Neural Networks 1 - Perceptrons with continuous activations

18NES1 - Lecture 5, Summer semester 2024/25

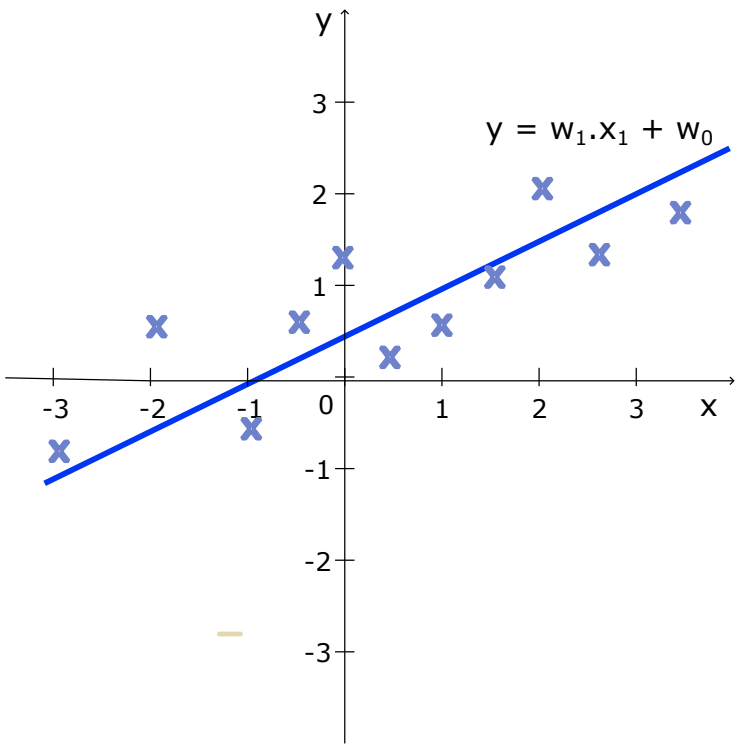
Zuzana Petříčková

March 18, 2025

Review of the Previous Lecture

Linear Neuron and the Linear Regression Task

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



- We fit the points in the input space with a hyperplane (a line, a plane, etc.).
- We assume a linear relationship between the input variables $x_1, ..., x_n$ and the output y.

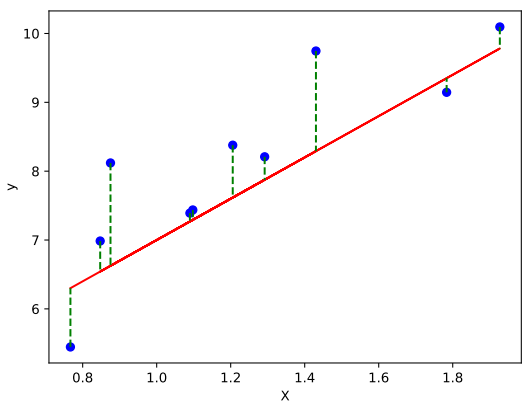
Review of the Previous Lecture

Linear Neuron and the Linear Regression Task

- During training, we minimize the difference between the actual and desired output for each training sample.
- Instead of SAE (sum of absolute errors), we minimize the quadratic error function (sum of squared errors, SSE):

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

$\rightarrow$ Least Squares Methods (LSQ)

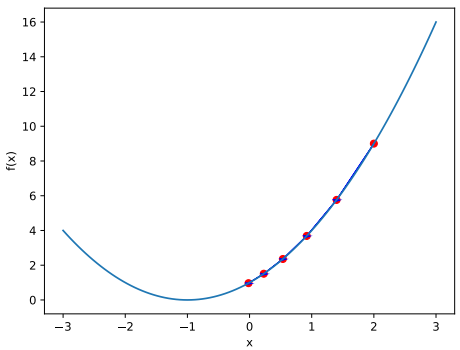


- Learning algorithms for a linear neuron (linear regression):
  - **LSQ Method** - based on explicit computation.
  - **Gradient Descent Method** - a local optimization method (with broader applicability).

Training a Linear Neuron Using Gradient Descent

General Algorithm Scheme (Gradient 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



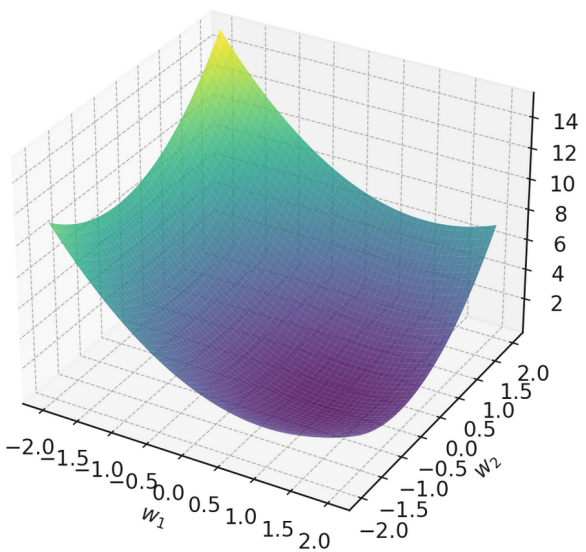
- **Solution (gradient descent method):**
  1. Start at an (random) initial point $\vec{x}(0)$
  2. Compute the gradient: $\nabla f(\vec{x}) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right)$ The gradient represents the direction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

Training a Linear Neuron 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.

