# Neural Networks 1 - Linear Neurons

## 18NES1 - Lecture 4, Summer semester 2024/25

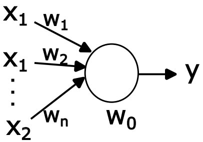
Zuzana Petříčková

March 11, 2025

# What We Covered Last Time

## Perceptron with a Step Activation Function and Its Learning Algorithms

- Internal potential:
  $\xi = \sum_{i=1}^{n} w_i x_i + w_0$
- Output: $y = f(\xi)$




- Learning algorithms:
  - Rosenblatt's learning algorithm and its variants
  - Hebbian learning

- **Step** activation function:

$$f(\xi) = \begin{cases} 1 & \text{for } \xi > 0 \quad \text{... neuron is active} \\ -1 & \text{for } \xi < 0 \quad \text{... neuron is passive (inactive)} \\ 0 & \text{for } \xi = 0 \quad \text{... neuron is silent} \end{cases}$$

# Examples - Various Practical Tasks - Completion

**Example 5 - Letters letters_example.ipynb**

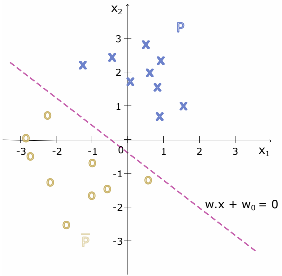
- We use the prepared dataset **letters.csv**
- The letters were segmented from **letters.png**
- Explore the dataset and visualize some of the letters
- Create a test set: data with added noise or subsequently smoothed
- Train a perceptron using different learning algorithms (and their variants) to recognize individual letters
- Determine the classification error on the training set as well as on the test sets (optionally include the number of epochs / training time)
- How much noise in the data could the perceptron still handle?
- Identify which letters the perceptron had the most trouble recognizing
- Which learning algorithm performed the best?

# Examples - Various Practical Tasks - Completion

**Example 6 - Handwritten Digits digits_example.ipynb**

- We use the prepared dataset **OcrData.csv** containing handwritten digits
- Explore the dataset and visualize some digits (use the provided script)
- Train a perceptron using different learning algorithms (and their variants) to recognize individual digits
- Determine (and compare) the classification error on the training set (optionally include the number of epochs / training time)
- Identify which digits the perceptron had the most trouble recognizing
- Which learning algorithm performed the best?

# Perceptron with a Step Activation Function




**Applications:**

- Linear classifier for two classes
- Implementation of logical functions

**Problem:** If the data is not linearly separable (e.g., XOR)
**What can we do?**

1. Quadratic or cubic expansion of the feature space
   e.g., $x_1, x_2, x_1^2, x_2^2, x_1 x_2$
2. A neural network with more perceptrons and layers ... but how do we train it? :(

→ What if we use a **continuous** activation function instead of a step function?

→ This allows us to solve other types of tasks (e.g., regression).

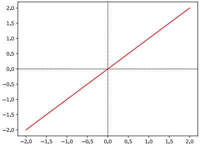
# Today's Lesson

1. **Linear Neuron and the Task of Linear Regression**
   - Training a linear neuron using the Least Squares (LSQ) method
   - Training a linear neuron using the Gradient Descent method

# Linear Neuron

- One of the oldest models: ADALINE (Adaptive Linear Element, 1960, Widrow-Hoff)

- **Identity** activation function: $f(\xi) = \xi$
- Neuron output:
  $y = \xi = \sum_{i=1}^{n} w_i x_i + w_0 = \vec{x}\vec{w} + w_0$
  ($\vec{w}$ is a column vector)




**Learning objective:**

- We have a training dataset in the form $T = (X, \vec{d})$

| | | | |
|---|---|---|---|
| $x_{11}$ | $\dots$ | $x_{1n}$ | $d_1$ |
| $\dots$ | | $\dots$ | $\dots$ |
| $x_{N1}$ | $\dots$ | $x_{Nn}$ | $d_N$ |

- Neuron output in matrix form: $\vec{y} = X\vec{w} + w_0$

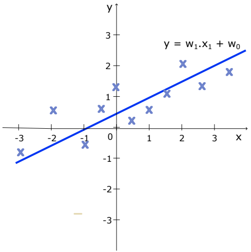
$\rightarrow$ We seek $\vec{w}$ such that ideally: $\vec{d} = \vec{y}$, i.e., $\vec{d} = X\vec{w} + w_0$

- This is a **linear regression** problem.

