Friedrich-Alexander-Universität
Technische Fakultät

Machine Learning
Data Analytics

Machine Learning for Time Series

Dr. Emmanuelle Salin
Machine Learning and Data Analytics (MaD) Lab
Friedrich-Alexander-Universität Erlangen-Nürnberg
05.12.2024

Topics overview

1. Time series fundamentals and definitions (Part 1)
2. Time series fundamentals and definitions (Part 2)
3. Bayesian Inference and Gaussian Processes
4. State space models (Kalman Filters)
5. State space models (Particle Filters)
6. Autoregressive models
7. Data mining on time series
8. Deep Learning (DL) for Time Series (Introduction to DL)
9. DL – Convolutional models (CNNs)
10. DL – Recurrent models (RNNs and LSTMs)
11. DL – Attention-based models (Transformers)
12. DL – From BERT to ChatGPT
13. DL – New Trends in Time Series processing
14. Time series in the real world

Topics overview

1. Time series fundamentals and definitions (Part 1)
2. Time series fundamentals and definitions (Part 2)
3. Bayesian Inference and Gaussian Processes
4. State space models (Kalman Filters)
5. State space models (Particle Filters)
6. Autoregressive models
7. Data mining on time series
8. **Deep Learning (DL) for Time Series (Introduction to DL)**
9. DL – Convolutional models (CNNs)
10. DL – Recurrent models (RNNs and LSTMs)
11. DL – Attention-based models (Transformers)
12. DL – From BERT to ChatGPT
13. DL – New Trends in Time Series processing
14. Time series in the real world

In this Lecture...

1. Introduction to Deep Learning

2. From Perceptron to Multi-Layer Perceptron

3. Training a neural network

4. Evaluating a deep learning model

Why Deep Learning?

Previous method needed **handcrafted features**:

- MFCCs (speech processing) (1)
- I-Vector (speech processing) (2)
- Sift (scene alignment, videos) (3) → Needs expert knowledge about domain

(1) Mermelstein, P. (1976). Distance measures for speech recognition, psychological and instrumental. *Pattern recognition and artificial intelligence*, *116*, 374-388.

(2) V. Gupta, P. Kenny, P. Ouellet and T. Stafylakis, "I-vector-based speaker adaptation of deep neural networks for French broadcast audio transcription," *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2014, pp. 6334-6338, doi: 10.1109/ICASSP.2014.6854823.

(3) Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, *60*(2), 91-110.

Why Deep Learning?

What if we can not define generally applicable features?

- High dimensional data
- Hard to come up with generally applicable features

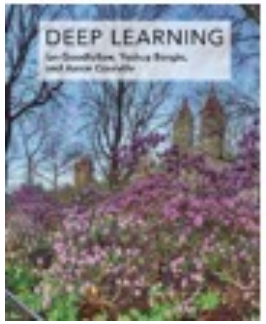
→ With deep learning, we can find features in a data driven way

→ Can help capture complex non-linear relationships

References

Deep Learning

by Ian Goodfellow, Yoshua Bengio, and Aaron Courville (2016)



Deep learning for Time Series
Introduction to Deep Learning

The Human Brain

The human brain is our reference for an intelligent agent.

- It contains different areas specialized for some tasks (e.g., the **visual cortex**)
- It consists of neurons as the fundamental unit of "**computation**"

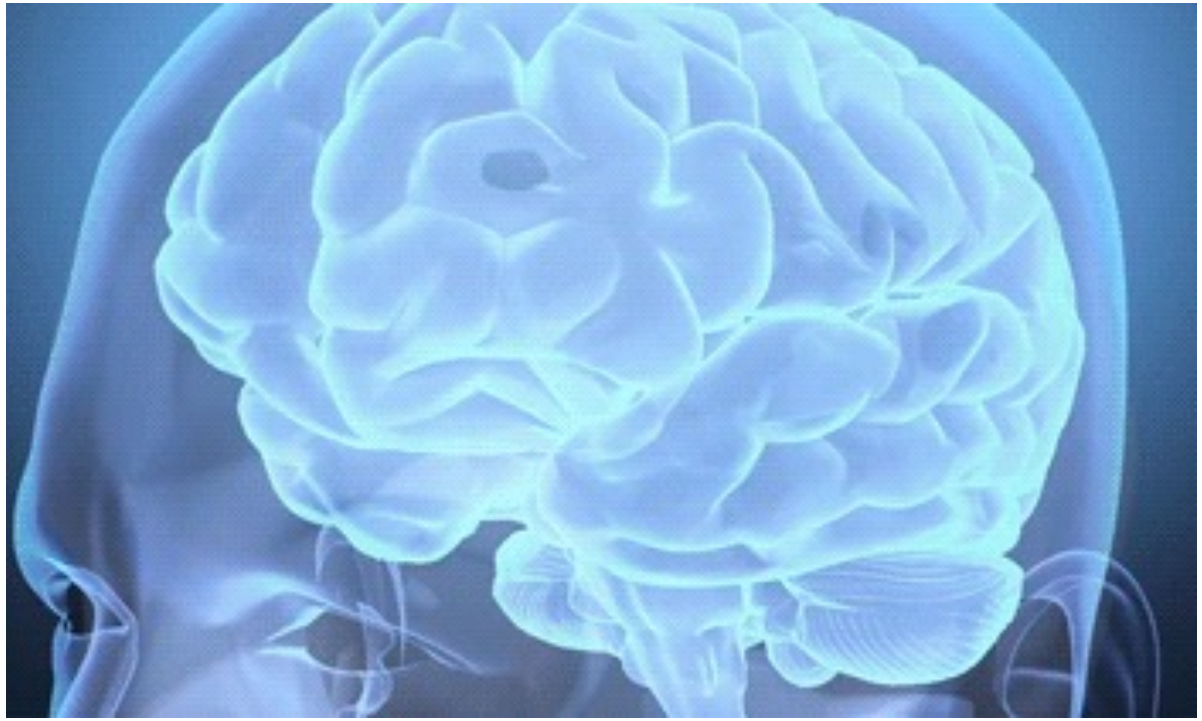
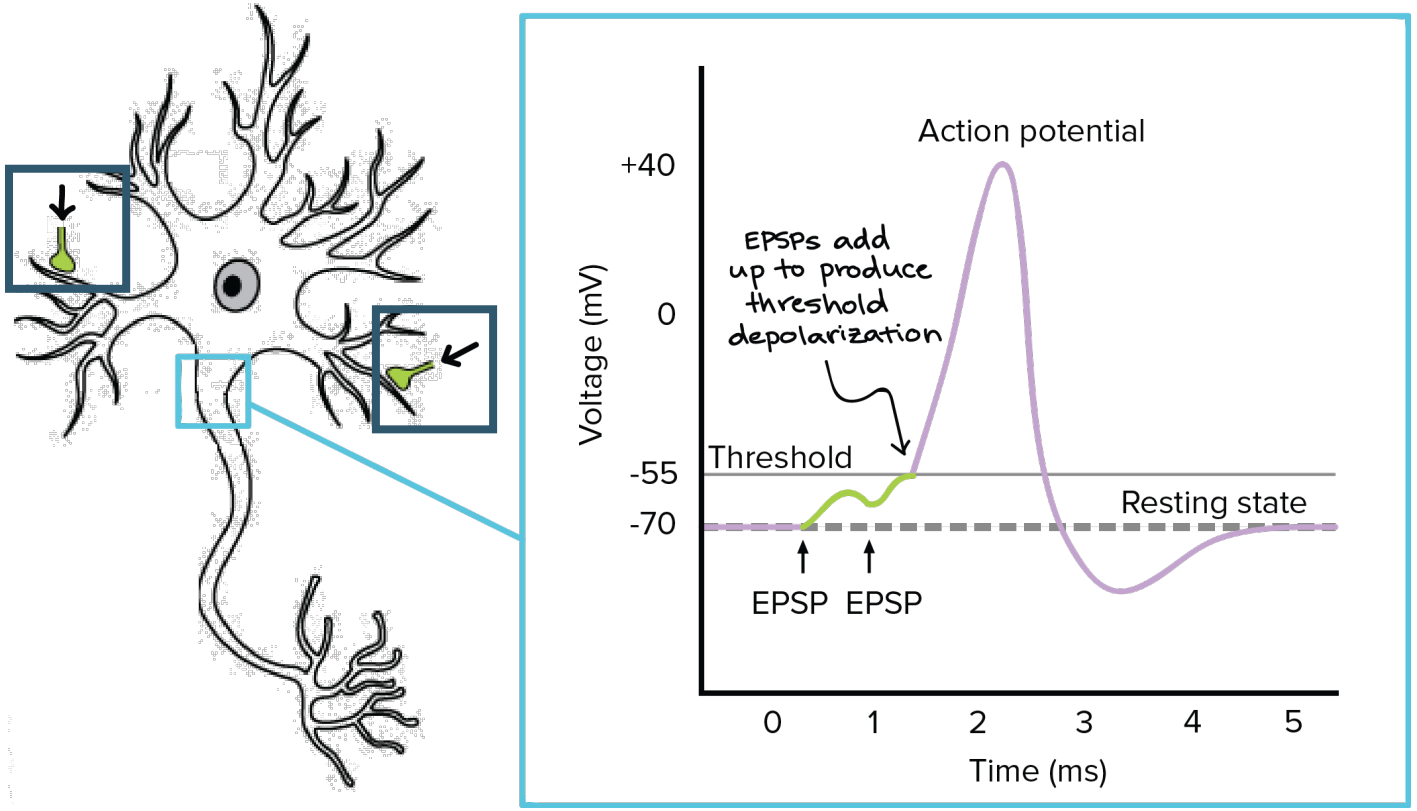


Image from https://ucresearch.tumblr.com/post/138868208574

The Brain's Neuron



EPSP = Excitatory postsynaptic potential

- Excitatory stimuli reach the neuron
- Threshold is reached
- Neuron fires and triggers action potential

Image from https://www.khanacademy.org/science/biology/human-biology/neuron-nervous-system/a/the-synapse

Deep learning for Time Series
From Perceptron to Multi-Layer Perceptron

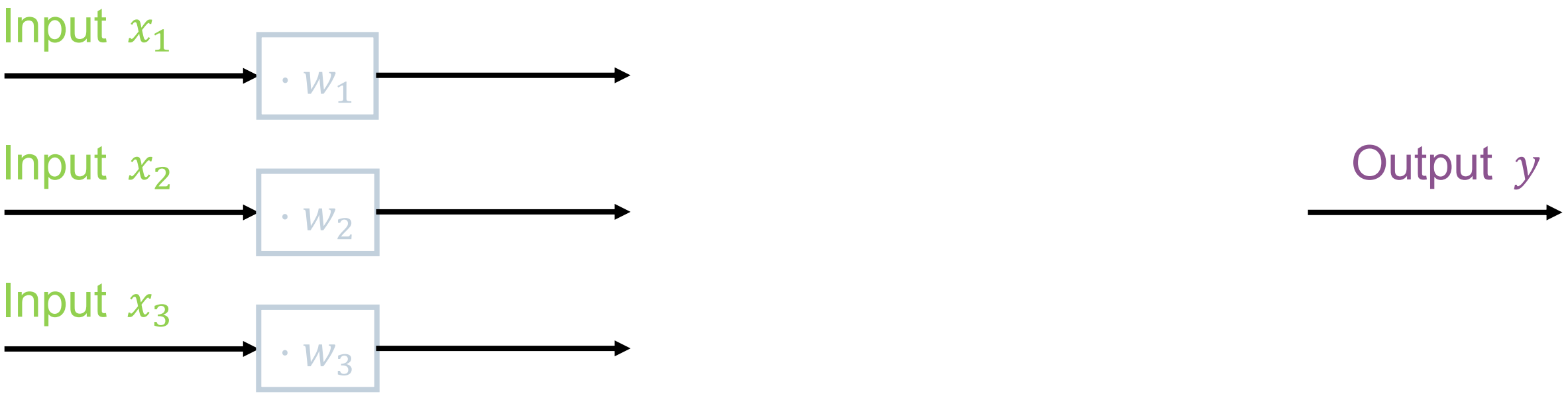
The Perceptron – Computational Model of a Neuron

