Lecture 5: Neural Networks

Machine Learning (BBWL)

Michael Mommert, University of St. Gallen

Today's lecture

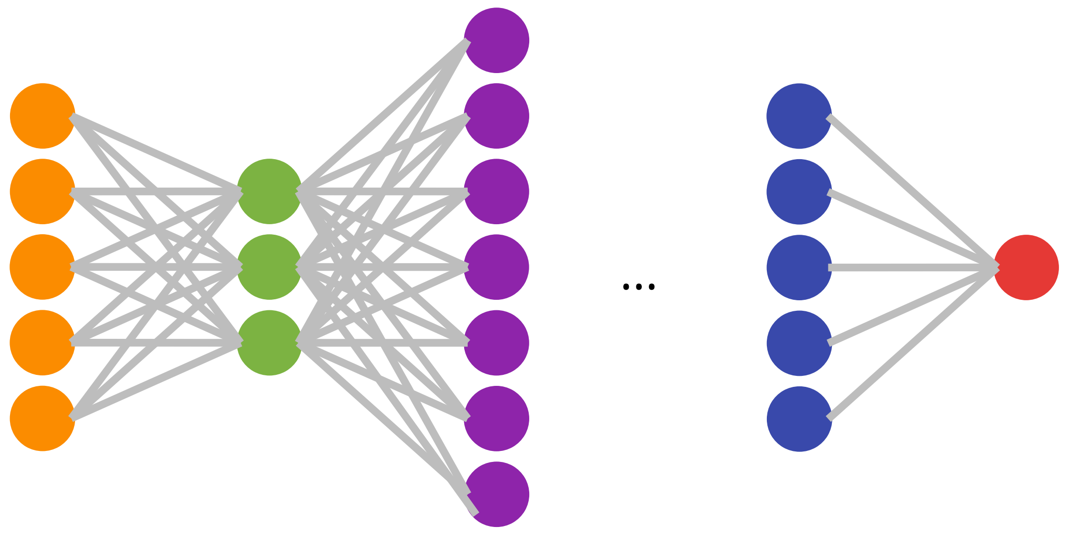
Neurons and Neural Networks

Activation Functions

Loss functions,
Backpropagation and
Gradient Descent

Neural Network Training
and Evaluation

Neurons and Neural Networks



Biological learning process

How do organisms learn?

- Observation and imitation
- Trial and error
- Study (e.g., from literature)
- Cannibalism (only for some worms)

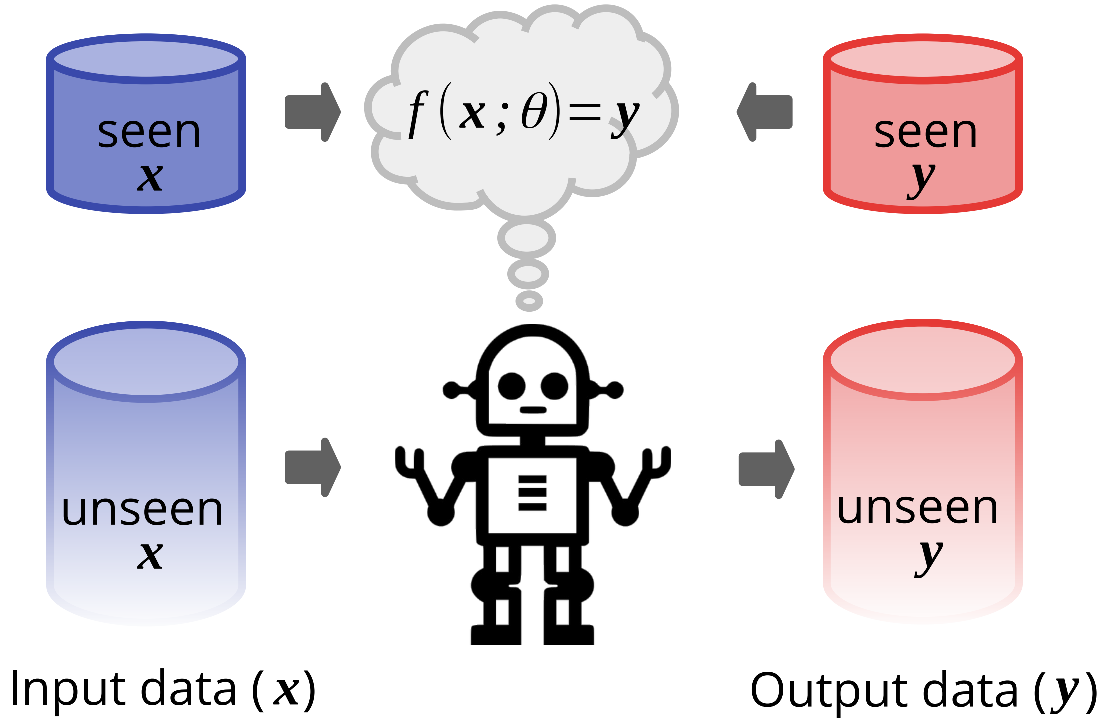
Getty Images via dailymail.co.uk

BBC

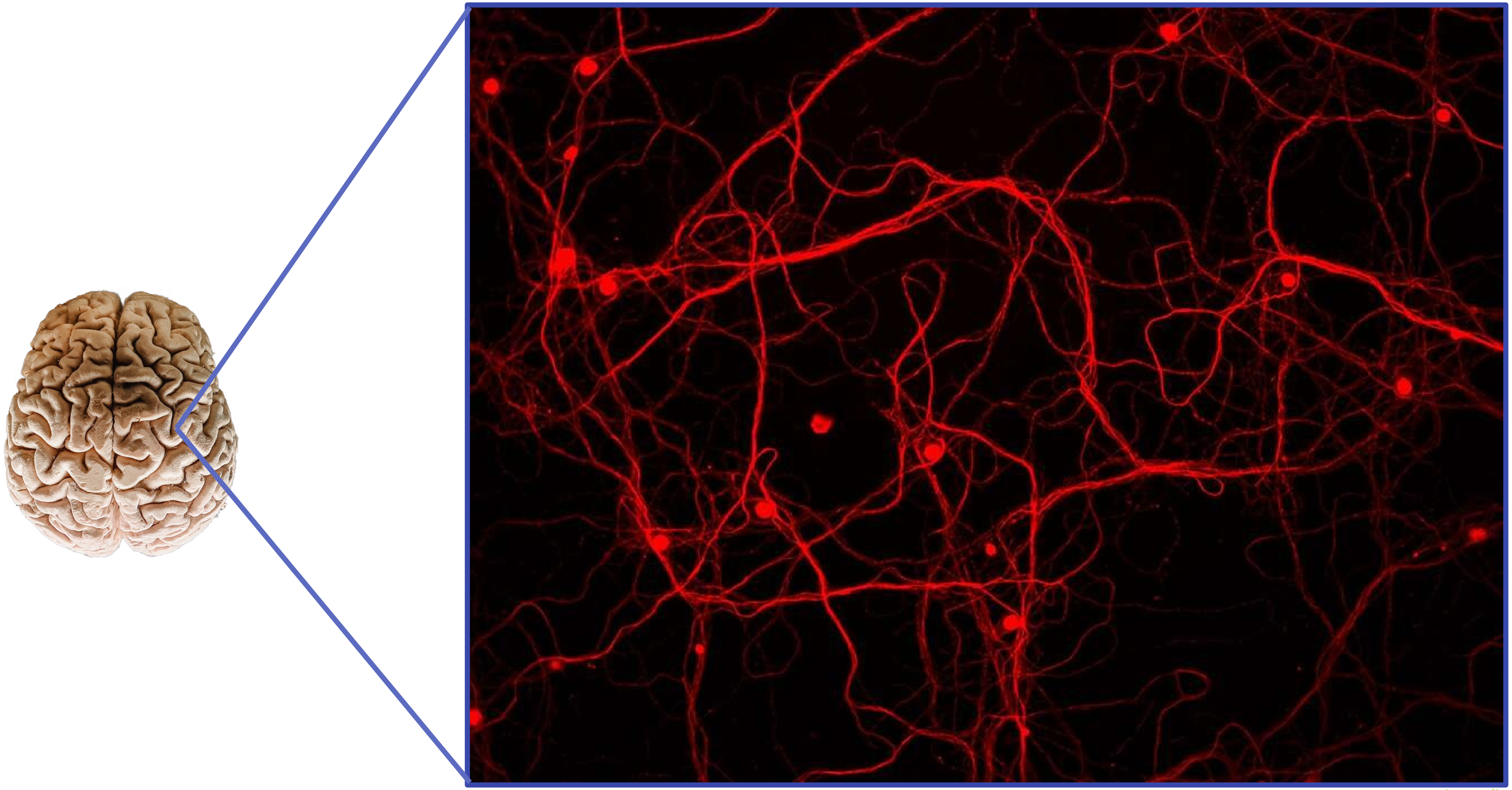
Biological learning is an iterative process. Can we synthesize this process?

Machine learning so far…

- Have the models we talked about so far really "learned " anything?
    - *k*-NN computes distances and compares distribution of unseen data points with distribution of seen data points
    - Linear models are fitted to seen data based and the task
    - Tree-based models identify and memorize patterns relevant to the task
- Most traditional Machine Learning methods do not learn in an iterative process


seen $\boldsymbol{x}$
$f(\boldsymbol{x};\theta)=\boldsymbol{y}$
seen $\boldsymbol{y}$
unseen $\boldsymbol{x}$
unseen $\boldsymbol{y}$
Input data ($\boldsymbol{x}$)
Output data ($\boldsymbol{y}$)

How does the human brain work?



micropedia

How does the human brain work?

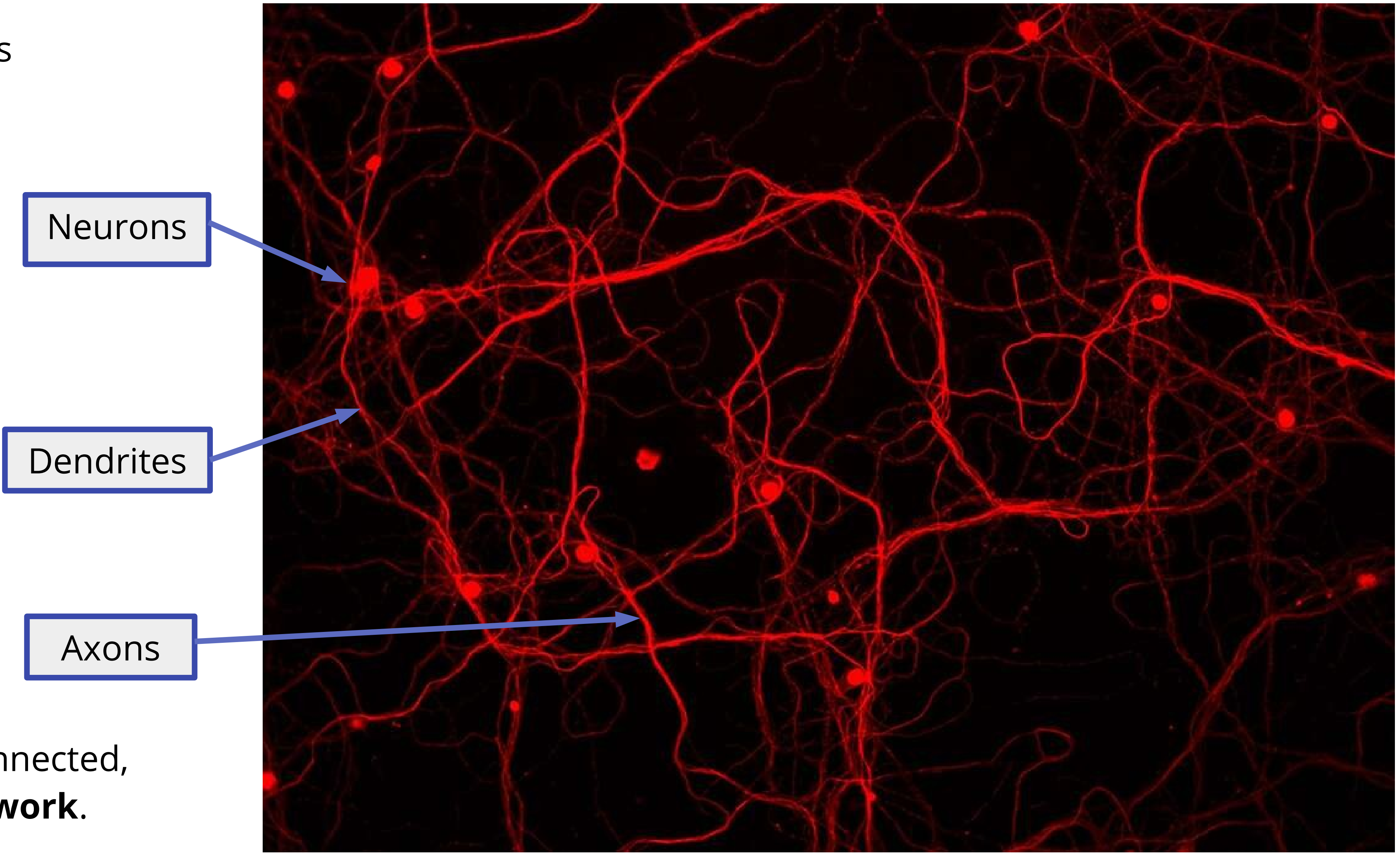
Human brain contains ~10^9 neurons

Neurons are nerve cells that process electrical signals

Dendrites connect nearby neurons and enable signal exchange between them

Axons provide long-distance connections

Neurons are inter-connected, forming a **dense network**.



micropedia

How does the human brain work?

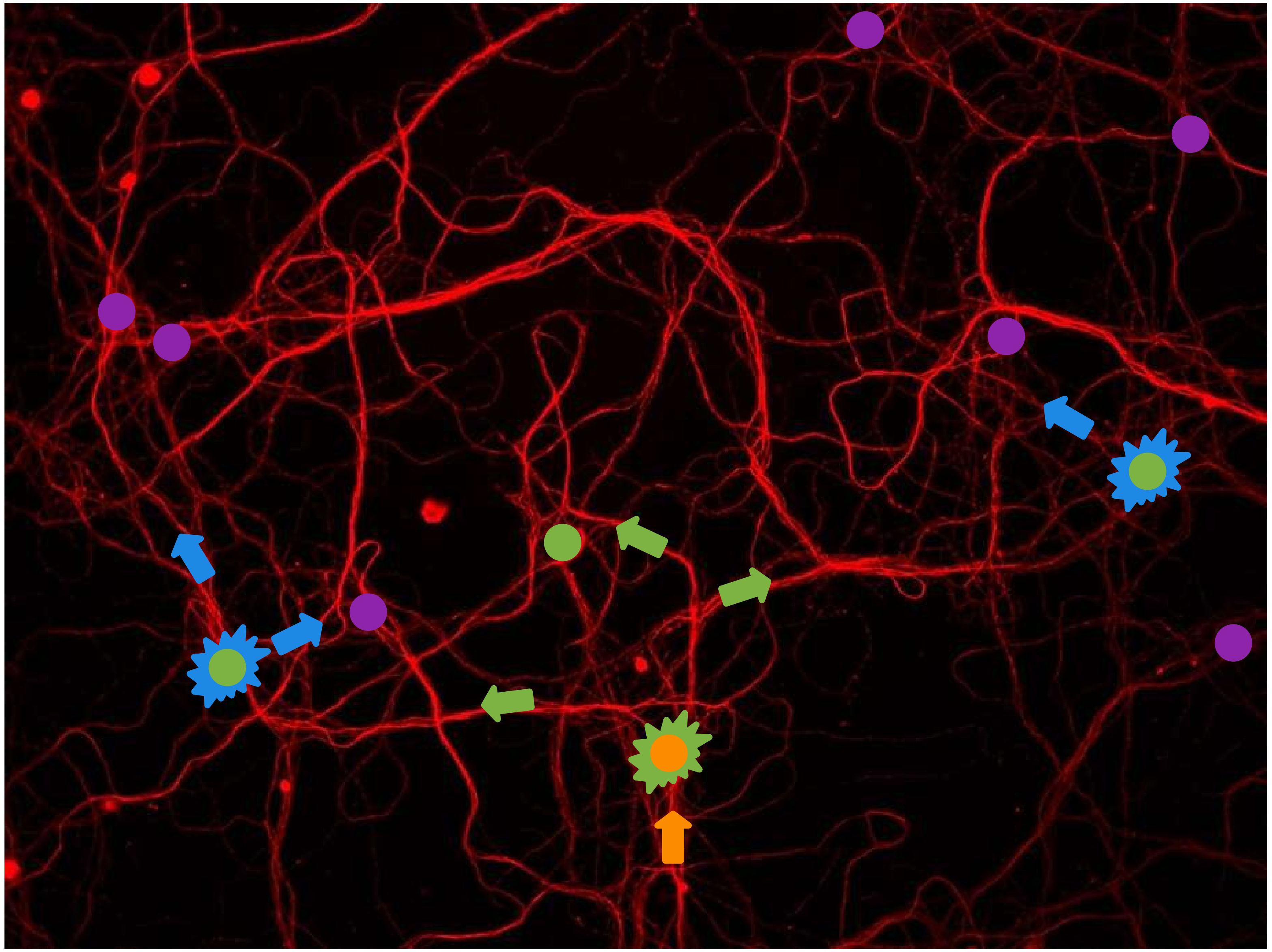
Information is passed to neurons through electrical signals.

The first neuron absorbs and processes the signal, leading it to "**fire**": a new signal is created and sent to all connected neurons.

The new signals are passed on to all connected neurons.

Connected neurons absorb the incoming signals and process them. Some of them will fire, but not all.