# Linear Neuron - Geometric Interpretation

**For a single input feature:**

- Neuron output: $y = w_1 x + w_0$
- $(x_k, d_k)$ are points in the plane
- We fit the points with a straight line:




**In general:** We fit the points with a hyperplane
$y = w_1 x_1 + \cdots + w_n x_n + w_0$

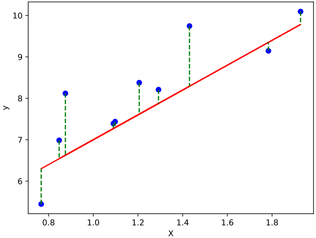
- assuming a linear relationship between input variables $x_1, \ldots, x_n$ and the output $y$

# How to Train a Linear Neuron (i.e., Linear Regression Model)?

- Given a training sample: $(\vec{x}_p, d_p)$
- Compute the actual neuron output: $y_p = \vec{x}_p \vec{w} + w_0$
- The actual and desired outputs differ:

$$y_p = d_p + e_p \quad \text{where } e_p \text{ is the error for a single sample}$$

- We want the actual and desired outputs to be as close as possible for each training sample:



# Training a Linear Neuron (i.e., Linear Regression Model)

**How do we define an error function to minimize during training?**

1. SAE (Sum of Absolute Errors)

$$E = \sum_p |e_p| = \sum_p |d_p - y_p|$$

   - **Disadvantage:** Absolute function is not continuously differentiable, making optimization difficult.

2. SSE/SSQ (Sum of Squared Errors) – **Least Squares Method**

$$E = \frac{1}{2} \sum_p e_p^2 = \frac{1}{2} \sum_p (d_p - y_p)^2$$

   - Quadratic function is continuously differentiable, allowing for efficient optimization
   - It penalizes large deviations more strongly than small ones making it sensitive to outliers

# Training a Linear Neuron (i.e., Linear Regression Model)

**Least Squares Method**

- Minimize

$$E = \frac{1}{2} \sum_p (d_p - y_p)^2$$

**How do we do this?**

1. LSQ method – based on an explicit calculation
2. Gradient method (steepest descent method)

# Linear Neuron - Learning Using the LSQ Method

- We have an extended training dataset in the form $T = (X, \vec{d})$

$$\begin{array}{|cccc|c|} \hline x_{10} = 1 & x_{11} & \dots & x_{1n} & d_1 \\ \dots & \dots & & \dots & \dots \\ x_{N0} = 1 & x_{N1} & \dots & x_{Nn} & d_N \\ \hline \end{array}$$

$\rightarrow$ We seek $\vec{w}$ such that: $\vec{d} = \vec{y}$, i.e., $\vec{d} = X\vec{w}$

This leads to solving the system of equations:

$$\begin{array}{ccccccccc} w_0 x_{10} & + & w_1 x_{11} & + & \dots & + & w_n x_{1n} & = & d_1 \\ \dots & \dots & & \dots & \dots & & \dots & & \dots \\ w_0 x_{N0} & + & w_1 x_{N1} & + & \dots & + & w_n x_{Nn} & = & d_N \end{array}$$

# Linear Neuron - Learning Using the LSQ Method

**When does the system have a unique solution?**

- Condition: The columns of $X$ must be linearly independent, i.e., $h(X) = n + 1$
- Rank condition: $h(X|\vec{d}) = h(X)$

**In general:**

- The system may have infinitely many solutions (or none)
- The objective is to minimize the sum of squared errors:

$$\frac{1}{2}\sum_{p=1}^{N}(d_p - y_p)^2 = \min, \quad ||X\vec{w} - \vec{d}||^2 = \min$$