Training a Linear Neuron Using Gradient Descent

General Algorithm Scheme (Gradient Descent)

1. Initialize weights with small random values:
   $\vec{w}(0) = (w_0, w_1, ..., w_n)^T$
   Initialize learning rate: α_0, where $1 \gg \alpha_0 > 0$.
2. Present the next training sample $(\vec{x}_t, d_t)$ and compute the neuron output:

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

3. Update weights (move in the direction opposite to the gradient of the loss function):

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

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

Gradient Descent

- Previously, we explored different variants of gradient descent, its hyperparameters, and demonstrated the importance of proper hyperparameter tuning through examples.

Conclusion

- Gradient descent can solve linear regression as effectively as the classical LSQ method, **BUT** it is more challenging to apply:
  - It is a local optimization method: results may vary across runs.
  - It is highly sensitive to hyperparameter tuning, weight initialization, and data preprocessing.
- Compared to LSQ, gradient descent is:
  - More robust to numerical errors.
  - Applicable to a broader range of problems where LSQ struggles:
    - Large datasets (many features or samples).
    - Data with outliers or with significant noise.

Today's Lecture

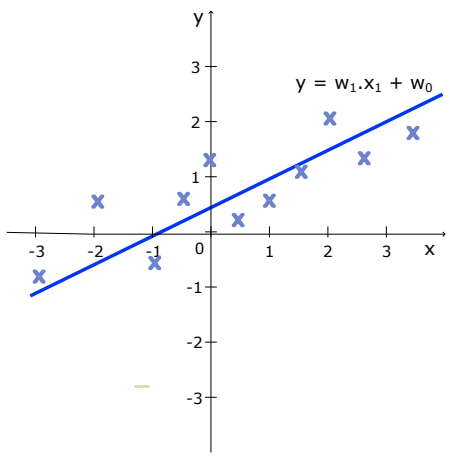
1. Perceptron with a Continuous Activation Function
   - The most common activation functions for a perceptron
   - Training a general perceptron using the gradient descent method
2. Neural network with a single layer of neurons

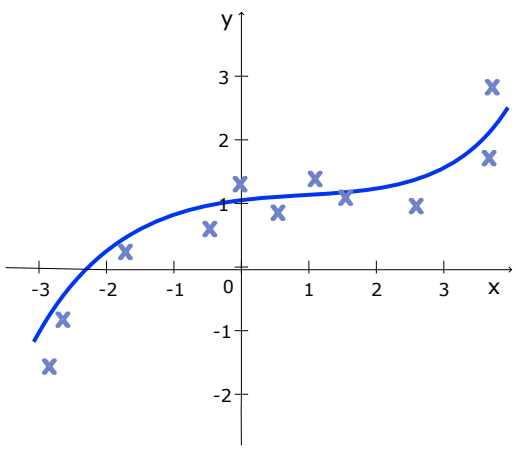
From Linear Regression to a More General Perceptron Model

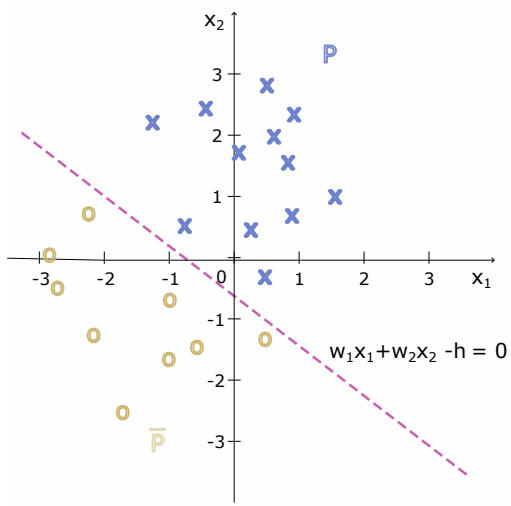
- Using gradient descent, we can train a linear neuron and solve the linear regression problem.

Question: Could the gradient descent method be used for other activation functions?

- If so, we could solve nonlinear regression or classification tasks:





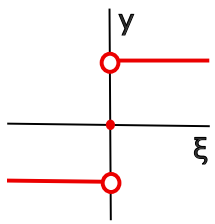


From Linear Regression to a More General Perceptron Model

Question: Could the gradient descent method be used for other activation functions?

- Yes, but only for certain types: the activation function must be **continuous** and **differentiable**.

$\rightarrow$ It cannot be applied to a step function: (Question: What is its derivative?)

