INTRODUCTION TO DATA SCIENCE

Introduction to deep learning - part II

Maciej Świtała, PhD

Autumn 2025

UNIVERSITY OF WARSAW | FACULTY OF ECONOMIC SCIENCES

1. Recurrent neural networks
   - Basic architecture and its extensions
   - Time series example processing
   - Backpropagation through time
   - Text example processing

2. Convolutional neural networks
   - Basic architecture and its extensions
   - Image example processing

Recurrent neural networks

Recurrent neural networks

- **Recurrent neural networks** (RNNs) are a class of neural networks designed **to model sequential data**.
- Traditional feedforward neural networks assume that inputs are independent of one another.
- RNNs have an **internal state (or memory)** that captures **information about previous inputs** in a sequence.
- This makes RNNs particularly effective for tasks involving **time series** and **natural language processing**.

Recurrent neural networks: basic architecture

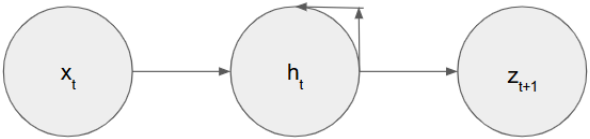
- At each time step t, the RNN processes an input vector x_t and updates its hidden state h_t based on the current input and the previous hidden state h_{t-1}:

$$h_t = \sigma_{tanh}(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

$$z_{t+1} = W_{hz}h_t + b_z$$

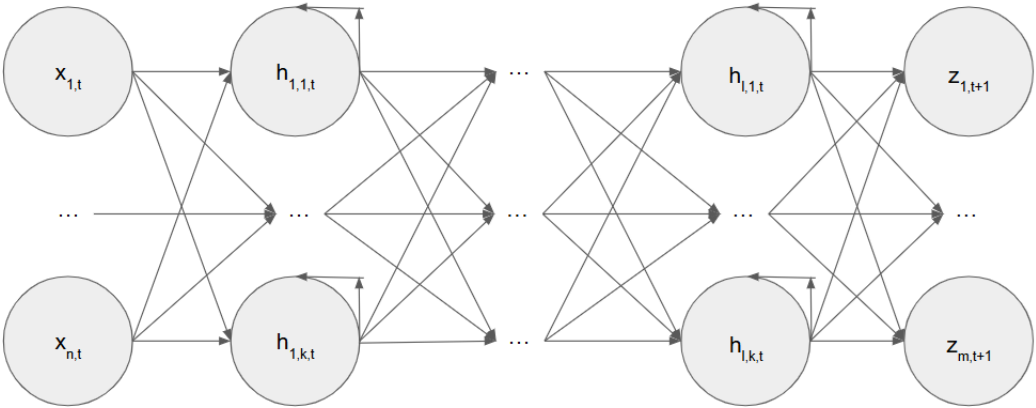
  where:
  - x_t is the input at time step t,
  - h_t is the hidden state at time t,
  - z_{t+1} is the output at time $t+1$,
  - W_{xh}, W_{hh}, W_{hz} are weight matrices,
  - b_h, b_z are bias vectors,
  - σ_{tanh} is the hyperbolic tangent activation function.
- The hidden state h_t acts as a form of memory, carrying information from previous time steps forward through the sequence.

Recurrent neural networks: basic architecture



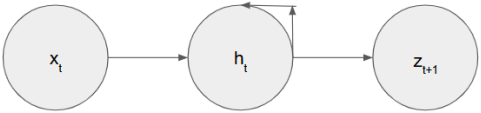
Source: own elaboration.

Recurrent neural networks: deep learning architecture



Source: own elaboration.

Recurrent neural networks: time series example processing



Source: own elaboration.

