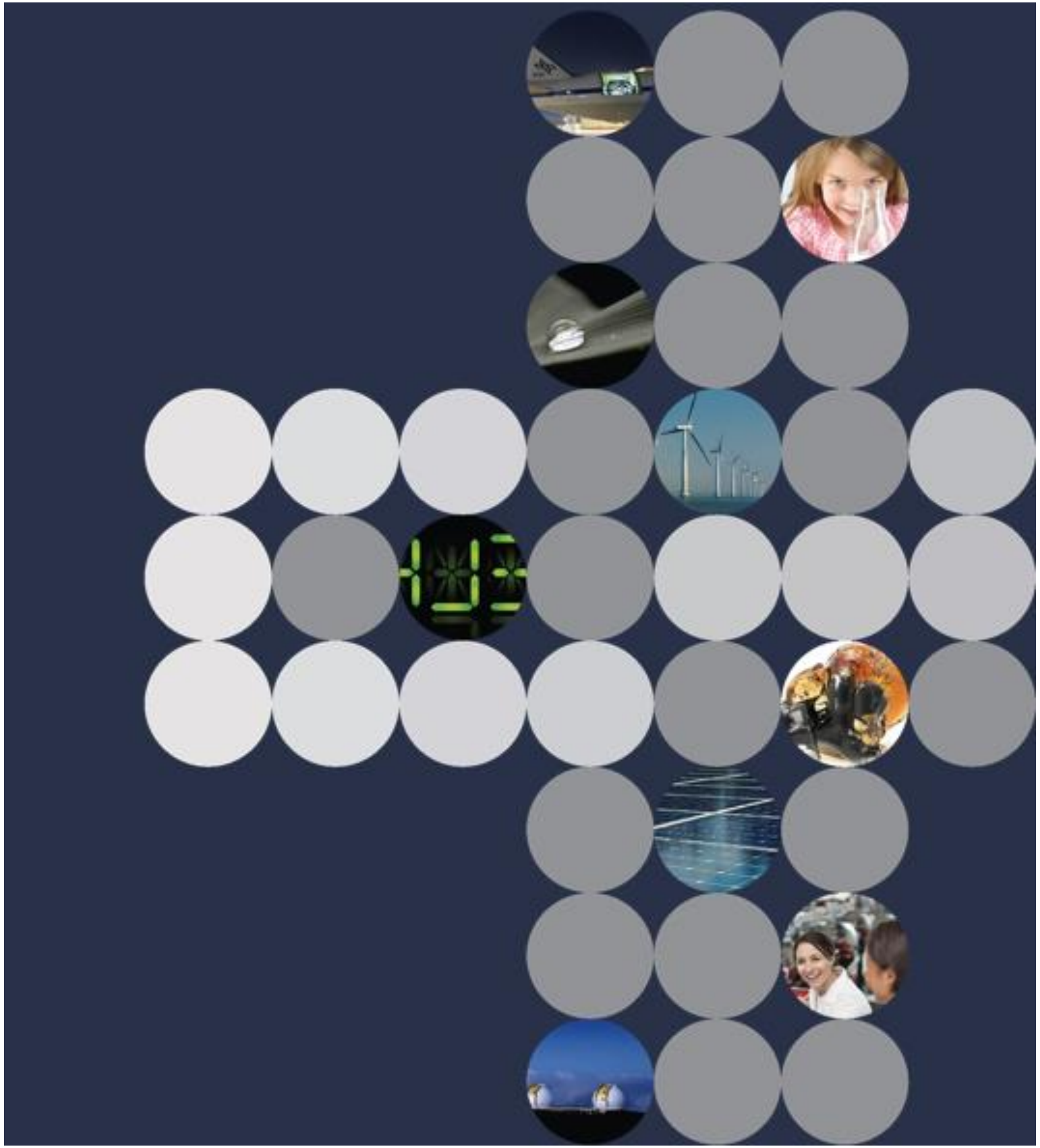
EE512 – Applied Biomedical Signal Processing

Introduction to neural networks

Clémentine AGUET
CSEM Signal Processing & AI Group



Content

- Introduction
- Perceptron
- Multilayer perceptron (MLP)
- Loss function
- Gradient descent
- Backpropagation
- Convolutional neural network
- Recurrent neural network
- Bias-variance trade-off
- Data

Introduction – What is deep learning?

Artificial Intelligence

Ability to mimic intelligent human behavior

Machine Learning

Ability to automatically learn and improve from experience

Deep Learning

Machine learning based on deep neural networks

Examples – Non-exhaustive list

Rule-based systems
Depth-first search algorithm
Breadth-first search algorithm
Propositional calculus
Predicate calculus logic

Linear regression
Logistic regression
Support vector machine
Decision trees
Gradient boosting
Principal component analysis
K-means clustering

Recurrent neural networks
Convolutional neural networks
Deep reinforcement learning
Generative adversarial networks

Introduction – Brief history

McCulloch and Pitts
First mathematical model of artificial neuron
1943

1958
Rosenblatt
Perceptron

Widrow and Hoff
ADALINE
Single layer artificial NN
1960

1969
Minsky and Papert
XOR problem

Rumelhart, Hinton, Williams
Multilayer perceptron (MLP)
Backpropagation
1986

1989
LeCun et al.
Convolutional NN

Krizhevsky, Sutskever, Hinton
AlexNet
2012

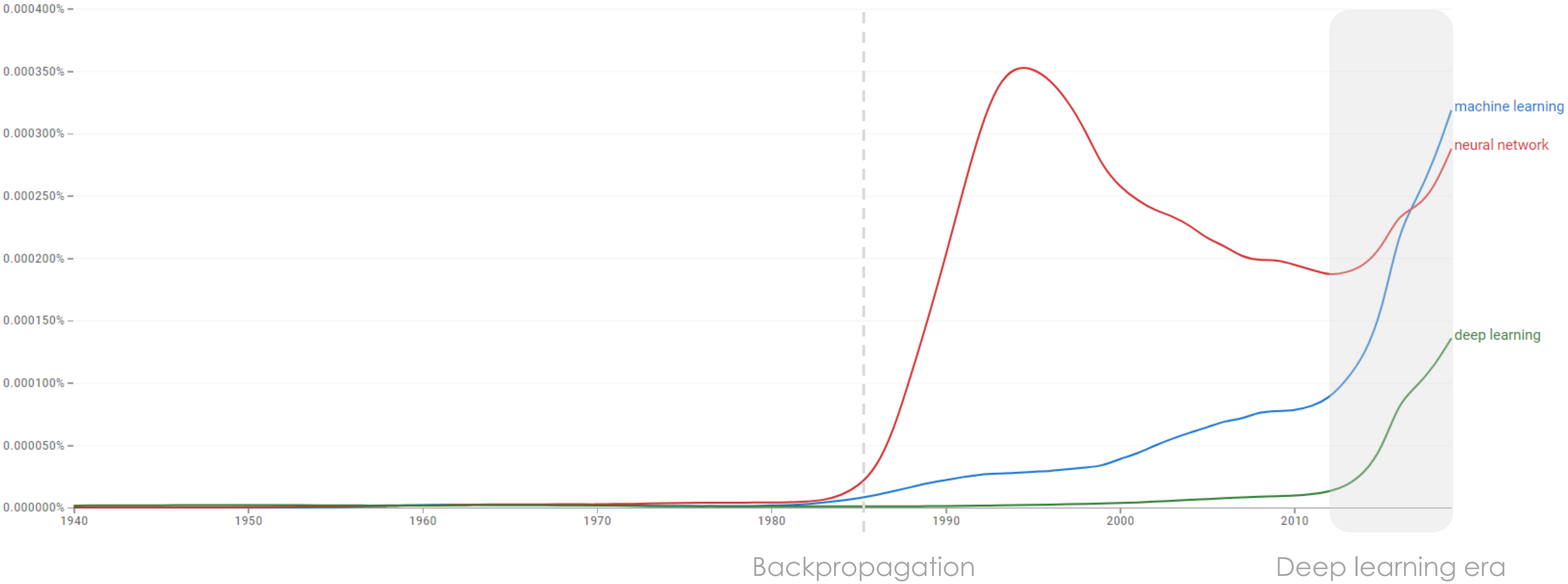
2014
Goodfellow
Generative adversarial networks

He et al.
ResNet
Skip connection
2015

Deep learning era

Introduction – Interest

NGRAM – Frequency of words found in printed sources



EPFL csem

Introduction – Why Now?

Neural networks date back decades. What has changed?

Big Data

- Larger datasets
- Easier collection and storage

Hardware

- Graphics processing units (GPUs)
- Parallelization

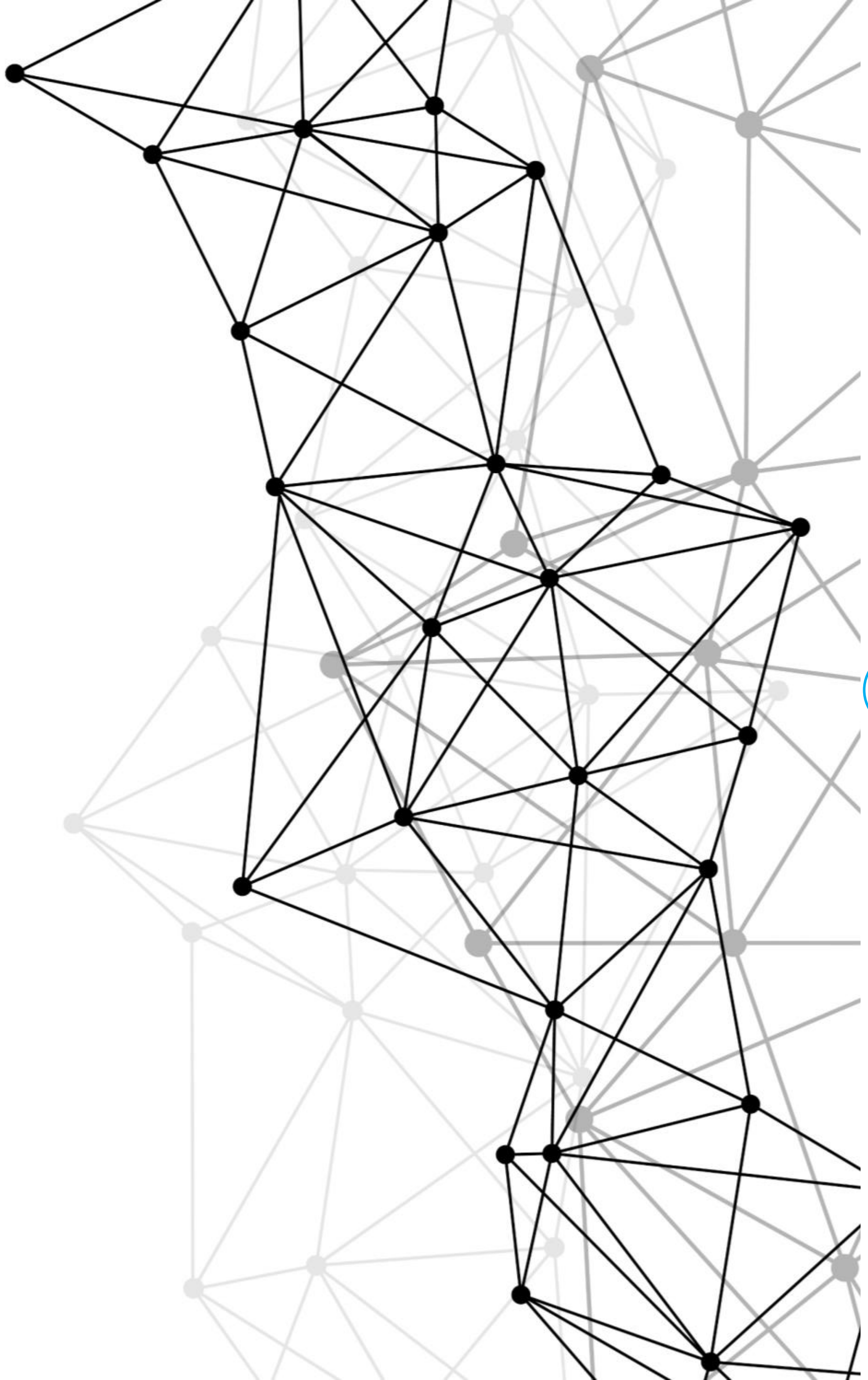
Software

- Improved techniques
- New models
- Toolboxes

PYTORCH
TensorFlow
Caffe MODELS

Content

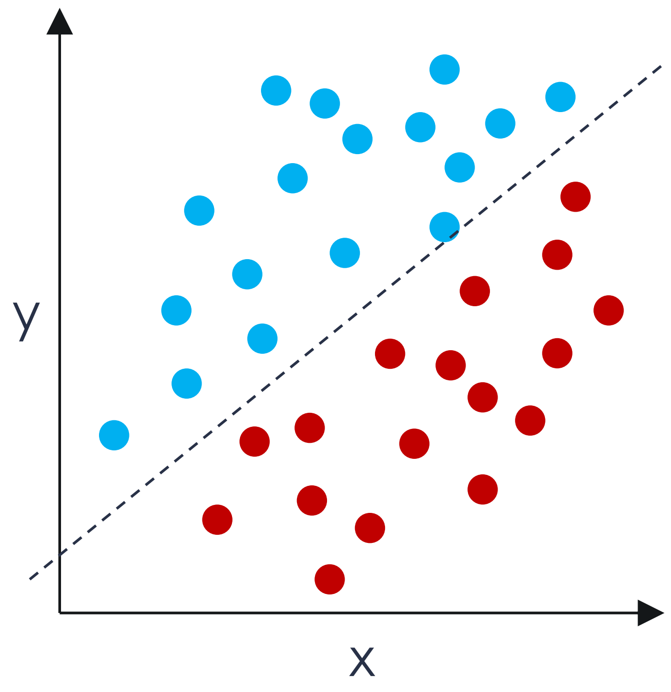
- Introduction
- Perceptron
- Multilayer perceptron (MLP)
- Loss function
- Gradient descent
- Backpropagation
- Convolutional neural network
- Recurrent neural network
- Bias-variance trade-off
- Data



EPFL csem

Perceptron

- Simplest form of a neural network
- Inspired by brain structure and function
- Structural building block of deep learning
- Developed in 1958 by Rosenblatt
- Built based on McCulloch-Pitts model of a neuron
- Supervised learning
- Linear binary classifier



Perceptron

1. Inputs: $\boldsymbol{x} \in \mathbb{R}^m$ with m features
2. Weights and bias

   Strength of connection between units
3. Weighted sum

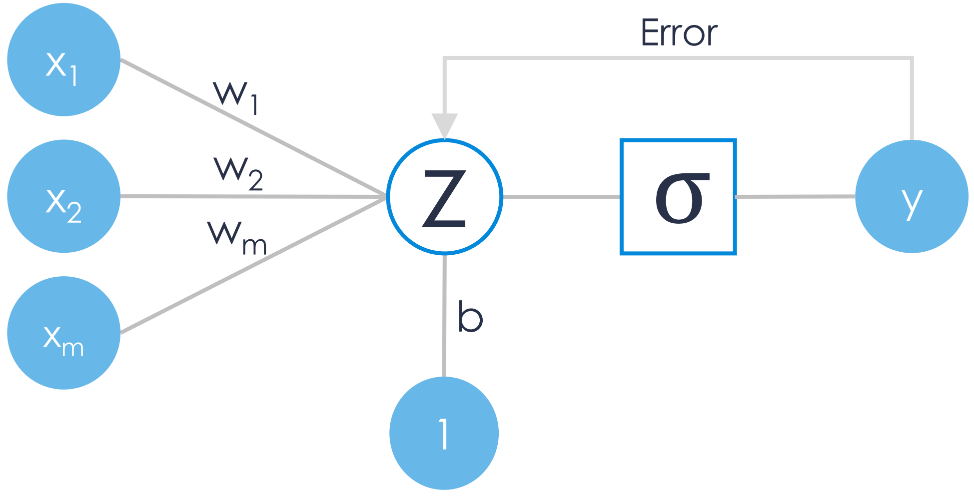
   $z = w_1 \cdot x_1 + \; ... + w_m \cdot x_m + b$