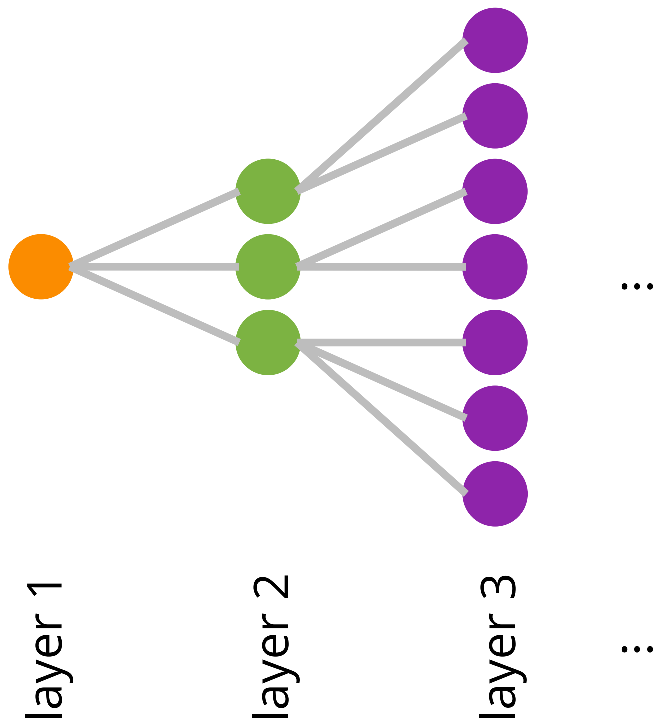
New signals are created by those neurons that fire, creating a **cascade of signals**.

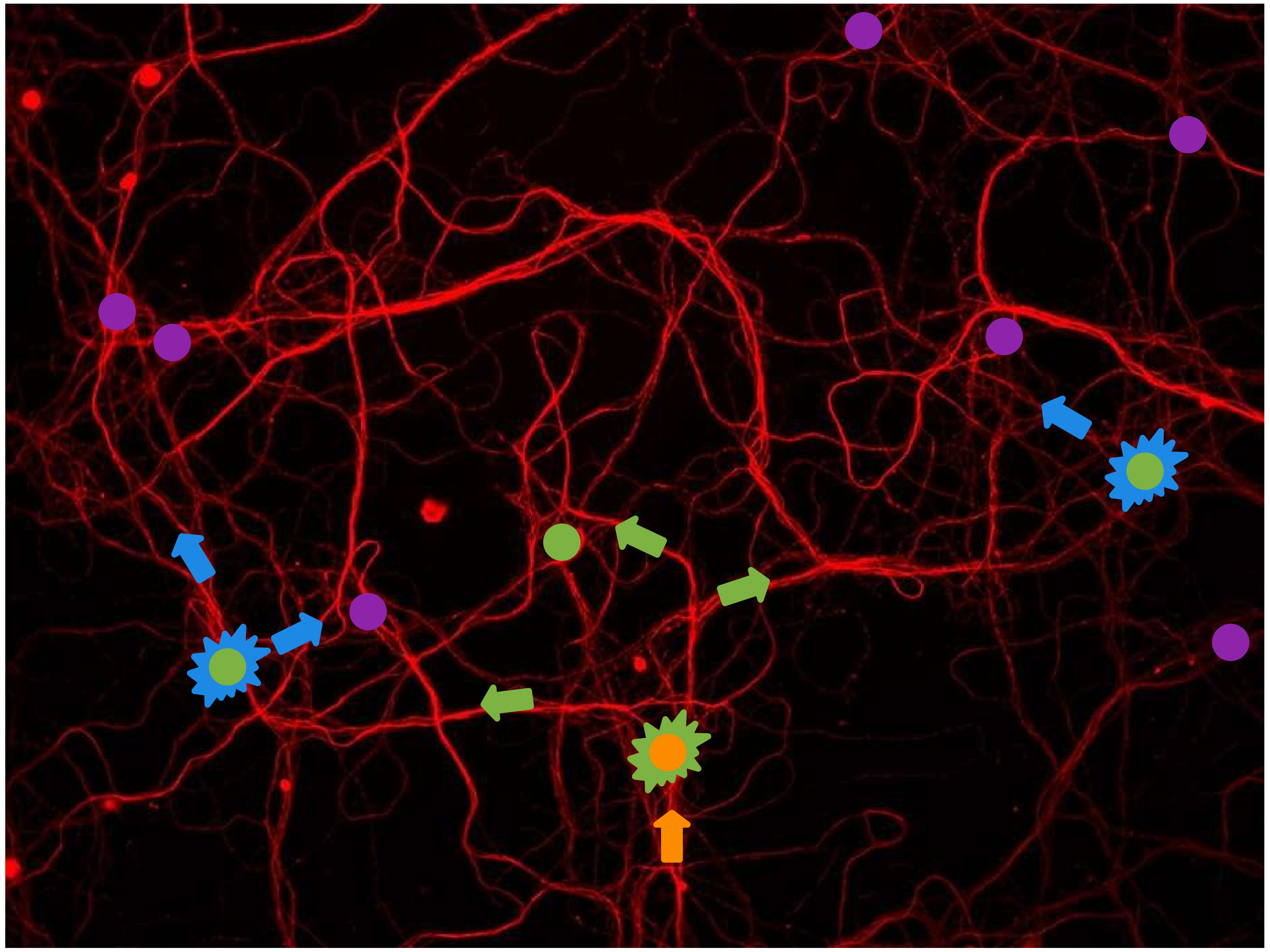


micropedia

How does the human brain work?

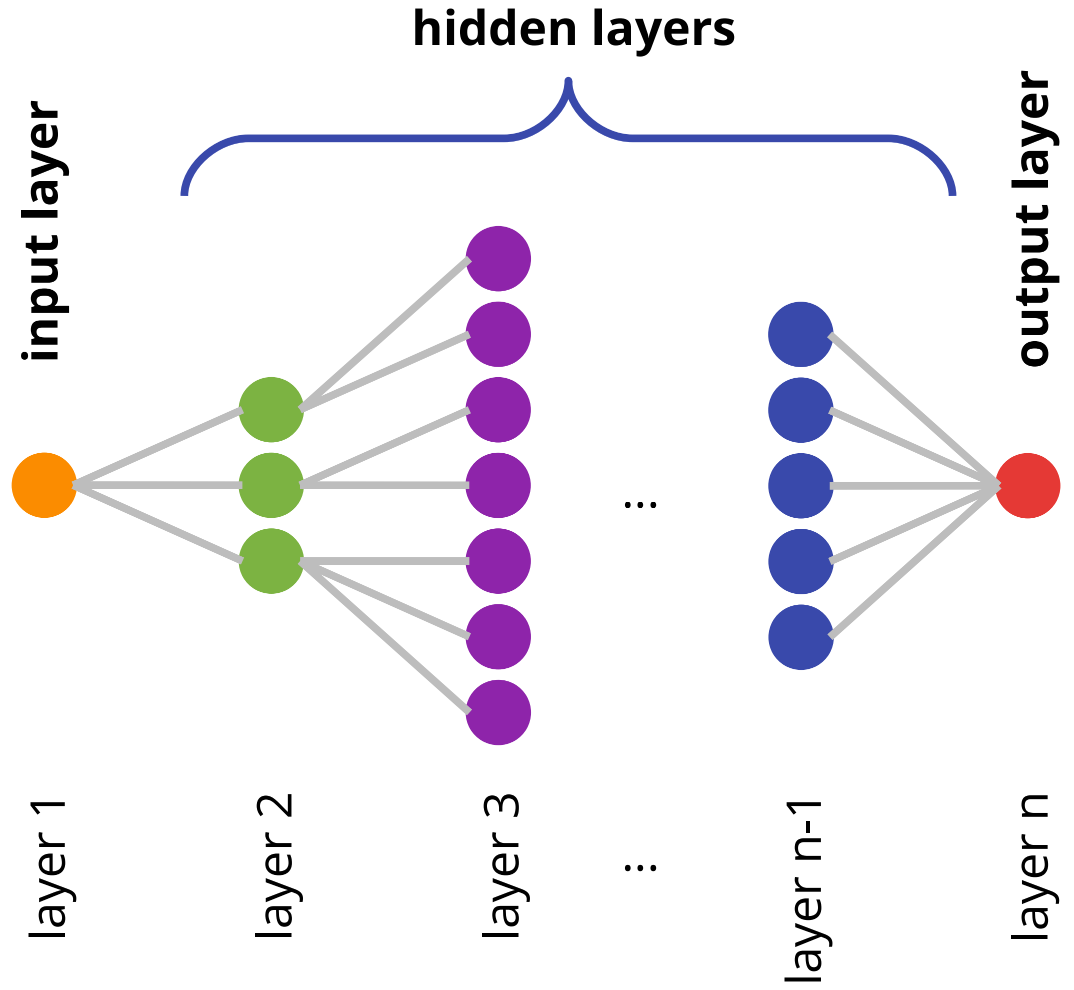
We can re-arrange our **neural network** into a graph representation:





micropedia

Neural network



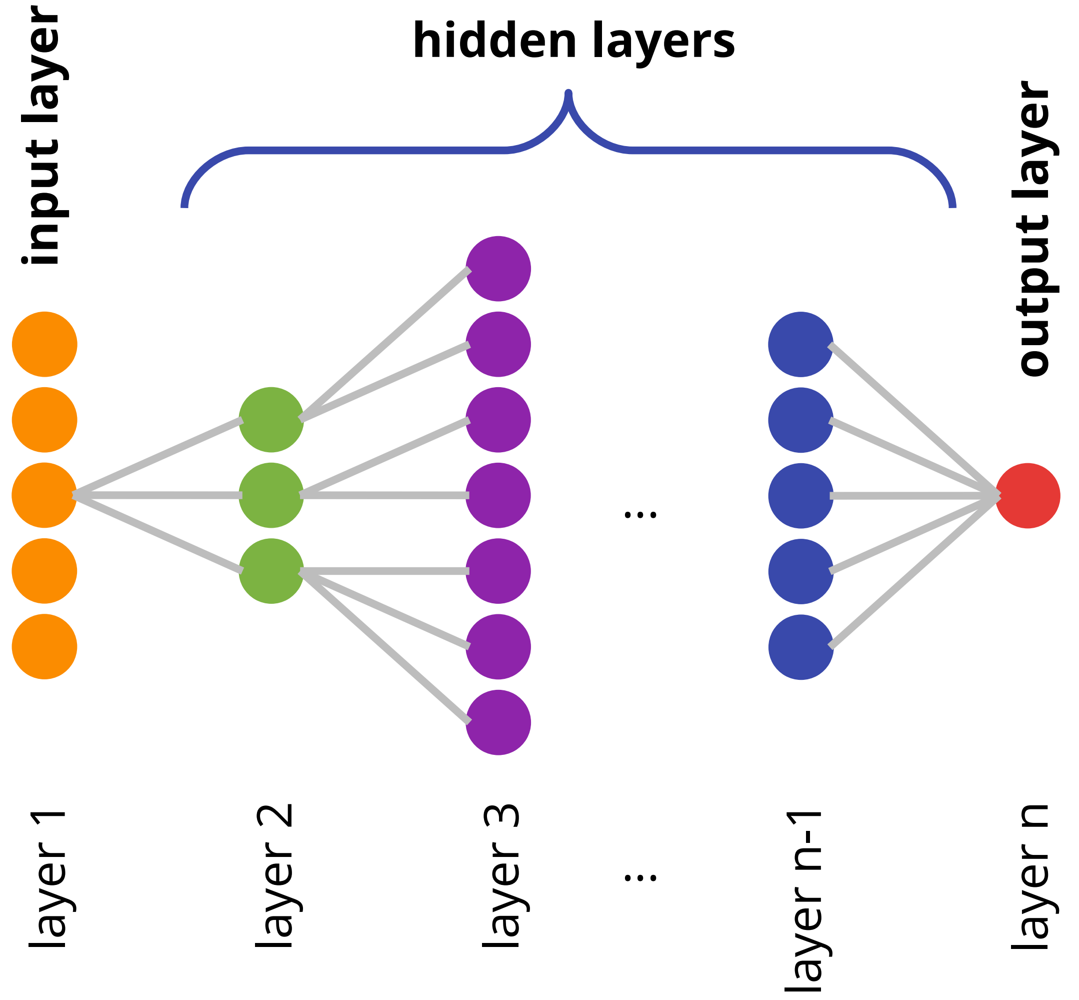
This is a neural network: a **deep cascade** of layers of neurons.

So that the network can learn a meaningful task, it requires **input data**, which it processes in its **hidden layers**, and generates **output data**.

We can further generalize this model:

1

Neural network



This is a neural network: a **deep cascade** of layers of neurons.

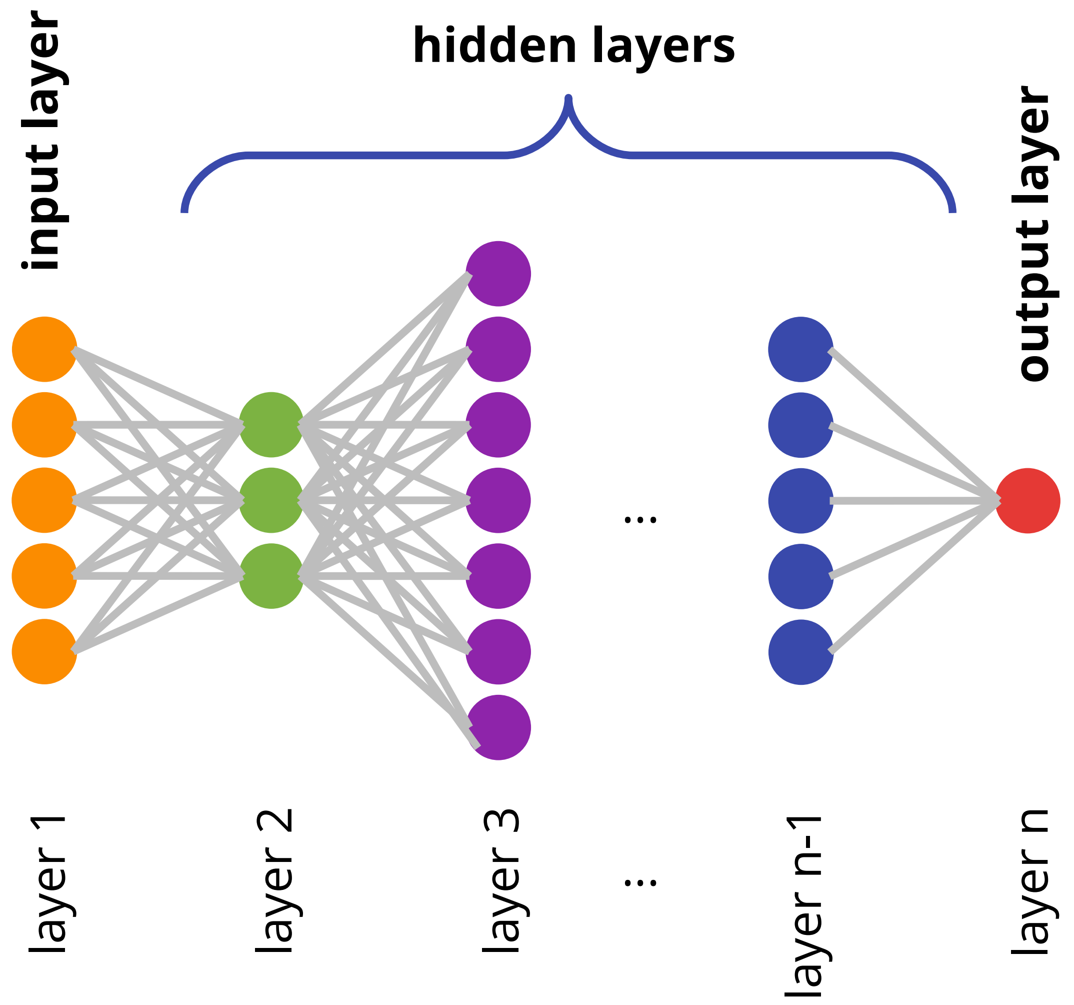
So that the network can learn a meaningful task, it requires **input data**, which it processes in its **hidden layers**, and generates **output data**.

We can further generalize this model:

- Instead of a single input neuron (single input value), we can use more input neurons to support more complex input data.

1

Neural network



This is a neural network: a **deep cascade** of layers of neurons.

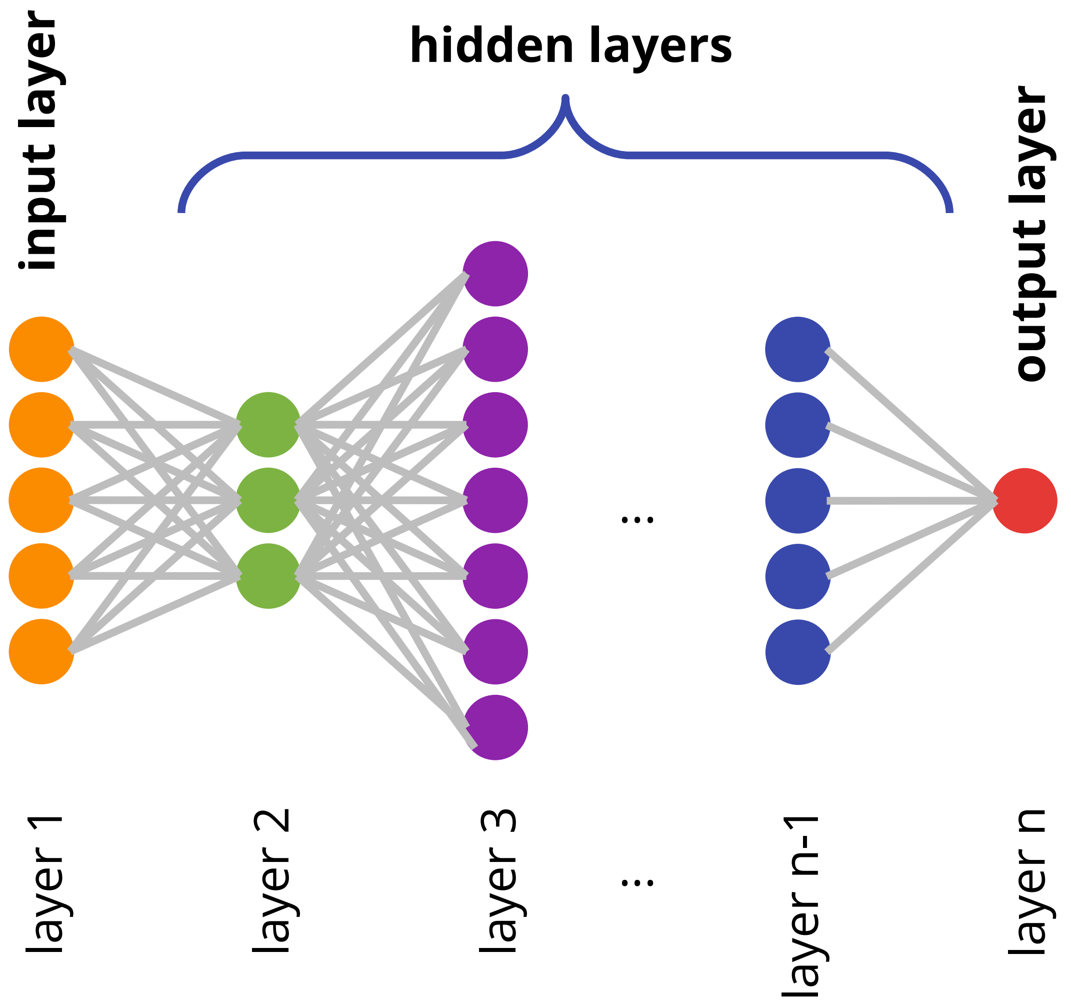
So that the network can learn a meaningful task, it requires **input data**, which it processes in its **hidden layers**, and generates **output data**.

