Neural Networks 1 - Single-layer and multilayer neural networks

18NES1 - Lecture 5, Summer semester 2024/25

Zuzana Petříčková

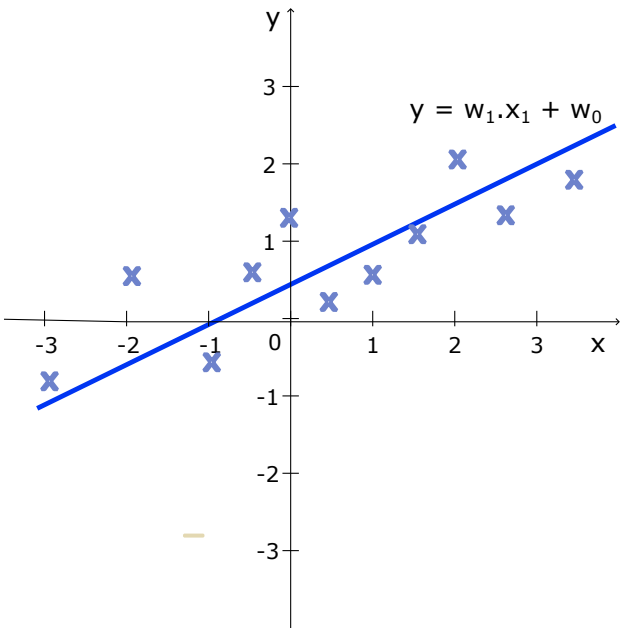
March 25, 2025

What We Covered Last Time

1. **Most common continuous activation functions for neurons**

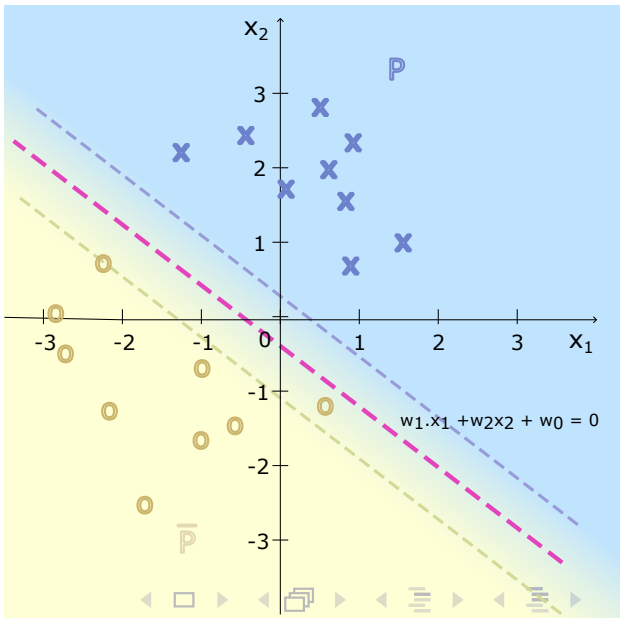
- Linear (identity) function

Task: Linear regression



- Sigmoid, hyperbolic tangent

Task: Linear classification



What We Covered Last Time

2. **Gradient-based learning for l neurons with continuous and differentiable activation functions**
   1. Initialize weights with small random real values:
      $\vec{w}(0) = (w_0, w_1, ..., w_n)^T$
      Initialize the learning rate: α_0 where $1 > \alpha_0 > 0$
   2. Present the next training sample $(\vec{x}_t, d_t)$ and compute the neuron's potential and actual output:
      $\xi_t = \vec{x}_t \vec{w}$
      $y_t = f(\xi_t)$
   3. Update the weights (in the opposite direction of the error gradient):

$$\vec{w}(t+1) = \vec{w}(t) - \alpha \frac{\partial E_p}{\partial w_i} = \vec{w}(t) + \alpha_t f'(\xi_t)(d_t - y_t)\vec{x}_t^T$$

      (for the SSE loss function)
   4. Optionally update the learning rate: $\alpha_t \rightarrow \alpha_{t+1}$
   5. If not finished, go back to step 2.

What We Covered Last Time

2. **Gradient-based learning for general neurons with continuous and differentiable activation functions**

Sum of Squared Errors (SSE) loss function:

$$E(\vec{w}) = \frac{1}{2}\sum_{p=1}^{N}(d_p - y_p)^2 = \frac{1}{2}\sum_{p=1}^{N}\left(d_p - f\left(\sum_{i=0}^{n} w_i x_{pi}\right)\right)^2 = \sum_{p=1}^{N} E_p(\vec{w})$$

Partial derivative:

$$\frac{\partial E_p}{\partial w_i} = \frac{\partial E_p}{\partial y_p}\cdot\frac{\partial y_p}{\partial \xi_p}\cdot\frac{\partial \xi_p}{\partial w_i} = -(d_p - y_p)\cdot f'(\xi_p)\cdot x_{pi}$$

Weight update rule (after presenting the p-th sample at time t):

$$w_i(t+1) = w_i(t) - \alpha\frac{\partial E_p}{\partial w_i} = w_i(t) + \alpha f'(\xi_p)(d_p - y_p)x_{pi}$$

In vector form:

$$\vec{w}(t+1) = \vec{w}(t) - \alpha\nabla E_p(\vec{w}) = \vec{w}(t) + \alpha(d_p - y_p)f'(\xi_p)\vec{x}_p^T$$

What We Covered Last Time

Cross-Entropy Loss Function

- Particularly suitable for linear classification combined with sigmoid or tanh activation.
- Strongly penalizes low probabilities for the correct class.
- For the sigmoid function, for example:

$$E = -\sum_p \left(d_p \log y_p + (1 - d_p) \log(1 - y_p)\right)$$

- The gradient with respect to the output is:
  $\frac{\partial E_p}{\partial y_p} = -\frac{d_p}{y_p} + \frac{(1-d_p)}{1-y_p}$
- Substituting into the weight update rule leads to beneficial simplification:

$$\frac{\partial E_p}{\partial w_i} = \frac{\partial E_p}{\partial y_p} \cdot \frac{\partial y_p}{\partial \xi_p} \cdot \frac{\partial \xi_p}{\partial w_i} = \left(\frac{1 - d_p}{1 - y_p} - \frac{d_p}{y_p}\right) y_p(1-y_p)x_{pi} = (y_p - d_p)x_{pi}$$

$\rightarrow$ More efficient learning and reduced risk of neuron saturation

What We Covered Last Time

3. **Preprocessing of Categorical Data**

Ordinal (Label) Encoding

- Used for ordered categories (e.g., *low, medium, high*) or for binary categories (e.g., *left, right*), which are converted to numbers and optionally normalized:
  - *Low* = -1, *Medium* = 0, *High* = 1
  - *Left* = -1, *Right* = 1

One-Hot Encoding

- Used for nominal (unordered) categories such as car color or animal type. Converted to a binary representation (one feature becomes as many columns as there are categories), and optionally normalized:
  - *Red*$\rightarrow [1, -1, -1]$, *Blue*$\rightarrow [-1, 1, -1]$, *Green*$\rightarrow [-1, -1, 1]$

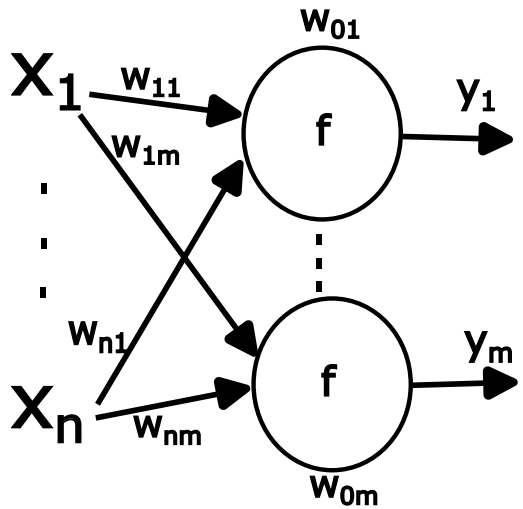
Embeddings

- Used for capturing complex relationships between categorical values, such as in word or sentence representations in natural language

Today's Class

1. Neural network consisting of a single layer of neurons
2. Neural network with multiple layers of neurons – Introduction

Neural Network with a Single Layer of Neurons



1. Neurons with a linear activation function
   → **Multivariate Linear Regression**
   - Linear neural network
2. Neurons with logistic or tanh activation function
   → **Multi-class Linear Classification (Pattern Recognition Task)**
   - Single-layer perceptron

Neural Network with a Single Layer of Neurons

Motivating Example: Multi-class Classification and Categorical Features

Size	Fur	Talks?	Class
Small	Short	No	Cat
Large	Long	No	Dog
Small	None	Yes	Parrot
Medium	Short	No	Cat
...			...

Size	Fur	Talks?	Cat	Dog	Parrot	Carp
-1	0	-1	1	-1	-1	-1
1	1	-1	-1	1	-1	-1
-1	-1	1	-1	-1	1	-1
0	0	-1	1	-1	-1	-1
...	...	...	...		...	

Example notebook: categorical_values.ipynb

Neural Network with a Single Layer of Neurons

Example: Multi-Class Classification with Categorical Features

Size	Fur	Speaks?	Cat	Dog	Parrot	Carp
-1	0	-1	1	-1	-1	-1
1	1	-1	-1	1	-1	-1
-1	-1	1	-1	-1	1	-1
0	0	-1	1	-1	-1	-1
...	...	...	...	...	...	...