   $z = \sum_1^m w_i \cdot x_i + b$

   $z = \boldsymbol{w}^T \boldsymbol{x} + b$

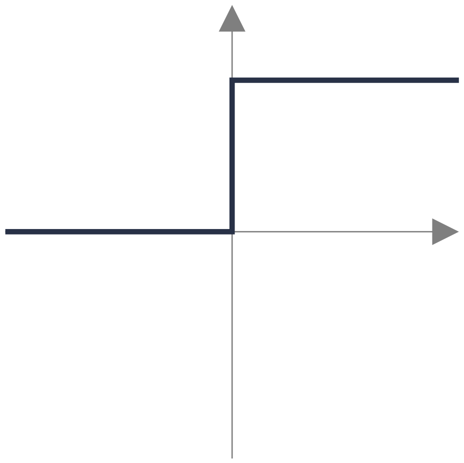
   $\boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix} \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$
4. Activation function

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



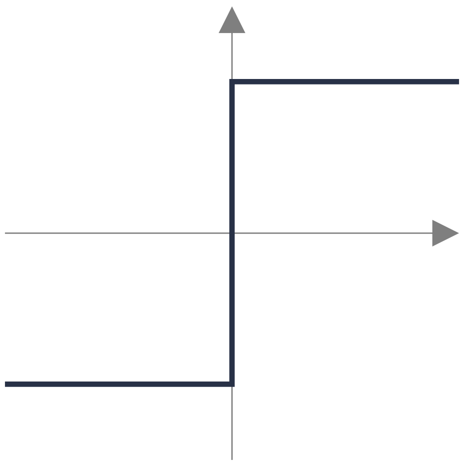
Perceptron – Activation function

- Determine whether neuron fires or not
- Map weighted sum into desired range
- Introduce non-linearity



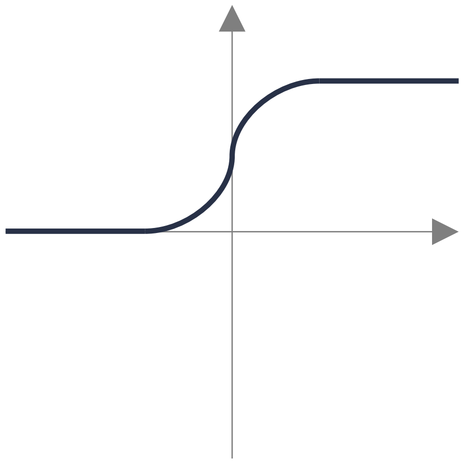
Step function

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



Sign function

$$\sigma(z) = \begin{cases} -1 & if\ z \leq 0 \\ 1 & if\ z > 0 \end{cases}$$

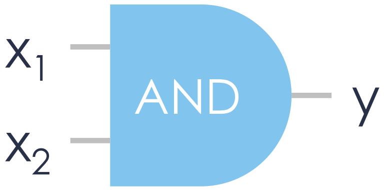


Sigmoid function

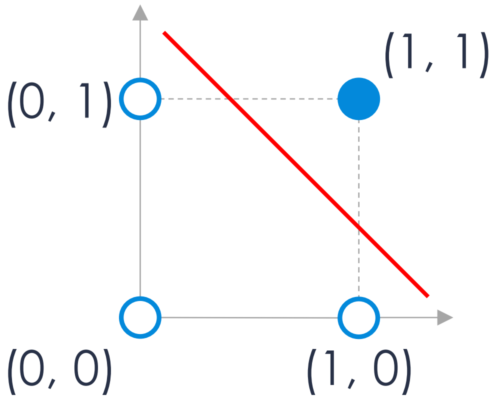
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

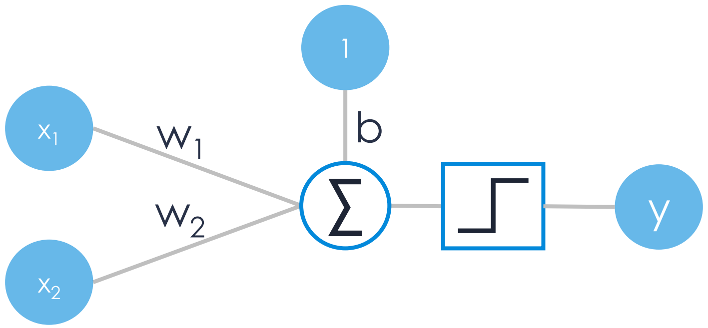
Perceptron example – Logic gates

- Building blocks of digital system - **AND**



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





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

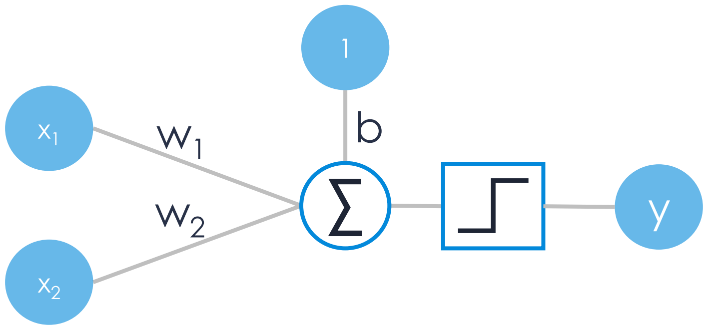
Example with $b = -0.8$, $w_1 = 0.5$, $w_2 = 0.5$

$$x_1 = 0,\ x_2 = 0$$
$$z = 0.5 \cdot 0 + 0.5 \cdot 0 + (-0.8) = -0.8 < 0$$
$$\hat{y} = \sigma(z) = \sigma(-0.8) = 0$$

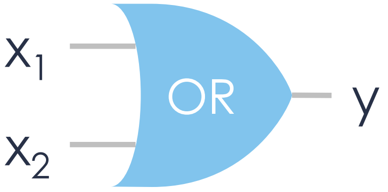
$$x_1 = 0,\ x_2 = 1$$
$$z = 0.5 \cdot 0 + 0.5 \cdot 1 + (-0.8) = -0.3 < 0$$
$$\hat{y} = \sigma(z) = \sigma(-0.3) = 0$$

$$x_1 = 1,\ x_2 = 1$$
$$z = 0.5 \cdot 1 + 0.5 \cdot 1 + (-0.8) = 0.2 > 0$$
$$\hat{y} = \sigma(z) = \sigma(0.2) = 1$$

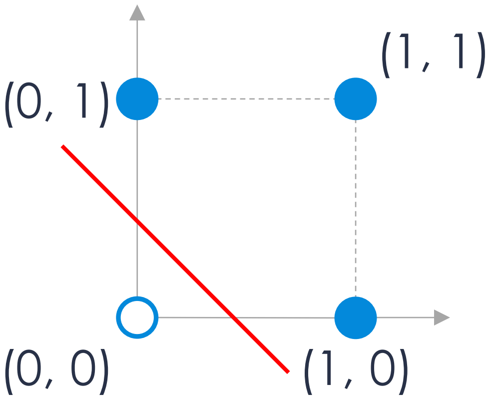
Perceptron example – Logic gates



- Building blocks of digital system - **OR**



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



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

Example with $b = -0.3$, $w_1 = 0.5$, $w_2 = 0.5$

$x_1 = 0,\ x_2 = 0$
$z = 0.5 \cdot 0 + 0.5 \cdot 0 + (-0.3) = -0.3 < 0$
$\hat{y} = \sigma(z) = \sigma(-0.3) = 0$

$x_1 = 0,\ x_2 = 1$
$z = 0.5 \cdot 0 + 0.5 \cdot 1 + (-0.3) = 0.2 > 0$
$\hat{y} = \sigma(z) = \sigma(0.2) = 1$

$x_1 = 1,\ x_2 = 1$
$z = 0.5 \cdot 1 + 0.5 \cdot 1 + (-0.3) = 0.7 > 0$
$\hat{y} = \sigma(z) = \sigma(0.7) = 1$

Perceptron algorithm

- Learn weights to draw decision boundary
- Converge if there exists a separating hyperplane between the two classes

---

Perceptron Learning Algorithm

Input: $D = \{\boldsymbol{x}_j, y_j\}_{j=1}^{n}$ training set of n samples with $\boldsymbol{x}_j \in \mathbb{R}^m$ (m features)

\# Weights and bias initialization

$\boldsymbol{w}^{(0)} = \boldsymbol{0}$

$b^{(0)} = 0$

$t \leftarrow 0$

While *no convergence* **do**

  for every $(\boldsymbol{x}_j, y_j) \in D$ **do**

    \# Compute prediction

    $\hat{y}_j = \sigma\left(\boldsymbol{x}_j{}^T \boldsymbol{w}^{(t)} + b\right)$

    \# Compute error

    $error = y_j - \hat{y}_j$

    \# Update weights and bias

    if $error \mathrel{!}= 0$ **then**

      $\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} + \eta \cdot error \cdot \boldsymbol{x}_j$

      $b^{(t+1)} \leftarrow b^{(t)} + \eta \cdot error$

    else

      $\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)}$

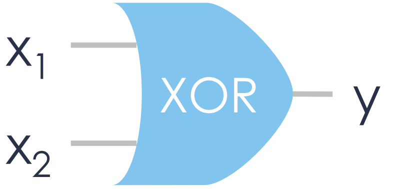
      $b^{(t+1)} \leftarrow b^{(t)}$

  $t \leftarrow t + 1$

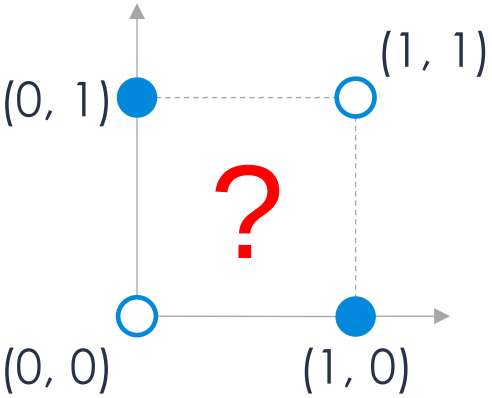
Output: $\boldsymbol{w}$, b

Perceptron limitations

- Only solve linearly separable problems
- Cannot solve XOR problem

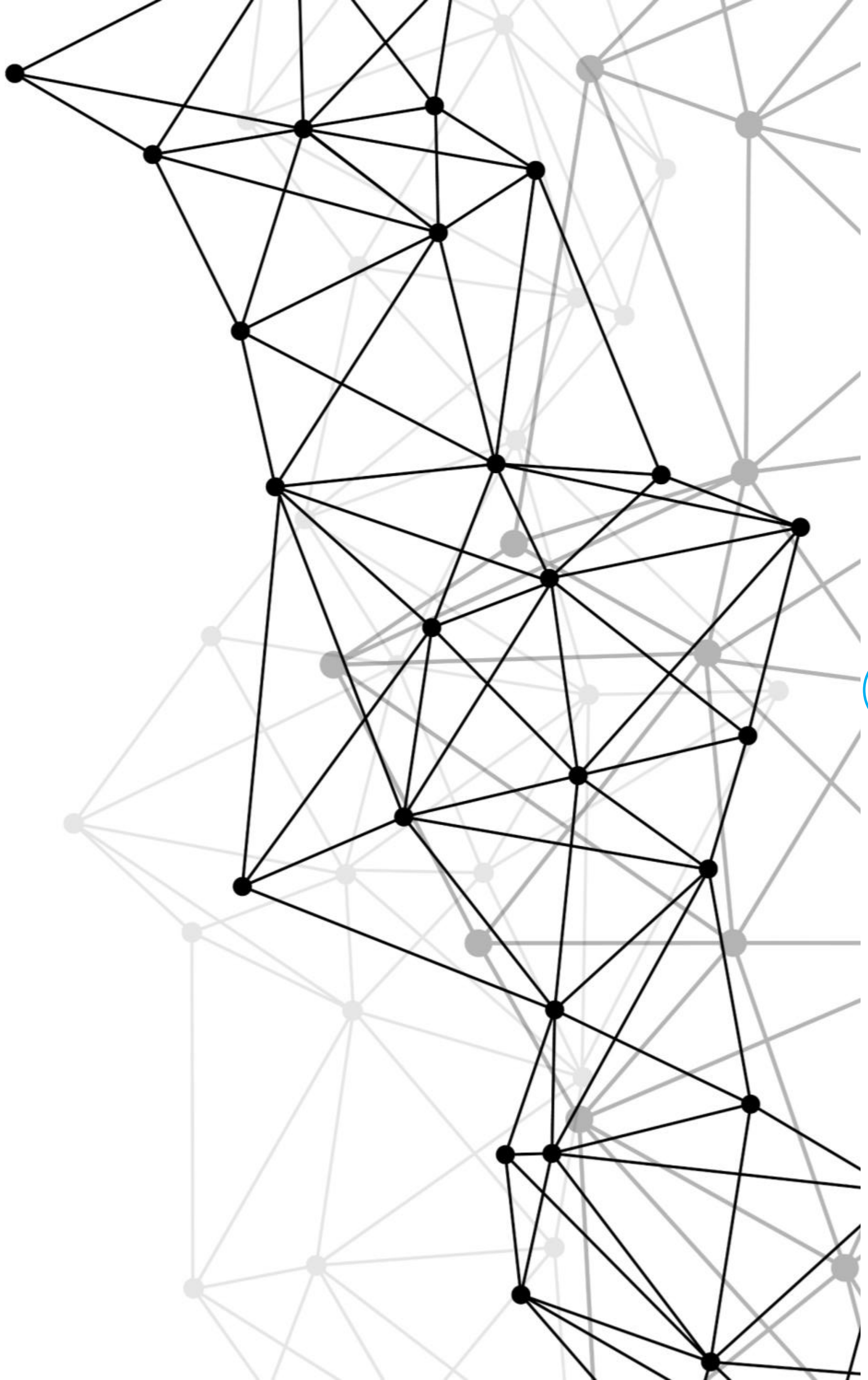


x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



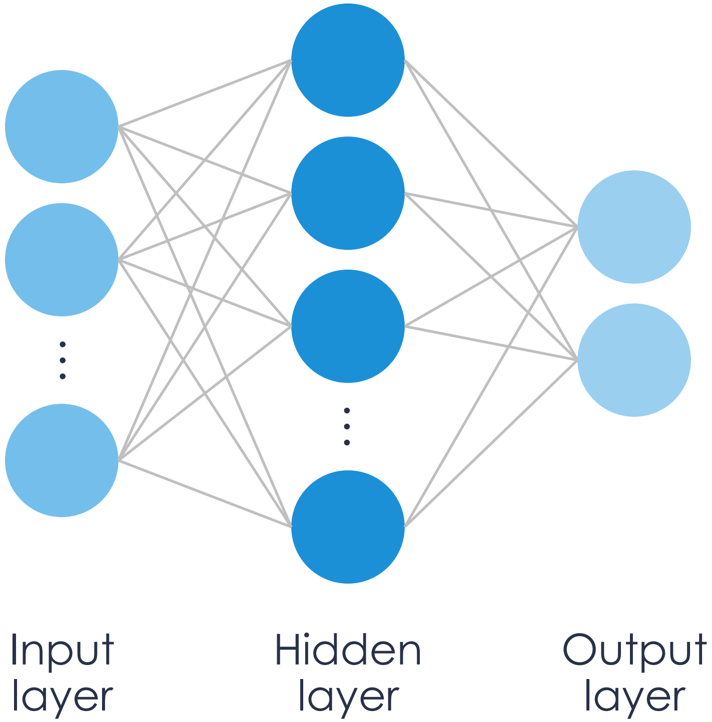
Content

- Introduction
- Perceptron
- Multilayer perceptron (MLP)
- Loss function
- Gradient descent
- Backpropagation
- Convolutional neural network
- Recurrent neural network
- Bias-variance trade-off
- Data



Multilayer perceptron (MLP)