We can further generalize this model:

- Instead of a single input neuron (single input value), we can use more input neurons to support more complex input data.
- We can connect each neuron to all neurons in the previous layers and all neurons in the following layer. This is a **fully connected network**.

1

Fully-connected neural network



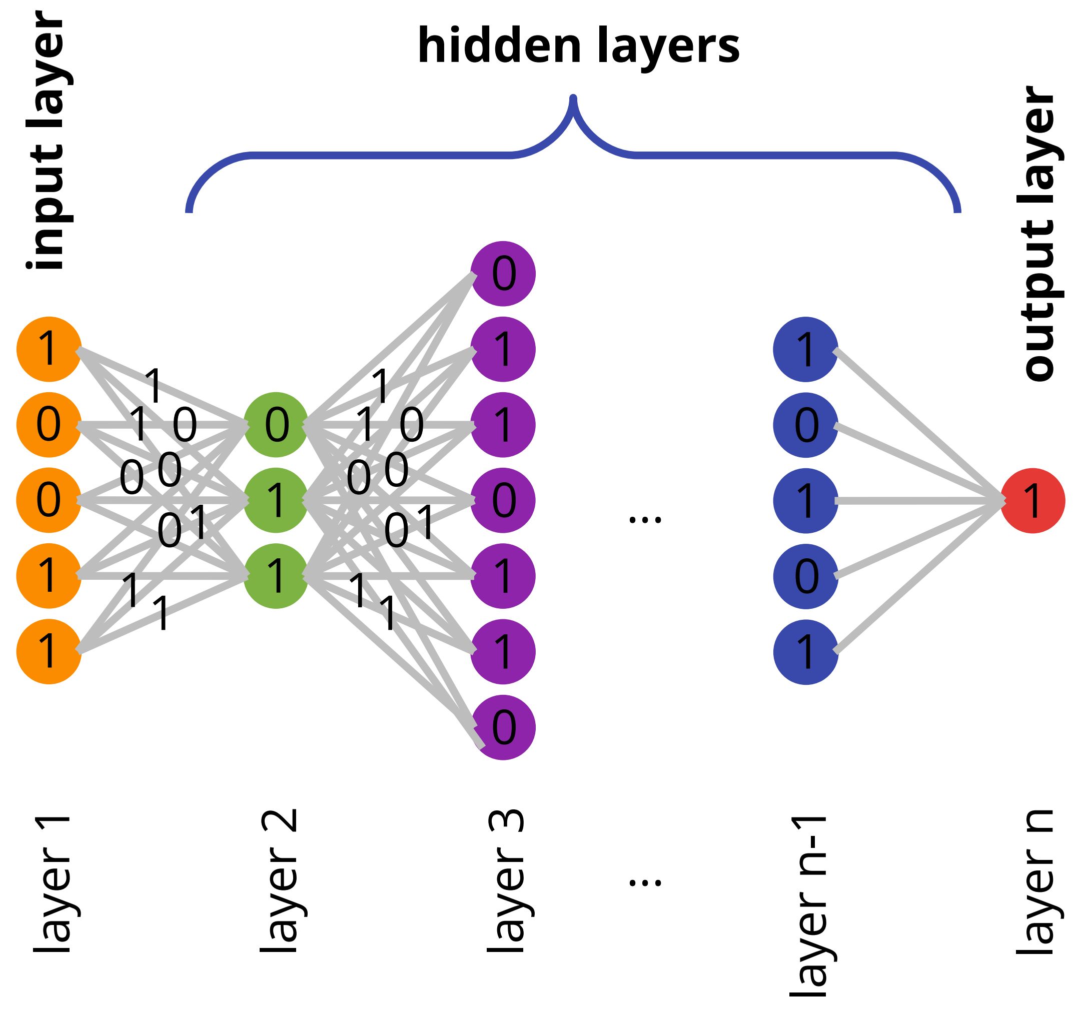
In this network type, all neurons of one layer are connect to all neurons of the previous layer and all neurons of the following layer.

The network can be characterized by:

- the number of layers (its **depth**)
- the number of neurons in each layer
- the number of input variables (= number of neurons in the first layer, here: 5)
- the number of output variables (= number of neurons in the final layer, here: 1)

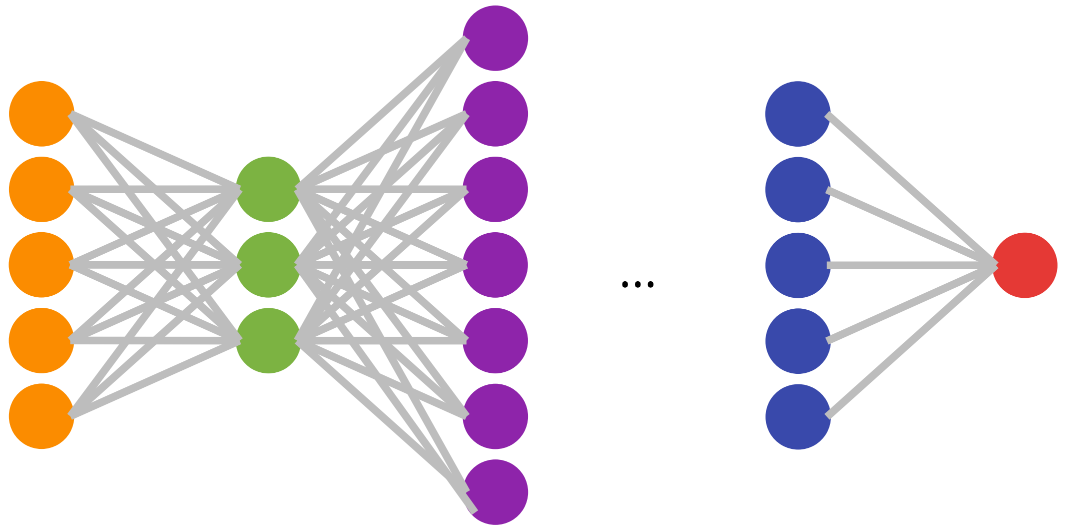
1

Fully-connected neural network: how does it work?



- Vectorial input data are provided to the network, one value per neuron in the input layer (in this case: x = [1, 0, 0, 1, 1])
- All inputs are seen by each neuron in the underlying layer.
- Each neuron will process the incoming information, firing (1) under some conditions, idling (0) otherwise.
- Repeat…
- The output is generated in the final layer (in this case: scalar output).

Artificial neural networks: Connectionism

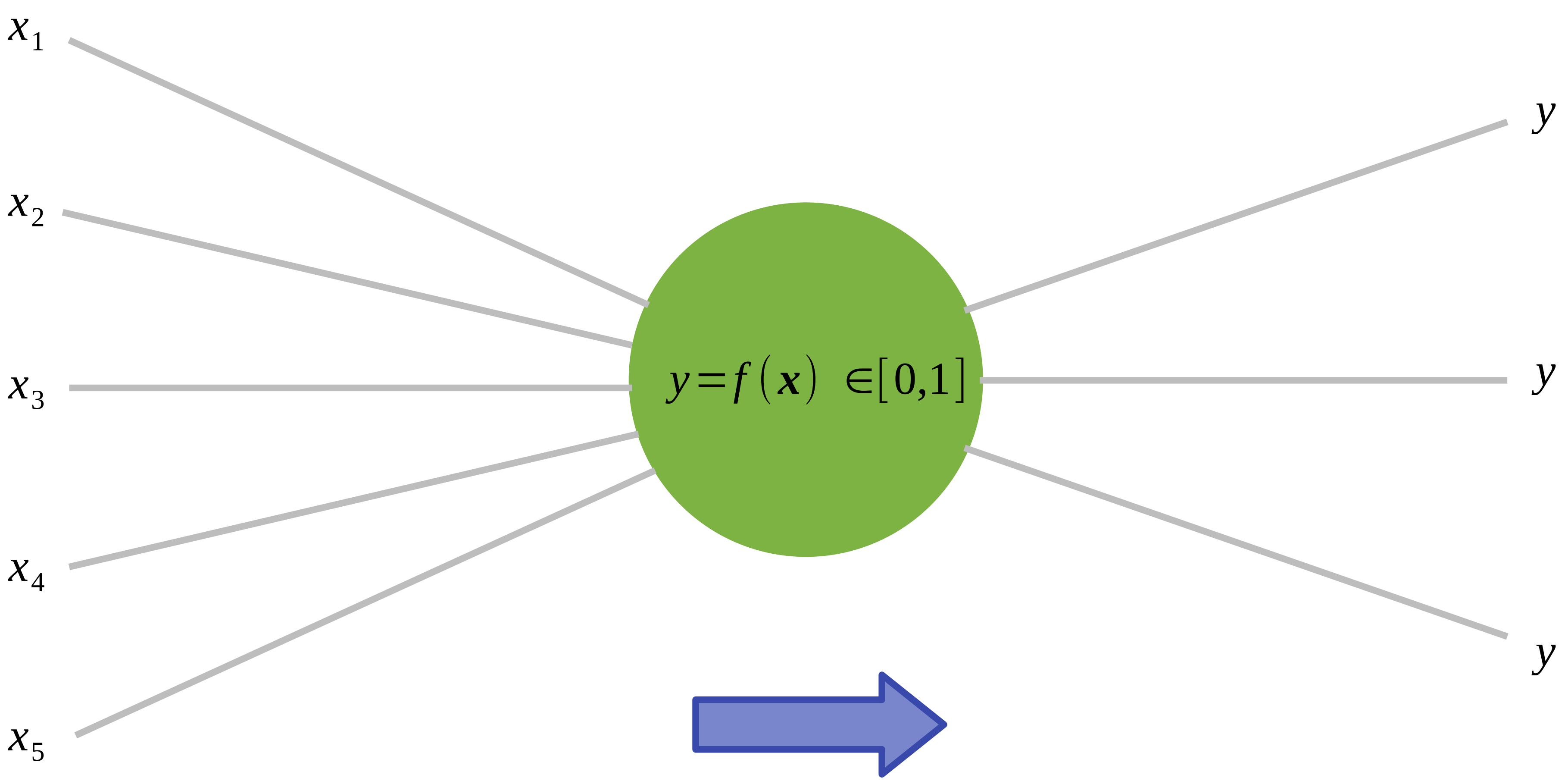


- Can we implement artificial neural networks to learn specific tasks?
  → **Connectionism**

- In general terms, a neural network acts as a **function approximator**: any mathematical function can be approximated!

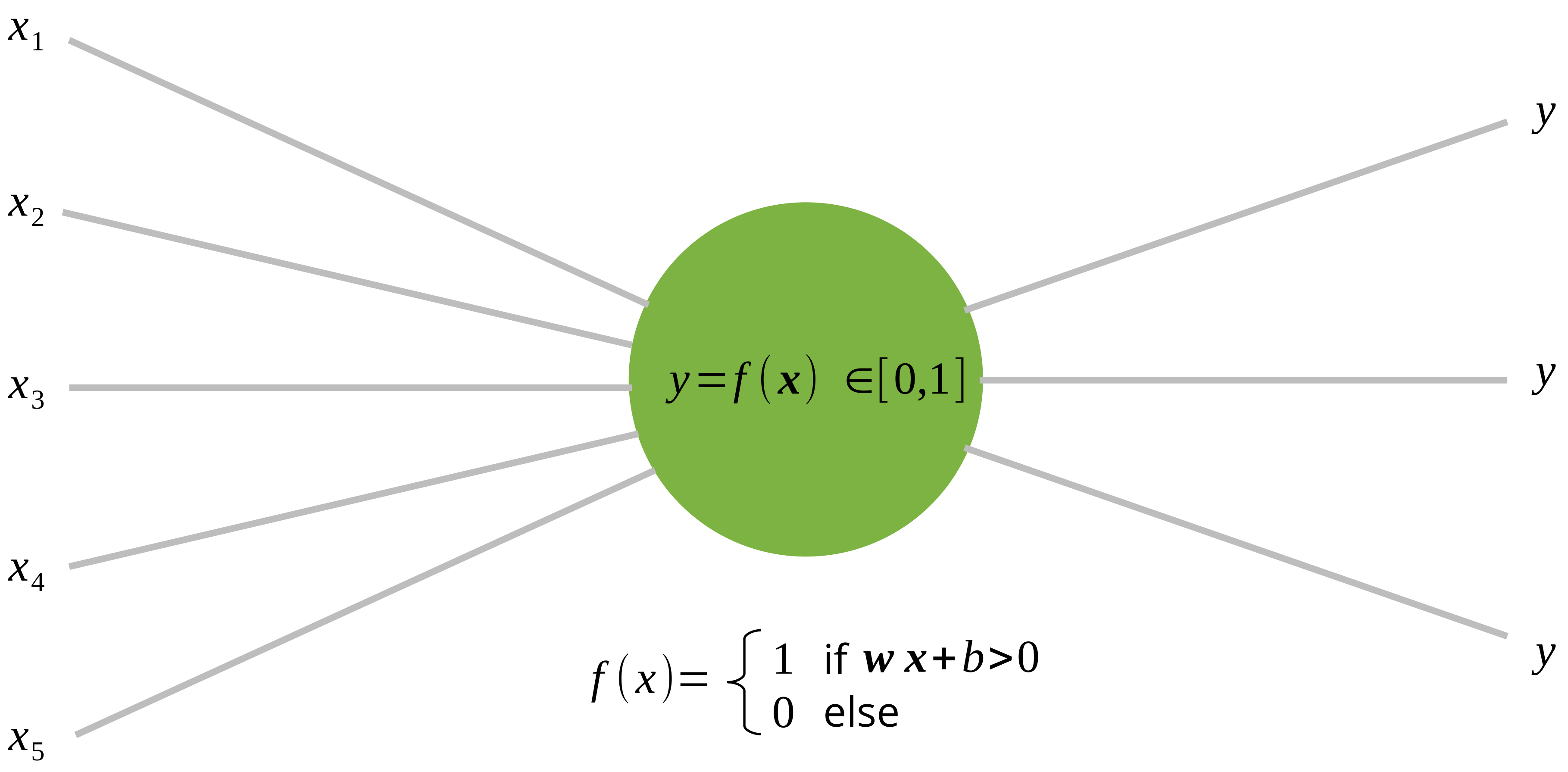
- Before we can implement artificial neural networks, we have to solve two problems:
  - How to implement neurons?
  - How to train the network?

A general neuron



A neuron takes in a vector of values, processes them and returns a binary signal based on its learned behavior, which is then passed on to all neurons in the following layer.

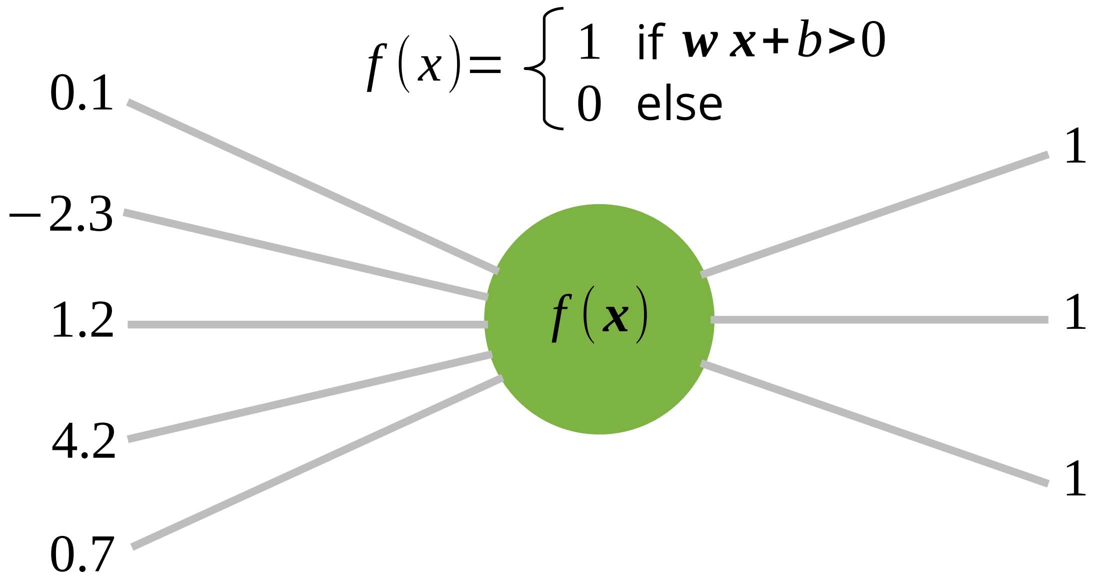
An artificial neuron: the Perceptron



$$f(x) = \begin{cases} 1 & \text{if } \boldsymbol{w}\,\boldsymbol{x} + b > 0 \\ 0 & \text{else} \end{cases}$$

Compute the dot product between input variables $\boldsymbol{x}$ and **weights** $\boldsymbol{w}$, add a **bias** value (b); if the resulting value is greater zero, the perceptron neuron fires ($y=1$), otherwise not ($y=0$). The step function is called the **activation function**: it introduces **non-linearity** into the output of the Perceptron.

The Perceptron: an example



$f(x) = \begin{cases} 1 & \text{if } \boldsymbol{w}\boldsymbol{x}+b>0 \\ 0 & \text{else} \end{cases}$
0.1
-2.3
1.2
4.2
0.7
$f(\boldsymbol{x})$
1
1
1

$$\boldsymbol{x}\boldsymbol{w}+b = x_1 w_1 + x_2 w_2 + \ldots + x_5 w_5 + b = 0.08$$

$$\boldsymbol{x}\boldsymbol{w}+b>0 \Rightarrow f(\boldsymbol{x})=1$$

The neuron fires!