Let's build the computational model step by step:

1. Show the input and output of our neuron (which depends on the data and task)

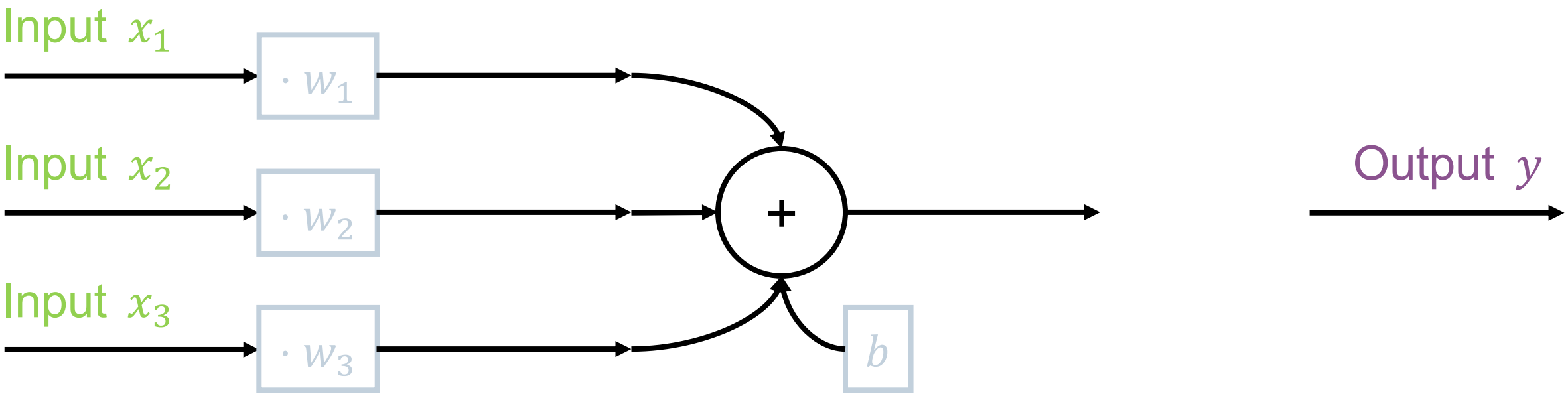
The Perceptron – Computational Model of a Neuron

2. Weights can “select” or “deselect” input channels (not all are relevant for subsequent computations)



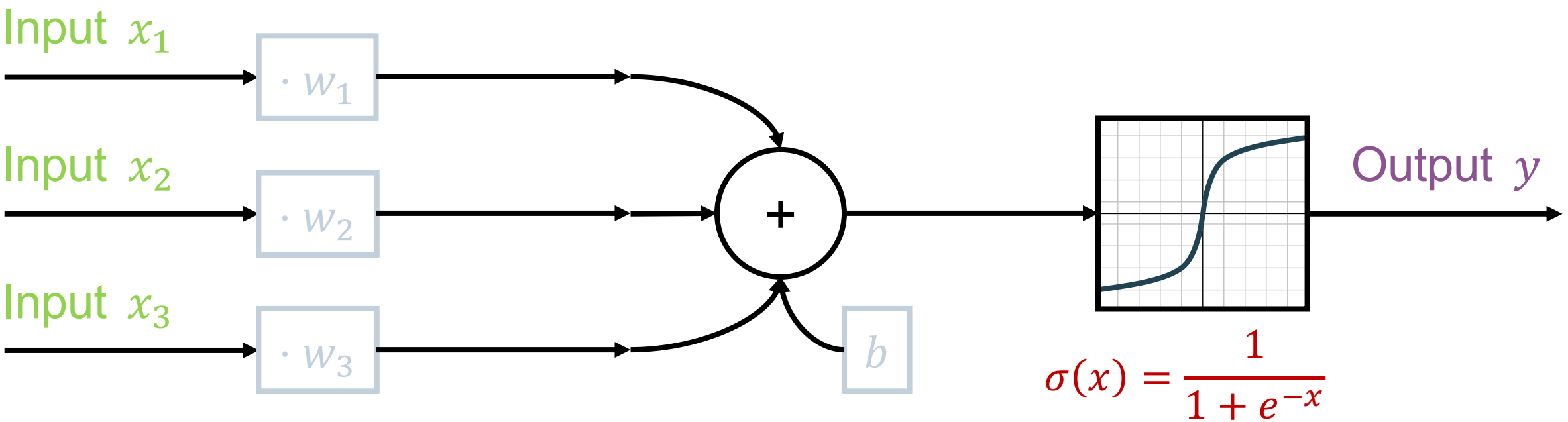
The Perceptron – Computational Model of a Neuron

3. We add up all the excitatory signals and the **resting potential (or bias)** to determine the current potential.



The Perceptron – Computational Model of a Neuron

4. A **threshold function (or activation function)** σ is applied to determine whether an action potential has to be sent in the output



Activation Functions

The goal of an **activation function** is to activate (or not) neuron.

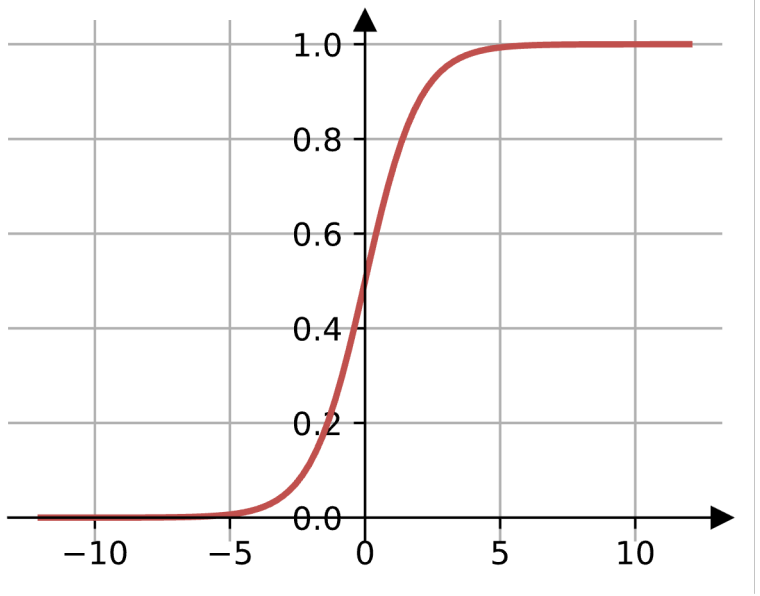
It brings **non-linearity** to a neural network:

→ The output is **not** a linear function of the input

Different activation functions can be considered, taking into account their:

- Output range
- Mean
- Gradient

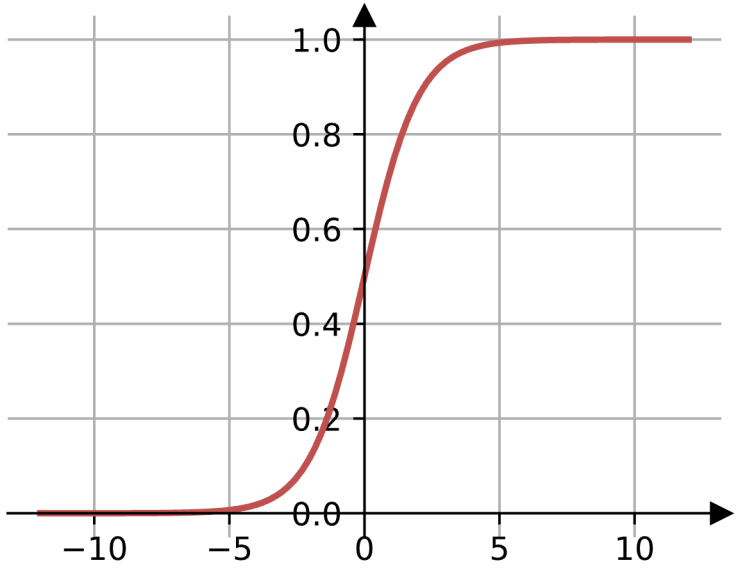
Sigmoid



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

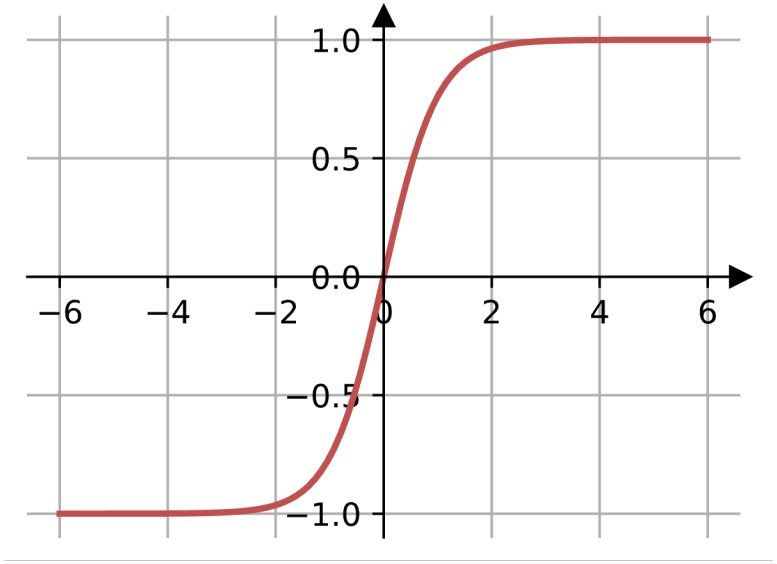
Activation Functions

Sigmoid



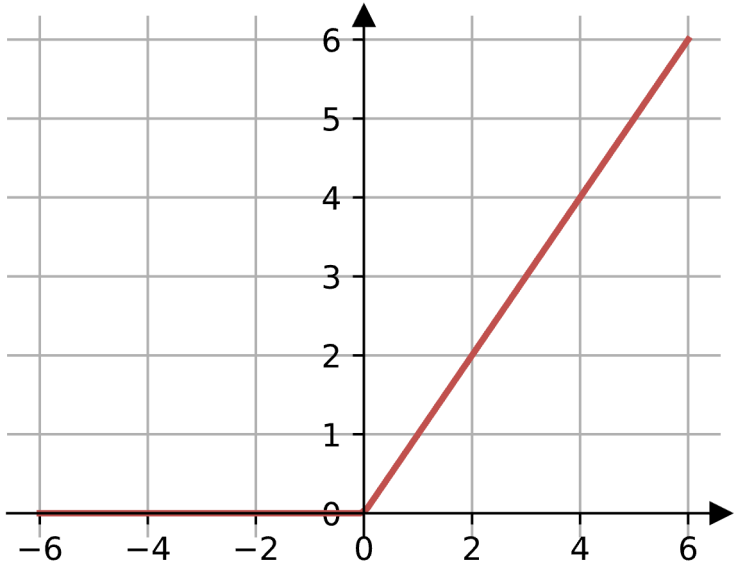
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Hyperbolic Tangent