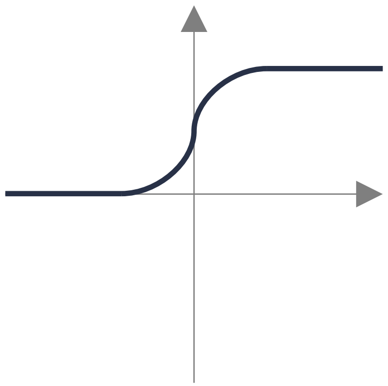
- Solve non-linear problems
- Feedforward artificial neural network
- 3 types of layers:
  - Input layer: Feed in input features
  - Hidden layers: Weighted sum and activation function
  - Output layer
- Each node in a layer is connected to all nodes in next layer
- Each connection has a weight



EPFL csem

MLP – Activation function

- Introduce non-linearity

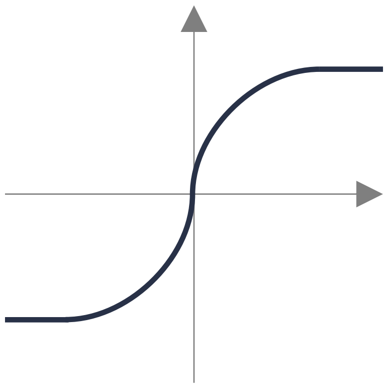


Sigmoid

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

$$\frac{\partial\sigma}{\partial z}(z) = \sigma(z)\big(1-\sigma(z)\big)$$

PyTorch `torch.nn.Sigmoid()`

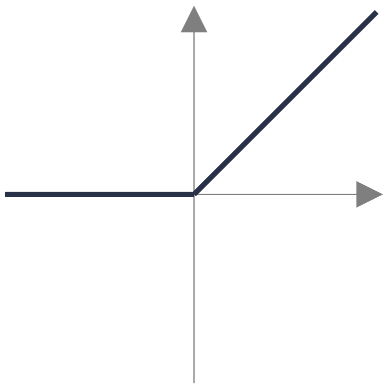


Hyperbolic tangent (tanh)

$$\sigma(z) = \frac{1-e^{-2z}}{1+e^{-2z}}$$

$$\frac{\partial\sigma}{\partial z}(z) = \big(1-\sigma(z)\big)^2$$

PyTorch `torch.nn.Tanh()`

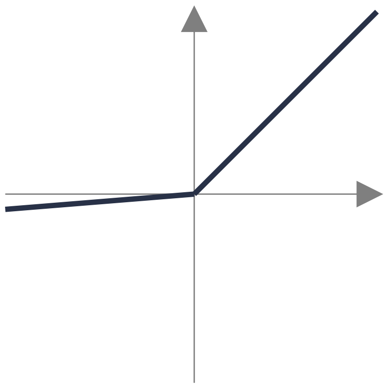


Rectified linear (ReLU)

$$\sigma(z) = \max(0, z)$$

$$\frac{\partial\sigma}{\partial z}(z) = \begin{cases} 0 & if\ z < 0 \\ 1 & if\ z \geq 0 \end{cases}$$

PyTorch `torch.nn.ReLU()`



Leaky ReLU

$$\sigma(z) = \begin{cases} \alpha z & if\ z < 0 \\ z & if\ z \geq 0 \end{cases}$$

$$\frac{\partial\sigma}{\partial z}(z) = \begin{cases} \alpha & if\ z < 0 \\ 1 & if\ z \geq 0 \end{cases}$$

PyTorch `torch.nn.LeakyReLU()`

Content

- Introduction
- Perceptron
- Multilayer perceptron (MLP)
- Loss function
- Gradient descent
- Backpropagation
- Convolutional neural network
- Recurrent neural network
- Bias-variance trade-off
- Data

Loss

- Measure the cost of incorrect predictions
- How close a predicted value is to the actual value → $l(y_i, \hat{y}_i)$
- Guide the model training process towards correct predictions
- Different loss functions for different problems

- **Empirical loss** measures the total loss over the entire dataset
- Also known as objective function, cost function, or empirical risk

$$L(W, b) = \frac{1}{n} \sum_{i=0}^{n} l(y_i, \hat{y}_i)$$

Classification loss function

- Discrete values
  - **Cross entropy loss**
    - Classification with k classes and output probabilities

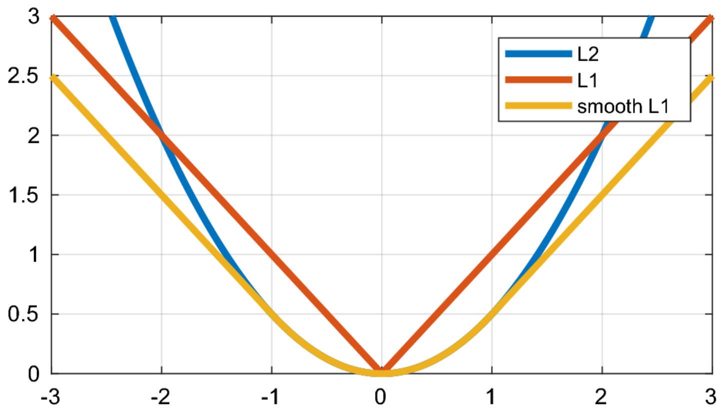
$$l(y_i, \hat{y}_i) = \sum_{j=1}^{k} y_{ij} \cdot log\hat{y}_{ij}$$

    - PyTorch : `torch.nn.functional.cross_entropy(input, target)`
  - **Binary cross entropy loss**
    - Binary classification with output probabilities

$$l(y_i, \hat{y}_i) = -[\, y_i \cdot log\hat{y}_i + (1 - y_i) \cdot log(1 - \hat{y}_i)]$$

    - PyTorch : `torch.nn.functional.binary_cross_entropy(input, target)`

Regression loss functions



- Continuous values
  - **Mean absolute error** (MAE) or L1 loss

$$l(y_i, \hat{y}_i) = |y_i - \hat{y}_i|$$

    - PyTorch : `torch.nn.functional.l1_loss(input, target)`
  - **Mean square error** (MSE) or L2 loss

$$l(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$$

    - Sensitive to outliers
    - PyTorch : `torch.nn.functional.mse_loss(input, target)`
  - **Smooth L1 loss**

$$l(y_i, \hat{y}_i) = \begin{cases} 0.5 \cdot (y_i - \hat{y}_i)^2/\beta & if\, |y_i - \hat{y}_i| < \beta \\ |y_i - \hat{y}_i| - 0.5 \cdot \beta & otherwise \end{cases}$$

    - Combines benefits of MSE loss and MAE loss
    - Beta (β): threshold at which to switch between L1 and L2 loss
    - PyTorch : `torch.nn.functional.smooth_l1_loss(input, target, beta=1.0)`

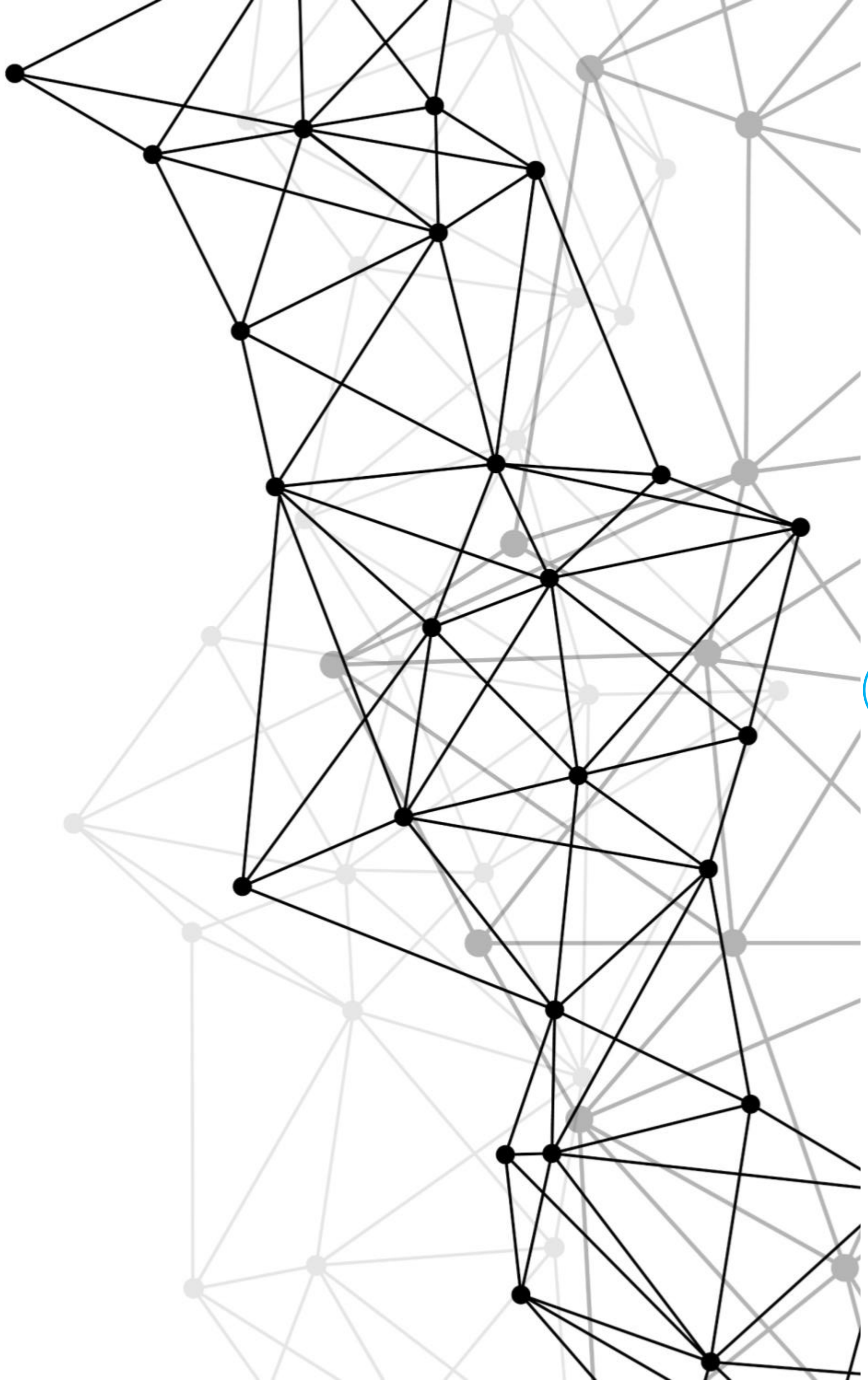
Loss optimization

- Find the network weights and bias that achieve the lowest loss

$$W^*, b^* = argmin_{W,b} \frac{1}{n} \sum_{i=0}^{n} l\big(y_i, f(x_i; W, b)\big) = argmin_{W,b} L(W, b)$$

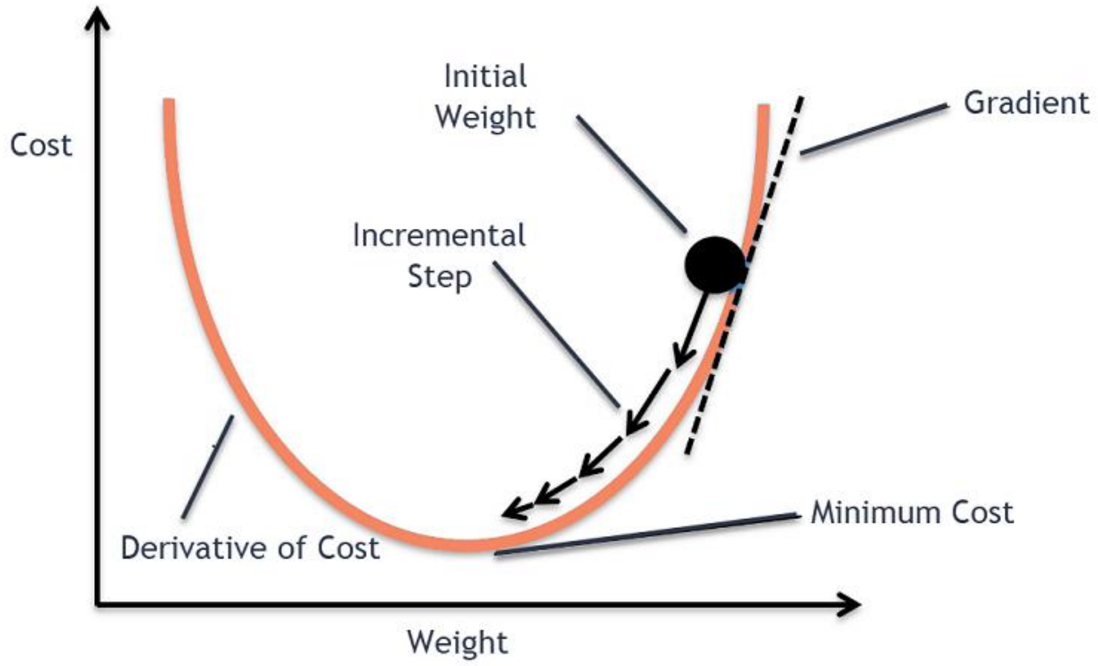
Content

- Introduction
- Perceptron
- Multilayer perceptron (MLP)
- Loss function
- Gradient descent
- Backpropagation
- Convolutional neural network
- Recurrent neural network
- Bias-variance trade-off
- Data



Gradient descent

- Proposed by Cauchy in 1847
- Iterative first-order optimization algorithm
- Iteratively adjusts parameters to minimize the cost function
- Function requirements
  - Differentiable – has a derivative for each point in its domain
  - Convex – line connecting two points does not cross the curve



Gradient descent

1. Initialize parameters (weights and biases) with random values
2. Calculate the empirical loss

$$L(W,b) = \frac{1}{n}\sum_{i=0}^{n} l(y_i, \hat{y}_i)$$

3. Calculate the derivative of the empirical loss (gradient)

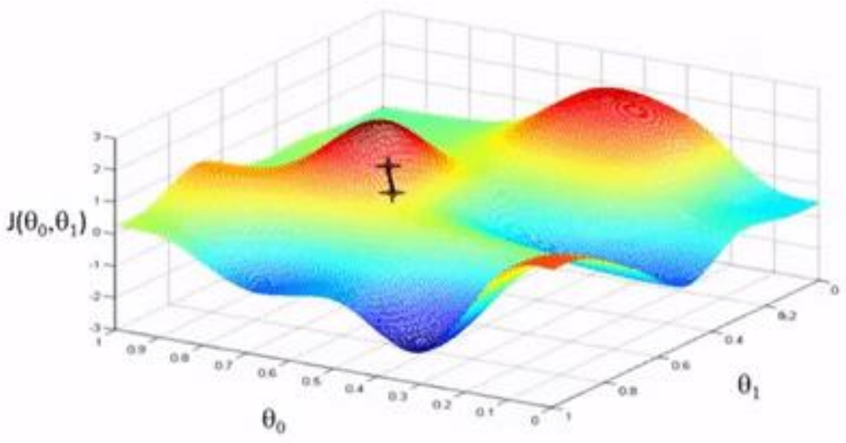
$$\frac{\partial L(W,b)}{\partial W} \text{ and } \frac{\partial L(W,b)}{\partial b}$$

4. Make a step in the opposite direction of the gradient. Update parameters

$$W \leftarrow W - \eta \frac{\partial L(W,b)}{\partial W}$$

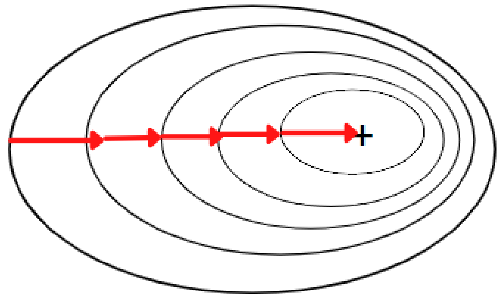
$$b \leftarrow b - \eta \frac{\partial L(W,b)}{\partial b}$$

