Basics of neural networks

Simon Leglaive

Artificial Intelligence and Deep Learning course
CentraleSupélec

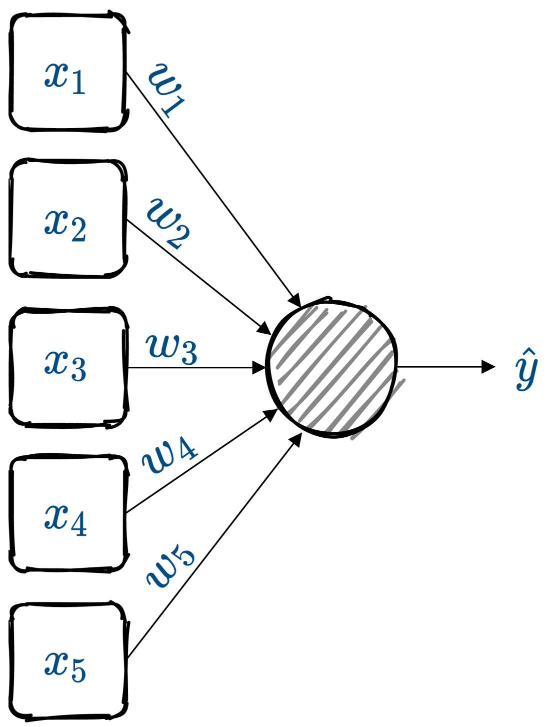
The multi-layer perceptron

Artificial neuron

$$\hat{y} = f(\mathbf{x};\theta) = \sigma\left(\mathbf{w}^T\mathbf{x} + b\right) = \sigma\left(\sum_i w_i x_i + b\right),$$

where

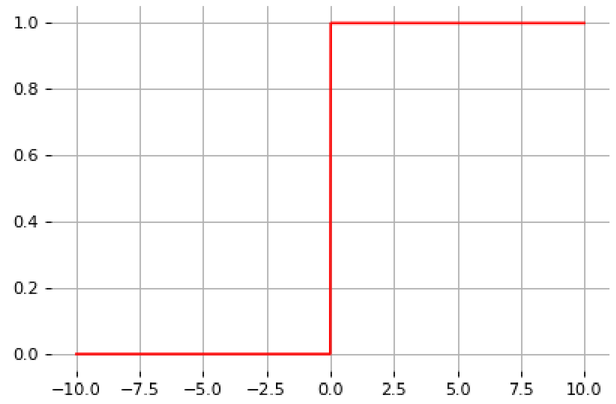
- $\mathbf{x}$ is the input vector;
- $\hat{y}$ the scalar output;
- $\mathbf{w}$ is the weight vector;
- b is the scalar bias;
- σ is a non-linear activation function;
- $\theta = \{\mathbf{w}, b\}$ are the neuron's parameters.



Perceptron

In the Rosenblatt's Perceptron (1957), the activation function is the Heaviside step function:

$$\sigma(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$



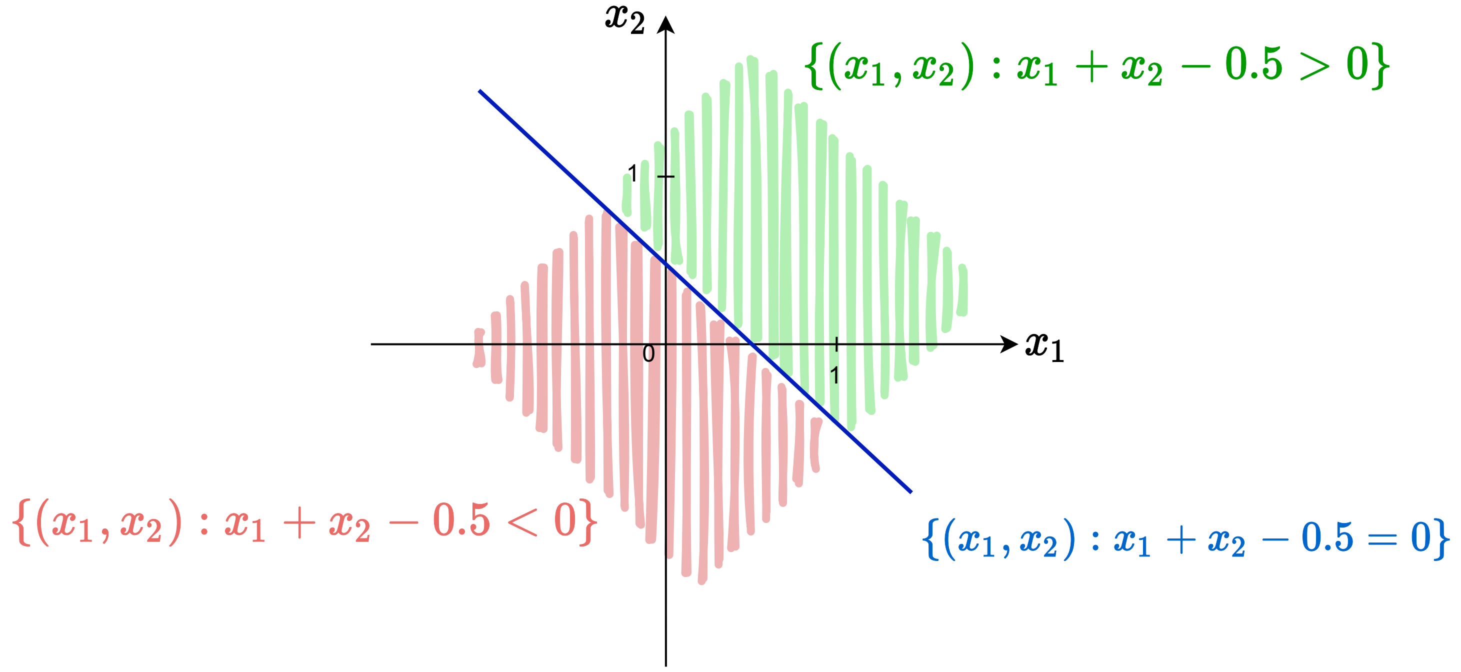
The output of the artificial neuron is 0 or 1, which is suitable for binary classification.

The perceptron classification rule can be rewritten as:

$$\hat{y} = f(\mathbf{x};\theta) = \begin{cases} 1 & \text{if } \sum_i w_i x_i + b \geq 0 \\ 0 & \text{otherwise} \end{cases}.$$

The neuron's parameters define a hyperplane (a line in 2D) which separates the input space into two areas, one for each class.

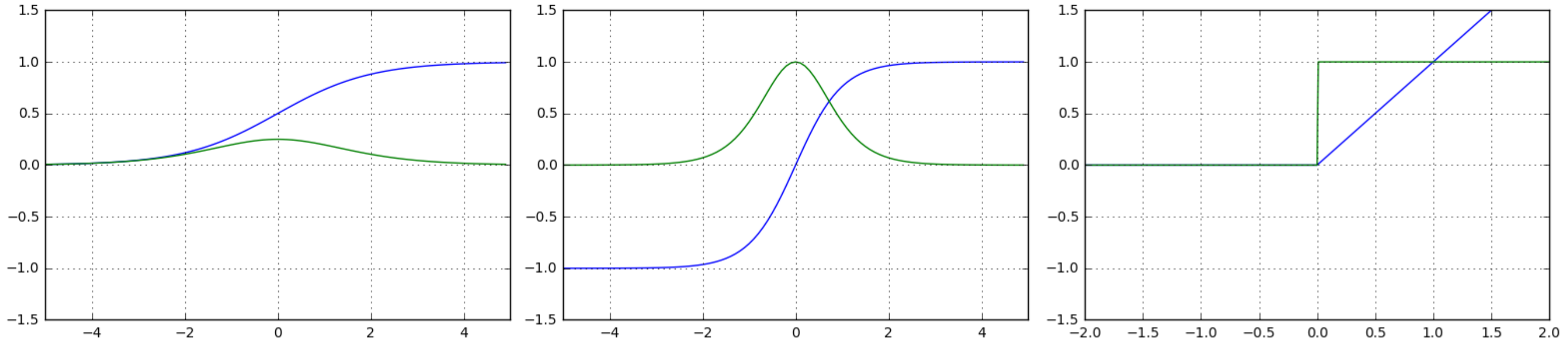
$$f(x_1, x_2; \theta) = \begin{cases} 1 & \text{if } x_1 + x_2 - 0.5 \geq 0 \\ 0 & \text{otherwise} \end{cases}$$


x_2
x_1
1
0
1
$\{(x_1, x_2) : x_1 + x_2 - 0.5 > 0\}$
$\{(x_1, x_2) : x_1 + x_2 - 0.5 < 0\}$
$\{(x_1, x_2) : x_1 + x_2 - 0.5 = 0\}$

The perceptron rule is thus limited to linearly separable classification problems.