Input: $\boldsymbol{x}=[0.1, -2.3, 1.2, 4.2, 0.7]$

Weights: $\boldsymbol{w}=[1.3, 0.2, -4.5, 1.6, -0.3]$

Bias: $b=-0.7$

Weights and Bias are **learned** in a supervised setup based on labeled training data (more in a bit).

A single Perceptron can be considered as a linear classifier...

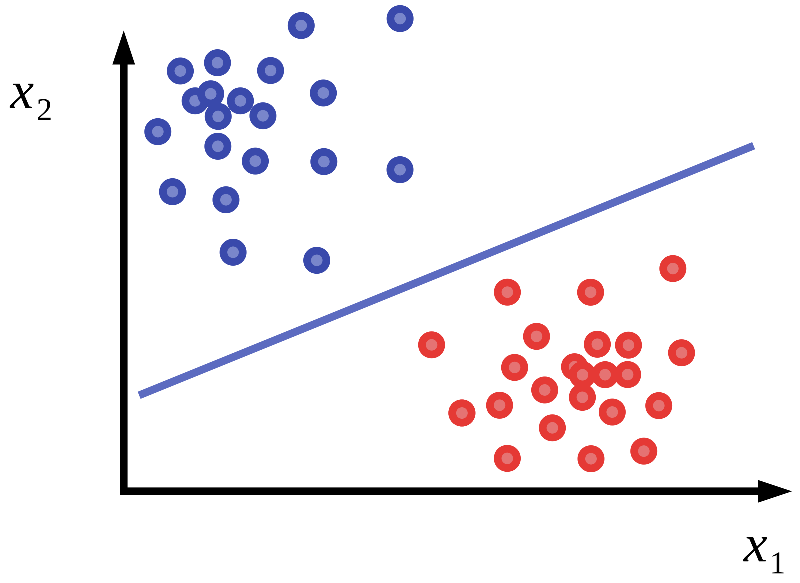
Perceptron: interpretation as linear classifier

Consider the case of two input variables for the sake of simplicity.

In this case, $\boldsymbol{x}\boldsymbol{w}+b>0$ simply means that the Perceptron fires only for data points that are above a line in this two-dimensional space.

Therefore, the Perceptron acts as a linear classifier (as we have already seen in lecture 3).

But this means that we already know a way to train Perceptrons...

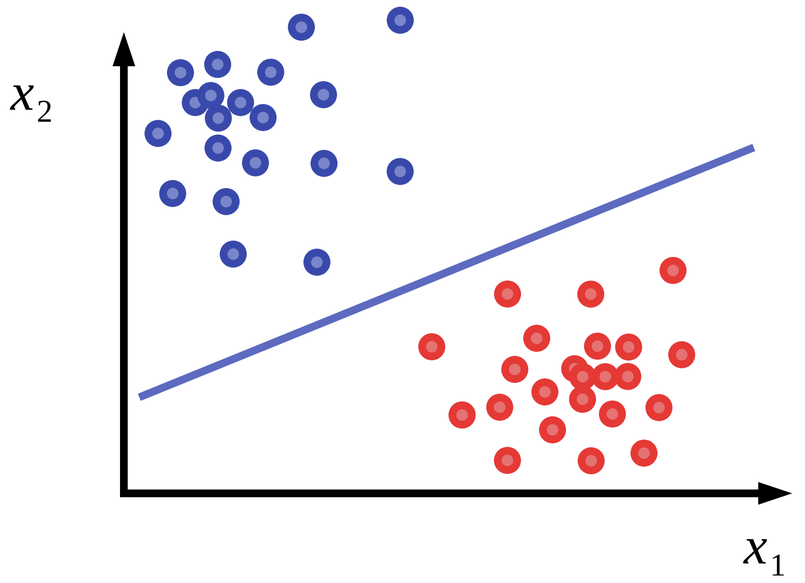


Perceptron: training

We already mentioned the **Perceptron learning rule**:

We consider each data point, consisting of $\boldsymbol{x}$ and ground-truth label y' and check whether the prediction $f(\boldsymbol{x})$ is correct, or not. If…

- $\bar{f}(\boldsymbol{x})=y$, then do nothing.
- $\bar{f}(\boldsymbol{x})=0$ but $y'=1$, then increase w_i if $x_i \geq 0$, or vice versa.
- $\bar{f}(\boldsymbol{x})=1$ but $y'=0$, then decrease w_i if $x_i \geq 0$, or vice versa.

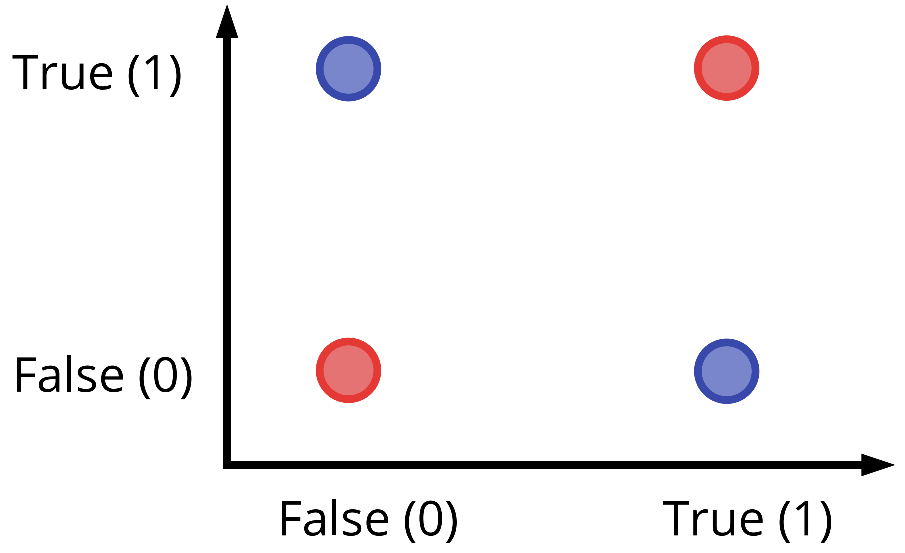


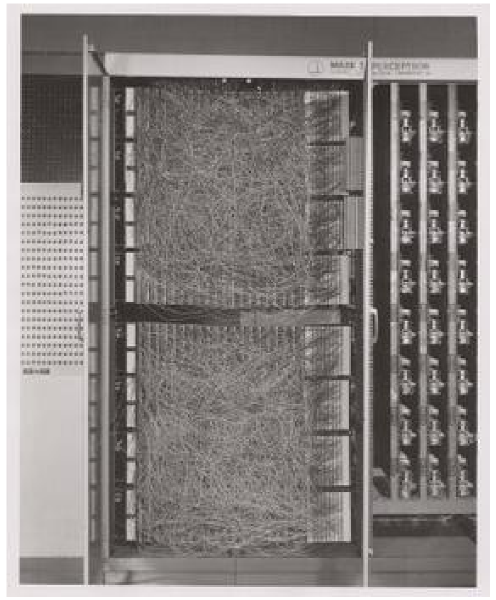
Weights are adjusted by a step size that is called the **learning rate**. By iteratively running this algorithm over your training data multiple times, the weights can be learned so that the model performs properly.

Perceptron: Limitations

Individual Perceptrons are able to learn specific tasks, such as classification of linearly separable problems.

However, there are strong limitations. The most famous one is its inability to reproduce the logical **exclusive-or** (**XOR**) **function**:

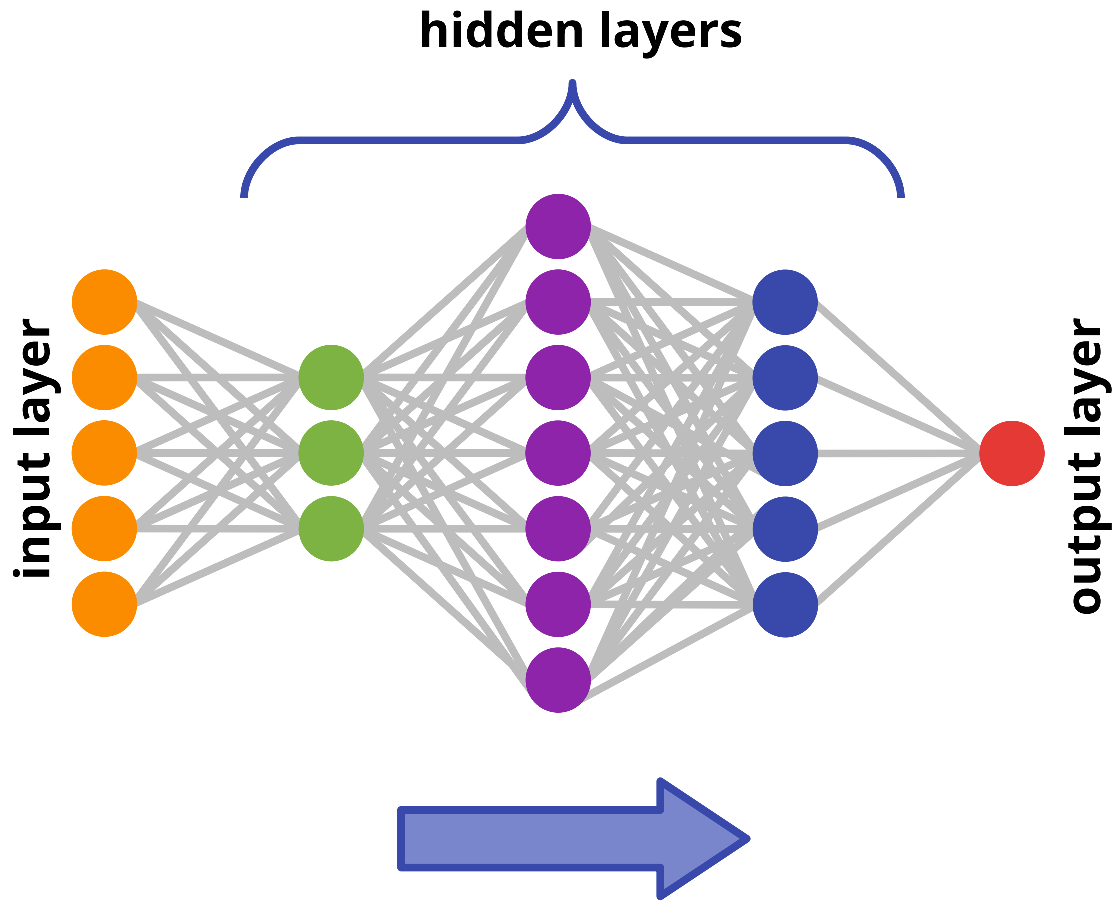



Cornell University Library

The major reason for this inability is that Perceptrons are simply **linear functions**.

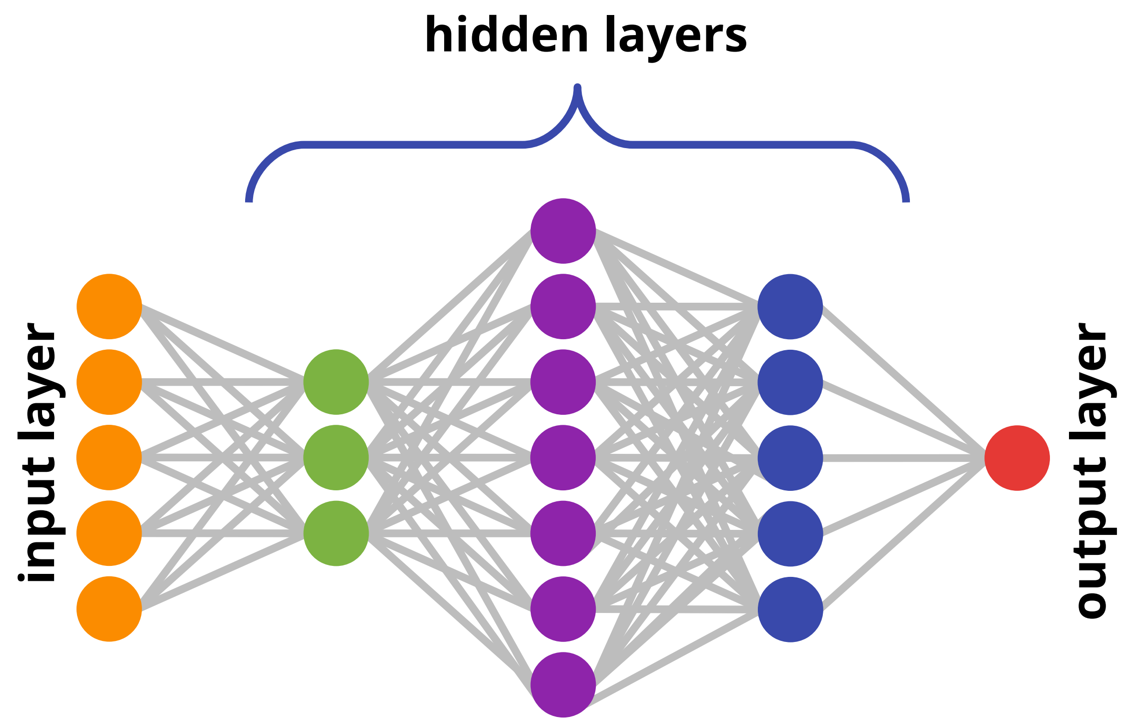
Multi-layer Perceptrons concatenate layers of Perceptrons, which makes them much more powerful.

Multi-layer Perceptron (MLP)



- MLPs are simple **feed-forward** neural networks: information traverses the graph in only one direction
- MLPs are **fully-connected**: every neuron is connected to all neurons in the previous layer and all neurons in the following layer
- MLPs can learn more complex relations from data than single Perceptrons: each layer adds additional **non-linearities** that increase the model's capacity
- Modern MLPs utilize additional layers and other non-linear activation functions that support the learning process

MLP: number of parameters



Consider a MLP with 3 hidden layers and following numbers of neurons per layer:

- Input layer: 5 features
- Hidden layer 1: 3 neurons (5 features in, 3 out)
- Hidden layer 2: 7 neurons (3 features in, 7 out)
- Hidden layer 3: 5 neurons (7 features in , 5 out)
- Output layer: 1 neuron

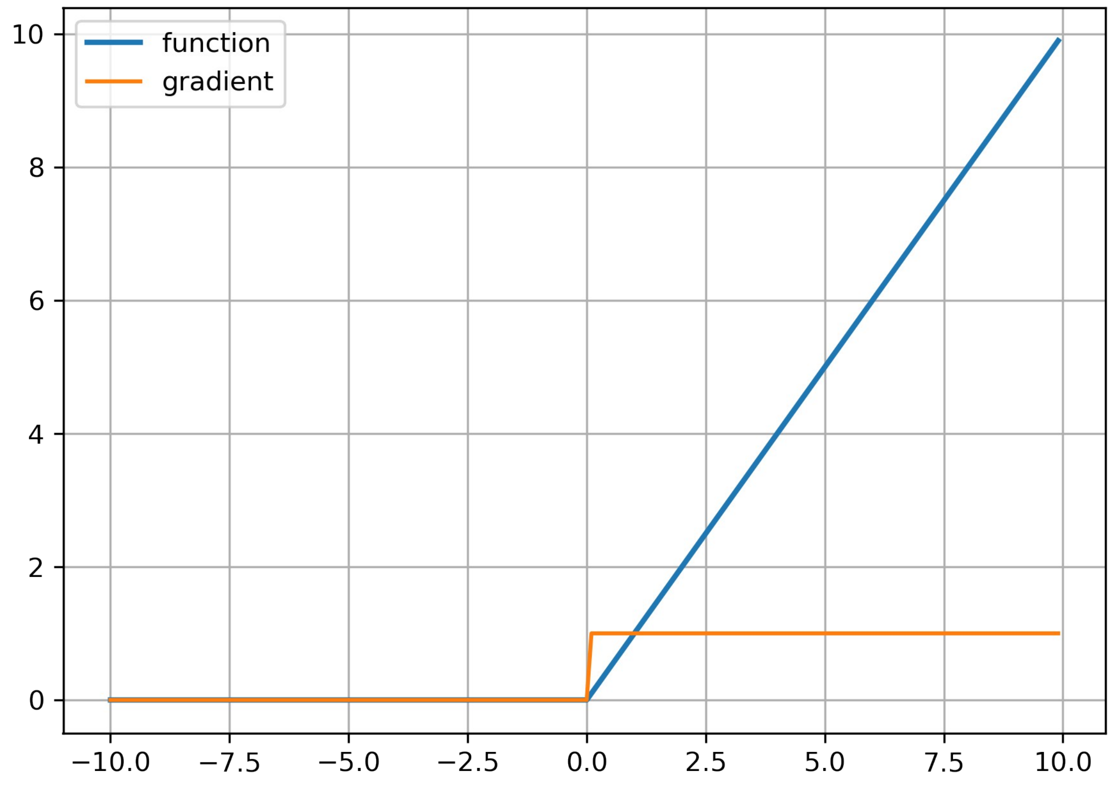
Each neuron is defined by $\boldsymbol{x}\boldsymbol{w}+b>0$, i.e., $\boldsymbol{w}$ has the same number of parameters as the number of input features. Therefore, the total number of weight parameters in this network is:

(5 + 1) x 3 + (3 + 1) x 7 + (7 + 1) x 5 + (5 + 1) = 92 parameters

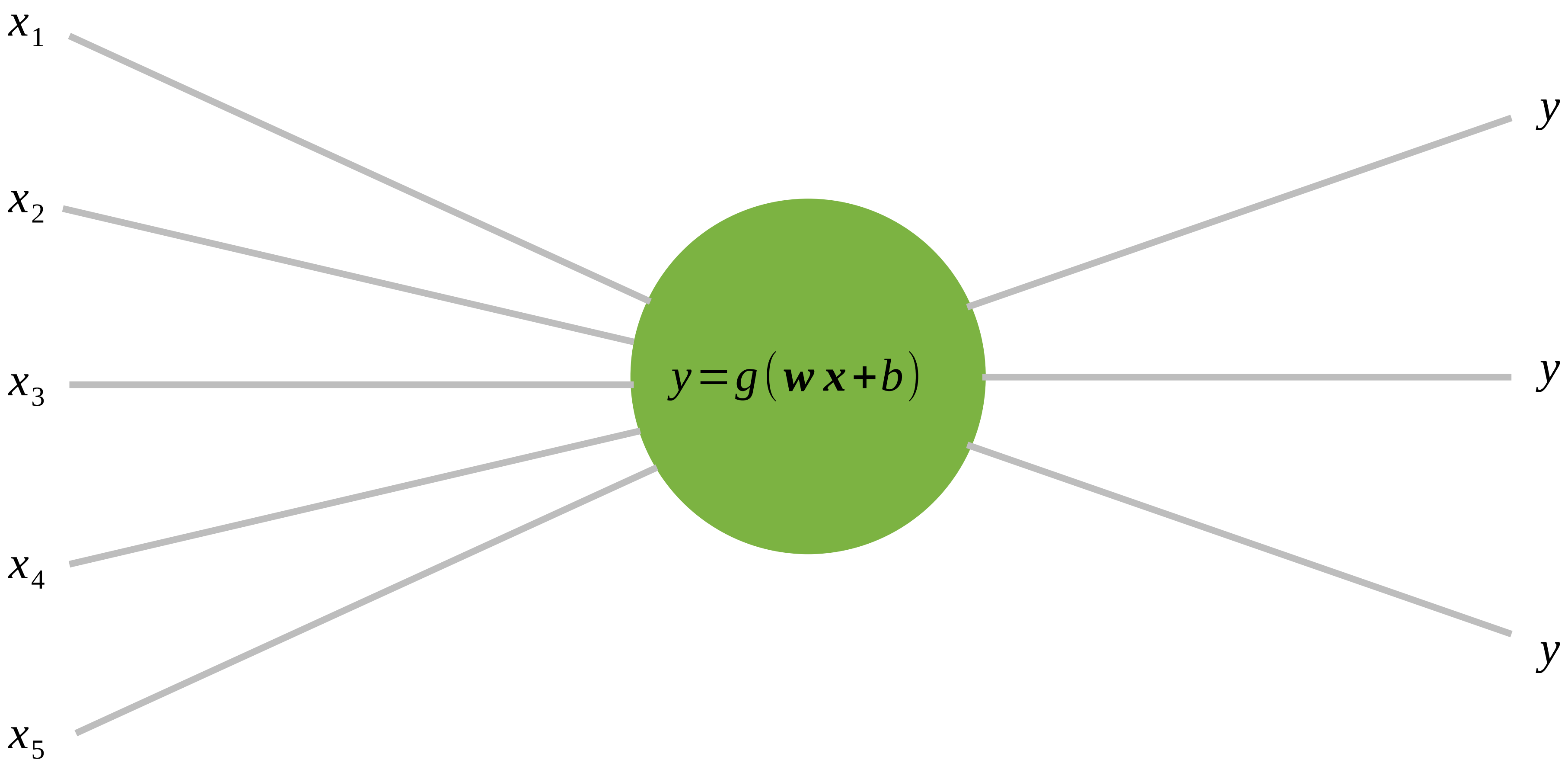
Neurons and neural networks: summary

- Artificial neural networks mimic cascades of large numbers of neurons in brain tissue.
- Artificial neurons compute the dot-product between input vectors and learned weights and produce an output signal that propagates through all deeper layers.
- The Perceptron is a simple artificial neuron that produces a binary output.
- A single Perceptron can be interpreted as a linear classifier.
- A Multi-layer Perceptron is an early fully-connected neural network.

Activation Functions



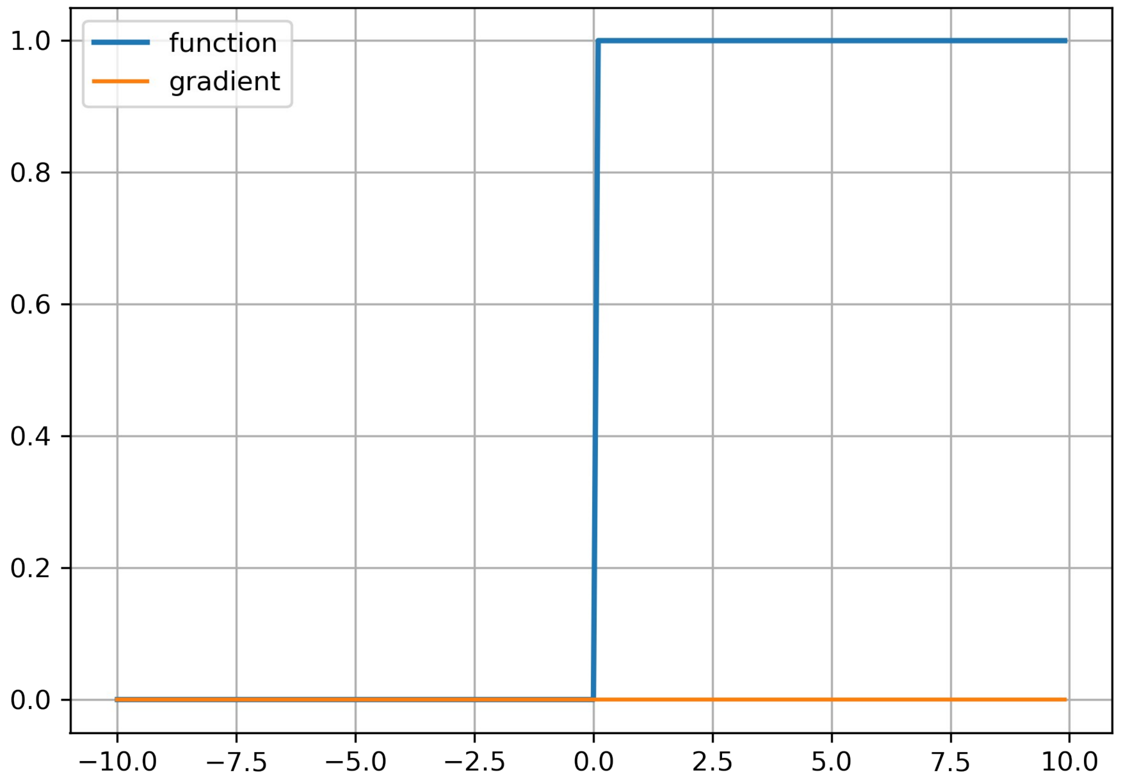
Activation function



The activation function defines when a neuron “fires”. **Non-linearity** increases the model’s **capacity**.

So far, we used a simple step function to define whether the neuron fires: $g(x)=\begin{cases} 1 & \text{If <condition>} \\ 0 & \text{else} \end{cases}$

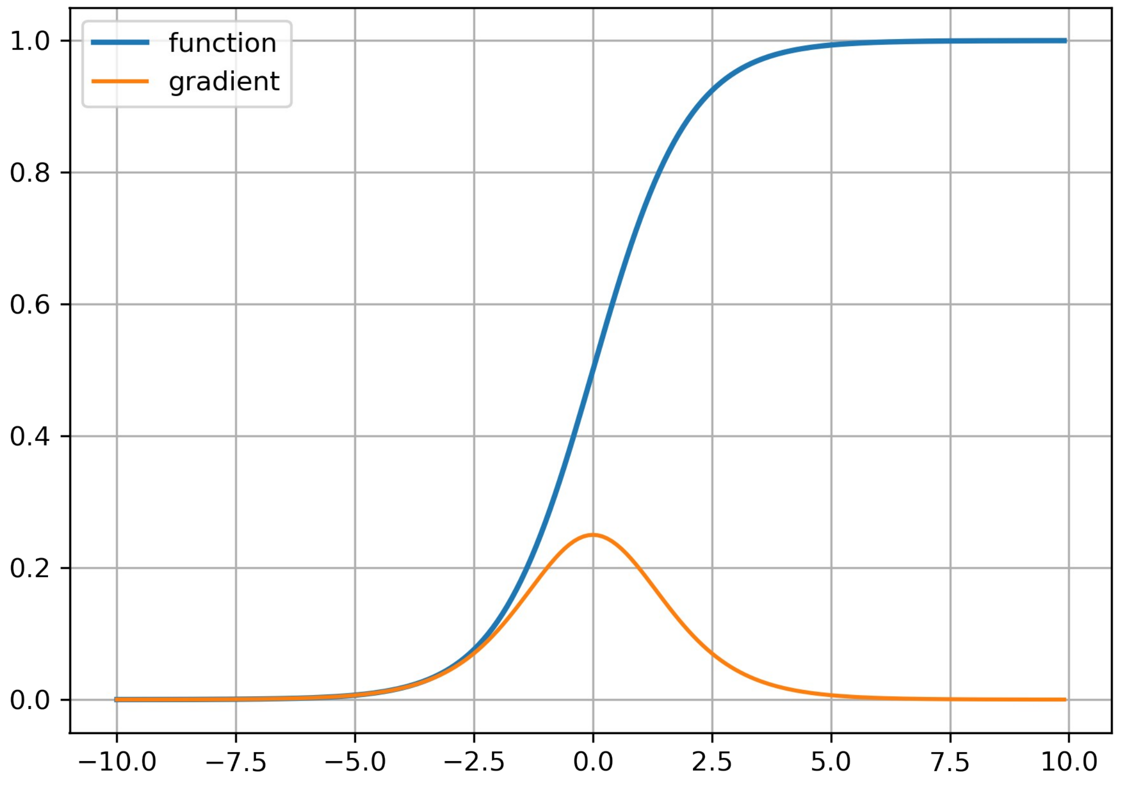
Activation functions: Step function



$$g(x) = \begin{cases} 1 & \text{If } x>0 \\ 0 & \text{else} \end{cases}$$

- (+): simple to implement
- (+): computationally inexpensive
- (-): only binary (discrete) output
- (-): no gradient

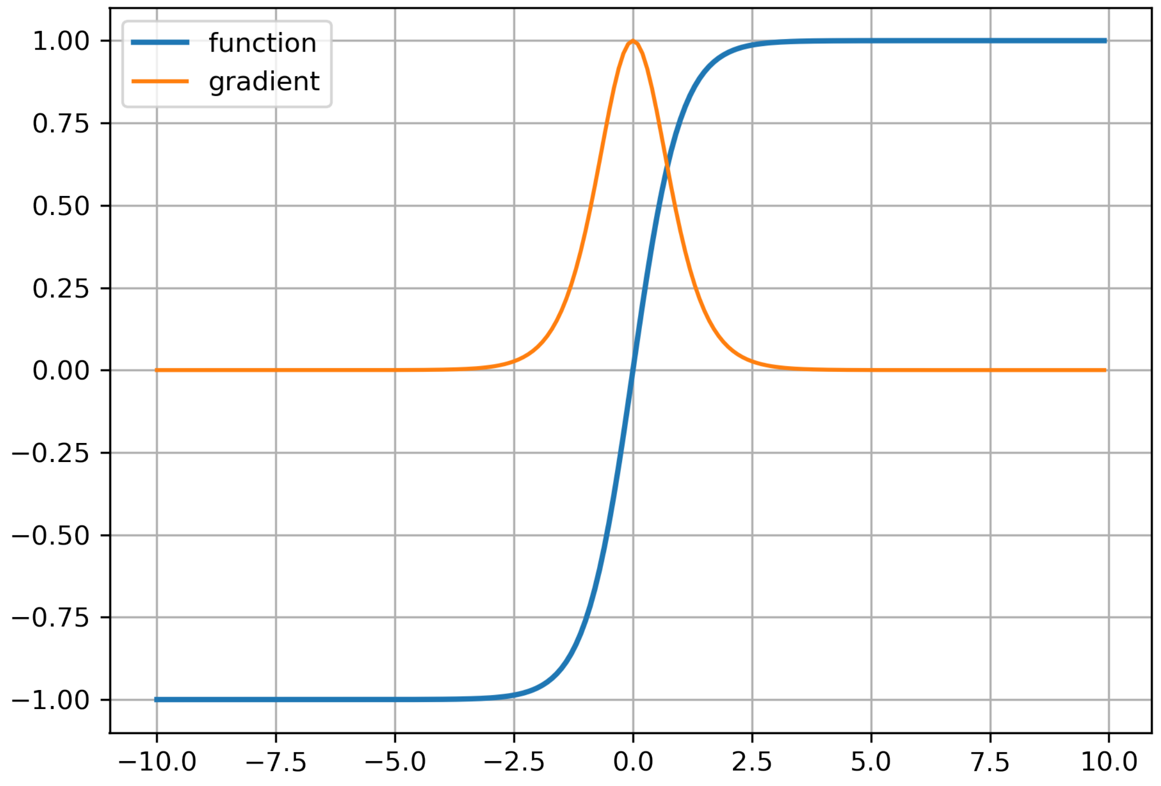
Activation functions: Sigmoid function



$$\sigma(x) = \frac{\exp(x)}{1+\exp(x)}$$

- (+): continuous non-linear function
- (+): gradient defined
- (-): asymmetric output value range [0, 1]
- (-): computationally expensive

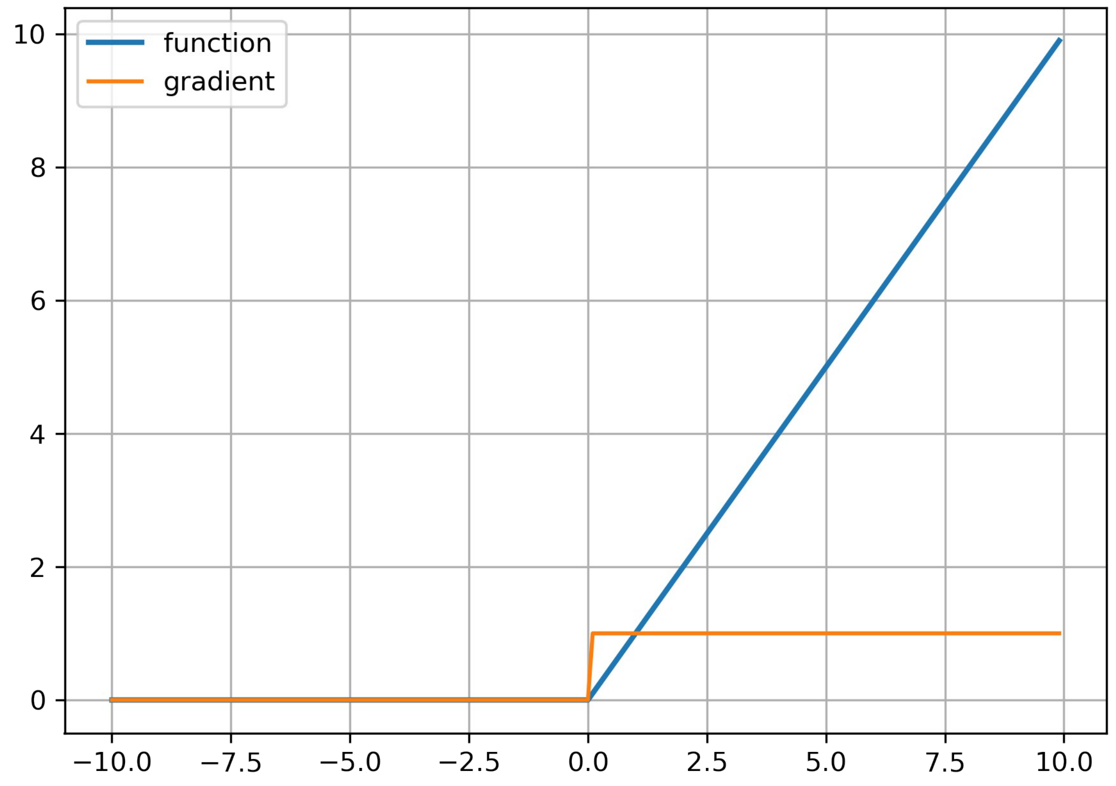
Activation functions: Tanh function



$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$$

- (+): continuous non-linear function
- (+): gradient defined
- (+): symmetric output value range [-1, 1]
- (-): computationally expensive

Activation functions: Rectified Linear Unit (ReLU) function



$$ReLU(x) = \begin{cases} x & \text{If x>0} \\ 0 & \text{else} \end{cases}$$

- (+): continuous non-linear function
- (+): gradient defined, and simple to compute
- (+): computationally inexpensive

Activation Functions: which one to use?

Many different choices for activation functions exist. But which one is the best one?

We will see later the importance of the activation function to be **differentiable**: we need the gradient to be computable. Therefore, a step function is not a good choice as it has no gradient.

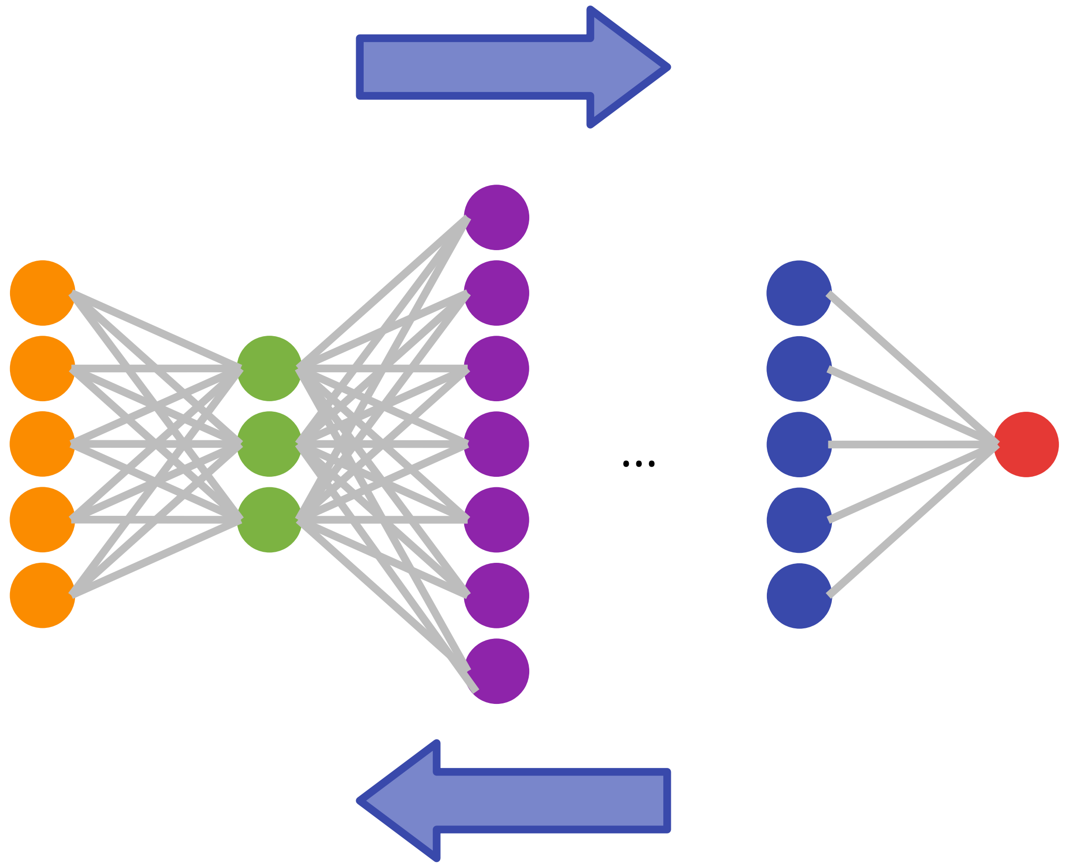
Non-linear activation functions enable deep neural networks to learn complex tasks and to approximate any mathematical function.