5. Repeat steps 2, 3 and 4 until convergence

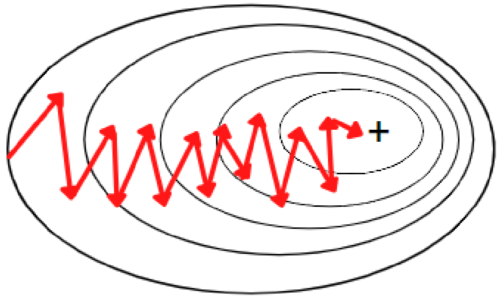


Gradient descent - Variants

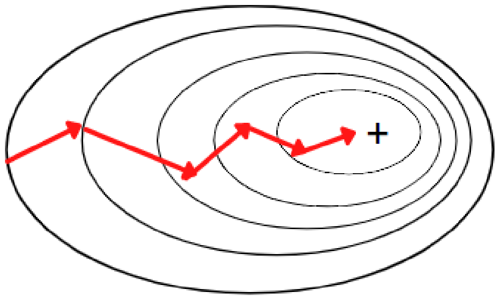
1. Batch gradient descent
   - Update parameters after all training samples
   - Long processing time for large training dataset
   - Stable gradient and convergence but can be stuck in local minimum instead of global one



2. Stochastic gradient descent (SGD)
   - Updates parameters after each training sample
   - Faster but loss in computational efficiency
   - Noisy gradient due to frequent updates but can escape local minimum

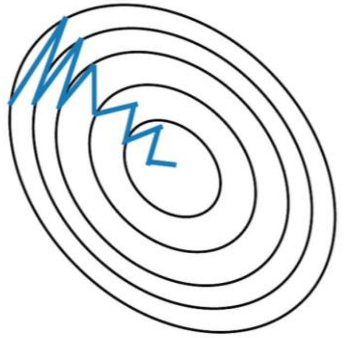


3. Mini-batch gradient descent
   - Combination of batch gradient descent and stochastic gradient descent
   - Split training dataset into small batches and updates parameters on each of these batches.
   - Balance between computational efficiency and speed



Gradient descent - Variants

4. Momentum
   - Reduce high variance in SGD
   - Faster convergence in relevant direction
   - Reduce fluctuation in irrelevant direction
   - Exponential moving average over the past gradients
   - Assign greater weights on most recent values



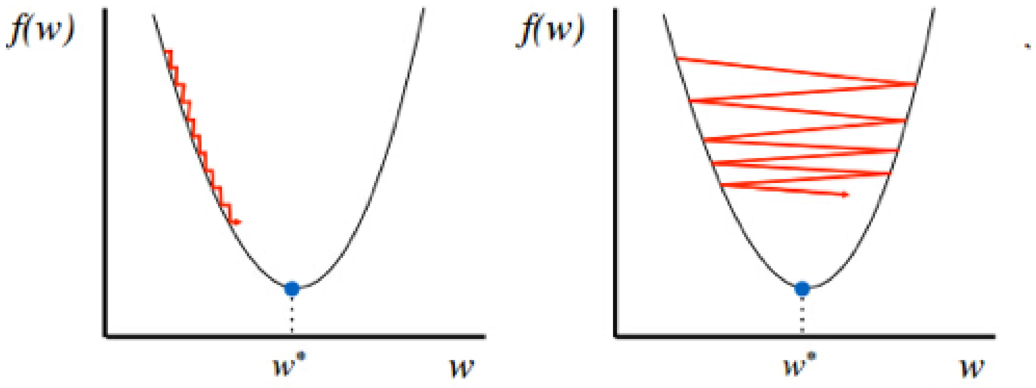
5. Adam (adaptive moment estimation)
   - Works with momentum of first and second order
   - Adaptive learning rate for each parameter
   - Reduce speed to avoid jumping over minimum

Gradient descent - Variants

Variant	Gradient computation	Convergence stability	Computational cost	Noise in updates	Memory usage	Best suited for
Batch GD	Entire dataset	Stable	High	Low	High	Small datasets
Stochastic GD	Single data point	Noisy	Low per iteration	High	Low	Large datasets, online learning
Mini-Batch GD	Small batch of data	Moderate	Moderate	Moderate	Moderate	Most practical applications
Momentum	History + gradient	Stable	Moderate	Reduced	Moderate	Problems with oscillations
ADAM	Adaptive per parameter	Stable	Moderate to high	Low	Moderate	Deep learning, large dataset

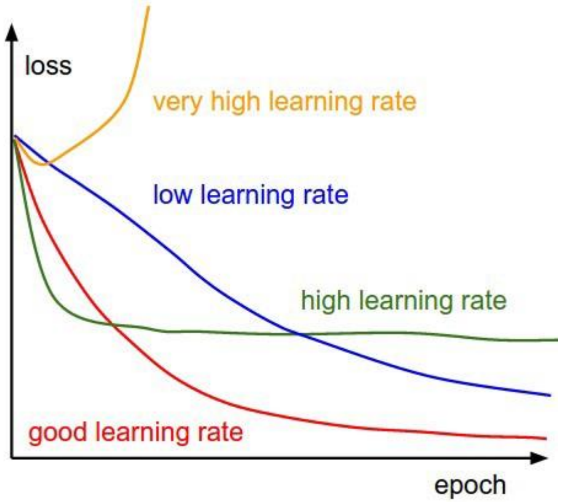
Gradient descent – Learning rate (η)

- Size of steps taken to reach the minimum
- Must be chosen carefully!
- **Too high value** → Large steps, risk of overshooting minimum
- **Too small value** → Small steps, take more time and computations to reach minimum



Too small

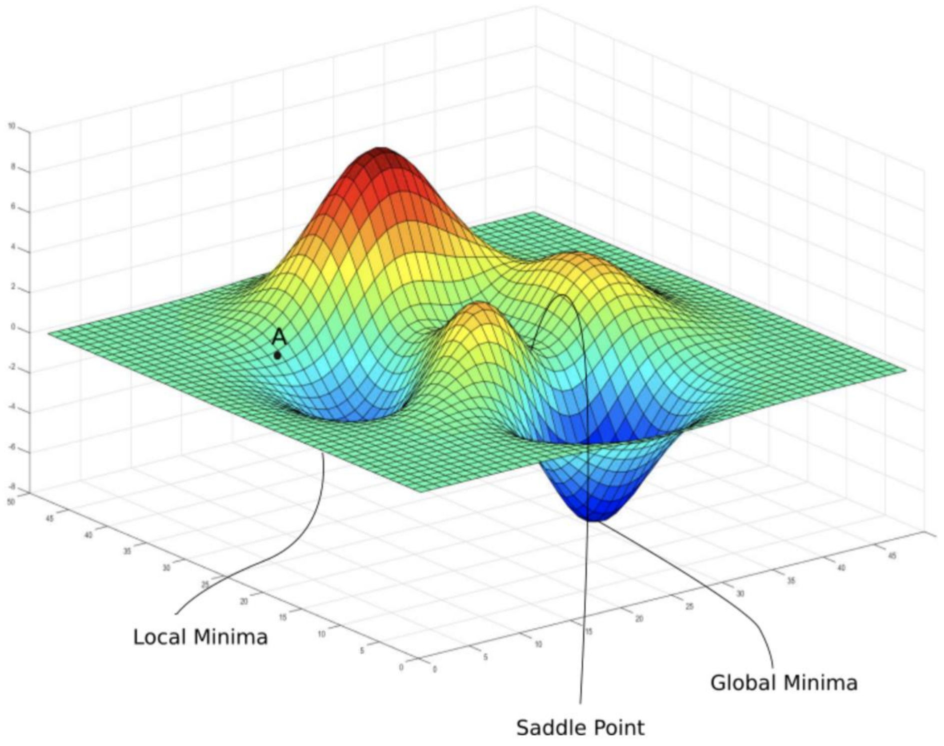
Too large



EPFL csem

Gradient descent – Challenges

1. Local minima and saddle points
   - Learning stops when slope of cost function is at or close to 0
   - Gradient descent can struggle to find the global minimum
2. Vanishing gradient descent
   - Gradients get smaller and smaller, and approach 0
   - Parameters updates become insignificant
   - Stop learning and never converges
3. Exploding gradients
   - Gradients get larger and larger
   - Big parameter updates
   - Gradient descent diverges