Activation functions

In modern neural networks, the Heaviside step function is replaced by a **differentiable** (almost everywhere) non-linear activation function.



$$\text{sigm}(x) = \frac{1}{1+e^{-x}} \qquad \tanh(x) = \frac{e^{2x}-1}{e^{2x}+1} \qquad \text{relu}(x) = \max(0, x)$$

$$\text{sigm}'(x) = \text{sigm}(x)(1-\text{sigm}(x)) \qquad \tanh'(x) = 1 - \tanh(x)^2 \qquad \text{relu}'(x) = 1_{x>0}$$

Image credit: Olivier Grisel and Charles Ollion

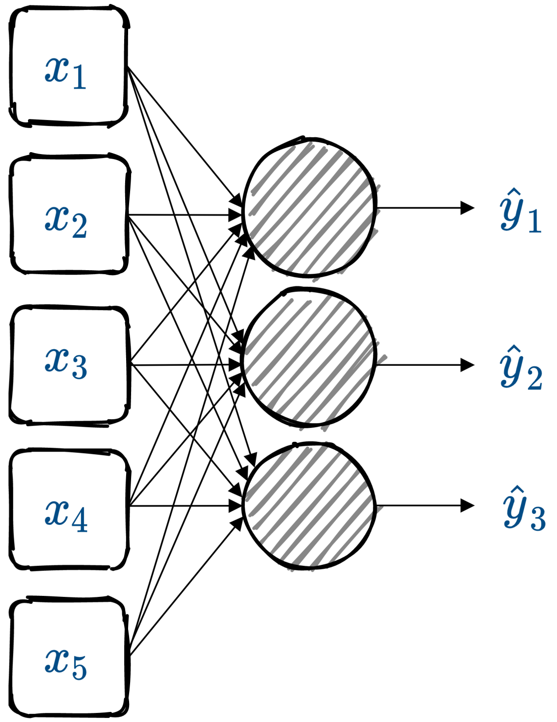
Layer

Neurons can be composed in parallel to form a **layer** with multiple outputs:

$$\hat{\mathbf{y}} = f(\mathbf{x}; \theta) = \sigma\left(\mathbf{W}^T \mathbf{x} + \mathbf{b}\right),$$

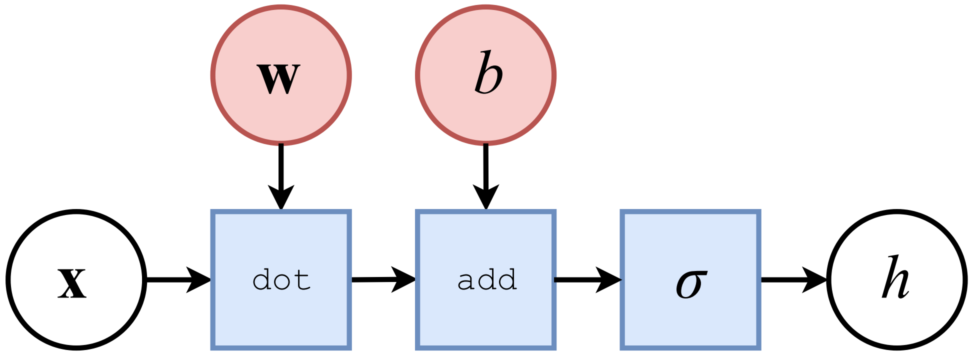
where we have now

- an element-wise activation function,
- an output vector $\hat{\mathbf{y}}$,
- a weight matrix $\mathbf{W}$,
- a bias vector $\mathbf{b}$,
- such that $\theta = \{\mathbf{W}, \mathbf{b}\}$.



The computation can be represented as a computational graph where

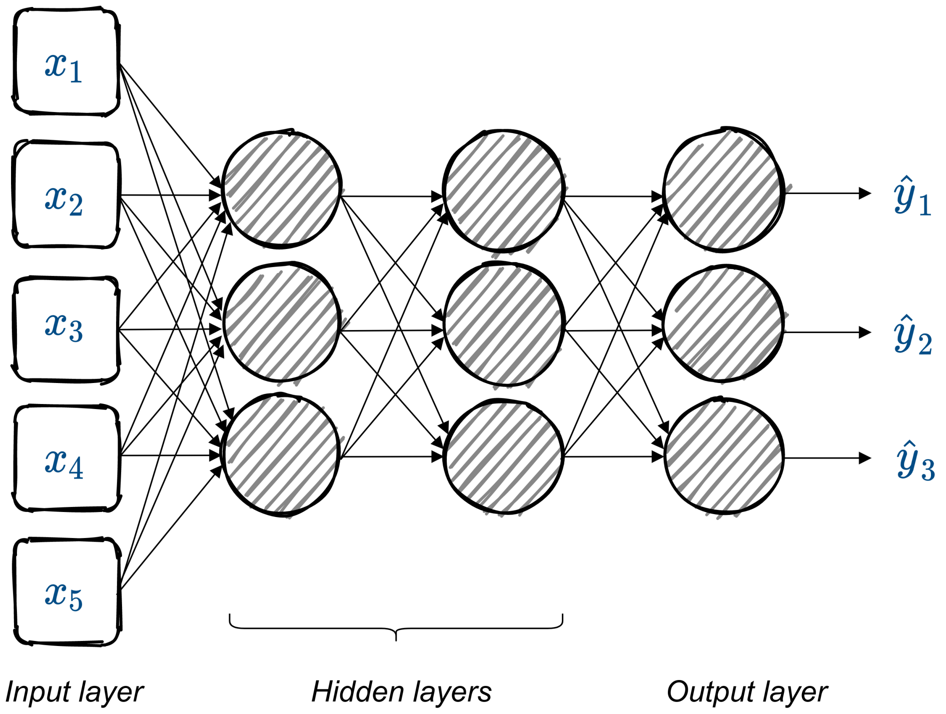
- white nodes correspond to inputs and outputs;
- red nodes correspond to model parameters;
- blue nodes correspond to intermediate operations.



This unit is the lego brick of all neural networks!

Multi-layer Perceptron

Similarly, layers can be composed in series, to form a multi-layer Perceptron, or feed-forward fully-connected neural network.



The model parameters are the weight matrices and bias vectors of all layers.

