INTRODUCTION TO DATA SCIENCE

Introduction to deep learning - part I

Maciej Świtała, PhD

Autumn 2025

UNIVERSITY OF WARSAW | FACULTY OF ECONOMIC SCIENCES

1. Structure of a neural network
   - Input / hidden / output layers & neurons
   - Weights, bias & activation function

2. Training a neural network
   - Forward propagation & backpropagation
   - Example: perceptron training
   - Example: multi-layer perceptron training

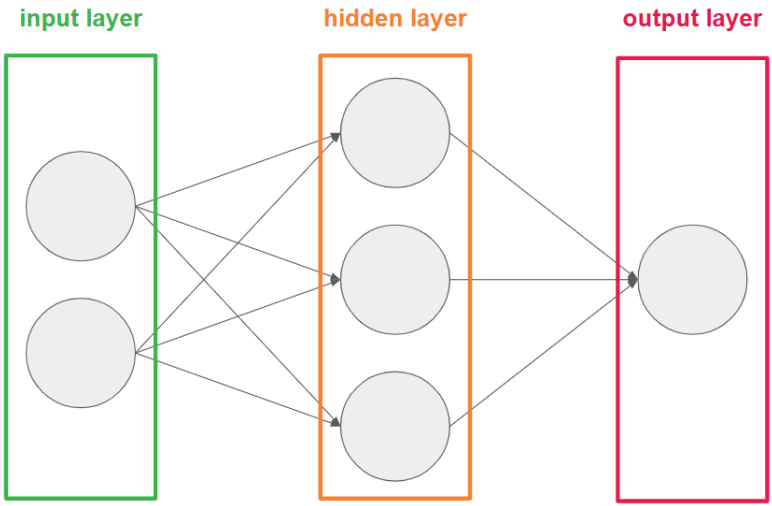
3. Training process improvements
   - Most common loss functions
   - Most common optimisers
   - Regularisation & dropout
   - Batch normalisation
   - Learning rate scheduling & data augmentation

Structure of a neural network

Input / hidden / output layers & neurons

- A **neural network** is a type of machine learning algorithm inspired by the human brain, i.e., it is made up of **layers of interconnected units called neurons**. These neurons work together to process input data and make predictions.
- Basic neural network structure includes:
  - **input layer** - this is where the network receives data; each neuron in this layer represents one feature of the input,
  - **hidden layers** - these are the layers between the input and the output; each neuron in a hidden layer takes inputs from the previous layer, transforms it, and passes the result to the next layer,
  - **output layer** - this layer gives the final result.
- **Deep learning** refers to neural networks that constitute sufficiently deep architectures, i.e., the ones that include *many* layers.

Input / hidden / output layers & neurons



Source: own elaboration.

Neuron operation

1. Each neuron j performs this basic computations:

$$z_j = w_{1j}x_1 + w_{2j}x_2 + ... + w_{nj}x_n + b_j$$

where x_i are inputs, w_i are **weights**, b is a **bias**; weights determine the strength of the conncetion between neurons and bias allows the activation function to shift left or right, increasing flexibility.

2. Then it applies an **activation function**:

$$a_j = \sigma(z_j)$$

where σ aims at adding non-linearity, allowing the network to learn complex patterns.

Most common activation functions

Activation function	**Formula**	**Typical use**
ReLU	$\text{ReLU}(z_j) = \max(0, z_j)$	most common in general
sigmoid	$\text{sigmoid}(z_j) = \frac{1}{1+e^{-z_j}}$	binary classification
tanh	$\tanh(z_j) = \frac{e^{z_j}-e^{-z_j}}{e^{z_j}+e^{-z_j}}$	zero-centered output
softmax	$\text{softmax}(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{n} e^{z_k}}$	multi-class output

Training a neural network

Training a neural network

Simply put, training a neural network means **adjusting its internal weights and biases**. It involves:

1. **Initialisation** - all weights and biases are initialised, usually randomly.
2. **Forward propagation** - input data is passed through the network, layer by layer.
3. **Loss calculation** - the output is compared with true labels using a **loss function** L, e.g., cross-entropy or mean squared error.
4. **Backpropagation** - the network calculates gradients $\frac{\partial L}{\partial w_{ij}}$, $\frac{\partial L}{\partial b_j}$, i.e., how much each weight w_{ij} contributes to the error made L; these gradients are computed using a chain rule of calculus, i.e., **layer by layer in reverse**.
5. **Weights update** (gradient descent) - the network updates its weights to reduce the loss $w_{ij} := w_{ij} - \eta \frac{\partial L}{\partial w_{ij}}$, $b_j := b_j - \eta \frac{\partial L}{\partial b_j}$, where *eta* is a **learning rate**; there are more advanced weights update algorithms called **optimisers**, e.g., Adam, RMSProp, SGD with momentum.

The whole process is repeated many times **epochs**, and the data is often split into **batches** for efficiency.

Neural network performance evaluation

- Let us emphasise that, **after each epoch**, the network is evaluated on **validation data** to check if it is **overfitting**.
- Obviously, **after training**, the neural network is tested on a **test set** to measure its real performance.
- In contrast to non-network machine-learning algorithms, **cross-validation is often neglected** due to its high computational cost. It is **just train-validation-test split**.

Example: perceptron training

- For prediction, let us use a **perceptron**, i.e., the simplest type of neural network, fundamental building block for more complex ones.
- Perceptron is a **binary classifier** with just an input layer and output layer (**no hidden layers**).
- Perceptron uses a **step function** as an activation function. It guarantees binary output, i.e.,

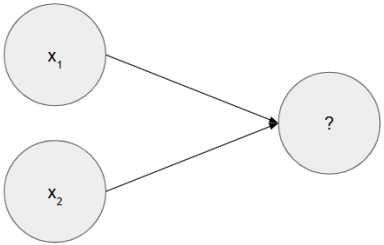
$$a = \sigma(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if otherwise} \end{cases}$$

- Perceptron does not consider weights update with loss function gradients, instead it uses **heuristic rules for updating the weights**:

$$w_i := w_i + \eta(y - \hat{y})x_i$$

$$b := b + \eta(y - \hat{y})$$

Example: perceptron training



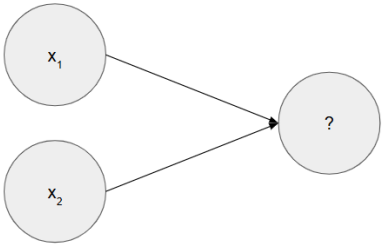
Source: own elaboration.

- Let us consider example training data:

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

- As for the weights and bias, let us start with $w_1 = 0$, $w_2 = 0$, b, $\eta = 0.1$.
- In classic perceptron there is **1 batch**, i.e., weights are updated after each observation.
- Let us consider **as many epochs as needed to converge**, i.e., the whole sample goes through the perceptron once.

Example: perceptron training



Source: own elaboration.

- Epoch 1, iteration 1: $x_1 = 0$, $x_2 = 0$, $y = 0$.

$$z = w_1 x_1 + w_2 x_2 + b = 0 \cdot 0 + 0 \cdot +0 = 0$$

$$\hat{y} = \sigma(z) = 0$$

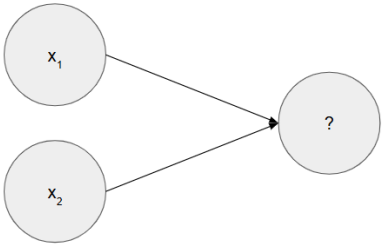
$$w_1 := w_1 + \eta(y - \hat{y})x_1 = 0 + 0.1 \cdot 0 \cdot 0 = 0$$

$$w_2 := w_2 + \eta(y - \hat{y})x_2 = 0 + 0.1 \cdot 0 \cdot 0 = 0$$

$$b := b + \eta(y - \hat{y}) = 0 + 0.1 \cdot 0 = 0$$

- No update in w_1, w_2, b.
- Weights at the moment: $w_1 = 0$, $w_2 = 0$, $b = 0$.