How do we train the model?

1. Train a separate neuron for each category (one at a time) and merge the results...

Size	Fur	Speaks?	Cat
-1	0	-1	1
0	1	-1	-1
-1	-1	1	-1
0	0	-1	1
...			...

Neural Network with a Single Layer of Neurons

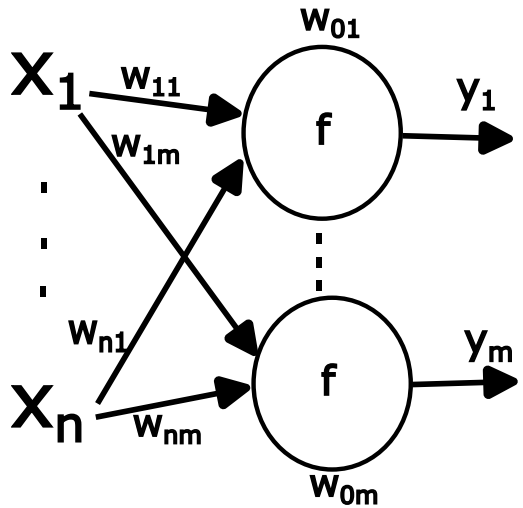
Example: Multi-Class Classification with Categorical Features

Size	Fur	Speaks?	Cat	Dog	Parrot	Carp
-1	0	-1	1	-1	-1	-1
1	1	-1	-1	1	-1	-1
-1	-1	1	-1	-1	1	-1
0	0	-1	1	-1	-1	-1
...	...	...	...	...	...	...

How do we train the model?

2. **Better approach:** Construct a neural network with a single layer of neurons and train it all at once.

Neural Network with a Single Layer of Neurons

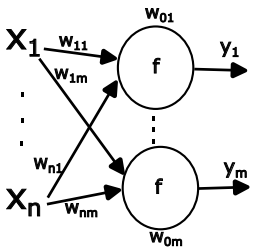


- The model is represented by the weight matrix:

$$W = \begin{pmatrix} w_{01} & w_{02} & \dots & w_{0m} \\ \dots & \dots & & \dots \\ w_{n1} & w_{n2} & \dots & w_{nm} \end{pmatrix}$$

  - Each neuron corresponds to one column of the matrix.
  - The first row corresponds to biases (alternativelly, bias vector can be represented separately).

Neural Network with a Single Layer of Neurons



$$W = \begin{pmatrix} w_{01} & w_{02} & \dots & w_{0m} \\ \dots & \dots & & \dots \\ w_{n1} & w_{n2} & \dots & w_{nm} \end{pmatrix}$$

- If individual neurons have an activation function $f : R \to R$, we define: $f(\vec{\xi}) = (f(\xi_1), ..., f(\xi_N))^T$
- Model output:

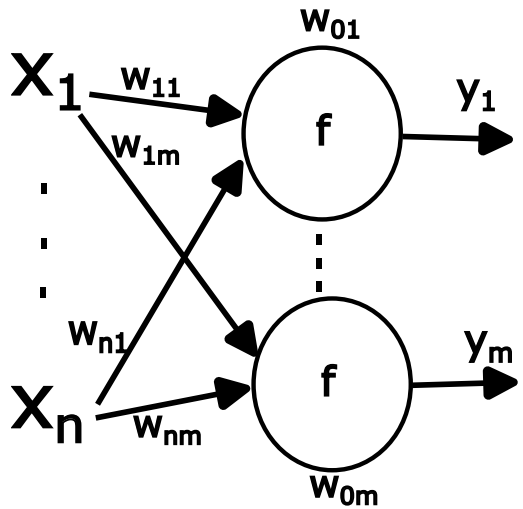
$$\vec{y} = f(\vec{\xi}) = f(\vec{x}W)$$

- Training set format: $T = (X, D)$

$x_{10} = 1$	x_{11}	...	x_{1n}	d_{11}	...	d_{1m}
...	...		...	...		...
$x_{N0} = 1$	x_{N1}	...	x_{Nn}	d_{N1}	...	d_{Nm}

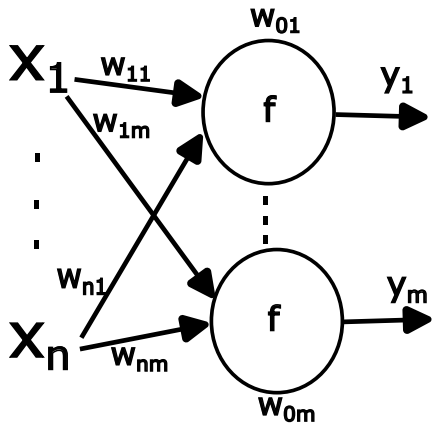
$$Y = f(\Xi) = f(XW)$$

Neural Network with a Single Layer of Neurons



1. Neurons with a linear activation function
   → **Multidimensional Linear Regression**
   - Linear neural network
2. Neurons with logistic or tanh activation function
   → **Linear classification into multiple classes (pattern recognition task)**
   - Single-layer perceptron

Linear Neural Network



- Composed of a single layer of linear neurons (adding more layers would provide no additional benefit - worth considering).
- Multidimensional linear regression.
- Model output:

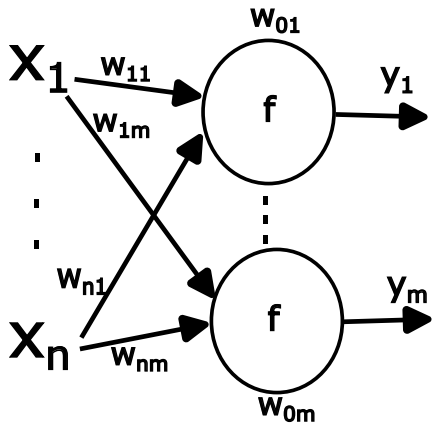
$$Y = XW$$

Training using the Least Squares (LSQ) Method

$$W = (X^T X)^{-1} X^T D$$

Training using Gradient Descent

Single-Layer Neural Network with a Continuous Activation Function



Training Using Gradient Descent - Different Strategies:

1. In each step, update the weights of only one randomly chosen neuron.
2. Simultaneously update the weight vectors of all neurons.

Single-Layer Neural Network with a Continuous Activation Function

Training Using Gradient Descent ... e.g., SSE Loss Function

$$E_{SSE}(W) = \sum_{j=1}^{m} E_{SSE}(\vec{w}_j) = \frac{1}{2}\sum_{j=1}^{m}\sum_{p=1}^{N}(d_{pj} - y_{pj})^2$$

$$= \frac{1}{2}\sum_{j=1}^{m}\sum_{p=1}^{N}\left(d_{pj} - f\left(\sum_{i=0}^{n} w_{ij}x_{pi}\right)\right)^2 = \sum_{p=1}^{N} E_p(W)$$

$$E_p(W) = \frac{1}{2}\sum_{j=1}^{m}(d_{pj} - y_{pj})^2 \quad \text{(error function for a single sample)}$$

Partial Derivatives:

$$\frac{\partial E_p}{\partial w_{ij}} = \frac{\partial E_p}{\partial y_{pj}}\frac{\partial y_{pj}}{\partial \xi_{pj}}\frac{\partial \xi_{pj}}{\partial w_{ij}} = -(d_{pj} - y_{pj})f'(\xi_{pj})x_{pi}$$

Single-Layer Neural Network with a Continuous Activation Function

Training Using Gradient Descent

- Update rule for a single weight:

$$w_{ij}(t+1) = w_{ij}(t) + \alpha_t(d_{tj} - y_{tj})f'(\xi_{tj})x_{ti}$$

- Update rule for a single neuron with weight vector $\vec{w}_j$:

$$\vec{w}_j(t+1) = \vec{w}_j(t) + \alpha_t(d_{tj} - y_{tj})f'(\xi_{tj})\vec{x}_t^T$$

- Update rule for a single layer of neurons with weight matrix W:

$$W(t+1) = W(t) + \alpha_t \vec{x}_t^T [f'(\vec{\xi}_t) \circ (\vec{d}_t - \vec{y}_t)]$$

  - Highly efficient computation.
  - $\circ$ ... Hadamard (element-wise) product of vectors.

Single-Layer Neural Network with a Continuous Activation Function

Training Using Gradient Descent

- Update rule for a single layer of neurons with weight matrix W:

$$W(t+1) = W(t) + \alpha_t \vec{x}_t^T [f'(\vec{\xi}_t) \circ (\vec{d}_t - \vec{y}_t)]$$

  - Highly efficient computation.
  - $\circ$ ... Hadamard (element-wise) product of vectors.
- Update rule for the batch version:

$$Xi_t = XW$$

$$Y_t = f(Xi_t)$$

$$W(t+1) = W(t) + \alpha_t X_t^T [f'(Xi_t) \circ (D_t - Y_t)]$$

  - Even more efficient computation.