$\rightarrow$ Setting the derivative of this function to zero, after some algebraic manipulation, we obtain:

$$(X^T X)\vec{w} - X^T\vec{d} = 0$$

# Linear Neuron - Learning Using the LSQ Method

**When does the system have a unique solution?**

- Condition: The columns of $X$ must be linearly independent, i.e., $h(X) = n + 1$
- Rank condition: $h(X|\vec{d}) = h(X)$

**In general:**

- The system may have infinitely many solutions (or none)
- The objective is to minimize the squared error:

$$\frac{1}{2}\sum_{p=1}^{N}(d_p - y_p)^2 = \min, \quad ||X\vec{w} - \vec{d}||^2 = \min$$

**Alternative derivation (Gauss):**

$$X\vec{w} = \vec{d}$$

$$X^T(X\vec{w}) = X^T\vec{d}$$

$$(X^TX)\vec{w} = X^T\vec{d}$$

# Linear Neuron - Learning Using the LSQ Method

$$(X^T X)\vec{w} = X^T \vec{d}$$

1. If the inverse matrix exists, i.e., $det(X^T X) \neq 0$:
$$\vec{w} = (X^T X)^{-1} X^T \vec{d}$$
2. If $det(X^T X) = 0$ (the system has infinitely many or no solutions):
$\rightarrow$ Apply regularization (using the pseudoinverse matrix):
   1. **Tikhonov regularization (ridge regression):**
$$\vec{w} = (X^T X + \lambda I)^{-1} X^T \vec{d}, \quad \lambda > 0$$
   2. **Moore-Penrose pseudoinverse** (solution with the smallest weights):
$$\vec{w} = \lim_{\lambda \to 0+} (X^T X + \lambda I)^{-1} X^T \vec{d}$$

# Linear Neuron – Training with LSQ Method

**Example 1** ... $\vec{w} = (X^T X)^{-1} X^T \vec{d}$

| $x_0$ | $x_1$ | $x_2$ | $d$ |
|---|---|---|---|
| +1 | -1 | -1 | +1 |
| +1 | -1 | +1 | +1 |
| +1 | +1 | -1 | +1 |
| +1 | +1 | +1 | -1 |

$$X^T X = \begin{pmatrix} 4 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 4 \end{pmatrix}$$

$$(X^T X)^{-1} = \begin{pmatrix} \frac{1}{4} & 0 & 0 \\ 0 & \frac{1}{4} & 0 \\ 0 & 0 & \frac{1}{4} \end{pmatrix}$$

# Linear Neuron – Training with LSQ Method

**Example 1** ... $\vec{w} = (X^T X)^{-1} X^T \vec{d}$

| $x_0$ | $x_1$ | $x_2$ | $d$ |
|---|---|---|---|
| +1 | -1 | -1 | +1 |
| +1 | -1 | +1 | +1 |
| +1 | +1 | -1 | +1 |
| + 1 | +1 | +1 | -1 |

$$\vec{w} = \begin{pmatrix} \frac{1}{4} & 0 & 0 \\ 0 & \frac{1}{4} & 0 \\ 0 & 0 & \frac{1}{4} \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 1 \\ -1 & -1 & 1 & 1 \\ -1 & 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ -1 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ -\frac{1}{4} & -\frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & \frac{1}{4} \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ -1 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ -\frac{1}{2} \end{pmatrix}$$

# Linear Neuron – Training with LSQ Method

**Example 2** ... $\vec{w} = (X^T X)^{-1} X^T \vec{d}$

| $x_0$ | $x_1$ | $x_2$ | $d$ |
|---|---|---|---|
| +1 | +1 | -1 | +1 |
| +1 | +1 | +1 | -1 |

$$X^T X = \begin{pmatrix} 2 & 2 & 0 \\ 2 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix}$$

$$h(X^T X) = 2 \rightarrow det(X^T X) = 0 \rightarrow (X^T X)^{-1} \text{ does not exist}$$