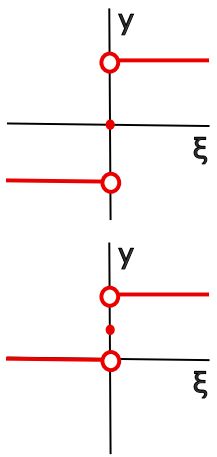


$\rightarrow$ Could we replace the step function with another activation function that is continuous and differentiable?

Searching for Continuous Functions that Resemble the Step Activation Function

For the bipolar neuron model

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



For the binary neuron model

$$f(\xi) = signum(\xi) = \begin{cases} 1 & \text{for } \xi > 0 \quad \text{... neuron is active} \\ 0.5 & \text{for } \xi = 0 \quad \text{... neuron is silent} \\ 0 & \text{for } \xi < 0 \quad \text{... neuron is passive} \end{cases}$$

Searching for Continuous Functions that Resemble the Step Activation Function

Requirements:

- $f(y)$ is defined and continuous on $(-\infty, \infty)$
- $lim_{y \to -\infty} f(y) = m < M = lim_{y \to \infty} f(y)$
- $f(y) \approx m$ ... neuron is passive
- $f(y) \approx M$ ... neuron is active

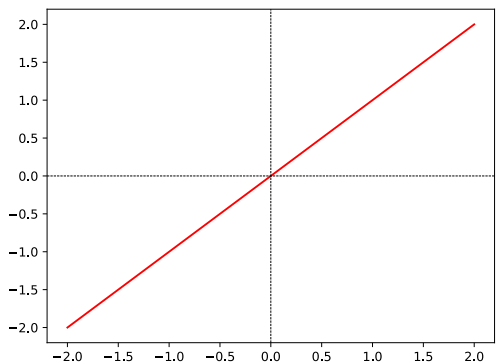
Typical examples:

- $(m, M) = (-\infty, \infty)$ ... linear function
- $(m, M) = (0, 1)$ - binary model ... sigmoid function
- $(m, M) = (-1, 1)$ - bipolar model ... hyperbolic tangent (tanh)

Continuous Activation Functions

Linear (Identity)

- $f(\xi) = \xi$



Usage

- Not suitable for classification tasks
- Highly suitable for regression tasks

Where it is used:

- Single-layer linear neural networks
- Output layer of multilayer/deep networks for regression tasks

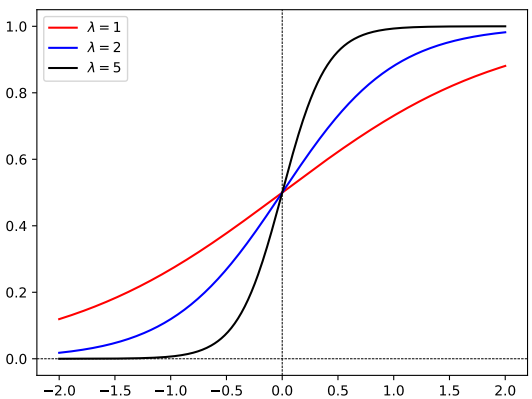
Continuous Activation Functions

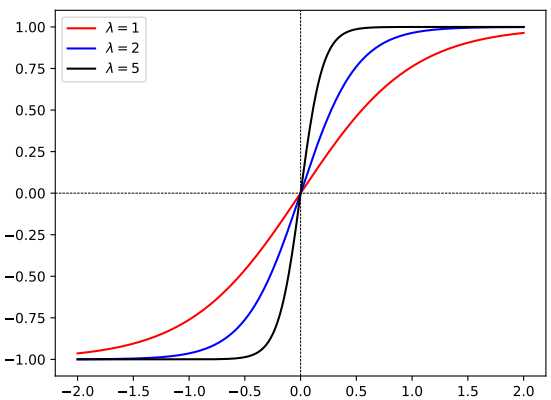
Sigmoidal

- $f(\xi) = \frac{1}{1+e^{-\lambda\xi}}$ ... logistic sigmoid (logsig)
- binary model

Hyperbolic Tangent

- $f(\xi) = \frac{1-e^{-2\lambda\xi}}{1+e^{-2\lambda\xi}}$ ... tanh
- bipolar model

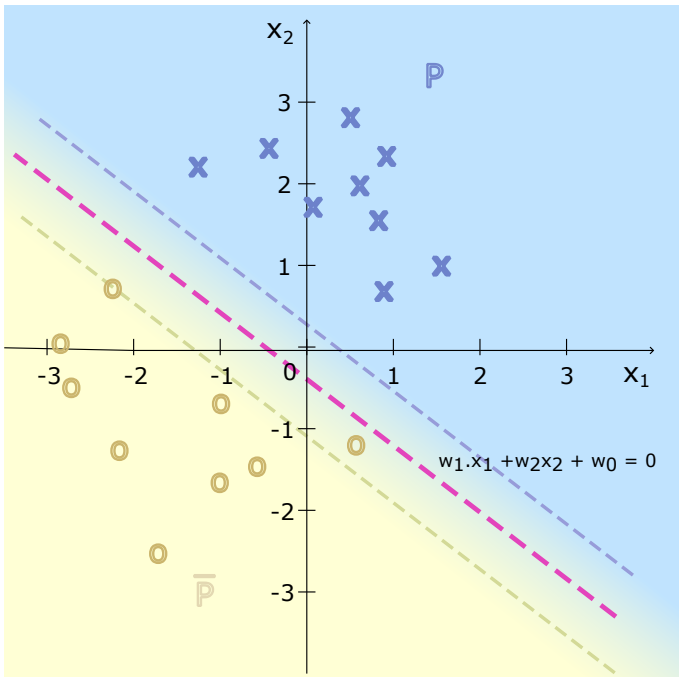




→ "Smoothed" versions of the step activation function

Continuous Activation Functions

Sigmoid and Hyperbolic Tangent - Geometric Interpretation:



... Linear classification task

- With a sigmoidal activation function, the neuron's output can be directly interpreted as the **probability** that the input belongs to a given class.

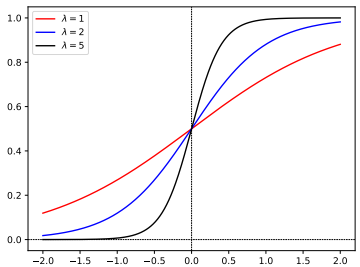
Continuous Activation Functions

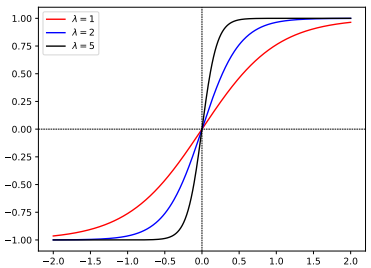
Sigmoidal

- $f(\xi) = \frac{1}{1+e^{-\lambda\xi}}$ ... logistic sigmoid (logsig)

Hyperbolic Tangent

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





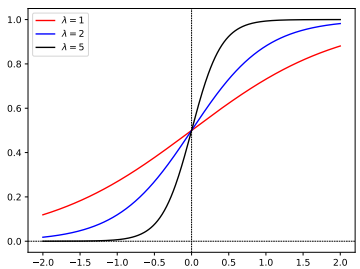
Parameter λ - Slope

- Determines classification uncertainty near decision boundaries
  - $\lambda \to \infty$ ... step activation function
  - Smaller λ ... wider boundary between classes
  - $\lambda \to 0$ ... neuron does not distinguish between the classes (output is always 0.5 or 0)
  - Typical choices: $\lambda = 1$ or $\lambda = 2$ for logsig, $\lambda = 1$ for tanh

Continuous Activation Functions

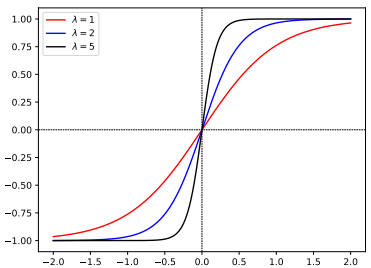
Sigmoidal

- $f(\xi) = \frac{1}{1+e^{-\lambda\xi}}$ ... logistic sigmoid (logsig)



Hyperbolic Tangent

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



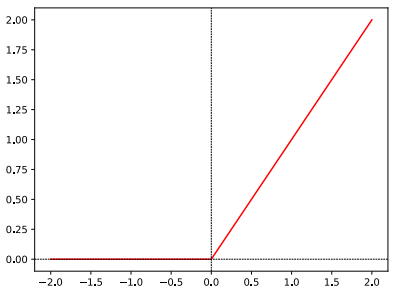
Usage

- Suitable for classification tasks (smoothed threshold function)
- Used in hidden layers of feedforward and deep neural networks, as well as in recurrent networks

Continuous Activation Functions

Rectified Linear Unit (ReLU)

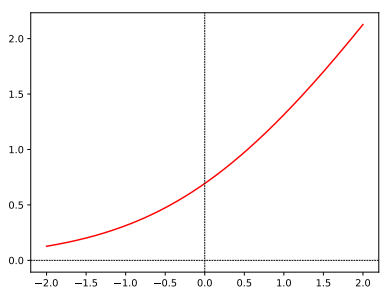
- $f(\xi) = \max(0, \xi) = \begin{cases} \xi, & \text{for } \xi > 0 \\ 0, & \text{for } \xi \leq 0 \end{cases}$



  - Not differentiable everywhere, but computationally efficient
  - Widely used in hidden layers of deep neural networks

Softplus (Smooth Alternative to ReLU)

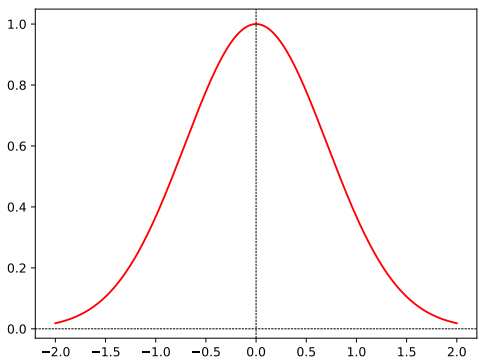
- $f(\xi) = \ln(1 + e^{\xi})$
- Behaves like ReLU for large ξ
- Behaves like a sigmoid function for small ξ
- Less commonly used than ReLU



