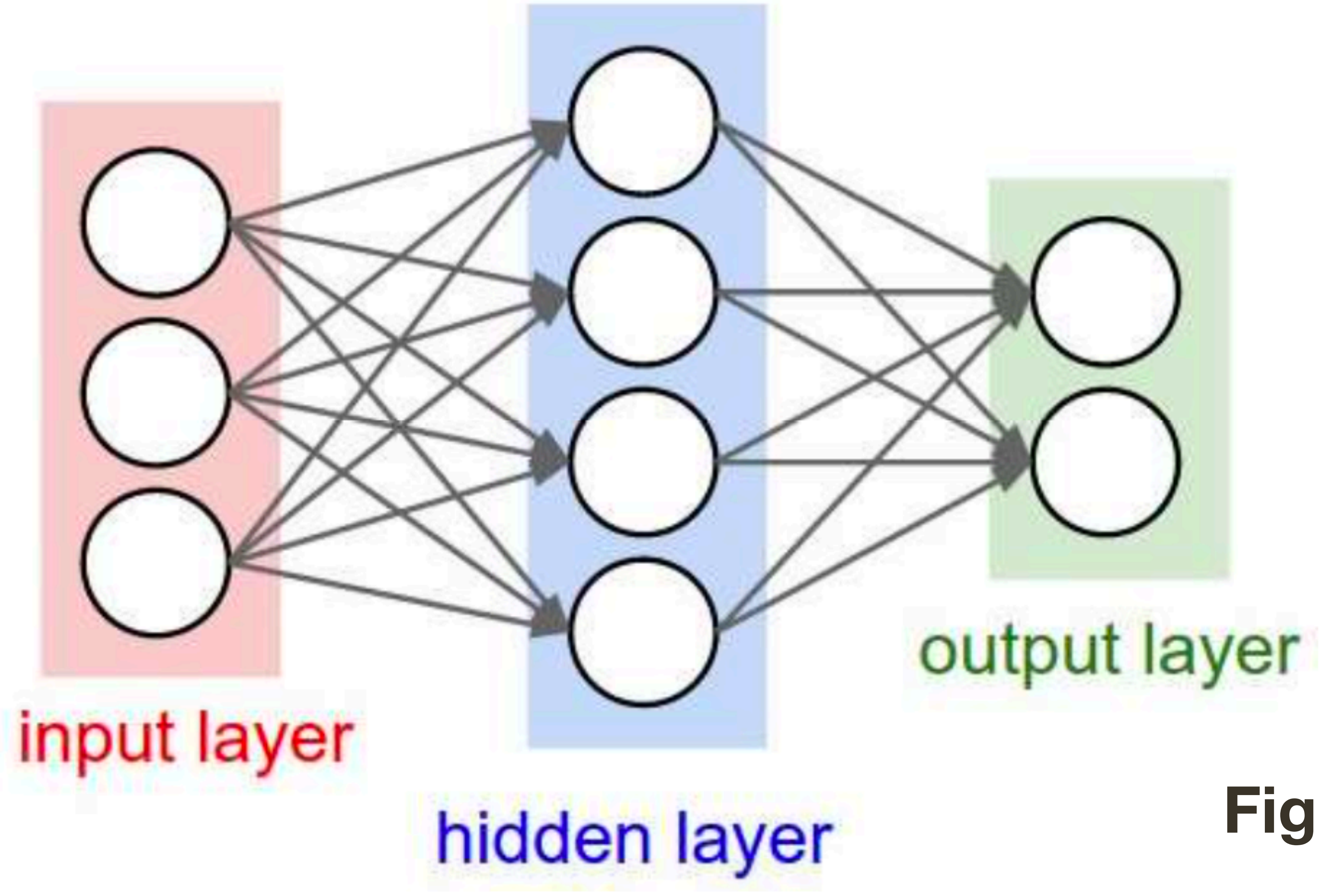



**Figure credit**: Fei-Fei and Karpathy

$$\hat{\mathbf{y}} = f(\mathbf{x}, \mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2) = \sigma\left(\mathbf{W}_2^{(2\times 4)}\sigma\left(\mathbf{W}_1^{(4\times 3)}\mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)}\right)$$

# **Gradient** Descent

**Loss:**

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = ||\mathbf{y} - \hat{\mathbf{y}}|| = ||\mathbf{y} - f(\mathbf{x}, \mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2)||$$

## **Gradient** Descent

$$\mathbf{W}_{1,i,j} = \mathbf{W}_{1,i,j} - \lambda \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{W}_{1,i,j}}$$

$$\mathbf{b}_{1,i} = \mathbf{b}_{1,i} - \lambda \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{b}_{1,i}}$$

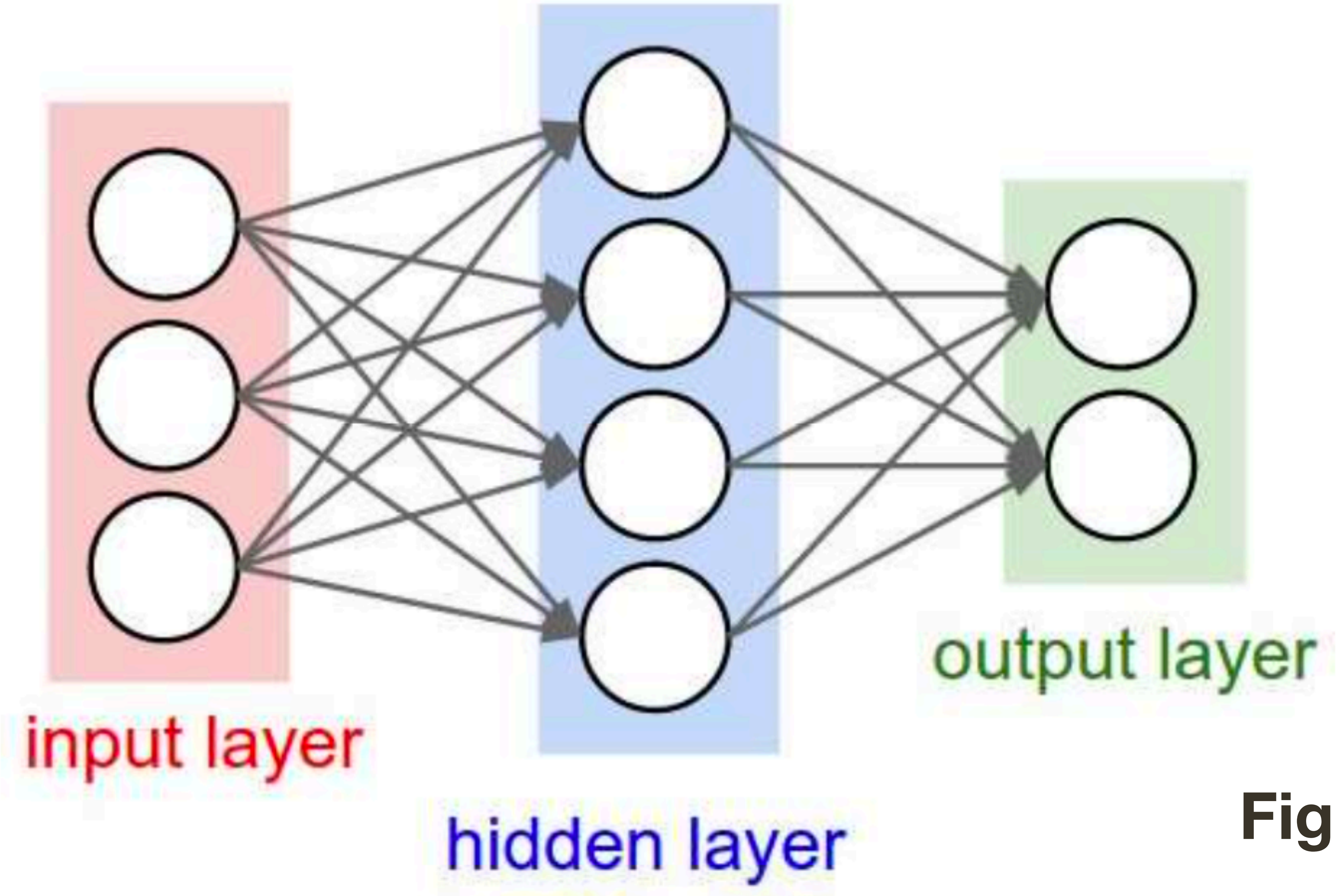



**Figure credit**: Fei-Fei and Karpathy

$$\hat{\mathbf{y}} = f(\mathbf{x}, \mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2) = \sigma\left(\mathbf{W}_2^{(2\times 4)} \sigma\left(\mathbf{W}_1^{(4\times 3)} \mathbf{x} + \mathbf{b}_1^{(4)}\right) + \mathbf{b}_2^{(2)}\right)$$

# Backpropagation

The parameters of a neural network are learned using **backpropagation**, which computes gradients via recursive application of the **chain rule** from calculus

# Backpropagation

The parameters of a neural network are learned using **backpropagation**, which computes gradients via recursive application of the **chain rule** from calculus

Suppose $f(x, y) = xy$. What is the partial derivative of $f$ with respect to $x$? What is the partial derivative of $f$ with respect to $y$?

