

HFT Summer School 2025

Fundamentals of Deep Learning

Michael Mommert

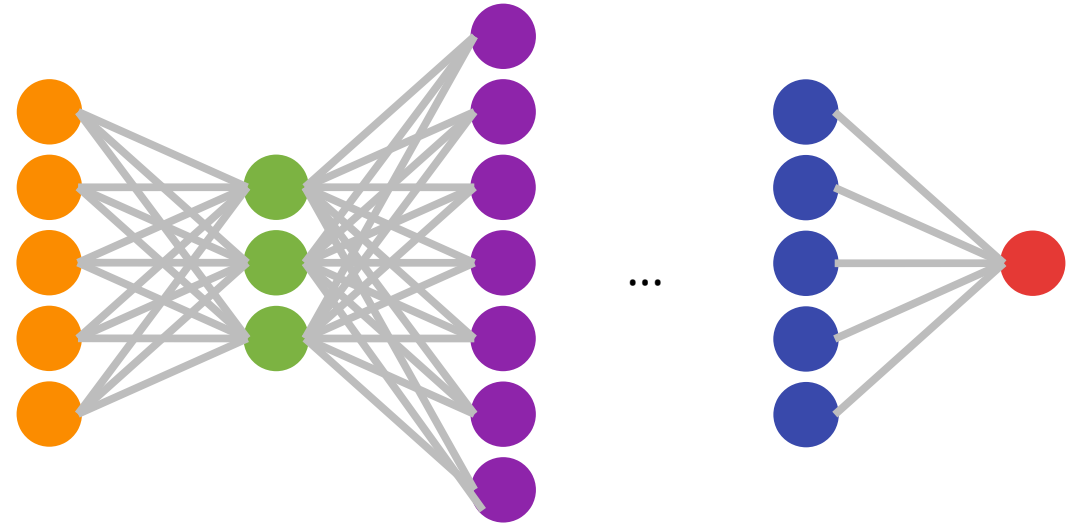
Stuttgart University of Applied Sciences

Learning Objectives

In this lecture you will learn the following:

- What is a neural network?
- How does a neural network work?
- How does a neuron work?
- What is an activation function and why is it necessary?
- How can we train a neural network?

Neurons and Neural Networks

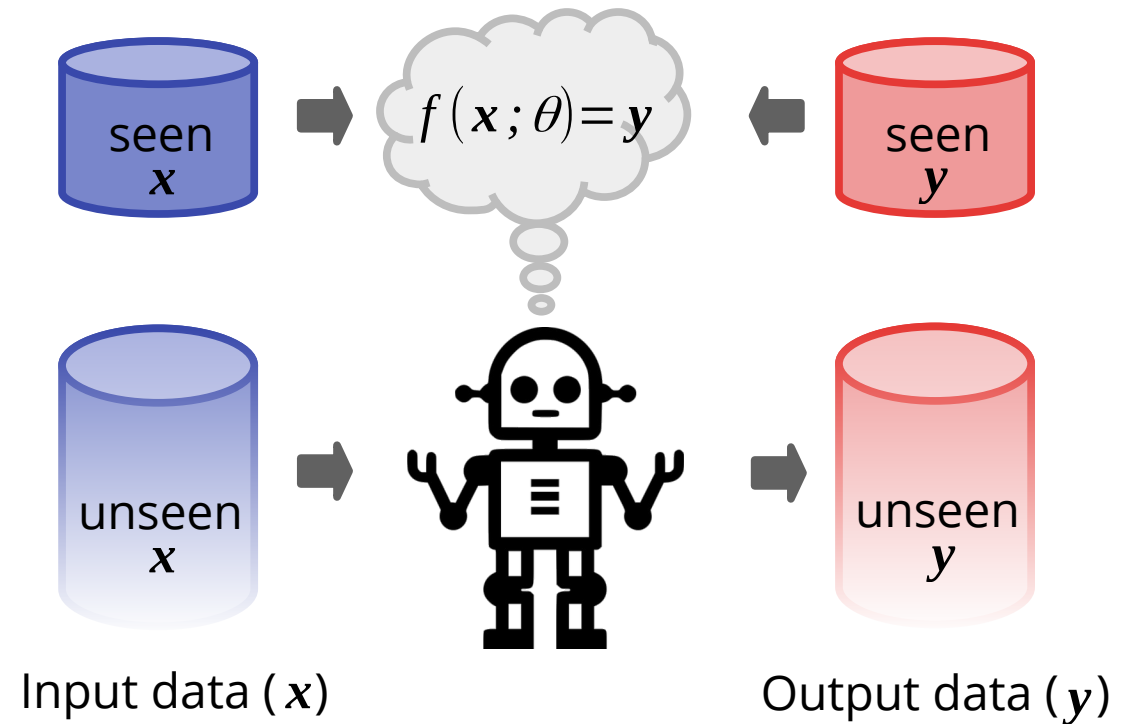


Machine learning so far...

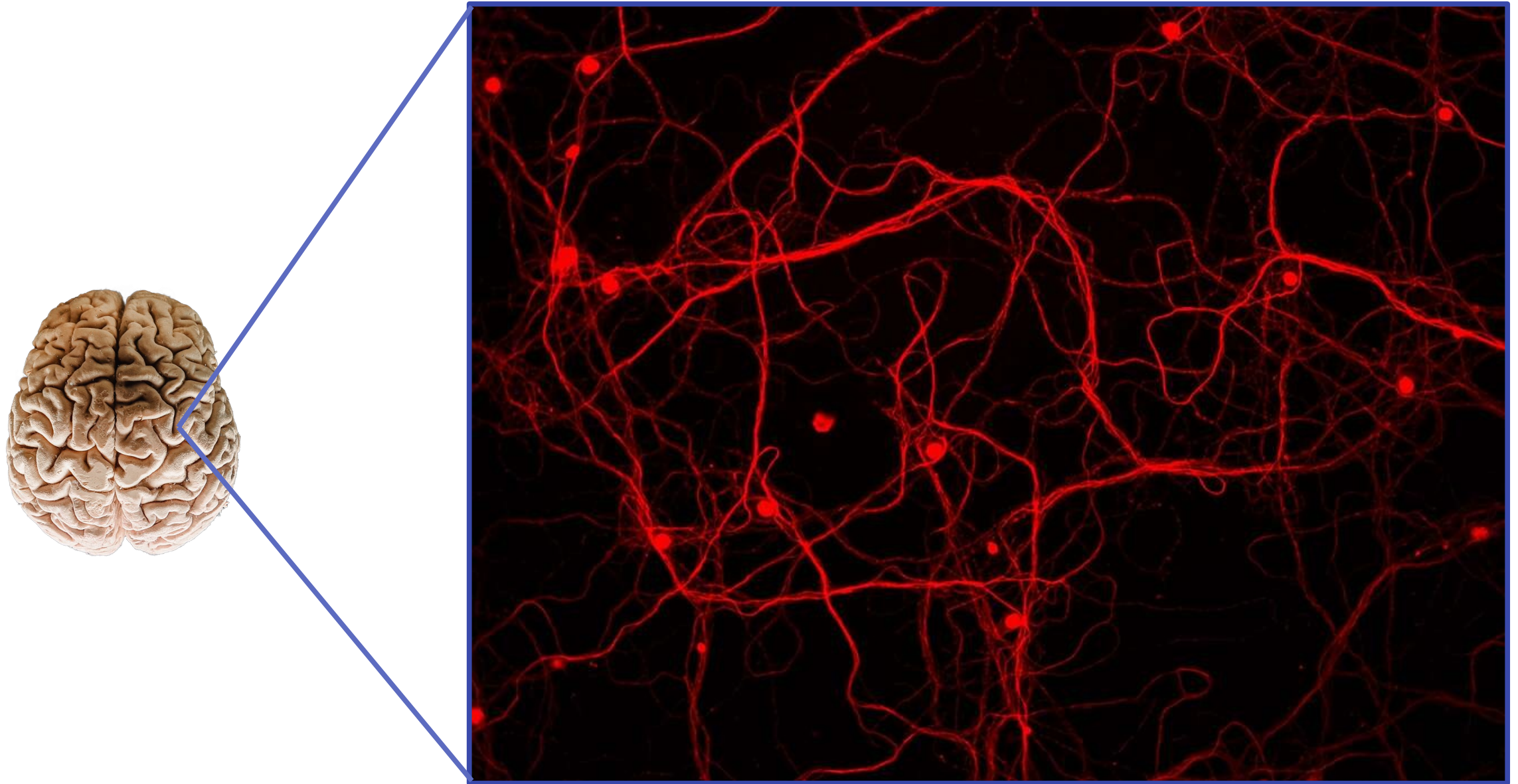
Have the models we talked about so far really “learned “ anything?

- k -NN computes distances and compares a distribution of unseen data points with a distribution of seen data points
- Linear models are simply fitted to seen data
- Tree-based models identify and memorize patterns relevant to the task

Most traditional Machine Learning methods do not learn in an iterative process



How does the human brain work?



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

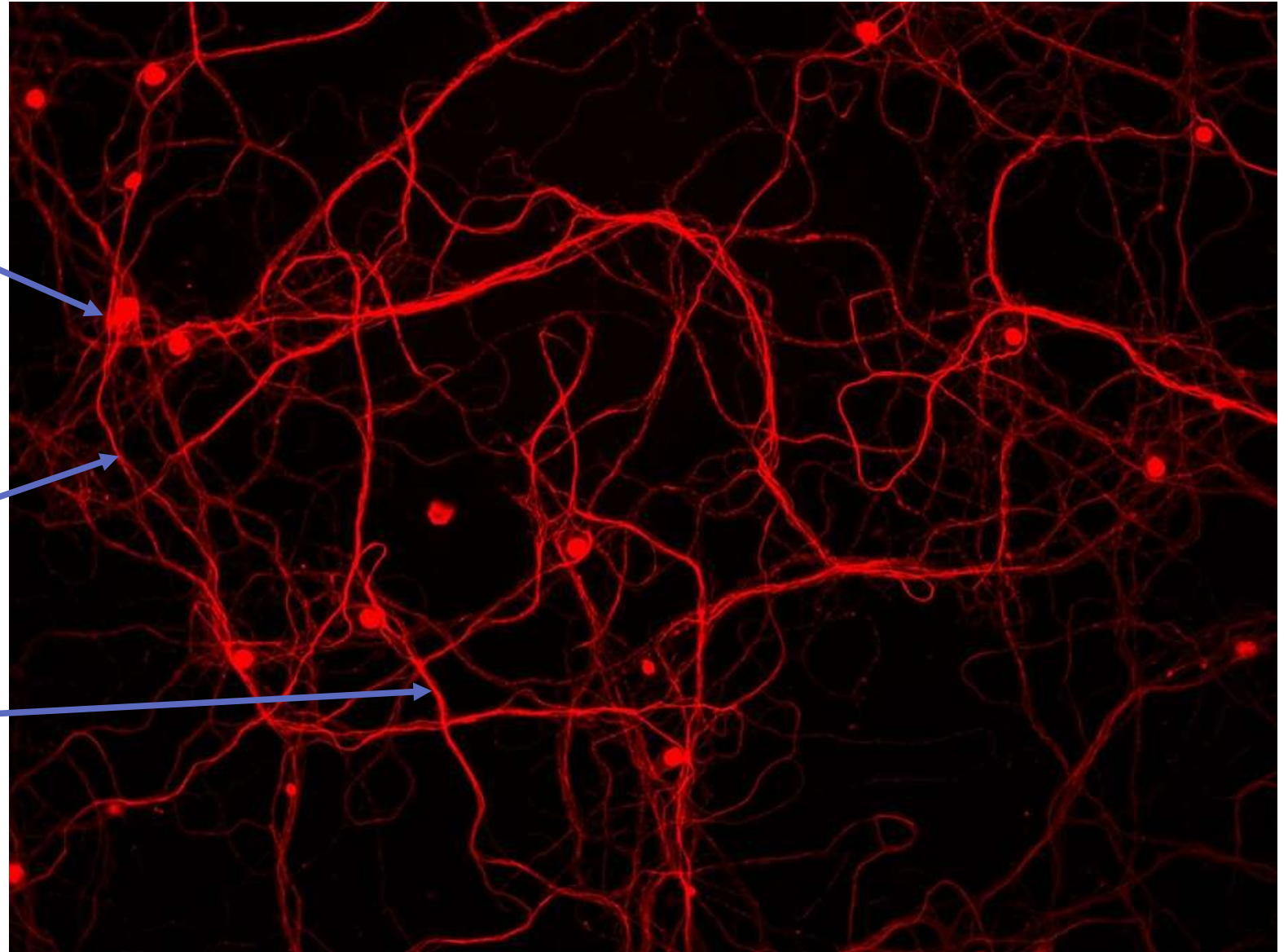
Axons provide long-
distance connections

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

Neurons

Dendrites

Axons



How does the human brain work?

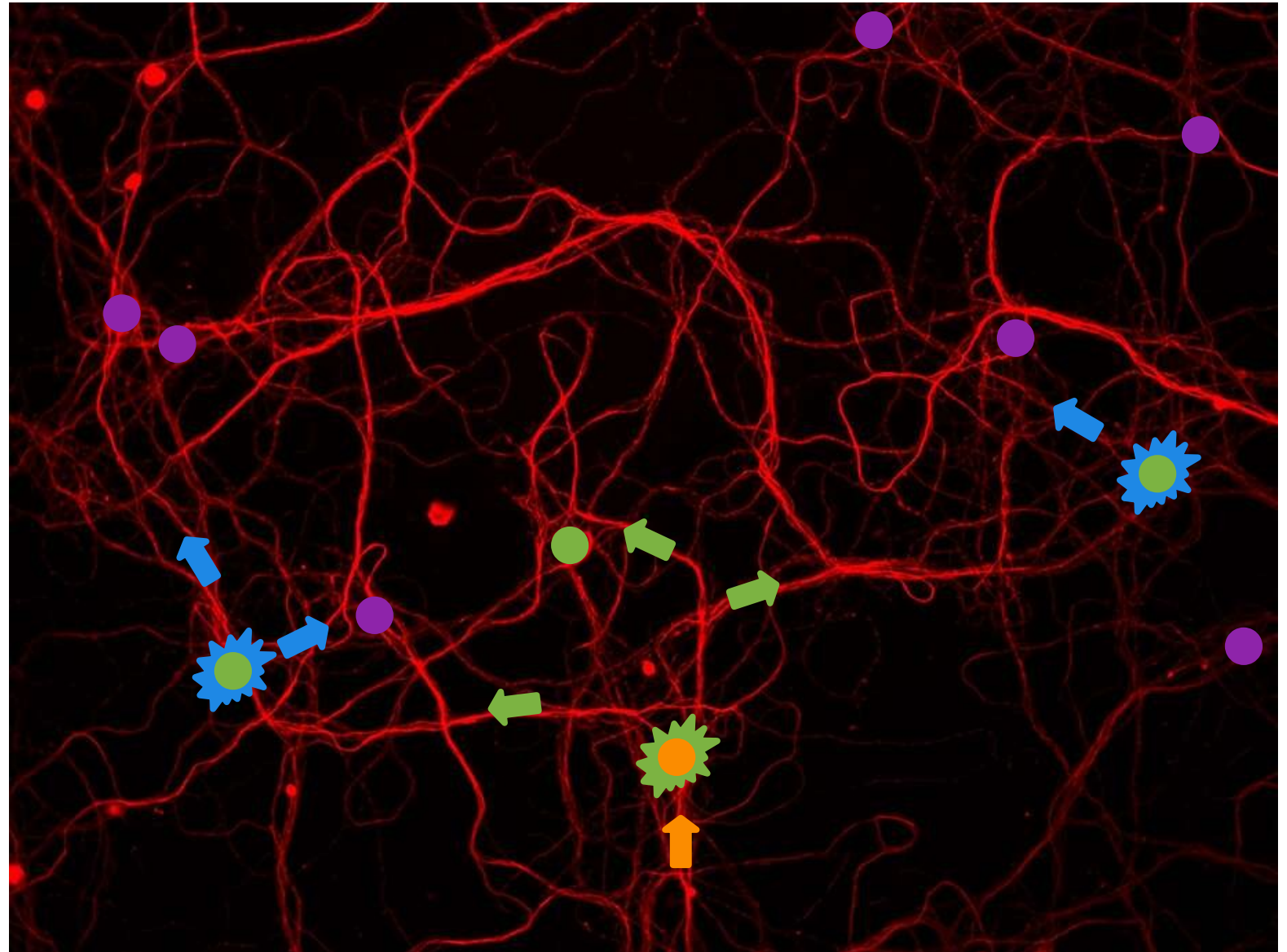
Information is passed to neurons through electrical signals.

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.

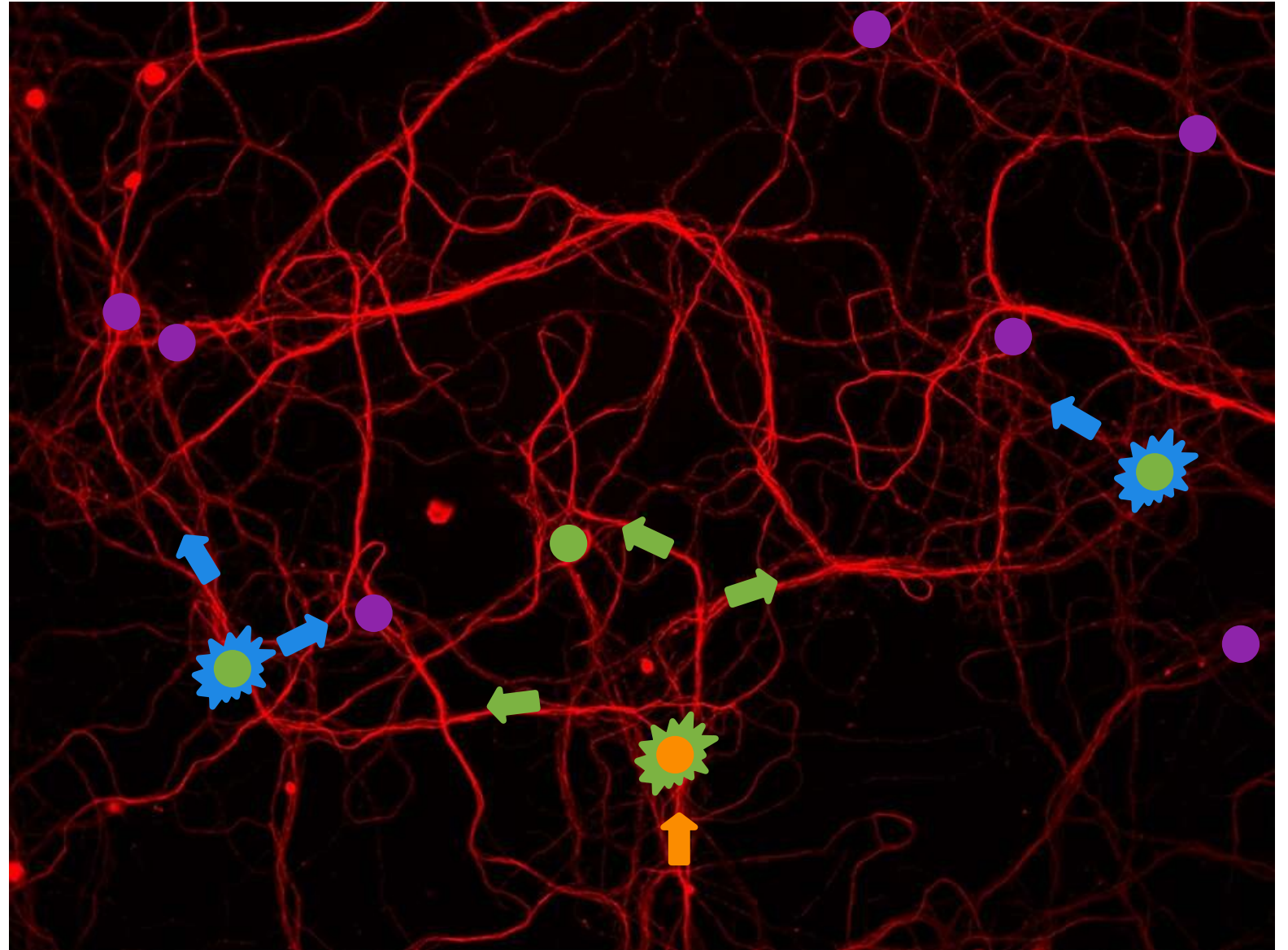
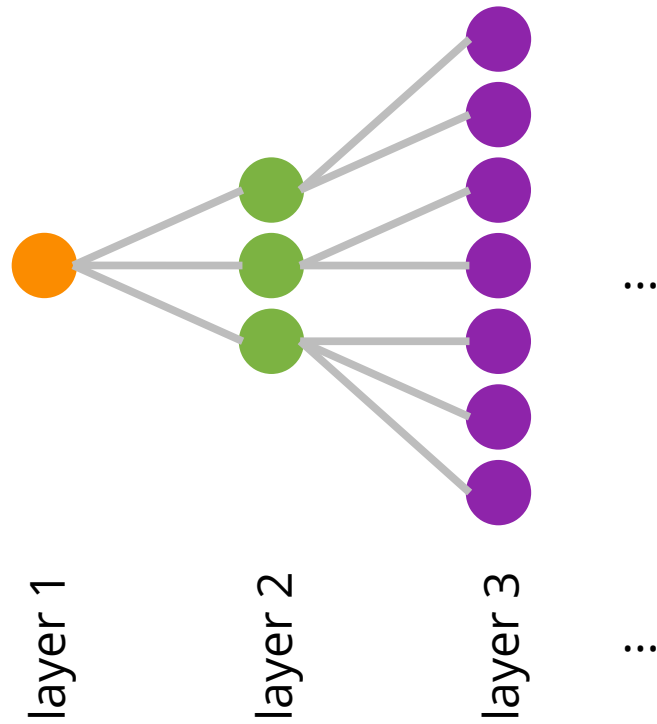
Connected neurons absorb the incoming signals and process them. Some of them will fire, but not all.