$$\sigma(x) = \tanh(x)$$

Rectified Linear Unit

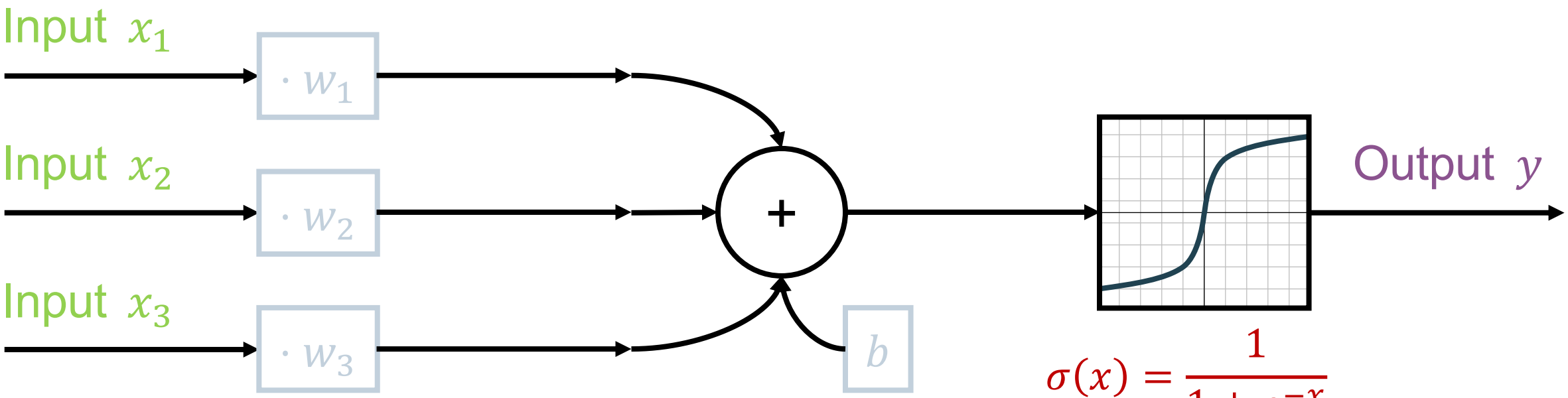


$$\sigma(x) = \max(x, 0)$$

The Perceptron – Computational Model of a Neuron

5. We can write the perceptron mathematical model to map inputs x_1, x_2, x_3 to the output y_1 using channel weights w_1, w_2, w_3, b:

$$y = \sigma(w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3 + b) = \sigma(\sum_i w_i \cdot x_i + b)$$

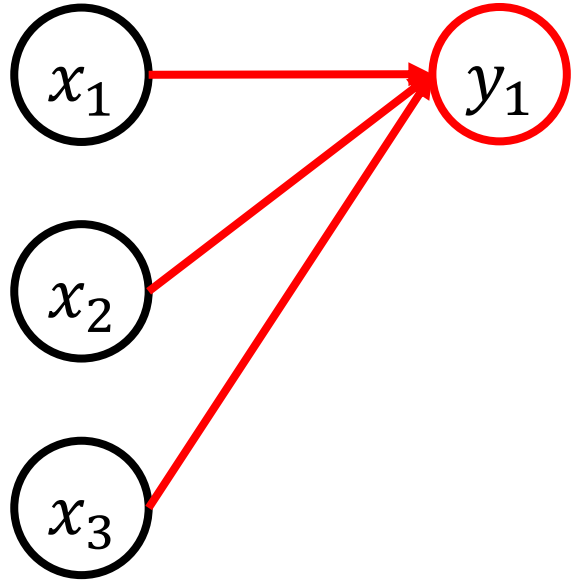


Single Layer Perceptron

We can combine multiple perceptrons to create a **layer.**

One perceptron has the following equation:

$$y_1 = \sigma(w_{11} \cdot x_1 + w_{12} \cdot x_2 + w_{13} \cdot x_3 + b)$$

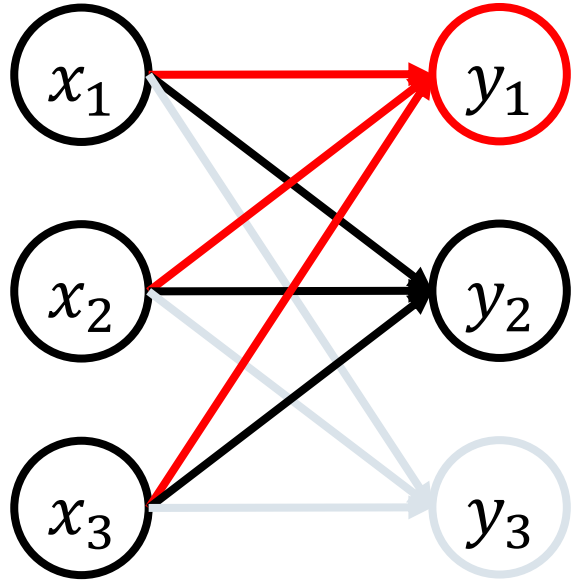


Single Layer Perceptron

We can combine multiple perceptrons to create a **layer.**

We can combine **three perceptrons** into one layer:

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \underbrace{\begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{pmatrix}}_{W} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}}_{b}$$



$\boldsymbol{W}$, $\boldsymbol{b}$

Or in a more simplified form: $y = \sigma(W \cdot x + b)$

Multi Layer Perceptron (MLP)

We can chain **multiple layers**

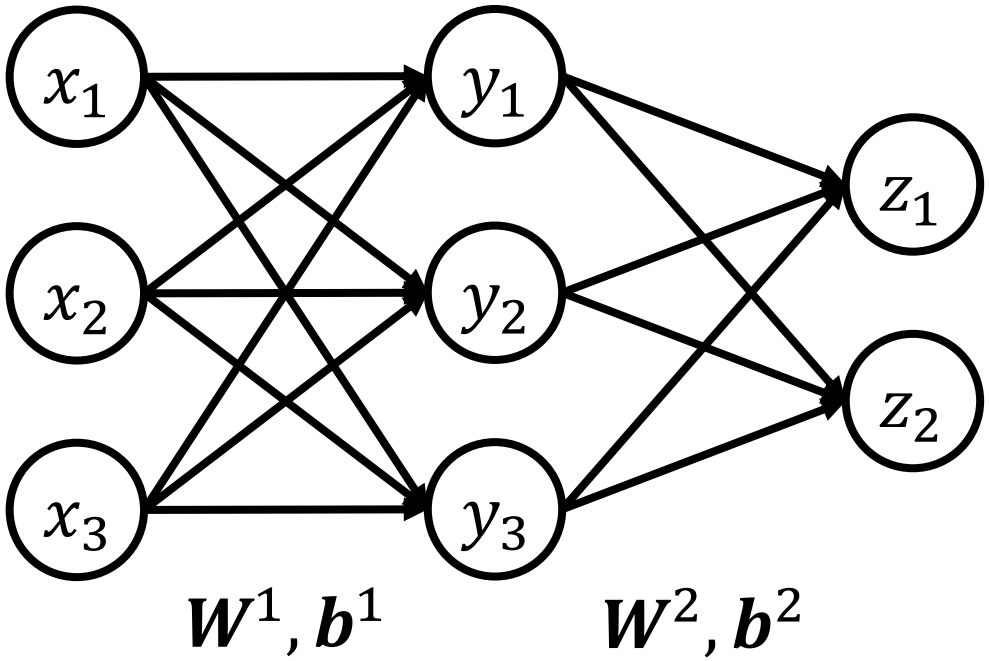
$$y = \sigma(W^1 \cdot x + b^1)$$

$$z = \sigma(W^2 \cdot y + b^2)$$

So: $z = \sigma(W^2 \cdot \sigma(W^1 \cdot x + b^1) + b^2)$

→ Each layer has its own set of parameters (weights $\boldsymbol{W}^i$ and bias $\boldsymbol{b}^i$)

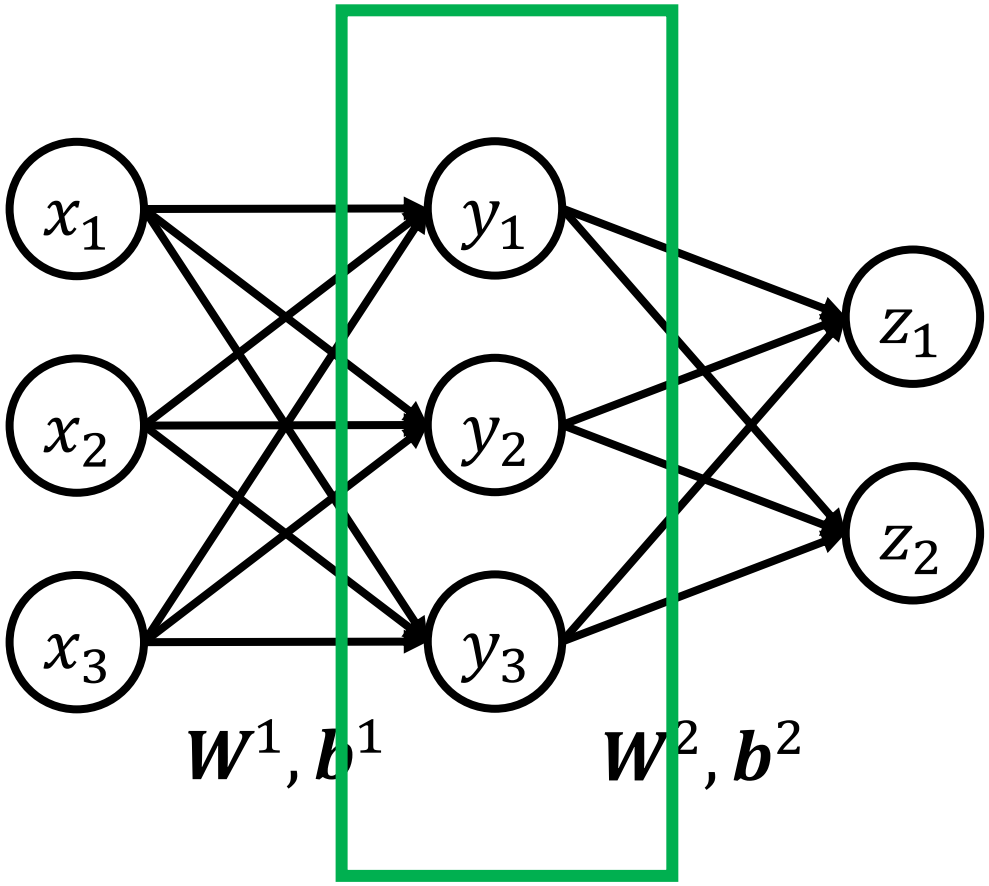
→ The activation function ensures that all layers do not collapse into one: it introduces **non-linearity**



Multilayer perceptron (MLP)

We call "**hidden layer**" any layer in between the input and the output layers.

For example, this neural network (image on the right) is a Multi-Layer-Perceptron (MLP) with a **single hidden layer** (highlighted by the green box).