# Backpropagation

The parameters of a neural network are learned using **backpropagation**, which computes gradients via recursive application of the **chain rule** from calculus

Suppose $f(x, y) = xy$. What is the partial derivative of $f$ with respect to $x$? What is the partial derivative of $f$ with respect to $y$?

$$\frac{\partial f}{\partial x} = y \qquad \frac{\partial f}{\partial y} = x$$

# Backpropagation

Suppose $f(x, y) = x + y$. What is the partial derivative of $f$ with respect to $x$? What is the partial derivative of $f$ with respect to $y$?

# Backpropagation

Suppose $f(x, y) = x + y$. What is the partial derivative of $f$ with respect to $x$? What is the partial derivative of $f$ with respect to $y$?

$$\frac{\partial f}{\partial x} = 1 \qquad \frac{\partial f}{\partial y} = 1$$

# Backpropagation

A trickier example: $f(x, y) = \max(x, y)$

# Backpropagation

A trickier example: $f(x, y) = \max(x, y)$

$$\frac{\partial f}{\partial x} = \mathbf{1}(x \geq y) \qquad \frac{\partial f}{\partial y} = \mathbf{1}(y \geq x)$$

That is, the (sub)gradient is 1 on the input that is larger, and 0 on the other input

— For example, say x = 4, y = 2. Increasing y by a tiny amount does not change the value of f (f will still be 4), hence the gradient on y is zero.

# Backpropagation

We can compose more complicated functions and compute their gradients by applying the **chain rule** from calculus

# Backpropagation

We can compose more complicated functions and compute their gradients by applying the **chain rule** from calculus

Suppose $f(x, y, z) = (x + y)z$. What are the partial derivatives of $f$ with respect to $x$? $y$? $z$?

# Backpropagation

We can compose more complicated functions and compute their gradients by applying the **chain rule** from calculus

Suppose $f(x, y, z) = (x + y)z$. What are the partial derivatives of $f$ with respect to $x$? $y$? $z$?

For illustration we break this expression into $q = x + y$ and $f = qz$. This is a sum and a product, and we have just seen how to compute partial derivatives for these.

# Backpropagation

We can compose more complicated functions and compute their gradients by applying the **chain rule** from calculus

Suppose $f(x, y, z) = (x + y)z$. What are the partial derivatives of $f$ with respect to $x$? $y$? $z$?

For illustration we break this expression into $q = x + y$ and $f = qz$. This is a sum and a product, and we have just seen how to compute partial derivatives for these.

By the chain rule

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = z \cdot 1 = z$$

# Backpropagation

We can compose more complicated functions and compute their gradients by applying the **chain rule** from calculus

Suppose $f(x, y, z) = (x + y)z$. What are the partial derivatives of $f$ with respect to $x$? $y$? $z$?

For illustration we break this expression into $q = x + y$ and $f = qz$. This is a sum and a product, and we have just seen how to compute partial derivatives for these.

By the chain rule

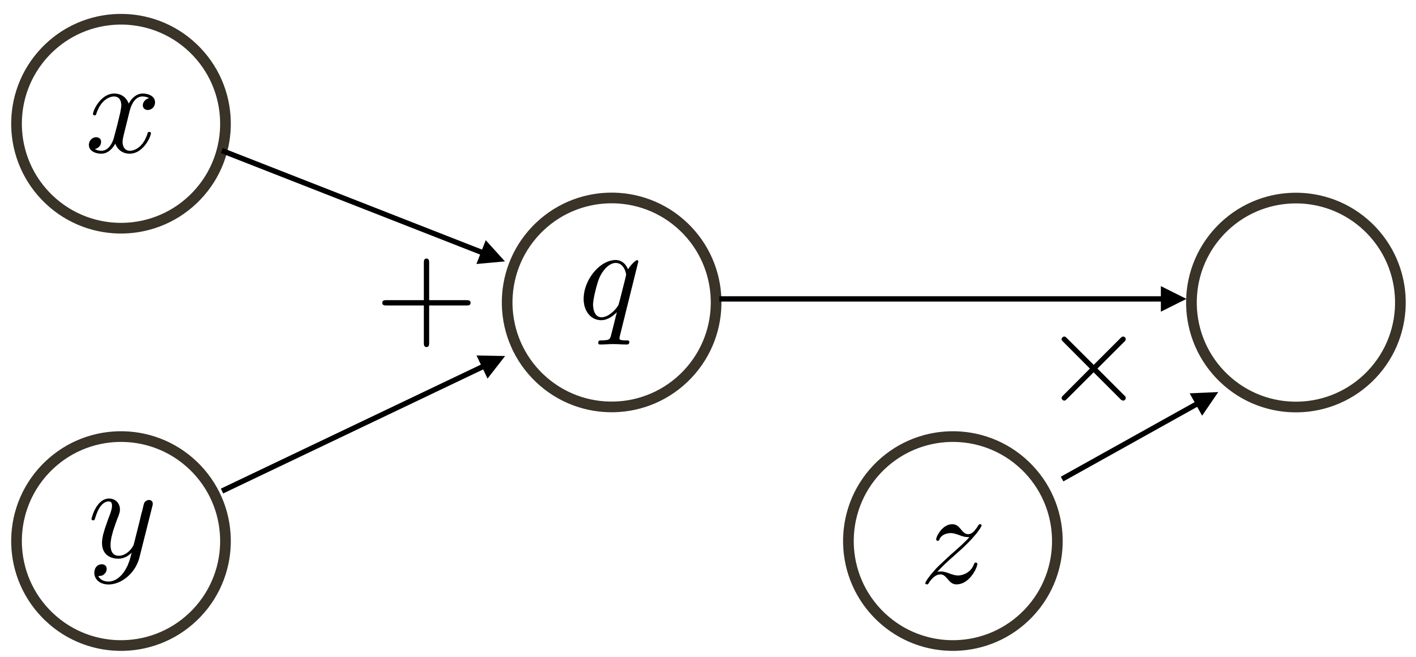
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = z \cdot 1 = z \qquad \frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = z \cdot 1 = z \qquad \frac{\partial f}{\partial z} = q$$