Single-Layer Neural Network with a Continuous Activation Function

General Gradient Descent Algorithm (GD)

1. Initialize weights with small random real values .... $W(0)$ with shape $(n+1) \times m$
   Initialize learning rate parameter α_0.... $1 > \alpha_0 > 0$
2. Present the next training sample $(\vec{x}_t, \vec{d}_t)$ and compute the potential and actual model output:

$$\vec{\xi}_t = \vec{x}_t W$$

$$\vec{y}_t = f(\vec{\xi}_t)$$

3. Update weights:

$$W(t+1) = W(t) + \alpha_t \vec{x}_t^T [f'(\vec{\xi}_t) \circ (\vec{d}_t - \vec{y}_t)]$$

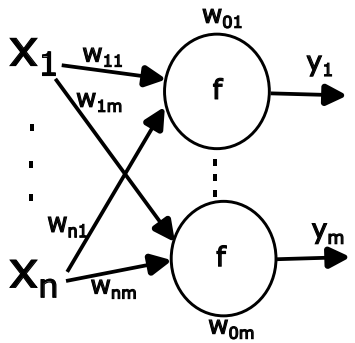
4. Optionally update the learning rate: $\alpha_t \rightarrow \alpha_{t+1}$
5. If the stopping condition is not met, return to step 2.

Single-Layer Neural Network with a Continuous Activation Function

General Batch Gradient Descent Algorithm (Batch GD)

1. Initialize weights with small random real values .... $W(0)$ with shape $(n+1) \times m$
   Initialize learning rate parameter α_0.... $1 > \alpha_0 > 0$
2. Present the training set X and compute the potential and actual model output:
$$Xi_t = XW$$
$$Y_t = f(Xi_t)$$
3. Update weights:
$$W(t+1) = W(t) + \alpha_t X_t^T [f'(Xi_t) \circ (D_t - Y_t)]$$
4. Optionally update the learning rate: $\alpha_t \rightarrow \alpha_{t+1}$
5. If the stopping condition is not met, return to step 2.

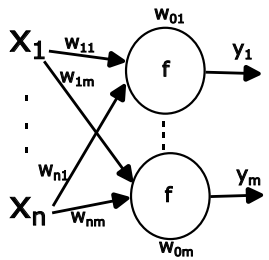
Multidimensional Linear Regression



- Model: A neural network consisting of a single layer of linear neurons (adding more layers would not provide any benefit - worth considering).
- Loss function for training using gradient descent: SSE (sum squared error), SAE (sum absolute error, suitable for data with outliers).

Atm.Pressure	Wind Intensity	Temperature	Precipitation	Risk
-1.3	-0.5	-0.2	0.9	
...	...	...	...	

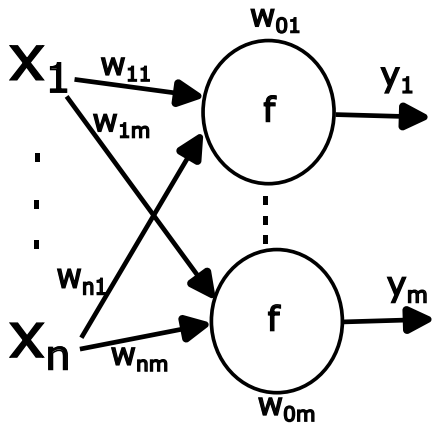
Linear Classification into Multiple Classes (Pattern Recognition)



- Model: A single layer of neurons with the tanh activation function or sigmoid (if using sigmoid, the target outputs must be binary-coded).
- Loss function for training using gradient descent: SSE, cross-entropy (lower risk of neuron saturation).
- Evaluation metrics: e.g., accuracy - percentage of correctly classified samples.

Size	Fur	Speaks?	Cat	Dog	Parrot	Carp
-1	0	-1	1	-1	-1	-1
1	1	-1	-1	1	-1	-1

Linear Classification into Multiple Classes (Pattern Recognition)



- The number of neurons in the network is chosen to be the same as the number of classes.
- Each neuron learns to recognize patterns from one specific class.

How to determine the winning class?

- by choosing the class with the highest output (**argmax**), or
- by applying a **softmax** activation to convert the outputs into class probabilities.

Linear Classification into Multiple Classes (Pattern Recognition)

How to determine the winning class?

- **argmax:**

$$k_{max} = \text{argmax}_k \, y_k$$

  ... returns index of the class with the highest output

- **Softmax - for continuous activation functions:** computes the probability distribution over classes:

$$softmax(y_k) = \frac{e^{y_k}}{\sum_{j=1}^{m} e^{y_j}}$$

  - In libraries, softmax is often implemented as a special fully connected output layer.

Cross-Entropy for Multiple Classes

Cross-Entropy Loss Function for Multi-Class Classification:

- Used for classification into multiple classes in combination with softmax output.

$$E = -\sum_{p=1}^{N}\sum_{j=1}^{m} d_{pj} \log y_{pj} \tag{1}$$

- d_{pj} is the desired output (1 for the correct class, otherwise 0).
- y_{pj} is the probability of class j obtained from softmax.

Gradient of the loss function with respect to weights:

$$\frac{\partial E_p}{\partial w_{ij}} = (y_{pj} - d_{pj}) x_{pi} \tag{2}$$

$\rightarrow$ More efficient training, natural interpretation as the negative log probability of the correct class.

Examples

perceptron_layer.ipynb

- Example implementation of a single-layer neural network.
- Simple example: Gallbladder (Example 1) extended with additional output variables.
  - Observation: The gradient method handles the task, although training takes longer and tuning the hyperparameters is more challenging.
- More complex example: Letter recognition.
  - Observations: Here, the advantage of cross-entropy loss over SSE becomes apparent.
  - We can observe jumps in the loss function and neuron saturation with SSE.
  - Batch training is significantly more efficient than iterative training.

Today's Class

1. Neural network consisting of a single layer of neurons
2. **Neural network with multiple layers of neurons – Introduction**

Multi-Layer Neural Network (Multi-Layer Perceptron, MLP, 1980)

- Hierarchical **sequential** architecture: neurons are arranged in layers
- **Dense (fully connected) layers**: every neuron in one layer is connected to every neuron in the next layer

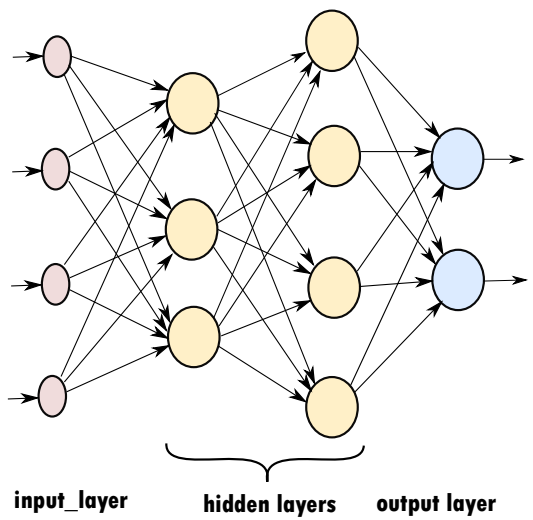
Special input layer:

- corresponds to the inputs of the neural network

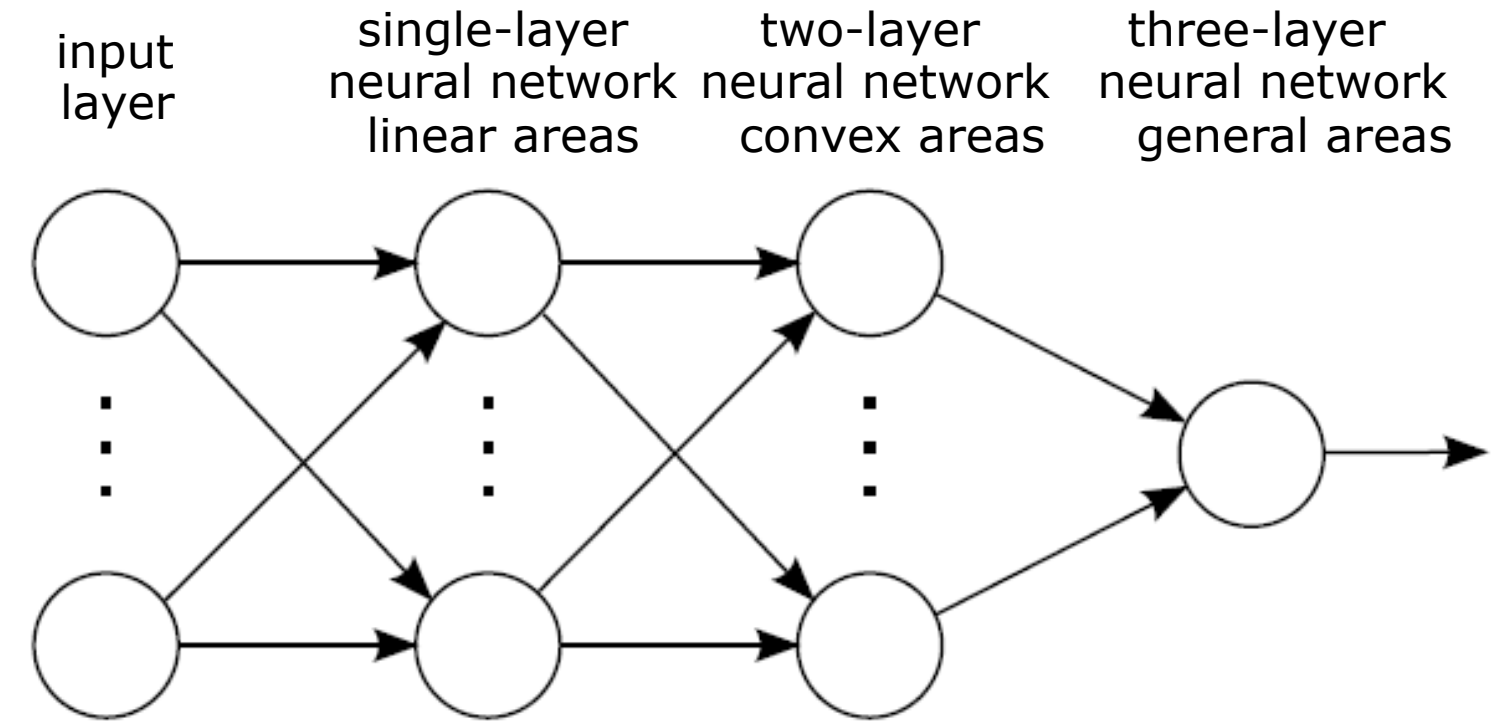
Output layer:

- the output (response) of the network corresponds to the activations of the output neurons

The remaining layers are **hidden layers**.