Research has shown that Sigmoid, Tanh and ReLU activations roughly lead to similar results. Therefore, **ReLU**, which is computationally the most efficient activation function, is now most commonly used.

Activation functions: summary

- The activation function of a neuron decides when it “fires”.
- A good activation function should be:
    - Continuously differentiable
    - Non-linear
    - Computationally inexpensive
- It turns out that ReLU is just as good as any other activation function. Therefore, ReLU is most commonly used nowadays.
- The non-linearity of activation functions enables deep neural networks to learn complex tasks.

Loss Functions, Backpropagation and Gradient Descent



Loss functions and Backpropagation

So far we learned how a neural network is set up and what it does.

To learn a task, neural networks require a training strategy.

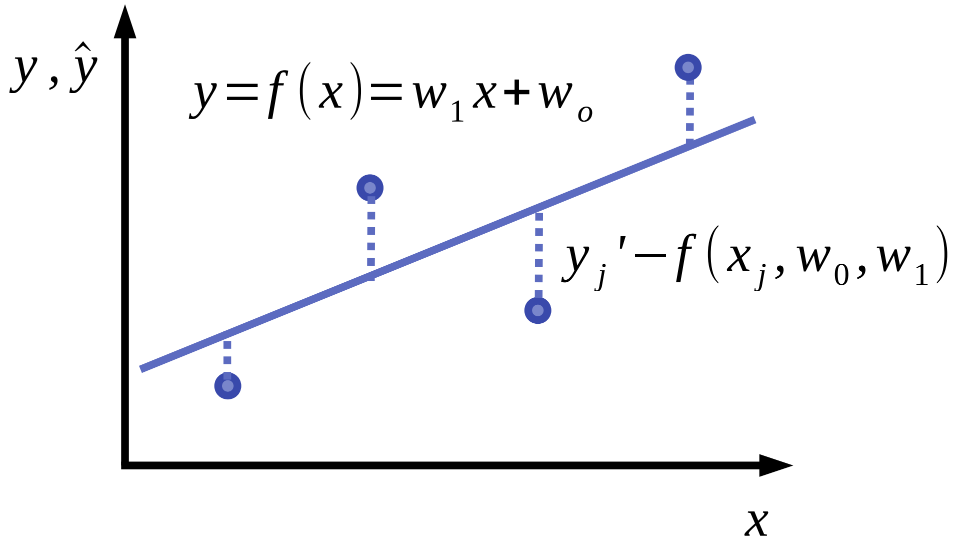
We will see in the following that we can use a loss function in combination with a method called “gradient descent” to enable learning in neural networks.

Loss functions

We saw loss functions when we talked about **linear regression**. A quick reminder from lecture 3:

We define a **Loss** (or Objective) function that is the sum of the squared errors over all data points:

$$L = \sum_j^N \left(y_j' - f(x_j, w_0, w_1)\right)^2 = \sum_j^N \left(y_j' - (w_1 x_j + w_0)\right)^2$$



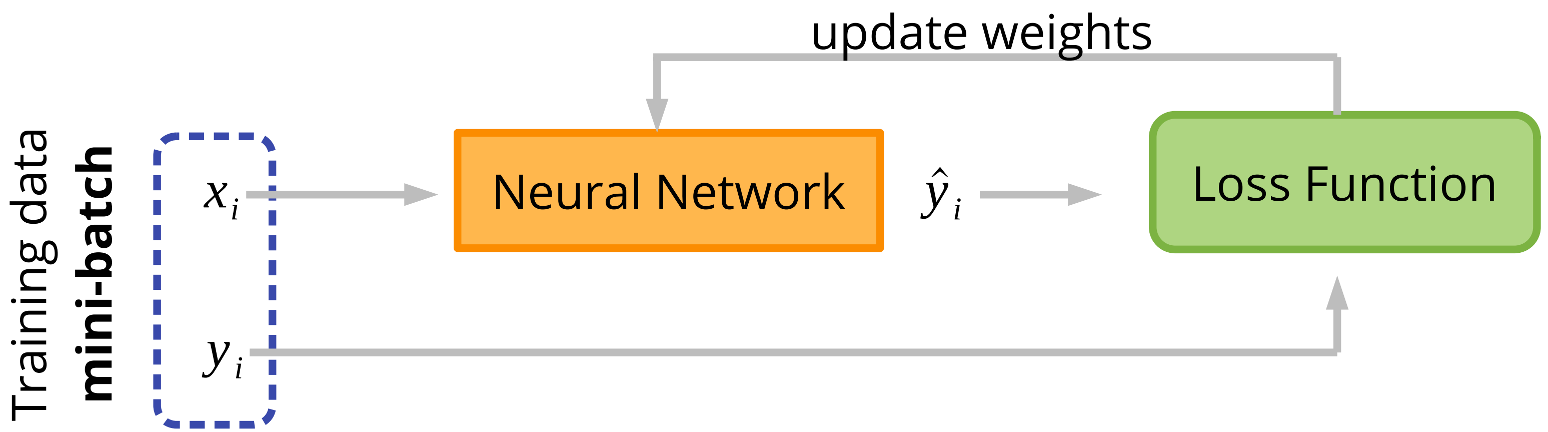
Our model performs best if we manage to minimize the loss by tuning the weight parameters.

"Least-squares" fitting in linear regression is a **convex optimization problem**: there is only one solution to the problem and it is per definition the best solution.

Neural network optimization

Optimization is much more complex for neural networks, since the optimization problem is **non-convex**: finding any solution (which might not be the best solution) is a stochastic process:

- We define a **loss function** that serves as a(n inverse) performance metric proxy. The loss function takes in the model prediction $\hat{y}_i$ and ground-truth target y_i for a given input data sample, x_i.
- The **goal** is to minimize the loss function based on the training data set.
- We **iteratively adjust the network weights** in such a way as to minimize the loss function.



Loss functions

Different loss functions are available for different tasks to learn.

Regression

L1-Loss:

$$L(y, \hat{y}) = \sum_i^N |y_i - \hat{y}_i|$$

L2-Loss:

$$L(y, \hat{y}) = \sum_i^N (y_i - \hat{y}_i)^2$$

Classification

Negative Log-Likelihood (NLL)-Loss:

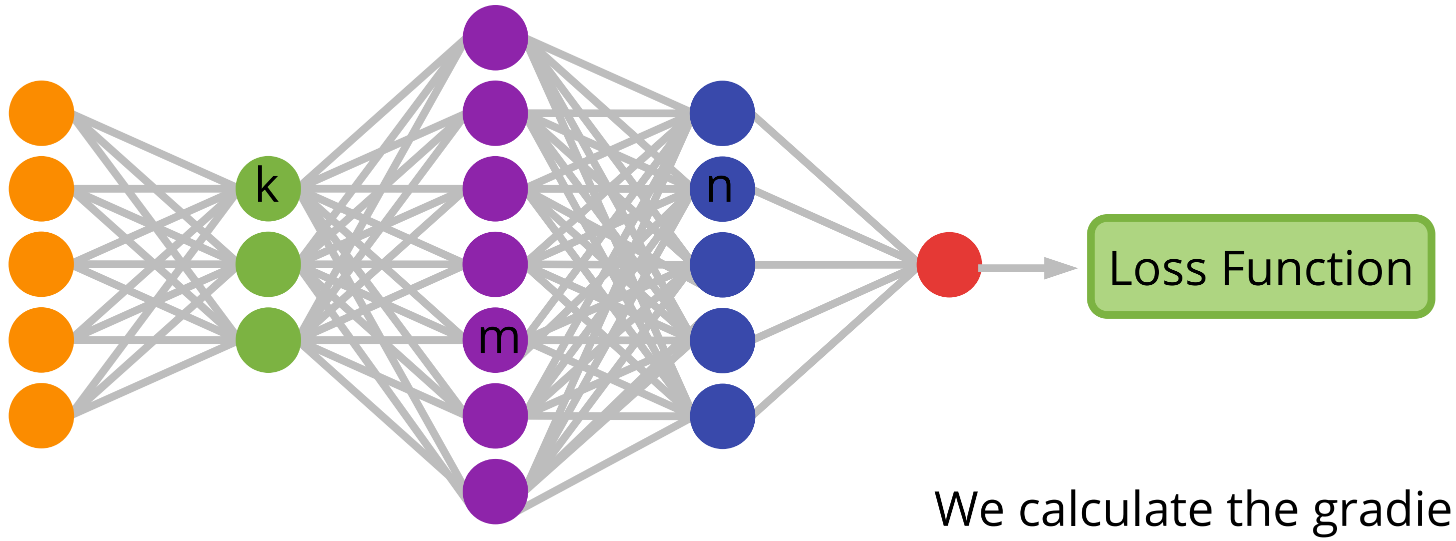
$$L(y, \hat{y}) = -\sum_i^N \log(\hat{y}_i)$$

Summation only over correct classes (see lab course for details)

Neural network optimization: Backpropagation

How do we modify the network weights to reduce the loss?

- **Random changes**: possible, but not very goal-oriented
- **Backpropagation**: we check for every single weight how changing it would affect the loss $\frac{\partial L}{\partial w_k}$



Chain rule of differentiation:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

$$\frac{\partial L}{\partial w_k} = \sum_n \sum_m \frac{\partial L}{\partial w_n}\frac{\partial w_n}{\partial w_m}\frac{\partial w_m}{\partial w_k}$$

We calculate the gradient of the loss function with respect to every weight parameter; different paths are summed up.

Neural network optimization: Stochastic Gradient Descent

Backpropagation allows us to compute a gradient for every single weight parameter and for every mini-batch.

Based on the computed gradients, we can modify each individual weight parameter, w_i:

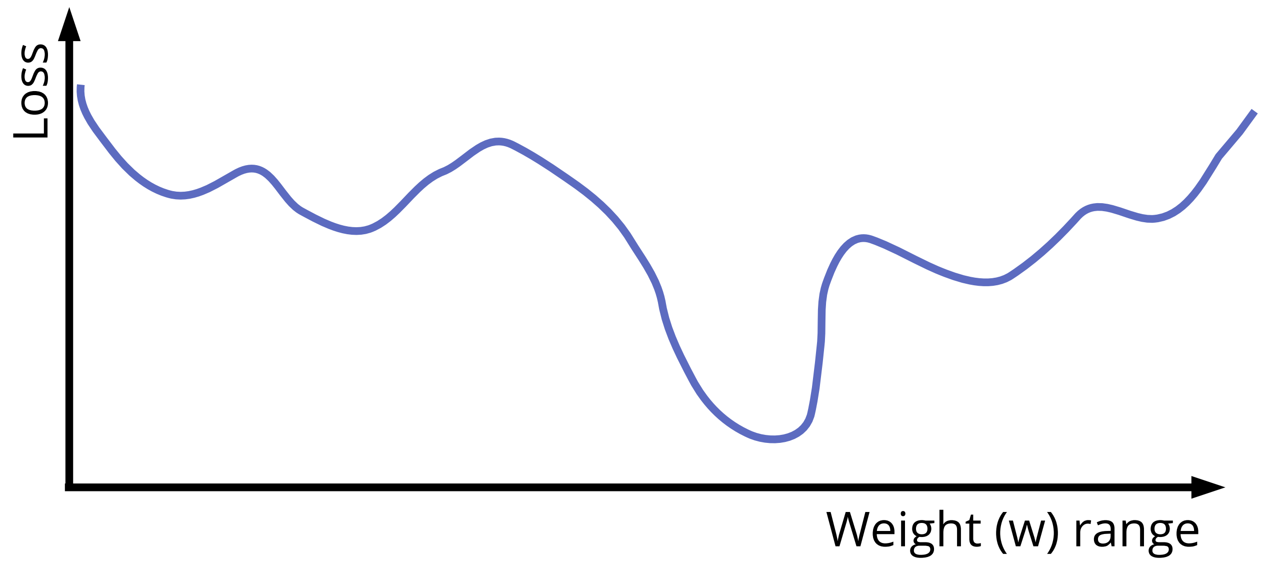
$$w_i = w_i - \alpha \nabla w_i$$

We define a step size for these modifications, which we call the **learning rate**, α.

This process is iterative and, since it depends on the random selection of mini-batches, also stochastic. Figuratively, we are following the gradients in the weight space to the lowest loss value. Therefore, this method is called **stochastic gradient descent**.

Visualization: gradient descent in one dimension

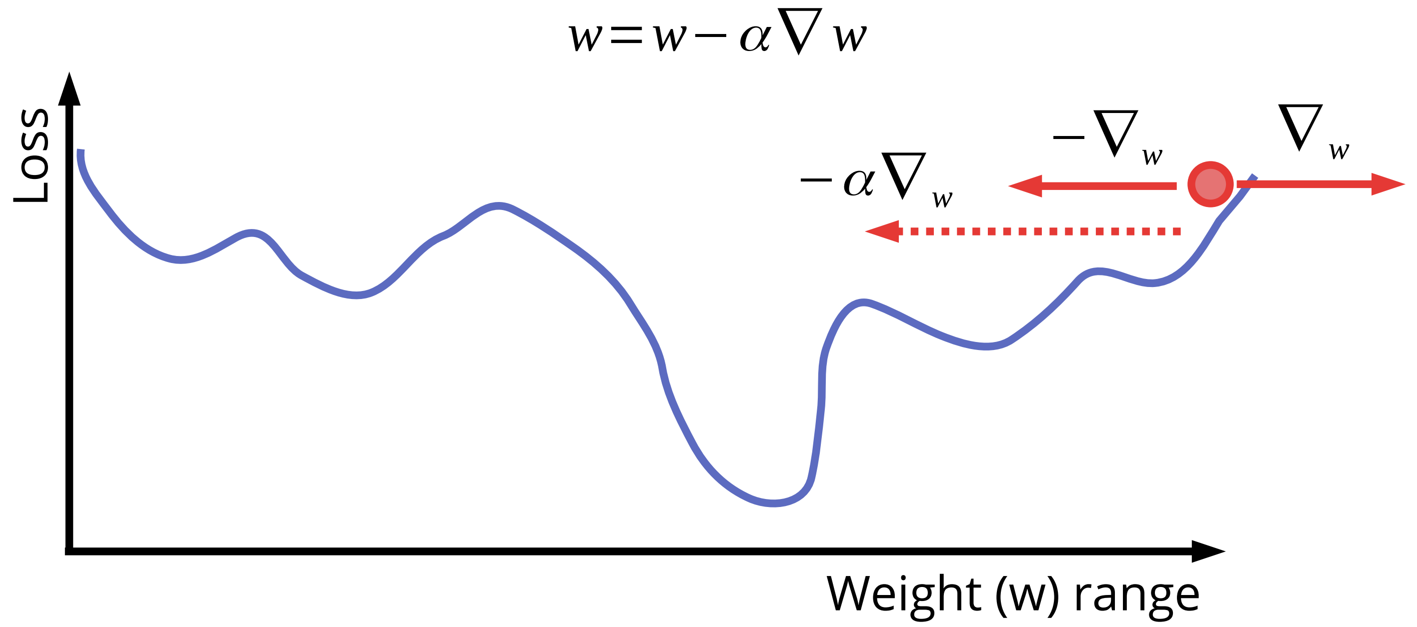
$$w = w - \alpha \nabla w$$



We visualize the idea behind gradient descent in one dimension. Therefore, we consider only a single weight over a range of possible values. For each of those values we can compute the loss function.

Gradient descent allows us to find the minimum of the loss in an iterative process.

Visualization: gradient descent in one dimension



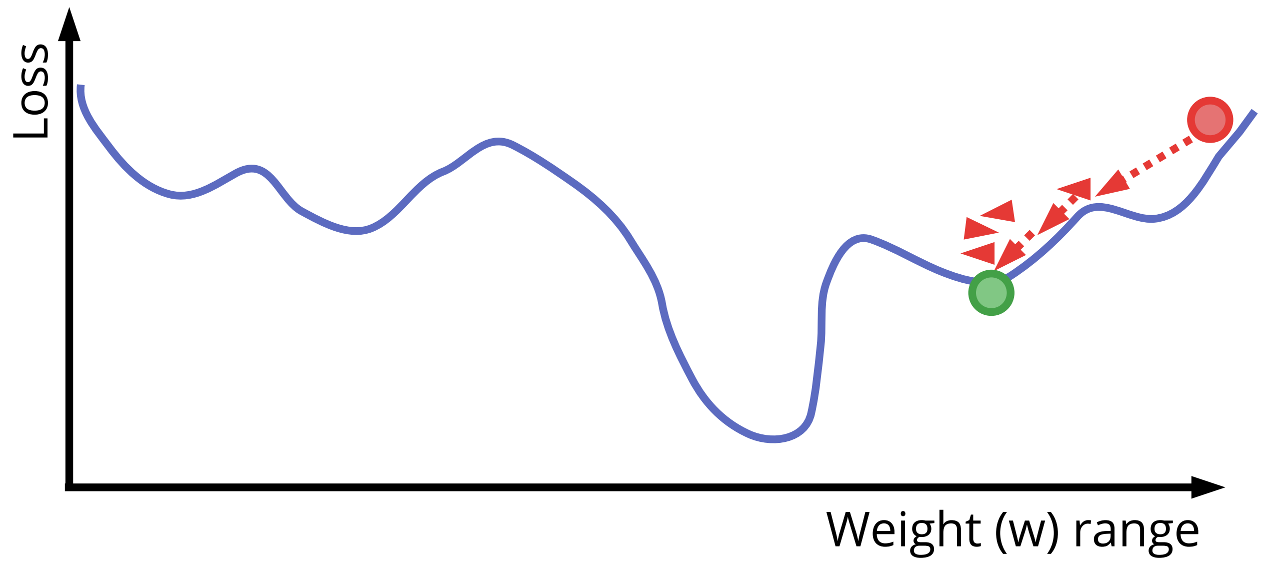
The gradient points in that direction in which the loss function slopes up.