Local Activation Functions

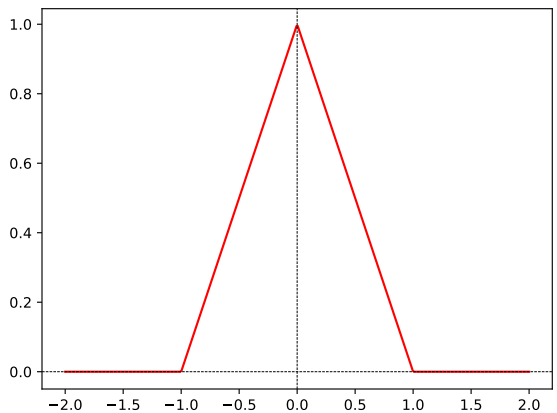
Radial Basis Function (Gaussian)

- $f(\xi) = e^{-\frac{\xi^2}{\alpha}}$ ... *radbas*
- $\xi = \frac{|\vec{x}-\vec{w}|}{\beta}$



Triangular Function

- $f(\xi) = \begin{cases} 1 - |\xi| & \text{for } |\xi| \leq 1 \\ 0 & \text{otherwise} \end{cases}$
  ... *tribas*
- $\xi = \frac{|\vec{x}-\vec{w}|}{\beta}$



$\rightarrow$ Networks with local units, **Radial Basis Function (RBF) networks**

Other Activation Functions

Stochastic Neuron Model: Stochastic Activation Function

- $f(\xi) = 1$ with probability $P(\xi)$
- $f(\xi) = 0$ with probability $1 - P(\xi)$

The probability function $P(\xi)$ is most commonly a sigmoid function:

- $P(\xi) = \frac{1}{1+e^{-\frac{\xi}{T}}}$
- T ... pseudo-temperature

$\rightarrow$ **Boltzmann Machines, Deep Belief Networks (DBN)**

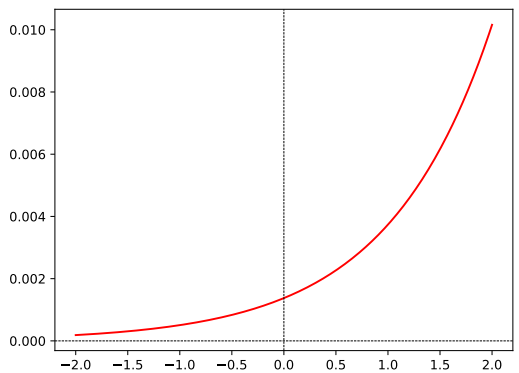
- Simplified explanation: Compute the sigmoid function value, interpret it as a probability, and use it to randomly determine the neuron's output (by tossing a coin)

Other Activation Functions

Softmax

- A specialized activation function for multi-class classification
- A generalization of the **argmax** function, converting numerical values into probabilities
- $f : R^n \rightarrow R^n$,

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$$



$\rightarrow$ Suitable only for the output layer in classification tasks

Gradient Descent for Training Neurons with a Continuous Activation Function

The gradient descent method can be used to train neurons with any **continuous** and **differentiable** activation function f (e.g., sigmoid function, hyperbolic tangent):

- Neuron output (for a single sample): $y = f(\xi) = f\left(\sum_{i=0}^{n} w_i x_i\right) = f(\vec{x}\vec{w})$
- Matrix form (for the entire training set): $\vec{y} = f(X\vec{w})$ ($\vec{w}$ is a column vector)

1. We aim to minimize the sum of squared errors (SSE) in weight space:

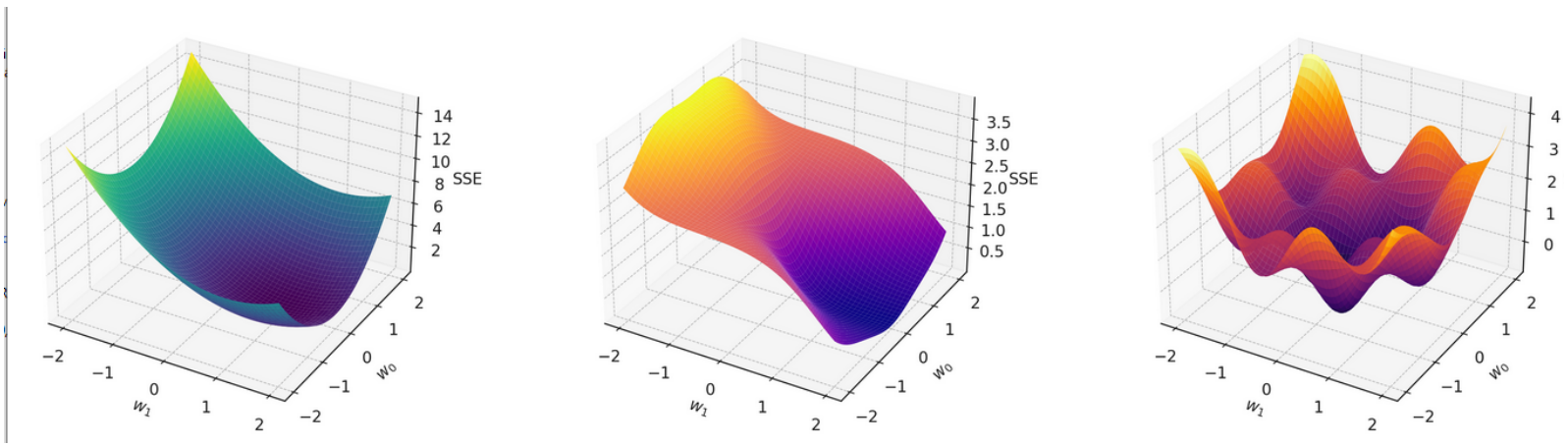
$$E(\vec{w}) = \frac{1}{2}\sum_{p=1}^{N}(d_p - y_p)^2 = \frac{1}{2}\sum_{p=1}^{N}\left(d_p - f\left(\sum_{i=0}^{n} w_i x_{pi}\right)\right)^2$$