Example: perceptron training



Source: own elaboration.

- Epoch 1, iteration 2: $x_1 = 0$, $x_2 = 1$, $y = 0$.

$$z = w_1 x_1 + w_2 x_2 + b = 0 \cdot 0 + 0 \cdot 1 + 0 = 0$$

$$\hat{y} = \sigma(z) = 0$$

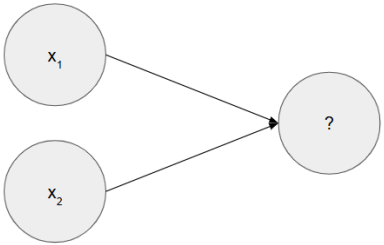
$$w_1 := w_1 + \eta(y - \hat{y})x_1 = 0 + 0.1 \cdot 0 \cdot 0 = 0$$

$$w_2 := w_2 + \eta(y - \hat{y})x_2 = 0 + 0.1 \cdot 0 \cdot 1 = 0$$

$$b := b + \eta(y - \hat{y}) = 0 + 0.1 \cdot 0 = 0$$

- No update in w_1, w_2, b.
- Weights at the moment: $w_1 = 0$, $w_2 = 0$, $b = 0$.

Example: perceptron training



Source: own elaboration.

- Epoch 1, iteration 3: $x_1 = 1$, $x_2 = 0$, $y = 0$.

$$z = w_1 x_1 + w_2 x_2 + b = 0 \cdot 1 + 0 \cdot 0 + 0 = 0$$

$$\hat{y} = \sigma(z) = 0$$

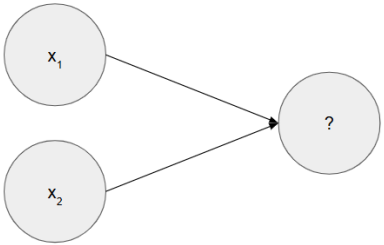
$$w_1 := w_1 + \eta(y - \hat{y})x_1 = 0 + 0.1 \cdot 0 \cdot 1 = 0$$

$$w_2 := w_2 + \eta(y - \hat{y})x_2 = 0 + 0.1 \cdot 0 \cdot 0 = 0$$

$$b := b + \eta(y - \hat{y}) = 0 + 0.1 \cdot 0 = 0$$

- No update in w_1, w_2, b.
- Weights at the moment: $w_1 = 0$, $w_2 = 0$, $b = 0$.

Example: perceptron training



Source: own elaboration.

- Epoch 1, iteration 4: $x_1 = 1$, $x_2 = 1$, $y = 1$.

$$z = w_1 x_1 + w_2 x_2 + b = 0 \cdot 1 + 0 \cdot 1 + 0 = 0$$

$$\hat{y} = \sigma(z) = 0$$

$$w_1 := w_1 + \eta(y - \hat{y}) x_1 = 0 + 0.1 \cdot 1 \cdot 1 = 0.1$$

$$w_2 := w_2 + \eta(y - \hat{y}) x_2 = 0 + 0.1 \cdot 1 \cdot 1 = 0.1$$

$$b := b + \eta(y - \hat{y}) = 0 + 0.1 \cdot 1 = 0.1$$

- Update in w_1, w_2, b.
- Weights after epoch = 1: $w_1 = 0.1$, $w_2 = 0.1$, $b = 0.1$.