→ The underlying computation is described by:

$$\boldsymbol{y}^{i+1} = \sigma(\boldsymbol{W}^i \cdot \boldsymbol{y}^i + \boldsymbol{b}^i)$$

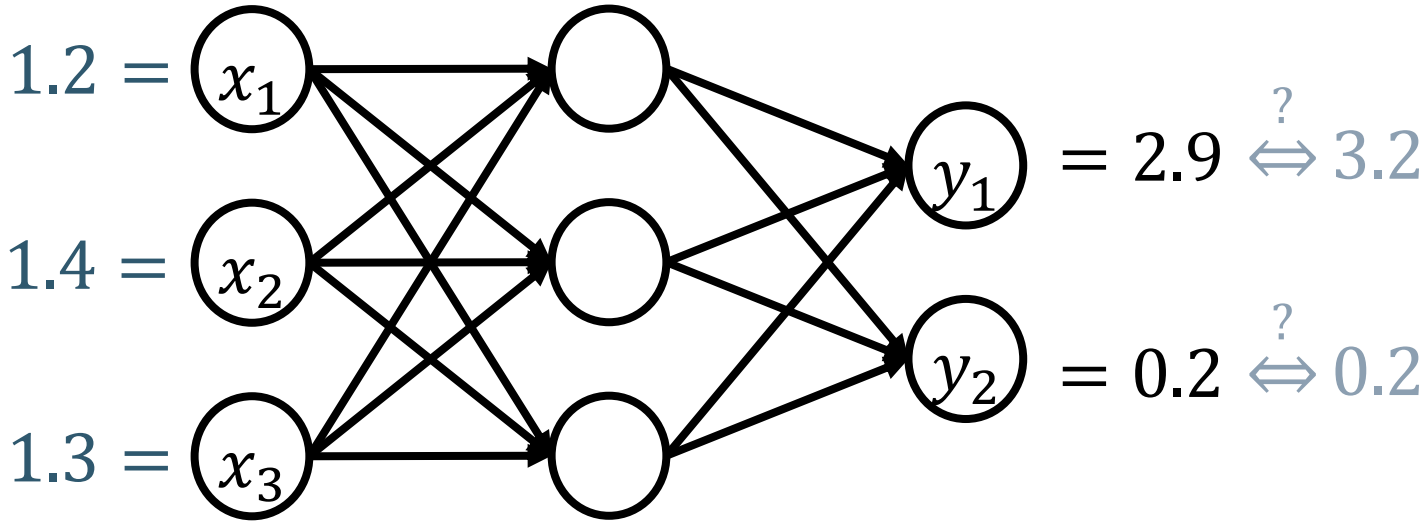
Also referred to feedforward networks (no feedback connection)

Deep learning for Time Series
Training a neural network

How are Model Parameters Learned?

1) **Forward propagation:** This phase refers to the computation of the output using input and parameters.

2) **Loss calculation:** The output and expected output are then compare using a loss function.



Loss Functions

The loss function is a comparison metric between the predicted outputs $\hat{y}_i$ and the expected outputs y_i. Its choice is important, as the network will aim to **minimize** it.

The choice of the loss function usually depends on the type of problem:

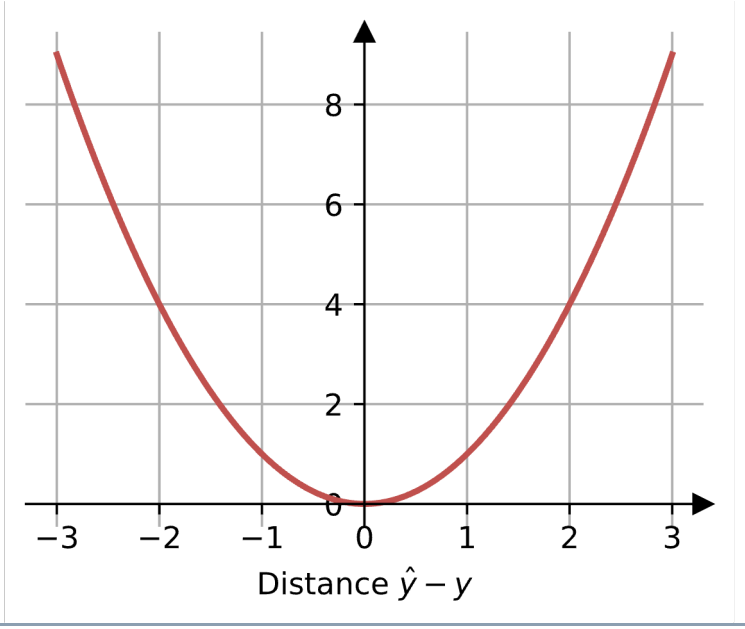
- **Regression:** The predicted output and expected output are numerical values. The goal of the task is to minimize the difference between those two values.
  Example: Predicting temperature values
- **Classification**: The expected output is a class label, and the predicted output is the probability of each class being positive. The goal is to minimize differences between predicted class probability and true labels.
  Example: Activity recognition using sensors (classes: walking, running…)

Loss Functions

The loss function is a comparison metric between the predicted outputs $\hat{y}_i$ and the expected outputs y_i. Its choice is important, as the network will aim to **minimize** it.

- For **regression**, a common metric is the **mean squared error** loss.

$$MSE(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

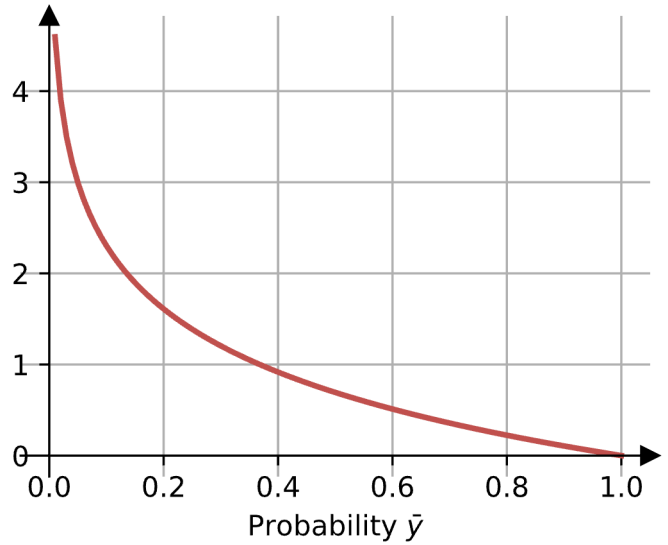


Loss Functions

The loss function is a comparison metric between the predicted outputs $\hat{y}_i$ and the expected outputs y_i. Its choice is important, as the network will aim to **minimize** it.

- For **classification**, a common metric is the **cross entropy** loss:

$$CE(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_i y_i \log(\hat{y}_j)$$



The raw outputs should be converted in probabilities that sum to 1 using an activation function (e.g. **softmax**).

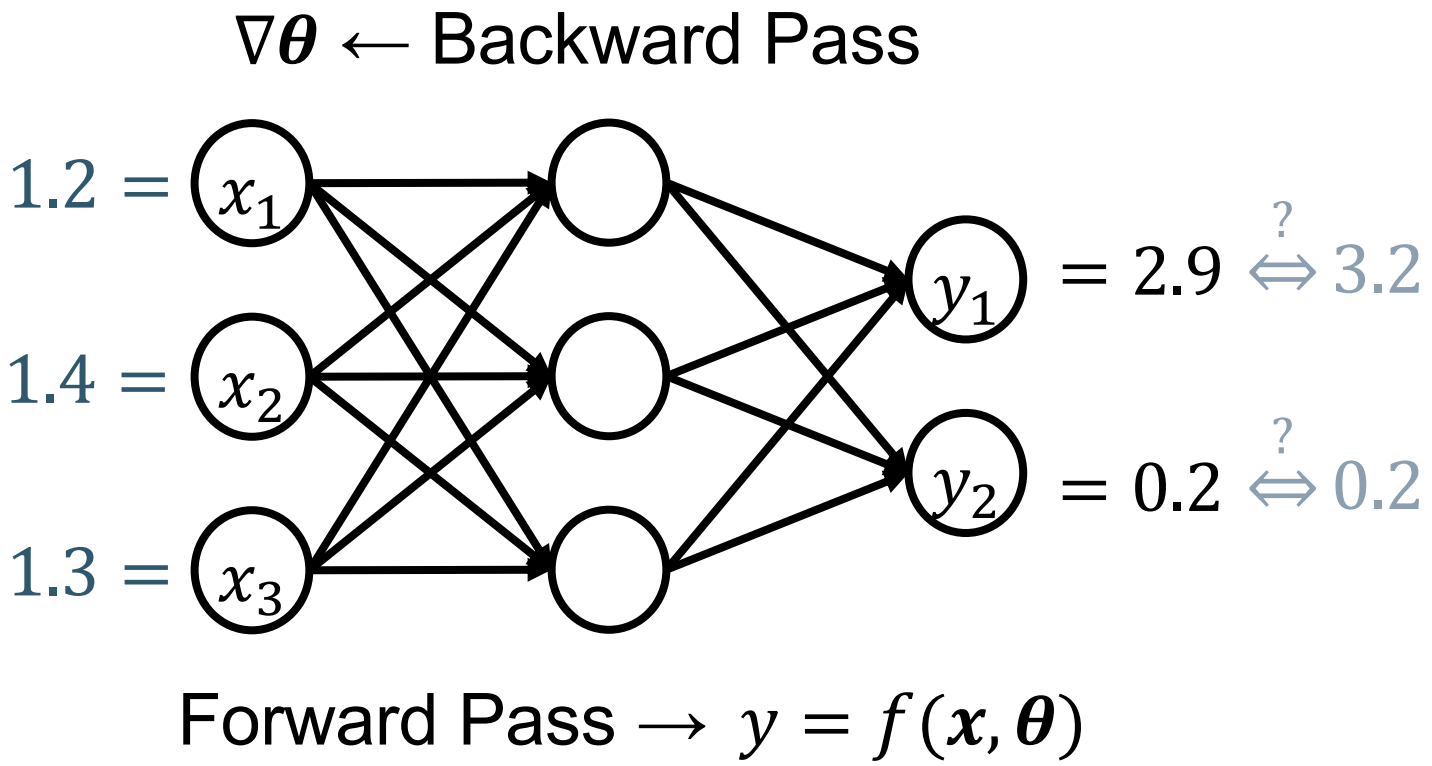
$$s\left(x_i\right) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

How are Model Parameters Learned?

1) **Forward propagation:** This phase refers to the computation of the output using input and parameters.

2) **Loss calculation:** The output and expected output are then compare using a loss function.

3) **Backward propagation**: During this phase, the model computes the gradients of the loss with respect to each parameter ($\boldsymbol{\theta}$).



Backward propagation: Gradient Descent

Gradient descent is used to incrementally adjust the parameters θ based on their gradient $\nabla\theta$

For each iteration:

- Compute the **gradient** $\nabla\theta^i$ of the loss $\mathcal{L}$ with respect to the parameter θ^i: $\nabla\theta^i = \partial\mathcal{L}/\partial\theta^i$
- **Update** parameter using $\theta^{i+1} = \theta^i - \lambda \cdot \nabla\theta^i$

- The goal of gradient descent is to update model parameters move towards a **minimum** of the loss
- The **learning rate** λ is used to control the size of each **step**

Backpropagation Algorithm

Numerically evaluating the gradient can be computationally **expensive**.

The **backpropagation algorithm** is widely used to train artificial neural networks.

- Backpropagation computes **efficiently** the gradients of the loss function with respect to all weights in the network.
- This efficiency makes it feasible to use **gradient methods** for training multilayer networks and updating weights to minimize loss.
- The algorithm makes use of the **chain rule**, computing the gradient one layer at a time, to avoid redundant calculations.