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

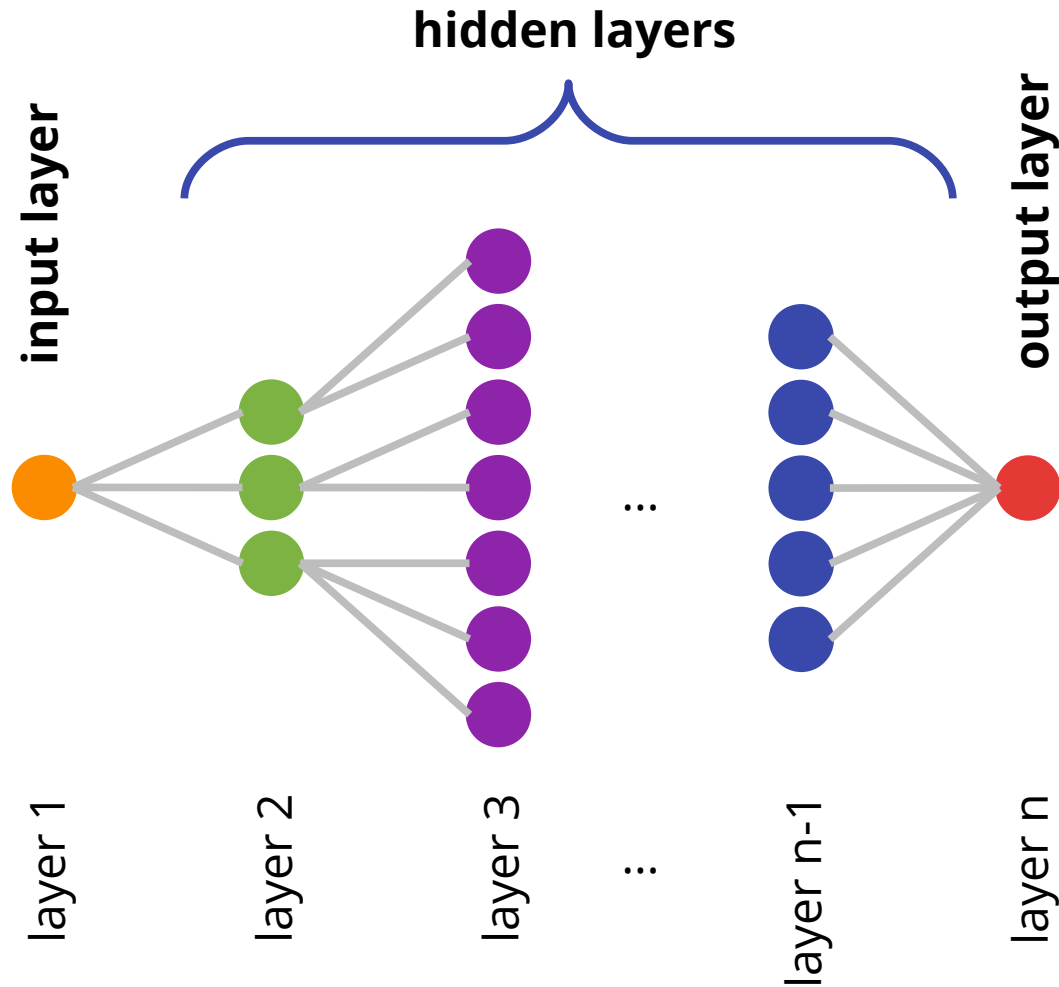


How does the human brain work?

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



Neural network

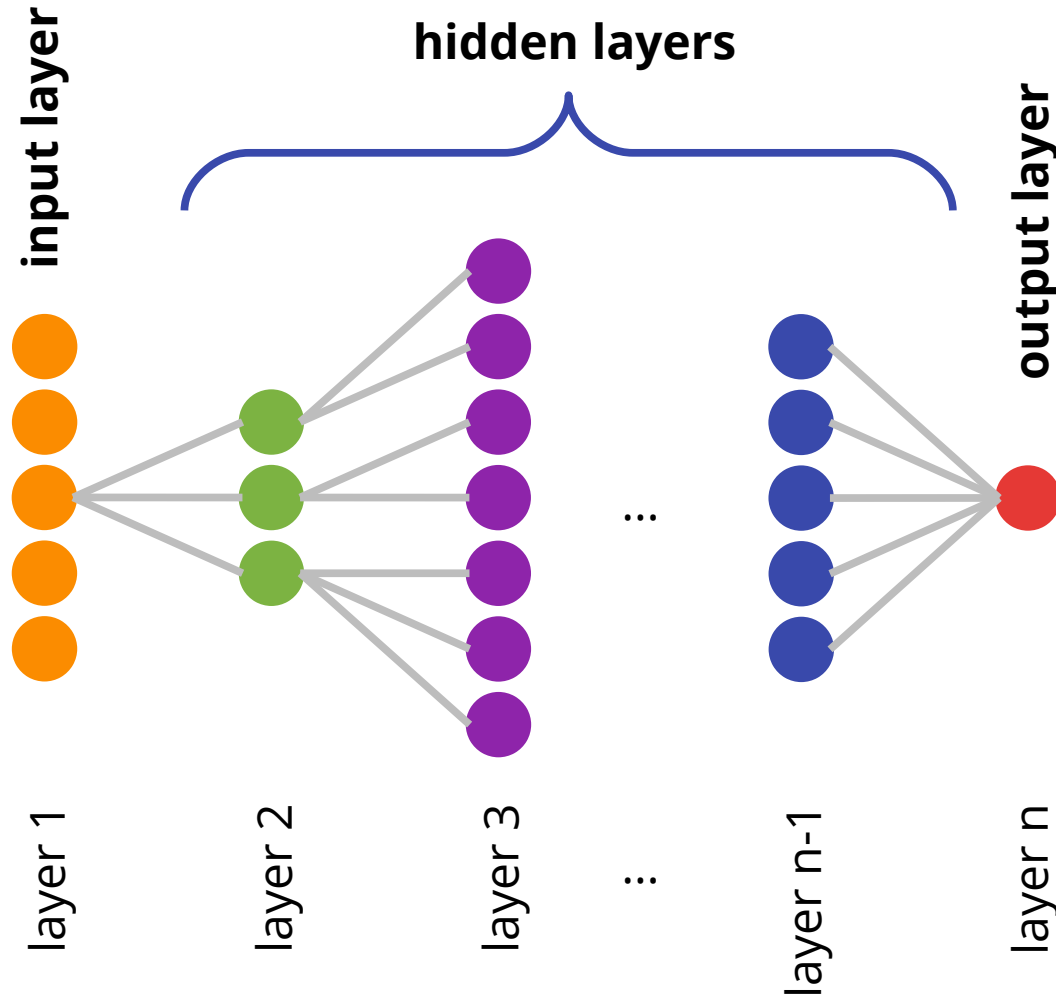


This is a neural network: a **deep cascade** of layers of neurons.

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:

Neural network



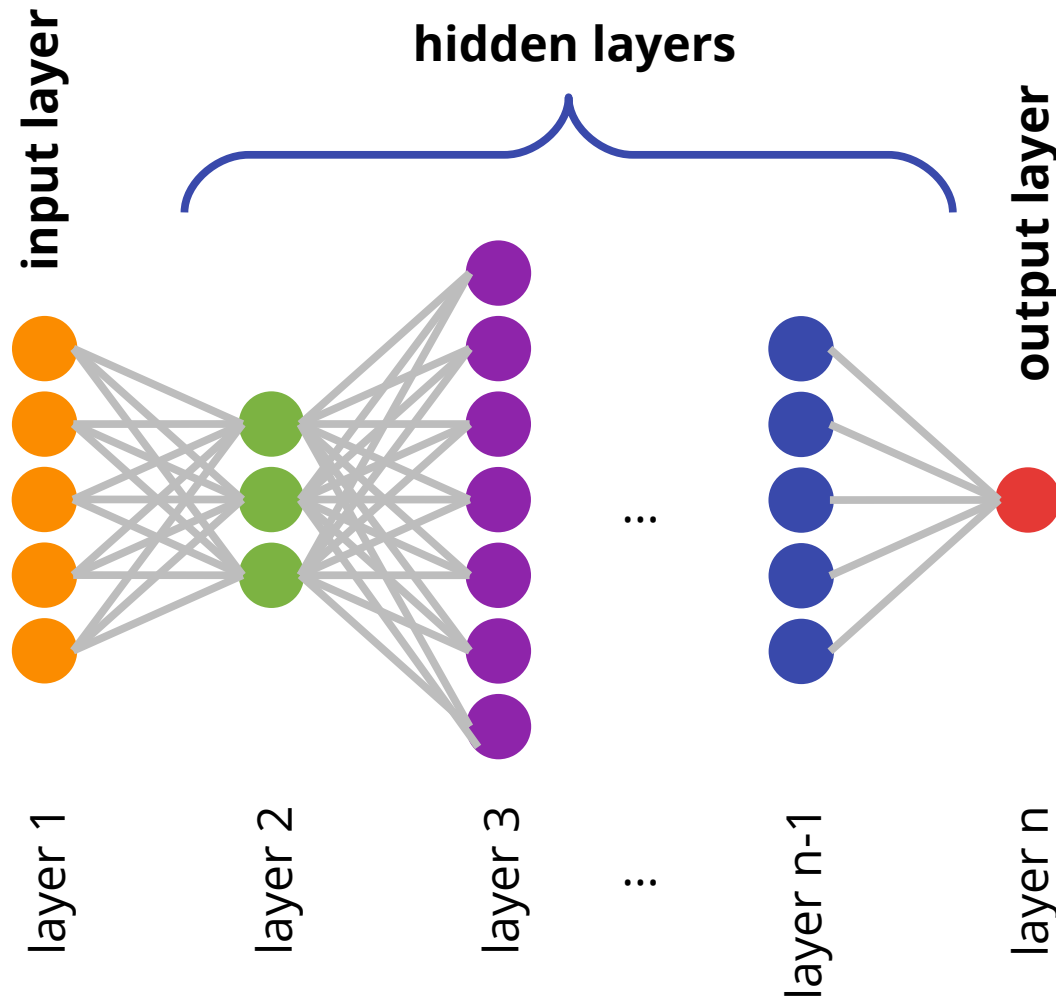
This is a neural network: a **deep cascade** of layers of neurons.

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.

Neural network



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

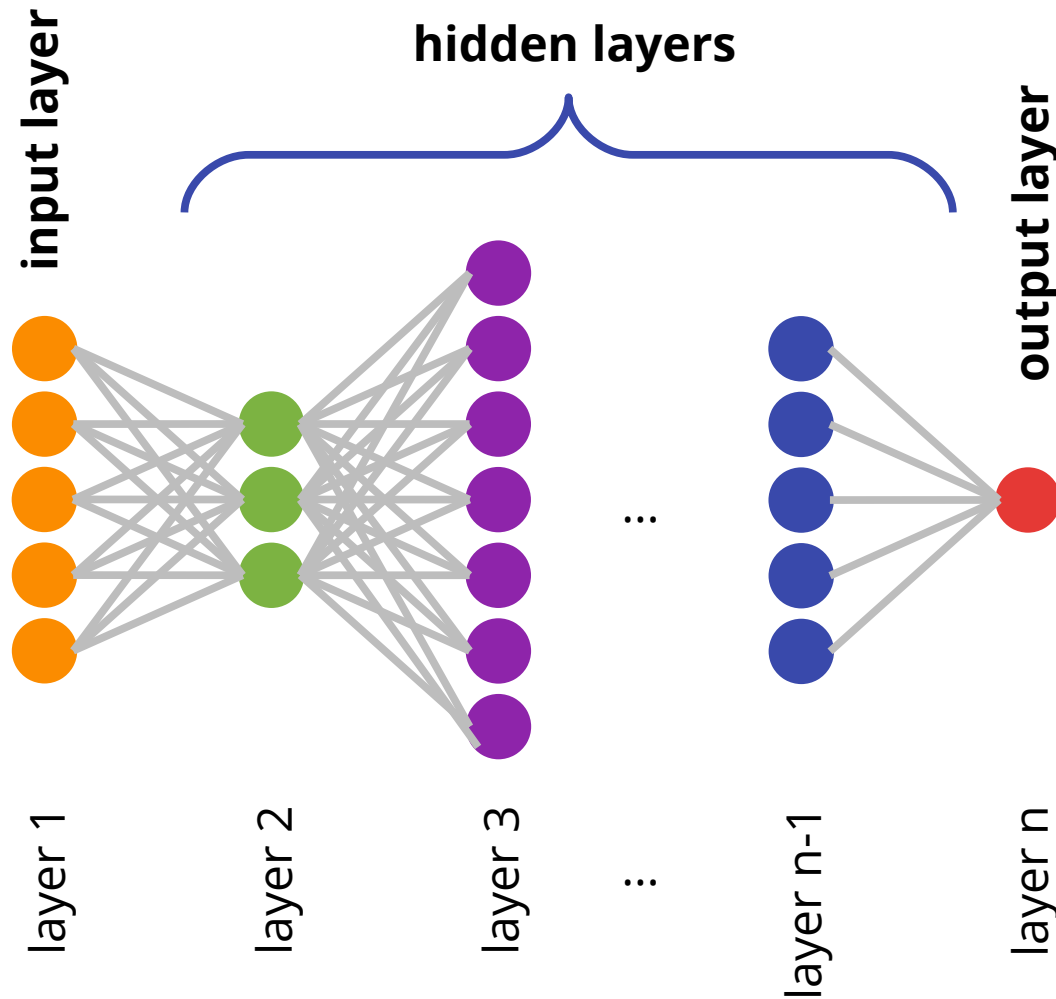
So that the network can learn a meaningful task, it requires **input data**, which it processes in its **hidden layers**, and generates **output data**.

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**.

Fully-connected neural network

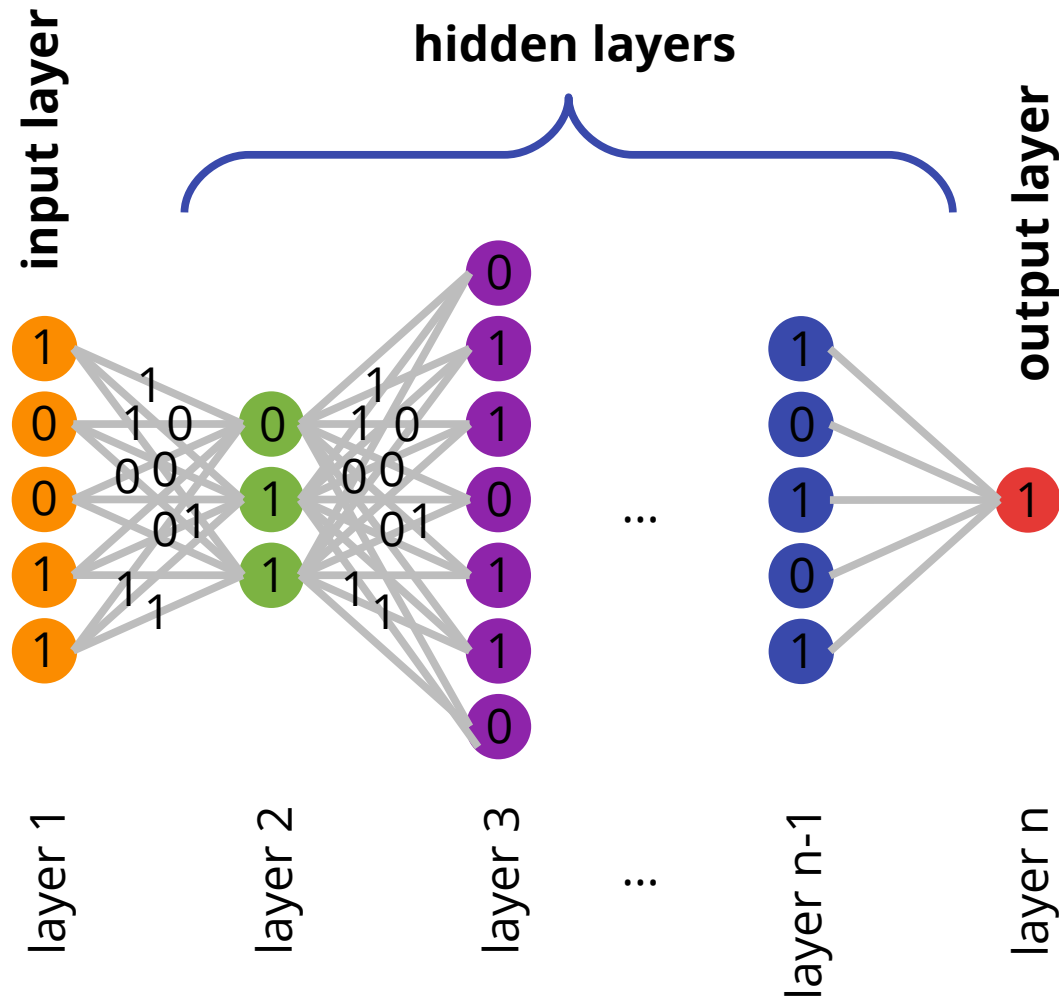


In this network type, all neurons of one layer are connect to all neurons of the previous layer and all neurons of the following layer.