- Suppose we are given a time series of daily temperatures:

$$x_1 = 15.0, \quad x_2 = 16.0, \quad x_3 = 15.5$$

- Our goal is to predict the next temperature value y_4 using a simple RNN.
- Let us assume:
  - hidden state size: 1 (scalar h_t),
  - input size: 1,
  - weights:

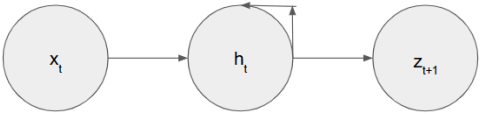
$$W_{xh} = 0.6, \quad W_{hh} = 0.4, \quad W_{hz} = 1.0$$

  - biases:

$$b_h = 0, \quad b_z = 0$$

  - activation function: σ_{tanh},
  - initial hidden state: $h_0 = 0$.

Recurrent neural networks: time series example processing



Source: own elaboration.

1. For input $x_1 = 15.0$:

   $$h_1 = \sigma_{tanh}(W_{xh} \cdot x_1 + W_{hh} \cdot h_0 + b_h) = \sigma_{tanh}(9.0) \approx 1.0$$

2. For input $x_2 = 16.0$:

   $$h_2 = \sigma_{tanh}(W_{xh} \cdot x_2 + W_{hh} \cdot h_1 + b_h) = \sigma_{tanh}(10.0) \approx 1.0$$

3. For input $x_3 = 15.5$:

   $$h_3 = \sigma_{tanh}(W_{xh} \cdot x_3 + W_{hh} \cdot h_2 + b_h) = \sigma_{tanh}(9.7) \approx 1.0$$

4. Output z:

   $$z_4 = W_{hz} \cdot h_3 + b_z = 1.0$$

5. As returns min-max normalised values, i.e., $z_4 = \frac{\hat{y}_4 - y_{min}}{y_{max} - y_{min}}$, the actual prediction $\hat{y}_4$ is:

   $$\hat{y}_4 = 16$$

Recurrent neural networks: training procedure

- RNNs are trained using a variant of backpropagation called **backpropagation through time** (BPTT).
- BPTT is an extension of standard backpropagation, where the **network is unrolled over time**, i.e., **each time step is treated as a separate layer**.
- During training, the loss from each time step is backpropagated through the unrolled structure, **updating shared weights** (e.g., W_{xh}, W_{hh}).

Recurrent neural networks: time series example processing

- Let us follow the already established example, and assume that $y_4 = 15$.
- The loss, let us consider MSE, is then

$$L = (\hat{y}_4 - y_4)^2 = 1$$

- BPTT computes gradients of L with respect to all parameters by **backpropagating through time**, considering how each h_t depends on previous h_{t-1}:

$$\frac{\partial L}{\partial W_{xh}}, \quad \frac{\partial L}{\partial W_{hh}}, \quad \frac{\partial L}{\partial W_{hz}}$$

- These gradients are then used to update the weights via gradient descent.

Recurrent neural networks: time series example processing

- First of all, let us calculate the gradients:

$$\frac{\partial L}{\partial z_4} = \frac{\partial L}{\partial \hat{y}_4} \cdot \frac{\partial \hat{y}_4}{\partial z_4} = 2, \quad \frac{\partial L}{\partial W_{hz}} = \frac{\partial L}{\partial z_4} \cdot \frac{\partial z_4}{\partial W_{hz}} = 2, \quad \frac{\partial L}{\partial h_3} = \frac{\partial L}{\partial z_4} \cdot \frac{\partial z_4}{\partial h_3} = 2$$

- Now, let us *go back in time*:

$$\frac{\partial L}{\partial a_3} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial a_3} \approx 0, \text{ because:}$$

$$\frac{\partial h_3}{\partial a_3} = 1 - \sigma^2_{\tanh}(9.7) \approx 0, \quad \text{where } h_3 = \sigma_{\tanh}(a_3), \quad a_3 = W_{xh}x_3 + W_{hh}h_2 = 9.7, \text{ which implies:}$$

$$\frac{\partial L}{\partial W_{xh}}^{(t=3)} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial a_3} \cdot \frac{\partial a_3}{\partial W_{xh}} = 0, \quad \frac{\partial L}{\partial W_{hh}}^{(t=3)} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial a_3} \cdot \frac{\partial a_3}{\partial W_{hh}} = 0$$

