Deep Learning

Last update: 05 April 2024

Academic Year 2023 – 2024

Alma Mater Studiorum · University of Bologna

Contents

1 Neural networks expressivity

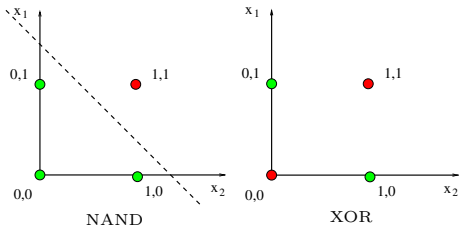
1.1 Perceptron

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

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

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_{in} is the dimension of the input of the layer.
- S_{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

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.

Learning rate

Learning rate Size of the step. Usually denoted with μ.

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

Optimizer

Optimizer Algorithm that tunes the learning rate during training.

Stochastic gradient descent

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

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

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

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

Nesterov momentum

Nesterov momentum Apply the momentum before computing the gradient.

Overfitting

Overfitting Model too specialized on the training data.

Methods to reduce overfitting are:

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

Underfitting

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

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.
- $\sigma_{l,i} : \mathbb{R} \to \mathbb{R}$ is the activation function[1].
- $z_{l,i}(\mathbf{w}_{l,i}, b_{l,i} | \mathbf{a}_{l-1}) = \mathbf{w}_{l,i}^T \mathbf{a}_{l-1} + b_{l,i}$ 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:

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

where:

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

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

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

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 $\mathbf{z}_i(\boldsymbol{W}_i, \mathbf{b}_i)$ 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:

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

[1] Even if it is possible to have a different activation function in each neuron, in practice, each layer has the same activation function.

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

$$\vdots$$

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

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_K} = \frac{\partial \mathcal{L}}{\partial \sigma_K}\frac{\partial \sigma_K}{\partial \mathbf{z}_K}\frac{\partial \mathbf{z}_K}{\partial \mathbf{b}_K} = \underset{\mathbb{R}^{n_K\times 1}}{\nabla \mathcal{L}(\mathbf{a}_K)} \odot \underset{\mathbb{R}^{n_K\times 1}}{\nabla \sigma_K(\mathbf{z}_K)} \cdot 1 \in \mathbb{R}^{n_K}$$

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

$$\vdots$$

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

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

and reused them to compute the derivatives as follows:

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

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

3.1 Convolutions

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).

3.1.1 Parameters

The number of parameters of a convolutional layer is given by:

$$(K_{\mathrm{w}} \cdot K_{\mathrm{h}}) \cdot D_{\mathrm{in}} \cdot D_{\mathrm{out}} + D_{\mathrm{out}}$$

where:

- K_{w} is the width of the kernel.

- K_h is the height of the kernel.
- D_in is the input depth.
- D_out is the output depth.

Therefore, the number of FLOPS is of order:

$$(K_\text{w} \cdot K_\text{h}) \cdot D_\text{in} \cdot D_\text{out} \cdot (O_\text{w} \cdot O_\text{h})$$

where:

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

3.2 Backpropagation

A convolution can be expressed as a dense layer by representing it through a sparse matrix. Therefore, backpropagation can be executed in the standard way, with the only exception that the positions of the convolution matrix corresponding to the same cell of the kernel should be updated with the same value (e.g. the mean of all the corresponding updates).

Example. Given a 4×4 image I and a 3×3 kernel K with stride 1 and no padding:

$$I = \begin{pmatrix} i_{0,0} & i_{0,1} & i_{0,2} & i_{0,3} \\ i_{1,0} & i_{1,1} & i_{1,2} & i_{1,3} \\ i_{2,0} & i_{2,1} & i_{2,2} & i_{2,3} \\ i_{3,0} & i_{3,1} & i_{3,2} & i_{3,3} \end{pmatrix} \qquad K = \begin{pmatrix} w_{0,0} & w_{0,1} & w_{0,2} \\ w_{1,0} & w_{1,1} & w_{1,2} \\ w_{2,0} & w_{2,1} & w_{2,2} \end{pmatrix}$$

The convolutional layer can be represented through a convolutional matrix and by flattening the image as follows:

$$\begin{pmatrix} w_{0,0} & 0 & 0 & 0 \\ w_{0,1} & w_{0,0} & 0 & 0 \\ w_{0,2} & w_{0,1} & 0 & 0 \\ 0 & w_{0,2} & 0 & 0 \\ w_{1,0} & 0 & w_{0,0} & 0 \\ w_{1,1} & w_{1,0} & w_{0,1} & w_{0,0} \\ w_{1,2} & w_{1,1} & w_{0,2} & w_{0,1} \\ 0 & w_{1,2} & 0 & w_{0,2} \\ w_{2,0} & 0 & w_{1,0} & 0 \\ w_{2,1} & w_{2,0} & w_{1,1} & w_{1,0} \\ w_{2,2} & w_{2,1} & w_{1,2} & w_{1,1} \\ 0 & w_{2,2} & 0 & w_{1,2} \\ 0 & 0 & w_{2,0} & 0 \\ 0 & 0 & w_{2,1} & w_{2,0} \\ 0 & 0 & w_{2,2} & w_{2,1} \\ 0 & 0 & 0 & w_{2,2} \end{pmatrix}^T \cdot \begin{pmatrix} i_{0,0} \\ i_{0,1} \\ i_{0,2} \\ i_{0,3} \\ i_{1,0} \\ i_{1,1} \\ i_{1,2} \\ i_{1,3} \\ i_{2,0} \\ i_{2,1} \\ i_{2,2} \\ i_{2,3} \\ i_{3,0} \\ i_{3,1} \\ i_{3,2} \\ i_{3,3} \end{pmatrix} = \begin{pmatrix} o_{0,0} \\ o_{0,1} \\ o_{1,0} \\ o_{1,1} \end{pmatrix} \mapsto \begin{pmatrix} o_{0,0} & o_{0,1} \\ o_{1,0} & o_{1,1} \end{pmatrix}$$

3.3 Pooling layer

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

3.4 Inception hypothesis

Depth-wise separable convolution Decompose a 3D kernel into a 2D kernel followed by a 1D kernel.

