

# Deep Learning

Last update: 28 March 2024

Academic Year 2023 – 2024

Alma Mater Studiorum · University of Bologna

# Contents

<b>1</b>	<b>Neural networks expressivity</b>	<b>1</b>
1.1	Perceptron . . . . .	1
1.2	Multi-layer perceptron . . . . .	1
1.2.1	Parameters . . . . .	1
<b>2</b>	<b>Training</b>	<b>2</b>
2.1	Gradient descent . . . . .	2
2.2	Backpropagation . . . . .	3
<b>3</b>	<b>Convolutional neural networks</b>	<b>5</b>
3.1	Parameters . . . . .	6

# 1 Neural networks expressivity

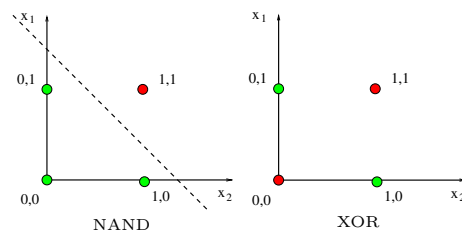
## 1.1 Perceptron

Single neuron that defines a binary threshold through a hyperplane:

$$\begin{cases} 1 & \sum_i w_i x_i + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

**Expressivity** A perceptron can represent a NAND gate but not a XOR gate.

Perceptron  
expressivity



**Remark.** Even if NAND is logically complete, the strict definition of a perceptron is not a composition of them.

## 1.2 Multi-layer perceptron

Composition of perceptrons.

**Shallow neural network** Neural network with one hidden layer.

Shallow NN

**Deep neural network** Neural network with more than one hidden layer.

Deep NN

**Expressivity** Shallow neural networks allow to approximate any continuous function

Multi-layer  
perceptron  
expressivity

$$f : \mathbb{R} \rightarrow [0, 1]$$

**Remark.** Still, deep neural networks allow to use less neural units.

### 1.2.1 Parameters

The number of parameters of a layer is given by:

$$S_{\text{in}} \cdot S_{\text{out}} + S_{\text{out}}$$

where:

- $S_{\text{in}}$  is the dimension of the input of the layer.
- $S_{\text{out}}$  is the dimension of the output of the layer.

Therefore, the number of FLOPS is of order:

$$S_{\text{in}} \cdot S_{\text{out}}$$

## 2 Training

### 2.1 Gradient descent

1. Start from a random set of weights  $w$ .
2. Compute the gradient  $\nabla \mathcal{L}$  of the loss function.
3. Make a small step of size  $-\nabla \mathcal{L}(w)$ .
4. Go to 2., until convergence.

Gradient descent

**Learning rate** Size of the step. Usually denoted with  $\mu$ .

Learning rate

$$w = w + \mu \nabla \mathcal{L}(w)$$

**Optimizer** Algorithm that tunes the learning rate during training.

Optimizer

**Stochastic gradient descent** Use a subset of the training data to compute the gradient.

Stochastic gradient descent

**Full-batch** Use the entire dataset.

**Mini-batch** Use a subset of the training data.

**Online** Use a single sample.

**Remark.** SGD with mini-batch converges to the same result obtained using a full-batch approach.

**Momentum** Correct the update  $v_t$  at time  $t$  considering the update  $v_{t-1}$  of time  $t - 1$ .

Momentum

$$\begin{aligned} w_{t+1} &= w_t + v_t \\ v_t &= \mu \nabla \mathcal{L}(w_t) + \alpha v_{t-1} \end{aligned}$$

**Nesterov momentum** Apply the momentum before computing the gradient.

Nesterov momentum

**Overfitting** Model too specialized on the training data.

Overfitting

Methods to reduce overfitting are:

- Increasing the dataset size.
- Simplifying the model.
- Early stopping.
- Regularization.
- Model averaging.
- Neurons dropout.

**Underfitting** Model too simple and unable to capture features of the training data.

Underfitting

## 2.2 Backpropagation

**Chain rule** Refer to SMM for AI (Section 5.1.1).

Chain rule

**Backpropagation** Algorithm to compute the gradient at each layer of a neural network.

Backpropagation

The output of the  $i$ -th neuron in the layer  $l$  of a neural network can be defined as:

$$a_{l,i} = \sigma_{l,i}(\mathbf{w}_{l,i}^T \mathbf{a}_{l-1} + b_{l,i}) = \sigma_{l,i}(z_{l,i})$$

where:

- $a_{l,i} \in \mathbb{R}$  is the output of the neuron.
- $\mathbf{w}_{l,i} \in \mathbb{R}^{n_{l-1}}$  is the vector of weights.
- $\mathbf{a}_{l-1} \in \mathbb{R}^{n_{l-1}}$  is the vector of the outputs of the previous layer.
- $b_{l,i} \in \mathbb{R}$  is the bias.
- $\sigma_{l,i} : \mathbb{R} \rightarrow \mathbb{R}$  is the activation function<sup>1</sup>.
- $z_{l,i}(\mathbf{w}_{l,i}, b_{l,i} | \mathbf{a}_{l-1}) = \mathbf{w}_{l,i}^T \mathbf{a}_{l-1} + b_{l,i}$  is the argument of the activation function and is parametrized on  $\mathbf{w}_{l,i}$  and  $b_{l,i}$ .