Backpropagation Algorithm

The loss function can be expressed as a **sum** of terms, one for each data point in the training set of size N:

$$L_N = \sum_{n=1}^{N} L_n$$

Using the sum of squared errors as an example, we can compute the following loss term:

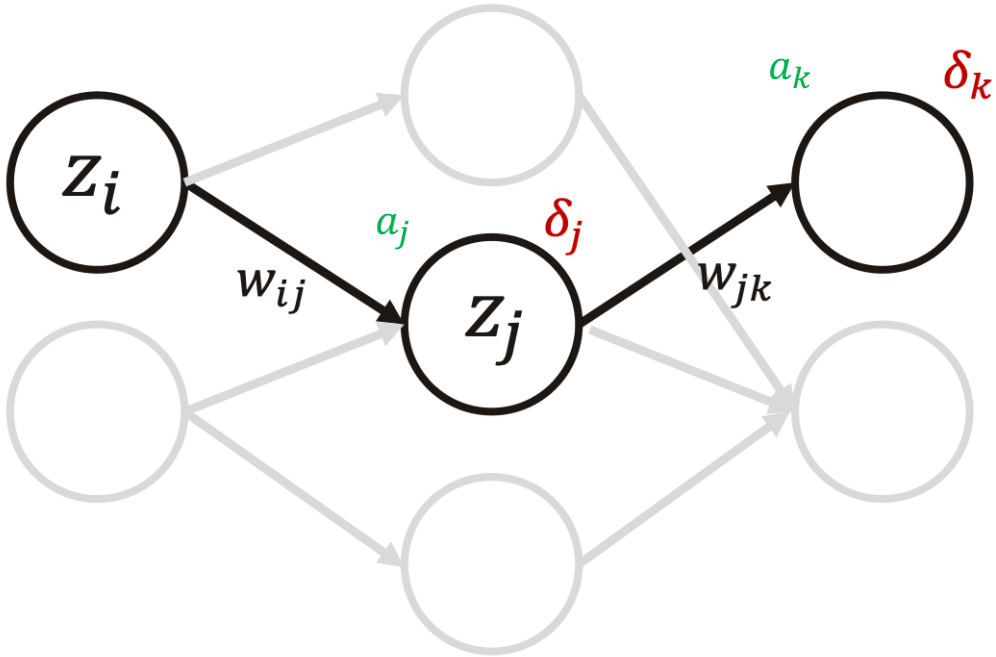
$$L_n = \frac{1}{2} \sum_{i} (\hat{y}_{in} - y_{in})^2$$

where y_i is the desired output and $\hat{y}_i$ is the predicted output from the neural network.

Backpropagation Algorithm

The goal is to compute the **gradient** of the loss term $\boldsymbol{L_n}$ with respect to the parameter $\boldsymbol{w_{ij}}$.

$$\frac{\partial L_n}{\partial w_{ij}}$$



We first denote $\boldsymbol{a_j} = \sum_i w_{ij} z_i$.

We have $z_j = \sigma(a_j)$

Backpropagation Algorithm

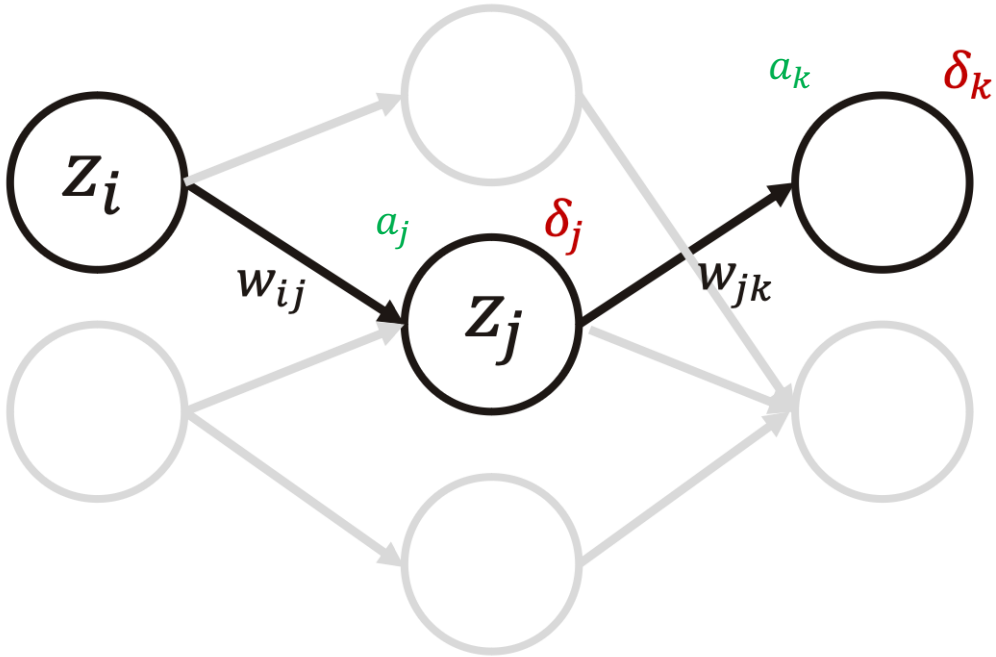
The derivation of the backpropagation algorithm begins by applying the **chain** rule to the error function partial derivative.

$$\frac{\partial L_n}{\partial w_{ij}} = \frac{\partial L_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ij}}$$

We denote $\delta_j \equiv \frac{\partial L_n}{\partial a_j}$

As $a_j = \sum_i w_{ij} z_i$, we get $\frac{\partial a_j}{\partial w_{ij}} = z_i$

Thus, we have: $\boldsymbol{\frac{\partial L_n}{\partial w_{ij}} = \delta_j \, z_i}$



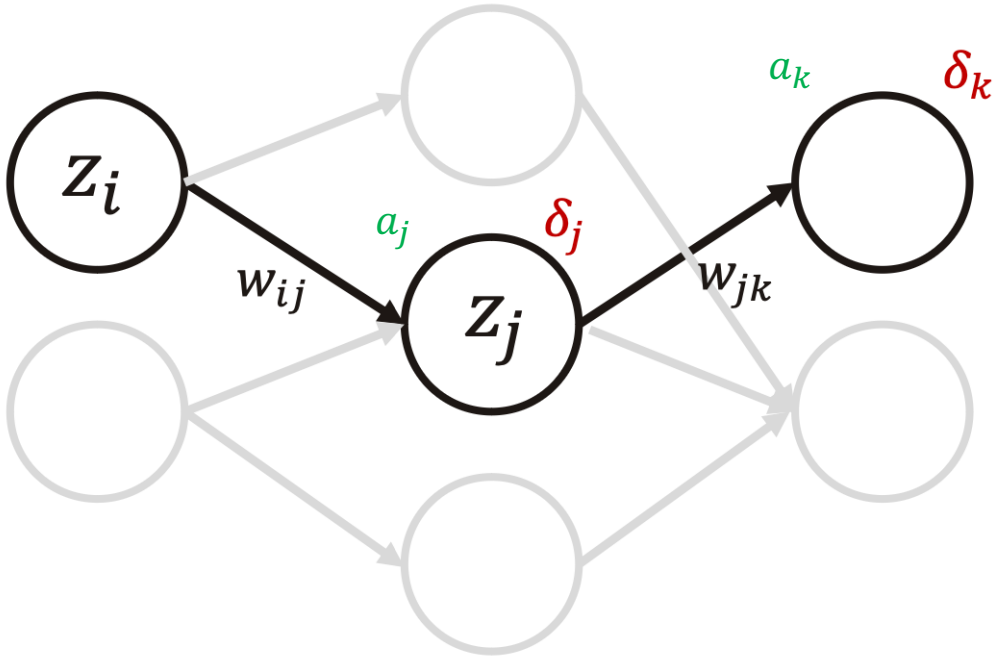
Backpropagation Algorithm: Sum of Squared Errors

Using the **chain** rule, we have:

$$\delta_j = \frac{\partial L_n}{\partial a_j} = \sum_k \frac{\partial L_n}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$

We notice that: $a_k = \sum_j w_{jk} \sigma(a_j)$

So: $\frac{\partial a_k}{\partial a_j} = w_{jk} \sigma'(a_j)$



We get the **backpropagation formula**:

$$\delta_j = \sigma'(a_j) \sum_k w_{jk} \delta_k$$

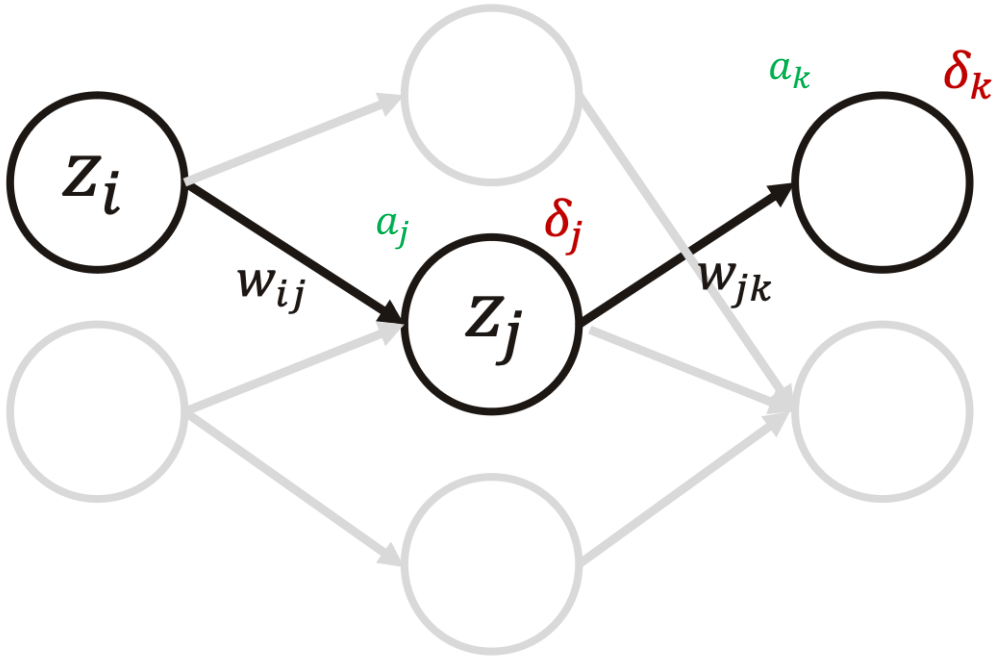
Backpropagation Algorithm: Sum of Squared Errors

Our considered loss is the sum of squared errors:

$$L_n = \frac{1}{2}\sum_i (\hat{y}_{in} - y_{in})^2$$

Thus, for the final layer, we get:

$$\delta_k = \frac{\partial L_n}{\partial a_k} = (\hat{y}_{nk} - y_{nk})$$



Backpropagation Algorithm: Overview

1. Calculate the forward pass and **store** results for $\hat{y}$, a_j^k, and z_j^k.
2. Calculate the backward pass and **store** results for $\frac{\partial L}{\partial w_{ij}}$, proceeding from the last layer:
    a) **Evaluate** the error terms for the last layer δ_k
    b) **Backpropagate** the error term for the computation of δ_j
    c) **Iterate** to all previous layers
3. Combine the individual gradients (average)
4. Update the weights according to the **learning rate** λ

How are Model Parameters Learned?

1) **Forward propagation:** This phase refers to the computation of the output using input and parameters.

2) **Loss calculation:** The output and expected output are then compare using a loss function.