$$= \sum^{N} E_p(\vec{w}) \ldots E_p(\vec{w}) \text{ is the error function for a single sample}$$

Gradient Descent for Training Neurons with a Continuous Activation Function

- We minimize the sum of squared errors (SSE) in weight space:

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



- Since the error function may no longer be quadratic, gradient descent may fail to find the global minimum, even with optimal hyperparameter settings.

Gradient Descent for Neurons with a Continuous Activation Function

1. First, compute the error function

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

Compute its partial derivatives:

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

Gradient Descent for Neurons with a Continuous Activation Function

1. Compute the partial derivatives of the error function:

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

2. Formulate the weight update rule:

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

For the weight vector:

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

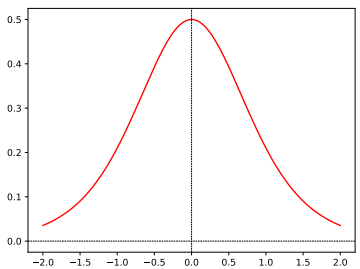
Gradient Descent for Neurons with a Continuous Activation Function

Computing the Derivative of the Activation Function

Sigmoidal Function ... logsig

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

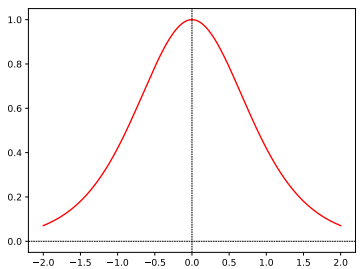
$$f'(\xi_p) = \lambda y_p (1 - y_p)$$



Hyperbolic Tangent ... tanh

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

$$f'(\xi_p) = \lambda^2 (1 + y_p)(1 - y_p)$$



Gradient Descent for Neurons with a Continuous Activation Function

General Algorithm Schema (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: α_0 such that $1 \gg \alpha_0 > 0$.
2. Present the next training sample $(\vec{x}_t, d_t)$ and compute the potential and actual neuron output:
   $\xi_t = \vec{x}_t \vec{w}$
   $y_t = f(\xi_t)$
3. Update the weights:

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

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

Discussion on the Gradient Descent Method

Different Learning Strategies

- **Pattern Presentation Strategies:**
  1. **Iterative sample presentation (Online GD)**
     - Fast learning (in terms of the number of epochs) but relatively unstable (the algorithm reduces error for the current sample $\rightarrow$ error may increase for other samples)
  2. **Batch sample presentation (Batch GD)**
     - More stable learning but often leads to "worse" solutions
     - Efficient for small datasets, but requires more memory for large datasets
  3. **Mini-batch sample presentation (SGD, Stochastic GD)**
     - A compromise solution, widely used in deep networks
- **Learning Rate Strategies:**
  - **Constant learning rate** - must be set appropriately
  - **Adaptive learning rate** - e.g., decreasing over time; the decay rate must be properly tuned

Discussion on the Gradient Descent Method

Different Learning Strategies

- **Weight Initialization:**
  - Weights are initialized with small random values (ideally centered around 0)
  - Too large or biased weights make perceptron training difficult
- **Stopping Criteria:**
  1. A predefined maximum number of epochs
  2. When the average error is sufficiently small ... $E < E_{min}$
  3. When the validation set error stops decreasing ... **early stopping**
  4. When the weight update Δw is too small ... $|\Delta w| < \delta_{min}$
- It is also possible to use **alternative loss functions** (e.g., cross-entropy for classification) or regularization terms