Hence, the outputs of the  $l$ -th layer can be defined as:

$$\mathbf{a}_l = \sigma_l(\mathbf{W}_l^T \mathbf{a}_{l-1} + \mathbf{b}_l) = \sigma_l(\mathbf{z}_l(\mathbf{W}_l, \mathbf{b}_l | \mathbf{a}_{l-1}))$$

where:

- $\sigma_l : \mathbb{R}^{n_l} \rightarrow \mathbb{R}^{n_l}$  is the element-wise activation function.
- $\mathbf{W}_l \in \mathbb{R}^{n_l \times n_{l-1}}$ ,  $\mathbf{a}_{l-1} \in \mathbb{R}^{n_{l-1}}$ ,  $\mathbf{b}_l \in \mathbb{R}^{n_l}$ ,  $\mathbf{a}_l \in \mathbb{R}^{n_l}$ .

Finally, a neural network with input  $\mathbf{x}$  can be expressed as:

$$\begin{aligned} \mathbf{a}_0 &= \mathbf{x} \\ \mathbf{a}_i &= \sigma_i(\mathbf{z}_i(\mathbf{W}_i, \mathbf{b}_i | \mathbf{a}_{i-1})) \end{aligned}$$

Given a neural network with  $K$  layers and a loss function  $\mathcal{L}$ , we want to compute the derivative of  $\mathcal{L}$  w.r.t. the weights of each layer to tune the parameters.

First, we highlight the parameters of each of the functions involved:

**Loss**  $\mathcal{L}(a_K) = \mathcal{L}(\sigma_K)$  takes as input the output of the network (i.e. the output of the last activation function).

**Activation function**  $\sigma_i(\mathbf{z}_i)$  takes as input the value of the neurons at the  $i$ -th layer.

**Neurons**  $\mathbf{z}_i(\mathbf{W}_i, \mathbf{b}_i)$  takes as input the weights and biases at the  $i$ -th layer.

Let  $\odot$  be the Hadamard product. By exploiting the chain rule, we can compute the derivatives w.r.t. the weights going backward:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_K} = \frac{\partial \mathcal{L}}{\partial \sigma_K} \frac{\partial \sigma_K}{\partial \mathbf{z}_K} \frac{\partial \mathbf{z}_K}{\partial \mathbf{W}_K} = \underbrace{\nabla \mathcal{L}(\mathbf{a}_K)}_{\mathbb{R}^{n_K \times 1}} \odot \underbrace{\nabla \sigma_K(\mathbf{z}_K)}_{\mathbb{R}^{n_K \times 1}} \cdot \underbrace{\mathbf{a}_{K-1}^T}_{1 \times \mathbb{R}^{n_{K-1}}} \in \mathbb{R}^{n_K \times n_{K-1}}$$

---

<sup>1</sup>Even if it is possible to have a different activation function in each neuron, in practice, each layer has the same activation function.

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{K-1}} &= \frac{\partial \mathcal{L}}{\partial \sigma_K} \frac{\partial \sigma_K}{\partial \mathbf{z}_K} \frac{\partial \mathbf{z}_K}{\partial \sigma_{K-1}} \frac{\partial \sigma_{K-1}}{\partial \mathbf{z}_{K-1}} \frac{\partial \mathbf{z}_{K-1}}{\partial \mathbf{W}_{K-1}} \\
&= \underbrace{(\nabla \mathcal{L}(\mathbf{a}_K))}_{\mathbb{R}^{n_K \times 1}} \odot \underbrace{\nabla \sigma_K(\mathbf{z}_K)}_{\mathbb{R}^{n_K \times 1}})^T \cdot \underbrace{\mathbf{W}_K}_{\mathbb{R}^{n_K \times \mathbb{R}^{n_{K-1}}}} \odot \underbrace{\nabla \sigma_{K-1}(\mathbf{z}_{K-1})}_{\mathbb{R}^{n_{K-1} \times 1}} \cdot \underbrace{\mathbf{a}_{K-2}^T}_{1 \times \mathbb{R}^{n_{K-2}}} \in \mathbb{R}^{n_{K-1} \times n_{K-2}} \\
&\vdots
\end{aligned}$$