Motivation: Multi-Layer Perceptron

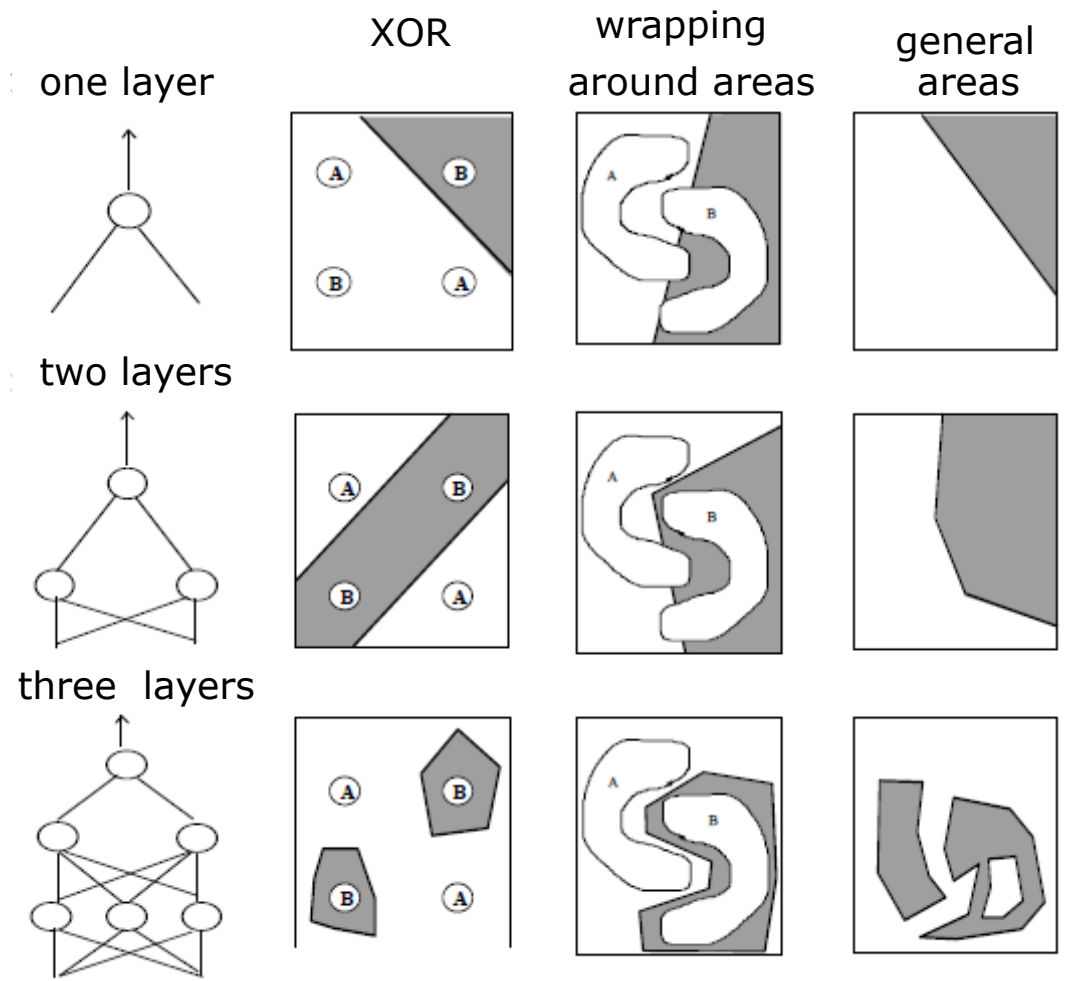


Multilayer neural network with non-linear (a.g., tanh) activations

Source: E. Volná: Neuronové sítě 1, Ostrava, 2008

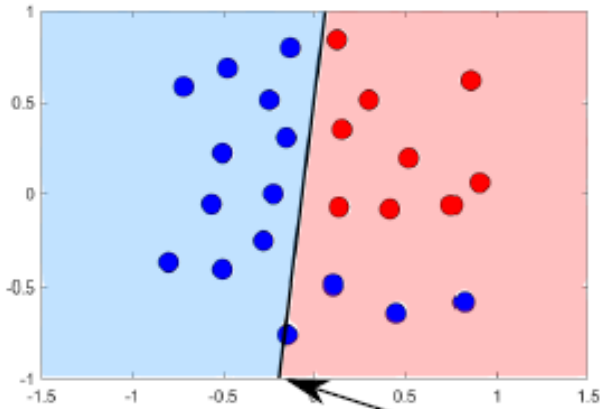
Motivation: Multi-Layer Perceptron

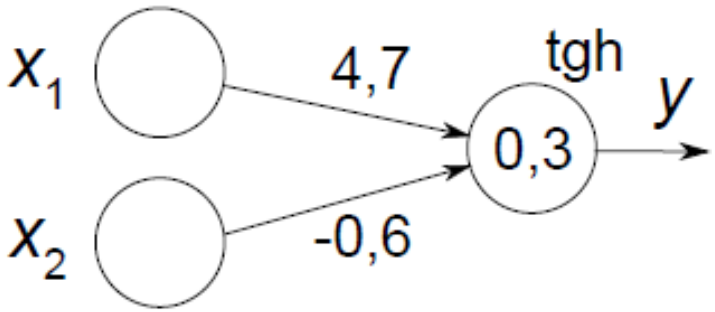
Multilayer Perceptron



Source: E. Volná: Neuronové sítě 1, Ostrava, 2008

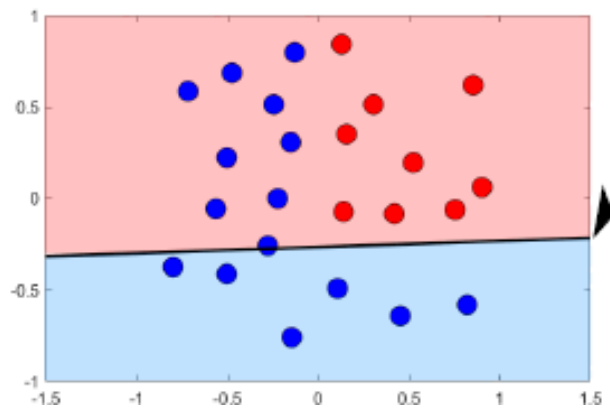
Motivation: Multi-Layer Perceptron

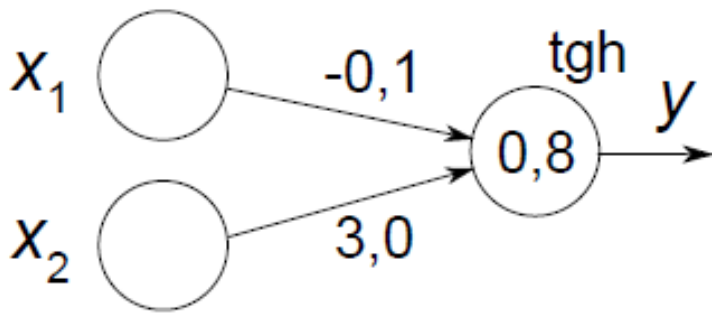




$$0{,}3 + 4{,}7x_1 - 0{,}6\,x_2 = 0$$

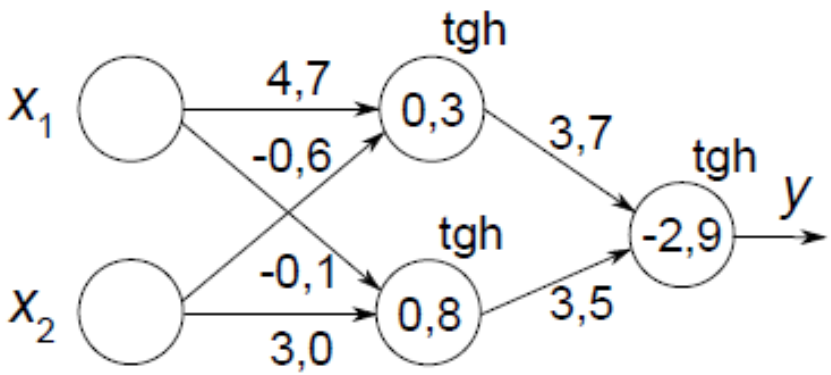
$$0{,}8 - 0{,}1x_1 + 3{,}0\,x_2 = 0$$

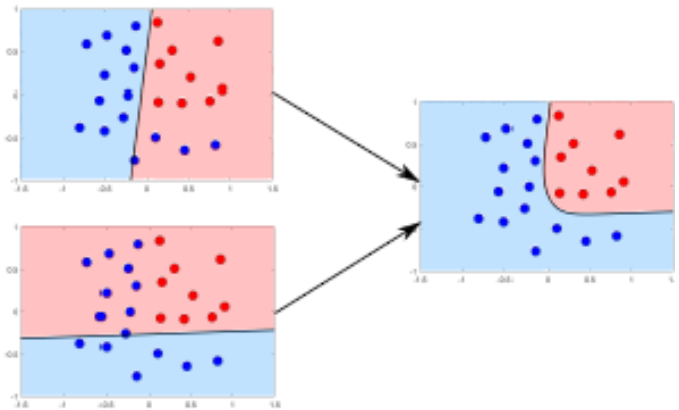




Source: K. Horaisová, Neural Networks 2, FNSPE CTU Děčín

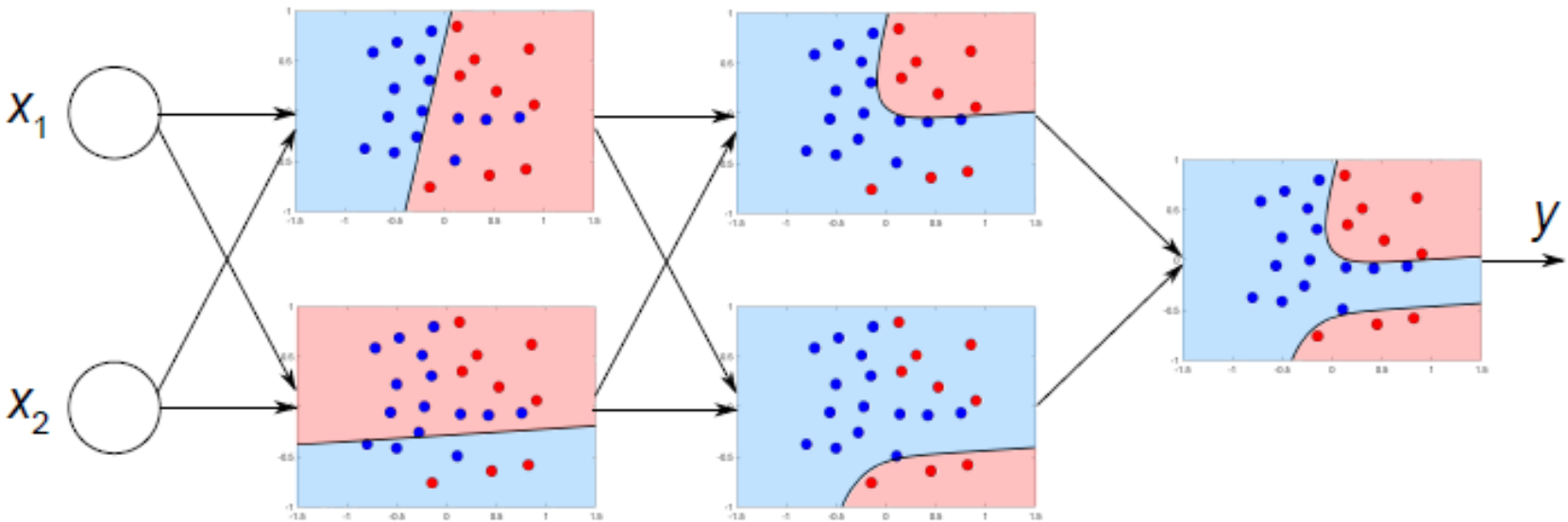
Motivation: Multi-Layer Perceptron





Source: K. Horaisová, Neural Networks 2, FNSPE CTU Děčín

Motivation: Multi-Layer Perceptron



Source: K. Horaisová, Neural Networks 2, FNSPE CTU Děčín

Motivation: Multi-Layer Perceptron

$\rightarrow$ For general classification tasks, it is sufficient to use a neural network with two hidden layers and one output layer.

- All layers with non-linear activations

What about multi-layer linear networks?

- Stacking linear layers has no benefit — a composition of linear transformations is equivalent to a single linear layer.

Can an MLP be used for regression tasks?

- Yes — the output layer is typically linear in that case.
- To capture non-linear dependencies, the hidden layers must use non-linear activation functions (e.g., ReLU, sigmoid, tanh).

Motivation: Multi-Layer Perceptron

Universal Approximation Theorem (Cybenko, 1989; Hornik et al., 1989)

- A layered neural network with a single hidden layer, a sufficient number of neurons, and a non-linear activation function can approximate any continuous function on a compact domain.
- In practice, networks with more hidden layers often perform better:
  - faster and easier convergence during training
  - more efficient representation of complex functions with fewer parameters (deeper networks often require fewer neurons)
  - better generalization ability

Multi-Layer Neural Network (Multi-Layer Perceptron, MLP, 1980)