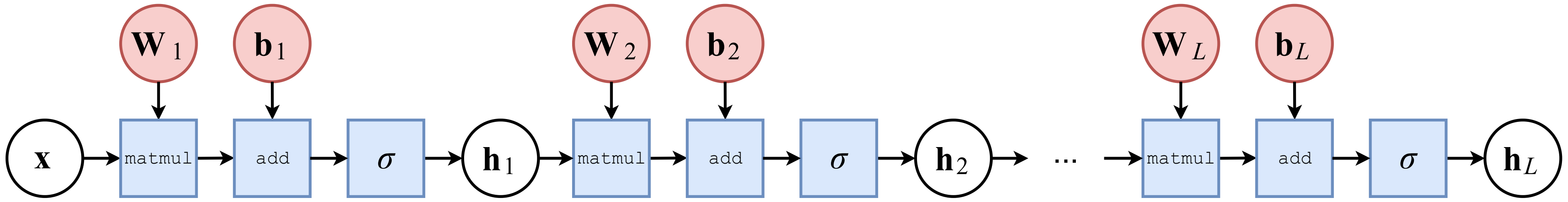


Image credits: Gilles Louppe, INFO8010 - Deep Learning, ULiège.

- The dimension of the input layer, along with the dimension and activation function of the output layer depend on the problem we want to solve.

If we want to classify cat/dog images of 64 x 64 pixels we will have an input layer of dimension 4096 and a single neuron on the output layer, with a sigmoid activation function whose output lies between 0 and 1 and represents the probability of one of the two classes.

- The number of hidden layers, the number of neurons and the activation function for each hidden layer are arbitrarily fixed, there is no proper methodology to set these hyper-parameters.

Supervised learning

In supervised learning, we have access to a dataset $\mathcal{D} = \{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^N$ where the $\mathbf{x}_i$'s are the inputs, and $\mathbf{y}_i$'s the ground-truth labels.

In supervised learning, we have access to a dataset $\mathcal{D} = \{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^N$ where the $\mathbf{x}_i$'s are the inputs, and $\mathbf{y}_i$'s the ground-truth labels.

We want to find a model $f(\cdot; \theta)$ which depends on some parameters θ such that the prediction $\hat{\mathbf{y}} = f(\mathbf{x}; \theta)$ is as close as possible to the true label $\mathbf{y}$, for all the examples $(\mathbf{x}, \mathbf{y})$ in the dataset $\mathcal{D}$.

In supervised learning, we have access to a dataset $\mathcal{D} = \{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^N$ where the $\mathbf{x}_i$'s are the inputs, and $\mathbf{y}_i$'s the ground-truth labels.

We want to find a model $f(\cdot;\theta)$ which depends on some parameters θ such that the prediction $\hat{\mathbf{y}} = f(\mathbf{x};\theta)$ is as close as possible to the true label $\mathbf{y}$, for all the examples $(\mathbf{x}, \mathbf{y})$ in the dataset $\mathcal{D}$.

To do so, we define a loss function:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{D}} \ell\Big(\mathbf{y}_i, f(\mathbf{x}_i;\theta)\Big),$$

where $\ell(\cdot, \cdot)$ is task-dependent.

In supervised learning, we have access to a dataset $\mathcal{D} = \{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^N$ where the $\mathbf{x}_i$'s are the inputs, and $\mathbf{y}_i$'s the ground-truth labels.

We want to find a model $f(\cdot;\theta)$ which depends on some parameters θ such that the prediction $\hat{\mathbf{y}} = f(\mathbf{x};\theta)$ is as close as possible to the true label $\mathbf{y}$, for all the examples $(\mathbf{x}, \mathbf{y})$ in the dataset $\mathcal{D}$.

To do so, we define a loss function:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{D}} \ell\Big(\mathbf{y}_i, f(\mathbf{x}_i;\theta)\Big),$$

where $\ell(\cdot, \cdot)$ is task-dependent.

The model parameters θ are estimated by minimizing the loss function, using (variants of) gradient descent.

Loss function

For both classification and regression, we can interpret $f(\mathbf{x};\theta)$ as defining a model of the posterior distribution $p(y|\mathbf{x};\theta)$.

Therefore, we can define the loss $\ell(\cdot,\cdot)$ as the negative log-likelihood:

$$\begin{aligned}\ell(y, f(\mathbf{x};\theta)) &= -\ln p(\mathbf{x}, y;\theta) \\ &= -\ln p(y|\mathbf{x};\theta) - \ln p(\mathbf{x}) \\ &= -\ln p(y|\mathbf{x};\theta) + cst(\theta)\end{aligned}$$

Binary classification

- Training data: $(\mathbf{x}, y) \in \mathcal{X} \times \mathcal{Y}$ with $\mathcal{X} = \mathbb{R}^p$ and $\mathcal{Y} = \{0, 1\}$.
- Model: We assume $p(y = 1|\mathbf{x}; \theta) = f(\mathbf{x}; \theta)$ and $p(y = 0|\mathbf{x}; \theta) = 1 - f(\mathbf{x}; \theta)$.

  It can be compactly rewritten as follows for all $y \in \mathcal{Y}$:

$$p(y|\mathbf{x}; \theta) = \Big(f(\mathbf{x}; \theta)\Big)^y \Big(1 - f(\mathbf{x}; \theta)\Big)^{(1-y)}.$$

- Output layer: Width equal to 1 and sigmoid activation function such that $f(\mathbf{x}; \theta) \in [0, 1]$.
- The NLL gives the binary cross-entropy loss:

$$\begin{aligned}
\ell(y, f(\mathbf{x}; \theta)) &= -\ln p(\mathbf{x}, y; \theta) \\
&= -\ln p(y|\mathbf{x}; \theta) - \ln p(\mathbf{x}) \\
&= -y \ln \Big(f(\mathbf{x}; \theta)\Big) - (1 - y) \ln \Big(1 - f(\mathbf{x}; \theta)\Big) + cst(\theta)
\end{aligned}$$

C-class classification

- Training data: $(\mathbf{x}, y) \in \mathcal{X} \times \mathcal{Y}$ with $\mathcal{X} = \mathbb{R}^p$ and $\mathcal{Y} = \{1, ..., C\}$.
- Model: We assume $p(y = c|\mathbf{x}; \theta) = f_c(\mathbf{x}; \theta)$ for all $c \in \{1, ..., C\}$.

  It can be compactly rewritten as follows for all $y \in \mathcal{Y}$:

$$p(y|\mathbf{x}; \theta) = \prod_{c=1}^{C} p(y = c|\mathbf{x}; \theta)^{\mathbf{1}_{y=c}} = \prod_{c=1}^{C} f_c(\mathbf{x}; \theta)^{\mathbf{1}_{y=c}}.$$

- Output layer: Width equal to C and softmax activation function such that $f(\mathbf{x}; \theta) \in [0, 1]^C$ and $\sum_{c=1}^{C} f_c(\mathbf{x}; \theta) = 1$ where $f_c(\mathbf{x}; \theta)$ is the c-th entry of $f(\mathbf{x}; \theta)$.
- The NLL gives the cross-entropy loss:

$$\ell(y, f(\mathbf{x}; \theta)) = -\sum_{c=1}^{C} \mathbf{1}_{y=c} \ln \Big(f_c(\mathbf{x}; \theta) \Big) + cst(\theta).$$

Regression

- Training data: $(\mathbf{x}, \mathbf{y}) \in \mathcal{X} \times \mathcal{Y}$ with $\mathcal{X} = \mathbb{R}^p$ and $\mathcal{Y} = \mathbb{R}^q$.
- Output layer: Width equal to q and identity activation function such that $f(\mathbf{x}; \theta) \in \mathbb{R}^q$.
- Model: We assume $p(\mathbf{y}|\mathbf{x}; \theta) = \mathcal{N}\Big(\mathbf{y}; f(\mathbf{x}; \theta), \mathbf{I}\Big) = (2\pi)^{-q/2} \exp\Big(-\frac{1}{2} \parallel \mathbf{y} - f(\mathbf{x}; \theta) \parallel_2^2\Big)$

  .
- The NLL gives the squared error loss:

$$\ell(y, f(\mathbf{x}; \theta)) = \frac{1}{2} \parallel \mathbf{y} - f(\mathbf{x}; \theta) \parallel_2^2 + cst(\theta).$$

Gradient descent and backpropagation

Gradient descent

Gradient descent is an iterative optimization algorithm. It minimizes $\mathcal{L}(\theta)$ by updating the model parameters iteratively according to the following update rule:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta),$$

where

- θ_0 are the initial parameters of the model;
- η is the learning rate;
- both have a critical influence on the behavior of the algorithm.