# Backpropagation

$$f(x, y, z) = (x + y)z$$

# Backpropagation

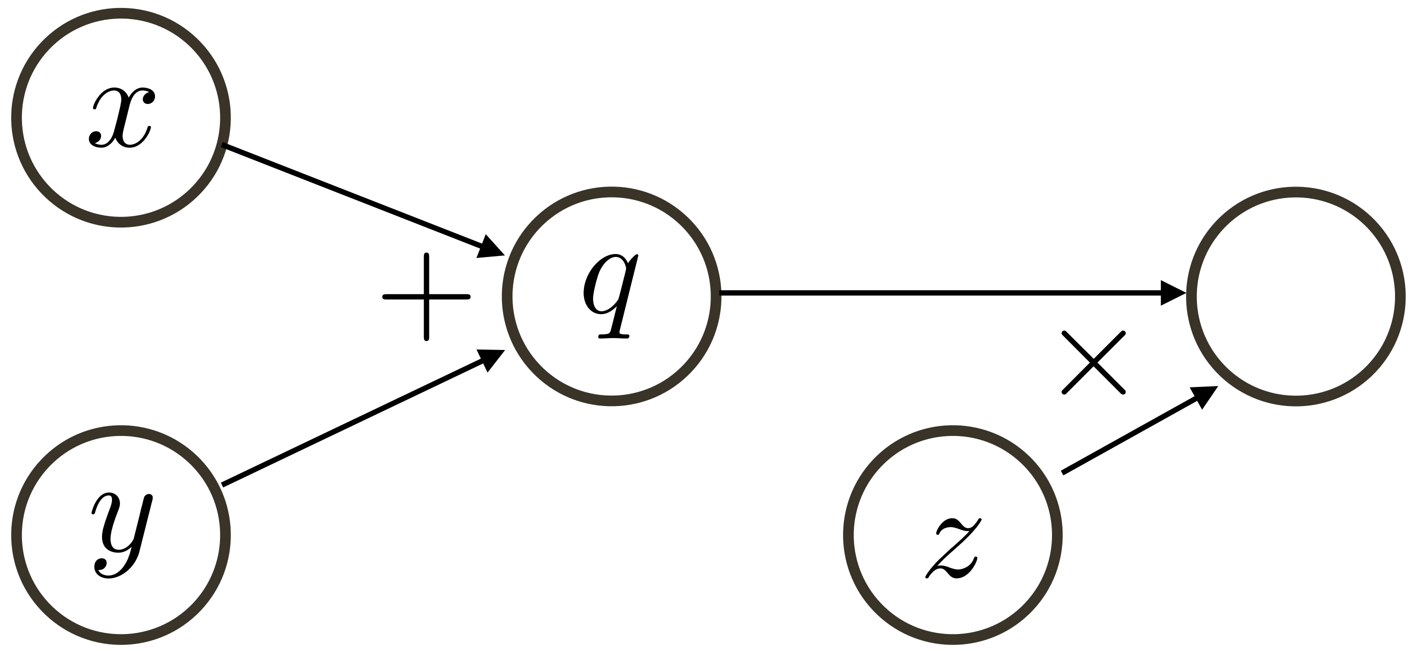
$$f(x, y, z) = (x + y)z$$




**Computational graph** (a DAG) with variable ordering from topological sort, where each **node** is an input, intermediate, or output variable

# Backpropagation

$$f(x,y,z) = (x+y)z$$



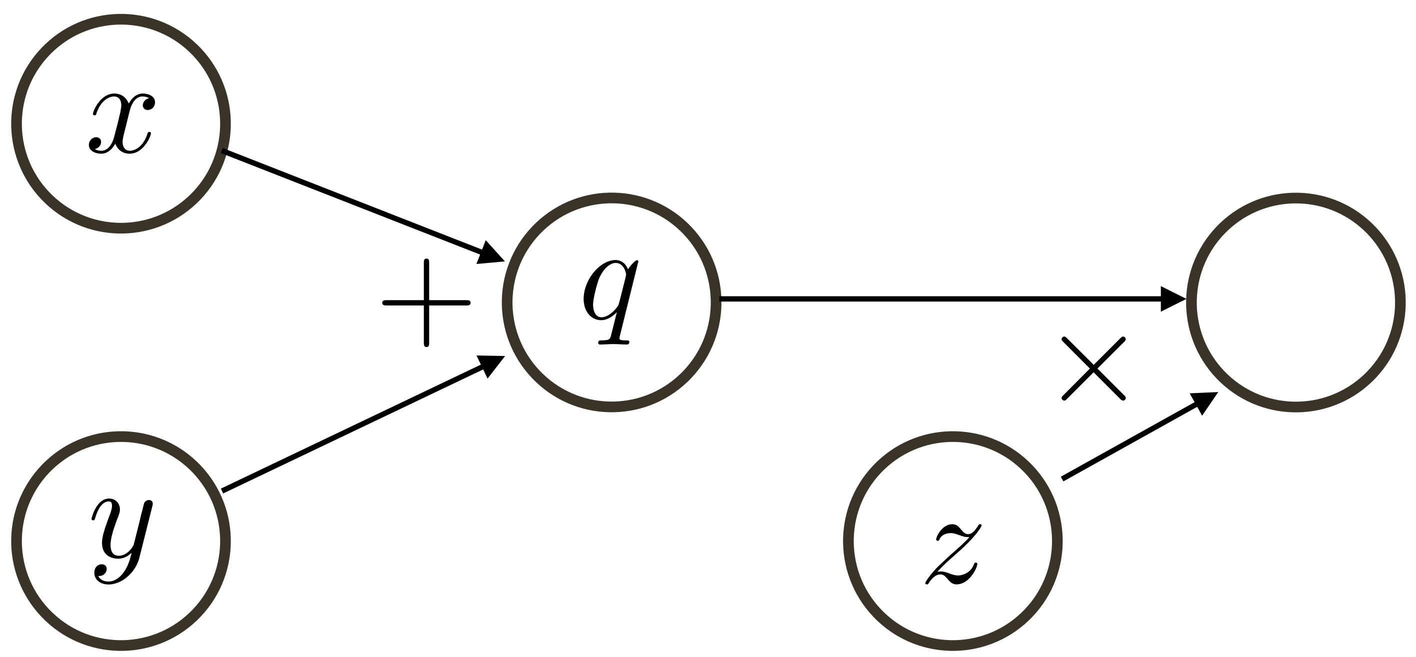

**Computational graph** (a DAG) with variable ordering from topological sort, where each **node** is an input, intermediate, or output variable

Suppose the network input is: $(x,y,z) = (-2, 5, -4)$

Then: $q = x + y = 3$ $\quad f = qz = -12$ (**forward** pass)

# Backpropagation

$$f(x, y, z) = (x + y)z$$



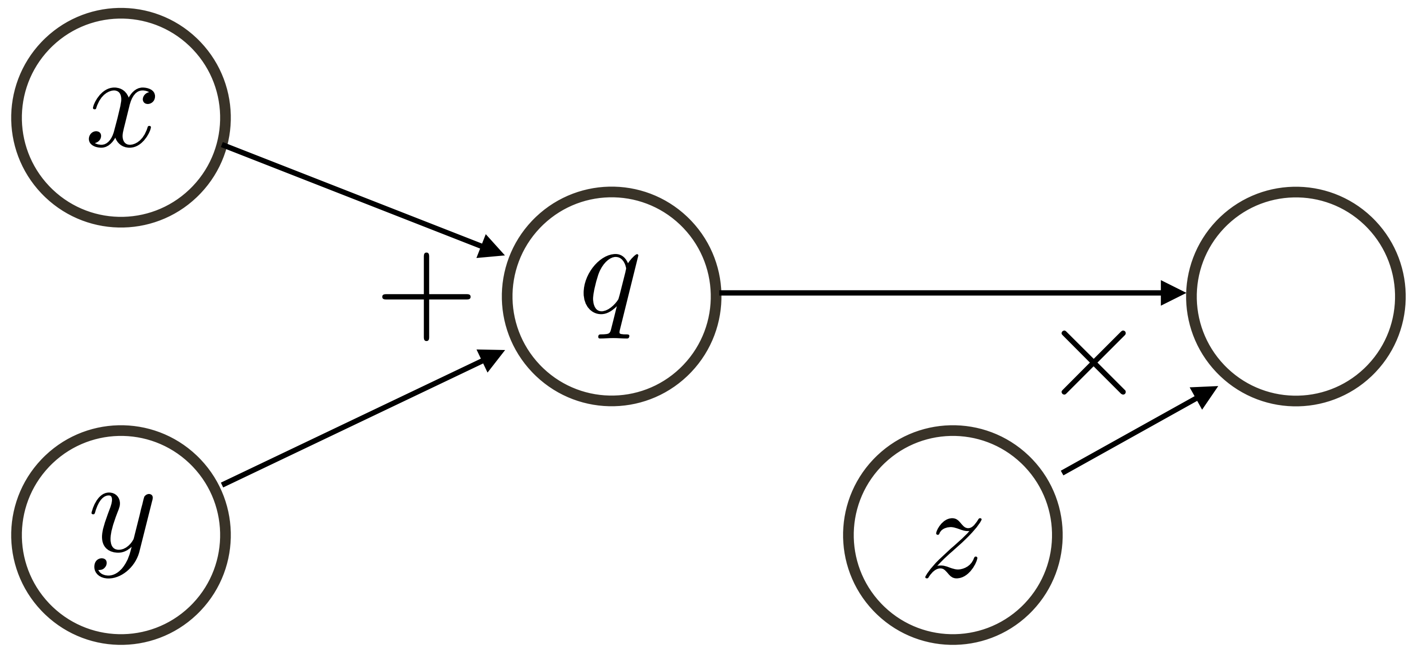

Suppose the network input is: $(x, y, z) = (-2, 5, -4)$

Then: $q = x + y = 3$ $\quad f = qz = -12$ (**forward** pass)

$$\frac{\partial f}{\partial q} = z = -4$$

(**backward** pass)