- Hierarchical **sequential** architecture: neurons are arranged in layers
- **Dense (fully connected) layers**: every neuron in one layer is connected to every neuron in the next layer

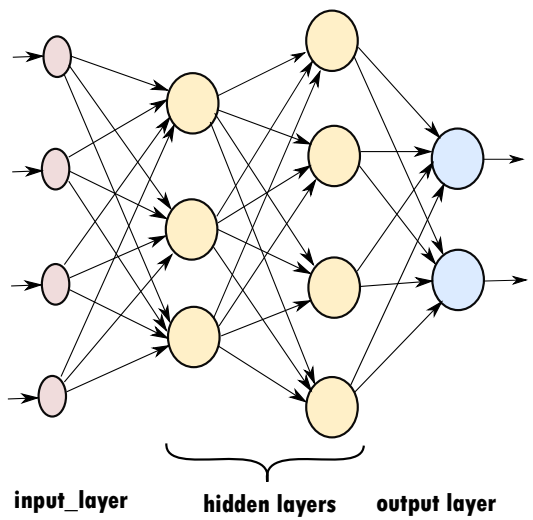
Special input layer:

- corresponds to the inputs of the neural network

Output layer:

- the output (response) of the network corresponds to the activations of the output neurons

The remaining layers are **hidden layers**.



Multi-Layer Perceptron (MLP)

How to implement the model?

- A sequence (list) of layers $L_0, ..., L_{max}$
- We already know how to implement a single layer

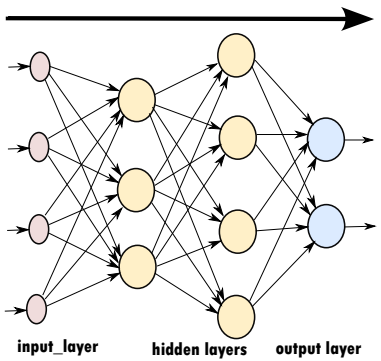
Computing the output (response) of a layered neural network:

- by performing a **forward pass**
- we process one layer at a time, starting from the input layer and going toward the output:
  - present the input to the current layer
  - compute the layer's output
  - use this output as the input to the next layer

Multi-Layer Perceptron (MLP)

Computing the output (response) of a layered neural network: by performing a **forward pass**

- Given an input vector $\vec{x}$ of length n, the model computes an output vector $\vec{y}$ of length m
  1. Output of the input layer neurons: $y_i = x_i$
  2. Then we move from the first hidden layer toward the output layer and compute (and store) the output y_j of each neuron j using the activations of neurons from the previous layer: $y_j = f(\xi_j) = f\left(\sum_i w_{ij} y_i + b_i\right)$ (i indexes neurons in the layer preceding neuron j)
  3. The network output $\vec{y} = (y_1, ..., y_m)$ is the vector of outputs from the output layer neurons



Multi-Layer Perceptron (MLP)

How to implement the model?

- A sequence (list) of layers $L_0, ..., L_{max}$
- We already know how to implement a single layer

Computing the output (response) of a layered neural network:

- by performing a **forward pass**
- we process one layer at a time, starting from the input layer and going toward the output:
  - present the input to the current layer
  - compute the layer's output
  - use this output as the input to the next layer

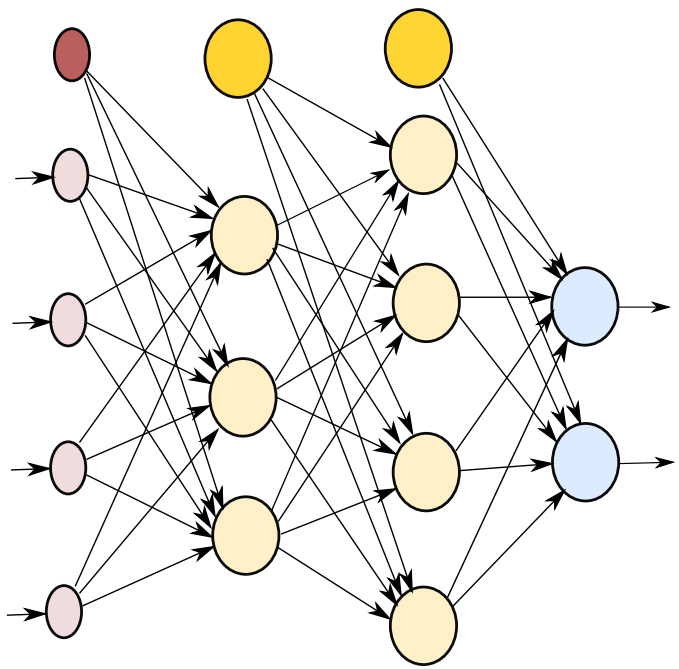
Multi-Layer Perceptron (MLP)