In the same way, we can compute the derivatives w.r.t. the biases:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{b}_K} &= \frac{\partial \mathcal{L}}{\partial \sigma_K} \frac{\partial \sigma_K}{\partial \mathbf{z}_K} \frac{\partial \mathbf{z}_K}{\partial \mathbf{b}_K} = \underbrace{\nabla \mathcal{L}(\mathbf{a}_K)}_{\mathbb{R}^{n_K \times 1}} \odot \underbrace{\nabla \sigma_K(\mathbf{z}_K)}_{\mathbb{R}^{n_K \times 1}} \cdot 1 \in \mathbb{R}^{n_K} \\
\frac{\partial \mathcal{L}}{\partial \mathbf{b}_{K-1}} &= \frac{\partial \mathcal{L}}{\partial \sigma_K} \frac{\partial \sigma_K}{\partial \mathbf{z}_K} \frac{\partial \mathbf{z}_K}{\partial \sigma_{K-1}} \frac{\partial \sigma_{K-1}}{\partial \mathbf{z}_{K-1}} \frac{\partial \mathbf{z}_{K-1}}{\partial \mathbf{b}_{K-1}} \\
&= \underbrace{(\nabla \mathcal{L}(\mathbf{a}_K))}_{\mathbb{R}^{n_K \times 1}} \odot \underbrace{\nabla \sigma_K(\mathbf{z}_K)}_{\mathbb{R}^{n_K \times 1}})^T \cdot \underbrace{\mathbf{W}_K}_{\mathbb{R}^{n_K \times \mathbb{R}^{n_{K-1}}}} \odot \underbrace{\nabla \sigma_{K-1}(\mathbf{z}_{K-1})}_{\mathbb{R}^{n_{K-1} \times 1}} \cdot 1 \in \mathbb{R}^{n_{K-1}} \\
&\vdots
\end{aligned}$$

It can be noticed that many terms are repeated from one layer to another. By exploiting this, we can store the following intermediate values:

$$\begin{aligned}
\delta_K &= \frac{\partial \mathcal{L}}{\partial \mathbf{z}_K} = \frac{\partial \mathcal{L}}{\partial \sigma_K} \frac{\partial \sigma_K}{\partial \mathbf{z}_K} = \nabla \mathcal{L}(\mathbf{a}_K) \odot \nabla \sigma_K(\mathbf{z}_K) \\
\delta_l &= \frac{\partial \mathcal{L}}{\partial \mathbf{z}_l} = \delta_{l+1}^T \cdot \mathbf{W}_{l+1} \odot \nabla \sigma_l(\mathbf{z}_l)
\end{aligned}$$

and reused them to compute the derivatives as follows:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mathbf{W}_l} &= \frac{\partial \mathcal{L}}{\partial \mathbf{z}_l} \frac{\partial \mathbf{z}_l}{\partial \mathbf{W}_l} = \delta_l \cdot \mathbf{a}_{l-1}^T \\
\frac{\partial \mathcal{L}}{\partial \mathbf{b}_l} &= \frac{\partial \mathcal{L}}{\partial \mathbf{z}_l} \frac{\partial \mathbf{z}_l}{\partial \mathbf{b}_l} = \delta_l \cdot 1
\end{aligned}$$

**Vanishing gradient** As backpropagation consists of a chain of products, when a component is small (i.e.  $< 1$ ), it will gradually cancel out the gradient when backtracking, causing the first layers to learn much slower than the last layers.

Vanishing gradient

**Remark.** This is an issue of the sigmoid function. ReLU was designed to solve this problem.

### 3 Convolutional neural networks

**Convolution neuron** Neuron influenced by only a subset of neurons in the previous layer. Convolution neuron

**Receptive field** Dimension of the input image influencing a neuron. Receptive field

**Convolutional layer** Layer composed of convolutional neurons. Neurons in the same convolutional layer share the same weights and work as a convolutional filter. Convolutional layer

**Remark.** The weights of the filters are learned.

A convolutional layer has the following parameters:

**Kernel size** Dimension (i.e. width and height) of the filter. Kernel size

**Stride** Offset between each filter application (i.e. stride > 1 reduces the size of the output image). Stride

**Padding** Artificial enlargement of the image. Padding

In practice, there are two modes of padding:

**Valid** No padding applied.

**Same** Apply the minimum padding needed.

**Depth** Number of different kernels to apply (i.e. augment the number of channels in the output image). Depth

The dimension along each axis of the output image is given by:

$$\frac{W + P - K}{S} + 1$$

where:

- $W$  is the size of the image (width or height).
- $P$  is the padding.
- $K$  is the kernel size.
- $S$  is the stride.

**Remark.** If not specified, a kernel is applied to all the channels of the input image in parallel (but the weights of the kernel change at each channel).

**Pooling** Layer that applies a function as a filter.

**Max-pooling** Filter that computes the maximum of the pixels within the kernel. Max-pooling

**Mean-pooling** Filter that computes the average of the pixels within the kernel. Mean-pooling

### 3.1 Parameters

The number of parameters of a layer is given by:

$$(K_w \cdot K_h) \cdot D_{\text{in}} \cdot D_{\text{out}} + D_{\text{out}}$$

where:

- $K_w$  is the width of the kernel.
- $K_h$  is the height of the kernel.
- $D_{\text{in}}$  is the input depth.
- $D_{\text{out}}$  is the output depth.

Therefore, the number of FLOPS is of order:

$$(K_w \cdot K_h) \cdot D_{\text{in}} \cdot D_{\text{out}} \cdot (O_w \cdot O_h)$$

where:

- $O_w$  is the width of the output image.
- $O_h$  is the height of the output image.