We apply regularization:

$$\vec{w} = (X^T X + \lambda I)^{-1} X^T \vec{d}, \lambda > 0$$

$$\vec{w} = \lim_{\lambda \to 0+} (X^T X + \lambda I)^{-1} X^T \vec{d}$$

# Linear Neuron – Training with LSQ Method

**Example 2** ... $\vec{w} = (X^T X + \lambda I)^{-1} X^T \vec{d}, \lambda > 0$

| $x_0$ | $x_1$ | $x_2$ | $d$ |
|---|---|---|---|
| +1 | +1 | -1 | +1 |
| +1 | +1 | +1 | -1 |

$$X^T X + \lambda I = \begin{pmatrix} 2+\lambda & 2 & 0 \\ 2 & 2+\lambda & 0 \\ 0 & 0 & 2+\lambda \end{pmatrix}$$

After further computations:

$$(X^T X + \lambda I)^{-1} = \begin{pmatrix} \frac{2+\lambda}{\lambda^2+4\lambda} & -\frac{2}{\lambda^2+4\lambda} & 0 \\ -\frac{2}{\lambda^2+4\lambda} & \frac{2+\lambda}{\lambda^2+4\lambda} & 0 \\ 0 & 0 & \frac{1}{2+\lambda} \end{pmatrix}$$

# Linear Neuron – Training with LSQ Method

**Example 2** ... $\vec{w} = (X^T X + \lambda I)^{-1} X^T \vec{d} = X^+ \vec{d}, \lambda > 0$

$$X^+ = (X^T X + \lambda I)^{-1} X^T = \begin{pmatrix} \frac{2+\lambda}{\lambda^2+4\lambda} & -\frac{2}{\lambda^2+4\lambda} & 0 \\ -\frac{2}{\lambda^2+4\lambda} & \frac{2+\lambda}{\lambda^2+4\lambda} & 0 \\ 0 & 0 & \frac{1}{2+\lambda} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 1 \\ -1 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{\lambda}{\lambda^2+4\lambda} & \frac{\lambda}{\lambda^2+4\lambda} \\ \frac{\lambda}{\lambda^2+4\lambda} & \frac{\lambda}{\lambda^2+4\lambda} \\ -\frac{1}{2+\lambda} & \frac{1}{2+\lambda} \end{pmatrix} = \begin{pmatrix} \frac{1}{\lambda+4} & \frac{1}{\lambda+4} \\ \frac{1}{\lambda+4} & \frac{1}{\lambda+4} \\ -\frac{1}{2+\lambda} & \frac{1}{2+\lambda} \end{pmatrix}$$

# Linear Neuron – Training with LSQ Method

**Example 2** ... $\vec{w} = (X^T X + \lambda I)^{-1} X^T \vec{d} = X^+ \vec{d}, \lambda > 0$

- $\lambda = 1$:
$$\vec{w} = X^+ \vec{d} = \begin{pmatrix} \frac{1}{5} & \frac{1}{5} \\ \frac{1}{5} & \frac{1}{5} \\ -\frac{1}{3} & \frac{1}{3} \end{pmatrix} \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -\frac{2}{3} \end{pmatrix}$$
- $\lambda = \frac{1}{10}$:
$$\vec{w} = X^+ \vec{d} = \begin{pmatrix} \frac{10}{41} & \frac{10}{41} \\ \frac{10}{41} & \frac{10}{41} \\ -\frac{10}{21} & \frac{10}{21} \end{pmatrix} \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -\frac{20}{21} \end{pmatrix}$$
- $\lambda \to 0$:
$$\vec{w} = X^+ \vec{d} = \begin{pmatrix} \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} \\ -\frac{1}{2} & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$

# Examples – Demonstration of LSQ Learning Algorithms in Python

**linear_neuron.ipynb**

- LSQ method applied to Examples 1 and 2 from the slides
- Three different algorithm implementations:
  - **Using a library function** (standard linear regression, LSQ)
  - Custom implementation **using the pseudoinverse matrix** (Moore-Penrose pseudoinverse, roughly equivalent to the library version)
  - Custom implementation **using Tikhonov regularization** (coarser approximation but can handle "difficult" cases)
- Observation: A linear neuron is not particularly suitable for classification tasks

# Examples – Linear Regression Task

**Example 3 and Example 4 – Linear Regression in One-Dimensional and Two-Dimensional Input Space**

- Artificially generated data: training samples are generated based on a known function with added random noise
- By examining the learned weights, we can easily determine whether the neuron has correctly learned the task
- We can experiment with different levels of noise in the training set
- We will visualize the resulting regression line/plane
- Questions: How close are the learned weights and bias to the actual values? Check the error values (MSE and SSE).