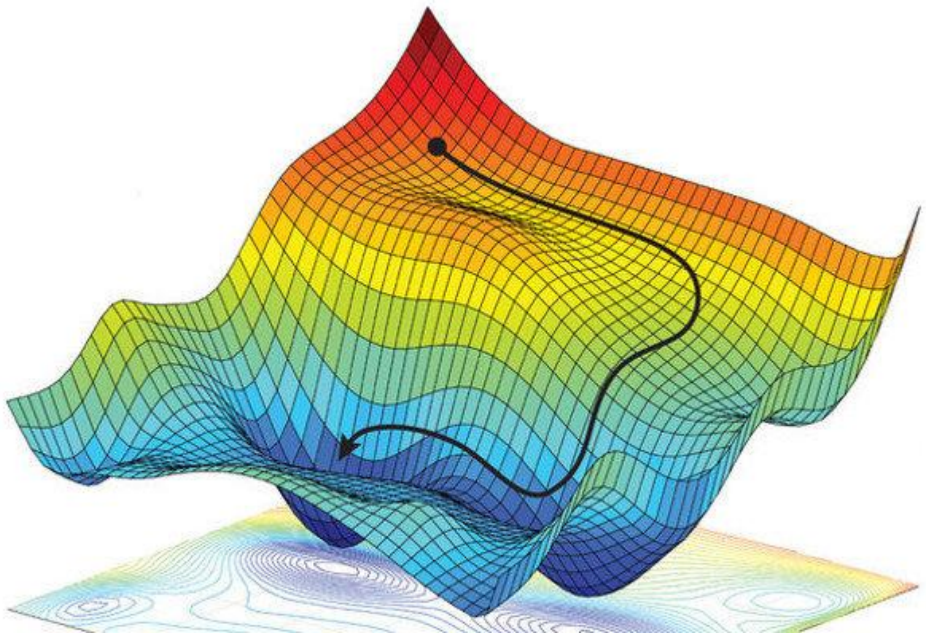


EPFL csem

PyTorch – Gradient-based optimization algorithm

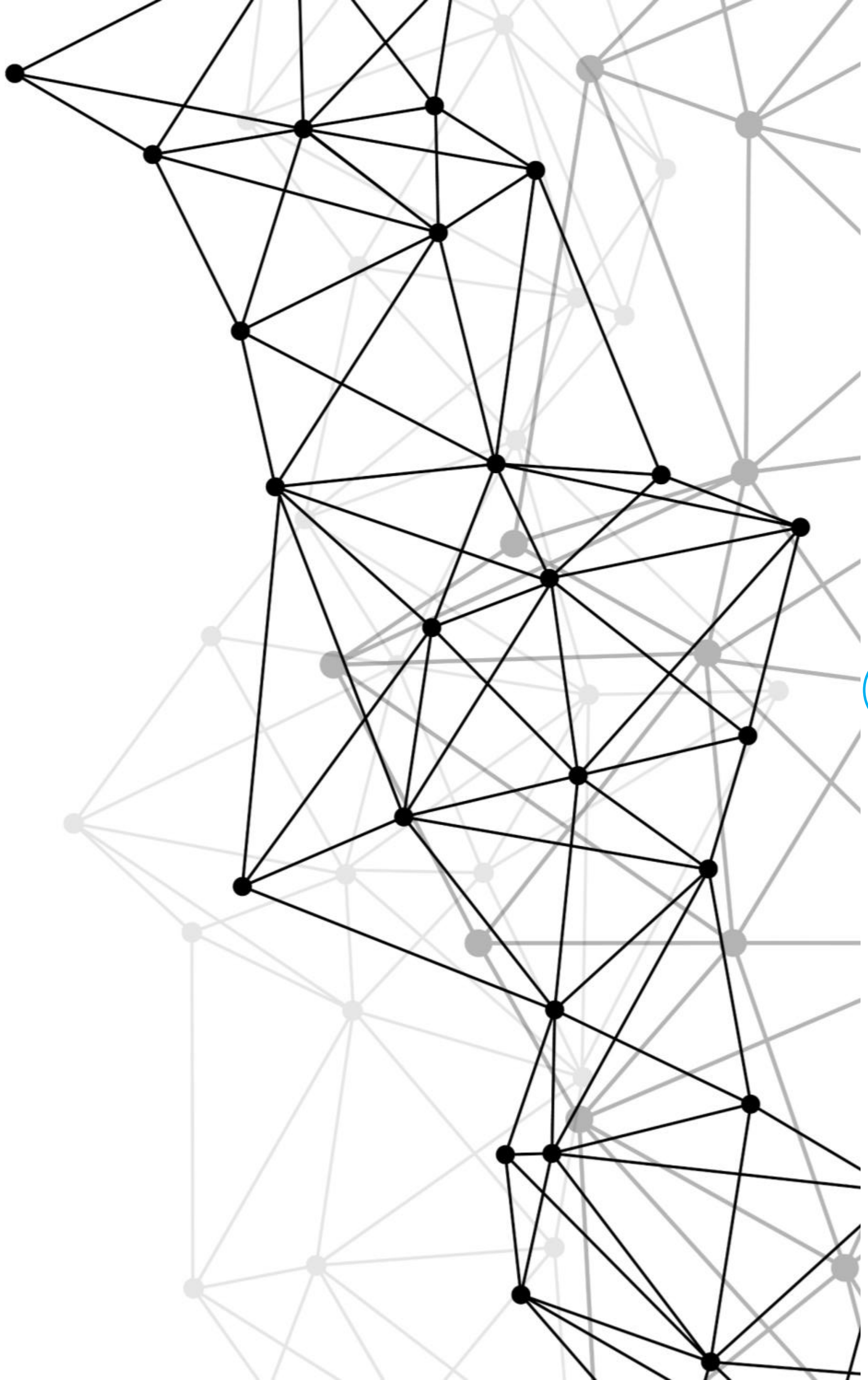
- Stochastic gradient descent (SGD)
  - *`torch.optim.SGD(params, lr)`*
- ADAM
  - *`torch.optim.Adam(params, lr)`*
- Taking an optimization step

```
for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```



Content

- Introduction
- Perceptron
- Multilayer perceptron (MLP)
- Loss function
- Gradient descent
- Backpropagation
- Convolutional neural network
- Recurrent neural network
- Bias-variance trade-off
- Data

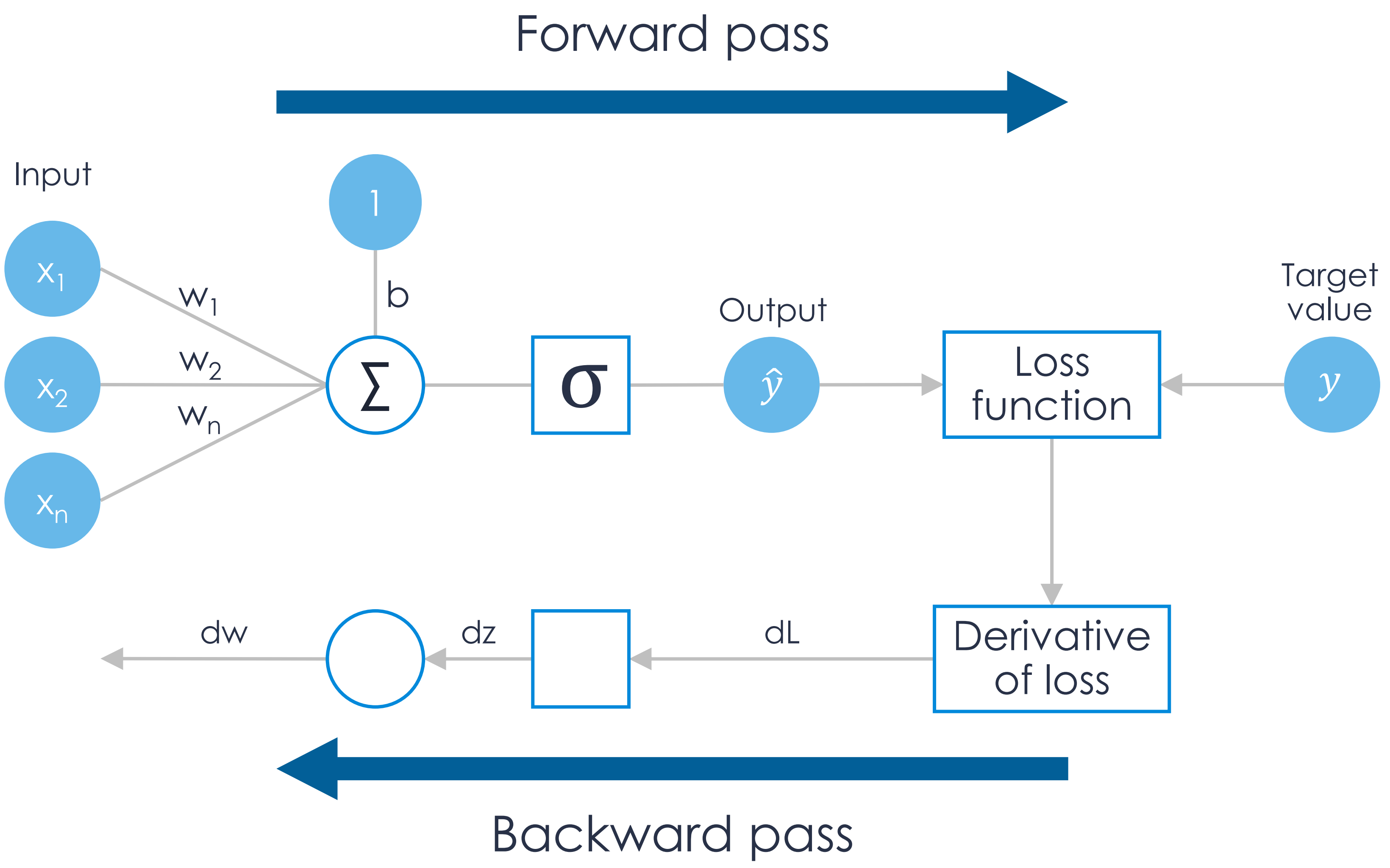


Backpropagation

How to compute the gradient?

- Two main passes
  - Forward pass
    - Compute the predicted values from input data
  - Backward pass
    - Essential part in the training
    - Compare predicted and target values
    - Compute the gradient of the loss function with respect to each parameter
    - Adjust model parameters to decrease error and get closer to target values

Backpropagation



Backpropagation

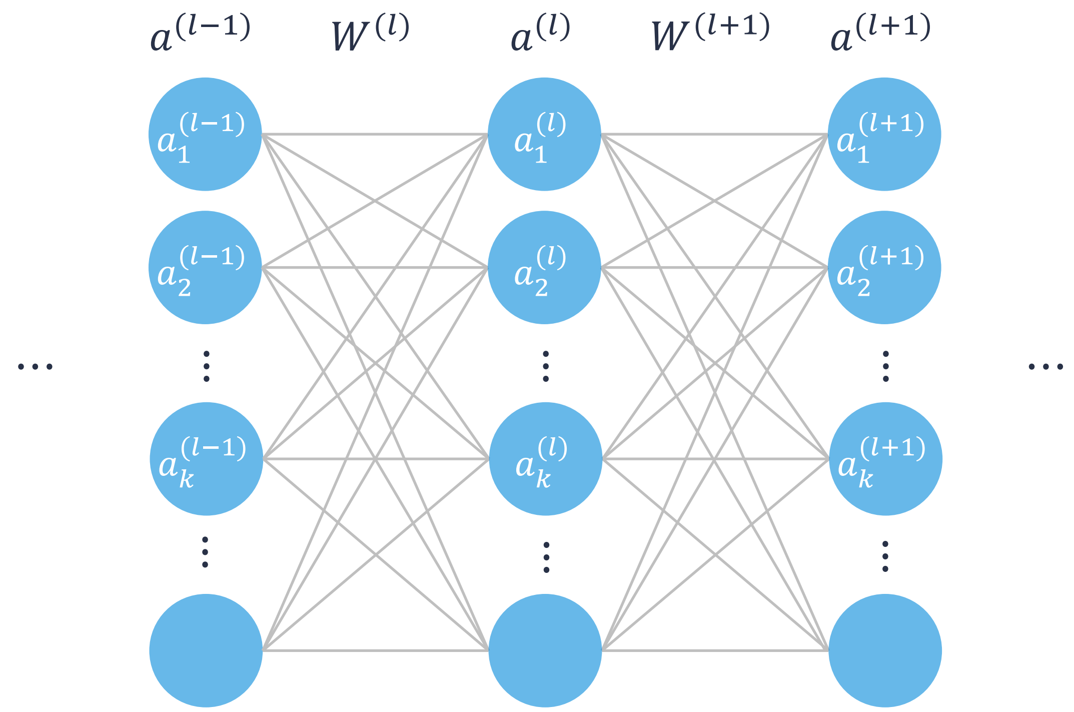
- Forward pass

$$x \xrightarrow{w,b} z^{(1)} \xrightarrow{\sigma} a^{(1)} \xrightarrow{w,b} z^{(2)} \xrightarrow{\sigma} a^{(2)} \longrightarrow \dots \xrightarrow{w,b} z^{(L)} \xrightarrow{\sigma} a^{(L)} = f(x; w, b) = \hat{y}$$

Layer	$\forall\, l = 1, \dots, L$
Weighted sum	$z^{(l)} = w^{(l)} a^{(l-1)} + b^{(l)}$
Activation	$a^{(l)} = \sigma\left(z^{(l)}\right)$

$$a^{(0)} = x$$

$$f(x; w, b) = a^{(L)}$$

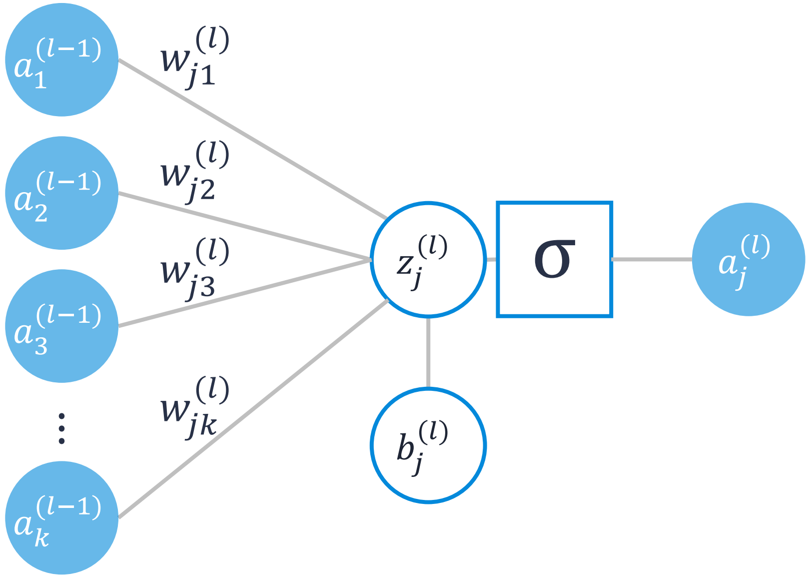


Backpropagation

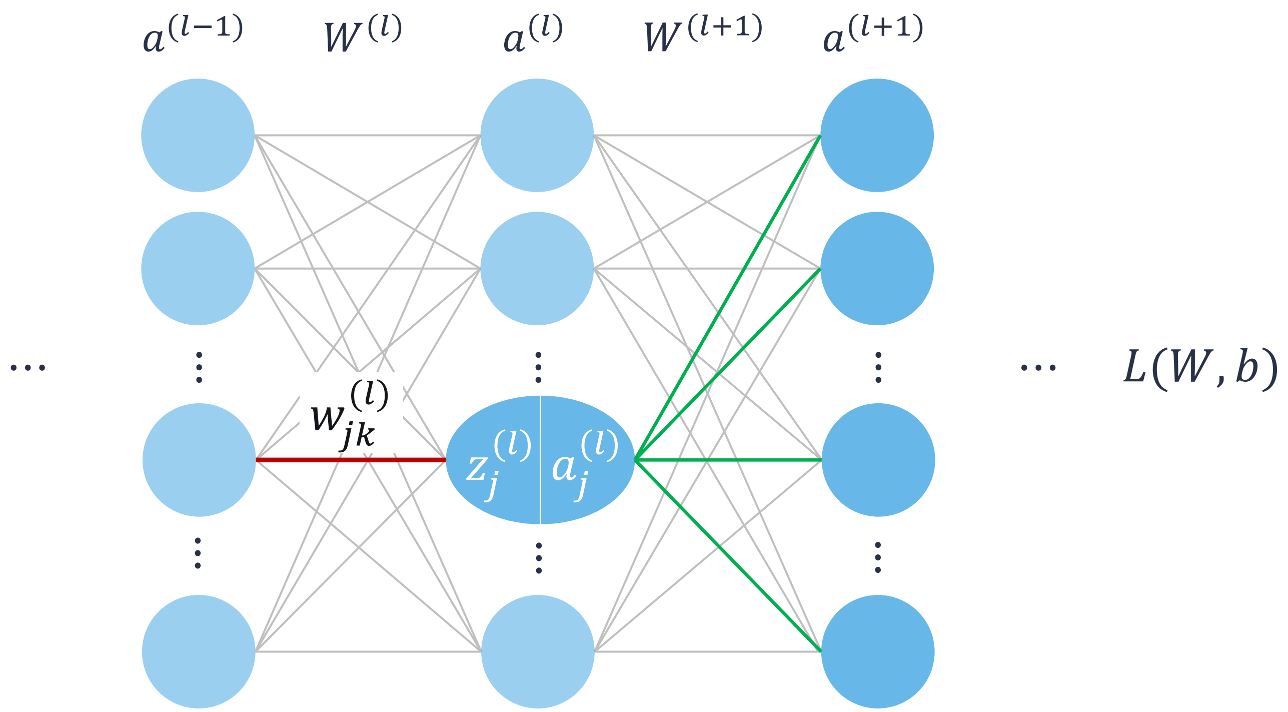
- Forward pass

$$x \xrightarrow{w,b} z^{(1)} \xrightarrow{\sigma} a^{(1)} \xrightarrow{w,b} z^{(2)} \xrightarrow{\sigma} a^{(2)} \longrightarrow \cdots \xrightarrow{w,b} z^{(L)} \xrightarrow{\sigma} a^{(L)} = f(x; w, b) = \hat{y}$$

$$z_j^{(l)} = \sum_k w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}$$
$$a_j^{(l)} = \sigma\left(z_j^{(l)}\right)$$



Backpropagation



Backpropagation – Weight update

- Update one weight $w_{jk}^{(l)}$
- Compute derivative of cost function with respect to this parameter $\frac{\partial L(W,b)}{\partial w_{jk}^{(l)}}$
- Apply **chain rule** to break a large problem into smaller ones

$$\frac{\partial L(W,b)}{\partial w_{jk}^{(l)}} = \frac{\partial L(W,b)}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

$$\frac{\partial L(W,b)}{\partial w_{jk}^{(l)}} = \frac{\partial L(W,b)}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

$$\frac{\partial L(W,b)}{\partial w_{jk}^{(l)}} = \left(\sum_m \frac{\partial L(W,b)}{\partial z_m^{(l+1)}} \frac{\partial z_m^{(l+1)}}{\partial a_j^{(l)}}\right) \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$$

$$\frac{\partial L(W,b)}{\partial w_{jk}^{(l)}} = \left(\sum_m \frac{\partial L(W,b)}{\partial z_m^{(l+1)}} w_{mj}^{(l+1)}\right) \sigma'\left(z_j^{(l)}\right) a_k^{(l-1)}$$

$$z_j^{(l)} = \sum_k w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}$$

$$a_j^{(l)} = \sigma\left(z_j^{(l)}\right)$$

EPFL csem

Backpropagation – Weight update

- Error signal of neuron j in layer l

$$\delta_j^{(l)} = \frac{\partial L(W,b)}{\partial z_j^{(l)}} = \left(\sum\nolimits_m \frac{\partial L(W,b)}{\partial z_m^{(l+1)}} w_{mj}^{(l+1)}\right) \sigma'\left(z_j^{(l)}\right) = \left(\sum\nolimits_m \delta_m^{(l+1)} w_{mj}^{(l+1)}\right) \sigma'\left(z_j^{(l)}\right)$$

$$\frac{\partial L(W,b)}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)}$$

- Update weight

$$w_{jk}^{(l)} \leftarrow w_{jk}^{(l)} - \eta \cdot \delta_j^{(l)}\, a_k^{(l-1)}$$

Backpropagation – Bias update

- Update one bias $b_j^{(l)}$
- Compute derivative of cost function with respect to this parameter $\frac{\partial L(W,b)}{\partial b_j^{(l)}}$
- Apply **chain rule** to break a large problem into smaller ones

$$\frac{\partial L(W,b)}{\partial b_j^{(l)}} = \frac{\partial L(W,b)}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}}$$

$$\frac{\partial L(W,b)}{\partial b_j^{(l)}} = \frac{\partial L(W,b)}{\partial z_j^{(l)}} \cdot 1$$

$$\frac{\partial L(W,b)}{\partial b_j^{(l)}} = \delta_j^{(l)}$$

$$z_j^{(l)} = \sum_k w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}$$

$$\delta_j^{(l)} = \frac{\partial L(W,b)}{\partial z_j^{(l)}}$$

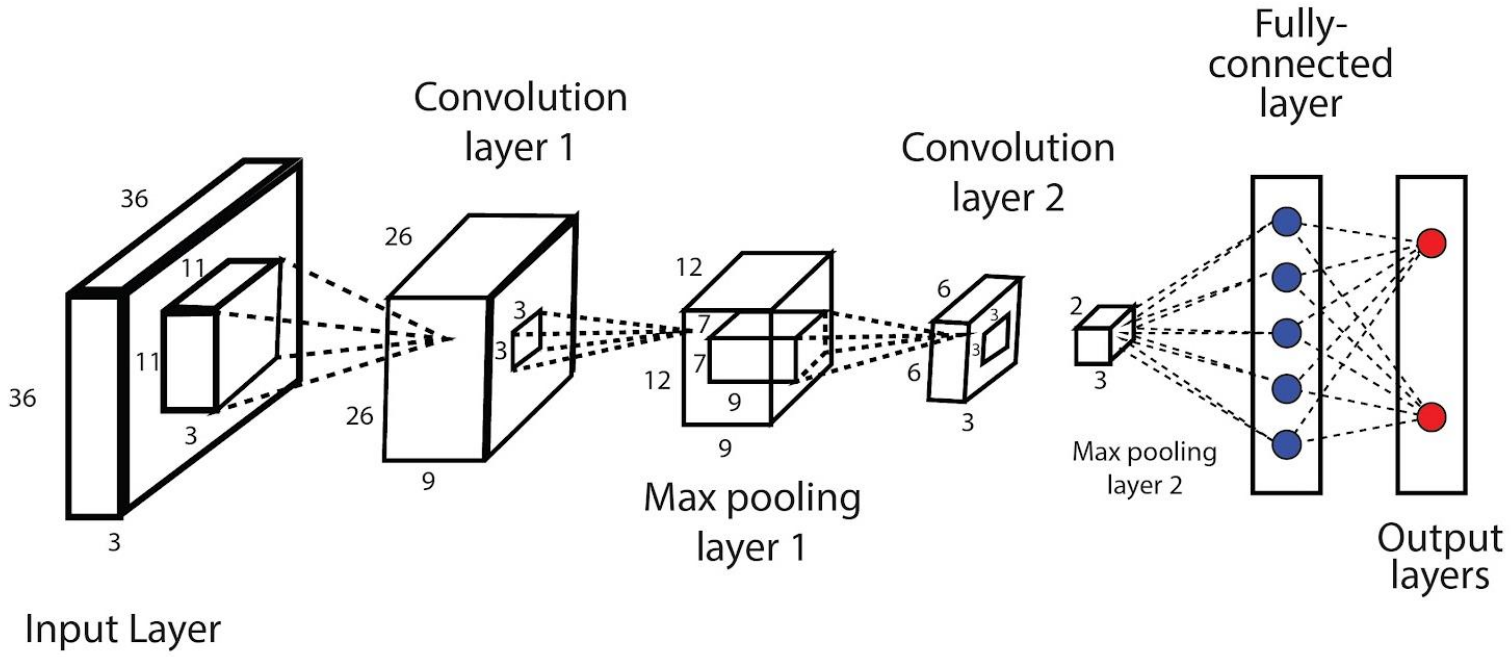
- Update bias $b_j^{(l)} \leftarrow b_j^{(l)} - \eta \cdot \delta_j^{(l)}$

Content

- Introduction
- Perceptron
- Multilayer perceptron (MLP)
- Loss function
- Gradient descent
- Backpropagation
- Convolutional neural network