3) **Backward propagation**: During this phase, the model computes the gradients of the loss with respect to each parameter ($\boldsymbol{\theta}$).

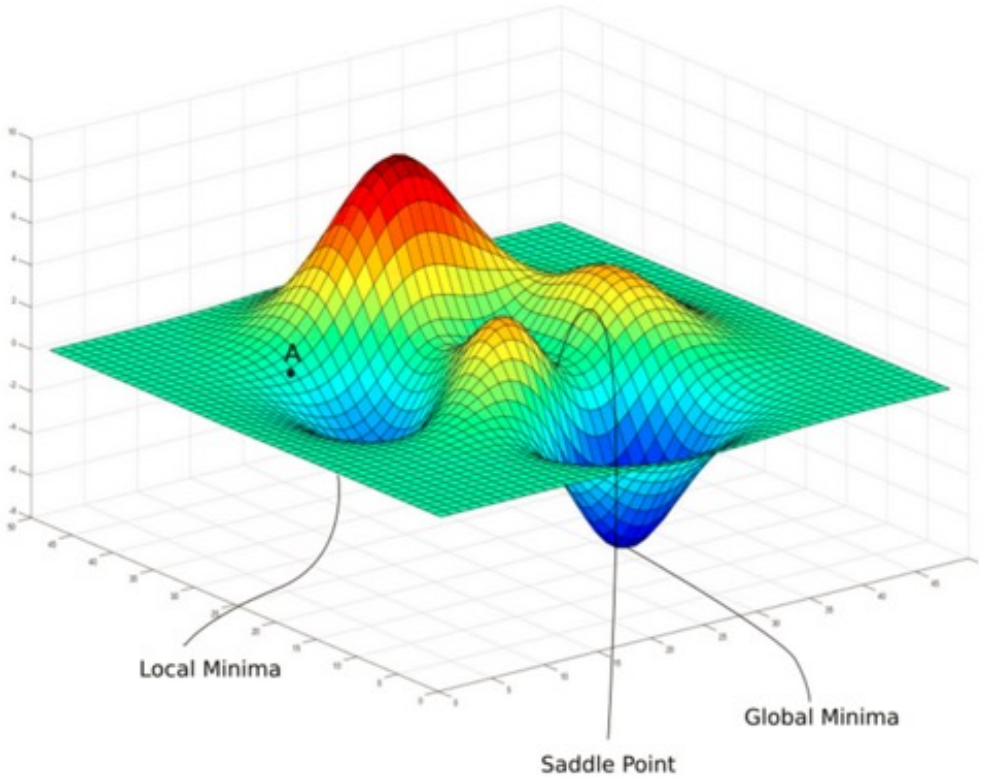
- The goal of backward propagation is to minimize the loss
- Convergence of the parameters towards the loss minimum can be hard to obtain
- It depends on the learning rate that can be difficult to optimize (slow convergence vs missing the global minimum)

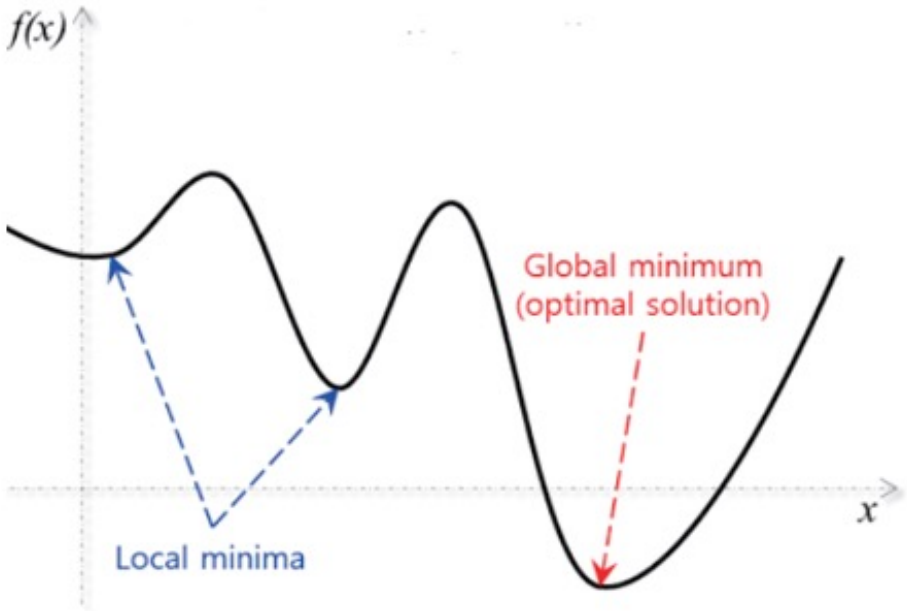
How are Model Parameters Learned?

1) **Fo**
phase
of the
paran

2) **Lo**
and e
comp

3) **Ba**
Durin
comp
loss w
paran





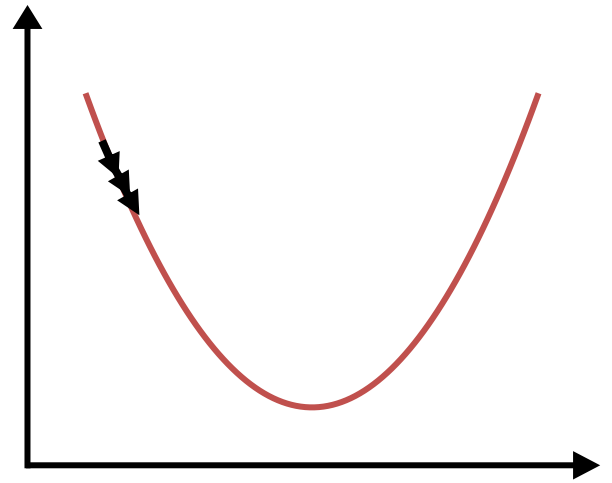
https://medium.com/analytics-vidhya/neural-networks-part-3-understanding-back-propagation-learning-rates-3482a981a2f0

Deep learning for Time Series
Optimizing Training

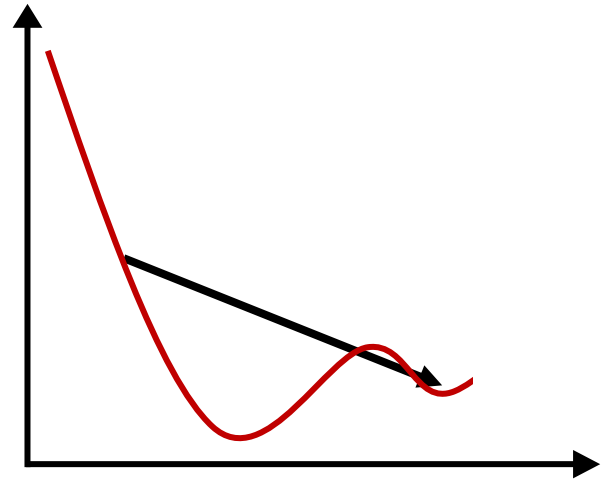
Effects of the Learning Rate

If not properly configured, the **learning rate** λ can lead to :

- **slow** convergence
- **missing** the minimum of the loss function



→ There is no guarantee that the gradient descent algorithm converges to the **global** optimum.

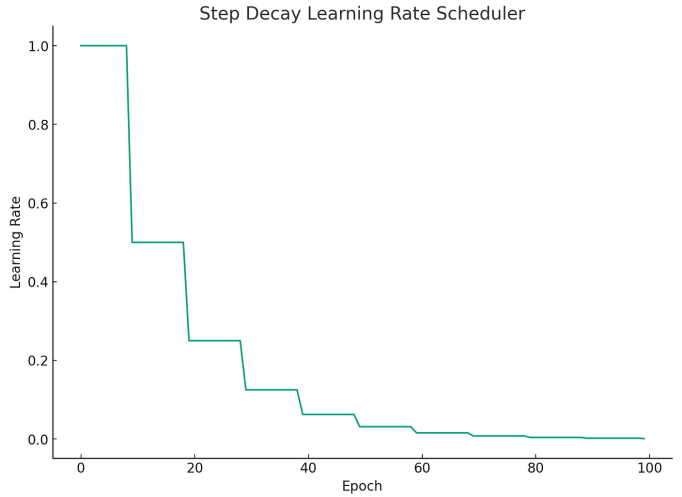


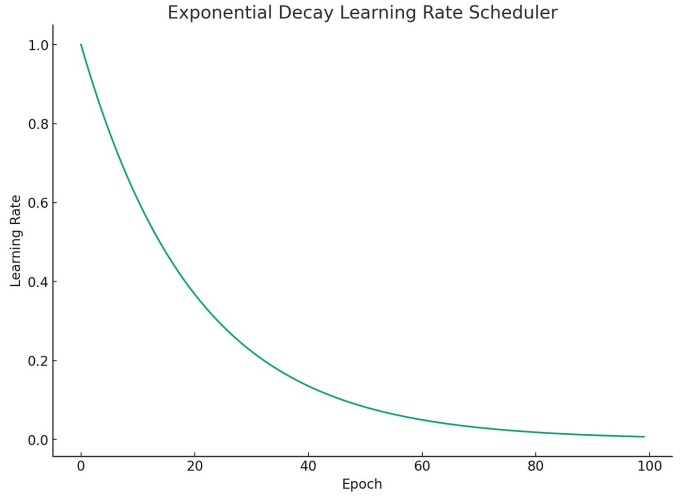
Schedulers

Learning rate should also be **adapted** throughout the training. The closer a network is to convergence, the smaller the parameters updates should be.

Schedulers can guide learning rate updates.

- **Step decay:** Learning rate is decreased by a fixed factor after a set number of training steps.
- **Exponential decay:** Learning rate decreases exponentially over time at each time step





Images from https://medium.com/@theom/a-very-short-visual-introduction-to-learning-rate-schedulers-with-code-189eddffdb00

Optimizers

The standard gradient descent method updates the model parameters based on all instances of the dataset.

→ This is time consuming, especially for larger datasets

Optimizers can be used to update model parameters in different (and more efficient) ways by adapting the learning rate.

- **Stochastic Gradient Descent:** Parameters are updated through more frequent updates (single dataset instance, or mini-batches). It can lead to noisy gradients.
- **Adam (Adaptive Moment Estimation):** Adam adjusts the learning rate for each parameter based on the decaying average of its gradients and squared gradients. The convergence is faster and more stable.