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)

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: $x = [1, 0, 0, 1, 1]$)

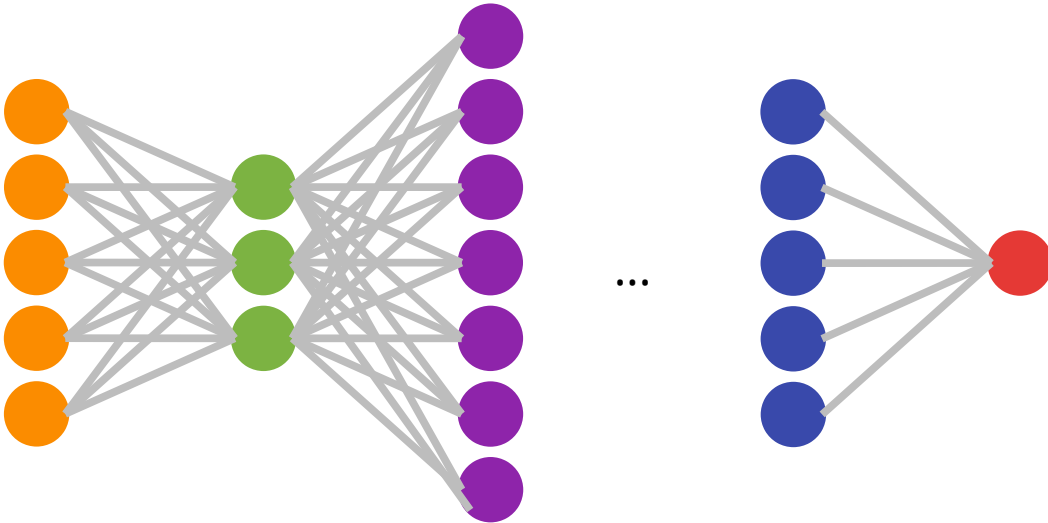
All inputs are seen by each neuron in the underlying layer.

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?

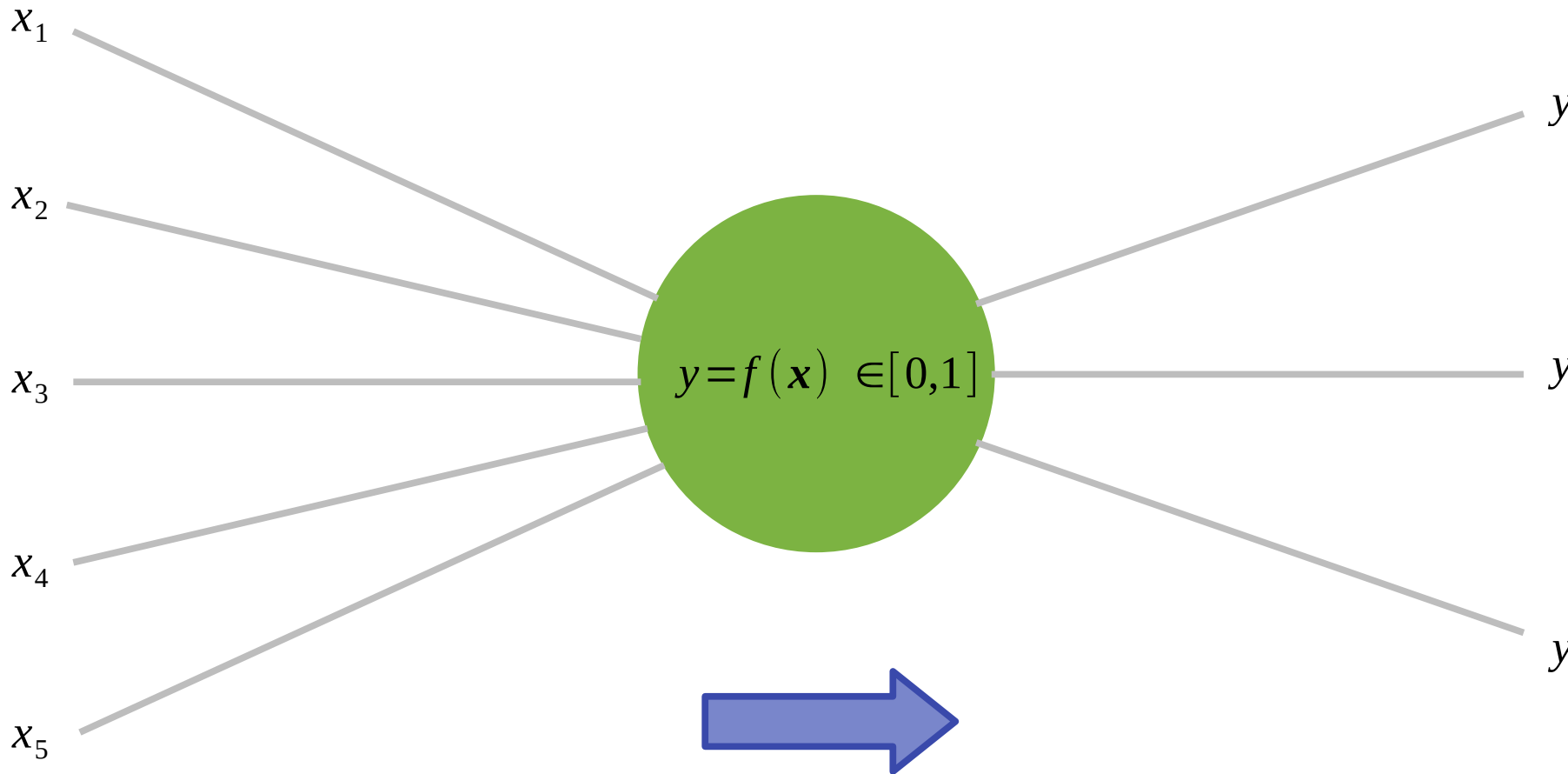
→ **Connectionism**

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:

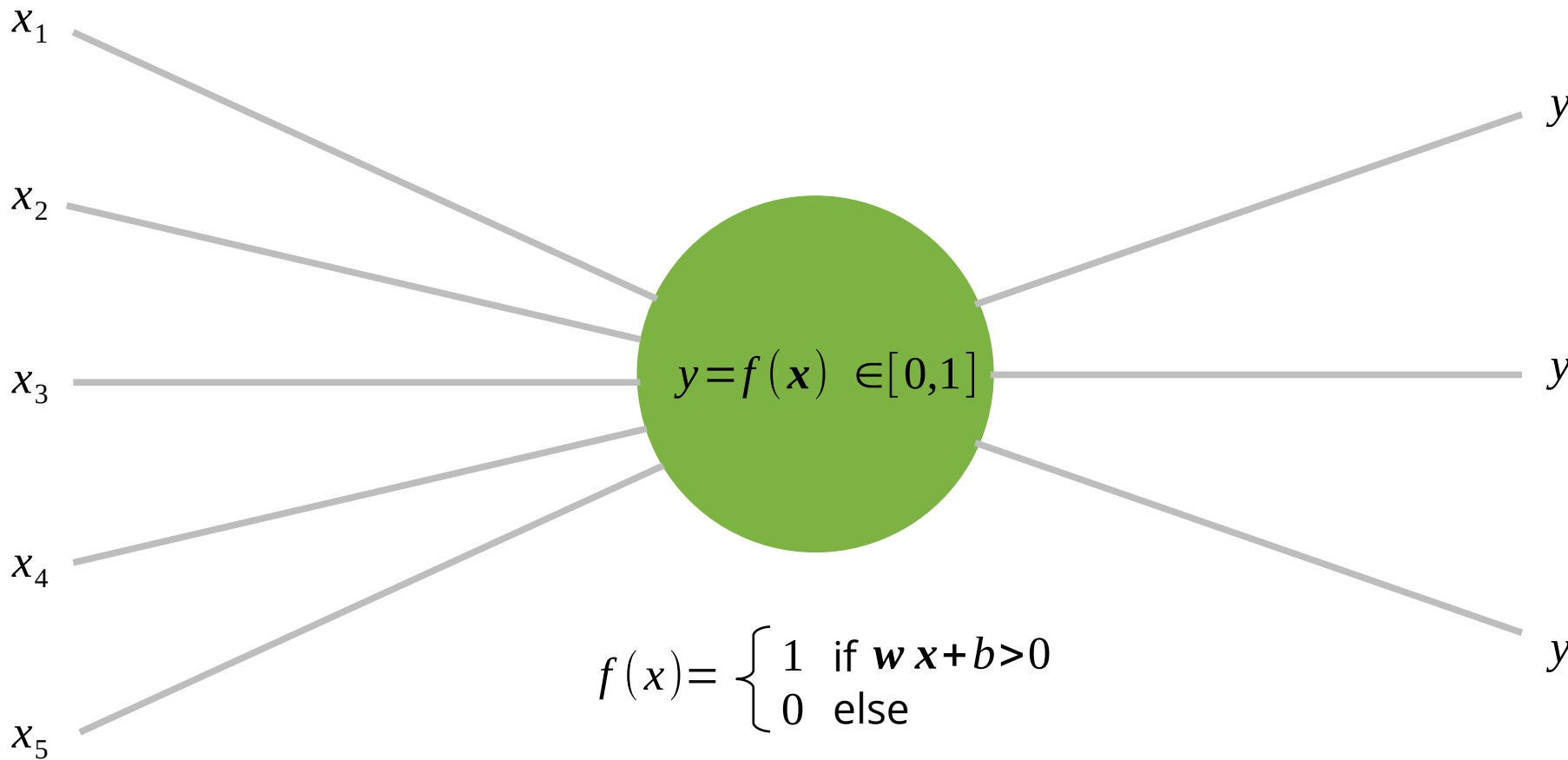
- How to implement neurons?
- How to train the network?

A general neuron



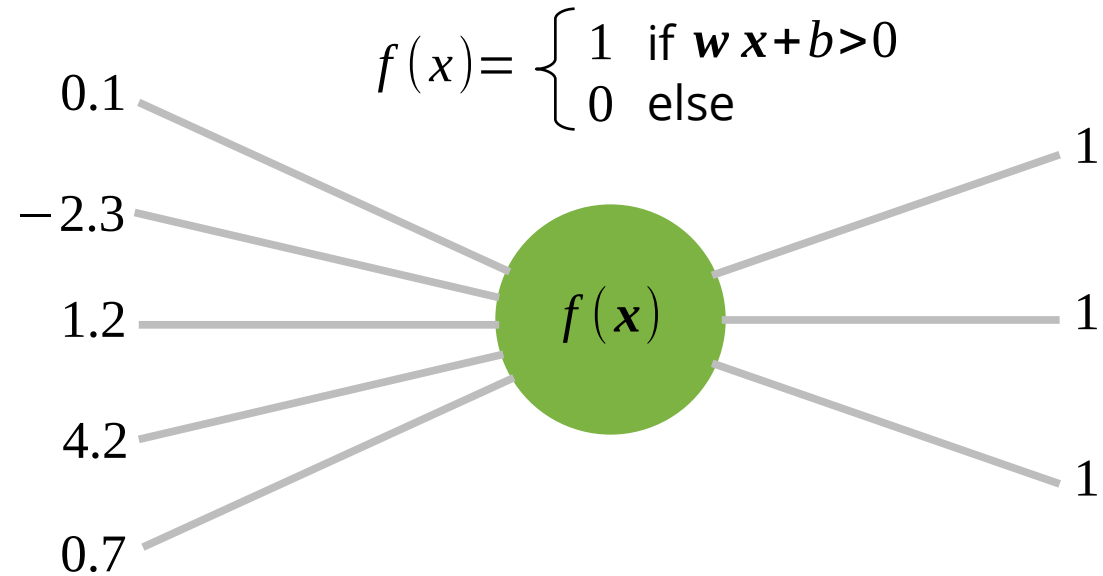
A neuron takes in a vector of values, processes them and returns a binary signal based on its learned behavior, which is then passed on to all neurons in the following layer.

An artificial neuron: the Perceptron



Compute the dot product between input variables $\mathbf{x}$ and **weights** $\mathbf{w}$, 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 modulates the output of the Perceptron in a **non-linear** way.

The Perceptron: an example



$$\mathbf{x} \cdot \mathbf{w} + b = x_1 w_1 + x_2 w_2 + \dots + x_5 w_5 + b = 0.08$$

$\mathbf{x} \cdot \mathbf{w} + b > 0 \quad \Rightarrow \quad f(\mathbf{x}) = 1 \quad \text{The neuron fires!}$

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

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

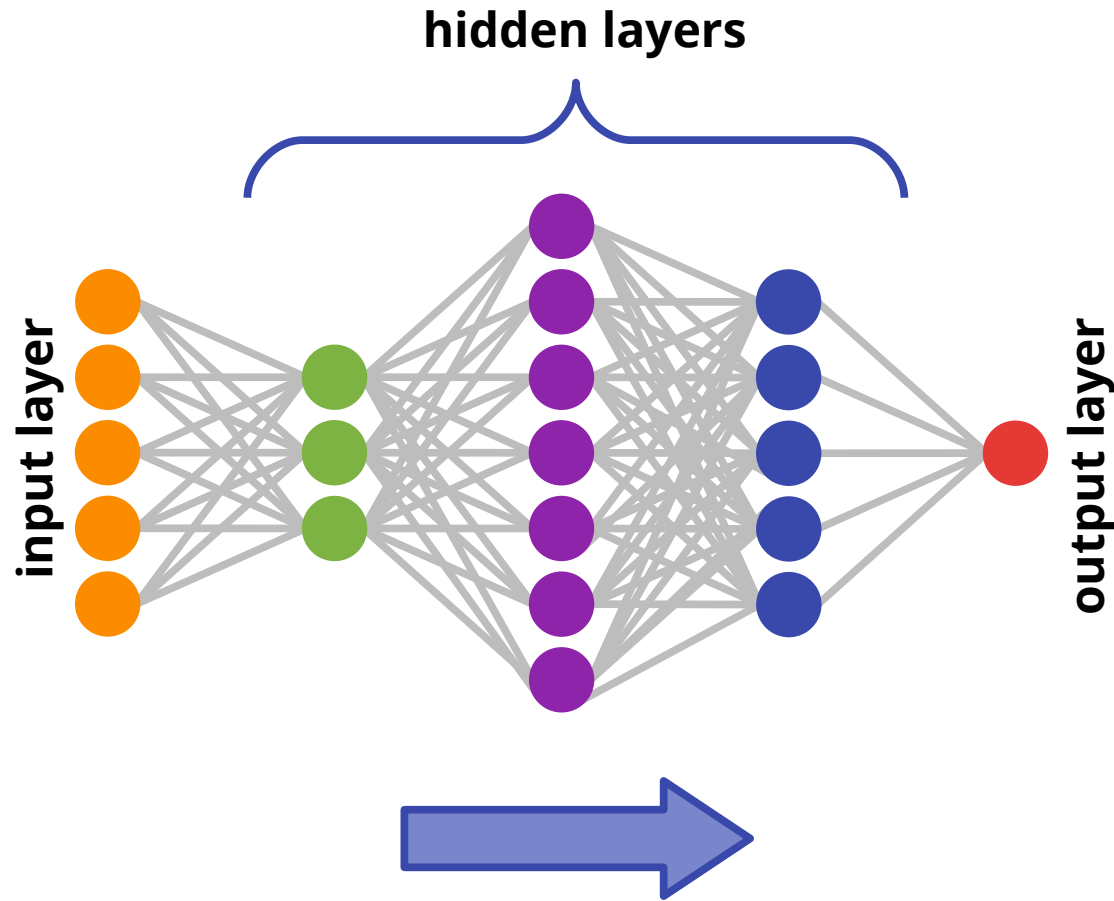
Bias: $b = -0.7$

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

A single Perceptron is not that powerful.

But we can combine a number of them in a Multilayer Perceptron...

Multi-layer Perceptron (MLP)



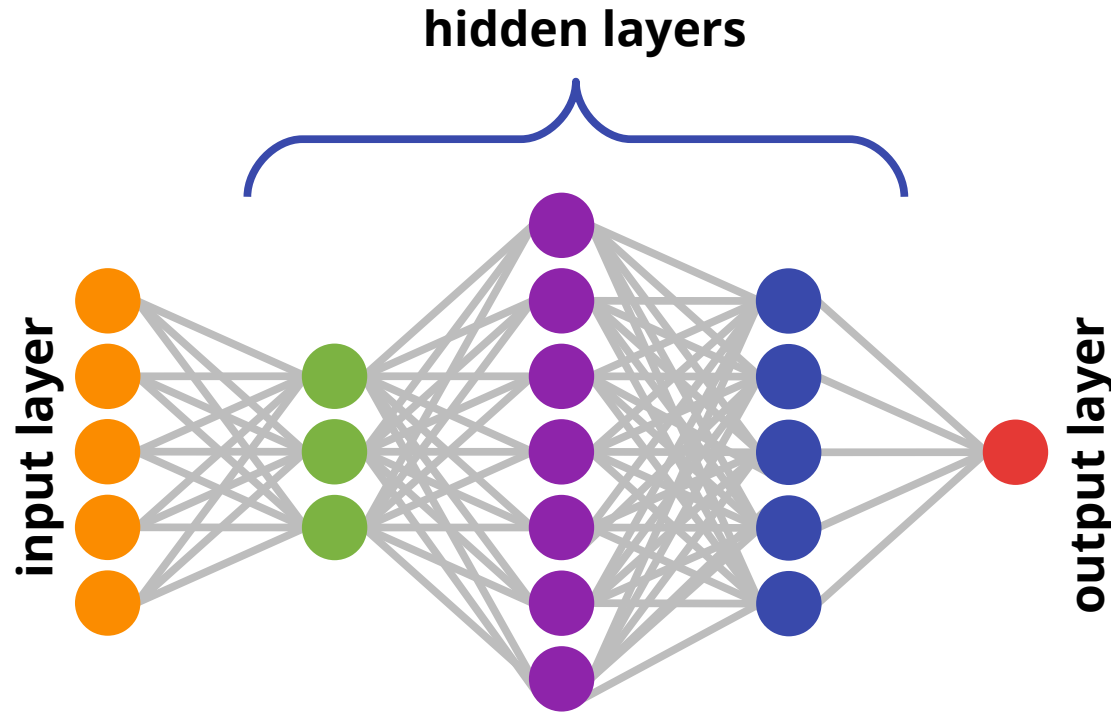
MLPs are simple **feed-forward** neural networks: information traverses the graph in only one direction