- Recurrent neural network
- Bias-variance trade-off
- Data

Convolutional neural network (CNN)

- 3 main types of layers
  - Convolutional layers
  - Pooling layers
  - Fully connected layers

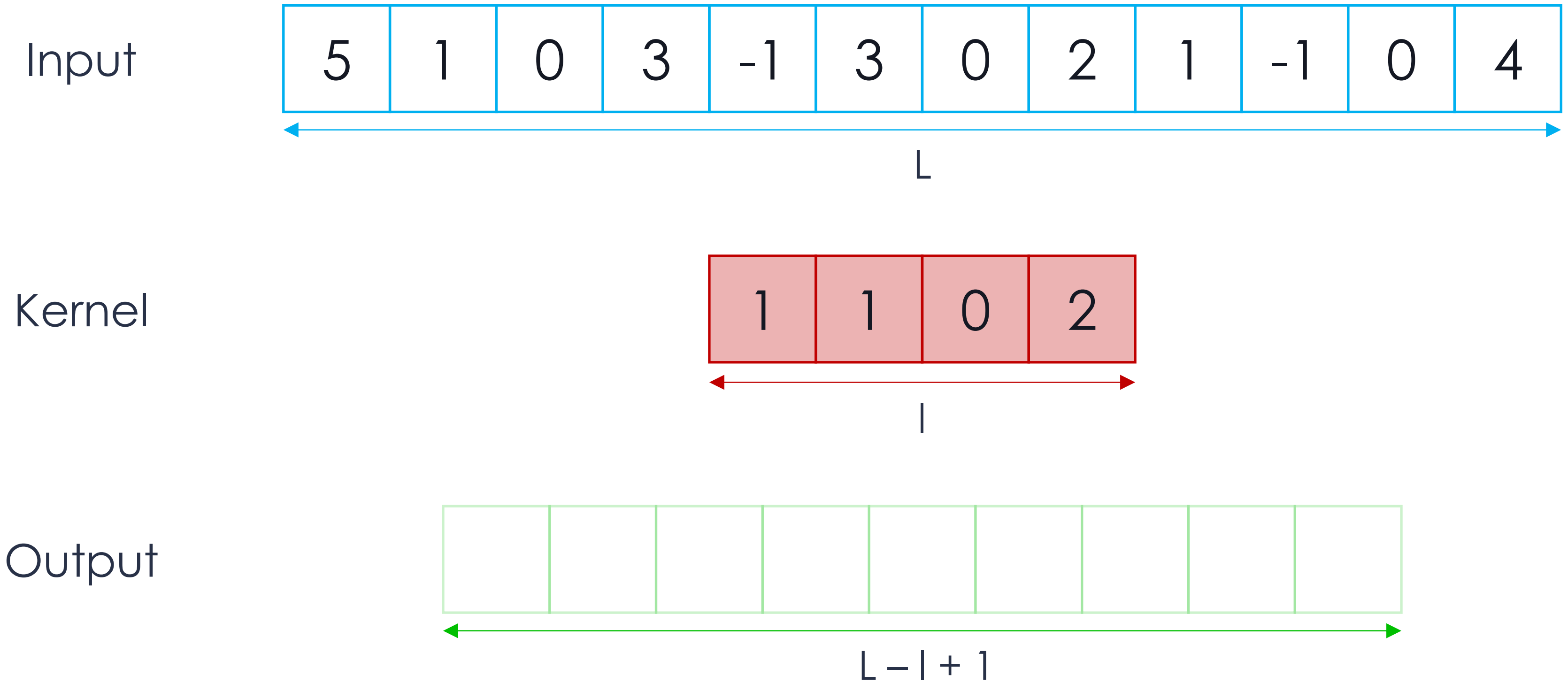


EPFL csem

CNN – Convolutional layer

- Core building block of a CNN
- Use filters that perform convolution operations as it is scanning the input
- Weight sharing
- Generalize to multidimensional input → Convolution 1D, 2D, or 3D
- Preserve signal structure
  - 1D signal remains a 1D signal after convolutional layer
  - 2D signal remains a 2D signal after convolutional layer
  - Etc.
- Kernel → Field of view of the convolution
- Activation map or output feature map
- Receptive field → Area of the input that the kernel can see

CNN – Convolutional layer



CNN – Convolutional layer

Input

3	1	0	3	-1	3	0	2	1	-1	0	4

Kernel

2	0	1	1

$$3 \cdot 2 + 1 \cdot 0 + 0 \cdot 1 + 3 \cdot 1 = 9$$

Output

9								

CNN – Convolutional layer

Input	3	1	0	3	-1	3	0	2	1	-1	0	4

Kernel	2	0	1	1

$$1 \cdot 2 + 0 \cdot 0 + 3 \cdot 1 + (-1) \cdot 1 = 4$$

Output	9	4							

CNN – Convolutional layer

Input

3	1	0	3	-1	3	0	2	1	-1	0	4

Kernel

2	0	1	1

$$0 \cdot 2 + 3 \cdot 0 + (-1) \cdot 1 + 3 \cdot 1 = 2$$

Output

9	4	2						

CNN – Convolutional layer

Input

3	1	0	3	-1	3	0	2	1	-1	0	4

Kernel

2	0	1	1

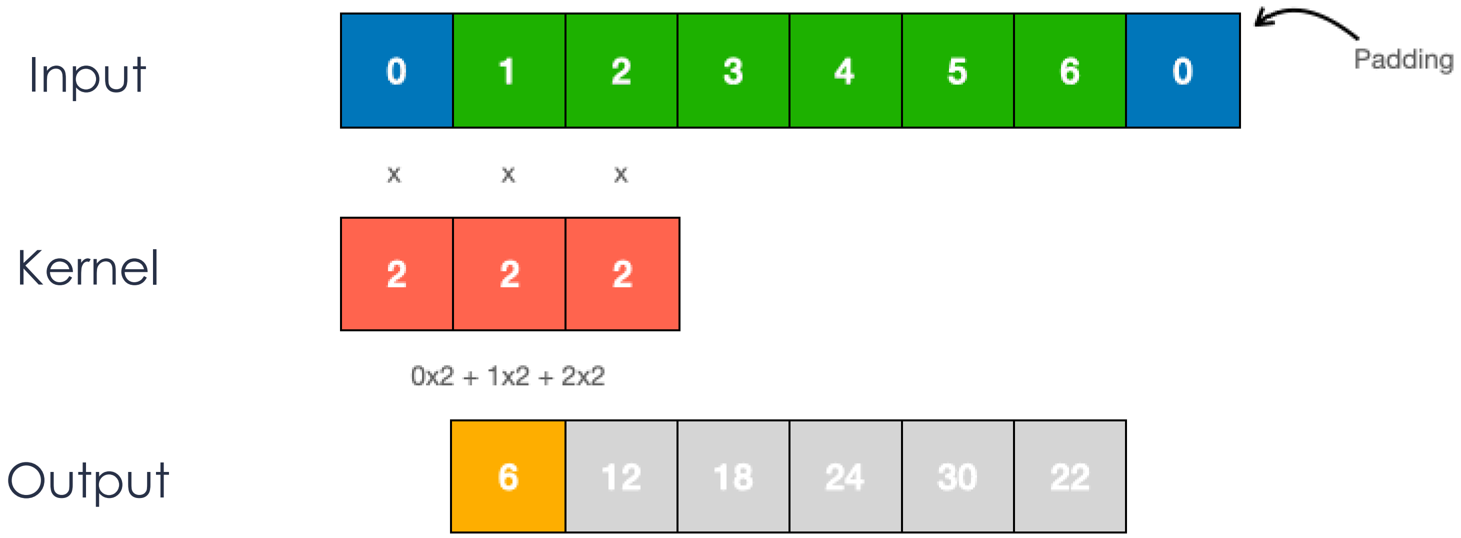
$$1 \cdot 2 + (-1) \cdot 0 + 0 \cdot 1 + 4 \cdot 1 = 2$$

Output

9	4	2	9	0	9	0	3	6

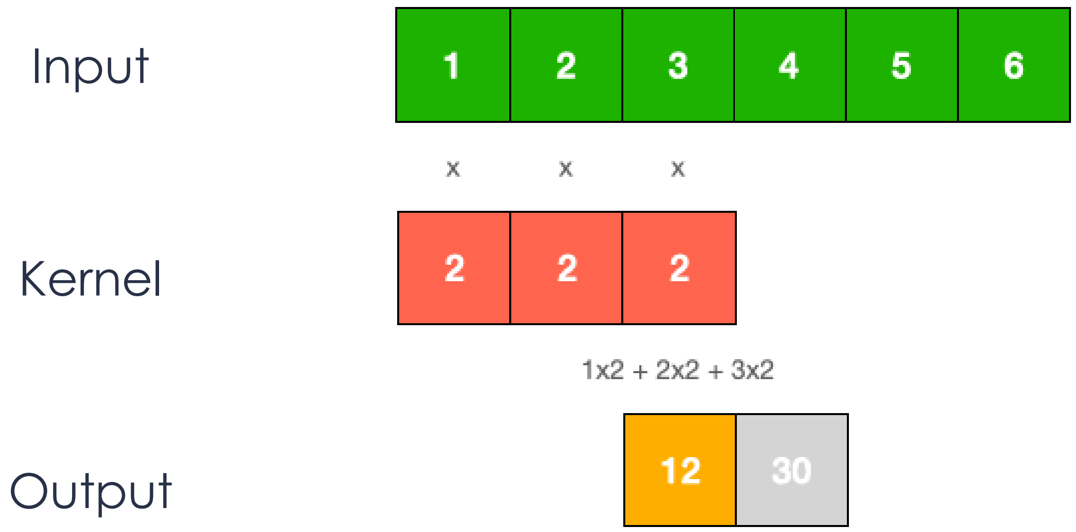
CNN – Padding

- How the border of a sample is handled.
- Padded convolution keeps the output of same length as the input
- Unpadded convolution with kernel >1 crops some border of the sample



CNN – Stride

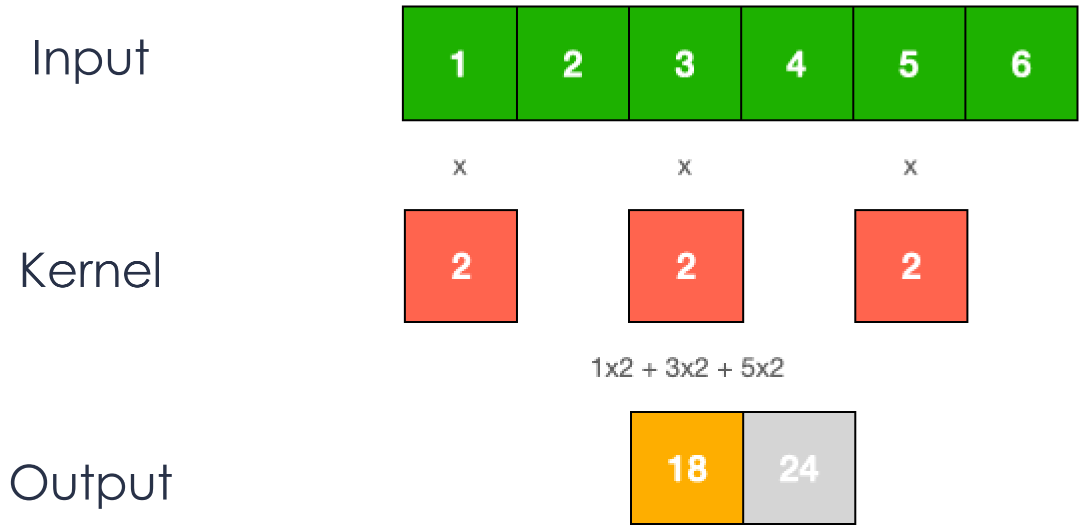
- Step size when moving kernel across signal
- By default, slide the kernel by 1 step a time
- Increase stride to downsample the input vector



EPFL csem

CNN – Dilatation

- Insert spaces between kernel elements
- Expansion of the receptive field while preserving resolution
- Same computational and memory costs



EPFL csem

CNN – Convolutional layer PyTorch

- Convolutional layer 1D
  - `torch.nn.Conv1d(in_channels, out_channels, kernel_size, stride, padding, dilatation)`
  - Input: (N, C_{in}, L_{in})
  - Output: (N, C_{out}, L_{out})

$$L_{out} = \frac{L_{in} + 2 \cdot padding - dilation \cdot (kernel\_size - 1) - 1}{stride} + 1$$

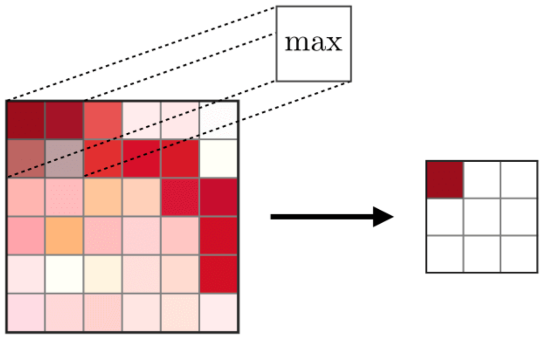
- Convolutional layer 2D
  - `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding, dilatation)`
  - Input: $(N, C_{in}, H_{in}, W_{in})$
  - Output: $(N, C_{out}, H_{out}, W_{out})$

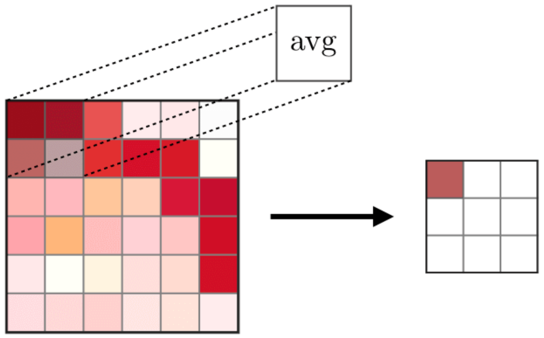
$$H_{out} = \frac{H_{in} + 2 \cdot padding[0] - dilation[0] \cdot (kerne\_size[0] - 1) - 1}{stride[0]} + 1$$

$$W_{out} = \frac{W_{in} + 2 \cdot padding[1] - dilation[1] \cdot (kerne\_size[1] - 1) - 1}{stride[1]}$$

CNN – Pooling layer

- Downsampling operation
- Grouping several activations into a more meaningful one
- Sweep filter across the entire input but no weights
- Typically applied after convolutional layer
- Provide invariance to deformations
- Most common:
  - Max pooling: each operation selects the maximum value of the current view
  - Average pooling: each operation averages the values of the current view





CNN – Maxpooling layer

Input

3	1	0	3	-1	3	0	2	1	-1	0	4

Kernel

$$\max(3, 0) = 3$$

Output

3					

CNN – Maxpooling layer

Input

3	1	0	3	-1	3	0	2	1	-1	0	4

Kernel

$$\max(0, 3) = 3$$

Output

3	3				

EPFL csem

CNN – Maxpooling layer

Input

3	1	0	3	-1	3	0	2	1	-1	0	4

Kernel

$$\max(0, 4) = 4$$

Output

3	3	3	2	1	4

CNN – Pooling layer PyTorch

- Average pooling layer 1D
  - `torch.nn.AvgPool1d(kernel_size, stride, padding)`
  - Input: (N, C_{in}, L_{in})
  - Output: (N, C_{out}, L_{out})

$$L_{out} = \frac{L_{in} + 2 \cdot padding - kernel\_size}{stride} + 1$$

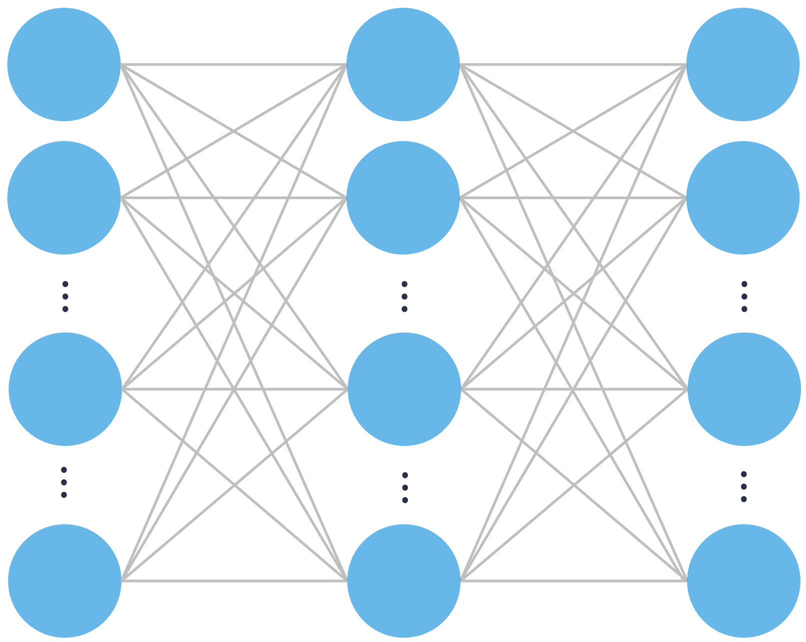
- Average pooling layer 2D
  - `torch.nn. AvgPool2d(kernel_size, stride, padding)`
  - Input: $(N, C_{in}, H_{in}, W_{in})$
  - Output: $(N, C_{out}, H_{out}, W_{out})$

$$H_{out} = \frac{H_{in} + 2 \cdot padding[0] - kernel\_size[0]}{stride[0]} + 1$$

$$W_{out} = \frac{W_{in} + 2 \cdot padding[1] - kernel\_size[1]}{stride[1]} + 1$$

CNN – Fully-connected layer

- Operates on flattened input
- Every input influences every output
- At the end of CNN architecture
- PyTorch
  - *torch.nn.Linear(in_features, out_features)*
  - Input: (N, H_{in})
  - Output: (N, H_{out})



EPFL csem

CNN – PyTorch