- Similarly:

$$\frac{\partial L}{\partial W_{xh}}^{(t=2)} = 0, \quad \frac{\partial L}{\partial W_{hh}}^{(t=2)} = 0, \quad \frac{\partial L}{\partial W_{xh}}^{(t=1)} = 0, \quad \frac{\partial L}{\partial W_{hh}}^{(t=1)} = 0, \text{ because:}$$

$$\frac{\partial L}{\partial a_2} = \frac{\partial L}{\partial h_2} \cdot \frac{\partial h_2}{\partial a_2} = 0, \quad \text{where } \frac{\partial h_2}{\partial a_2} = 1 - \sigma^2_{\tanh}(10) \approx 0, \quad a_2 = W_{xh}x_2 + W_{hh}h_1 = 10$$

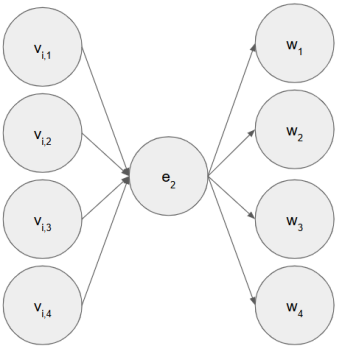
$$\frac{\partial L}{\partial a_1} = \frac{\partial L}{\partial h_1} \cdot \frac{\partial h_1}{\partial a_1} = 0, \quad \text{where } \frac{\partial h_1}{\partial a_1} = 1 - \sigma^2_{\tanh}(9) \approx 0, \quad a_1 = W_{xh}x_1 + W_{hh}h_0 = 9$$

- As a result, assuming learning rate $\eta = 0.1$:

$$W_{hz} := W_{hz} - \eta \cdot \frac{\partial L}{\partial W_{hz}} = 0.8$$

and the other weights remain unchanged.

Recurrent neural networks: text example processing



Source: own elaboration.

- Suppose we consider a sentence *"Hello world! My"*, and we want to predict the next word.
- Firstly, we clean the text, tokenise it, and build a vocabulary - it includes: [*"hello"*, *"world"*, *"my"*, *"name"*].
- Next, we need to obtain **one-hot vectors** for words: *"hello"* $\rightarrow x_1 = [1, 0, 0, 0]$, *"world"* $\rightarrow x_2 = [0, 1, 0, 0]$, *"my"* $\rightarrow x_3 = [0, 0, 1, 0]$.
- Let us assume a hidden state size 1. Then we deal with weights of dimensions, and initialised values, as follows:

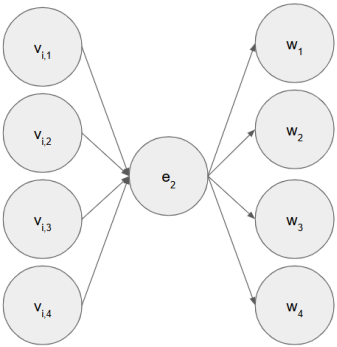
$$W_{xh} = [0.5, -0.4, 0.3, 0.1]$$

$$W_{hh} = [0.2]$$

$$W_{hz} = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \\ 0.4 \end{bmatrix}$$

- Additionally, let us assume: biases $b_h = \vec{0}, \quad b_y = \vec{0}$, activation function σ_{tanh}, and an initial hidden state: $h_0 = 0$.

Recurrent neural networks: text example processing



Source: own elaboration.

1. For *"hello"* $\Rightarrow x_1 = [1, 0, 0, 0]$:

$$a_1 = W_{xh} \cdot x_1 + W_{hh} \cdot h_0 = 0.5,\ h_1 = \sigma_{tanh}(0.5) \approx 0.462$$

2. For *"world"* $\Rightarrow x_2 = [0, 1, 0, 0]$:

$$a_2 = W_{xh} \cdot x_2 + W_{hh} \cdot h_1 \approx -0.308,\ h_2 = \sigma_{tanh}(-0.308) \approx -0.298$$