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

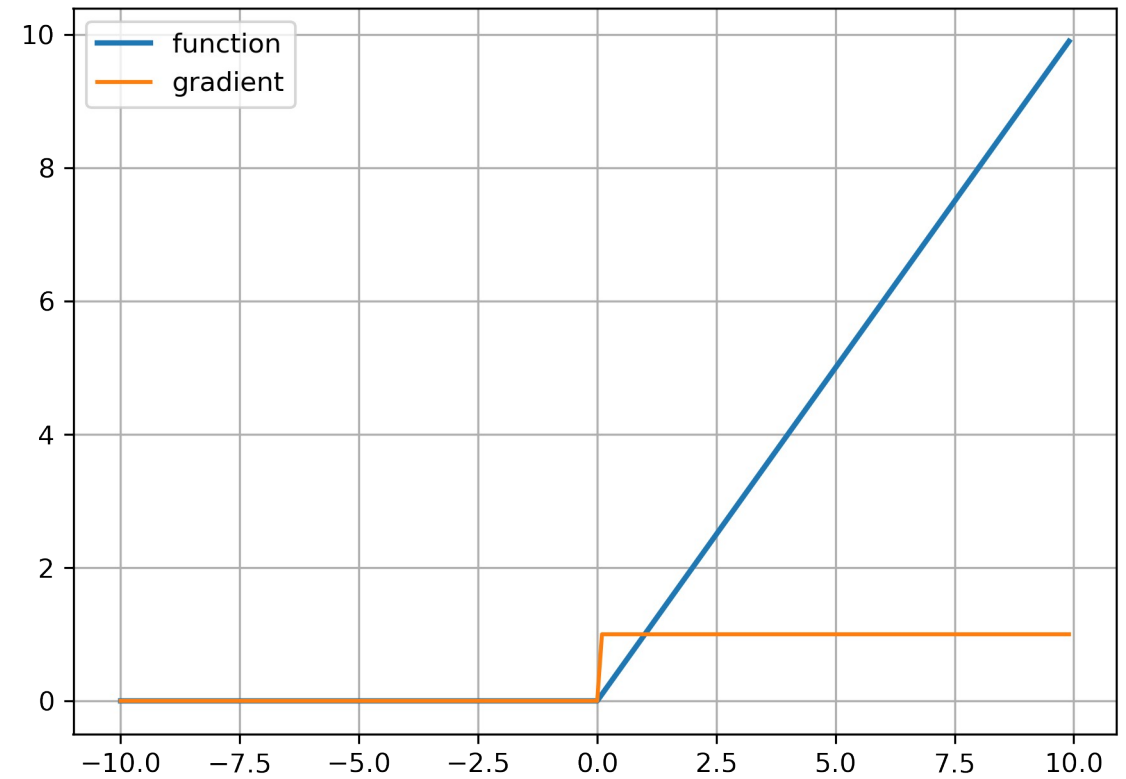
Each neuron is defined by $\mathbf{xw} + b > 0$, i.e., $\mathbf{w}$ has the same number of parameters as the number of input features. Therefore, the total number of weight parameters in this network is:

$$(5 + 1) \times 3 + (3 + 1) \times 7 + (7 + 1) \times 5 + (5 + 1) = 92 \text{ parameters}$$

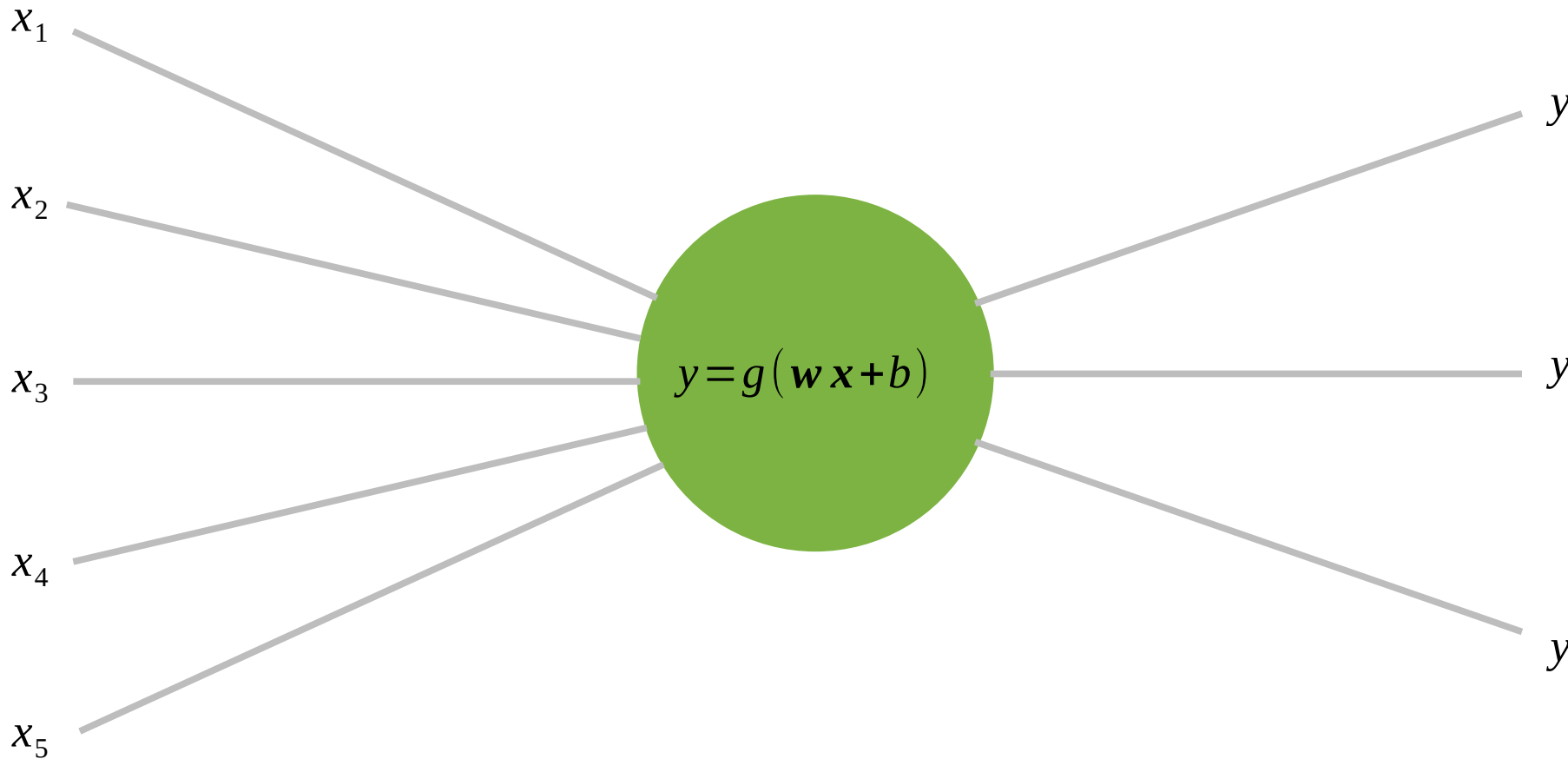
Neurons and neural networks: summary

- 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 large number of Perceptron can be combined into a Multi-layer Perceptron, which is a fully-connected neural network.

Activation Functions



Activation function



The activation function defines when a neuron “fires”. Activation functions should be non-linear for a very important reason...

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

Non-linearity increases the model's **capacity**.

Let's assume, that the neurons in a neural network do not have non-linear activations. What would be the consequence?

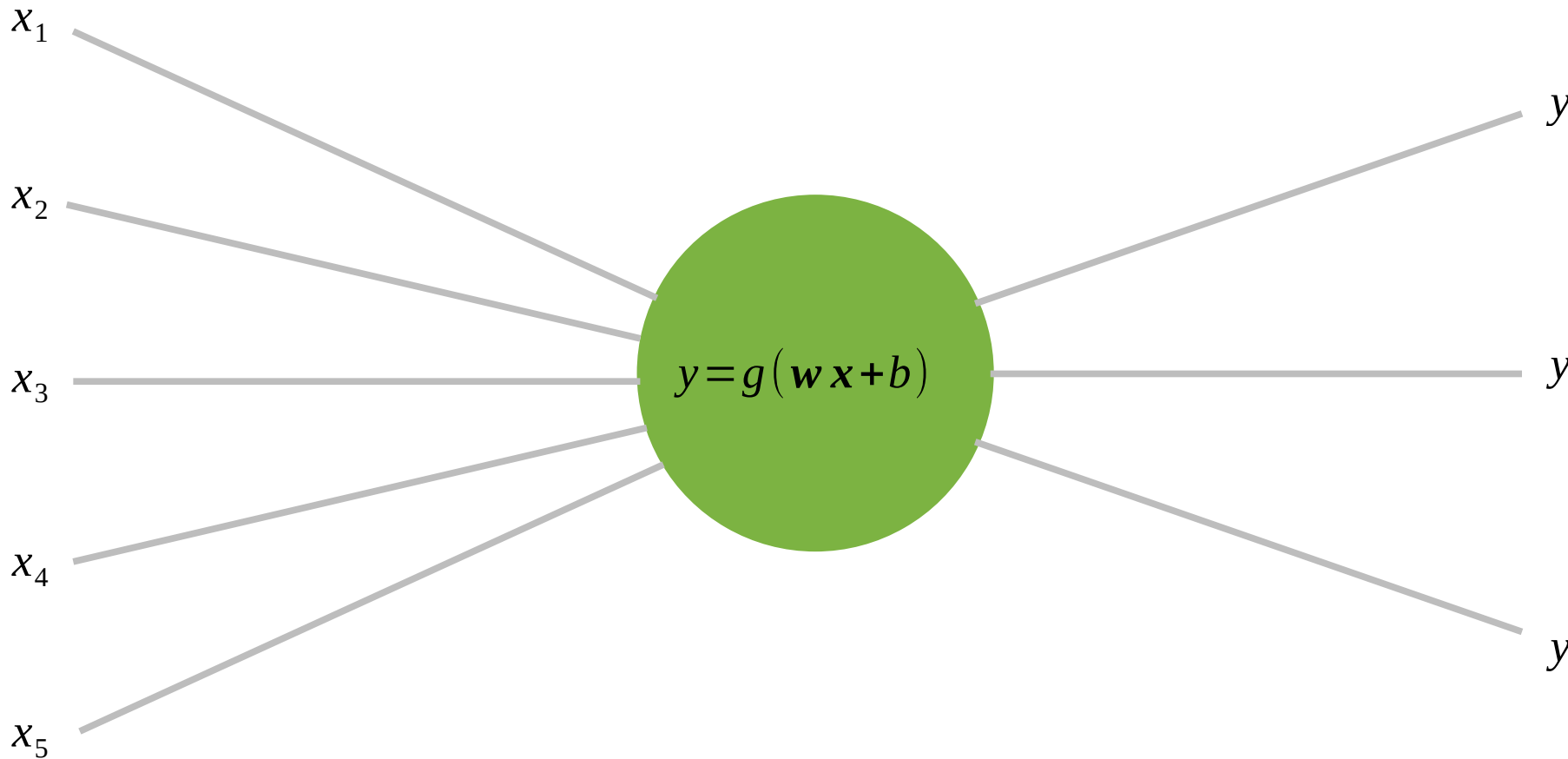
Each layer performs the following well-known computation: $y_i(x) = w_i \cdot x + b_i$

If we stack 3 layers, the output would look like this: $y_3 \otimes y_2 \otimes y_1(x) = w_3 \cdot (w_2 \cdot (w_1 \cdot x + b_1) + b_2) + b_3 = \tilde{w} \cdot x + \tilde{b}$

→ If we do not include non-linear activation functions, the entire network reduces to a single linear function.

To learn more complex problems than linear problems, non-linear activations are necessary.

Activation function

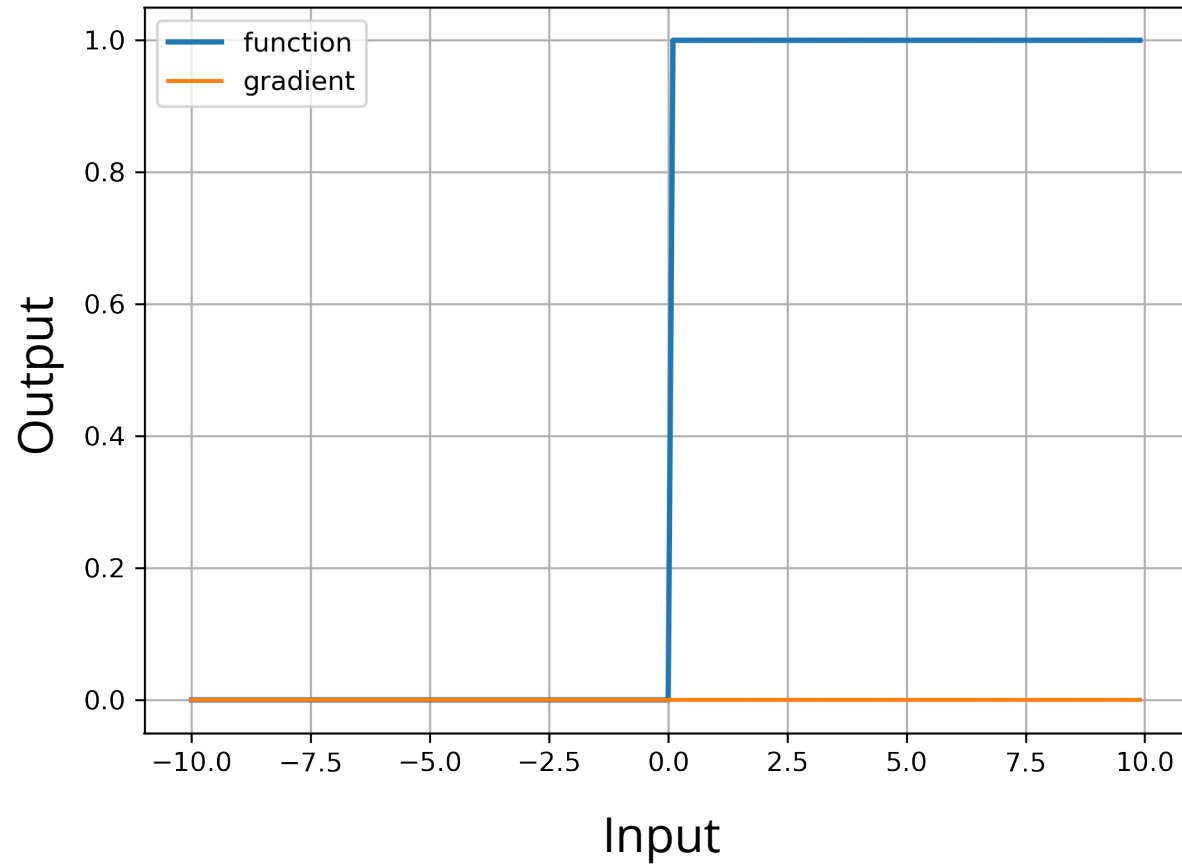


So far, we used a simple step function as our activation function:

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

Let's have a look at this function and some alternatives...

Activation functions: Step function



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

(+): simple to implement

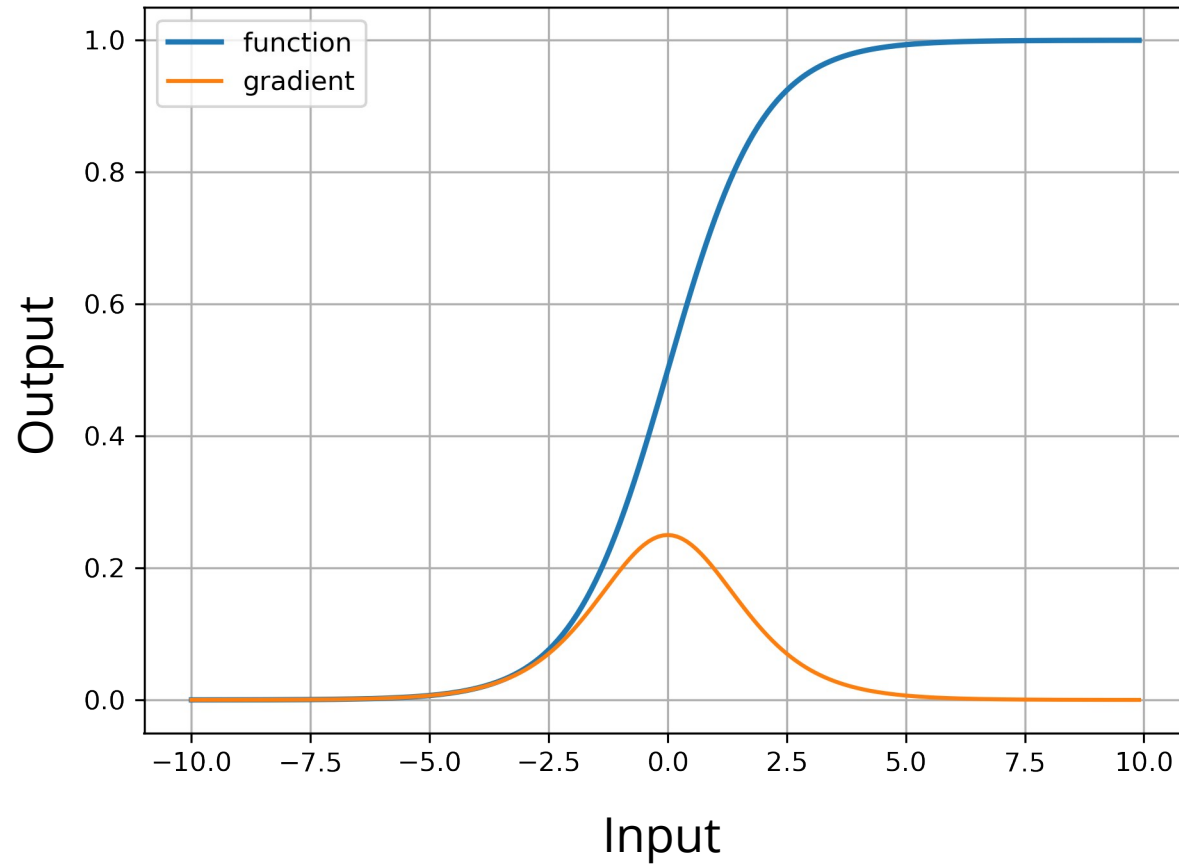
(+): computationally inexpensive

(-): only binary (discrete) output

(-): no gradient

↑ We will see later that it is of paramount importance to be able to compute a gradient of your activation function...

Activation functions: Sigmoid function



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

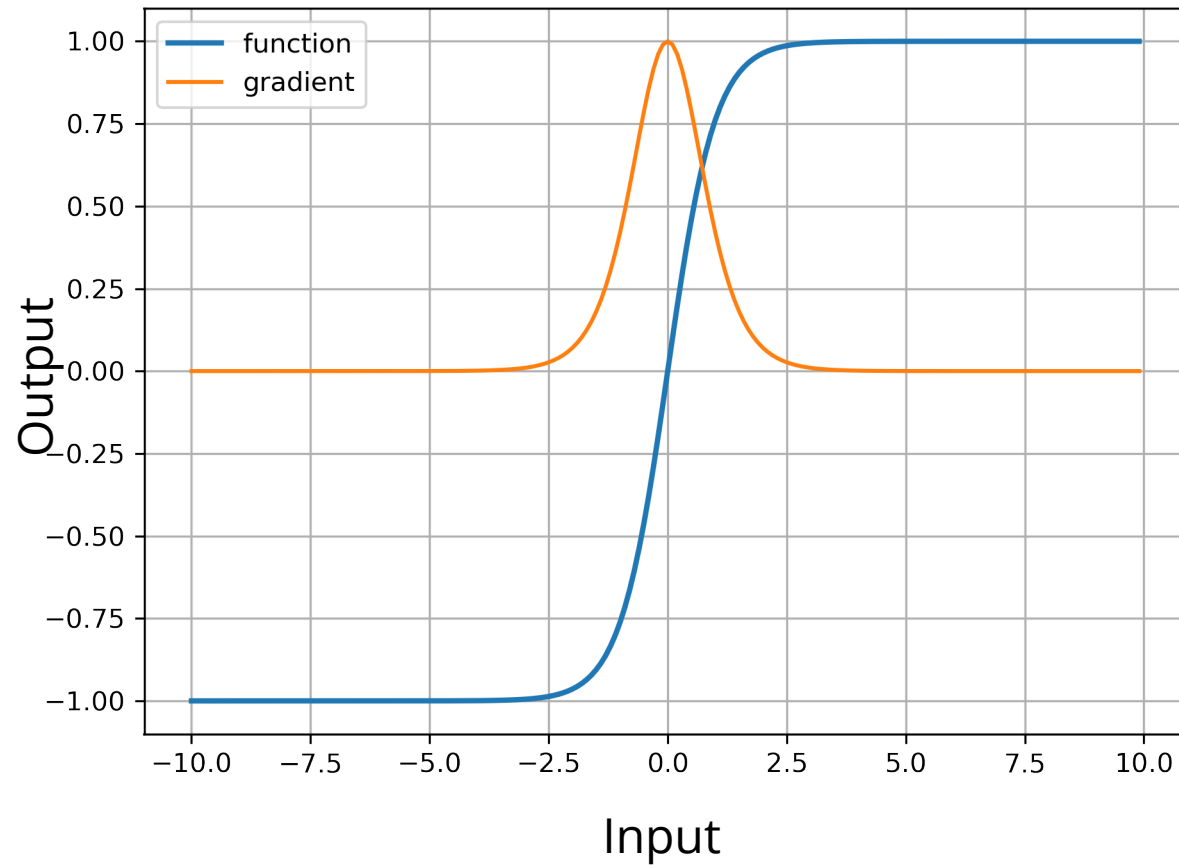
(+): continuous non-linear function

(+): gradient defined

(-): computationally expensive

Signals
between
layers are
continuous,
not binary!