```python
import torch.nn as nn


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.cnn = nn.Sequential(
            nn.Conv1d(1, 8, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool1d(2),

            nn.Conv1d(8, 16, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool1d(2),

            nn.Conv1d(16, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool1d(2),
        )
        self.fc = nn.Sequential(
            nn.Linear(512, 1),
        )

    def forward(self, x):
        x = self.cnn(x)
        # Flatten
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

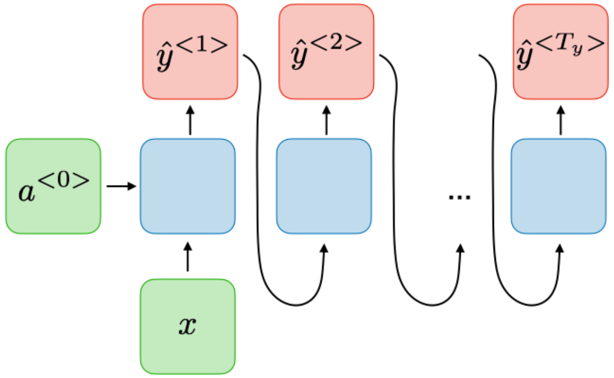
Content

- Introduction
- Perceptron
- Multilayer perceptron (MLP)
- Loss function
- Gradient descent
- Backpropagation
- Convolutional neural network
- Recurrent neural network
- Bias-variance trade-off
- Data

Recurrent neural network (RNN)

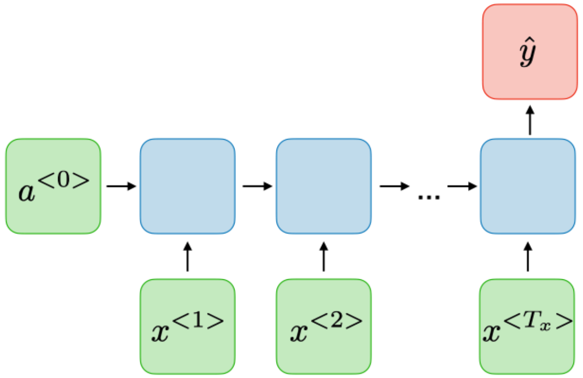
- Problems dealing with sequential or time series data
- Capture temporal representation
- Typical applications: language translation, natural language processing, speech recognition, image captioning
- Take into account historical information
- Possibility of processing input of any length
- Model size not increasing with input size
- Weights are shared across time

RNN different applications



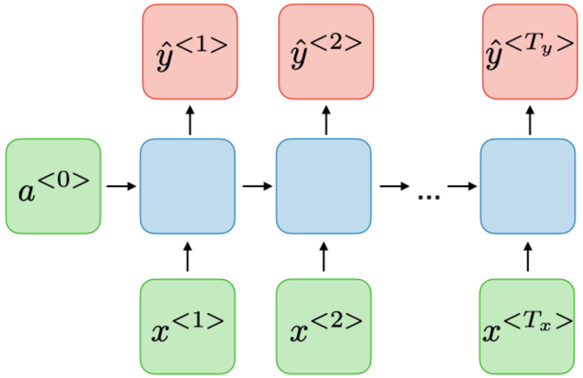
One-to-many

$T_x = 1, T_y > 1$



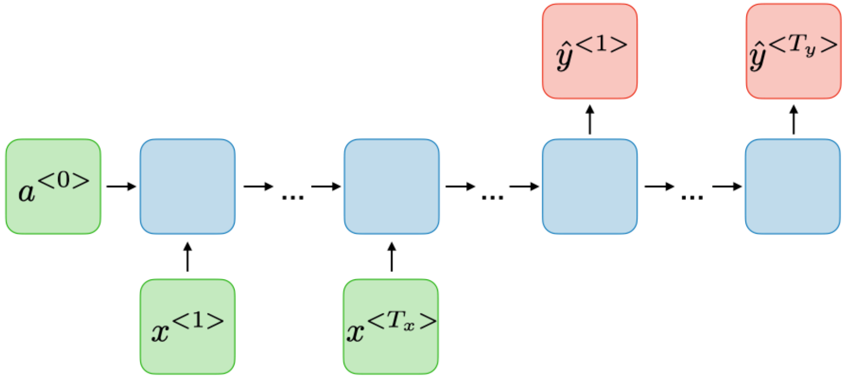
Many-to-one

$T_x > 1, T_y = 1$



Many-to-many

$T_x = T_y$



Many-to-many

$T_x \neq T_y$

RNN

- Empirical loss
  - Loss of all time steps defined based on loss at every time step

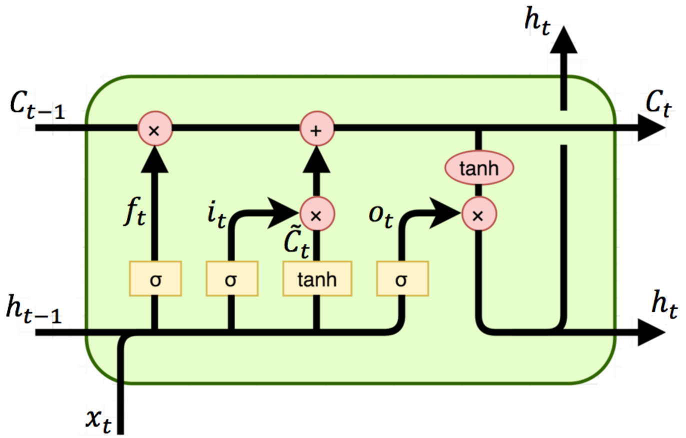
$$L(y,\hat{y}) = \sum_{t=1}^{T_y} L(y^{<t>},\hat{y}^{<t>})$$

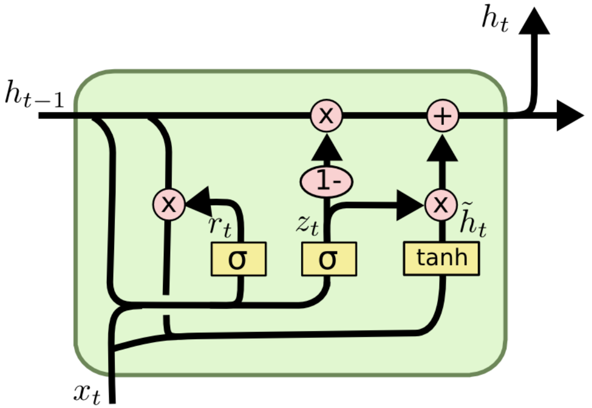
- Backpropagation through time
  - Done at each point in time

$$\frac{\partial L^{(T)}}{\partial W} = \sum_{t=1}^{T} \frac{\partial L}{\partial W}\bigg|_{(t)}$$

RNN variants

- Long Short-Term Memory units (LSTM)
  - Address the problem of long-term dependency
  - Internal memory cell
  - 3 gates (input gate, output gate and forget gate) to control the flow of information
- Gated Recurrent Unit (GRU)
  - Fewer parameters and faster training
  - 2 gates (reset gate and update gate) to control how much and which information to retain

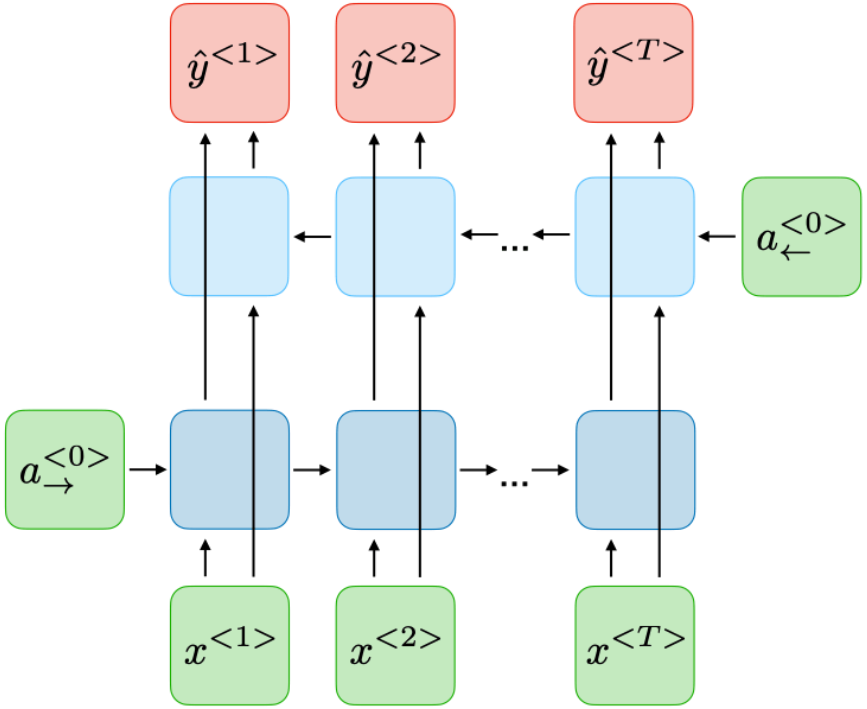




EPFL csem

Bidirectional RNN

- Learning not limited to past and present
- Train the network in two opposite directions
  - From beginning to end of a sequence
  - From end to beginning of a sequence



RNN – PyTorch

- LSTM
  - `torch.nn.LSTM(input_size, hidden_size, num_layers, bidirectional)`
  - Input: (L, N, H_{in})
  - h_0: $(D * num\_layers, H_{out})$
  - c_0: $(D * num\_layers, H_{cell})$
  - Output: $(L, N, D * H_{out})$
- GRU
  - `torch.nn.GRU(input_size, hidden_size, num_layers, bidirectional)`
  - Input: (L, N, H_{in})
  - h_0: $(D * num\_layers, H_{out})$
  - Output: $(L, N, D * H_{out})$

N: batch size
L: sequence length
D: 2 if bidirectional and 1 otherwise
H_{in}: input_size
H_{cell}: hidden_size
H_{out}: hidden_size

RNN – PyTorch

```python
import torch.nn as nn


class RnnModel(nn.Module):

    def __init__(self, input_size, output_size, hidden_size=64):
        super().__init__()
        self.input_size = input_size
        self.output_size = output_size
        self.hidden_size = hidden_size
        self.recurrent_layer = nn.GRU(
            input_size=1,
            hidden_size=self.hidden_size,
            num_layers=1,
            batch_first=False,
        )
        self.output_layer = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, x):
        y, h = self.recurrent_layer(x)
        return self.output_layer(y[-1])
```

Content

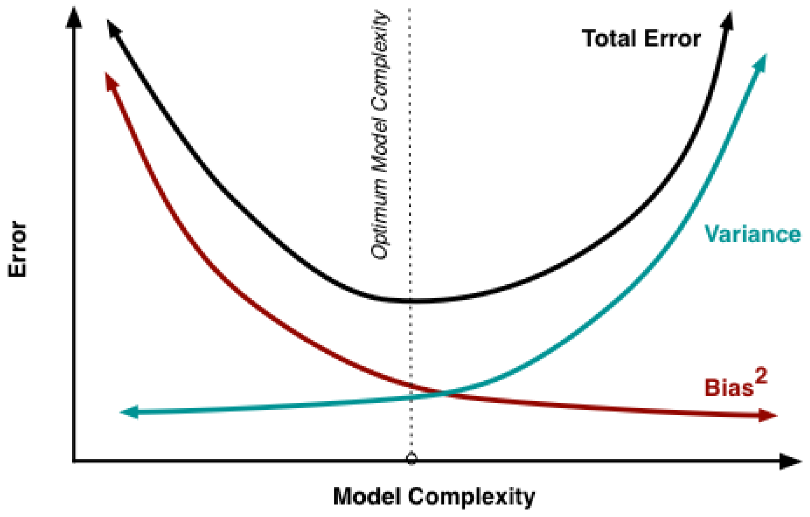
- Introduction
- Perceptron
- Multilayer perceptron (MLP)
- Loss function
- Gradient descent
- Backpropagation
- Convolutional neural network
- Recurrent neural network
- Bias-variance trade-off
- Data

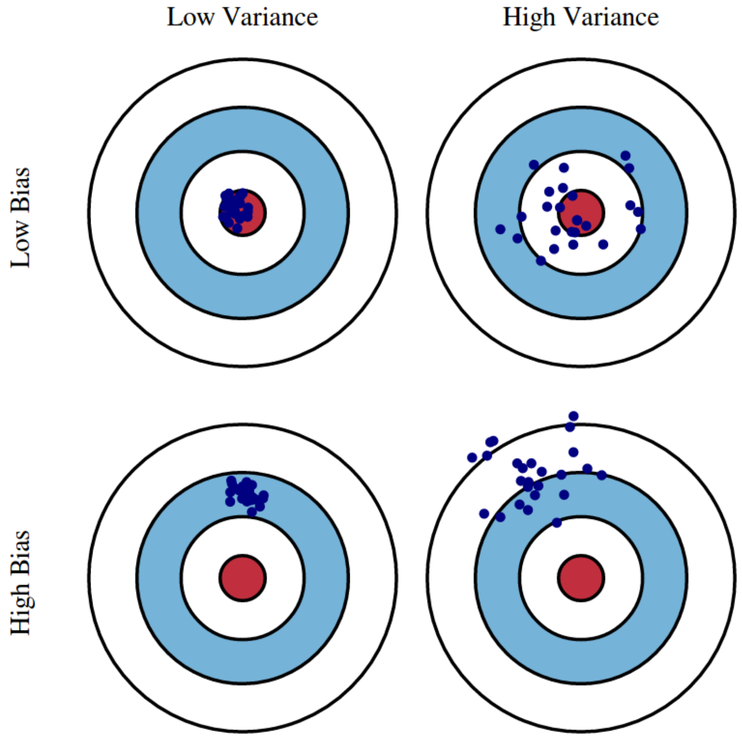
Bias-variance trade-off

- Bias → Difference between average prediction of model and correct value
- Variance → How much an estimate varies around its average
- Simple model → High bias and low variance
- Complex model → Low bias and high variance

$$Err(x) = \left(E[\hat{f}(x)] - f(x)\right)^2 + E\left[\left(\hat{f}(x) - E[\hat{f}(x)]\right)^2\right] + \sigma_e^2$$