3. For *"my"* $\Rightarrow x_3 = [0, 0, 1, 0]$:

$$a_3 = W_{xh} \cdot x_3 + W_{hh} \cdot h_2 \approx 0.240,\ h_3 = \sigma_{tanh}(0.240) \approx 0.236$$

4. Now, the output:

$$z = W_{hz} \cdot h_3 \approx [0.024,\ 0.047,\ 0.071,\ 0.094]$$

5. Prediction requires softmax application for getting probabilities:

$$\hat{y} = \sigma_{softmax}(z) = \frac{e^{z_i}}{\sum_j e^{z_j}} = [0.241, 0.247, 0.253, 0.259]$$

Recurrent neural networks: text example processing

- Let us calculate the loss using cross-entropy for the true class $y = [0, 0, 0, 1]$:

$$L = -\sum_{i=1}^{4} y_i \cdot \log(\hat{y}_i) = -\log(\hat{y}_4) = -\log(0.259) \approx 1.350$$

- Let us go through BPTT:

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} = \hat{y} - y = [0.241,\ 0.247,\ 0.253,\ -0.741]$$

- This simplification holds when using the softmax activation together with cross-entropy loss. The Jacobian of the softmax function is:

$$\frac{\partial \hat{y}_i}{\partial z_j} = \begin{cases} \hat{y}_i(1 - \hat{y}_i), & \text{if } i = j \\ -\hat{y}_i \hat{y}_j, & \text{if } i \neq j \end{cases}$$

- Going through the next steps:

$$\frac{\partial L}{\partial W_{hz}} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial W_{hz}} = (\hat{y} - y)^T \cdot h_3 \approx \begin{bmatrix} 0.111 \\ 0.114 \\ 0.117 \\ -0.342 \end{bmatrix}$$

Recurrent neural networks: text example processing

- As a result, there is no gradient flow further into earlier steps:

$$\frac{\partial L}{\partial h_3} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial h_3} \approx 0.0, \quad \frac{\partial L}{\partial a_3} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial a_3} = 0 \Rightarrow$$

$$\frac{\partial L}{\partial W_{xh}}^{(t=3)} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial a_3} \cdot \frac{\partial a_3}{\partial W_{xh}} = 0, \quad \frac{\partial L}{\partial W_{hh}}^{(t=3)} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial a_3} \cdot \frac{\partial a_3}{\partial W_{hh}} = 0$$

- Similarily:

$$\frac{\partial L}{\partial h_2} = 0, \quad \frac{\partial L}{\partial a_2} = 0 \Rightarrow \frac{\partial L}{\partial W_{hh}}^{(t=2)} = 0, \quad \frac{\partial L}{\partial W_{xh}}^{(t=2)} = 0$$

$$\frac{\partial L}{\partial h_1} = 0, \quad \frac{\partial L}{\partial a_1} = 0 \Rightarrow \frac{\partial L}{\partial W_{hh}}^{(t=1)} = 0, \quad \frac{\partial L}{\partial W_{xh}}^{(t=1)} = 0$$

- Final weight update (with learning rate $\eta = 0.1$):

$$W_{hz} := W_{hz} - \eta \cdot \frac{\partial L}{\partial W_{hz}} = \begin{bmatrix} 0.089 \\ 0.189 \\ 0.288 \\ 0.434 \end{bmatrix}$$

and the other weights remain unchanged.

Convolutional neural networks

Convolutional neural networks

- **Convolutional neural networks** (CNNs) are a class of deep neural networks specifically designed to automatically and adaptively learn **spatial hierarchies of features**.
- Unlike basic neural networks, CNNs exploit the spatial locality by:
  - hierarchical feature extraction,
  - shared weights,
  - local connectivity.
- CNNs are widely used in **computer vision**, such as image classification, object detection, and segmentation.