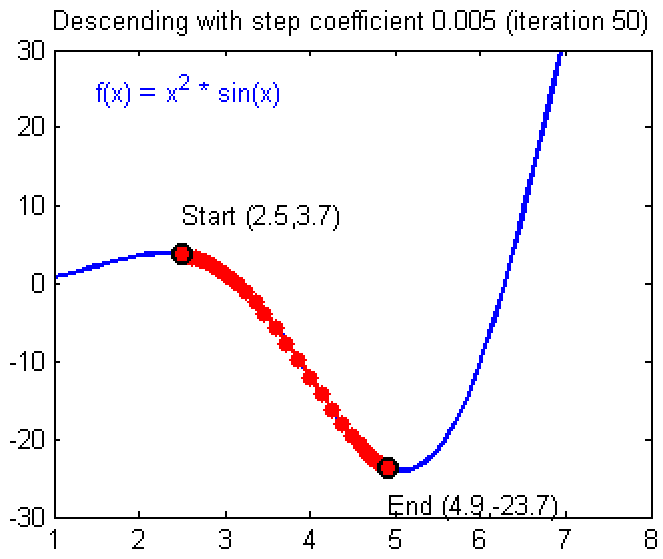


Image credits: Antoine Liutkus and Fabian-Robert Stöter, Deep learning for music separation, Tutorial at EUSIPCO 2019.

$\mathcal{L}(\theta)$ and its gradient decompose as

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{(\mathbf{x}_i, y_i) \in \mathcal{D}} \ell(y_i, f(\mathbf{x}_i; \theta))$$

$$\nabla \mathcal{L}(\theta) = \frac{1}{N} \sum_{(\mathbf{x}_i, y_i) \in \mathcal{D}} \nabla \ell(y_i, f(\mathbf{x}_i; \theta)).$$

Therefore, in standard gradient descent the complexity of an update grows linearly with the size N of the dataset.

In other words, to perform one single update of the parameters, we should do a complete pass over the N examples in the dataset in order to compute and sum N gradients.

Instead, stochastic gradient descent uses as update rule:

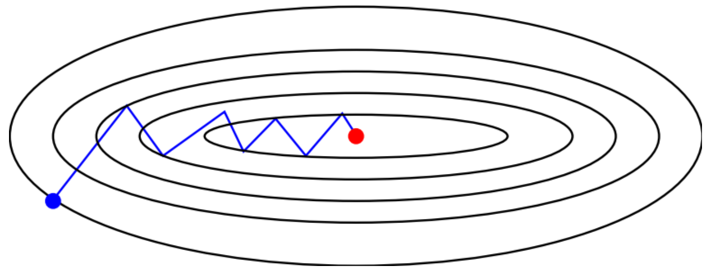
$$\theta_{t+1} = \theta_t - \eta \nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))$$

- The complexity of an update is independent of N.
- The stochastic process $\{\theta_t | t = 1, ...\}$ depends on the examples $i(t)$ picked randomly at each iteration.

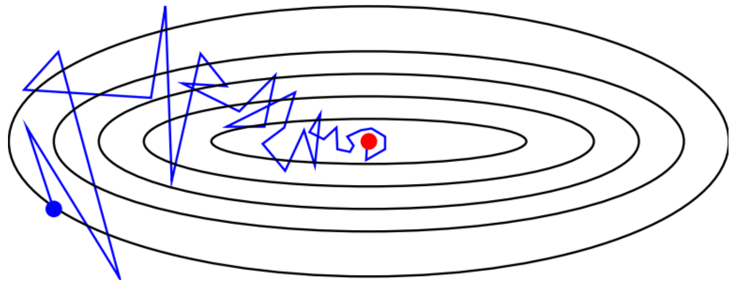
Instead, stochastic gradient descent uses as update rule:

$$\theta_{t+1} = \theta_t - \eta \nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))$$

- The complexity of an update is independent of N.
- The stochastic process $\{\theta_t | t = 1, ...\}$ depends on the examples $i(t)$ picked randomly at each iteration.



Batch gradient descent



Stochastic gradient descent

Credits: Gilles Louppe, INFO8010 - Deep Learning, ULiège.

In mini-batch gradient descent, we compute an average gradient over a small batch of training examples in order to perform an update of the model parameters:

$$\theta_{t+1} = \theta_t - \eta \sum_{(\mathbf{x}_i, y_i) \in \mathcal{B}(t+1)} \nabla \ell(y_i, f(\mathbf{x}_i; \theta_t)).$$

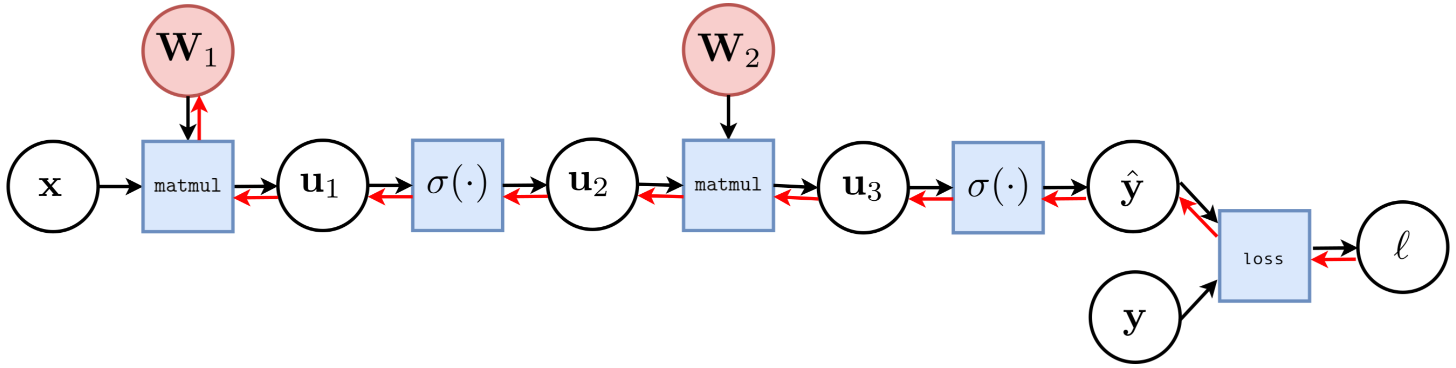
Backpropagation

To minimize $\mathcal{L}(\theta)$ with gradient descent, we need the evaluation of the (total) derivatives

$$\frac{\mathrm{d}\ell}{\mathrm{d}\mathbf{W}_k}, \frac{\mathrm{d}\ell}{\mathrm{d}\mathbf{b}_k}$$

of the loss ℓ with respect to all model parameters $\mathbf{W}_k$, $\mathbf{b}_k$, for all layers $k = 1, ..., L$.

- Since a neural network is a composition of differentiable functions, the derivatives of the loss can be evaluated backward, by applying the chain rule recursively over its computational graph.
- The implementation of this procedure is called reverse automatic differentiation, or backpropagation.