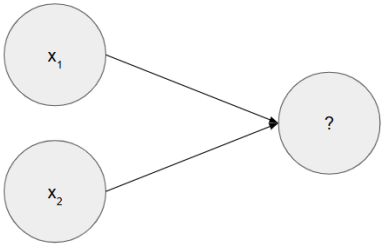
Example: perceptron training

epoch	(x_1, x_2)	y	$z = w_1x_1 + w_2x_2 + b$	$\hat{y}$	$y - \hat{y}$	weights (w_1, w_2)	bias b
1	(0,0)	0	0.0	0	0	(0.0, 0.0)	0.0
1	(0,1)	0	0.0	0	0	(0.0, 0.0)	0.0
1	(1,0)	0	0.0	0	0	(0.0, 0.0)	0.0
1	(1,1)	1	0.0	0	+1	(0.1, 0.1)	0.1
2	(0,0)	0	0.1	1	-1	(0.1, 0.1)	0.0
2	(0,1)	0	0.1	1	-1	(0.1, 0.0)	-0.1
2	(1,0)	0	0.0	0	0	(0.1, 0.0)	-0.1
2	(1,1)	1	0.0	0	+1	(0.2, 0.1)	0.0
3	(0,0)	0	0.0	0	0	(0.2, 0.1)	0.0
3	(0,1)	0	0.1	1	-1	(0.2, 0.0)	-0.1
3	(1,0)	0	0.1	1	-1	(0.1, 0.0)	-0.2
3	(1,1)	1	0.0	0	+1	(0.2, 0.1)	-0.1

Example: perceptron training

epoch	(x_1, x_2)	y	$z = w_1 x_1 + w_2 x_2 + b$	$\hat{y}$	$y - \hat{y}$	weights (w_1, w_2)	bias b
4	(0,0)	0	-0.1	0	0	(0.2, 0.1)	-0.1
4	(0,1)	0	0.0	0	0	(0.2, 0.1)	-0.1
4	(1,0)	0	0.1	1	-1	(0.1, 0.1)	-0.2
4	(1,1)	1	0.0	0	+1	(0.2, 0.2)	-0.1
5	(0,0)	0	-0.1	0	0	(0.2, 0.2)	-0.1
5	(0,1)	0	0.1	1	-1	(0.2, 0.1)	-0.2
5	(1,0)	0	0.0	0	0	(0.2, 0.1)	-0.2
5	(1,1)	1	0.1	1	0	(0.2, 0.1)	-0.2
6	(0,0)	0	-0.2	0	0	(0.2, 0.1)	-0.2
6	(0,1)	0	-0.1	0	0	(0.2, 0.1)	-0.2
6	(1,0)	0	0.0	0	0	(0.2, 0.1)	-0.2
6	(1,1)	1	0.1	1	0	(0.2, 0.1)	-0.2

Example: perceptron training



Source: own elaboration.

Final perceptron form:

$$z = w_1 x_1 + w_2 x_2 + b$$

$$\hat{y} = \sigma(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if otherwise} \end{cases}$$

where:

$$w_1 = 0.2$$

$$w_2 = 0.1$$

$$b = -0.2$$

Example: multi-layer perceptron training

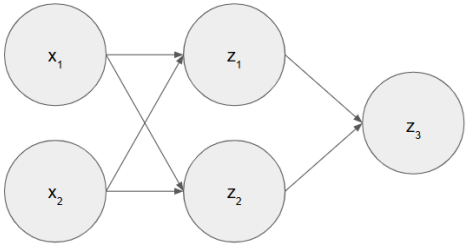
- Let us consider an extension, slowly crawling towards complex neural networks, i.e., **multi-layer perceptron** (MLP).
- MLP in this example is a **binary classifier** with an input layer, **one hidden layer** including **two neurons with ReLU activation function**, i.e., $a = \sigma(z) = \max(0, z)$, and an output layer with **sigmoid activation function**, i.e., $a = \sigma(z) = \frac{1}{1+e^{-z}}$.
- The weights shall be updated with **loss function gradients**:

$$w_{ij} := w_{ij} - \eta \frac{\partial L}{\partial w_{ij}}$$

$$b_j := b_j - \eta \frac{\partial L}{\partial b_j}$$

- Let us also assume that the loss function takes the form of **binary cross-entropy**, i.e., $L = -[y \cdot log(\hat{y}) + (1 - y) \cdot log(1 - \hat{y})]$.