Forward pass: matrix-based computation of network output

- We again simplify and speed up the computation using matrix operations and **bias neurons**

Bias neurons

- For each hidden layer (similarly to the input layer), we add a bias neuron with constant output 1 to represent the bias terms in the next layer

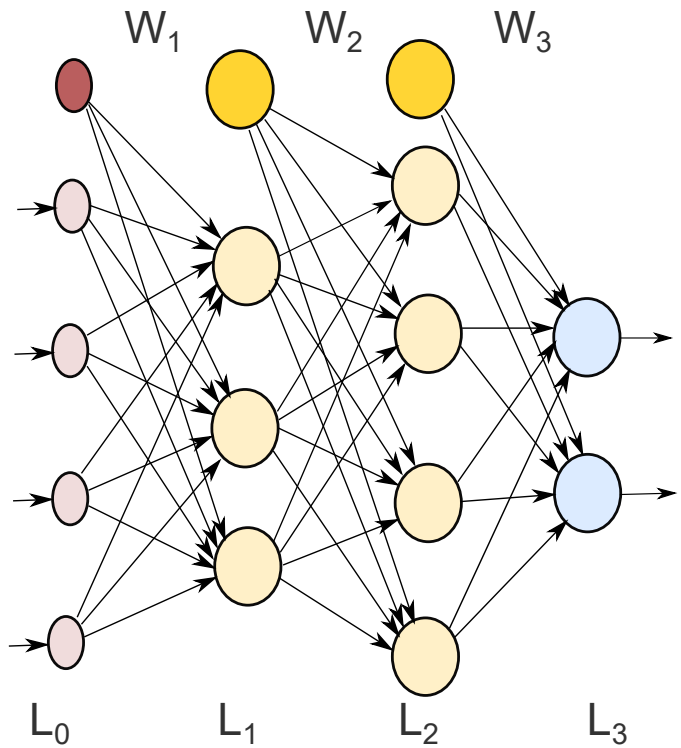


Matrix Representation of a Multi-Layer Neural Network

Matrix-based representation of an MLP:

- Let the network consist of layers L_0 (input), ..., L_{max} (output)
- The weights of all neurons can be represented by matrices $W_1, ..., W_{L_{max}}$
- W_L is the weight matrix between layers $L-1$ and L, of size $(n_{L-1}+1) \times n_L$

(just as in the case of a single-layer neural network)



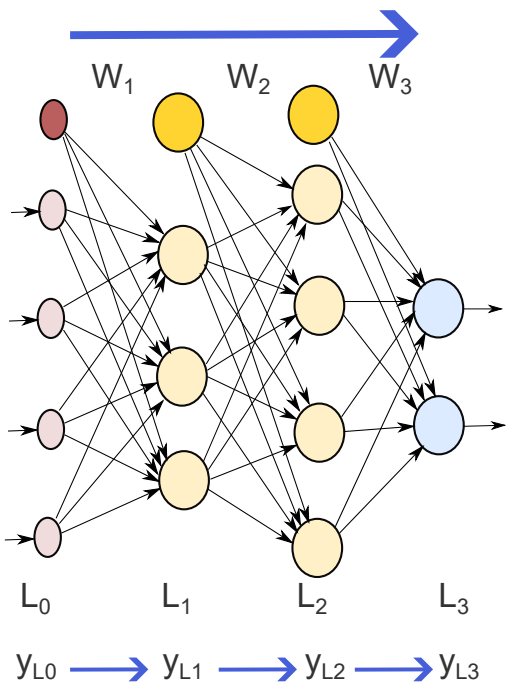
Computing the Output – Matrix Form I (Single Input)

1. Present input vector $\vec{x}$
2. For layers $L = L_0, L_1, ..., L_{max}$ compute the output vectors $\vec{y}_0, ..., \vec{y}_{L_{max}}$:
   - For $L = L_0$ (input layer): $\vec{y}_0 = \vec{x}$
   - For $L = L_1, ..., L_{max}$:

$$\vec{z}_L = f(\vec{\xi}_L) = f(\vec{y}_{L-1} W_L)$$

$$\vec{y}_L = (1 \,|\, \vec{z}_L)$$

   where W_L is the extended weight matrix between layers $L-1$ and L
3. The network output $\vec{y} = (y_1, ..., y_m) = \vec{y}_{L_{max}}$ is given by the output layer activations



$\rightarrow$ very efficient

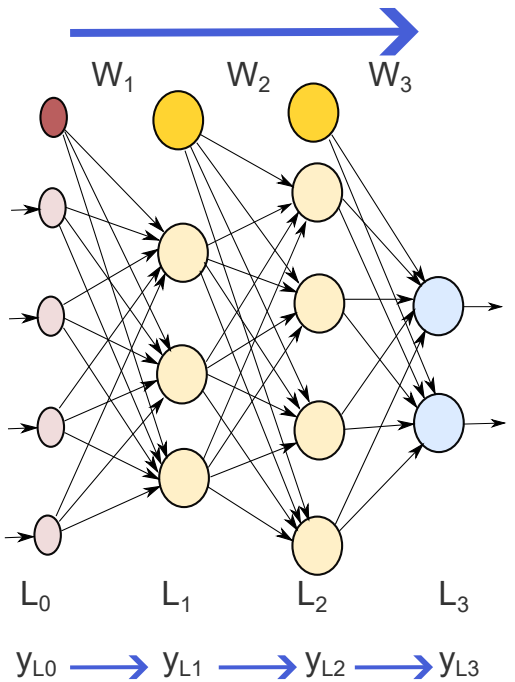
Computing the Output – Matrix Form II (Batch of Inputs)

1. Present a matrix of input vectors X
2. For layers $L = L_0, L_1, ..., L_{max}$ compute the output matrices $Y_0, ..., Y_{L_{max}}$:
   - For $L = L_0$ (input layer): $Y_0 = X$
   - For $L = L_1, ..., L_{max}$:

$$Z_L = f(\xi_L) = f(Y_{L-1} W_L)$$

$$Y_L = (1 \,|\, Z_L)$$

   where W_L is the extended weight matrix between layers $L-1$ and L
3. The network output $Y = Y_{L_{max}}$ is the matrix of outputs from the output layer



$\rightarrow$ even more efficient

Training a Multi-Layer Neural Network

Neural network configuration:

- The weight vector (and biases) of all neurons in the network, denoted by $\vec{w}$, represents all model parameters.

How do we train a multi-layer neural network?

- We use a gradient-based method for the chosen loss function E and the parameter vector $\vec{w}$:

$$\vec{w}(t+1) = \vec{w}(t) - \alpha \nabla E(\vec{w})$$

- Computing partial derivatives and updating weights is more complex than in a single-layer network
- The process is greatly simplified by the **backpropagation algorithm**

Backpropagation Algorithm

(Werbos, Rumelhart, 1974–1986)

We are given:

- A training set T with N training samples $(\vec{x}_p, \vec{d}_p)$:
  - $\vec{x}^p = (x_1^p, ..., x_n^p)$ — input pattern
  - $\vec{d}^p = (d_1^p, ..., d_m^p)$ — desired output

$x_{10} = 1$	x_{11}	...	x_{1n}	d_{11}	...	d_{1m}
...	...		...	...		...
$x_{N0} = 1$	x_{N1}	...	x_{Nn}	d_{N1}	...	d_{Nm}

- A multi-layer neural network with a defined architecture, with $n + 1$ input neurons and m output neurons.
- The neurons must use continuous, differentiable activation functions.

Goal:

- Adjust the weights of all neurons in the network so that the actual network output matches the desired output.

Backpropagation Algorithm

Loss function:

- Instead of SSE, we often use MSE (mean squared error), i.e., average over all patterns:
- For one training example:

$$E_p(\vec{w}) = \frac{1}{2}\sum_{j=1}^{m}(d_{pj} - y_{pj})^2$$

- For the whole training set:

$$E(\vec{w}) = \frac{1}{N}\sum_{p=1}^{N} E_p = \frac{1}{2N}\sum_{p=1}^{N}\sum_{j=1}^{m}(d_{pj} - y_{pj})^2$$

- Other loss functions are also used (e.g., cross-entropy)

Goal of the backpropagation algorithm:

- Minimize the loss function E on the given training set T

Backpropagation Algorithm

Core principle: it is a standard gradient descent method

1. Randomly initialize the model parameters (weights and biases)
2. Repeat for training epochs:
   - Prepare a batch of input samples X and their corresponding target outputs D
   - Compute the model's actual outputs Y
   - Compute the model error (difference between Y and D)
   - Update parameters (weights and biases) to slightly reduce the error (i.e., move in the opposite direction of the loss gradient):