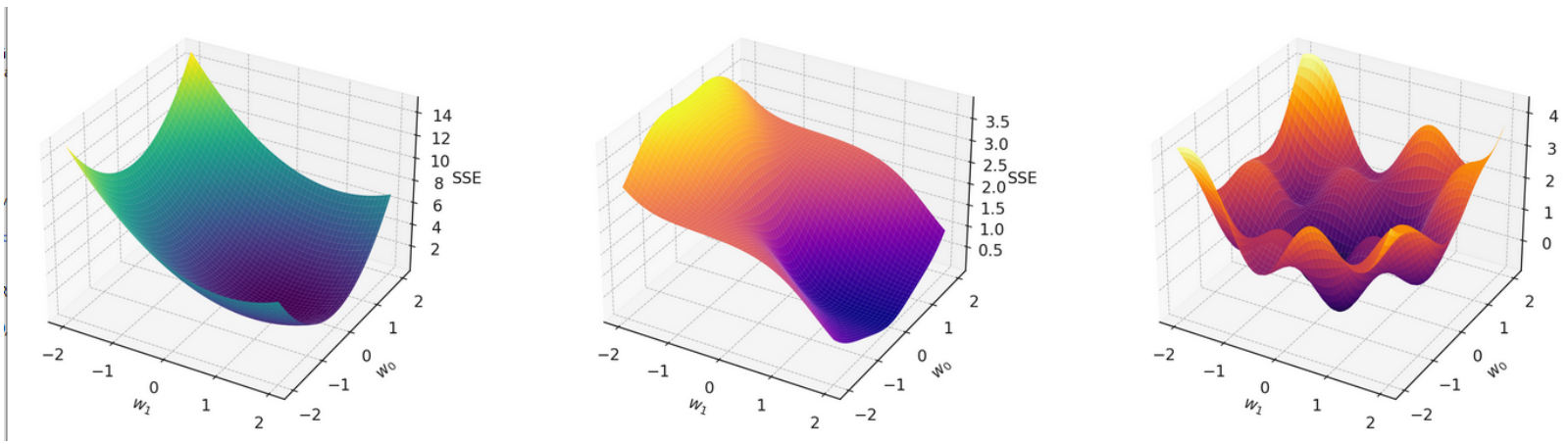
Discussion on the Gradient Descent Method

Importance of Preprocessing Training Data:

- For efficient and fast learning using gradient descent, both initial weights and input samples should have values in a small range, e.g., within $[-1, 1]$ or $[0, 1]$.
- Normalizing input data helps prevent issues with different input feature scales and leads to faster convergence.
- Normalization methods:
  - Min-max normalization to $[-1, 1]$ - suitable for uniformly distributed data without outliers
  - Standard deviation-based normalization - useful when data contains extreme values and outliers

Discussion on the Gradient Descent Method

- Unlike a linear neuron, the situation is more complex because the loss function may not be quadratic.
- Gradient descent is more prone to getting stuck in local minima.
- For general neurons, gradient descent is even more sensitive to hyperparameter selection and data preprocessing (normalization) compared to linear neurons.



Discussion on the Gradient Descent Method

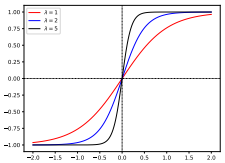
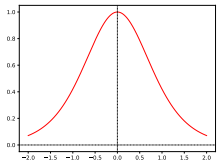
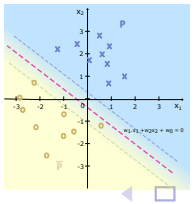
Risk of Neuron Saturation During Training:

- When weights become "too large," compared to the weight updates, the neuron "stops learning"

What affects the magnitude of the weight update?

$$\triangle \vec{w}(t) + \alpha_t f'(\xi_t)(d_t - y_t)\vec{x}_t^T$$

- $(d_t - y_t)$ ... difference between actual and desired output
- α ... learning rate
- $\vec{x}_t^T$ ... input sample $\rightarrow$ importance of normalization
- $f'(\xi_t)$ ... for sigmoid/hyperbolic tangent, the derivative decreases as the sample moves further from the decision boundary

Discussion on the Gradient Descent Method

How to Avoid Neuron Saturation?

- Normalize input data "around zero" and initialize weights "around zero"... both lead to an expected zero potential and fast initial learning.
- Handle outliers properly.
- For sigmoid/hyperbolic tangent, consider using a loss function other than SSE, such as **cross-entropy**:

$$E = -\sum_p \left(d_p \log y_p + (1 - d_p) \log(1 - y_p)\right) \quad \text{(for sigmoid)}$$

$$E = -\sum_p \left(\frac{1 + d_p}{2} \log \frac{1 + y_p}{2} + \frac{1 - d_p}{2} \log \frac{1 - y_p}{2}\right) \quad \text{(for tanh)}$$

Interpretation: How much does the predicted class probability (e.g., 0.9 vs. 0.1) differ from the ideal probability (1 vs. 0)?

Discussion on the Gradient Descent Method

Cross-Entropy Loss Function

- Highly beneficial for linear classification when combined with sigmoid or tanh activation functions.
- For sigmoid:

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

- The gradient is:

$$\frac{\partial E_p}{\partial y_p} = -\frac{d_p}{y_p} + \frac{(1 - d_p)}{(1 - y_p)}$$

- Substituting into the weight update formula eliminates problematic terms:

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

$\rightarrow$ This removes problematic terms, reducing the risk of saturation.