Example: multi-layer perceptron training



Source: own elaboration.

- Let us consider example training data again:

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

- As for the weights and bias, let us start with:
  - $w_{11} = 0$, $w_{12} = 0$, $b_1 = 0$ (first hidden layer),
  - $w_{21} = 0$, $w_{22} = 0$, $b_2 = 0$ (second hidden layer),
  - $w_{31} = 0$, $w_{32} = 0$, $b_3 = 0$ (output layer),
  - $\eta = 0.1$.
- Again, let us consider **1 batch**, and **as many epochs as needed to converge**.

Example: multi-layer perceptron training

- Epoch 1, iteration 1: $x_1 = 0$, $x_2 = 0$, $y = 0$.
- Forward pass:

$$z_1 = w_{11}x_1 + w_{12}x_2 + b_1 = 0 \cdot 0 + 0 \cdot 0 + 0 = 0,\ a_1 = \sigma_{ReLU}(z_1) = 0$$

$$z_2 = w_{21}x_1 + w_{22}x_2 + b_2 = 0 \cdot 0 + 0 \cdot 0 + 0 = 0,\ a_2 = \sigma_{ReLU}(z_2) = 0$$

$$z_3 = w_{31}a_1 + w_{32}a_2 + b_3 = 0 \cdot 0 + 0 \cdot 0 + 0 = 0,\ \hat{y} = \sigma_{sigmoid}(z_3) = \frac{1}{1 + e^{-z_3}} = 0.5$$

- Loss calculation:

$$L = -[y \cdot log(\hat{y}) + (1 - y) \cdot log(1 - \hat{y})] \approx 0.6$$

- Weights update:

$$w_{31} := w_{31} - \eta \frac{\partial L}{\partial w_{31}} = w_{31} - \eta \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_3} \cdot \frac{\partial z_3}{\partial w_{31}} = 0$$

$$w_{32} := w_{32} - \eta \frac{\partial L}{\partial w_{32}} = w_{32} - \eta \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_3} \cdot \frac{\partial z_3}{\partial w_{32}} = 0$$

$$b_3 := b_3 - \eta \frac{\partial L}{\partial b_3} = b_3 - \eta \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_3} \cdot \frac{\partial z_3}{\partial b_3} = -0.05$$

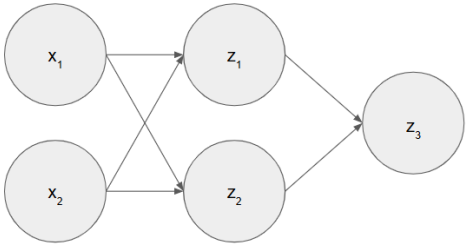
$$\text{ReLU'} = 0 \Rightarrow \frac{\partial L}{\partial w_{11}} = 0,\ \frac{\partial L}{\partial w_{12}} = 0,\ \frac{\partial L}{\partial b_1} = 0,\ \frac{\partial L}{\partial w_{21}} = 0,\ \frac{\partial L}{\partial w_{22}} = 0,\ \frac{\partial L}{\partial b_2} = 0$$

- Update in $b_3 = -0.05$.
- Weights at the moment: $w_{11} = 0$, $w_{12} = 0$, $b_1 = 0$, $w_{21} = 0$, $w_{22} = 0$, $b_2 = 0$, $w_{31} = 0$, $w_{32} = 0$, b_3=-0.05.