# Advantages and Disadvantages of LSQ Learning

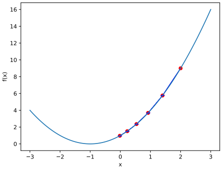
**Advantages:**

- Provides an exact analytical solution if the inverse of $X^T X$ exists
- Computationally efficient for small datasets (direct matrix inversion)
- Works well when data is linearly related
- Can be extended with regularization techniques (e.g., Moore-Penrose, Tikhonov)

**Disadvantages:**

- Sensitive to noise and outliers in the data
- Computationally expensive for large datasets (inverting large matrices is costly)
- Regularization is necessary in ill-conditioned cases where $X^T X$ is singular
- Poor performance for classification tasks (linear regression is not ideal for binary/multiclass classification)

# Introduction: Gradient Descent Method (Steepest Descent)




**Problem Definition:**

- We have a function $f(\vec{x}) : \mathbb{R}^n \rightarrow \mathbb{R}$
- We seek $\vec{x}$ such that $f(\vec{x})$ is minimized

$\rightarrow$ **Solution (gradient descent method):**

1. Start at an (random) initial point $\vec{x}(0)$
2. Compute the gradient: $\nabla f(\vec{x}) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right)$ The gradient represents the direction and magnitude of the greatest increase in $f(\vec{x})$
3. Iteratively move in small steps opposite to the gradient direction: $\vec{x}(t+1) = \vec{x}(t) - \alpha \nabla f(\vec{x})$ $\alpha$ is a small positive number (step size, learning rate)
4. For a single input feature: $x_i(t+1) = x_i(t) - \alpha \frac{\partial f}{\partial x_i}$

# Challenges in Gradient Descent

**Common Issues:**

- Small $\alpha$ leads to slow convergence
- Large $\alpha$ causes oscillations (overshooting)
- May converge to a local minimum instead of a global minimum

**How to Adjust the Learning Rate?**

- Start with an initial value $1 \gg \alpha_0 > 0$ and gradually decrease it
- Use a decreasing sequence:

$$\sum_{i=0}^{\infty} \alpha_i = \infty, \quad \sum_{i=0}^{\infty} \alpha_i^2 < \infty$$

- Heuristic approach:

$$\alpha_j = \frac{\alpha_0}{1+j} \quad \text{(Robbins-Monro, 1951)}$$

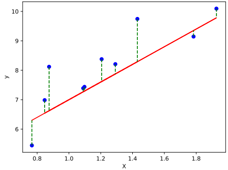
# Training a Linear Neuron Using Gradient Descent

**Reminder: Linear Neuron (Linear Regression Model)**

- Neuron output: $y = \xi = \sum_{i=0}^{n} w_i x_i = \vec{x}\vec{w}$
- In matrix form: $\vec{y} = X\vec{w}$ ($\vec{w}$ is a column vector)

**Least Squares Method**

- We want the neuron's actual output $y_p$ to be as close as possible to the desired output $d_p$




- Minimize the sum of squared errors (SSE):

$$E = \frac{1}{2}\sum_{p=1}^{N}(d_p - y_p)^2$$

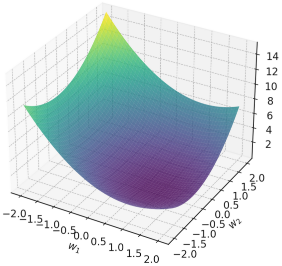
# Gradient Descent for Training a Linear Neuron

## Solution Using Gradient Descent

- We minimize the SSE loss function in weight space:

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

- $E_p(\vec{w})$ is the error function for a single sample




- The loss function is quadratic, convex, meaning gradient descent should reliably find its global minimum with appropriate parameter tuning.