Activation functions: Tanh function



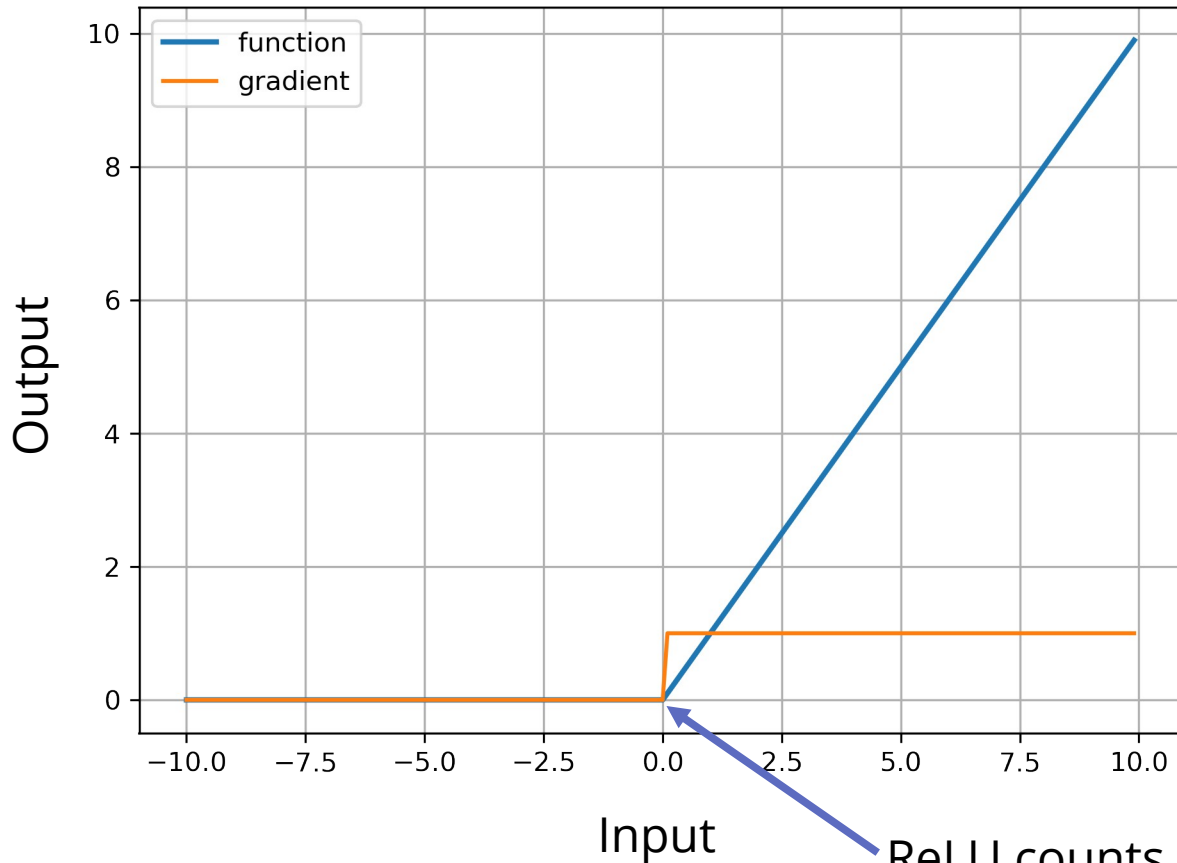
$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$$

(+): continuous non-linear function

(+): gradient defined

(-): computationally expensive

Activation functions: Rectified Linear Unit (ReLU) function



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

(+): continuous non-linear function

(+): gradient defined and piece-wise constant!

(+): computationally inexpensive

ReLU counts as a non-linear activation function because of this kink here; mathematically, the gradient is not defined here, we simply set it to zero

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”.

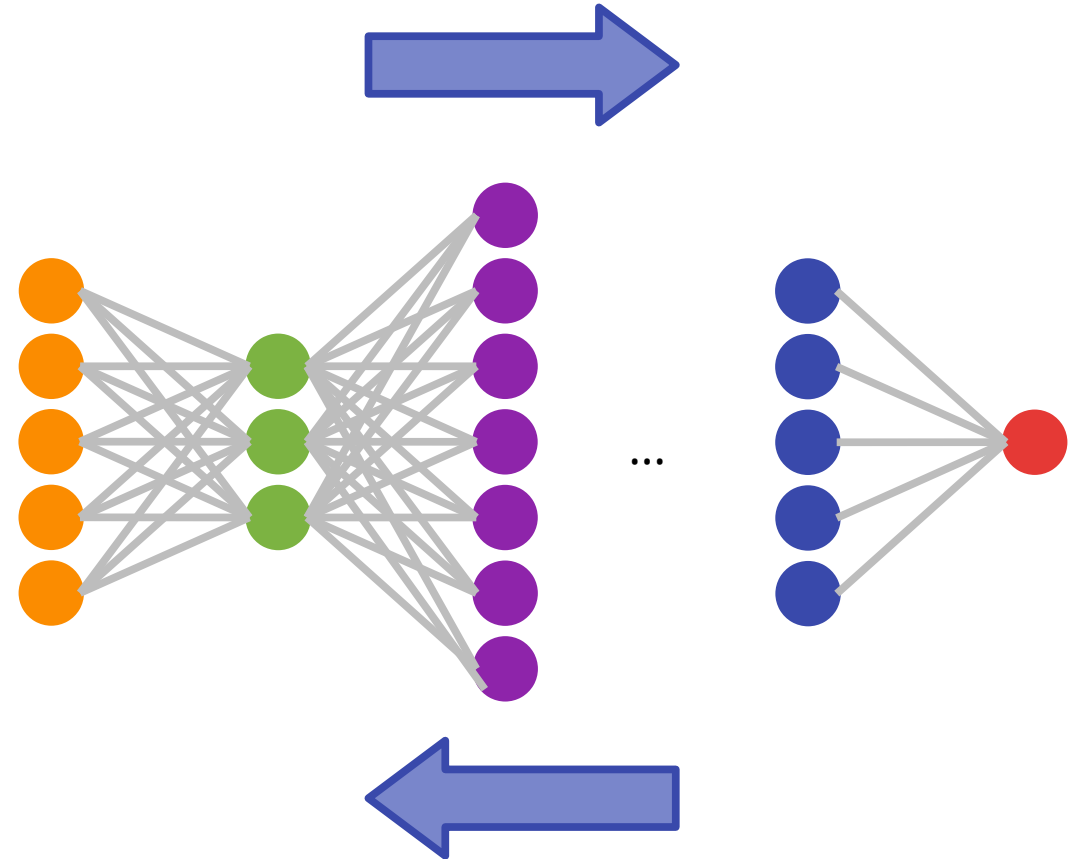
A good activation function should be:

- 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 about the architecture of a neural network and what a single neuron 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

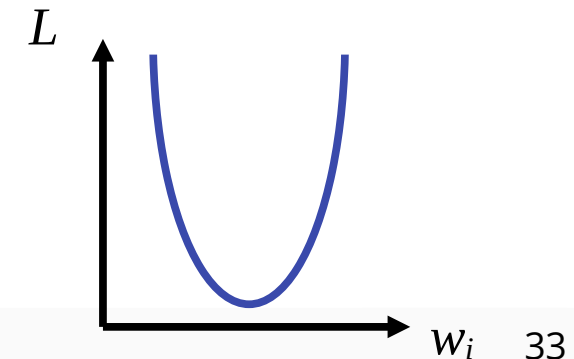
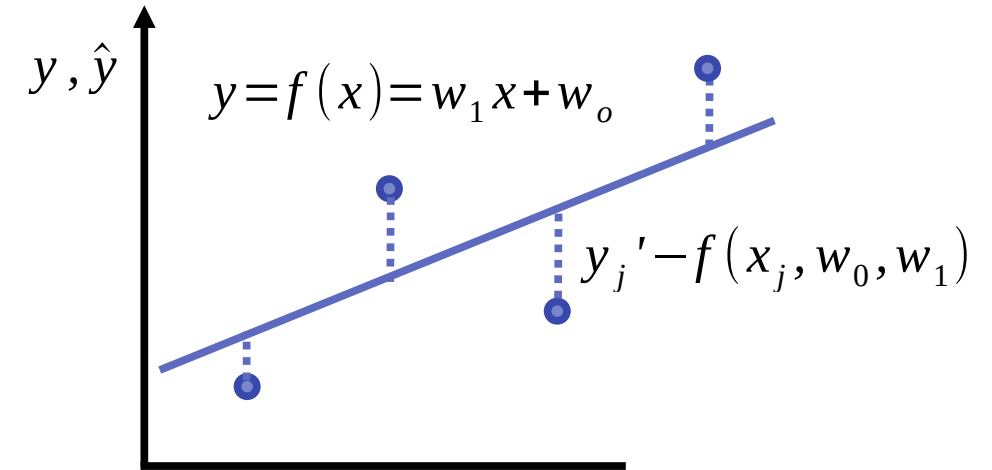
We saw loss functions when we talked about **linear regression**. A quick reminder from lecture 4:

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

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

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

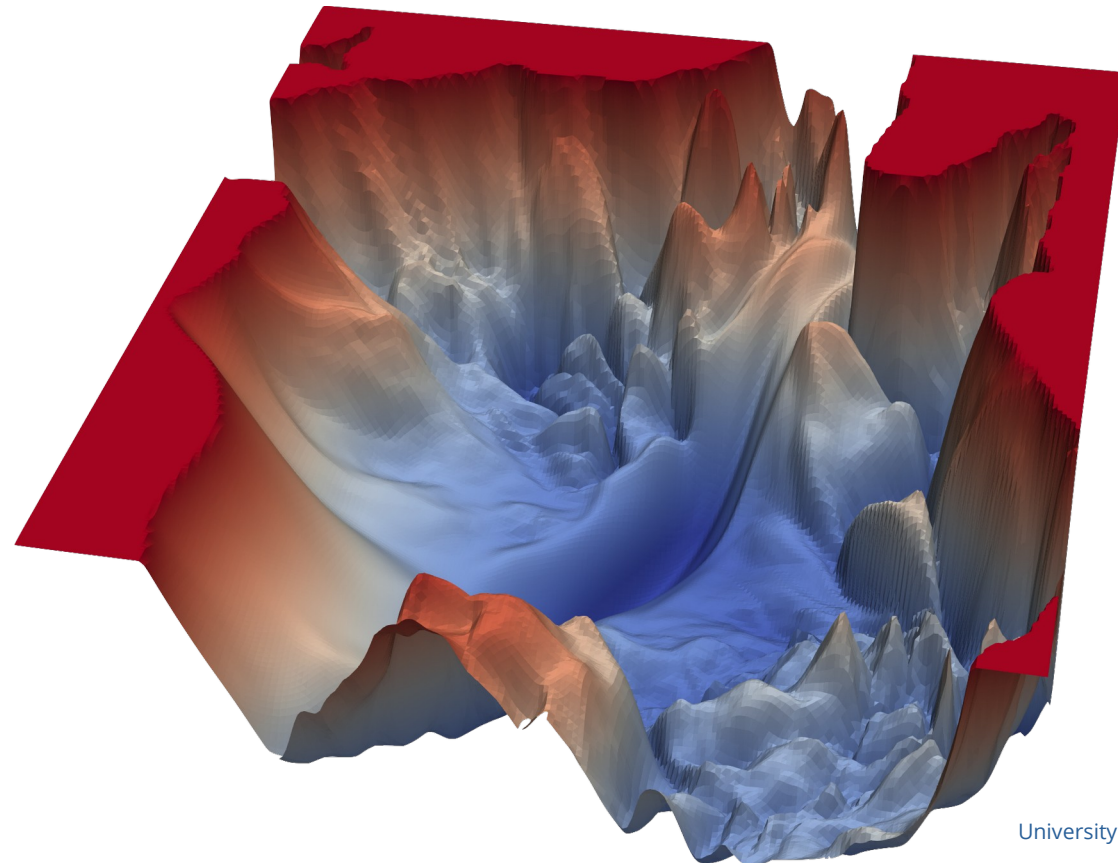
“Least-squares” fitting in linear regression is a **convex optimization problem**: there is only one solution to the problem and it is per definition the best solution.



Loss functions

Optimization is much more complex for neural networks, since the optimization problem is **non-convex**: the weight space is high-dimensional (each weight and bias values adds a dimension to this space) and highly complex.

Let's have a look at some suitable loss functions before we address this optimization problem.



University of Maryland

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)

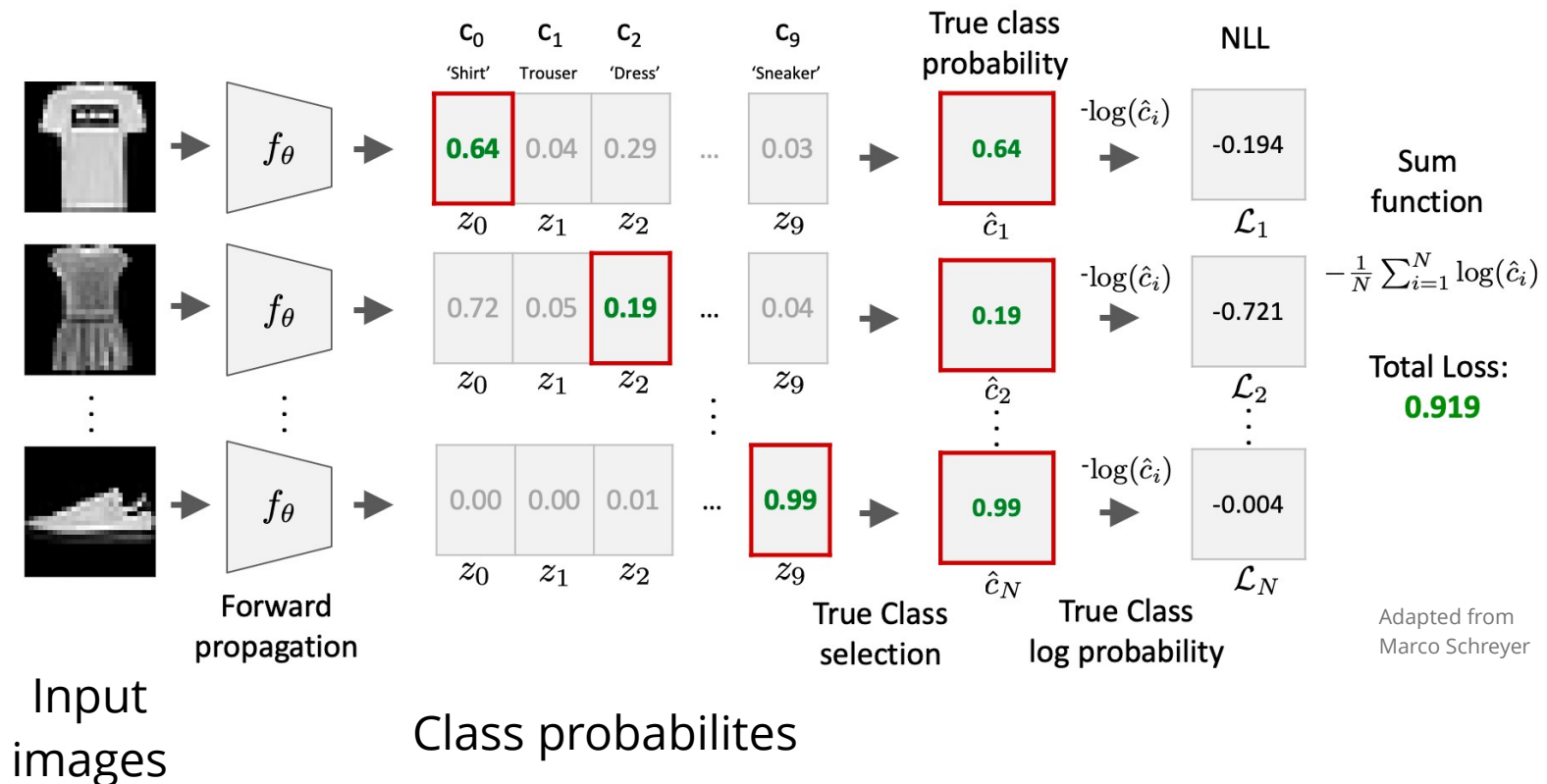
Negative Log-Likelihood Loss

Different loss functions are available. For a simple classification task, the Negative Log-Likelihood (NLL) Loss is very common. How does it work?

$$L(y, \hat{y}) = -\frac{1}{N} \sum_i \log(\hat{y}_i)$$

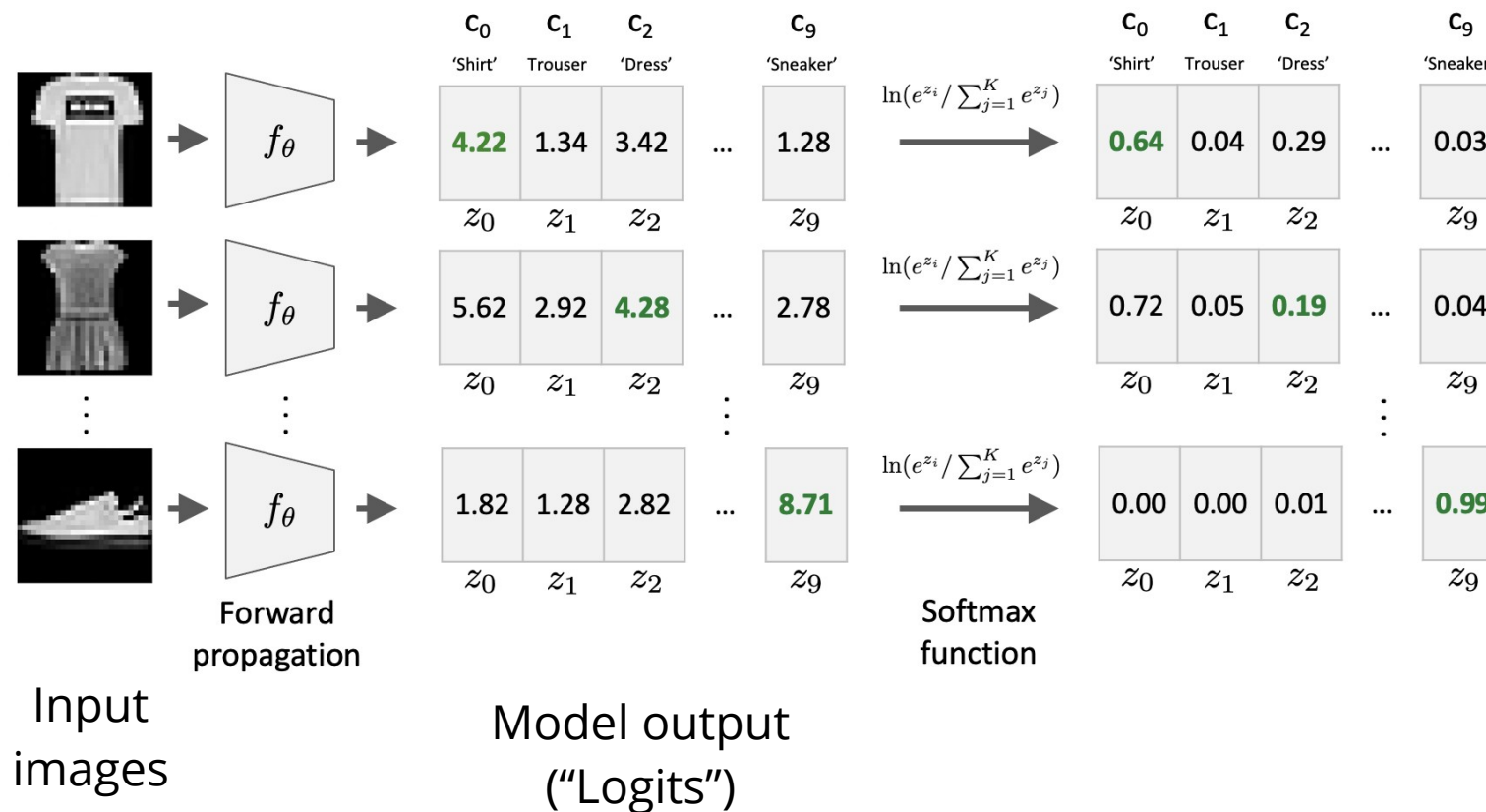
Example: Image classification

The intuition behind NLL is simple: the more confident the model is in making the right predictions, the lower the resulting loss. Wrong predictions or unconfident correct decisions are penalized.