# Backpropagation




$$f(x, y, z) = (x + y)z$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = \frac{\partial f}{\partial q} \cdot 1$$

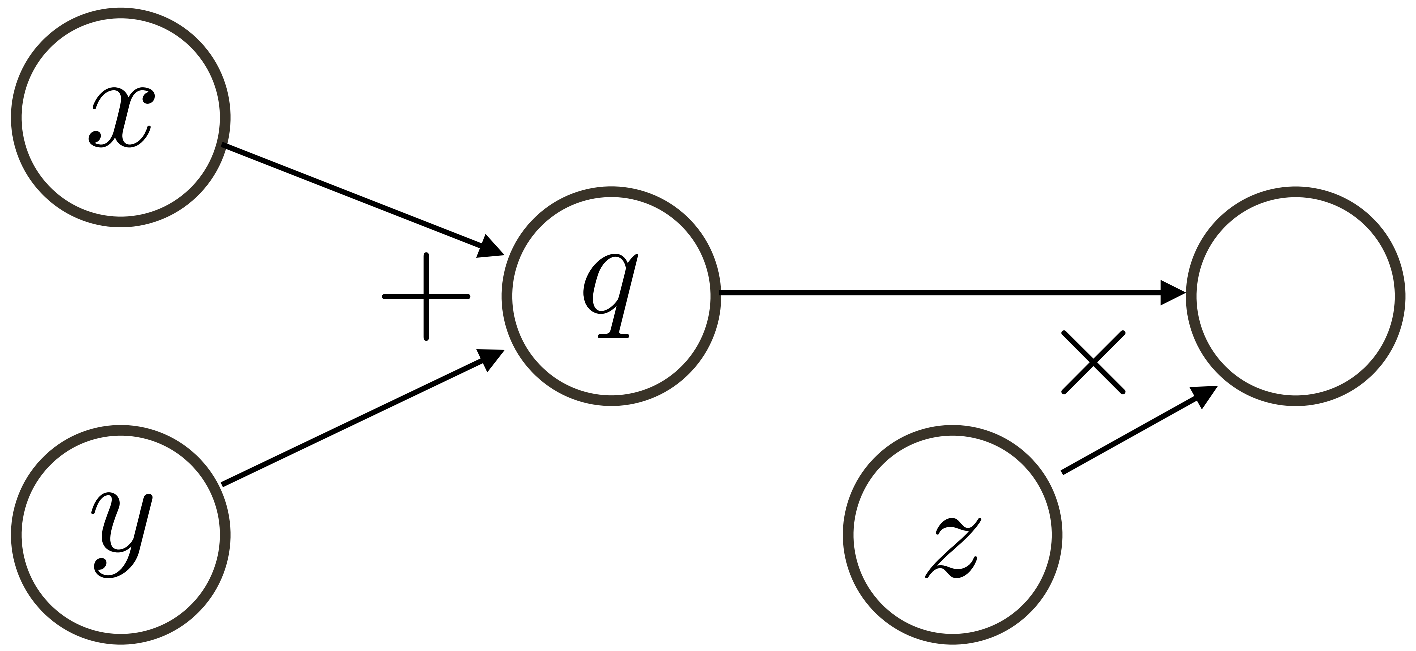
Suppose the network input is: $(x, y, z) = (-2, 5, -4)$

Then: $q = x + y = 3$ $\quad f = qz = -12$ (**forward** pass)

$$\frac{\partial f}{\partial q} = z = -4$$

(**backward** pass)

# Backpropagation




$$f(x, y, z) = (x + y)z$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = \frac{\partial f}{\partial q} \cdot 1$$

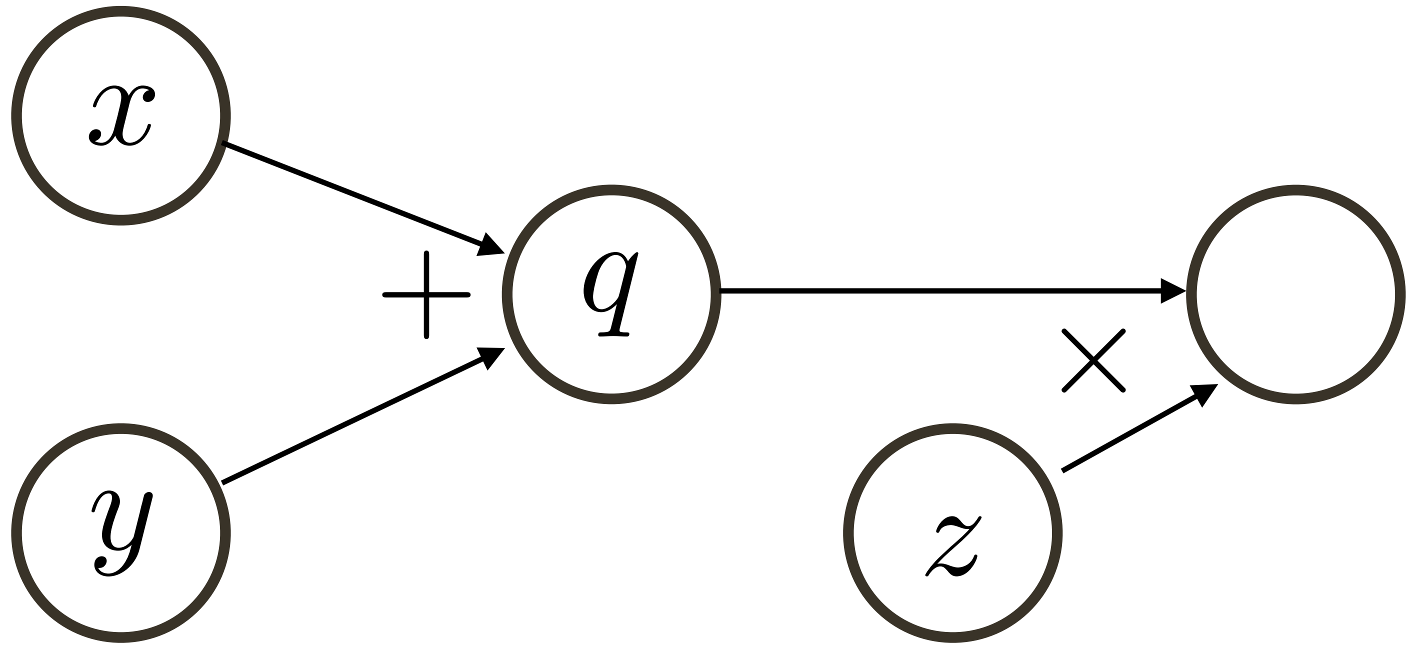
Suppose the network input is: $(x, y, z) = (-2, 5, -4)$

Then: $q = x + y = 3$ $\quad f = qz = -12$ (**forward** pass)

$$\frac{\partial f}{\partial q} = z = -4 \qquad \frac{\partial f}{\partial x} = -4$$

(**backward** pass)

# Backpropagation




$$f(x,y,z) = (x+y)z$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = \frac{\partial f}{\partial q} \cdot 1 \qquad \frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = \frac{\partial f}{\partial q} \cdot 1 \qquad \frac{\partial f}{\partial z} = q$$