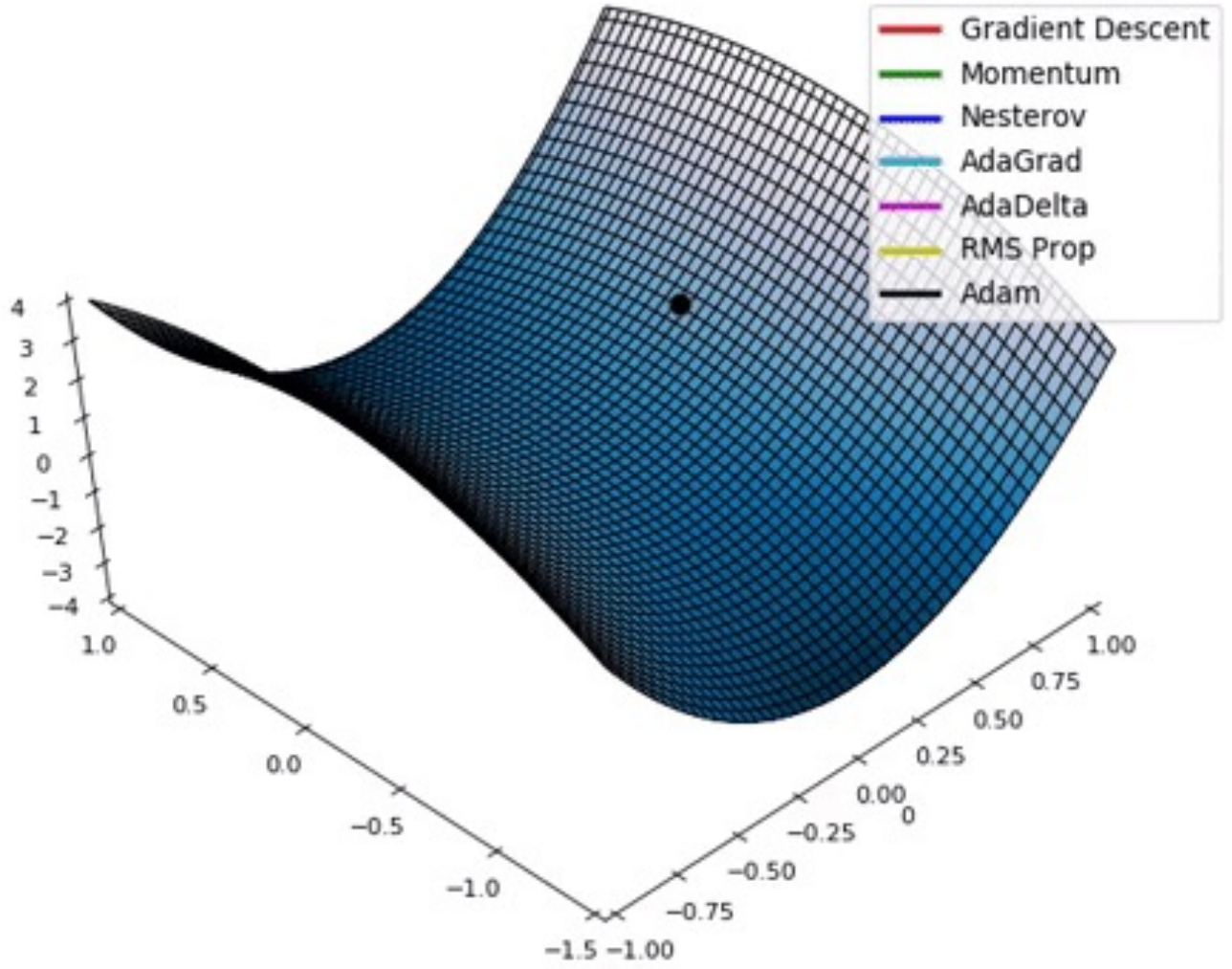
Optim

The s … l
instan

→ Th

Optim
efficie

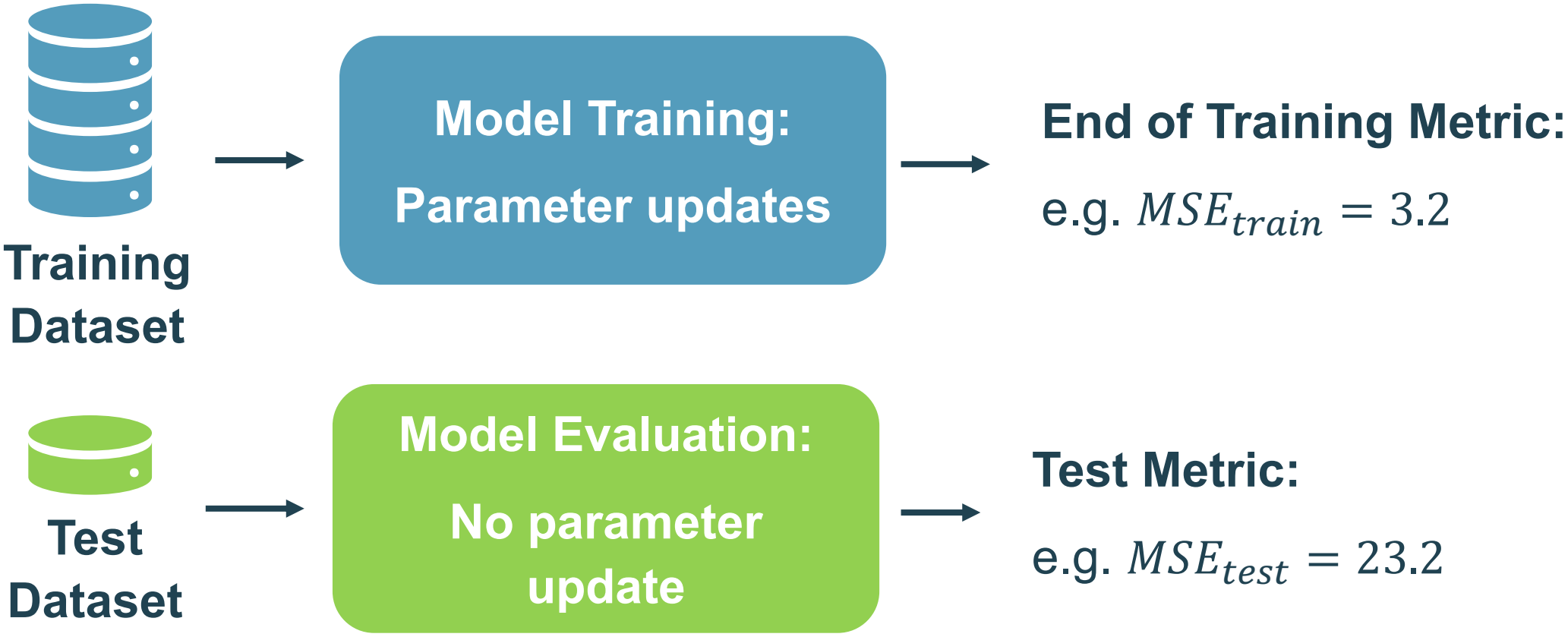
- **Sto**
  up … .
- **Ad**
  par … s.
  The



https://community.deeplearning.ai/t/difference-between-rmsprop-and-adam/310187/2

Standard Training and Evaluation Process

The dataset is split into a **training (or train) set** and a **test set**. The model is fitted on the training dataset, and evaluated on the unseen test set.



Hyperparameter Tuning

Model performance on the test set can be improved by tuning **hyperparameters**, which are the parameters fixed at start of the training (e.g. learning rate, optimizer, scheduler, number of layers).

The evaluation process is adapted:

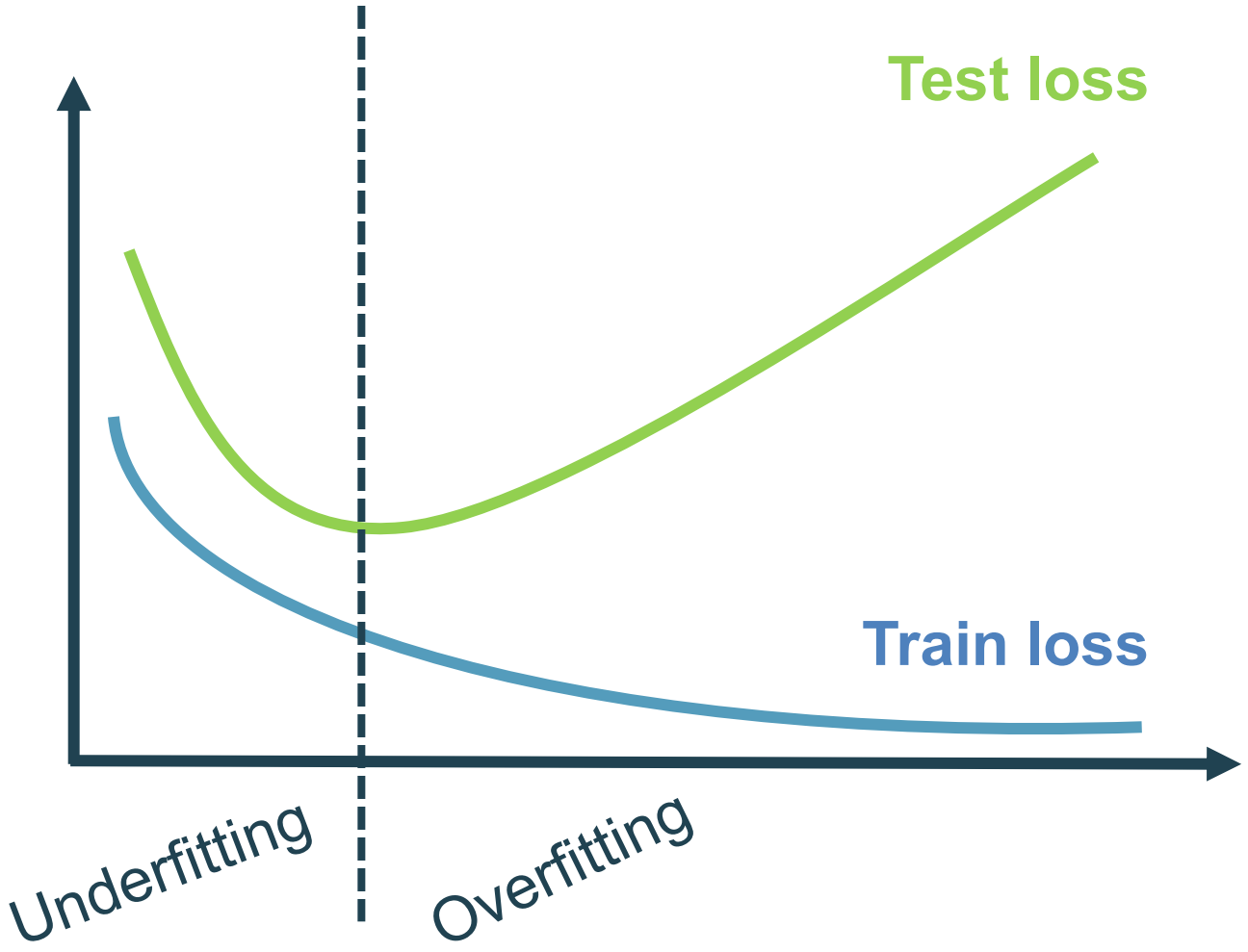
- 1) Set hyperparameters
- 2) Train model on **train** set
- 3) Evaluate on **val** set
- 4) Repeat 1 to 3
- 5) Evaluate final model on **test** set

In case of small test sets: cross validation

Underfitting and Overfitting

In deep learning, the two pitfalls of model training are **underfitting** and **overfitting**.