Softmax function

NLL requires class-wise probabilities for each sample. How do we obtain those?

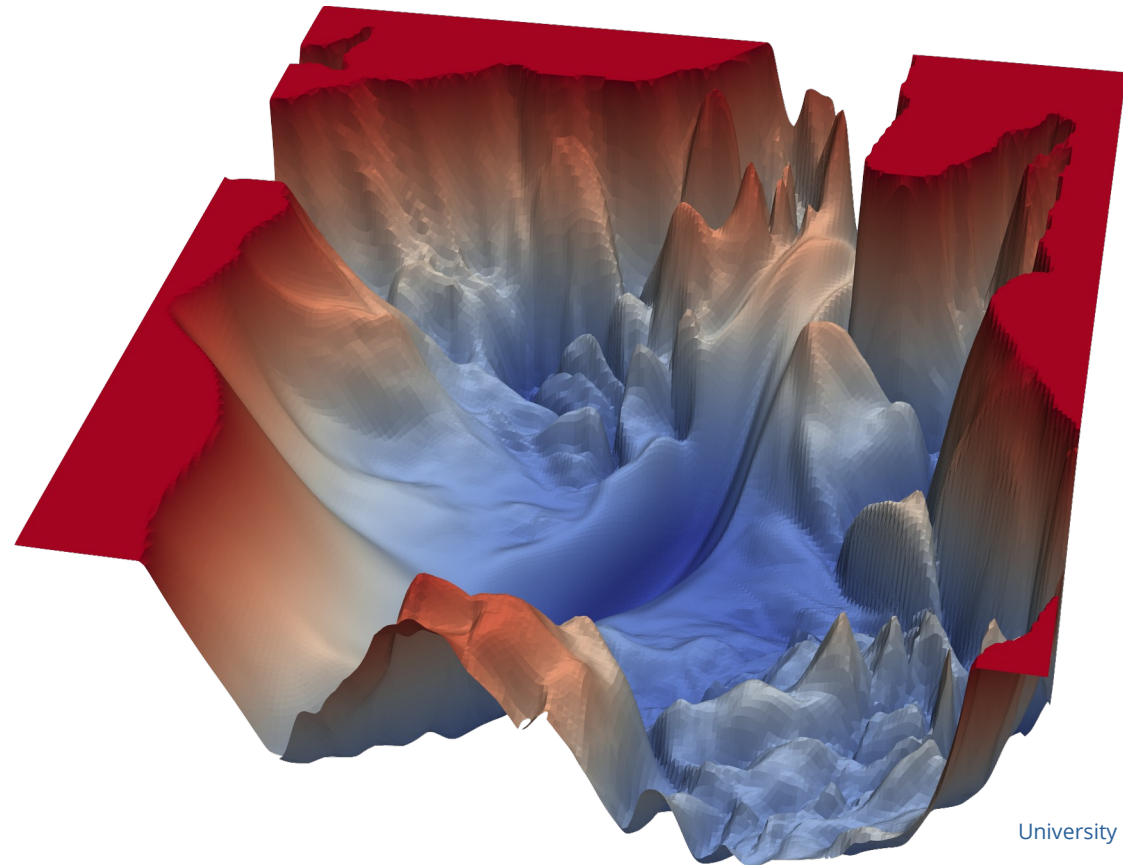


The Softmax function acts as a "smooth argmax" function: it turns a vector of logits (final activation output of a network) into quantities that can be interpreted as probabilities.

$$\sigma(\mathbf{z})_i = \frac{\exp(\mathbf{z}_i)}{\sum_j \exp(\mathbf{z}_j)}$$

Neural network optimization

Finding a favorable set of weights and biases (which might not be the best solution) is only possible in an **iterative** and **stochastic** process.



University of Maryland

Neural network optimization

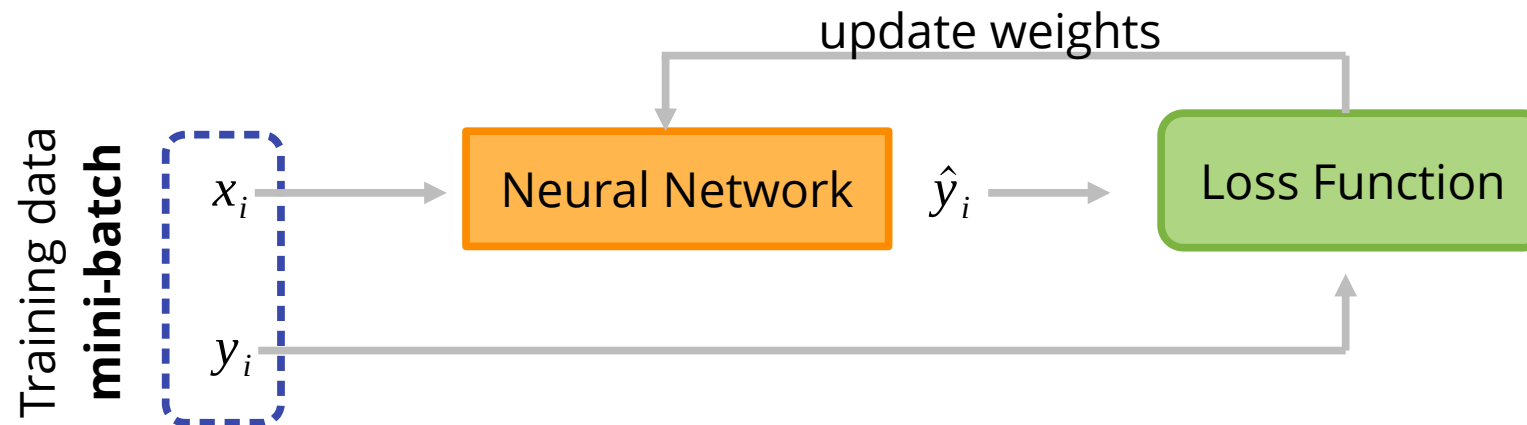
This iterative optimization approach works as follows:

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.

But how do we modify those weights?

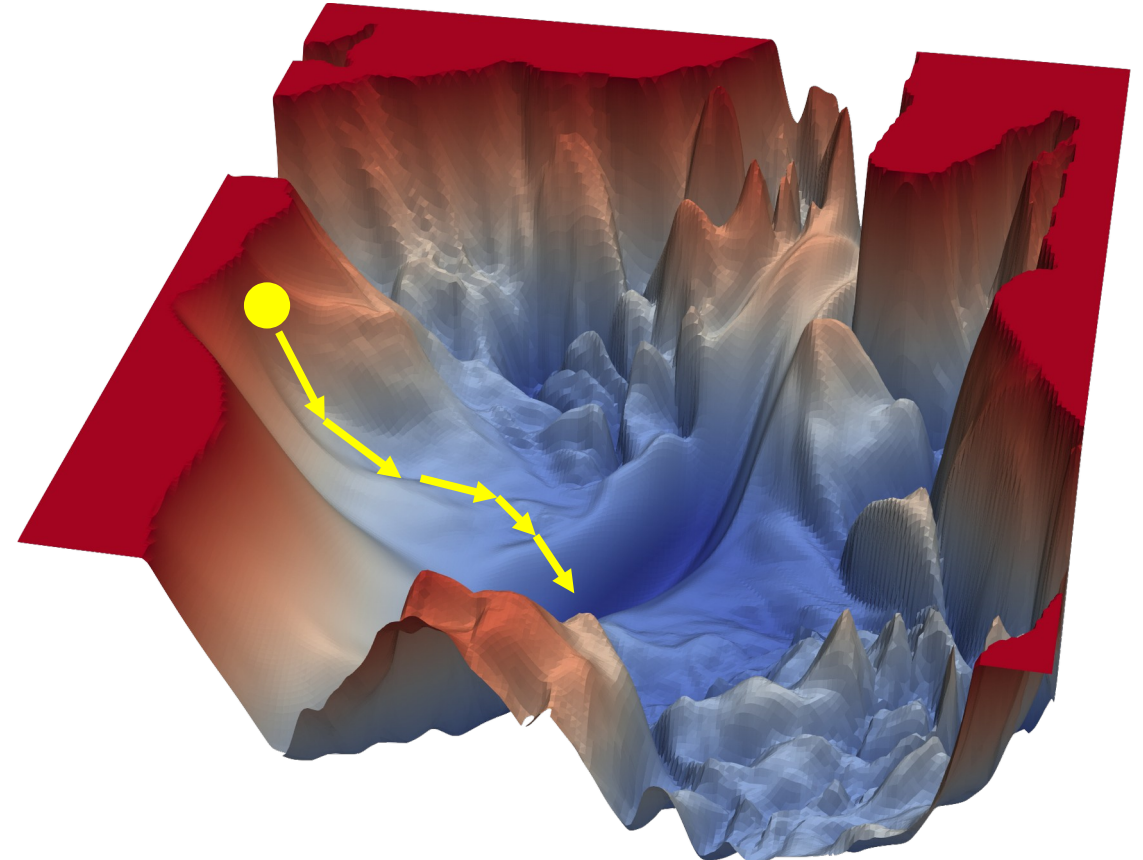


Neural network optimization: Gradient Descent

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

Random changes: possible, but not very goal-oriented

Gradient Descent: we update the weights in such a way that we iteratively approach the

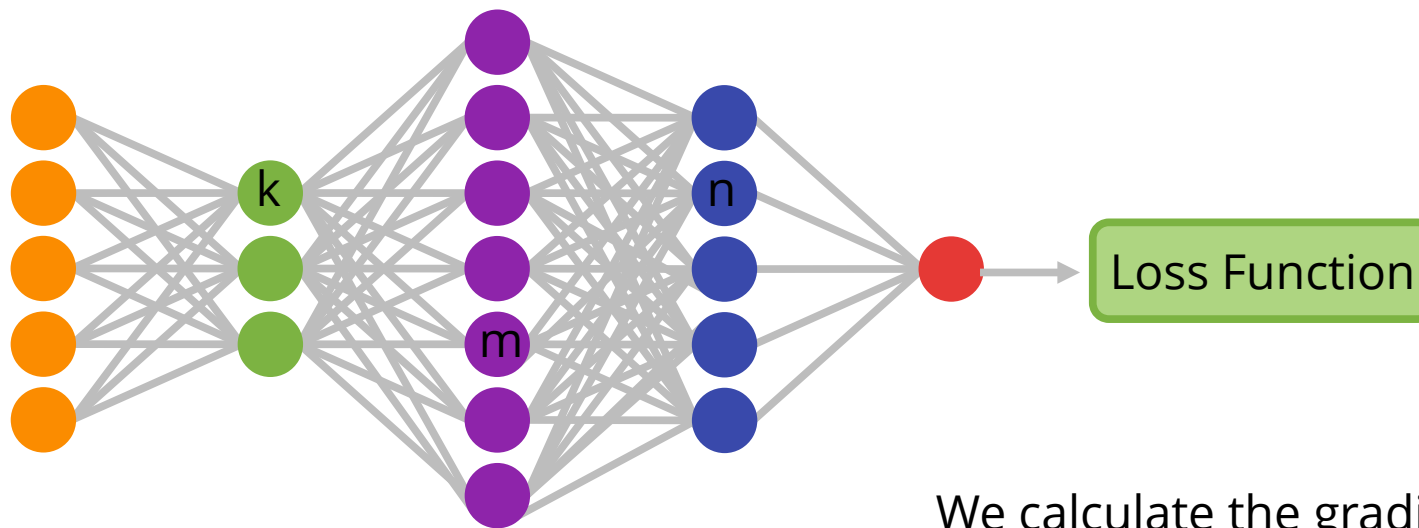


University of Maryland

Neural network optimization: Backpropagation

We have to compute the gradient at the current location in weight space. This is a complex task, but computationally feasible using a method called **backpropagation**. The method returns for every single weight parameter, 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.

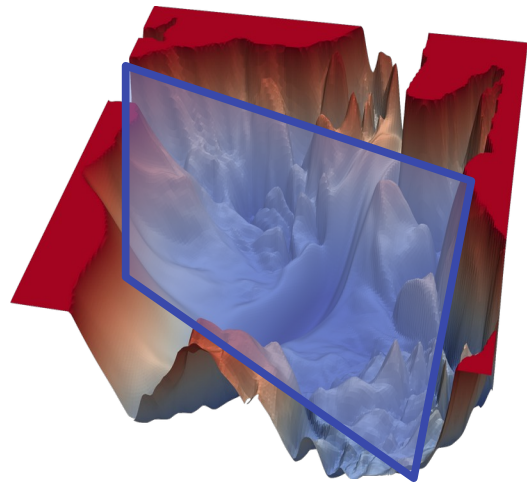
Based on the computed gradients, we can modify each individual weight parameter, w_i :

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

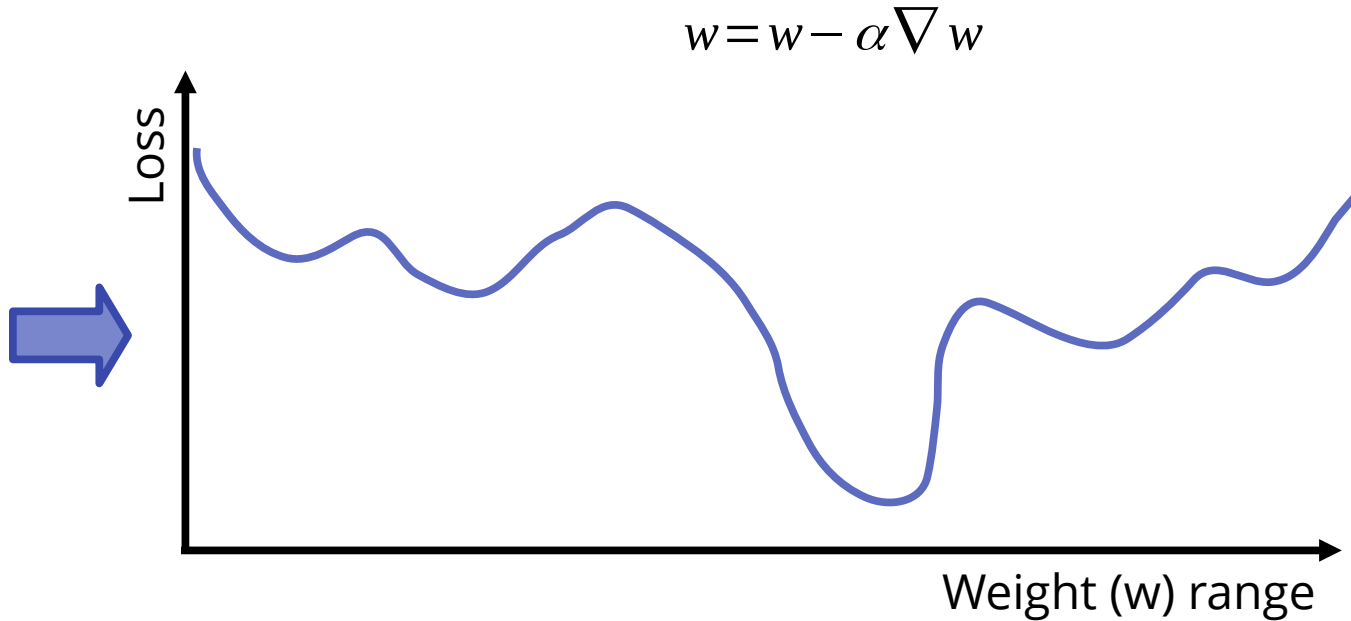
We define a step size for these modifications, which we call the **learning rate**, α .