Depth-wise separable convolution

Given an input image with C_{in} channels, a single pass of a traditional 3D convolution uses a kernel of shape $k \times k \times C_{\text{in}}$ to obtain an output of 1 channel. This is repeated for a desired C_{out} number of times (with different kernels).

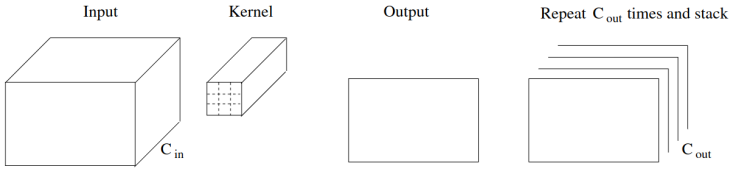


Figure 3.1: Example of traditional convolution

A single pass of a depth-wise separable convolution uses C_{in} different $k \times k \times 1$ kernels first to obtain C_{in} images. Then, a $1 \times 1 \times C_{\text{in}}$ kernel is used to obtain an output image of 1 channel. The last 1D kernel is repeated for a C_{out} number of times (with different kernels).

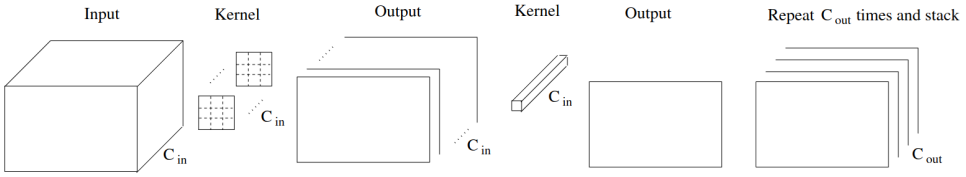


Figure 3.2: Example of depth-wise separable convolution

3.4.1 Parameters

The number of parameters of a depth-wise separable convolutional layer is given by:

$$(K_{\text{w}} \cdot K_{\text{h}}) \cdot D_{\text{in}} + (1 \cdot 1 \cdot D_{\text{in}}) \cdot D_{\text{out}}$$

where:

- K_{w} is the width of the kernel.
- K_{h} is the height of the kernel.
- D_{in} is the input depth.
- D_{out} is the output depth.

3.5 Residual learning

Residual connection Sum the input of a layer to its output.

Residual connection

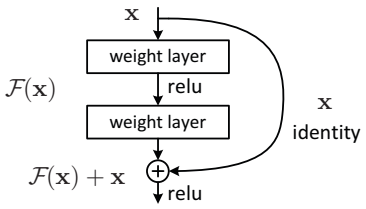


Figure 3.3: Residual connection

Remark. The sum operation can be substituted with the concatenation.

Remark. The effectiveness of residual connections is only shown empirically.

Remark. By adding the input, without passing through the activation function, might help to propagate the gradient from higher layers to lower layers and avoid the risk of vanishing gradient.

Another interpretation is that, by learning the function $F(x) + x$, it is easier for the model to represent, if it needs to, the identity function as the problem is reduced to learn $F(x) = 0$. On the other hand, without a residual connection, learning $F(x) = x$ from scratch might be harder.

3.6 Transfer learning and fine-tuning

Transfer learning

Transfer learning Reuse an existing model by appending some new layers to it. Only the new layers are trained.

Fine-tuning

Fine-tuning Reuse an existing model by appending some new layers to it. The existing model (or part of it) is trained alongside the new layers.

Remark. In computer vision, reusing an existing model makes sense as the first convolutional layers tend to learn primitive concepts that are independent of the downstream task.

3.7 Other convolution types

Transposed convolution / Deconvolution

Transposed convolution / Deconvolution Convolution to upsample the input (i.e. each pixel is upsampled into a $k \times k$ patch).

Remark. A transposed convolution can be interpreted as a normal convolution with stride < 1.

Dilated convolution

Dilated convolution Convolution computed using a kernel that does not consider contiguous pixels.

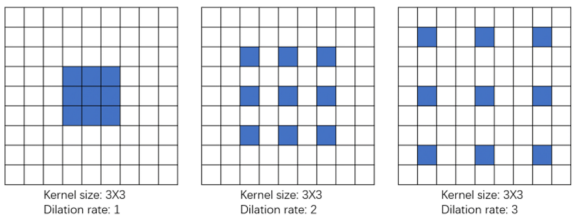


Figure 3.4: Examples of dilated convolutions

Remark. Dilated convolutions allow the enlargement of the receptive field without an excessive number of parameters.

Remark. Dilated convolutions are useful in the first layers when processing high-resolution images (e.g. temporal convolutional networks).

3.8 Normalization layer

A normalization layer has the empirical effects of:

- Stabilizing and possibly speeding up the training phase.
- Increasing the independence of each layer (i.e. maintain a similar magnitude of the weights at each layer).

Batch normalization

Batch normalization Given an input batch X, a batch normalization layer outputs the following:

$$\gamma \frac{X - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \beta$$

where:

- γ and β are learned parameters.
- ε is a small constant.
- μ is the mean and σ^2 is the variance. Depending on when the layer is applied, these values change:

  Training μ and σ^2 are computed from the input batch X.

  Inference μ and σ^2 are computed from the training data. Usually, it is obtained as the moving average of the values computed from the batches during training.