Convolutional neural networks: typical architecture

1. **Input layer**.
2. **Convolutional layer** that applies **learnable filters**, kernels $k \in \mathbb{R}^{n \times n}$, to extract local patterns from input x to return a set of feature maps h:

$$h = x * k + b, \quad h' = \sigma(h)$$

where:

- b is a bias, and $*$ denotes convolution,
- hyperparameters are: filter size, stride (step size of filter movement), padding (adding extra pixels, usually zeros, around the border of the input image), number of filters,
- activation function σ (typically ReLU) is applied element-wise to the feature map.

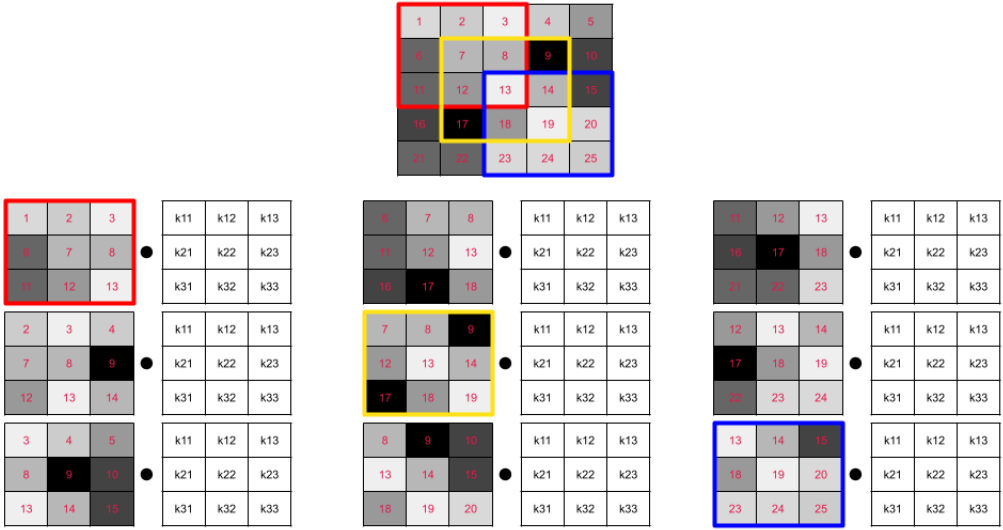
3. **Pooling layer** that applies subsampling or downsampling to reduce spatial dimensions of feature map; common types are: MaxPooling that takes the max value in a region, and AveragePooling that obviously takes the average in a region.

Convolutional neural networks: typical architecture

4. **Dropout layer**, optional but common, that randomly deactivates some neurons to prevent overfitting.
5. **Flatten layer** that convert the pooled feature maps into a 1D vector.
6. **Fully connected layer** (dense layer) that performs classification or regression based on extracted features.
7. **Output layer** that, depending on the task, incorporates softmax, sigmoid, or linear transformation.

Note: backpropagation is used to update weights as in standard neural networks, i.e., gradients are calculated with respect to the loss, and convolutional filters, kernels are updated using gradient descent or other optimisers.

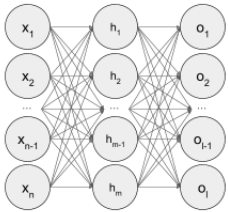
Convolutional neural networks: typical architecture



Source: own elaboration.

Convolutional neural networks: typical architecture

$$x = [\max(\mathrm{ReLU}(\begin{matrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{matrix})), \max(\mathrm{ReLU}(\begin{matrix} c_{12} & c_{13} \\ c_{22} & c_{23} \end{matrix})), \max(\mathrm{ReLU}(\begin{matrix} c_{21} & c_{22} \\ c_{31} & c_{32} \end{matrix})), \max(\mathrm{ReLU}(\begin{matrix} c_{22} & c_{23} \\ c_{32} & c_{33} \end{matrix}))]$$



Source: own elaboration.