# Gradient Descent for Training a Linear Neuron

**Gradient Computation:**

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

Compute the partial derivatives:

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

**Weight Update Rule:**

$$w_i(t+1) = w_i(t) - \alpha\frac{\partial E_p}{\partial w_i} = w_i(t) + \alpha(d_p - y_p)x_{pi}$$

**Vectorized Form:**

$$\vec{w}(t+1) = \vec{w}(t) - \alpha\nabla E_p(\vec{w}) = \vec{w}(t) + \alpha(d_p - y_p)\vec{x}_p^T$$

# Training a Linear Neuron Using Gradient Descent

**General Algorithm Scheme (GD, Gradient Descent)**

1. Initialize weights with small random real values:

$$\vec{w}(0) = (w_0, w_1, ..., w_n)^T$$

   Initialize the learning rate $\alpha_0$ with a small positive value:

$$1 \gg \alpha_0 > 0$$

2. Present the next training sample $(\vec{x}_t, d_t)$ and compute the neuron's actual output:

$$y_t = \vec{x}_t \vec{w}$$

3. Update the weights:

$$\vec{w}(t+1) = \vec{w}(t) + \alpha_t (d_t - y_t) \vec{x}_t^T$$

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

# Training a Linear Neuron Using Gradient Descent

**How to Present Training Samples?** Different Strategies:

1. **Iterative per epoch (Online GD)**:
   - Each sample is presented exactly once per epoch, with a random order within each epoch.
   - The number of epochs determines how many times the entire training set is presented.
2. **Batch processing per epoch (Batch GD)**:
   - The entire training set is presented at once, and weights are updated collectively:

$$\vec{y} = X\vec{w}$$

$$\vec{w}(t+1) = \vec{w}(t) + \alpha_t X^T(\vec{d} - \vec{y})$$

3. **Mini-batch processing (SGD, Stochastic Gradient Descent)**:
   - The training set is randomly split into batches (mini-batches), and weights are updated batch by batch.

# Training a Linear Neuron Using Gradient Descent

**Comparison of Training Strategies**

1. **Online GD (per sample)**:
   - Fast training (in number of epochs) but unstable (reducing the error for the current sample may increase the error for others).
   - Greater randomness, more sensitivity to outliers and hyperparameter choices (e.g., learning rate).
2. **Batch GD (per full dataset)**:
   - Stable learning process.
   - Efficient for small datasets but has high memory requirements for large datasets.
3. **Mini-batch SGD (hybrid approach)**:
   - Combines advantages of both methods.
   - Commonly used for deep learning and large-scale datasets.

# Training a Linear Neuron Using Gradient Descent

**How to Initialize Weights?**

- Learning should start from a random point.
- Weights should be initialized with small random values (preferably centered around 0) instead of setting them to zero.
- Large or biased initial weights may lead to poor learning performance.

**Constant vs. Adaptive Learning Rate**

- A constant learning rate may cause the algorithm to oscillate at the end of training.
- The algorithm is highly sensitive to the choice of learning rate.

**How and When to Update the Learning Rate?**

- Typically updated once per epoch:

$$\alpha_e = \frac{\alpha_0}{e}, \quad \alpha_e = \frac{\alpha_0}{\sqrt{e}}$$

# Stopping Criteria for Training

## When to Stop Training?

- Several strategies can be applied:
  1. A predefined number of epochs.
  2. When the average error falls below a threshold:

$$E < E_{min}$$

  3. When the validation error stops decreasing (**early stopping**).
  4. When weight updates become too small:

$$|\Delta w| < \delta_{min}$$

## Early Stopping – Preventing Overfitting

- Uses an independent dataset – **validation set**.
  - It should be entirely separate from the training set.
  - It allows continuous monitoring of model generalization.
- If validation error increases for several consecutive epochs, training is stopped.

# Feature Normalization in Gradient-Based Learning

**Why Normalize Input Features?**

- Large input values may cause instability during training (affecting learning speed and generalization).