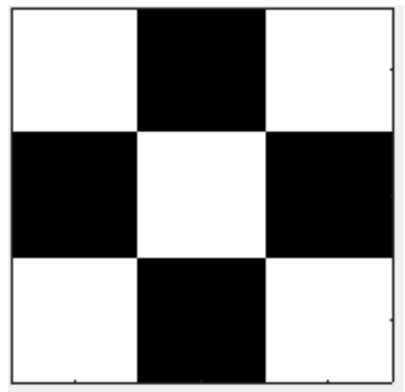
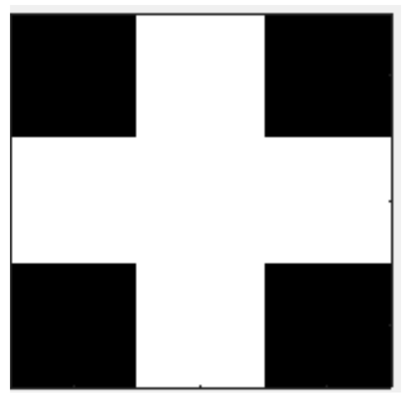
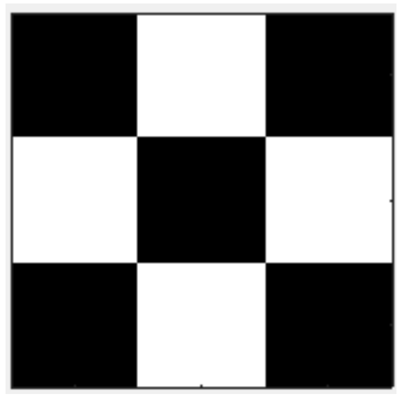
Suppose the network input is: $(x,y,z) = (-2, 5, -4)$

Then: $q = x + y = 3 \qquad f = qz = -12$ (**forward** pass)

$$\frac{\partial f}{\partial q} = z = -4 \qquad \frac{\partial f}{\partial x} = -4 \qquad \frac{\partial f}{\partial y} = -4 \qquad \frac{\partial f}{\partial z} = 3$$ (**backward** pass)

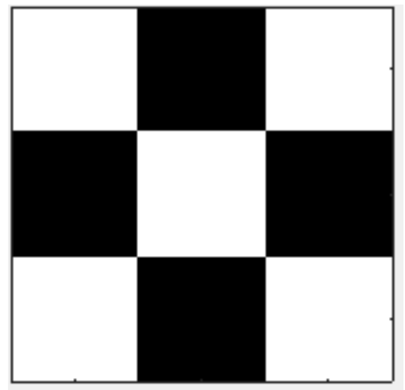
# **Example**: Let's Build (world smallest) Neural Network

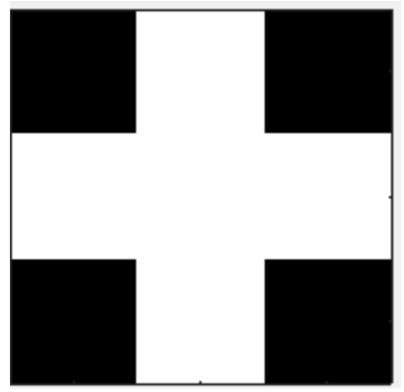
Lets create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images

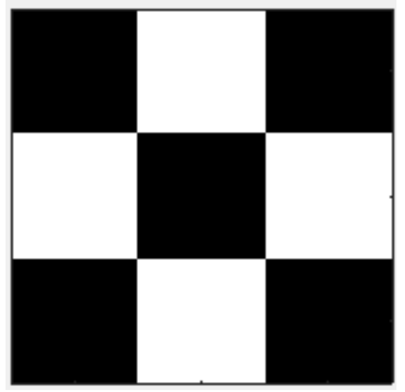




# **Example**: Let's Build (world smallest) Neural Network

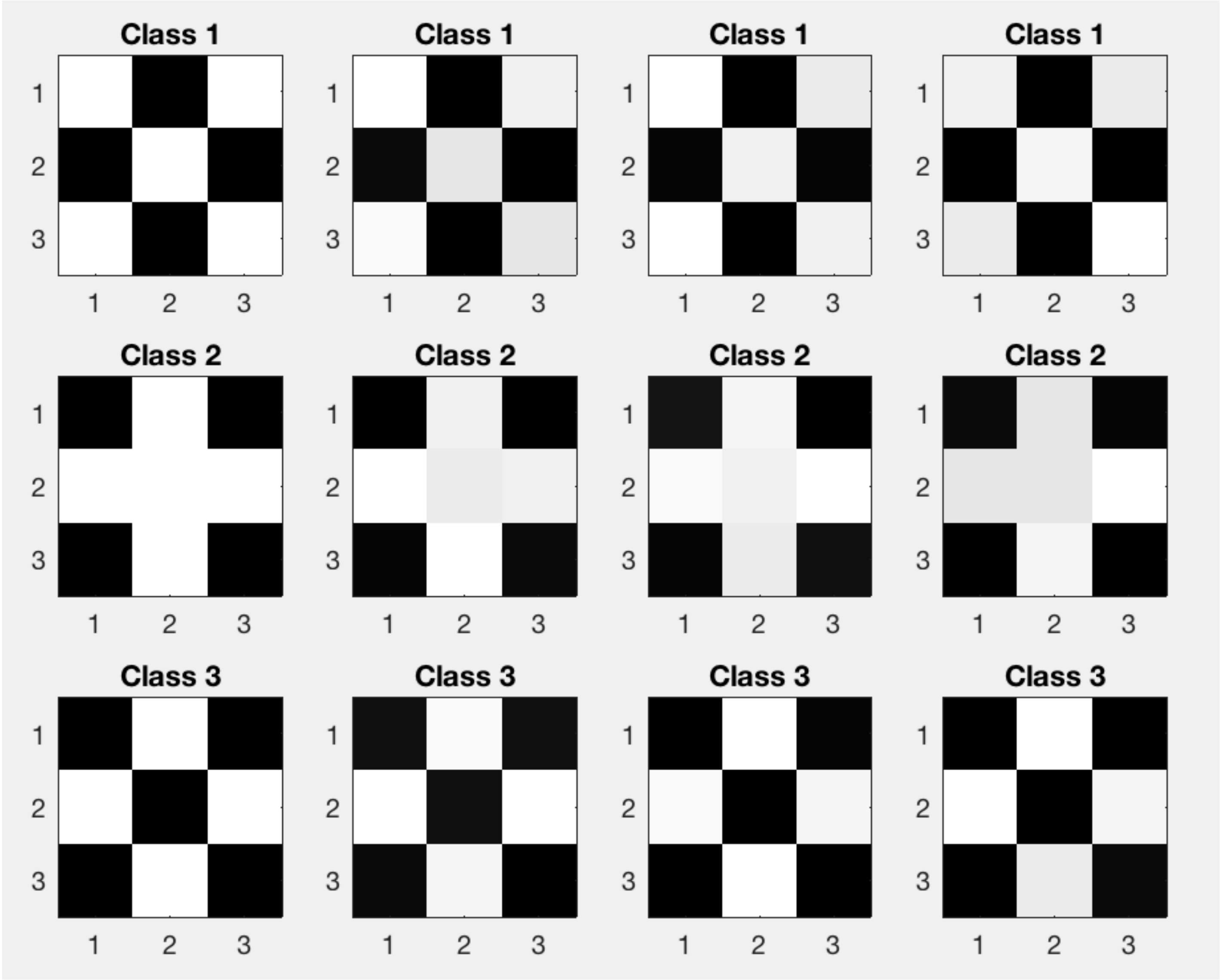
Lets create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images