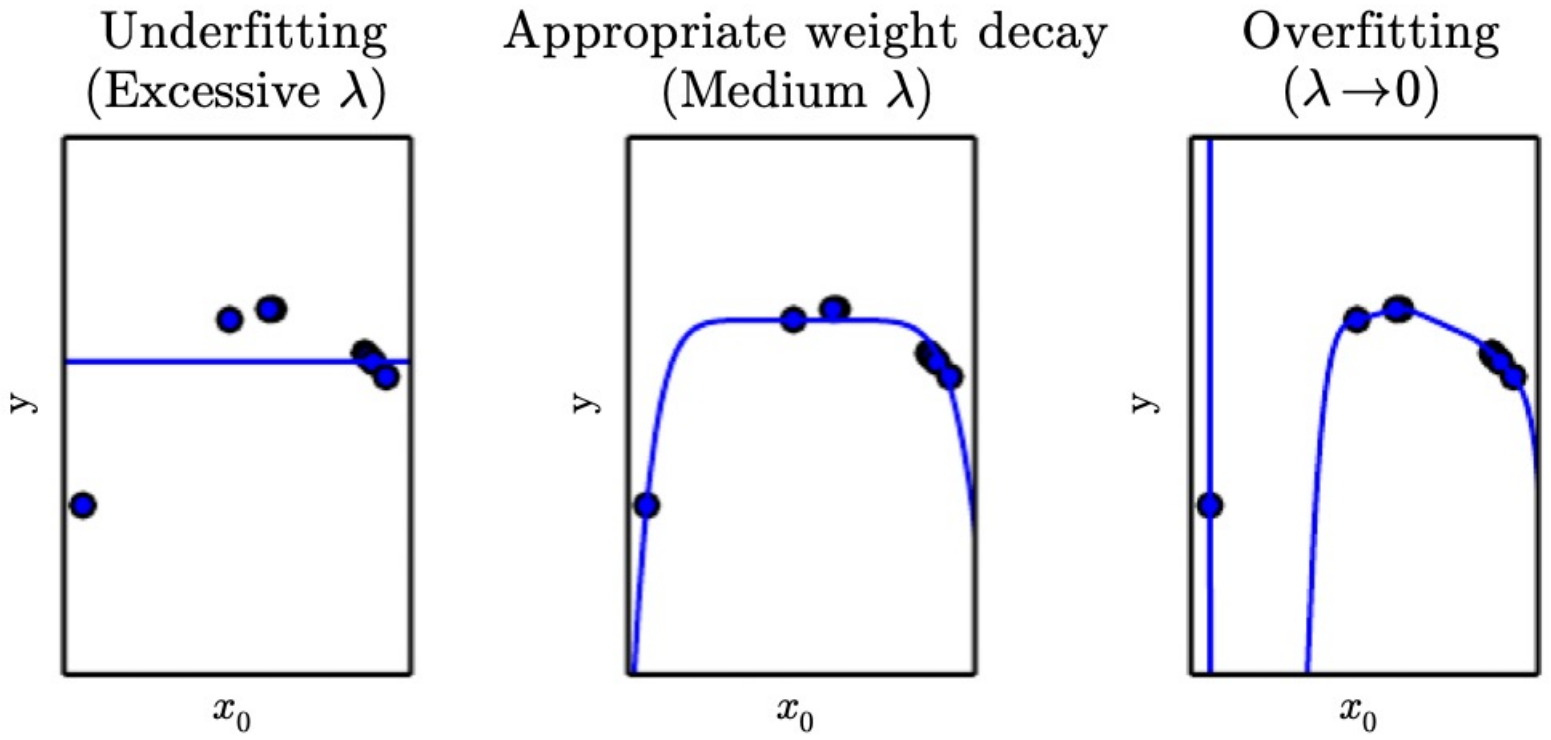
- **Underfitting:** The model has not finished capturing the underlying patterns in the data.
- **Overfitting:** The model has started capturing noise related to the training dataset that will not generalize.



Avoiding Overfitting: Regularisation

Weight decay or **L2 regularization** can be used to limit overfitting by penalizing large weights in a neural network, which limits overparameterization.

- **Weight decay:** Factor in the optimizer to reduce the weights at each step



Avoiding Overfitting: Regularisation

Weight decay or **L2 regularization** can be used to limit overfitting by penalizing large weights in a neural network, which limits overparameterization.

- **Weight decay:** Factor in the optimizer to reduce the weights at each step
- **L1 or L2 regularization:** add the sum of the squared weights to the loss function

L1 Regularization

$$\text{Modified loss function} = \text{Loss function} + \lambda \sum_{i=1}^{n} |W_i|$$

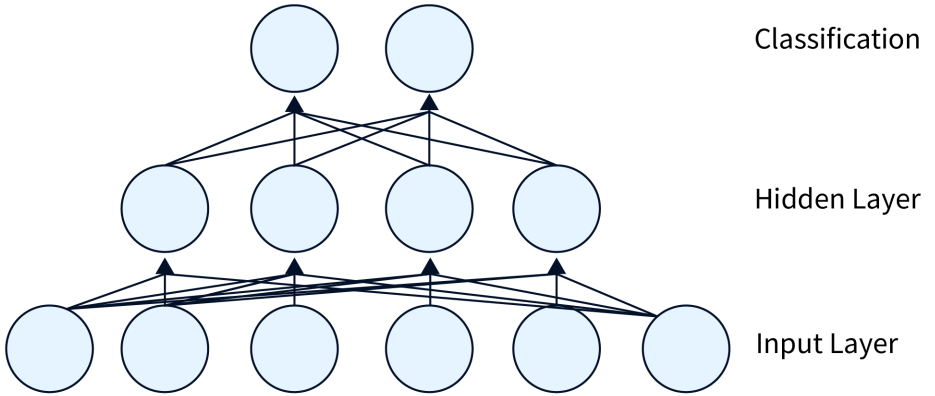
L2 Regularization

$$\text{Modified loss function} = \text{Loss function} + \lambda \sum_{i=1}^{n} W_i^2$$

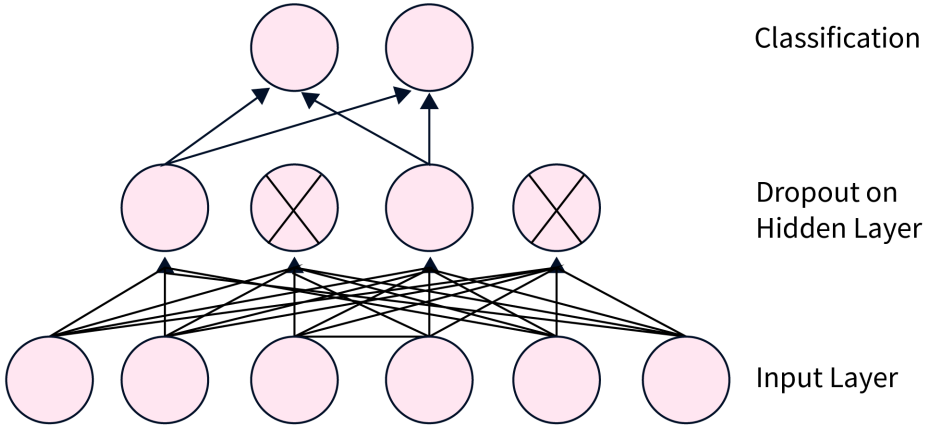
Avoiding Overfitting: Other Methods

Other methods can be used to limit overfitting:

- **Early stopping:** In some cases, it can be useful to configure early stopping methods, designed to stop training before the specified number of total epochs following a specific criteria (e.g. validation loss has not decreased)
- **Dropout:** During the forward pass, randomly set a portion of neurons to 0 in specified hidden layers. This can improve the generalization of the model by requiring more robust features.



Without Dropout



With Dropout

https://www.scaler.com/topics/dropout-tensorflow/

Deep learning for Time Series
Conclusion

Deep Learning for Time Series: Applications

Tasks:

- Forecasting
- Classification
- Imputation
- Anomaly detection

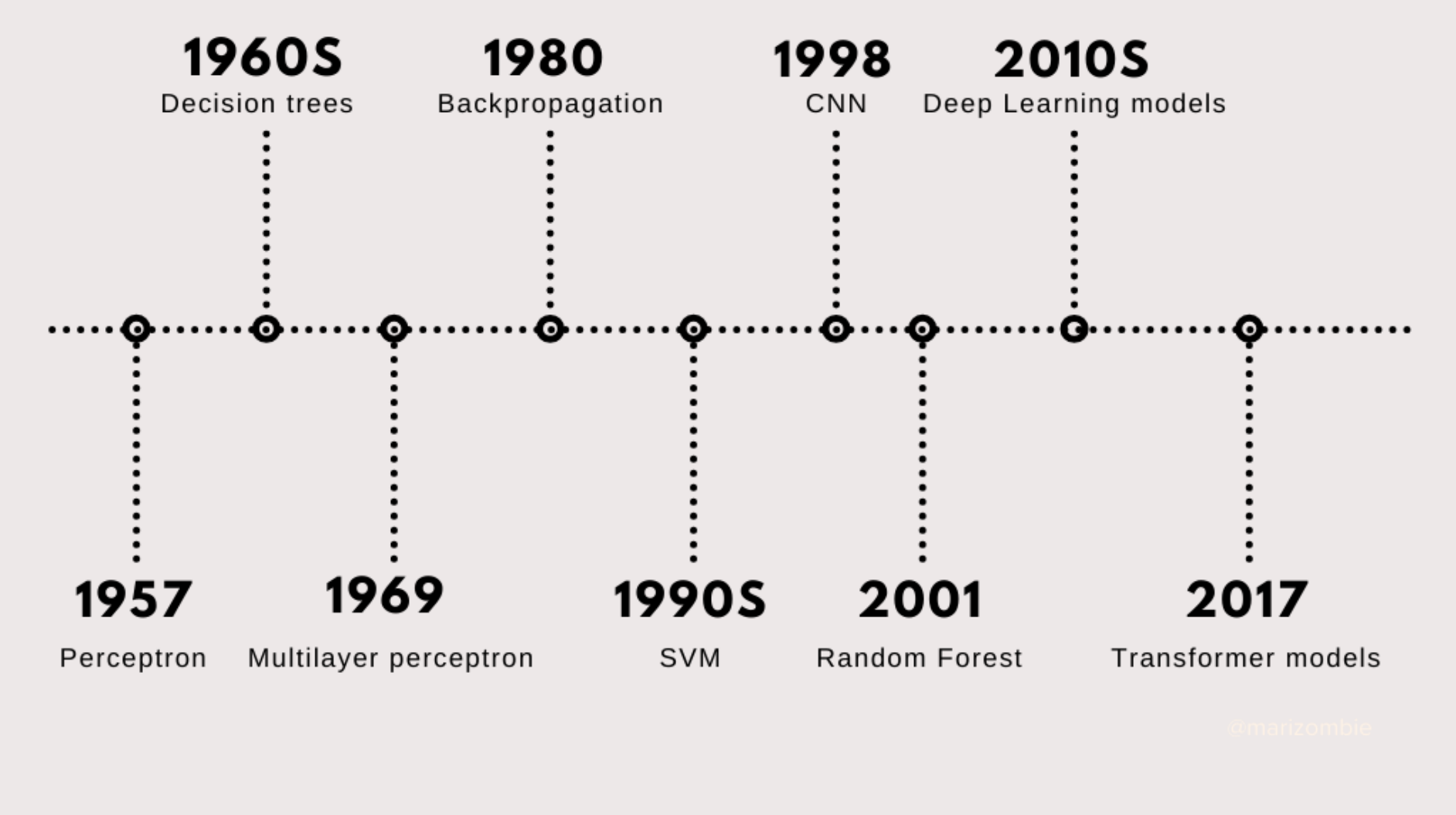
Domains:

- Biosignals
- Speech
- Finance
- Climate

In This Lecture

- **Perceptron**
- **Multi-Layer Perceptron**
- **Gradient Descent**
- **Backpropagation**
- **Evaluation**

Timeline of Deep Learning



https://medium.com/@marizombie/important-dates-in-history-of-machine-learning-d52d7f24b797

Deep Learning Frameworks & Libraries

Main frameworks:

- Pytorch
- Tensorflow

Library for open source models:

- Transformers (Hugging Face)

Other useful libraries

- Pytorch-lightning
- Tensorboard