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



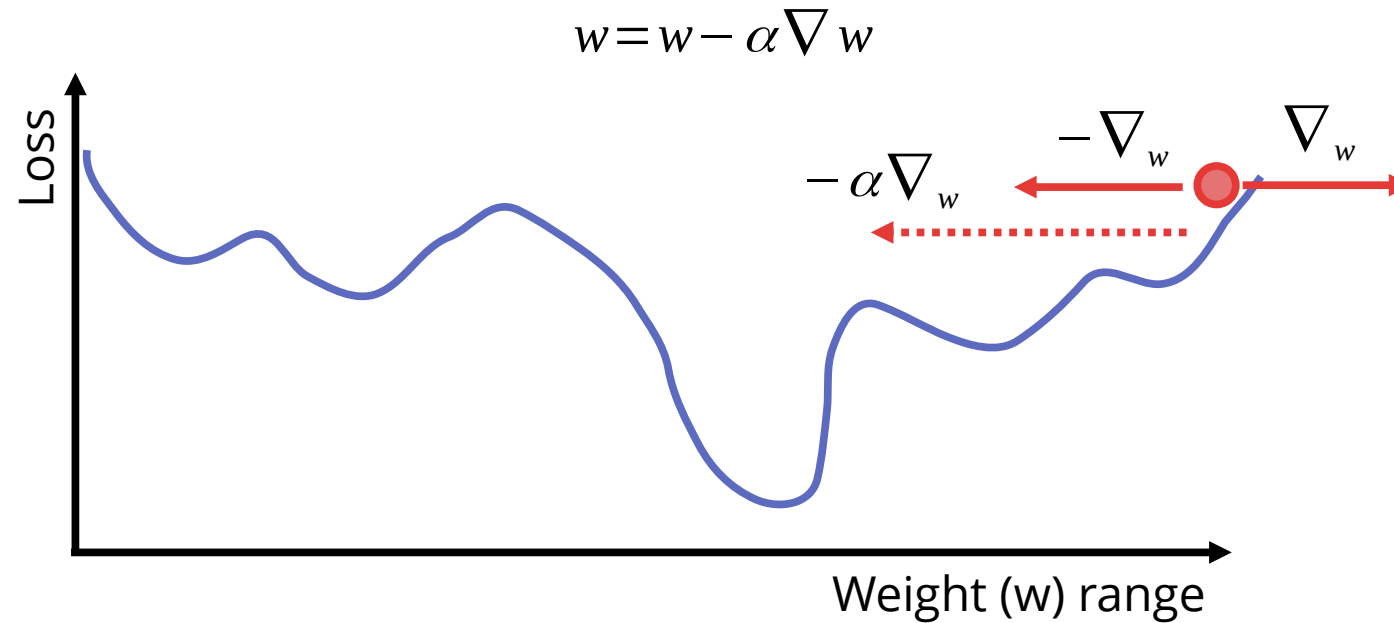
University of Maryland



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

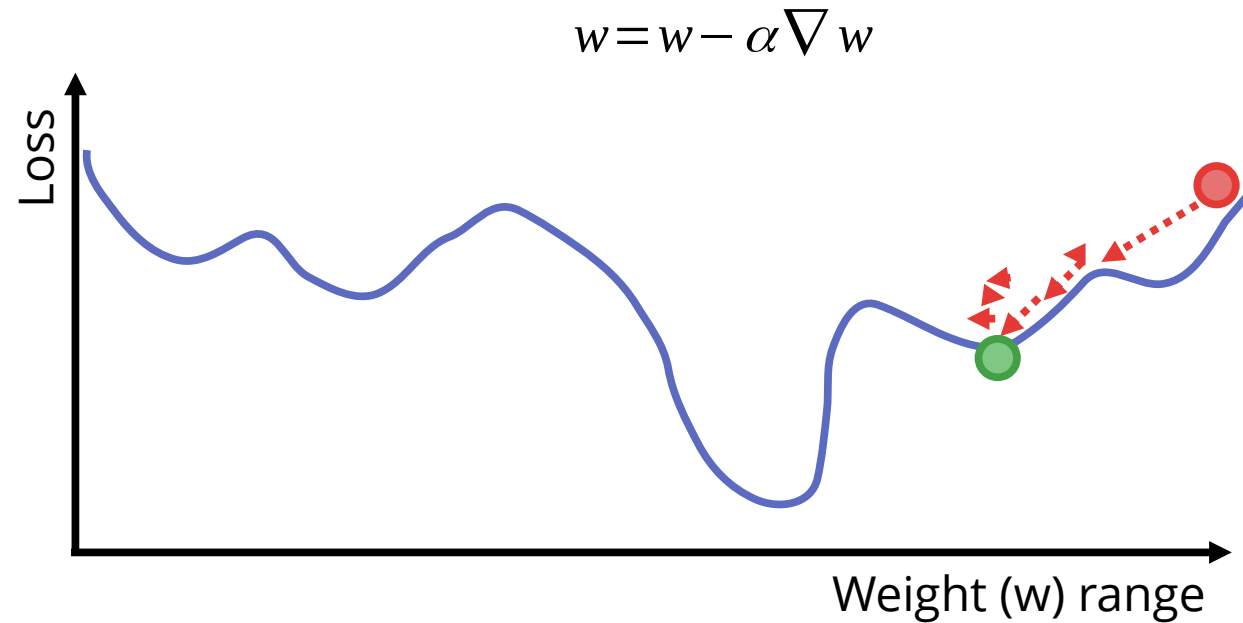


The gradient points in that direction in which the loss function slopes up.

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

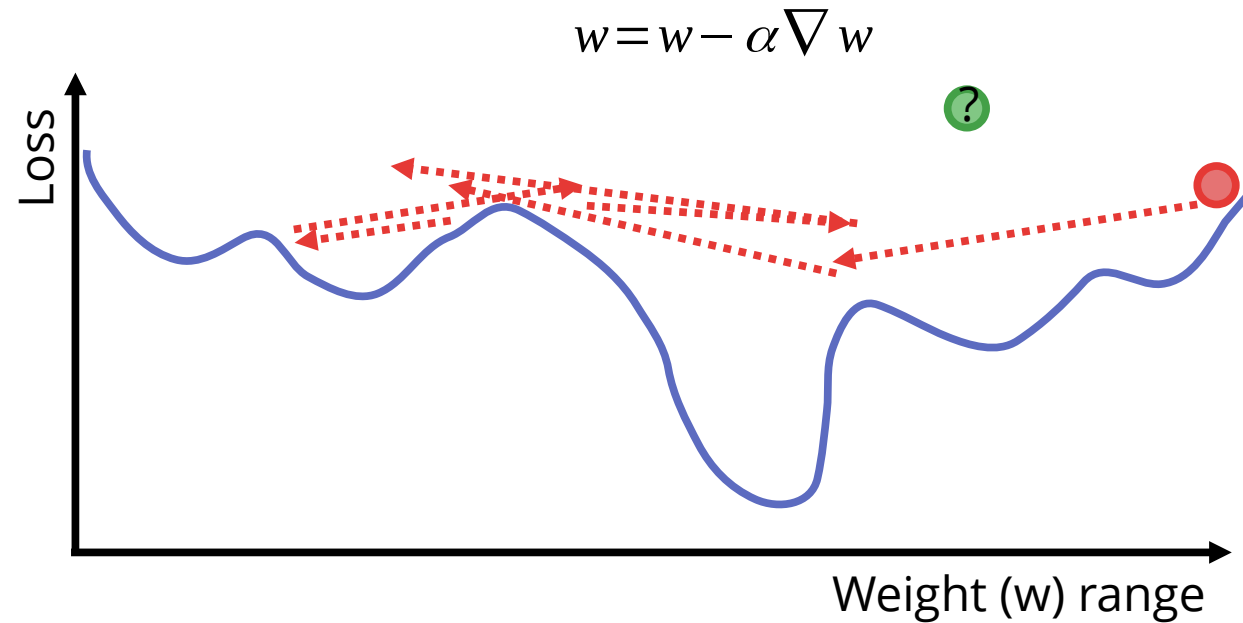
α modulates our step size.

Visualization: gradient descent in one dimension



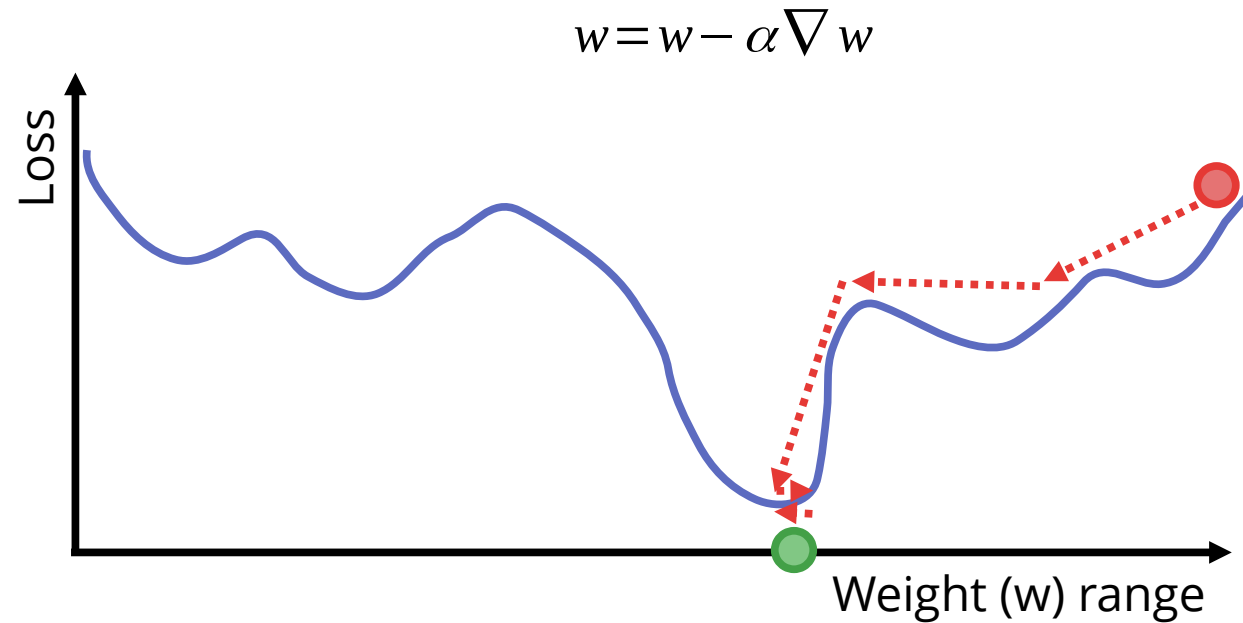
If we use a small learning rate α , it will take a long time to reach the global minimum. It is also possible to get stuck in a local minimum...

Visualization: gradient descent in one dimension



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



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.

Loss functions, backpropagation and gradient descent: summary

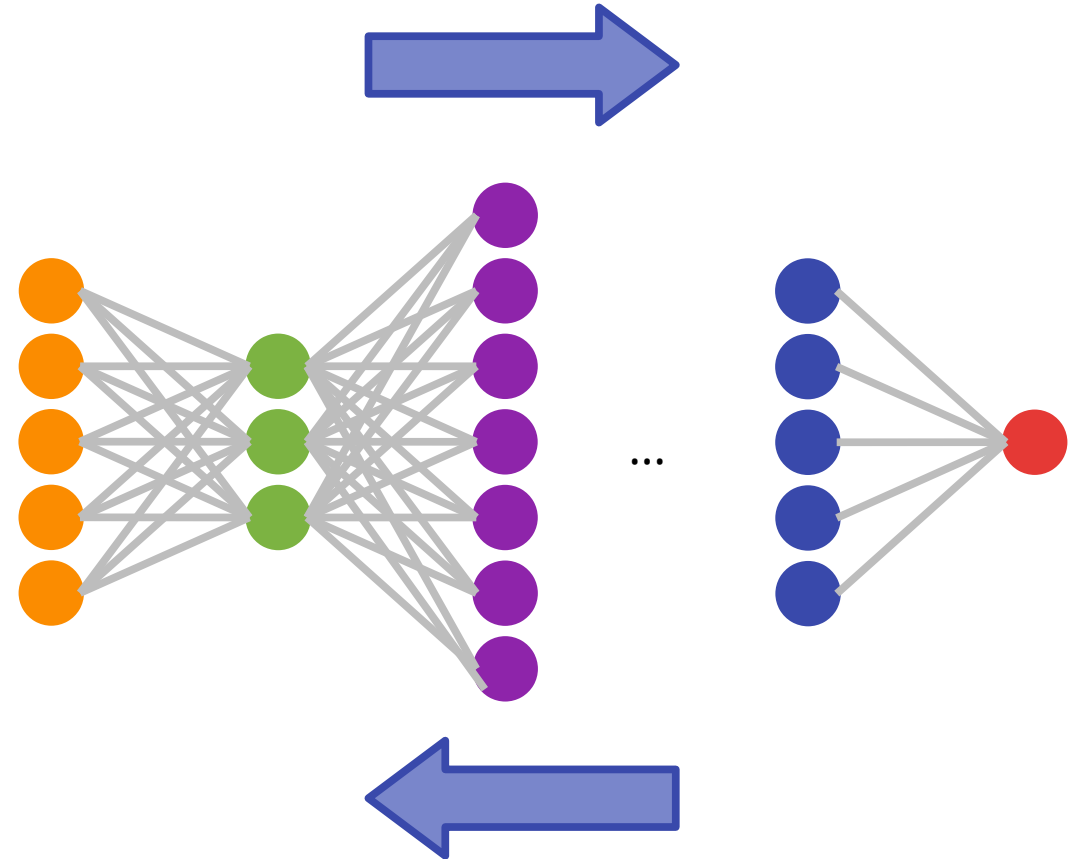
By defining a loss function that is related to the task the network has to learn, we can formulate the training process as an optimization problem.

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



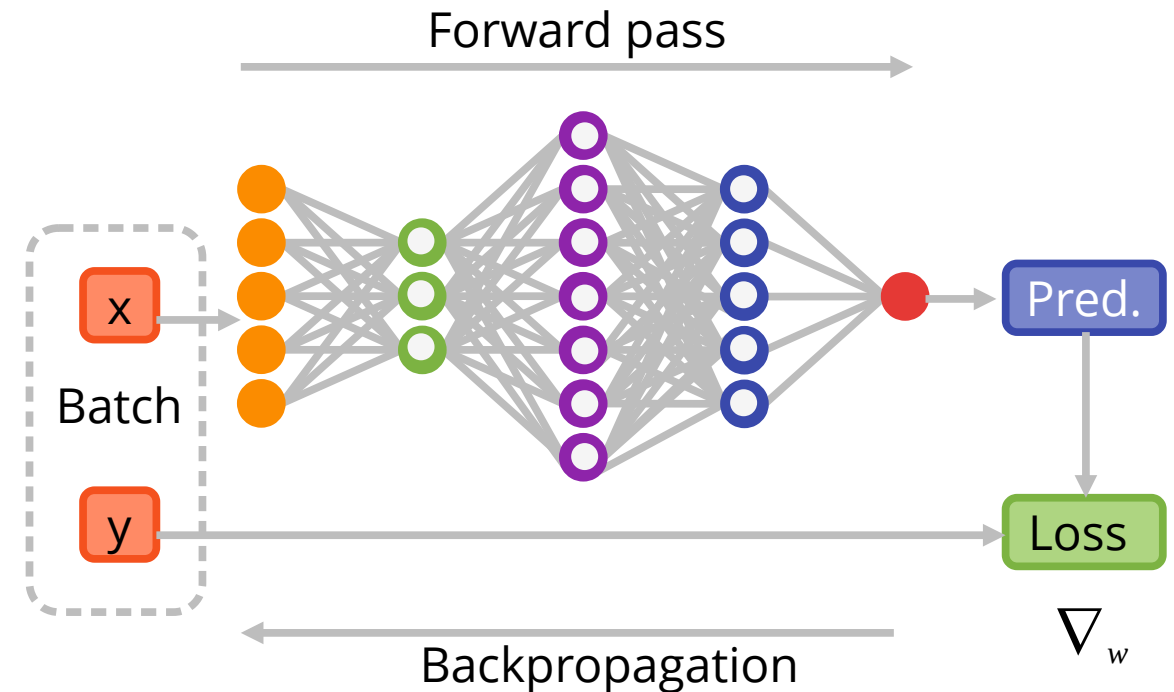
Neural network training pipeline

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

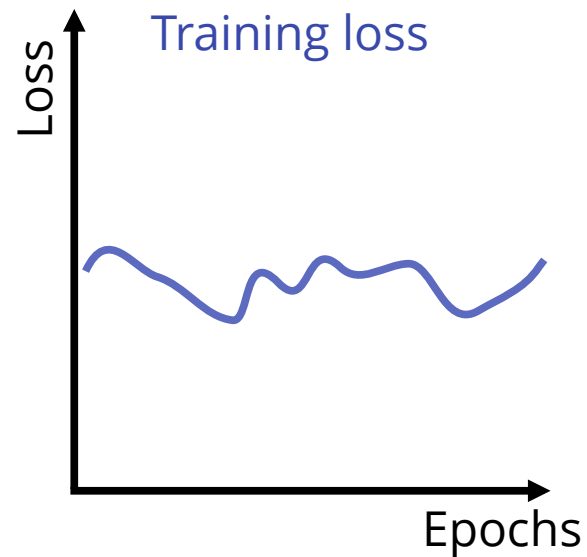
Repeat for a number of epochs, monitor training and validation loss + metrics

Stop before overfitting sets in

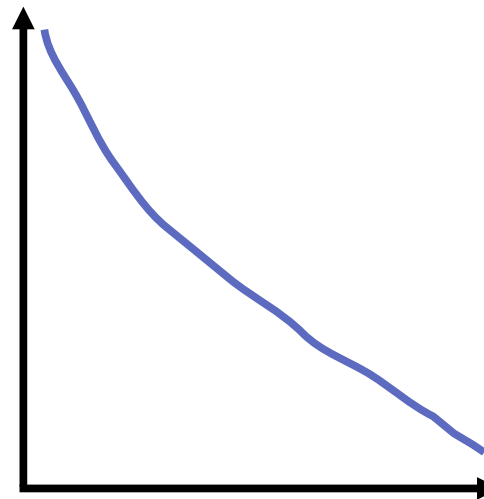


Neural network training process: reading the loss

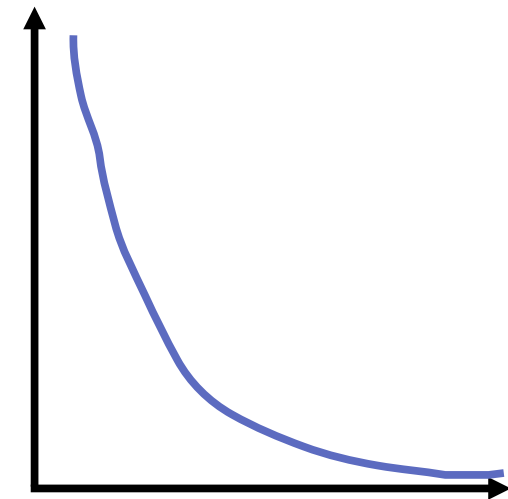
Studying the training loss allows the practitioner to understand whether the model learns anything useful.



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



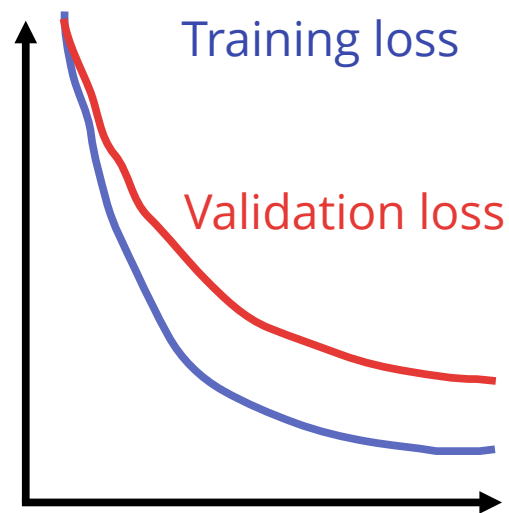
Model learns but could
learn some more
(more epochs)



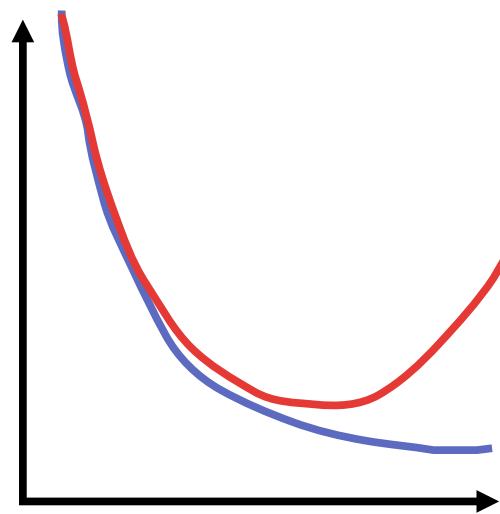
Model learned well
(good time to stop
learning)

Neural network training process: reading the loss

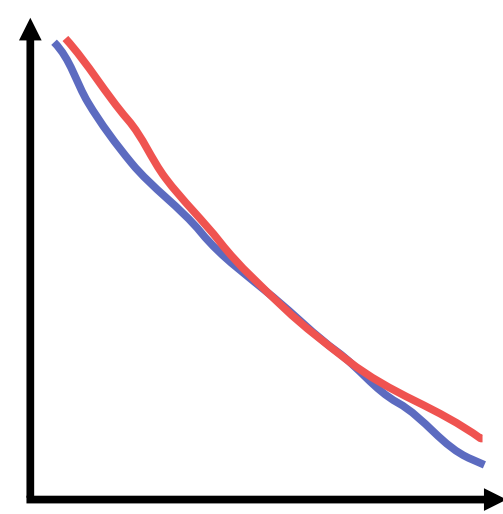
Comparing the training loss to the validation loss (loss over the validation dataset) allows the practitioner to understand whether the model is overfitting, or not.