We want to find the minimum of the loss function, so move in the opposite direction.

α modulates our step size.

Visualization: gradient descent in one dimension

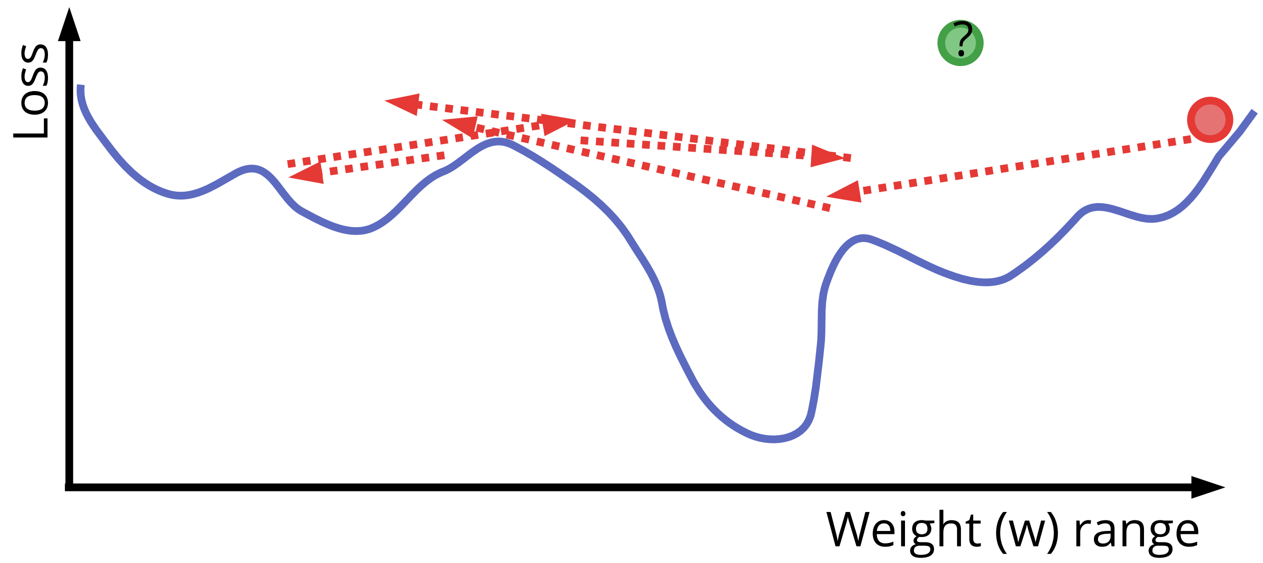
$$w = w - \alpha \nabla w$$



If we use a small learning rate α, it will take a long time to reach the global minimum. It is also possible to get stuck in a local minimum...

Visualization: gradient descent in one dimension

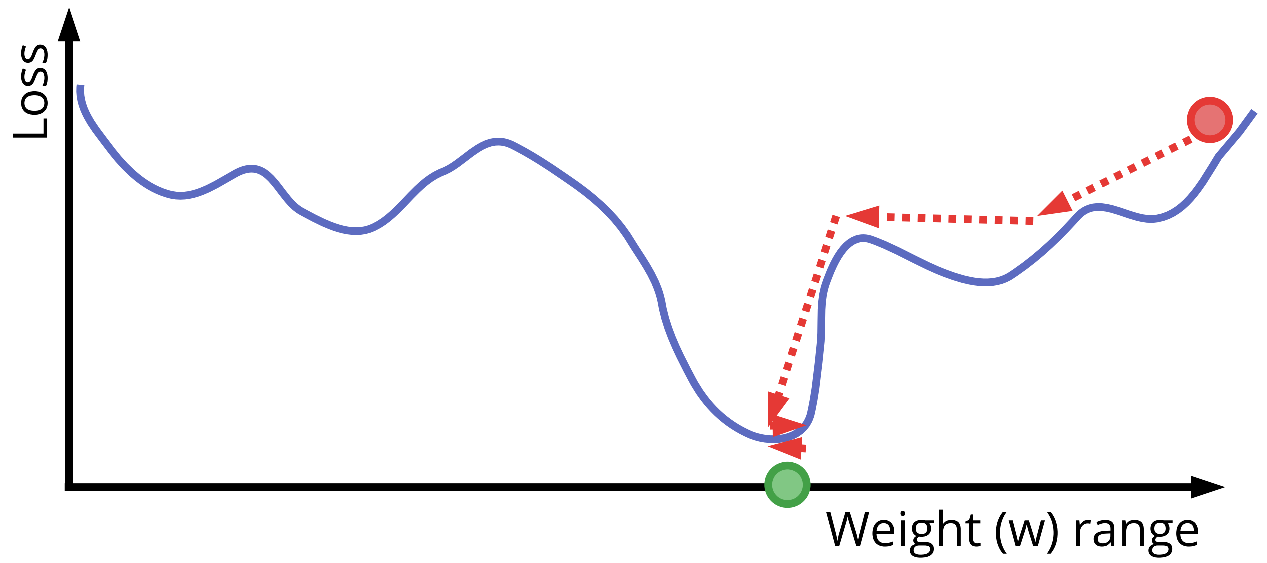
$$w = w - \alpha \nabla w$$



If we use a large learning rate α, it is possible that we miss the global minimum. Also, convergence is unlikely.

Visualization: gradient descent in one dimension

$$w = w - \alpha \nabla w$$

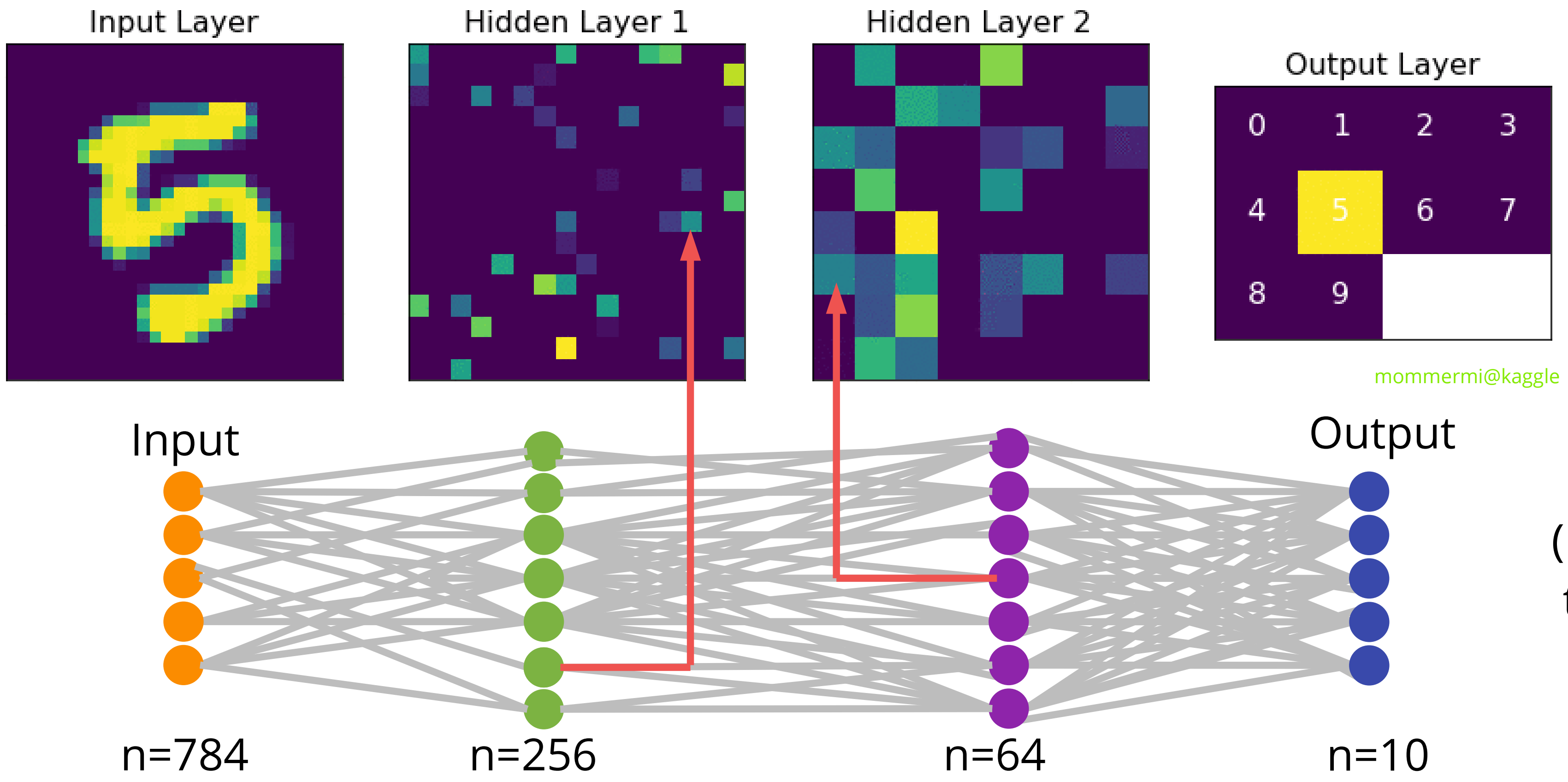


Finding a good learning rate maybe tricky. But it is mandatory for a successful training process.

The learning rate is a **hyperparameter** of every neural network model.

Example: what do neural networks learn?

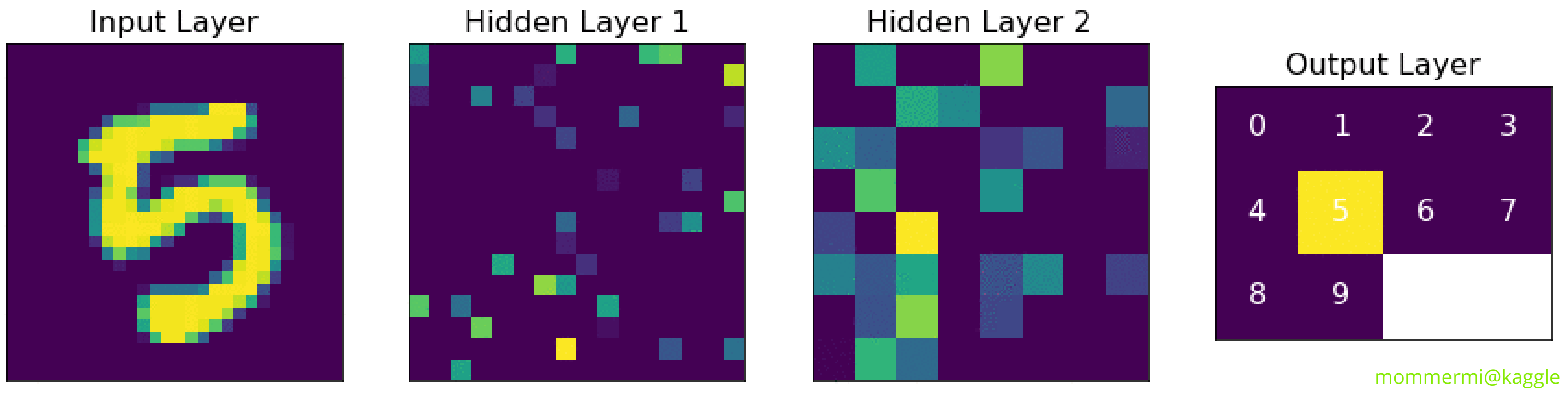
We consider a simple fully connected network and train it on the task of identifying hand-written digits from the MNIST dataset. We then visualize the activations in each neuron.



(Network not to scale)

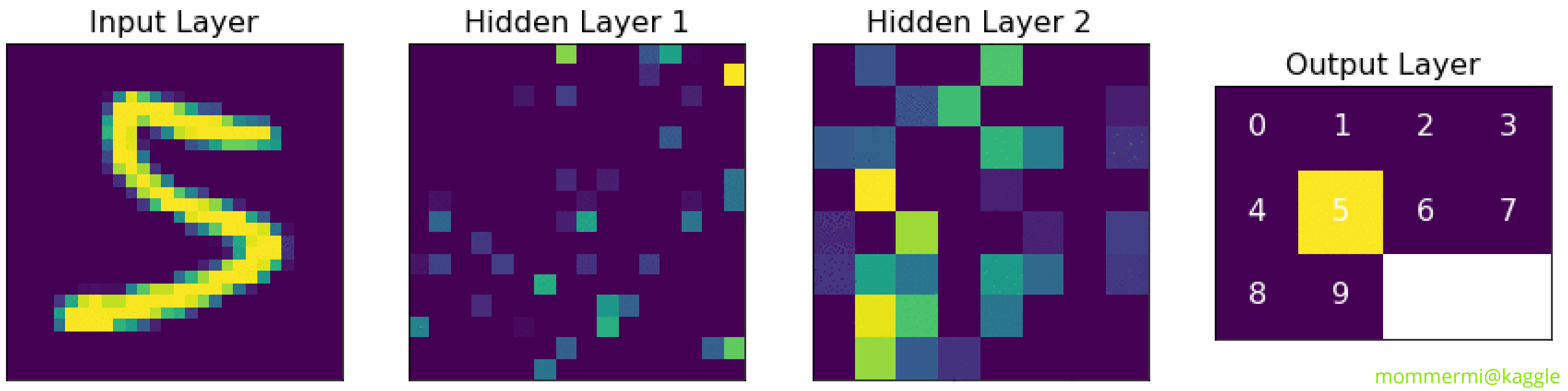
Example: what do neural networks learn?

Neural networks learn patterns from data to perform specific tasks.



Example: what do neural networks learn?

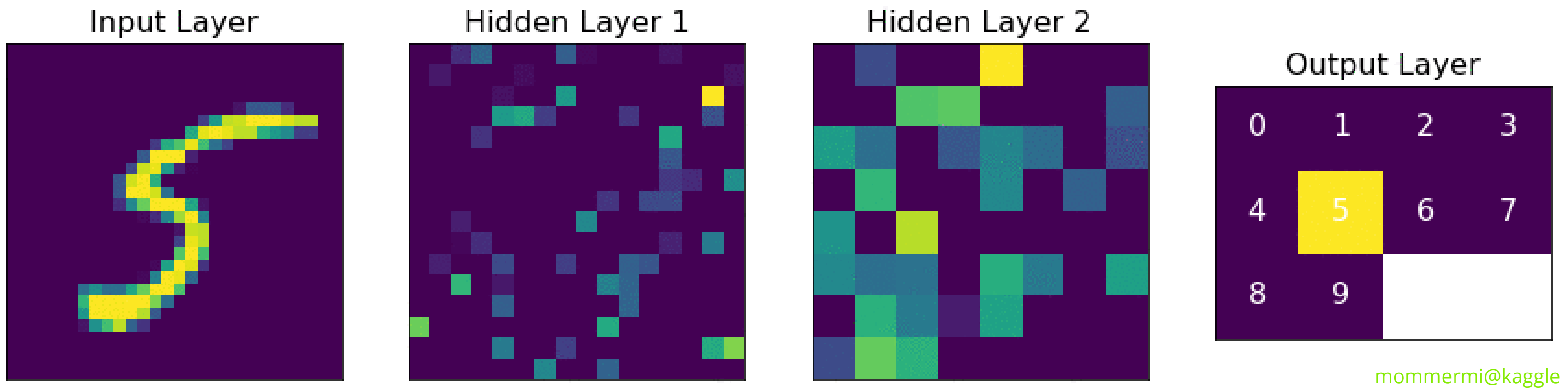
Neural networks learn patterns from data to perform specific tasks.



Seemingly different neurons fire for different input images.

Example: what do neural networks learn?

Neural networks learn patterns from data to perform specific tasks.

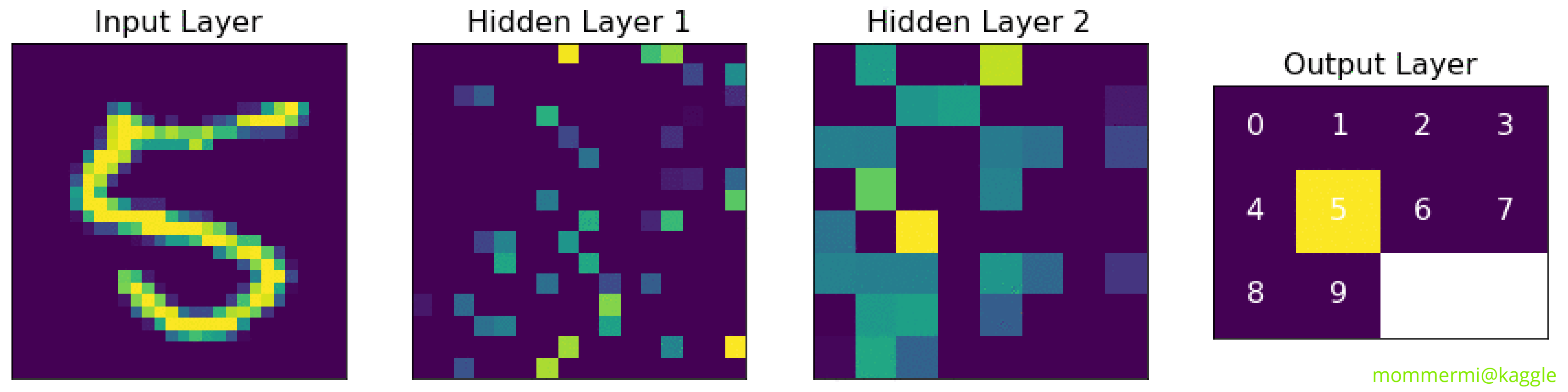


Seemingly different neurons fire for different input images.

But for images showing the same digit, there are some neurons that fire consistently – especially in the second hidden layer!

Example: what do neural networks learn?

Neural networks learn patterns from data to perform specific tasks.