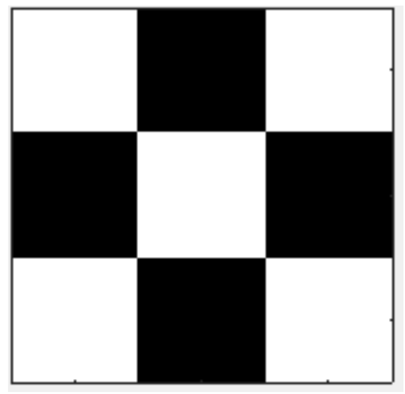


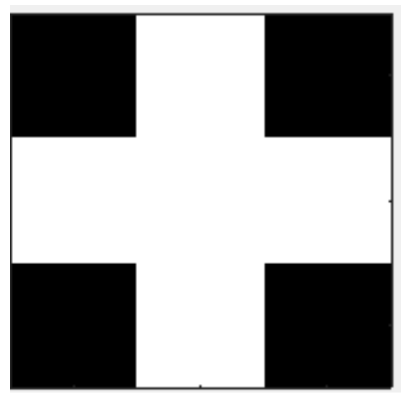
We will need some labeled data

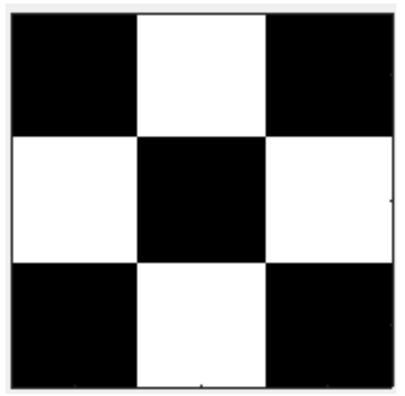


# **Example**: Let’s Build (world smallest) Neural Network

Lets create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images

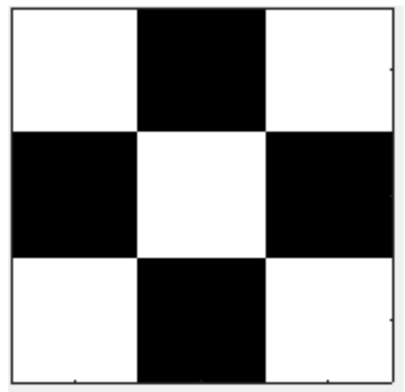
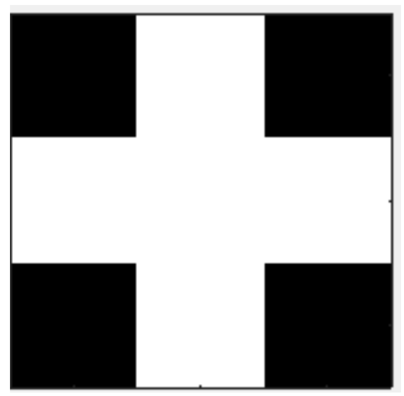
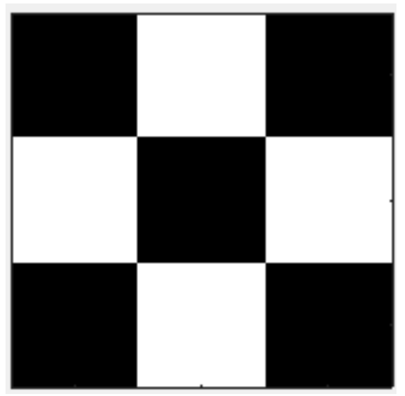






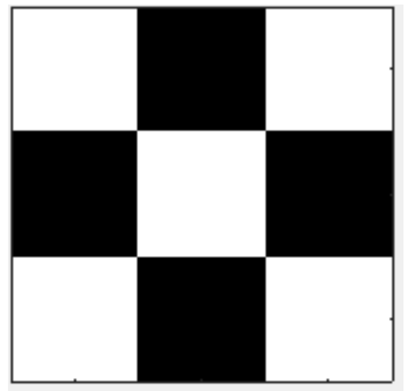
# **Example**: Let’s Build (world smallest) Neural Network

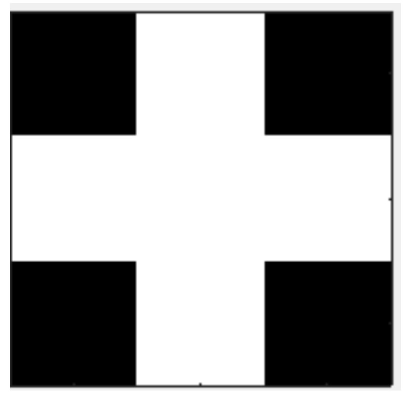
Lets create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images

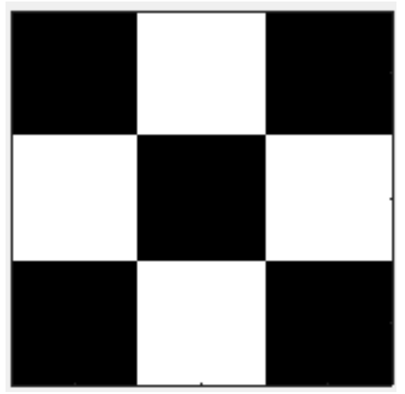




# **Example**: Let’s Build (world smallest) Neural Network

Lets create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images

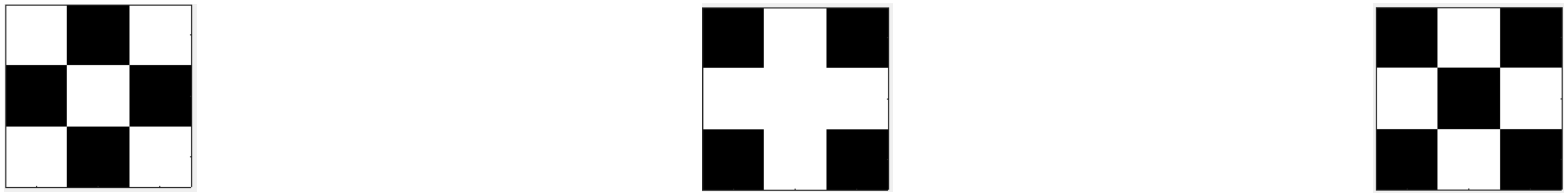






# **Example**: Let’s Build (world smallest) Neural Network

Lets create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images

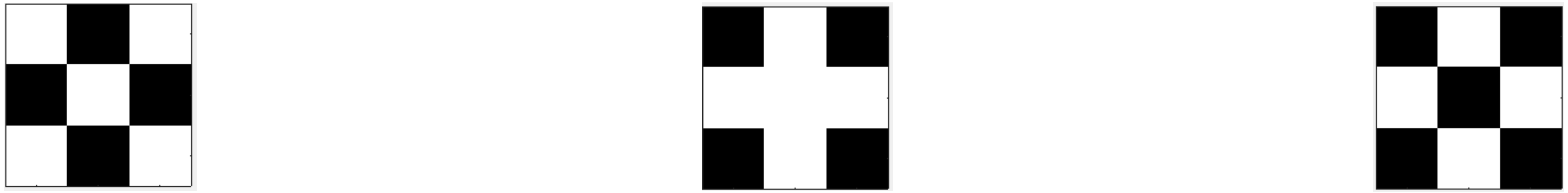


What do we need to do?


First, lets re-formulate the problem

# **Example**: Let’s Build (world smallest) Neural Network

Lets create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images

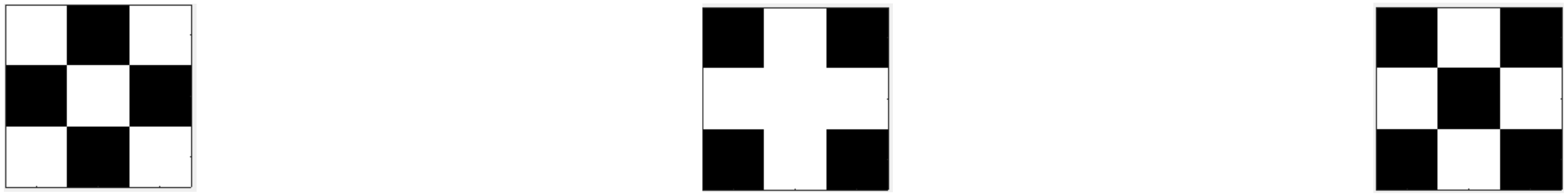


What do we need to do?


First, lets re-formulate the problem

# **Example**: Let’s Build (world smallest) Neural Network

Lets create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images

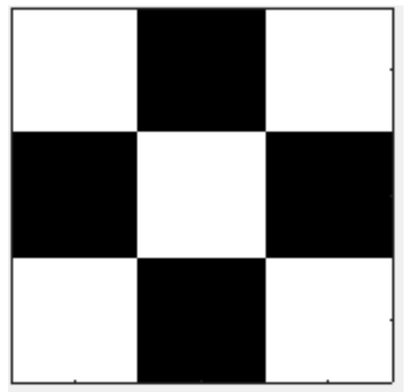


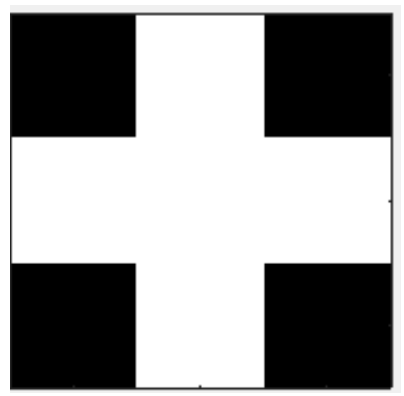
Now, lets build a **network**!