Example: multi-layer perceptron training

epoch	(x_1, x_2)	y	z_3	$\hat{y}$	loss	w_{11}	w_{12}	b_1	w_{21}	w_{22}	b_2	w_{31}	w_{32}	b_3
1	(0,0)	0	0.000	0.500	0.693	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	-0.050
1	(0,1)	0	-0.050	0.488	0.670	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	-0.098
1	(1,0)	0	-0.098	0.476	0.644	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	-0.146
1	(1,1)	1	-0.146	0.464	0.768	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	-0.199
2	(0,0)	0	-0.199	0.451	0.598	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	-0.244
2	(0,1)	0	-0.244	0.439	0.577	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	-0.287
2	(1,0)	0	-0.287	0.428	0.563	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	-0.330
2	(1,1)	1	-0.330	0.418	0.873	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	-0.372
...														
100	(0,0)	0	-1.094	0.251	0.287	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	-1.094
100	(0,1)	0	-1.094	0.251	0.287	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	-1.094
100	(1,0)	0	-1.094	0.251	0.287	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	-1.094
100	(1,1)	1	-1.094	0.251	1.383	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	-1.051

Example: multi-layer perceptron training



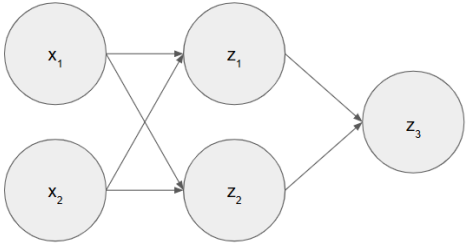
Source: own elaboration.

- In this example, MLP does not converge (at least within first 1,000 epochs).
- All starting weights are zeros, and $\text{ReLU}(0) = 0$, therefore $a_1 = 0$, $a_2 = 0$, and their gradients when backpropagated are also 0s.
- Eventually, there is **no opportunity to learn as the gradients stop at output layer**.
- Solution is **setting the starting values to non-zeros** (see: next slide).
- Another possible solution could be avoiding ReLU that makes the gradients 0s.

Example: multi-layer perceptron training

epoch	(x_1, x_2)	y	z_3	$\hat{y}$	loss	w_{11}	w_{12}	b_1	w_{21}	w_{22}	b_2	w_{31}	w_{32}	b_3
1	(0,0)	0	0.050	0.510	0.260	0.120	-0.100	0.030	0.150	0.070	-0.040	0.200	-0.120	0.010
1	(0,1)	1	0.100	0.530	0.220	0.120	-0.100	0.030	0.150	0.070	-0.040	0.200	-0.120	0.010
1	(1,0)	1	0.080	0.520	0.230	0.120	-0.100	0.030	0.150	0.070	-0.040	0.200	-0.120	0.010
1	(1,1)	0	0.030	0.510	0.260	0.120	-0.100	0.030	0.150	0.070	-0.040	0.200	-0.120	0.010
2	(0,0)	0	-0.020	0.490	0.240	0.140	-0.110	0.020	0.170	0.080	-0.050	0.220	-0.110	0.000
2	(0,1)	1	0.200	0.550	0.200	0.140	-0.110	0.020	0.170	0.080	-0.050	0.220	-0.110	0.000
2	(1,0)	1	0.180	0.540	0.210	0.140	-0.110	0.020	0.170	0.080	-0.050	0.220	-0.110	0.000
2	(1,1)	0	0.050	0.510	0.250	0.140	-0.110	0.020	0.170	0.080	-0.050	0.220	-0.110	0.000
...														
1000	(0,0)	0	-6.200	0.002	0.000	0.550	0.150	-0.200	0.570	0.280	-0.400	0.730	0.100	-0.300
1000	(0,1)	1	7.100	0.999	0.000	0.550	0.150	-0.200	0.570	0.280	-0.400	0.730	0.100	-0.300
1000	(1,0)	1	7.050	0.999	0.000	0.550	0.150	-0.200	0.570	0.280	-0.400	0.730	0.100	-0.300
1000	(1,1)	0	-0.200	0.450	0.300	0.550	0.150	-0.200	0.570	0.280	-0.400	0.730	0.100	-0.300

Example: multi-layer perceptron training



Source: own elaboration.