**Normalization Methods:**

- Min-max normalization to the range $[-1, 1]$:

$$X_{ij}^{new} = 2 \cdot \frac{X_{ij} - m_j}{M_j - m_j} - 1$$

  where $m_j = \min_k(X_{kj})$, $M_j = \max_k(X_{kj})$.

- Standardization using mean and standard deviation:

$$X_{ij}^{new} = \frac{X_{ij} - E(X_{kj})}{S(X_{kj})}$$

  - $E(X_{kj}) = \frac{1}{N} \sum_{k=1}^{N} X_{kj}$ is the mean of column $j$ in matrix $X$.
  - $S(X_{kj}) = \frac{1}{N-1} \sum_{k=1}^{N} (X_{kj} - E(X_{kj}))^2$ is the standard deviation of column $j$.

# Evaluating Regression Performance of a Linear Neuron

**Error Metrics:**

- SAE (Sum Absolute Error): $E(\vec{w}) = \sum_{p=1}^{N} |d_p - y_p|$
- SSE (Sum Squared Error): $E(\vec{w}) = \sum_{p=1}^{N} (d_p - y_p)^2$
- MAE (Mean Absolute Error) – readable for humans, represents the average deviation from expected values:

$$E(\vec{w}) = \frac{1}{N} \sum_{p=1}^{N} |d_p - y_p|$$

- MSE (Mean Squared Error) – commonly used for comparison:

$$E(\vec{w}) = \frac{1}{N} \sum_{p=1}^{N} (d_p - y_p)^2$$

# Evaluating Regression Performance of a Linear Neuron

**Assessing Whether the Model Has Learned the Regression Task Well**

- Compute MSE (Mean Squared Error) and SSE (Sum of Squared Errors) on the **training set**.
- To evaluate the model's generalization ability, compute MSE and SSE on the **test set** as well.
  - The test set should be entirely independent of the training and validation sets, containing completely unseen samples.

**How to Create a Validation and Test Set?**

- For synthetic tasks, we can generate them randomly (e.g., from the same probability distribution, possibly adding additional noise).
- For real-world datasets, it is common practice to randomly split the data into training, validation, and test subsets, typically in a 70-15-15 ratio.

# Examples – Demonstration of Gradient Descent in Python

**linear_neuron.ipynb**

**Gradient Descent on Examples 1 and 2 from the Slides**

- We will demonstrate the crucial role of hyperparameter selection in gradient descent (learning rate, whether it is adaptive, proper weight initialization).
- Comparison of gradient descent with the LSQ method:
  - **Observation:** Gradient descent requires careful tuning of hyperparameters and for small taksks it is more computationally demanding.
  - However, for more complex tasks, gradient descent can yield better results than LSQ.
- We will illustrate how hyperparameters can be fine-tuned step by step when solving a specific task.

# Examples – Linear Regression Task

**Example 3 and Example 4 – Linear Regression in One-Dimensional and Two-Dimensional Input Spaces**

- We will demonstrate the use of test and validation datasets during training and in evaluating how well the linear neuron has learned the task.
- Again, we will experiment with hyperparameter settings and attempt to fine-tune them for the given task.
- We will compare iterative and batch gradient descent.
- We will apply the early stopping technique.

# Example – Optional Homework for Bonus Points

- Modify dataset 4 (define your own unique linear function with unique noise).
- Experiment with hyperparameter settings for this task (learning rate, number of epochs, training strategies, etc.) and optimize them.
- Briefly evaluate your experiment results (which hyperparameter settings would you recommend for this task and why?).
- Compare the best results achieved with the LSQ method.
- Submit the final notebook (including textual evaluation) via email.

# Gradient Descent – Summary

- Gradient descent can solve the linear regression task as effectively as the classical LSQ method, **BUT** it is more challenging to apply:
  - It is a local optimization method – results may vary slightly with each run.
  - The method is highly sensitive to proper hyperparameter tuning.
- The advantage over LSQ is that gradient descent can be applied to a significantly broader range of problems where classical LSQ fails.