- Forward pass: values $\mathbf{u}_1$, $\mathbf{u}_2$, $\mathbf{u}_3$, $\hat{\mathbf{y}}$ and ℓ are computed by traversing the graph from inputs to outputs given $\mathbf{x}$, $\mathbf{y}$, $\mathbf{W}_1$ and $\mathbf{W}_2$.
- Backward pass: by the chain rule we have

$$\begin{aligned}\frac{\mathrm{d}\ell}{\mathrm{d}\mathbf{W}_1} &= \frac{\partial \ell}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{u}_3} \frac{\partial \mathbf{u}_3}{\partial \mathbf{u}_2} \frac{\partial \mathbf{u}_2}{\partial \mathbf{u}_1} \frac{\partial \mathbf{u}_1}{\partial \mathbf{W}_1} \\ &= \frac{\partial \ell}{\partial \hat{\mathbf{y}}} \frac{\partial \sigma(\mathbf{u}_3)}{\partial \mathbf{u}_3} \frac{\partial \mathbf{W}_2^T \mathbf{u}_2}{\partial \mathbf{u}_2} \frac{\partial \sigma(\mathbf{u}_1)}{\partial \mathbf{u}_1} \frac{\partial \mathbf{W}_1^T \mathbf{x}}{\partial \mathbf{W}_1}\end{aligned}$$

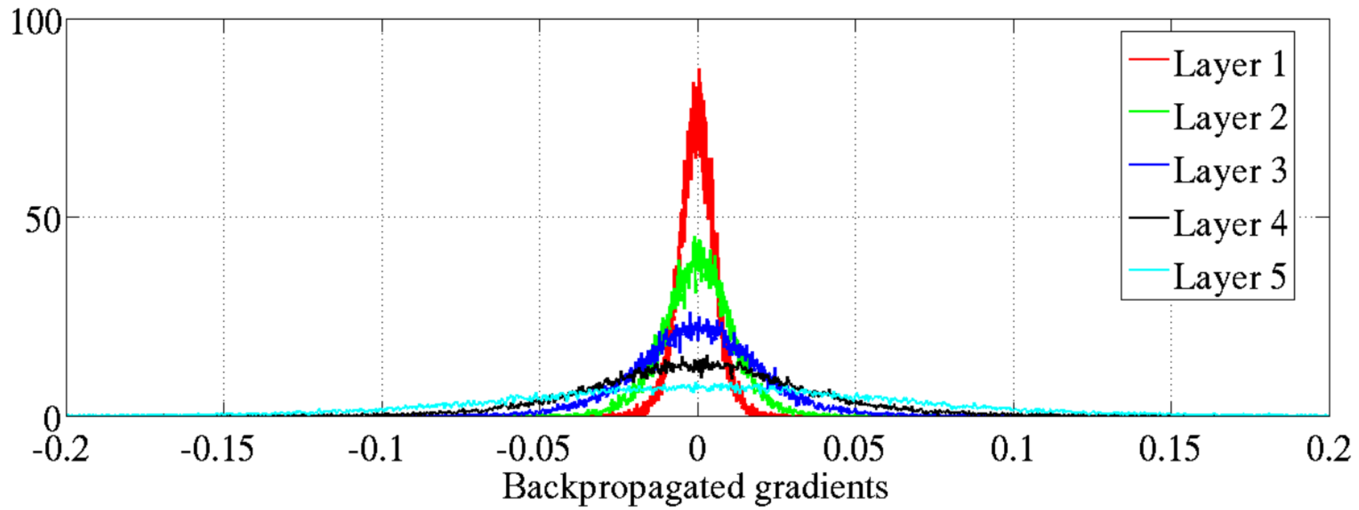
For a refresher about gradients, Jacobian, etc.: "Computing Neural Network Gradients" by K. Clark (CS224n course @ Stanford).

All deep learning libraries (Tensorflow, PyTorch, etc.) implement backpropagation for you, such that training a neural network is as simple as a few lines of code.

Vanishing gradients

Training deep MLPs with many layers has for long (pre-2011) been very difficult due to the vanishing gradient problem.

- Small gradients slow down, and eventually block, stochastic gradient descent.
- This results in a limited capacity of learning.



Backpropagated gradients normalized histograms (Glorot and Bengio, 2010).
Gradients for layers far from the output vanish to zero.

Let us consider a simplified 3-layer MLP, with $x, w_1, w_2, w_3 \in \mathbb{R}$, such that

$$f(x; w_1, w_2, w_3) = \sigma\left(w_3 \sigma\left(w_2 \sigma\left(w_1 x\right)\right)\right).$$

Let us consider a simplified 3-layer MLP, with $x, w_1, w_2, w_3 \in \mathbb{R}$, such that

$$f(x; w_1, w_2, w_3) = \sigma\left(w_3 \sigma\left(w_2 \sigma\left(w_1 x\right)\right)\right).$$

Under the hood, this would be evaluated as

$$u_1 = w_1 x; \qquad u_2 = \sigma(u_1); \qquad u_3 = w_2 u_2; \qquad u_4 = \sigma(u_3); \qquad u_5 = w_3 u_4; \qquad \hat{y} = \sigma(u_5)$$

Let us consider a simplified 3-layer MLP, with $x, w_1, w_2, w_3 \in \mathbb{R}$, such that

$$f(x; w_1, w_2, w_3) = \sigma\left(w_3 \sigma\left(w_2 \sigma\left(w_1 x\right)\right)\right).$$

Under the hood, this would be evaluated as

$$u_1 = w_1 x; \qquad u_2 = \sigma(u_1); \qquad u_3 = w_2 u_2; \qquad u_4 = \sigma(u_3); \qquad u_5 = w_3 u_4; \qquad \hat{y} = \sigma(u_5)$$

We can apply the chain rule to compute the derivative

$$\begin{aligned} \frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1} &= \frac{\partial \hat{y}}{\partial u_5} \frac{\partial u_5}{\partial u_4} \frac{\partial u_4}{\partial u_3} \frac{\partial u_3}{\partial u_2} \frac{\partial u_2}{\partial u_1} \frac{\partial u_1}{\partial w_1} \\ &= \frac{\partial \sigma(u_5)}{\partial u_5} w_3 \frac{\partial \sigma(u_3)}{\partial u_3} w_2 \frac{\partial \sigma(u_1)}{\partial u_1} x \end{aligned}$$

Let us consider a simplified 3-layer MLP, with $x, w_1, w_2, w_3 \in \mathbb{R}$, such that

$$f(x; w_1, w_2, w_3) = \sigma\left(w_3 \sigma\left(w_2 \sigma\left(w_1 x\right)\right)\right).$$

Under the hood, this would be evaluated as

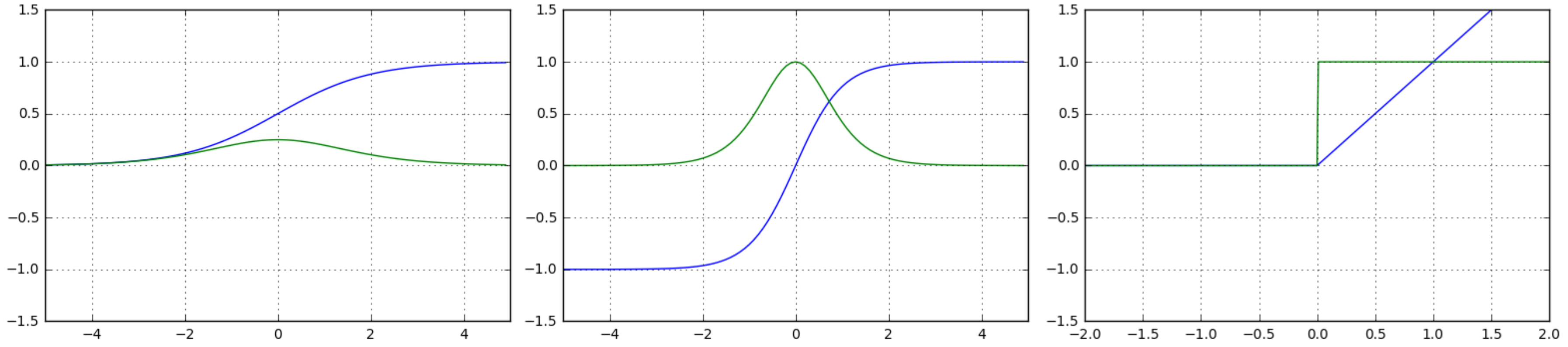
$$u_1 = w_1 x; \qquad u_2 = \sigma(u_1); \qquad u_3 = w_2 u_2; \qquad u_4 = \sigma(u_3); \qquad u_5 = w_3 u_4; \qquad \hat{y} = \sigma(u_5)$$