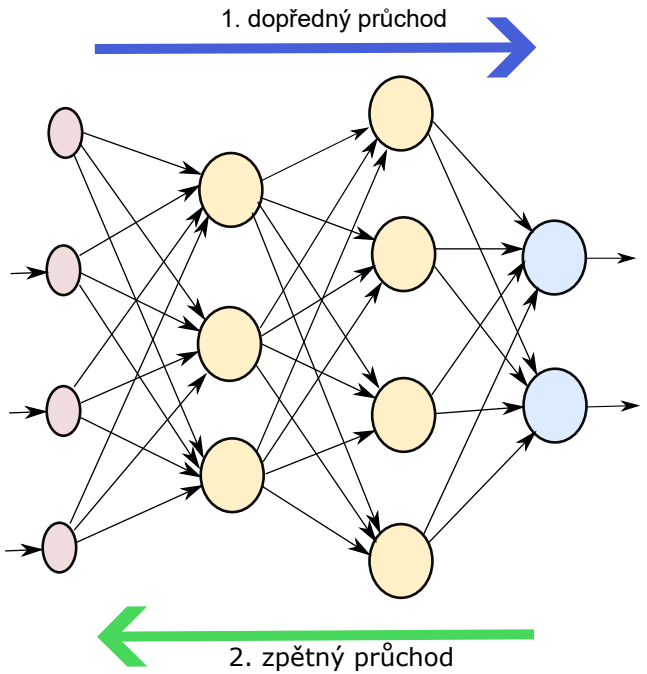
$$w_i(t+1) = w_i(t) - \alpha_t \frac{\partial E_t}{\partial w_i}$$

Nice visualizations of loss surfaces:
jithinjk.github.io/blog/nn_loss_visualized.md.html
izmailovpavel.github.io/curves_blogpost

Backpropagation Algorithm

Basic principle of backpropagation:

1. Compute the actual network output for the given batch of training samples
   - by a single pass from the input to the output layer (forward pass)
2. Compare the actual and desired outputs
3. Update the weights and biases:
   - in the direction opposite to the gradient of the loss
   - using a single pass from the output to the input layer (backward pass)



Backpropagation – Adaptation Rules

- Loss function for a single training sample $(\vec{x}_t, \vec{d}_t)$ with network output $\vec{y}_t$:

$$E_t = \frac{1}{2}\sum_{j=1}^{m}(d_{tj} - y_{tj})^2$$

- Partial derivative:

$$\frac{\partial E_t}{\partial w_{ij}} = \frac{\partial E_t}{\partial y_{tj}} \cdot \frac{\partial y_{tj}}{\partial \xi_{tj}} \cdot \frac{\partial \xi_{tj}}{\partial w_{ij}}$$

- Weight update rule from neuron i to neuron j at time t:

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

- Where $\Delta w_{ij}(t)$ is the weight increment that reduces E_t:

$$\Delta w_{ij}(t) = -\alpha \frac{\partial E_t}{\partial w_{ij}} = -\alpha \cdot \frac{\partial E_t}{\partial y_{tj}} \cdot \frac{\partial y_{tj}}{\partial \xi_{tj}} \cdot \frac{\partial \xi_{tj}}{\partial w_{ij}}$$

- α is the learning rate.

Backpropagation – Adaptation Rules

Key idea:

- We do not compute the derivative $\frac{\partial E_t}{\partial w_{ij}}$ separately for each weight (which would be extremely inefficient).
- To compute $\frac{\partial E_t}{\partial w_{ij}}$, specifically its component $\delta_j = \frac{\partial E_t}{\partial \xi_{tj}}$, we can reuse error terms δ_k from neurons k in the next layer.
- $\Rightarrow$ By a single backward pass (layer by layer), we compute the error term δ_j for each neuron j.
- Then we compute the weight gradient easily as:

$$\frac{\partial E_t}{\partial w_{ij}} = \delta_{tj} \cdot \frac{\partial \xi_{tj}}{\partial w_{ij}}$$

Backpropagation – Adaptation Rules

Define:

$$\delta_{tj} = \frac{\partial E_t}{\partial y_{tj}} \cdot \frac{\partial y_{tj}}{\partial \xi_{tj}} = \frac{\partial E_t}{\partial \xi_{tj}} \quad \text{(error term for neuron } j\text{)}$$

$\Rightarrow$ this is the value we will backpropagate.
Then:

$$\Delta w_{ij}(t) = -\alpha \cdot \frac{\partial E_t}{\partial y_{tj}} \cdot \frac{\partial y_{tj}}{\partial \xi_{tj}} \cdot \frac{\partial \xi_{tj}}{\partial w_{ij}} = -\alpha \delta_{tj} \cdot \frac{\partial \xi_{tj}}{\partial w_{ij}} = -\alpha \delta_{tj} \cdot y_{ti}$$

where:

$$\frac{\partial \xi_{tj}}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \left(\sum_k w_{kj} y_{tk} \right) = y_{ti}$$

(where k indexes neurons in the preceeding layer to j)

Backpropagation – Adaptation Rules

I. For neurons j in the output layer:

$$E_t = \frac{1}{2}\sum_{j=1}^{m}(d_{tj} - y_{tj})^2$$

$$\delta_{tj} = \frac{\partial E_t}{\partial y_{tj}} \cdot \frac{\partial y_{tj}}{\partial \xi_{tj}} = -(d_{tj} - y_{tj})f'(\xi_{tj})$$

For weights w_{ij} between the last hidden layer and output layer:

$$\Delta w_{ij}(t) = -\alpha\delta_{tj} \cdot y_{ti} = \alpha(d_{tj} - y_{tj})f'(\xi_{tj})y_{ti}$$

Backpropagation – Adaptation Rules

For neurons j in the last hidden layer:

$$\begin{aligned} E_t &= \frac{1}{2}\sum_{k=1}^{m}(d_{tk} - y_{tk})^2 = \frac{1}{2}\sum_{k=1}^{m}(d_{tk} - f(\xi_{tk}))^2 \\ &= \frac{1}{2}\sum_{k=1}^{m}(d_{tk} - f(\sum_{j} w_{jk} y_{tj}))^2 \end{aligned}$$

(k indexes all output neurons, j indexes all neurons in the last hidden layer)

$$\begin{aligned} \frac{\partial E_t}{\partial y_{tj}} &= \sum_{k=1}^{m}\frac{\partial E_t}{\partial y_{tk}} \cdot \frac{\partial y_{tk}}{\partial y_{tj}} = \sum_{k=1}^{m}\frac{\partial E_t}{\partial y_{tk}} \cdot \frac{\partial y_{tk}}{\partial \xi_{tk}} \cdot \frac{\partial \xi_{tk}}{\partial y_{tj}} \\ &= \sum_{k=1}^{m}\delta_{tk} \cdot w_{jk} \end{aligned}$$

Backpropagation – Adaptation Rules

For neurons j in the last hidden layer:

$$\frac{\partial E_t}{\partial y_{tj}} = \sum_{k=1}^{m} \delta_{tk} \cdot w_{jk}$$

$$\delta_{tj} = \frac{\partial E_t}{\partial y_{tj}} \cdot \frac{\partial y_{tj}}{\partial \xi_{tj}} = \left(\sum_{k=1}^{m} \delta_{tk} \cdot w_{jk} \right) f'(\xi_{tj})$$

For weights w_{ij} between the penultimate and last hidden layer:

$$\Delta w_{ij}(t) = -\alpha \delta_{tj} \cdot y_{ti} = -\alpha \left(\sum_{k=1}^{m} \delta_{tk} w_{jk} \right) f'(\xi_{tj}) y_{ti}$$

Backpropagation – Adaptation Rules

For neurons j in any hidden layer:

$$\begin{aligned}\frac{\partial E_t}{\partial y_{tj}} &= \sum_k \frac{\partial E_t}{\partial y_{tk}} \cdot \frac{\partial y_{tk}}{\partial y_{tj}} \\ &= \sum_k \frac{\partial E_t}{\partial y_{tk}} \cdot \frac{\partial y_{tk}}{\partial \xi_{tk}} \cdot \frac{\partial \xi_{tk}}{\partial y_{tj}} \\ &= \sum_k \delta_{tk} \cdot w_{jk}\end{aligned}$$

(k indexes neurons in the layer following j)

$$\delta_{tj} = \frac{\partial E_t}{\partial y_{tj}} \cdot \frac{\partial y_{tj}}{\partial \xi_{tj}} = \left(\sum_k \delta_{tk} \cdot w_{jk} \right) f'(\xi_{tj})$$

Backpropagation – Adaptation Rules

Summary:

$$w_{ij}(t+1) = w_{ij}(t) - \alpha \delta_{tj} y_{ti}$$

where:

- for output neuron j:

$$\delta_{tj} = f'(\xi_{tj})(y_{tj} - d_{tj})$$

- for hidden neuron j:

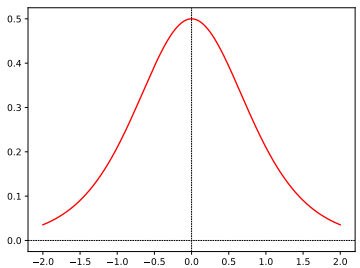
$$\delta_{tj} = f'(\xi_{tj}) \sum_k (\delta_{tk} w_{jk})$$

Reminder: Derivatives of Activation Functions

Sigmoid (logsig):

$$y = f(\xi) = \frac{1}{1 + e^{-\lambda\xi}}$$

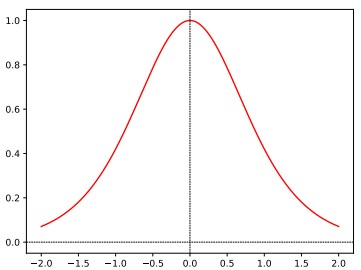
$$f'(\xi) = \lambda y(1 - y)$$



Hyperbolic tangent (tanh):

$$y = f(\xi) = \frac{1 - e^{-2\lambda\xi}}{1 + e^{-2\lambda\xi}}$$

$$f'(\xi) = \lambda^2(1 + y)(1 - y)$$



Backpropagation Algorithm (Iterative Variant)

1. **Network initialization**
   Initialize all weights and biases with small random values.
2. **Present a training sample** in the form $(\vec{x}_t, \vec{d}_t)$
3. **Forward pass:**
   - Process layer by layer from input to output.
   - For each neuron j, compute (and store) its output y_{tj} using the outputs of the previous layer (including bias neuron):

$$y_{tj} = f(\xi_{tj}) = f\left(\sum_i w_{ij} y_{ti}\right)$$

     where i indexes neurons in the preceding layer.
   - The forward computation is done efficiently using matrix operations.