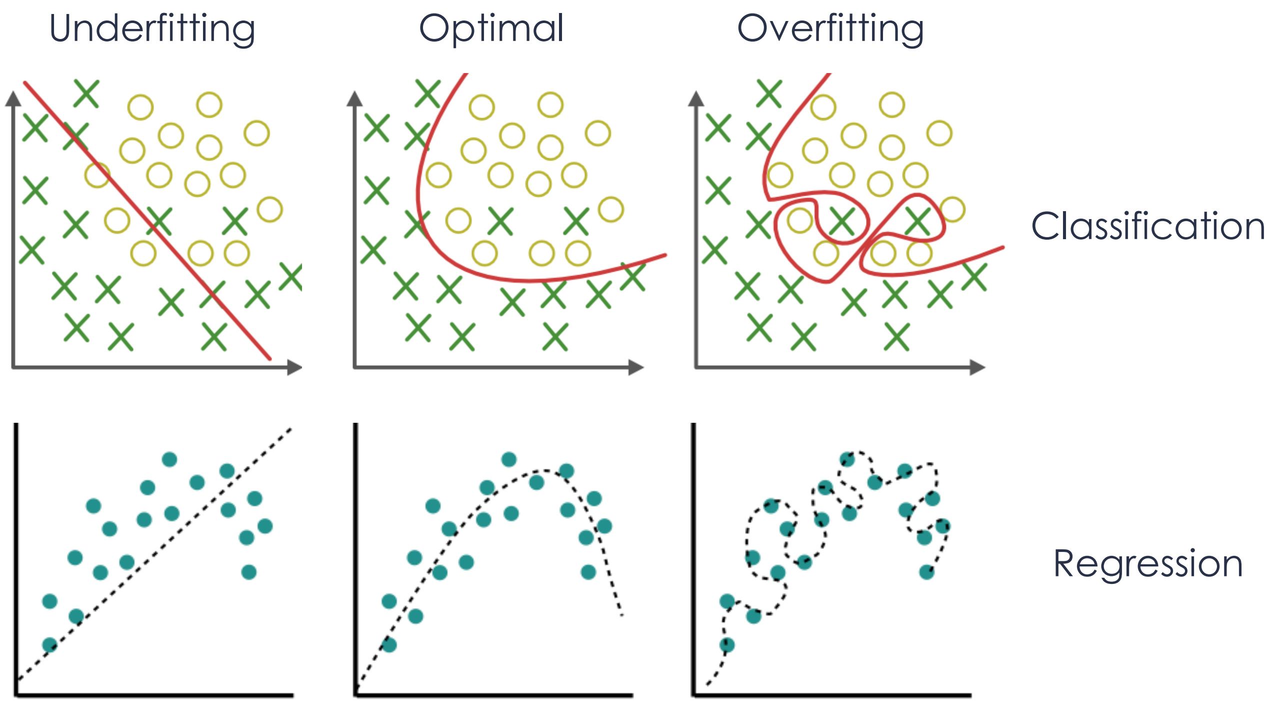
$$Err(x) = bias^2 + variance + noise$$





Bias-variance trade-off

- Underfitting
  - Model is unable to capture the underlying pattern of the data
  - High bias and low variance
  - Small amount of data, or linear model for non-linear data
  - High error in training and test sets
- Overfitting
  - Model captures the noise with the underlying pattern in data
  - Low bias and high variance
  - Complex model
  - Low error in training set and high error in test set



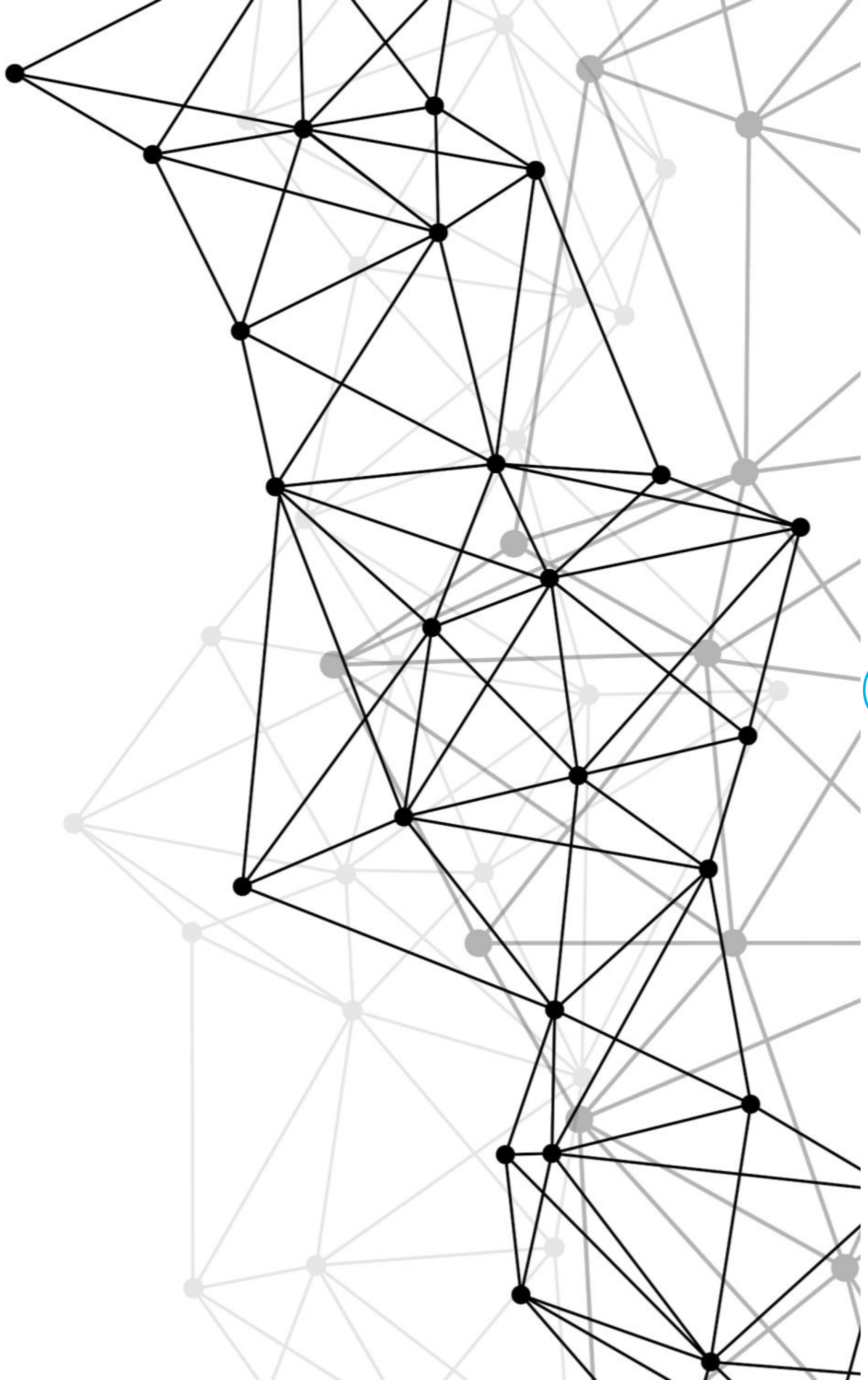
EPFL csem

Prevent overfitting

1. Get more data
2. Use a model that has the right capacity
   - Enough to fit data
   - Not too much to fit noise
   - Parameter tuning
3. Average many different models
   - Models with different forms
   - Trained on different subsets
4. Use specific regularization techniques

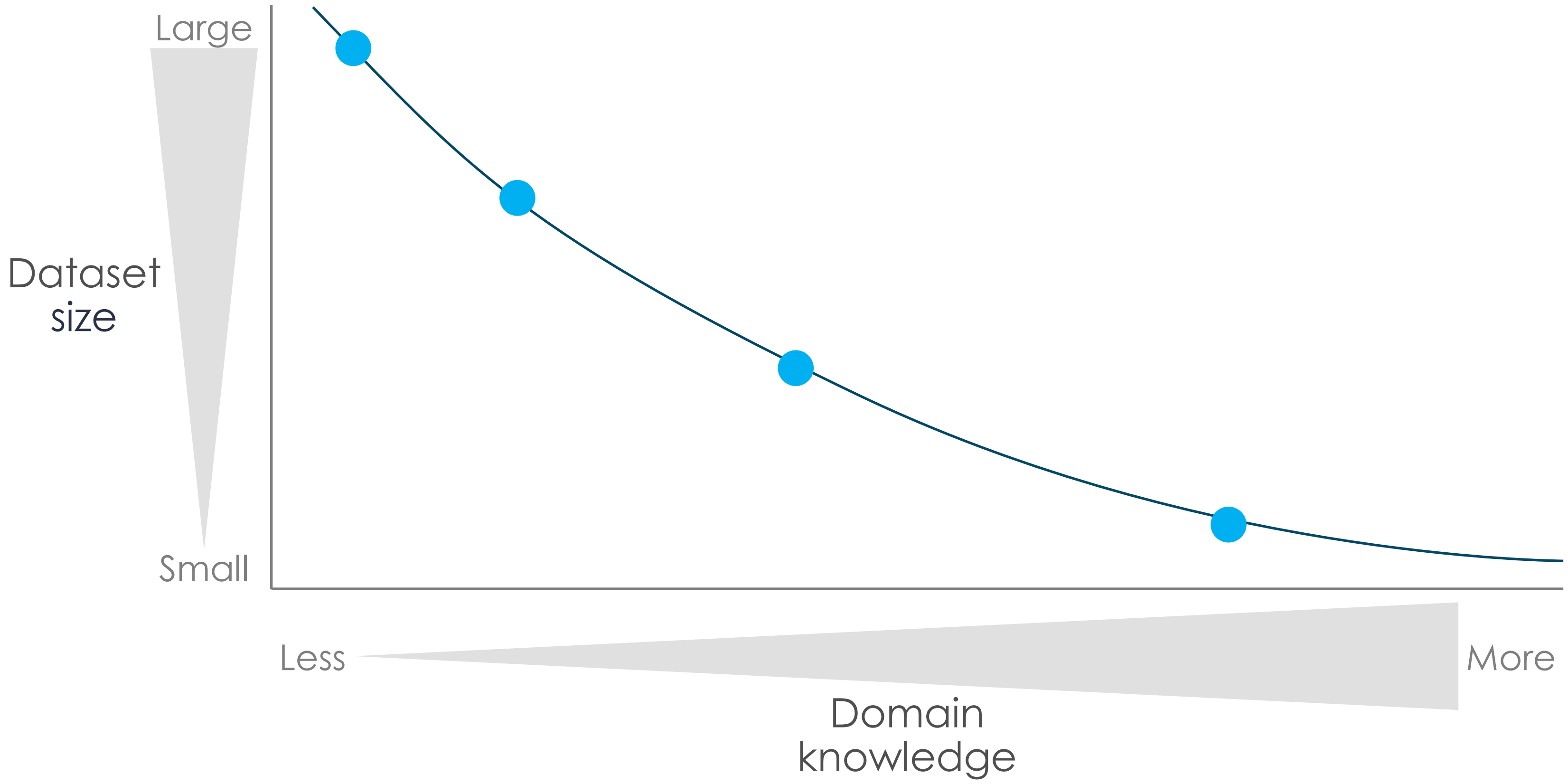
Content

- Introduction
- Perceptron
- Multilayer perceptron (MLP)
- Loss function
- Gradient descent
- Backpropagation
- Convolutional neural network
- Recurrent neural network
- Bias-variance trade-off
- Data



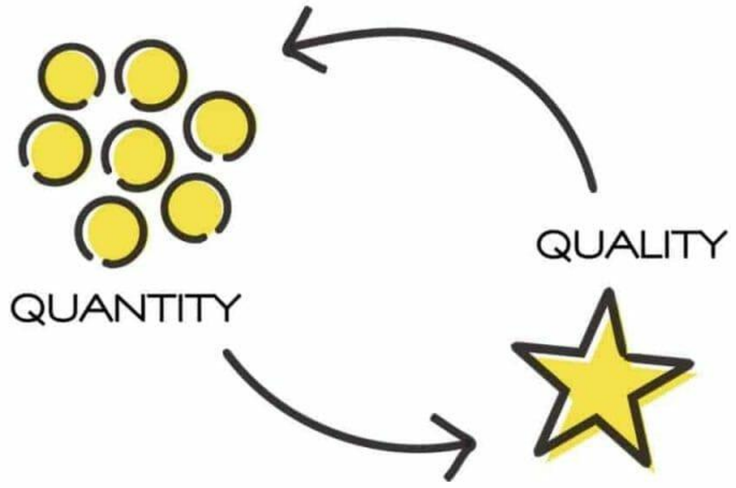
Data

- Very data hungry
- Rule-based and neural network both need data
- Why is so much training data necessary for neural network?



Data

- Quality vs quantity
- Model is as good as the data provided
- Data covering the entire solution space



- Labs

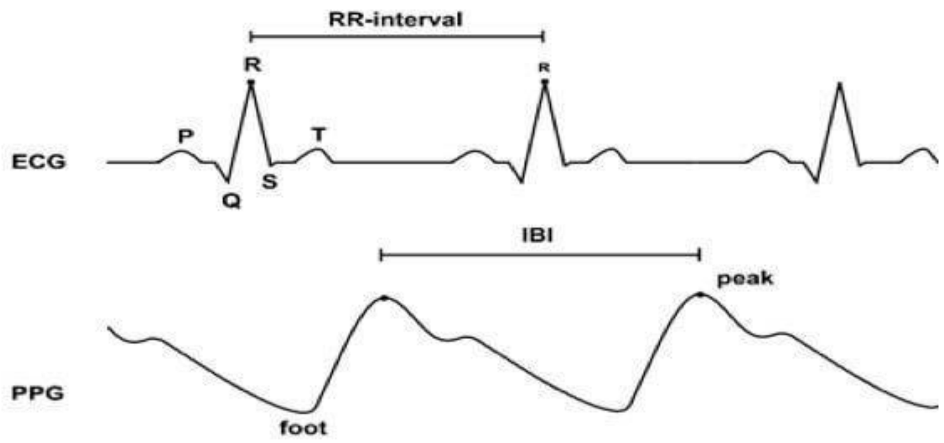
Lab – Instructions

- Submit report as **single PDF file**
- Recommended to work in groups of **3 students**
- You can prepare one single report for the group (name1_name2_name3_lab_NN.pdf)
- But every member must upload the file on Moodle

- Python code is given and provided as **Jupyter notebooks**
- This practical session is not focused on coding but on questions testing your understanding and interpretation of the results.

Labs

- **Aim**
  - Define different NN architectures (MLP, CNN, RNN)
  - Train model
  - Evaluate model
  - Interpret results
- **2 exercises** in this lab session on real-life biomedical problems
  - Atrial fibrillation classification from interbeat intervals (IBIs)
  - Gait classification from stride intervals (duration between steps when walking)



Labs

- Process locally with a virtual environment rather than using noto
    1. Uncompress the compressed file with the experiments.
    2. Open a terminal in the uncompressed directory.
    3. Create a Python virtual environment to avoid package conflicts.
       `python -m venv venv`
    - Activate it.
      Linux: `source venv/bin/activate`
    - Windows: `venv\Scripts\activate`
    4. Install the requirements with pip.
       ```
       python -m pip install --upgrade pip
       python -m pip install -r requirements.txt
       ```
    5. Start JupyterLab.
       `python -m jupyter lab`