We can apply the chain rule to compute the derivative

$$\begin{aligned} \frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1} &= \frac{\partial \hat{y}}{\partial u_5} \frac{\partial u_5}{\partial u_4} \frac{\partial u_4}{\partial u_3} \frac{\partial u_3}{\partial u_2} \frac{\partial u_2}{\partial u_1} \frac{\partial u_1}{\partial w_1} \\ &= \frac{\partial \sigma(u_5)}{\partial u_5} w_3 \frac{\partial \sigma(u_3)}{\partial u_3} w_2 \frac{\partial \sigma(u_1)}{\partial u_1} x \end{aligned}$$

If $\left|w_j\right| \leq 1$ and/or $\left|\frac{\partial \sigma(u_j)}{\partial u_j}\right| \leq 1$, the gradient exponentially shrinks to zero as the number of layers in the network increases.

Hence the vanishing gradient problem.



$$\mathrm{sigm}(x) = \frac{1}{1+e^{-x}} \qquad \tanh(x) = \frac{e^{2x}-1}{e^{2x}+1} \qquad \mathrm{relu}(x) = \max(0, x)$$

$$\mathrm{sigm}'(x) = \mathrm{sigm}(x)(1-\mathrm{sigm}(x)) \qquad \tanh'(x) = 1 - \tanh(x)^2 \qquad \mathrm{relu}'(x) = 1_{x>0}$$

- In general, bounded activation functions (sigmoid, tanh, etc.) are prone to the vanishing gradient problem. ReLUs do not have this issue.
- Note also the importance of the weights initialization.

Image credit: Olivier Grisel and Charles Ollion

Over- and under-fitting

The **capacity** of our deep learning model can be controlled through hyper-parameters, for example:

- number of layers in a neural network;
- number of neurons in each layer;
- number of training iterations;
- regularization terms;

Our goal is to adjust the capacity of the model such that the error gets as low as possible on examples that were not used for training.

We speak about the generalization capability of the model.

When the generalization error starts to increase while the training error is still decreasing, we say that the model is overfitting.

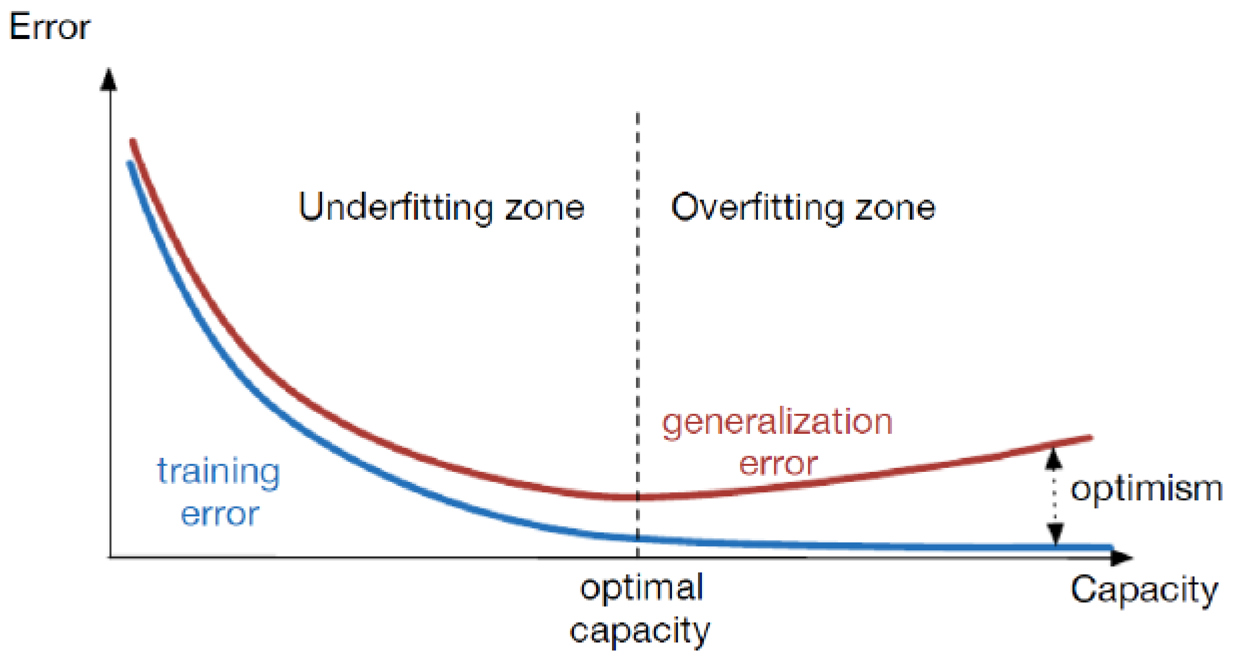
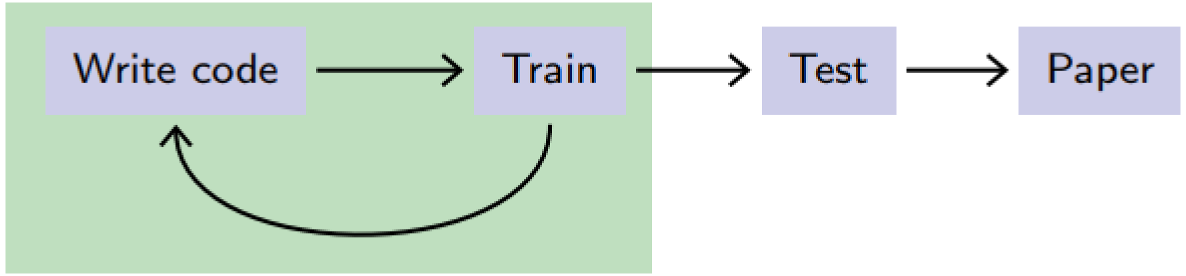
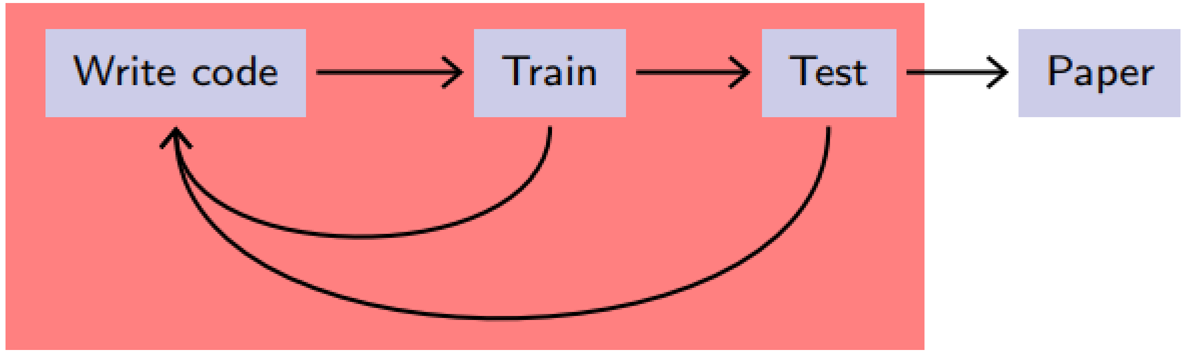