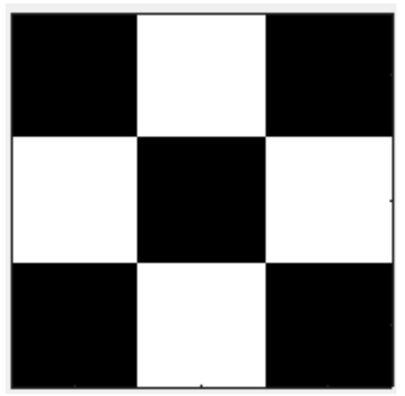

How many inputs should the network have? How neuron outputs?

# **Example**: Let's Build (world smallest) Neural Network

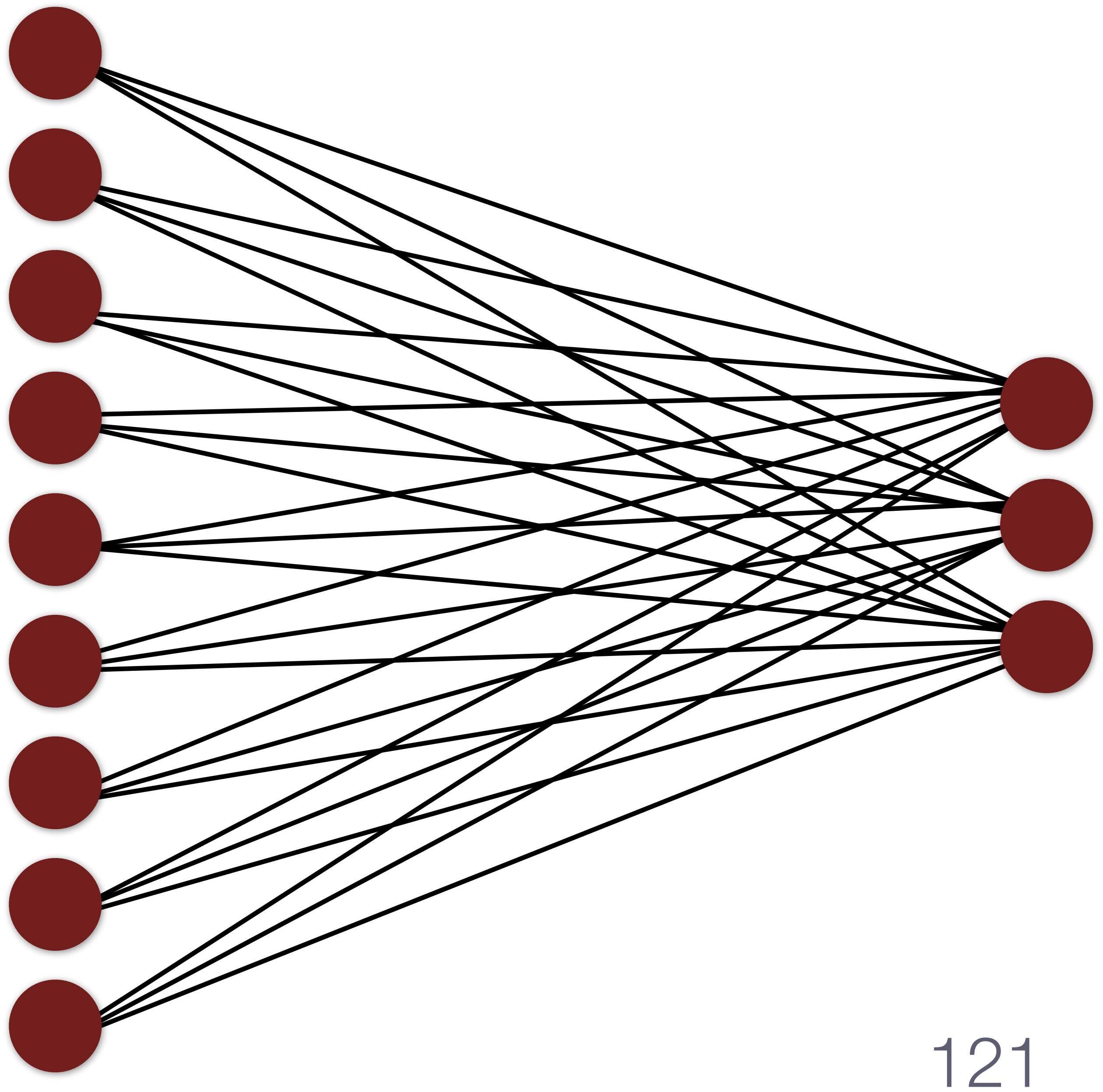
Lets create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images







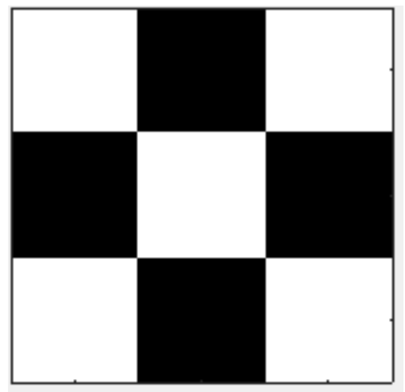
**Input** Layer **Output** Layer

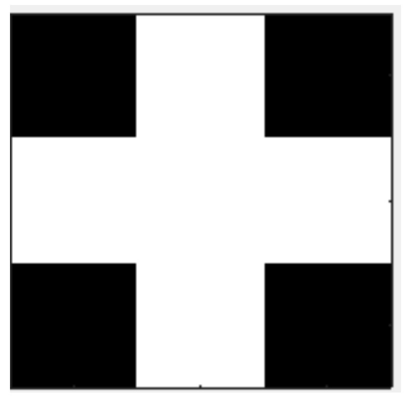


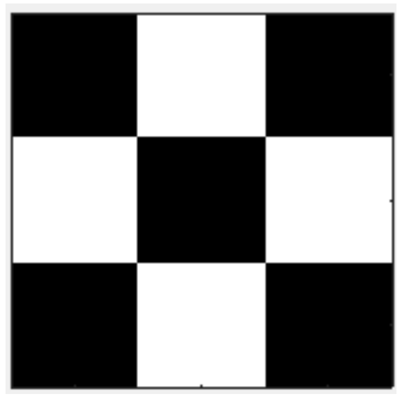
What else is missing for us to train it?