Final MLP form:

$$z_1 = 0.550 \cdot x_1 + 0.150 \cdot x_2 - 0.200$$

$$a_1 = \sigma_{ReLU}(z_1)$$

$$z_2 = 0.570 \cdot x_1 + 0.280 \cdot x_2 - 0.400$$

$$a_2 = \sigma_{ReLU}(z_2)$$

$$z_3 = 0.200 \cdot a_1 + 0.100 \cdot a_2 - 0.300$$

$$\hat{y} = \sigma_{sigmoid}(z_3)$$

Training process improvements

Most common loss functions

Loss function	**Formula**	**Typical use**
Mean Squared Error	$L = \frac{1}{n}\sum_{j=1}^{n}(y_j - \hat{y}_j)^2$	regression
Mean Absolute Error	$L = \frac{1}{n}\sum_{j=1}^{n}\lvert y_j - \hat{y}_j\rvert$	regression
Huber loss	$L = \begin{cases} \frac{1}{2}(y_j - \hat{y}_j)^2 & \text{if } \lvert y_j - \hat{y}_j\rvert \leq \delta \\ \delta\left(\lvert y_j - \hat{y}_j\rvert - \frac{1}{2}\delta\right) & \text{if } \lvert y_j - \hat{y}_j\rvert > \delta \end{cases}$	regression
Binary cross-entropy (log loss)	$L = -\frac{1}{n}\sum_{j=1}^{n}[y_j log(\hat{y}_j) + (1-y_j) log(1-\hat{y}_j)]$	binary classification
Categorical cross-entropy	$L = -\sum_{j=1}^{k} y_j log(\hat{y}_j)$	multi-categorical classification

Most common optimisers

- As mentioned before, the network **updates its weights** to reduce the loss, prominently **using gradients**:

$$w_{ij} := w_{ij} - \eta \frac{\partial L}{\partial w_{ij}}$$

$$b_j := b_j - \eta \frac{\partial L}{\partial b_j}$$

- Let us briefly consider the more advanced weights update algorithms, i.e., **optimisers**. The most common are:
  1. SGD with momentum,
  2. RMSProp,
  3. Adam.

Optimisers - SGD with momentum

- Stochastic Gradient Descent (SGD) with momentum **adds a velocity term** that **accumulates past gradients to accelerate learning** in consistent directions and dampen oscillations.
- Momentum helps the model build up speed in directions of consistent gradient, allowing faster convergence and less oscillation.
- It takes a following form:

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

where:

- θ_t are parameters at time step t,
- v_t is a velocity term,
- η is a learning rate,
- γ is a momentum coefficient (typically 0.9),
- $\nabla_\theta L(\theta_t)$ is a gradient of the loss function.

Optimisers - RMSProp

- RMSProp is an adaptive learning rate method that **divides the gradient by a running average of its recent magnitudes**.
- RMSProp scales the learning rate per parameter, helping with **training stability** especially in non-stationary settings.
- It takes a following form:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{s_t + \epsilon}} \nabla_\theta L(\theta_t)$$

$$s_t = \rho s_{t-1} + (1 - \rho) \left(\nabla_\theta L(\theta_t)\right)^2$$

where again:

- θ_t are parameters at time step t,
- η is a learning rate,
- $\nabla_\theta L(\theta_t)$ is a gradient of the loss function,

and additionally:

- s_t is a running average of squared gradients,
- ρ is a decay factor (typically 0.9),
- ϵ is a small constant to prevent division by zero.