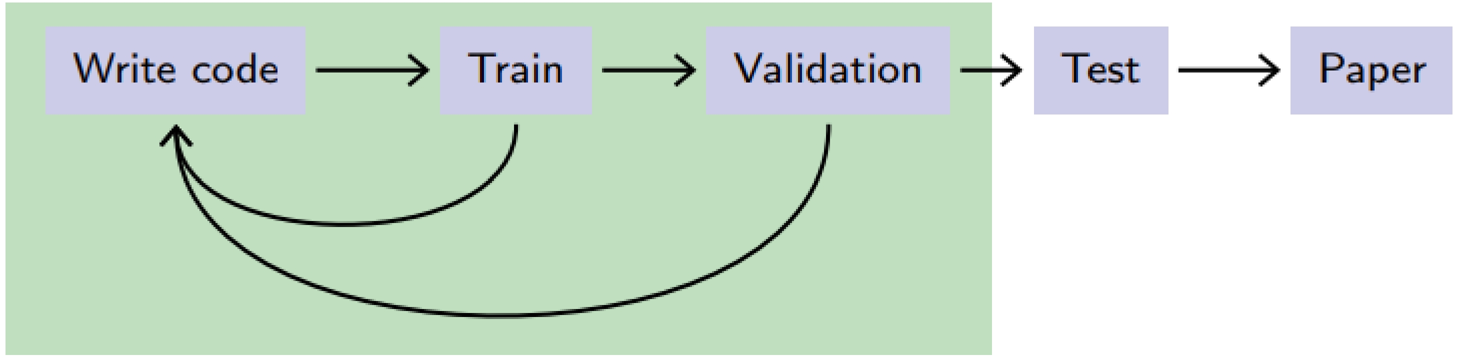
Image credits: Gilles Louppe, INFO8010 - Deep Learning, ULiège.



There may be over-fitting, but it does not bias the final performance evaluation.



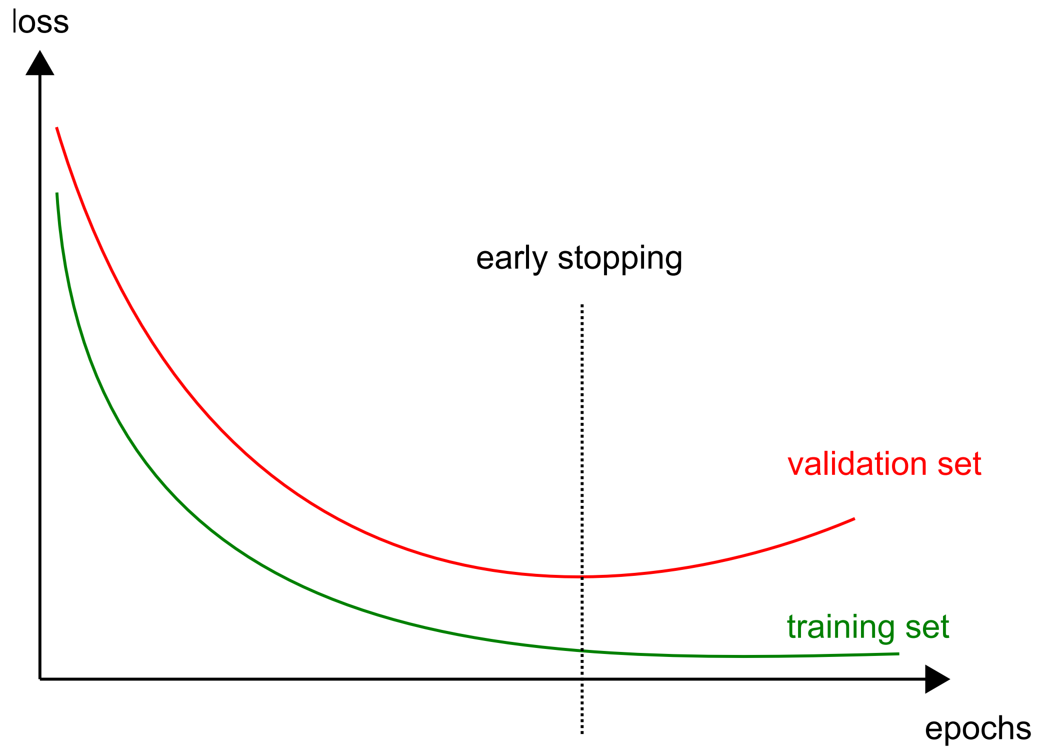
This should be avoided at all costs!

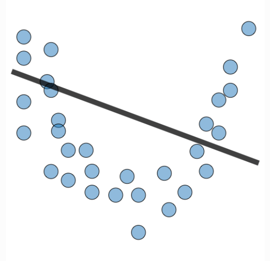
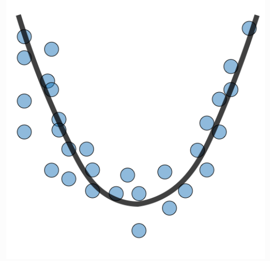
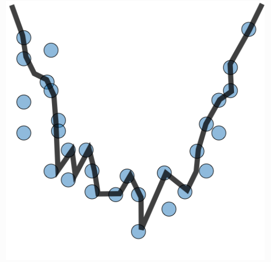
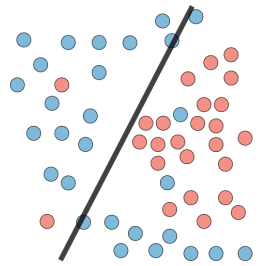
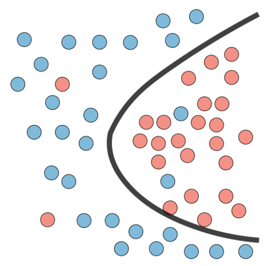
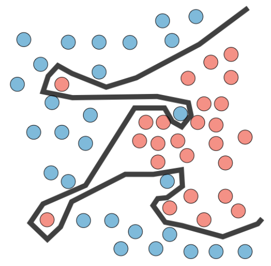
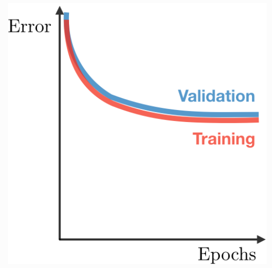
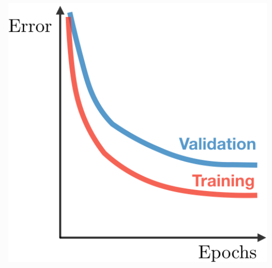
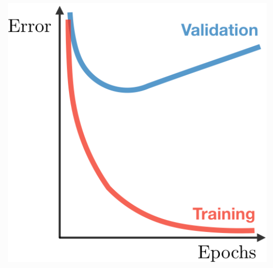


Instead, keep a separate validation set for tuning the hyper-parameters (e.g. 20 % of the original training dataset).

A simple strategy to prevent overfitting is to stop the training when the error starts to increase on the validation set.