Backpropagation Algorithm (Iterative Variant)

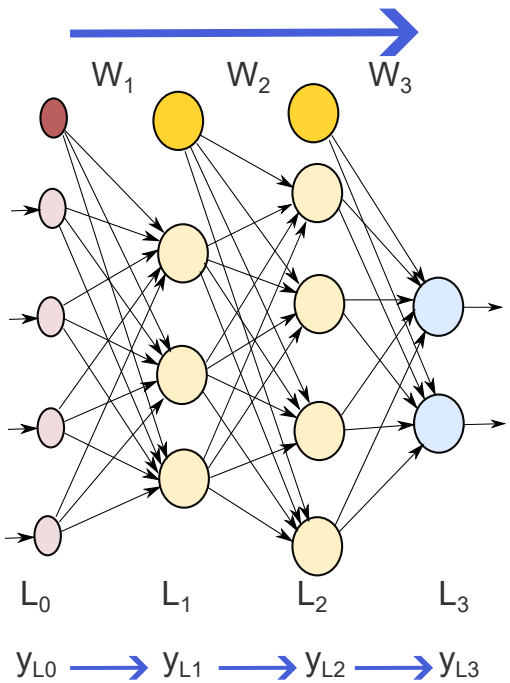
3. **Forward pass (matrix version):**
   For layers $L = L_0, L_1, ..., L_{max}$ compute the output vectors $\vec{y}_{L_0}, ..., \vec{y}_{L_{max}}$

- For $L = L_0$ (input layer): $\vec{y}_{L_0} = \vec{x}$
- For $L = L_1, ..., L_{max}$:

$$\vec{z}_L = f(\vec{\xi}_L) = f(\vec{y}_{L-1} W_L)$$

$$\vec{y}_L = (1 \,|\, \vec{z}_L)$$

W_L is the extended weight matrix between layers $L-1$ and L



- The network output $\vec{y} = (y_1, ..., y_m) = \vec{y}_{L_{max}}$ is given by the output layer activations.

Backpropagation Algorithm (Batch Variant)

Forward pass for batch GD (matrix version):

1. Present input matrix X
2. For layers $L = L_0, ..., L_{max}$ compute output matrices $Y_0, ..., Y_{L_{max}}$:
   - $L = L_0$: $Y_0 = X$
   - $L = L_1, ..., L_{max}$:

$$Z_L = f(\xi_L) = f(Y_{L-1} W_L)$$

$$Y_L = (1 \,|\, Z_L)$$

3. Network output: $Y = Y_{L_{max}}$

Backpropagation Algorithm (Iterative Variant)

4. **Backward pass:**
   Go from the output layer toward the first hidden layer. For each neuron j, compute and store its error term δ_{tj} and update the incoming weights w_{ij}:
   - For output neurons:

$$\delta_{tj} = f'(\xi_{tj})(y_{tj} - d_{tj})$$

   - For hidden neurons:

$$\delta_{tj} = f'(\xi_{tj}) \sum_k \delta_{tk} w_{jk}$$

     (k indexes neurons in the next layer)
   - For every connection from neuron i to j (including bias):

$$w_{ij}(t+1) = w_{ij}(t) - \alpha \delta_{tj} y_{ti}$$

Backpropagation Algorithm (Matrix – Iterative Variant)

4. **Backward pass (matrix version):** For layers $L = L_{max}, ..., L_1$ compute error terms $\vec{\delta}_L$ and update weight matrices W_L:
   - For output layer:

$$\vec{\delta}_{L_{max}} = f'(\vec{\xi}_{L_{max}}) \circ (\vec{y}_t - \vec{d}_t)$$

   - For hidden layers:

$$\vec{\delta}_L = \left(f'(\vec{\xi}_L) \circ \vec{\delta}_{L+1} \right) W_{L+1}^T$$

   - Update weights:

$$W_L(t+1) = W_L(t) - \alpha \vec{y}_{L-1}^T \vec{\delta}_L$$

Backpropagation Algorithm (Batch Variant)

Backward pass for batch GD (matrix version):

- For layers $L = L_{max}, ..., 1$ compute error terms Δ_L and update weight matrices W_L:
  - Output layer:

$$\Delta_{L_{max}} = f'(\xi_{L_{max}}) \circ (Y - D)$$

  - Hidden layers:

$$\Delta_L = (f'(\xi_L) \circ \Delta_{L+1}) W_{L+1}^T$$

  - Update weight matrix:

$$W_L(t+1) = W_L(t) - \alpha Y_{L-1}^T \Delta_L$$

Backpropagation Algorithm (Iterative Variant)

5. **Stopping condition:**
   If the stopping condition is not met, return to step 2.
   - Maximum number of epochs
   - Time limit
   - Training error drops below threshold: $E < E_{min}$
   - Validation error stops decreasing (early stopping)
   - Weight updates become very small: $|\Delta w| < \Delta_{min}$

Backpropagation – How to Present Training Samples

Presentation strategies:

1. **Sample-wise per epoch (online GD):** Each sample is presented once per epoch, samples are shuffled every epoch.
   - Maximum number of epochs = how many times the full dataset is presented.
2. **Batch-wise per epoch (batch GD):**
   - The entire training set is used at once to compute and apply a single weight update.
3. **Mini-batch training (stochastic GD, SGD):**
   - Training set is randomly split into small batches that are processed iteratively.

Backpropagation Algorithm

Discussion of Training Strategies

- **Online Gradient Descent (Online GD)**
  - Fast learning, but relatively unstable — the algorithm reduces the error for the current training sample, but the error may increase for other samples.
  - More sensitive to outliers and to hyperparameter settings; randomness can help escape local minima.
- **Batch Gradient Descent (Batch GD)**
  - More stable and efficient for small datasets.
  - Computationally and memory intensive for large datasets.
  - More sensitive to noise in the data.
- **Mini-batch Stochastic Gradient Descent (Mini-batch SGD)**
  - Combines advantages of both previous methods.
  - Commonly used for large datasets and deep neural networks.

Main Deep Learning Frameworks in Python

- **TensorFlow** – Open-source library by Google.
  - Powerful framework for AI applications (mobile, server).
  - Supports both static and dynamic computation graphs.
- **PyTorch** – Open-source library by Meta (Facebook).
  - Flexible and intuitive, ideal for research and academia.
  - Dynamic computation graphs, easy debugging.
- **Keras** – High-level universal API.
  - Beginner-friendly and easy to understand.
  - Great for fast prototyping. Runs on top of TensorFlow, JAX, or PyTorch.
- **PyTorch Lightning** – High-level wrapper for PyTorch.
  - Reduces boilerplate code in training routines.
  - Supports multi-GPU training, scaling, and reproducibility.
- **JAX** – Optimized for speed and experimental research.
- Previously popular **Theano** – now deprecated.

TensorFlow vs PyTorch – Comparison

TensorFlow – robust and production-ready, but more rigid:

- Part of a broader ecosystem (TensorBoard, TF Lite, etc.).
- Very efficient (C++/Python hybrid), supports distributed training, native TPU support.
- Optimized for deployment, mobile support (TF Lite), model compilation.
- Less developer-friendly: more code, harder to define custom models.
- Difficult debugging of complex models (C++ backend).

PyTorch – newer, rapidly evolving, research-focused:

- Pythonic, concise, and easier to use; gaining feature parity.
- Slightly less performant (pure Python), but highly flexible.
- Custom models and layers are very easy to implement and debug.

Other Useful Libraries

Data manipulation and numerical computing:

- **Scikit-learn (sklearn)** – classic ML algorithms; tools for data processing and model evaluation.
- **NumPy** – efficient numerical computing with arrays and tensors.
- **Pandas** – powerful data manipulation library for structured data (categorical, missing values).

Visualization:

- **Matplotlib** – general-purpose plotting (static, animated, interactive).
- **Plotly** – interactive visualizations.
- **Seaborn** – statistical data visualization (correlations, distributions, etc.).
- **TensorBoard** – learning visualization, especially for TensorFlow.

Practical Examples

NN_libraries.ipynb

- Commented examples comparing major deep learning frameworks (Keras, TensorFlow, PyTorch, Lightning) on a simple binary classification task.
- Demonstration of automatic symbolic tensor differentiation in TensorFlow and PyTorch.
- Frameworks and GPU support in practice.

NN_libraries_installation.ipynb

- Brief installation guide for running the examples locally on your own machine.

Interactive MLP Playground – Simple Visual Demos

https://playground.tensorflow.org/

- We'll explore it in more detail next time.
- Five classification tasks (of increasing difficulty – spiral is the hardest).
- You can configure network architecture and training parameters.
- Includes excellent visualizations: loss over time, weight signs and magnitudes, neuron behavior.
- Great for experimenting: how many layers and neurons are needed for which task?
- **Challenge:** can you train a model that solves the spiral task?