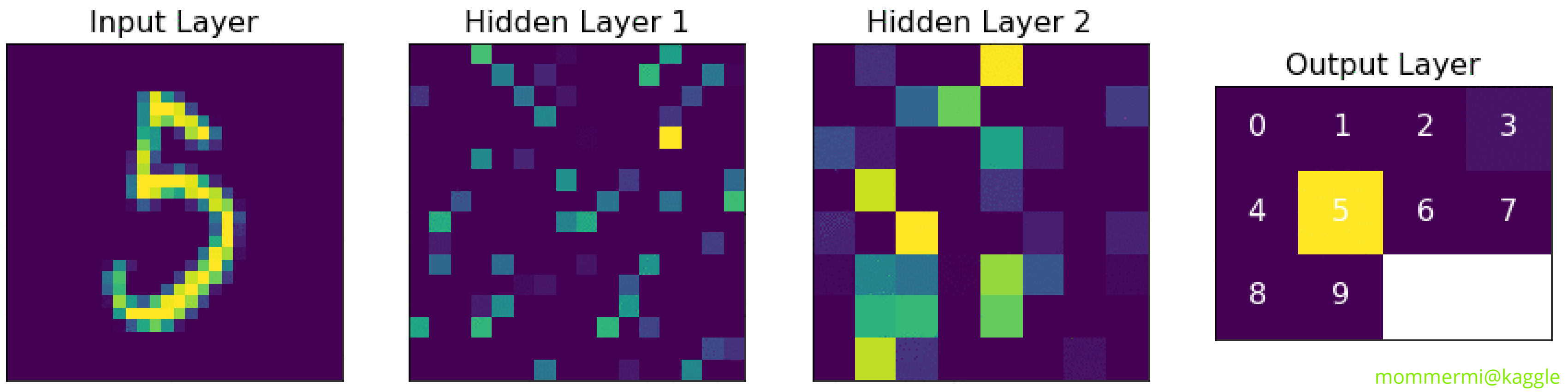


Seemingly different neurons fire for different input images.

But for images showing the same digit, there are some neurons that fire consistently – especially in the second hidden layer!

Example: what do neural networks learn?

Neural networks learn patterns from data to perform specific tasks.

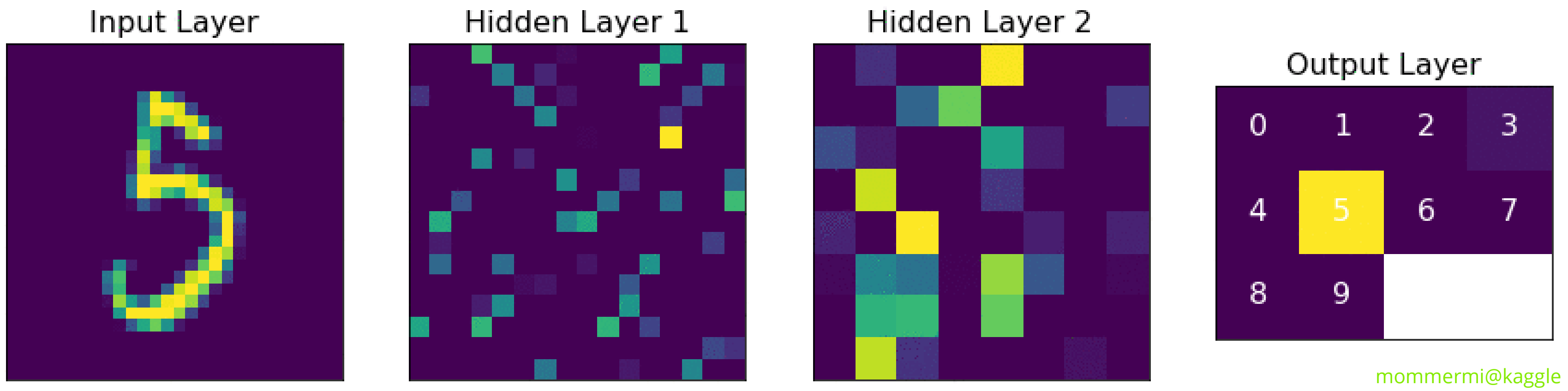


Seemingly different neurons fire for different input images.

But for images showing the same digit, there are some neurons that fire consistently – especially in the second hidden layer!

Example: what do neural networks learn?

Neural networks learn patterns from data to perform specific tasks.



Another interesting observation:

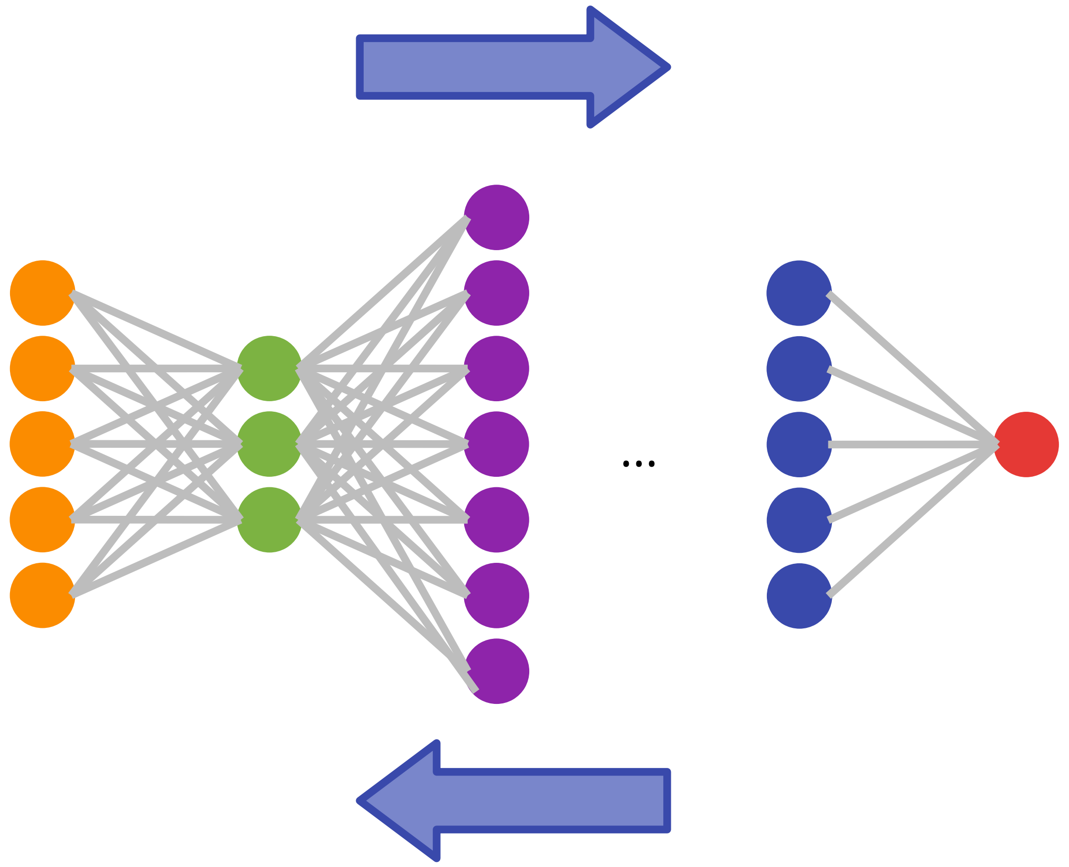
Early layers extract **low-level signals with spatial significance**, later layers interpret these signal and **provide semantic significance**!

End-to-end learning!

Loss functions, backpropagation and gradient descent: summary

- By defining a loss function that is related to the task the network has to learn, we can formulate the training process as an optimization problem.
- Key to a meaningful training process is the ability to compute the gradient of the loss function with respect to every single network weight parameter; this is achieved through a process called backpropagation.
- Stochastic gradient descent uses the gradients computed with backpropagation to update network weight parameters iteratively to reduce the model’s loss.
- Encoded in the trained weights are rules to perform the trained task. Early network layers deal with low-level information inherent to the data; later layers interpret high-level semantic information extracted in the early layers.

Neural Network Training and Evaluation



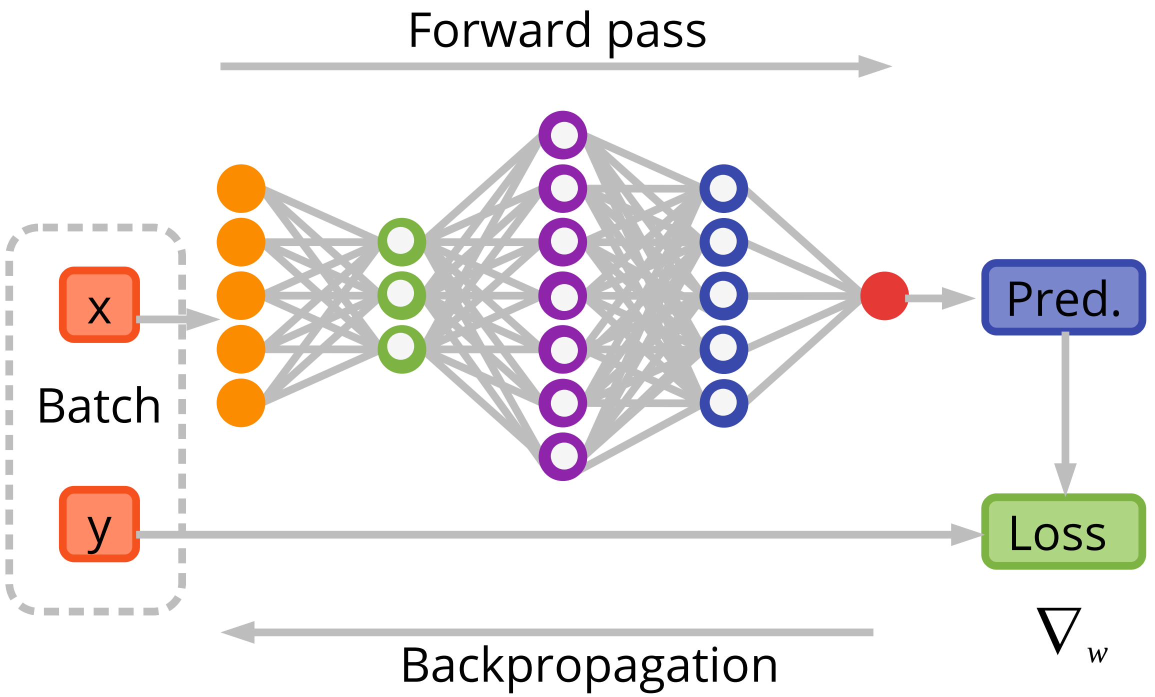
Neural network training pipeline

- Sample batch (input data x and target data y) from training dataset:

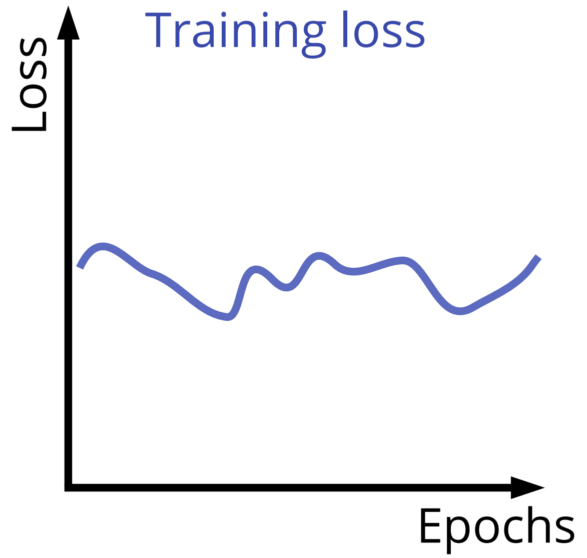
1 epoch

  - Evaluate model on batch input data (=prediction) in forward pass
  - Compute loss on prediction and target y
  - Compute weight gradients with backprop.
  - Modify weights based on gradients and learning rate
  - Repeat for all batches

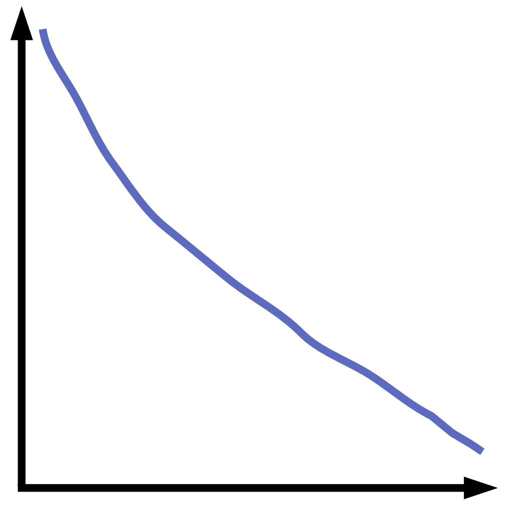
- Repeat for a number of epochs, monitor training and validation loss + metrics
- Stop before overfitting sets in



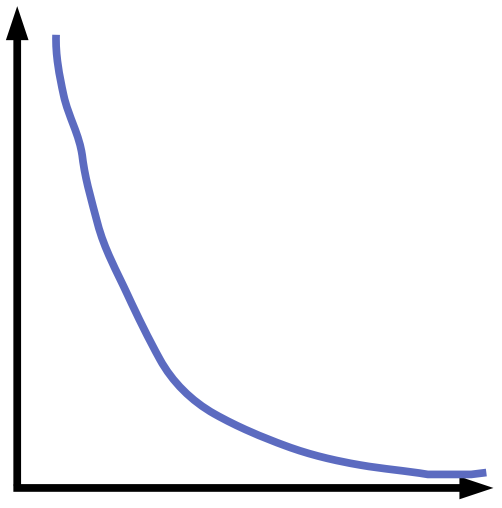
Neural network training process: reading the loss



Model does not learn (Learning too small or too large?)

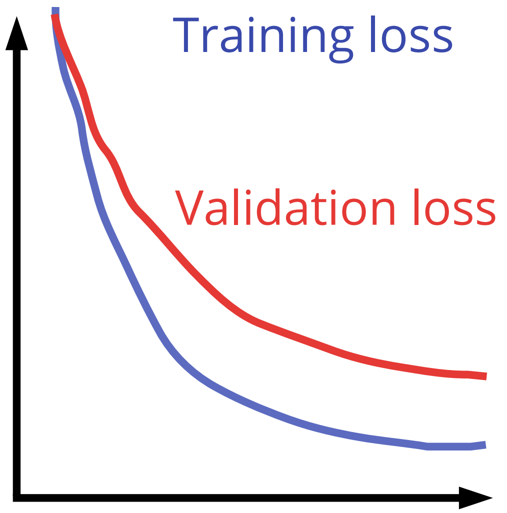


Model learns but could learn some more (more epochs)

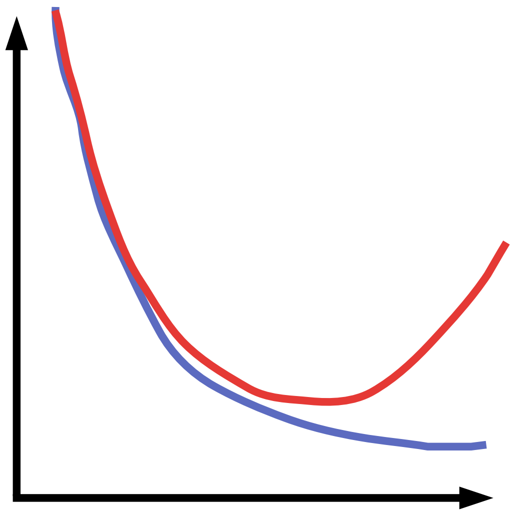


Model learned well (good time to stop learning)

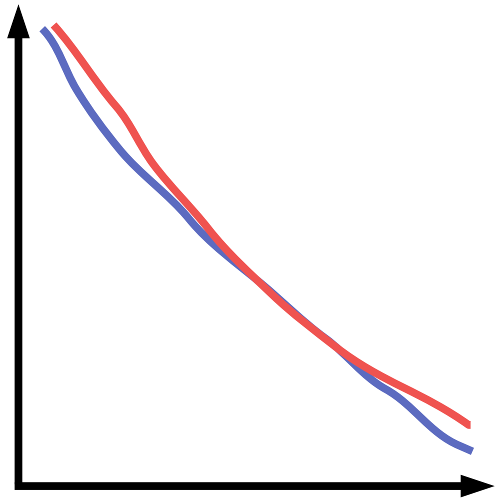
Neural network training process: reading the loss



Well-trained model



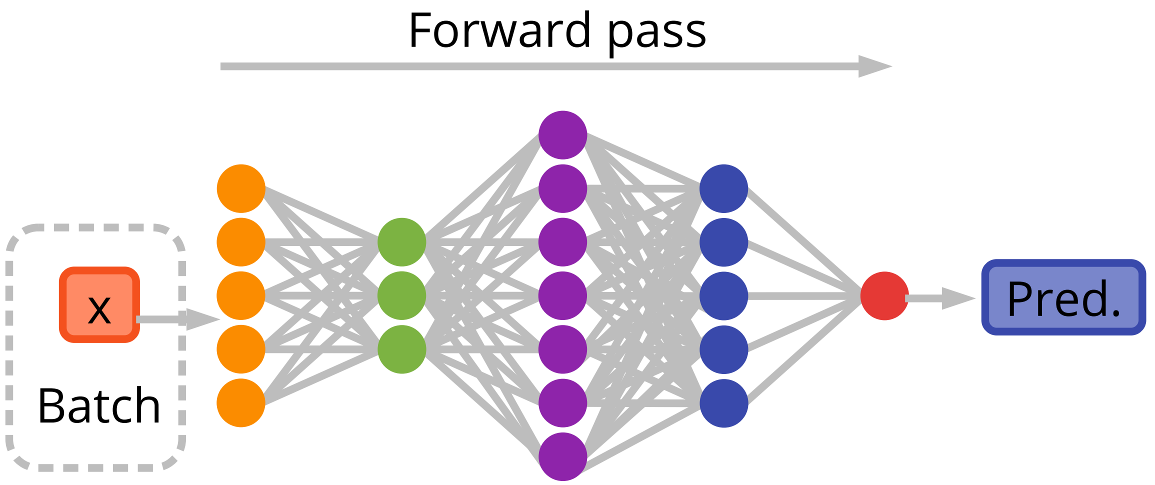
Overfitting



Underfitting

Neural network evaluation pipeline

- Sample batch (input data x) from val/test sample:
  - Evaluate model on batch input data (=prediction) in forward pass
  - Repeat for all batches
- Compute evaluation metric over all batches



That’s all folks!

Today's lecture

Neural Networks

Neurons and Neural Networks

Activation Functions

Loss Functions, Backpropagation and Gradient Descent

Neural Network Training and Evaluation

Next Week:

Lab 3: Neural Networks

Next lecture (May 1)

Convolutional Neural Networks

Convolutions

Convolutional Neural Networks

Semantic Segmentation

Object Detection