Optimisers - Adam

- Adaptive Moment Estimation (Adam) **combines ideas from momentum and RMSProp** optimisers.
- It computes **adaptive learning rates** using estimates of **first and second moments of the gradients**.
- Adam optimiser takes a following form:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla_\theta L(\theta_t)$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\left(\nabla_\theta L(\theta_t)\right)^2$$

with weights update made in a following way:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

where:

- m_t is the first moment (mean of gradients),
- v_t is the second moment (uncentered gradients variance),
- β_1, β_2 are exponential decay rates (e.g., 0.9 and 0.999),
- ϵ is a small constant to ensure numerical stability.

Regularisation

- Regularization techniques are used **to prevent overfitting**, which occurs when a model learns the noise in the training data instead of the underlying patterns.
- **L2 regularisation** adds a penalty to the loss function based on the magnitude of the weights:

$$L_{total} = L_{original} + \lambda \sum_i \theta_i^2$$

  eventually encouraging smaller weights.
- **L1 regularisation** uses a penalty based on the absolute value:

$$L_{total} = L_{original} + \lambda \sum_i |\theta_i|$$

  which promotes sparsity, i.e., some weights become exactly zero, which facilitates feature selection.
- Note: λ is obviously the regularisation parameter.

Dropout

- Dropout is a **stochastic regularization technique**.
- When using dropout, **during training, individual neurons are randomly "dropped" (i.e., set to 0)** with probability p.
- At test, all neurons are used but **scaled** to account for the dropped ones.
- The output obtained from each neuron is also **scaled** (divided by $1 - p$) to ensure that the expected output stays unchanged.
- Intuitively, dropout makes a network acting **like training ensemble of sub-networks**.

Batch normalisation

- Batch normalisation is a technique to **normalise the inputs of each layer**, reducing internal covariate shift, i.e., the problem where layers change during training.
- More formally, for batch $B = \{x_1, x_2, ..., x_m\}$ it requires computing:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i, \quad \sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \quad y_i = \gamma\hat{x}_i + \beta$$

  where γ and β are **learnable parameters** that restore representational power, and ϵ is a small constant for numerical stability.
- Batch normalisation **allows for higher learning rates, reduces sensitivity to initialisation, and acts as a regulariser**, i.e., often reduces the need for dropout.

Learning rate scheduling

Instead of using a fixed **learning rate** η, it is often beneficial to **change it over time**. The most common learning rate scheduling types are:

1. **step decay**:

$$\eta_t = \eta_0 \cdot \gamma^{\left\lfloor \frac{t}{k} \right\rfloor}$$

   where η_0 is an initial learning rate, t is a current epoch, γ is decay factor (e.g., 0.1), k is step size in epoch.

2. **exponential decay**:

$$\eta_t = \eta_0 \cdot e^{-kt}$$

   where again η_0 is an initial learning rate, t is a current epoch, k is step size in epoch, and additionally T is a total number of epochs or cycles.

Learning rate scheduling

3. **cosine annealing**:

$$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})\left(1 + \cos\left(\frac{t\pi}{T}\right)\right)$$

where obviously η_{min} and η_{max} denote minimum and maximum learning rate respectively, t is a current epoch, and T is a total number of epochs or cycles.

4. **cyclical learning rate** (CLR):

$$\eta_t = \eta_{\min} + (\eta_{\max} - \eta_{\min}) \cdot \text{triangular}(t)$$

where again η_{min} and η_{max} denote minimum and maximum learning rate respectively, t is a current epoch, and triangular(t) is a function that cyclically increases and decreases between 0 and 1; however there are different possible variants of triangular(t).

Data augmentation

- Augmentation artificially expands the training dataset by **applying random transformations**.
- It helps prevent overfitting by improving generalization.
- Common data augmentation techniques **for images** are: random crop, horizontal or vertical flip, rotation, scaling, color jitter, Gaussian noise.
- Common data augmentation techniques **for text** are: synonym replacement, random insertion or deletion.
- Common data augmentation techniques **for audio** are: time shifting, noise injection, pitch alteration.
- Together, learning rate schedules and data augmentation help the model **address the overfitting problem**.

Thank you for your attention!

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