Convolutional neural networks - image example processing

- We will consider a grayscale image of size 5×5, processed by one 3×3 convolutional filter, followed by ReLU, and then MaxPooling.
- Let us define input image x, and filter (kernel) k as:

$$x = \begin{bmatrix} 1,0,2,3,0 \\ 4,6,6,8,1 \\ 3,1,1,0,2 \\ 1,2,2,4,1 \\ 2,4,0,1,3 \end{bmatrix}, \quad k = \begin{bmatrix} 1,0,-1 \\ 1,0,-1 \\ 1,0,-1 \end{bmatrix}$$

- Output size H is obtained through:

$$H = \left\lfloor \frac{W-K}{S} + 1 \right\rfloor = \left\lfloor \frac{5-3}{1} + 1 \right\rfloor$$

where X is the input size, K is the kernel size, S is the **stride (i.e., the number of pixels the filter (kernel) moves each time during convolution operation)**; we assume no padding.

Convolutional neural networks - image example processing

- Each cell, starting from $c_{(0,0)}$, i.e., the subset of x (rows: $[0,1,2]$, columns: $[0,1,2]$):

$$c_{(0,0)} = \begin{bmatrix} 1,0,2 \\ 4,6,6 \\ 3,1,1 \end{bmatrix}$$

  is element-wise multiplied with k:

$$k = \begin{bmatrix} 1,0,-1 \\ 1,0,-1 \\ 1,0,-1 \end{bmatrix}$$

  and the result is summed:

$$\sum_{i=0}^{2}\sum_{j=0}^{2} c_{(0,0)}[i,j]\cdot k[i,j] = (1\times1+0\times0+2\times(-1))+(4\times1+6\times0+6\times(-1))+(3\times1+1\times0+1\times(-1)) = -1$$

- Next cells are subsets of x, moving 1 right and 1 down, i.e., $c_{(0,1)}$ includes rows $[0,1,2]$ and columns $[1,2,3]$ of x, $c_{(1,0)}$ includes rows $[1,2,3]$ and columns $[0,1,2]$ of x, etc.

Convolutional neural networks - image example processing

- Output after convolution is:

$$h = \begin{bmatrix} c_{(0,0)}, c_{(0,1)}, c_{(0,2)} \\ c_{(1,0)}, c_{(1,1)}, c_{(1,2)} \\ c_{(2,0)}, c_{(2,1)}, c_{(2,2)} \end{bmatrix} = \begin{bmatrix} -1, -4, 6 \\ -1, -3, 5 \\ 3, 2, -3 \end{bmatrix}$$

- After ReLU activation:

$$h' = \sigma_{ReLU}(h) = \begin{bmatrix} 0, 0, 6 \\ 0, 0, 5 \\ 3, 2, 0 \end{bmatrix}$$

Convolutional neural networks - image example processing

- Let us consider the MaxPooling with stride 1 as the next step. For each 2×2 subset of h' let us calculate its maximum and therefore reduce spatial dimensions of feature maps.

$$h'_{(0,0)} = \begin{bmatrix} 0, 0 \\ 0, 0 \end{bmatrix} \rightarrow max_{(0,0)} = 0, \quad h'_{(0,1)} = \begin{bmatrix} 0, 6 \\ 0, 5 \end{bmatrix} \rightarrow max_{(0,1)} = 6$$

$$h'_{(1,0)} = \begin{bmatrix} 0, 0 \\ 3, 2 \end{bmatrix} \rightarrow max_{(1,0)} = 3, \quad h'_{(1,1)} = \begin{bmatrix} 0, 5 \\ 2, 0 \end{bmatrix} \rightarrow max_{(1,1)} = 5$$

- Final output after MaxPooling:

$$h'' = \begin{bmatrix} max_{(0,0)}, max_{(1,0)} \\ max_{(1,0))}, max_{(1,1)} \end{bmatrix} = \begin{bmatrix} 0, 6 \\ 3, 5 \end{bmatrix}$$

Convolutional neural networks - image example processing

- The pooled data shall be flatten, i.e., converted into 1D vector:

$$h'' = \begin{bmatrix} 0, 6 \\ 3, 5 \end{bmatrix} \rightarrow h''' = [0, 6, 3, 5]$$