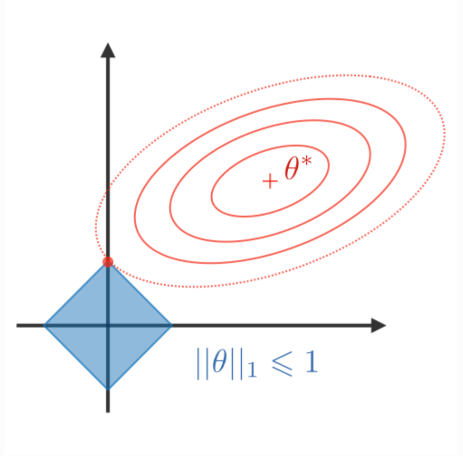
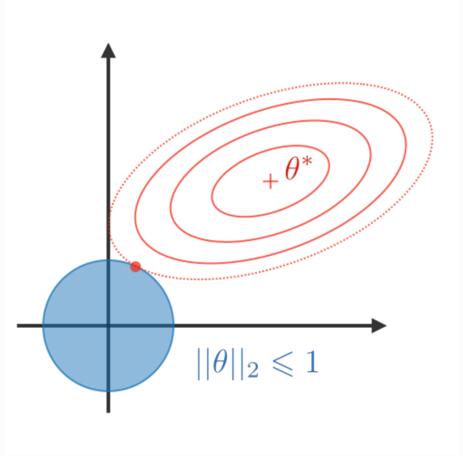
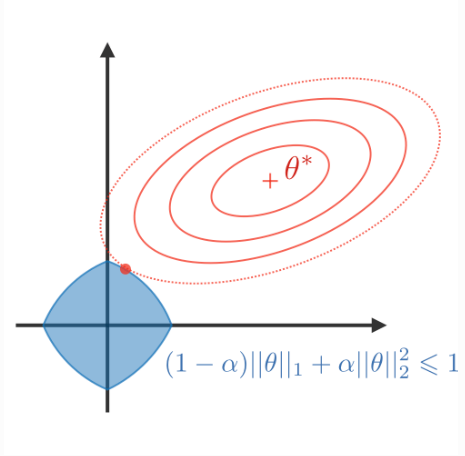
This is called early stopping.



	Underfitting	Just right	Overfitting
Symptoms	High training error; Training error close to test error	Training error slightly lower than test error	Very low training error; Training error much lower than test error
Regression			
Classification			
Learning curve			
Possible remedies	Complexify model; Add more features; Train longer		Perform regularization; Get more data

Credits: https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-machine-learning-tips-and-tricks

Regularization consits in constraining the model parameters to avoid overfitting. To do so, we add a regularization term to the loss function.

LASSO	Ridge	Elastic Net
• Shrinks coefficients to 0 • Good for variable selection	Makes coefficients smaller	Tradeoff between variable selection and small coefficients
		 θ^* $(1-\alpha)\|\|\theta\|\|_1 + \alpha\|\|\theta\|\|_2^2 \leqslant 1$
$... + \lambda\|\|\theta\|\|_1$ $\lambda \in \mathbb{R}$	$... + \lambda\|\|\theta\|\|_2^2$ $\lambda \in \mathbb{R}$	$... + \lambda\Big[(1-\alpha)\|\|\theta\|\|_1 + \alpha\|\|\theta\|\|_2^2\Big]$ $\lambda \in \mathbb{R}, \alpha \in [0,1]$

Credits: https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-machine-learning-tips-and-tricks

Deep learning

Recent advances and model architectures in deep learning are built on a natural generalization of a neural network: a graph of tensor operators, taking advantage of

- the chain rule,
- stochastic gradient descent,
- parallel operations on GPUs.

This does not differ much from networks from the 90s, as covered in today's lecture.

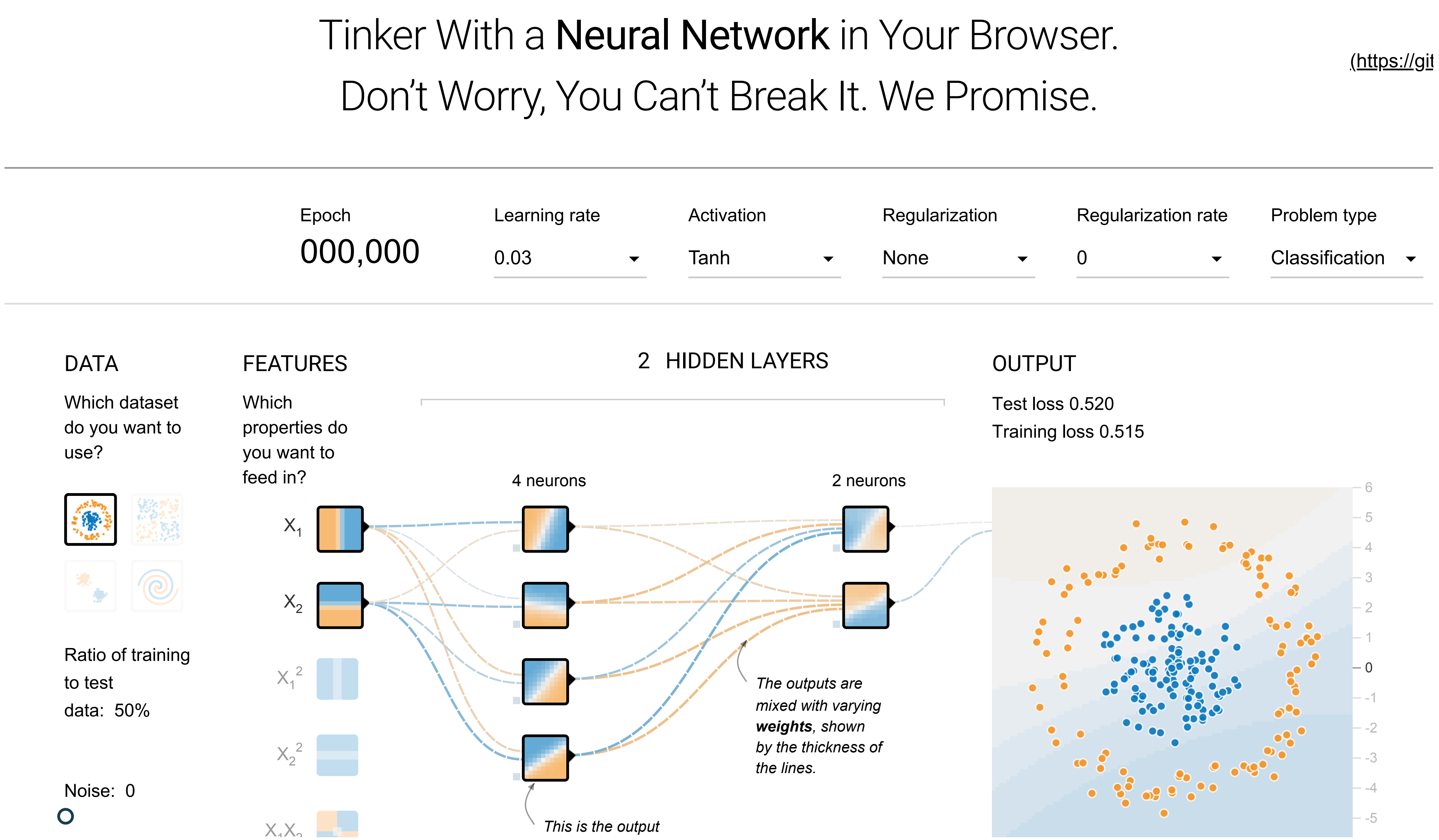
This generalization allows to compose and design complex networks of operators, possibly dynamically, dealing with images, sound, text, sequences, etc. and to train them end-to-end.

Credits: Francois Fleuret, EE559 Deep Learning, EPFL.

Cooking recipe

- Get data (loads of them).
- Get good hardware.
- Define the neural network architecture as a composition of differentiable functions.
- Define a differentiable loss function.
- Optimize with (variants of) stochastic gradient descent.

Credits: Gilles Louppe, INFO8010 - Deep Learning, ULiège.



Tinker With a **Neural Network** in Your Browser.
Don't Worry, You Can't Break It. We Promise.

(https://git

Epoch 000,000

Learning rate 0.03

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

FEATURES

Which properties do you want to feed in?

2 HIDDEN LAYERS

4 neurons

2 neurons

*The outputs are mixed with varying **weights**, shown by the thickness of the lines.*

This is the output

OUTPUT

Test loss 0.520

Training loss 0.515