Well-trained model

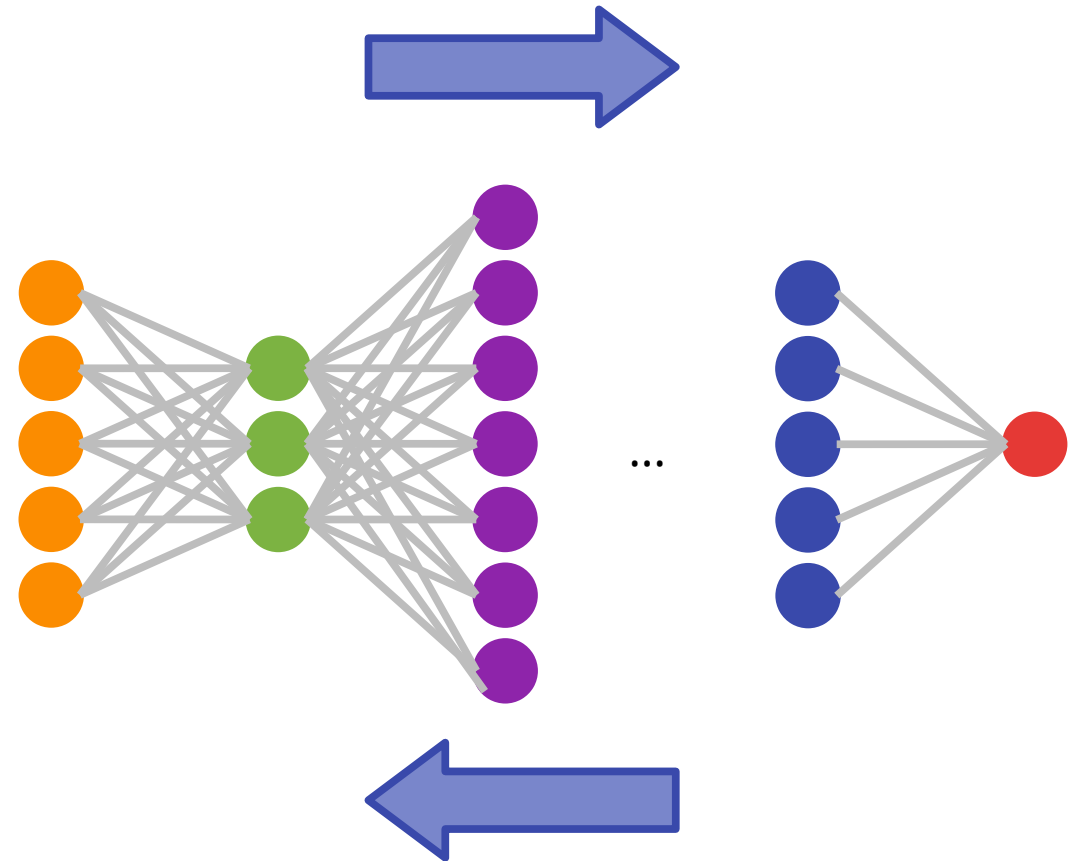


Overfitting



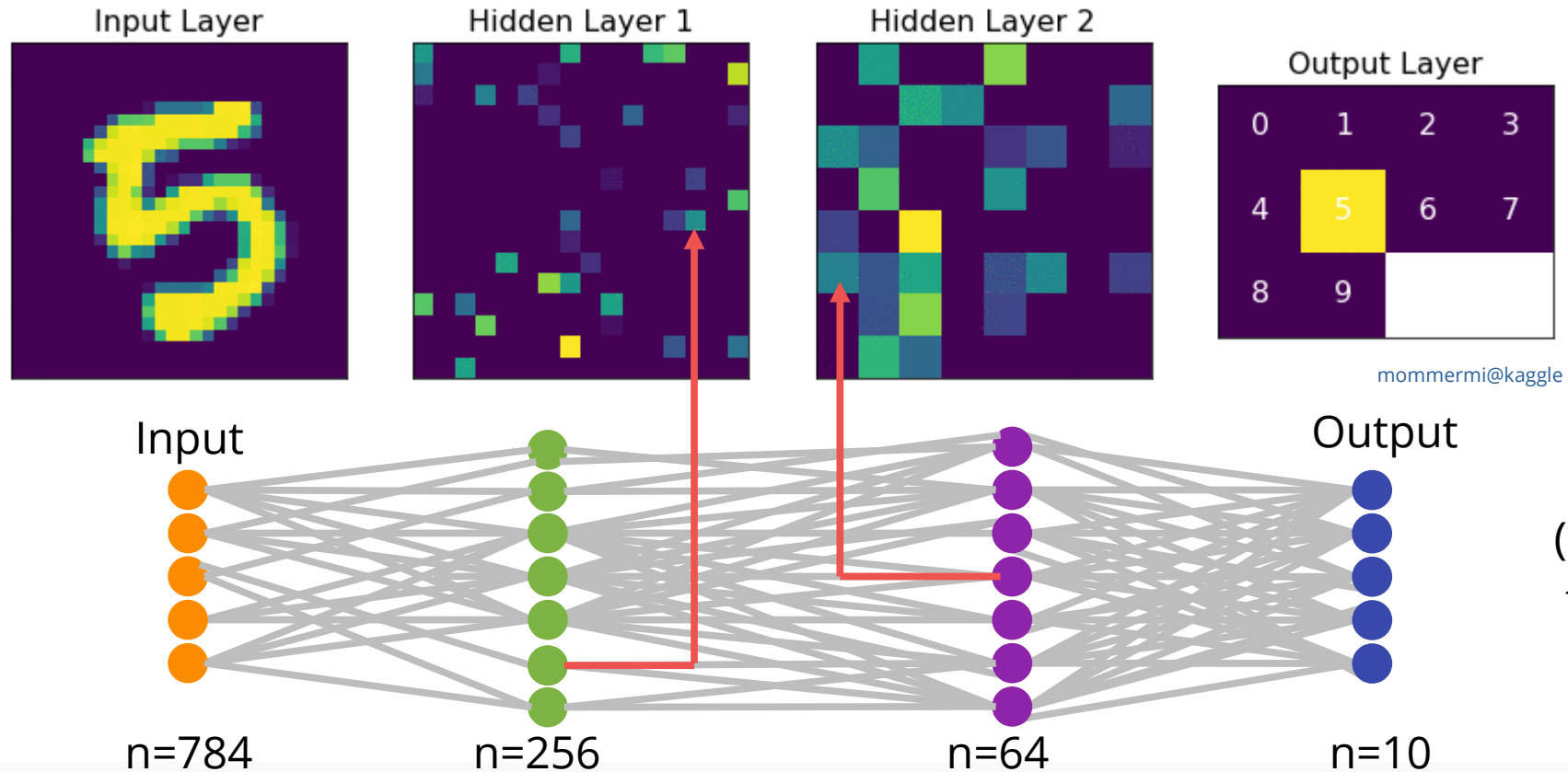
Underfitting

What happens inside a Neural Network?



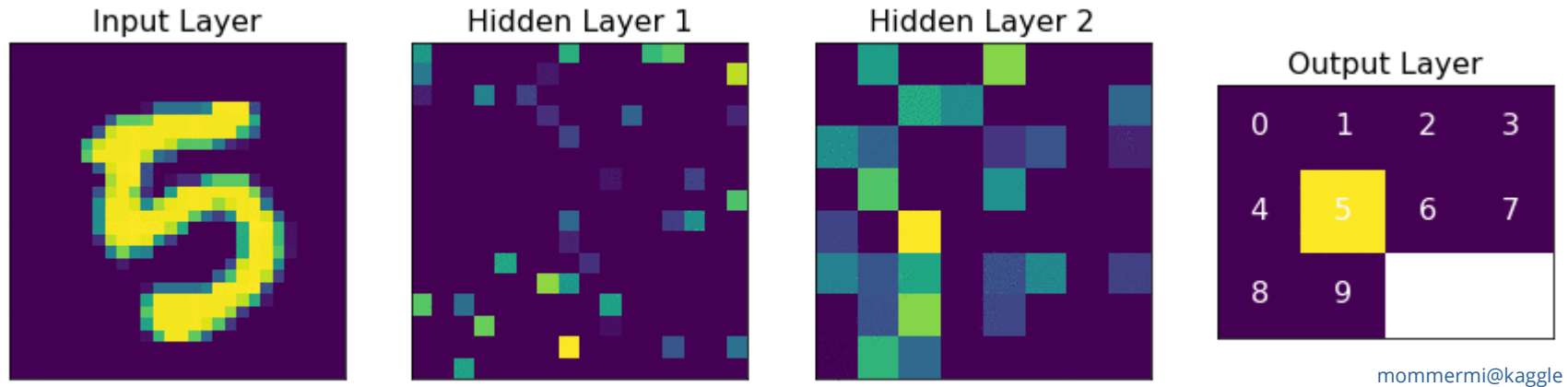
Example: what do neural networks learn?

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.



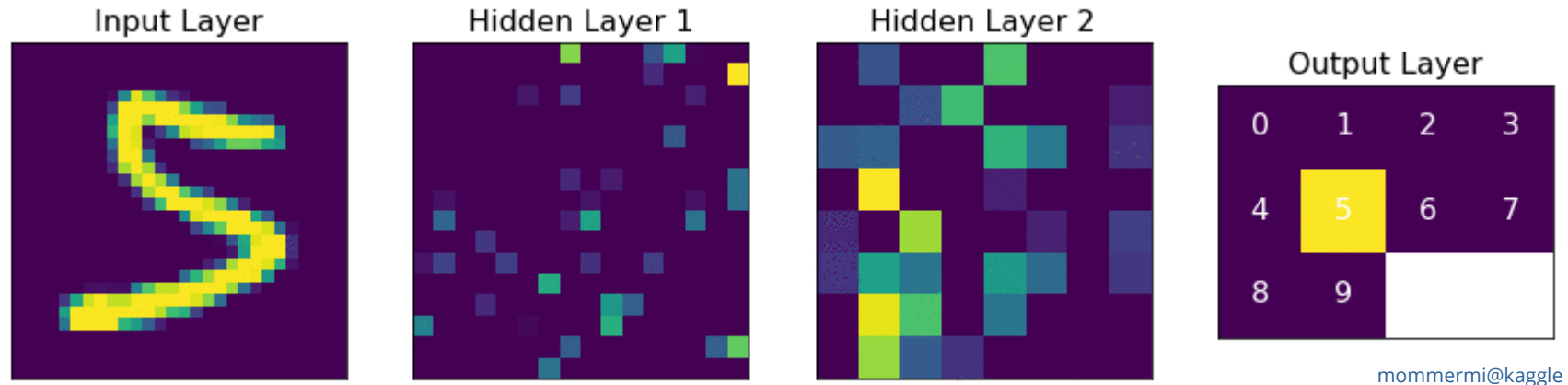
Example: what do neural networks learn?

Neural networks learn patterns from data to perform specific tasks.



Example: what do neural networks learn?

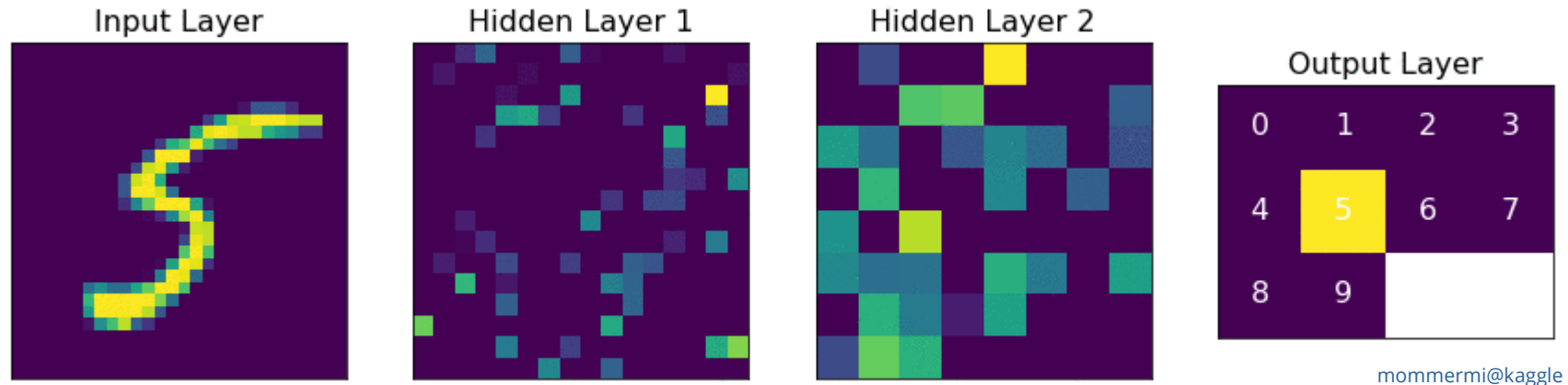
Neural networks learn patterns from data to perform specific tasks.



Seemingly different neurons fire for different input images.

Example: what do neural networks learn?

Neural networks learn patterns from data to perform specific tasks.

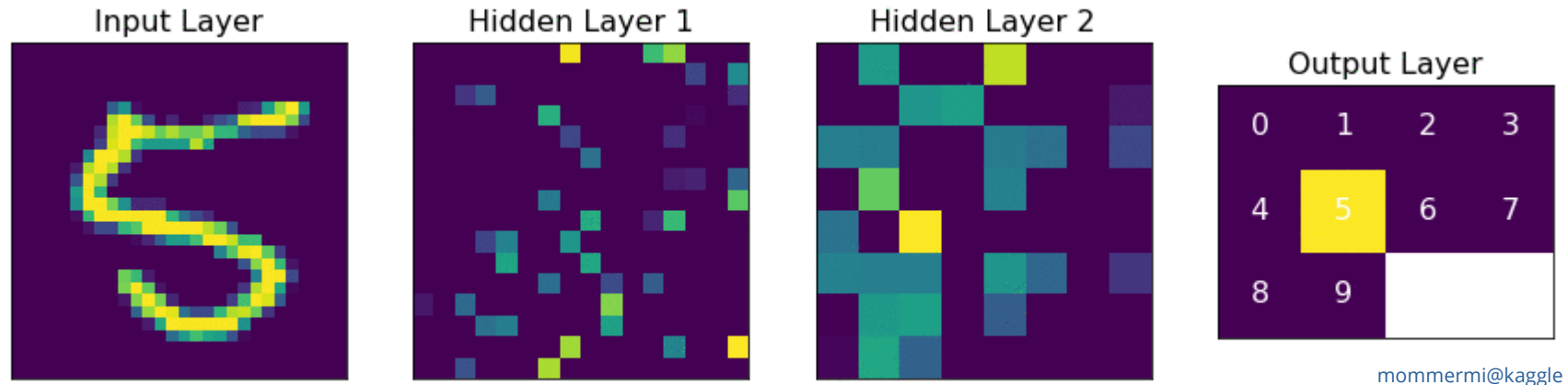


Seemingly different neurons fire for different input images.

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.

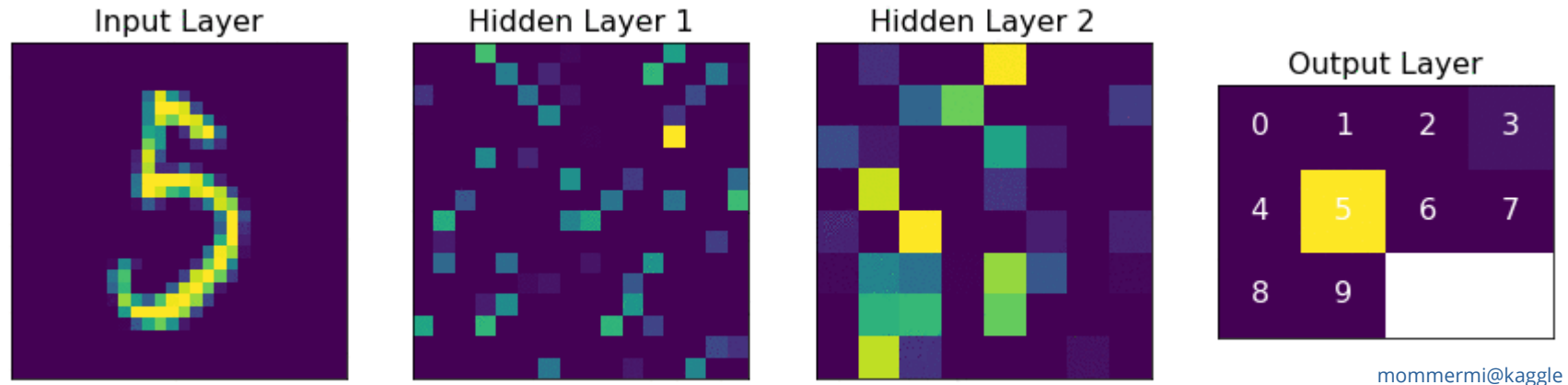


Seemingly different neurons fire for different input images.

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.

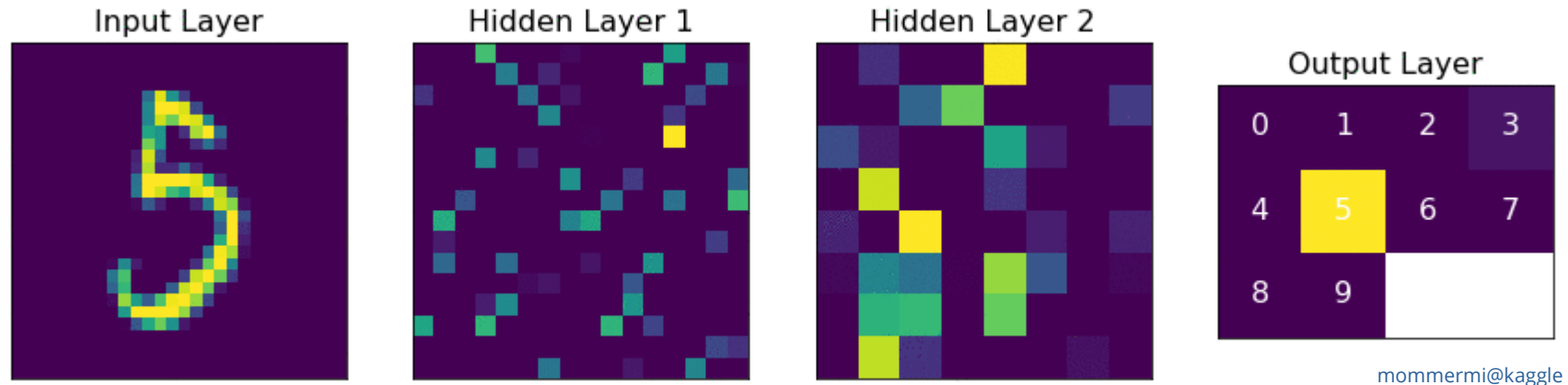


Seemingly different neurons fire for different input images.

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.



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

Early layers perform **feature extraction**, later layers perform the **actual classification task**.

This is called **end-to-end learning**: feature extraction is done by the network itself!

That's all folks!