# **Example**: Let's Build (world smallest) Neural Network

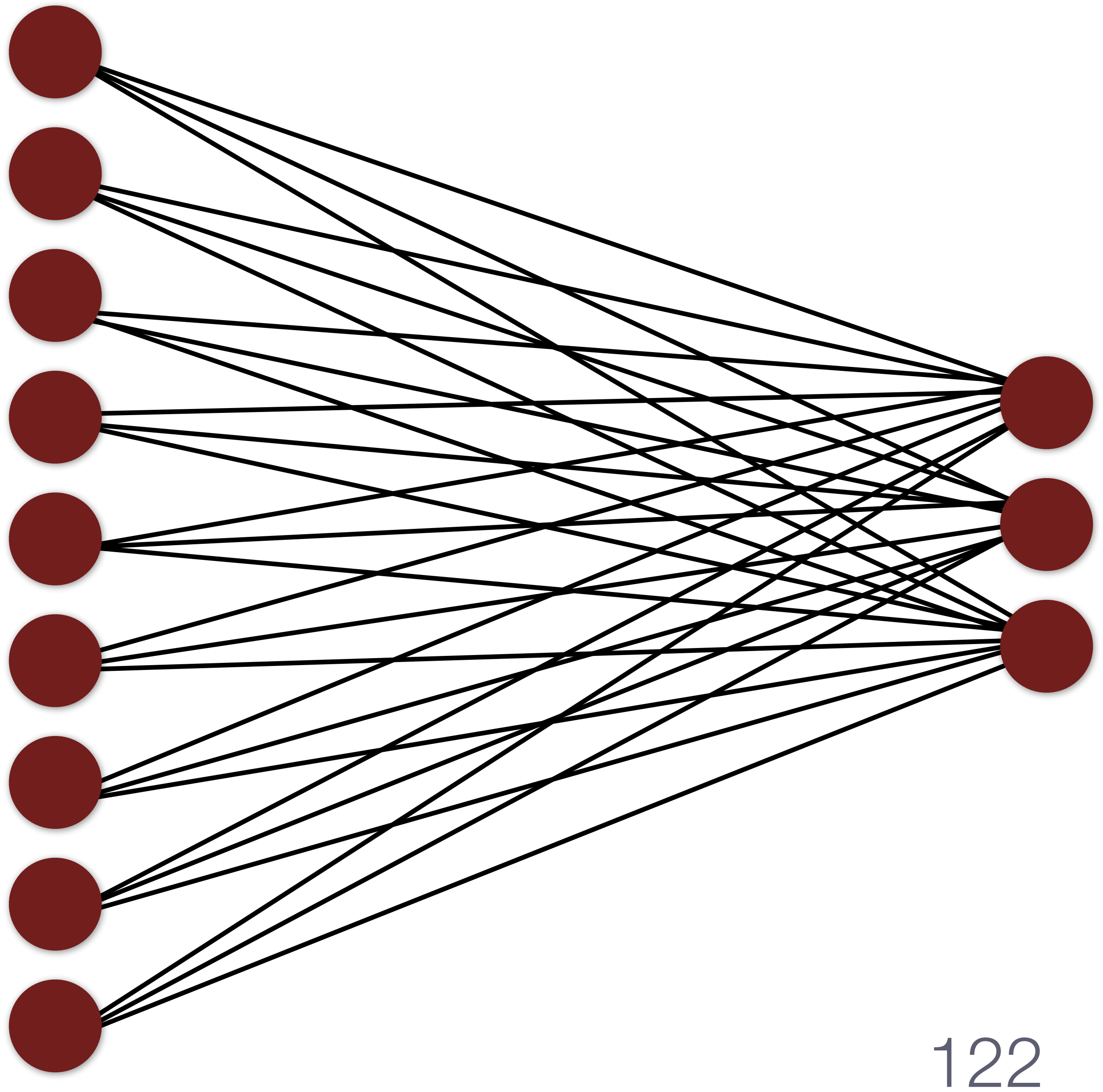
Lets create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images







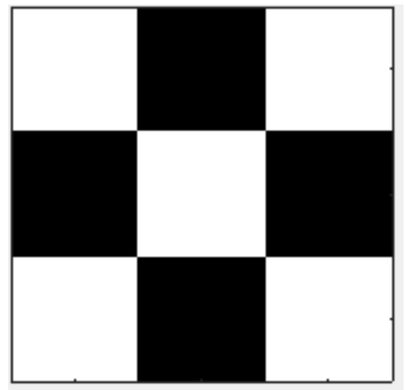
**Input** Layer **Output** Layer **Loss**

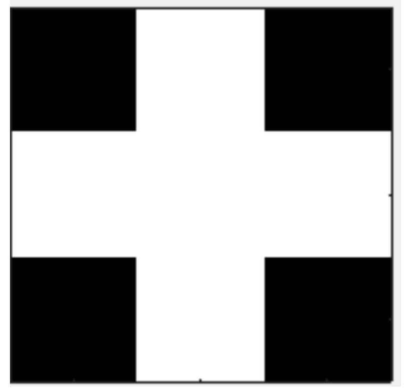


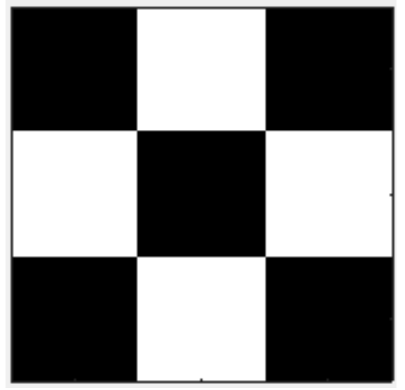
$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_{y_j}}}\right)$$

# **Example**: Let’s Build (world smallest) Neural Network

Lets create a neural network that will be able to differentiate (classify) these patterns of simple 3x3 pixel images



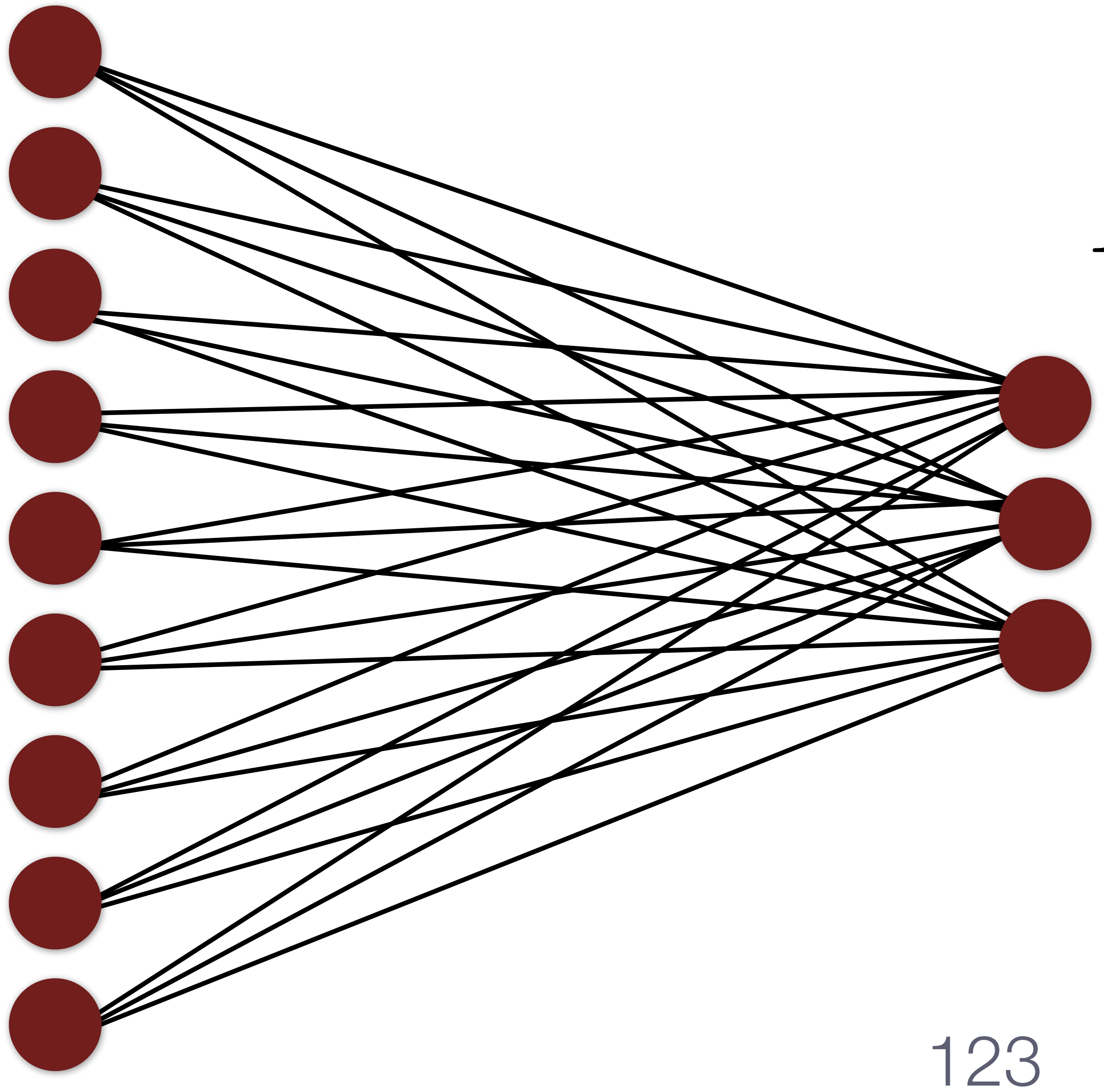




**Input** Layer

**Output** Layer

**Loss**



$$L_1 = -log\left(\frac{e^{\sum_{i=1}^{9}\sigma(w_{1,i}x_i+b_1)}}{\sum_{j=1}^{3} e^{\sum_{i=1}^{9}\sigma(w_{1,i}x_i+b_1)}}\right)$$