- Finally, the vector obtained can be passed into fully connected (dense) layer with sigmoid activation function, let us assume that we have two neurons with weights:

$$W_1 = [0.2, 0.4, 0.1, -0.5], \quad W_2 = [-0.3, 0.6, 0.2, 0.1]$$

and biases:

$$b_1 = 0.1, \quad b_2 = -0.2$$

then:

$$z_1 = W_1 \cdot h''' + b_1 = 0.3, \quad z_2 = W_2 \cdot h''' + b_2 = 4.5$$

- With sigmoid activation function:

$$\hat{y}_1 = \sigma_{softmax}(z_1) \approx 0.015, \quad \hat{y}_2 = \sigma_{sigmoid}(z_2) \approx 0.985$$

Convolutional neural networks - image example processing

- Let us assume that the target is: $y = [0, 1]$ (i.e., we deal with image binary classification). Meanwhile, with forward propagation we obtained $\hat{y} = [0.015, 0.985]$. Eventually, binary cross-entropy loss is:

$$L = -\sum_i y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i) \approx 0.015$$

- As for the backpropagation, we start with:

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} = \approx [0.015, -0.015]$$

- Gradients with respect to fully connected layer:

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial b} \approx [0.015, -0.015]$$

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial z_1} \cdot \frac{\partial z_1}{\partial W_1} \approx [0, 0.089, 0.044], \quad \frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2} \approx [0, -0.089, 0.074]$$

$$\frac{\partial L}{\partial h'''} = \frac{\partial L}{\partial z_1} \cdot \frac{\partial z_1}{\partial h'''} + \frac{\partial L}{\partial z_2} \cdot \frac{\partial z_2}{\partial h'''} \approx [0.007, -0.003, -0.001, -0.009]$$

- Reshape back to maxpooling shape:

$$\frac{\partial L}{\partial h''} = \begin{bmatrix} 0.007, -0.003 \\ -0.001, -0.009 \end{bmatrix}$$

Convolutional neural networks - image example processing

- Original matrix before pooling was:

$$h' = \sigma_{ReLU}(h) = \begin{bmatrix} 0, 0, 6 \\ 0, 0, 5 \\ 3, 2, 0 \end{bmatrix}$$

- We need to place the gradients calculated before at appropriate places (i.e., places of maximum values within 2×2 subsets of h'. As a result, gradient matrix looks like this:

$$\frac{\partial L}{\partial h'} = \begin{bmatrix} 0, 0, -0.003 \\ 0, 0, -0.009 \\ -0.001, 0, 0 \end{bmatrix}$$

- ReLU goes through only where $c > 0$, all the other gradients are made nullified. Therefore:

$$\frac{\partial L}{\partial h_{c>0}} = \frac{\partial L}{\partial h'}$$

$$\frac{\partial L}{\partial h'[0,2]} \approx -0.003, \quad \frac{\partial L}{\partial h'[1,2]} \approx -0.009, \quad \frac{\partial L}{\partial h'[2,0]} \approx -0.001$$

Convolutional neural networks - image example processing

- Each of these gradients is multiplied element-wise with the corresponding 3×3 region of the input image x to compute partial contributions to the kernel gradient:

$$\text{for region h[0,2]} \begin{bmatrix} 2,3,0 \\ 6,8,1 \\ 1,0,2 \end{bmatrix} \cdot (-0.003), \quad \text{for region h[1,2]} \begin{bmatrix} 6,8,1 \\ 1,0,2 \\ 2,4,1 \end{bmatrix} \cdot (-0.009),$$

$$\text{for region h[2,0]} \begin{bmatrix} 3,1,1 \\ 1,2,2 \\ 2,4,0 \end{bmatrix} \cdot (-0.001)$$

- Summing everything all together:

$$\frac{\partial L}{\partial k} \approx \begin{bmatrix} -0.064, -0.081, -0.010 \\ -0.028, -0.030, -0.024 \\ -0.023, -0.041, -0.015 \end{bmatrix}$$

- Gradient descent update rule:

$$k := k - \eta \cdot \frac{\partial L}{\partial k}$$

Thank you for your attention!

Maciej Świtała, PhD
ms.switala@uw.edu.pl