Examples - 1. Gradient Descent for a Perceptron with Hyperbolic Tangent Activation Function

tanh_perceptron.ipynb

- Simple examples: Gallbladder (Example 1), Pub (Example 2)
  - Observation: The gradient-based method successfully handles these tasks, although training takes longer and requires careful tuning of hyperparameters.
  - Note that the neuron's outputs are not exactly 1 and -1.
- Linear regression task (Example 3)
  - Observation: The tanh activation function is unsuitable for linear regression tasks.
- Randomly generated overlapping clusters (Example 4)
  - Observation: The gradient descent method again handles the task well; the same observations as in Examples 1 and 2 apply.
  - + Demonstration that poorly initialized weights pose a significant problem for gradient descent.
  - + Demonstration that with limited data, overfitting may occur, emphasizing the need for early stopping.

Examples - Gradient Descent for a Perceptron with Hyperbolic Tangent Activation Function

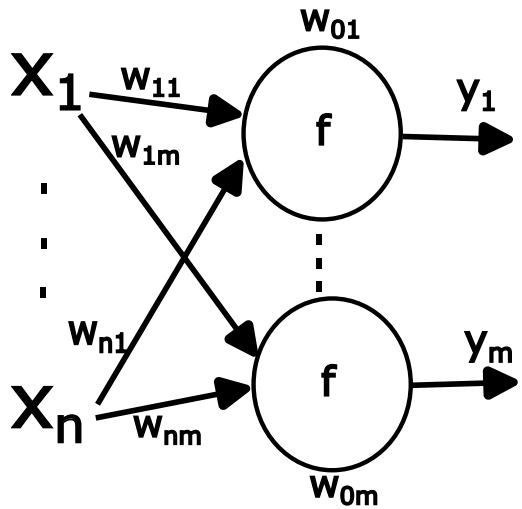
tanh_perceptron.ipynb

- Unnormalized data (Example 5)
  - Demonstration that unnormalized or biased data can significantly hinder the performance of gradient descent.
- Outlier data (Example 6)
  - Demonstration that outliers do not necessarily cause problems for iterative gradient descent when using the hyperbolic tangent activation function.
  - However, we will see that outliers can negatively affect data normalization based on min-max scaling.

Today's Lecture

1. Perceptron with a Continuous Activation Function
   - The most common activation functions for a perceptron
   - Training a general perceptron using gradient descent
2. Neural network with a single layer of neurons
   - Feature engineering

Neural Network with a Single Layer of Neurons



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

Interlude: Feature Engineering

Objective:

- Prepare input data so that the neural network model can learn from it and **learn effectively**.

What does feature engineering include?

- **Vectorization:** Perceptrons and perceptron networks work with numerical vectors $\rightarrow$ input samples must be converted into numerical vectors.
- **Normalization:** Numerical feature values should be normalized (ideally to the range $[-1, 1]$).
- Handling missing or incorrect values, noise, and outliers.
- Selecting relevant features (feature selection) and creating new features.

Interlude: Feature Engineering

- **Objective:** Prepare input data so that the neural network model can learn from it and **learn effectively**.
  - **Improved convergence:** Faster and more stable learning process.
  - **Increased predictive power:** The model better recognizes hidden patterns in the data.
  - **Noise reduction:** Eliminating irrelevant or redundant features.

Question about vectorization:

- How to handle **categorical** feature values?

Feature Engineering

Motivating Example: Multi-Class Classification with Categorical Features

Size	Fur	Speaks?	Movement	Class
Small	Short	No	Runs	Cat
Large	Long	No	Runs	Dog
Small	None	Yes	Flies	Parrot
Medium	Short	No	Runs	Cat
Small	None	No	Swims	Carp
...			...	

How to handle this?

1. First, convert categorical feature values to numerical values:
   - Ordinal encoding, one-hot encoding, embeddings, etc.
2. Then, optionally normalize the values.

Feature Engineering - Converting Categorical Variables to Numerical Values

Ordinal (Label) Encoding

- Suitable for categories with a natural order, e.g., *low, medium, high*, and for binary categories, e.g., *left, right*.
  - $Low = 0, \quad Medium = 1, \quad High = 2$
  - $Left = 0, \quad Right = 1$
- Values can be further normalized to the range $[-1, 1]$:
  - $Low = \text{-}1, \quad Medium = 0, \quad High = 1$
  - $Left = \text{-}1, \quad Right = 1$
- **Not suitable** for independent categories $\rightarrow$ such encoding can be misleading for the model:
  - $Cat = 0, \quad Dog = 1, \quad Parrot = 2$
  - Is a dog something between a cat and a parrot?

Feature Engineering - Converting Categorical Variables to Numerical Values

One-hot Encoding

- Suitable for independent categories, e.g., car color, animal type
  - $Red \rightarrow [1, 0, 0]$, $Blue \rightarrow [0, 1, 0]$, $Green \rightarrow [0, 0, 1]$
  - $Dog \rightarrow [1, 0, 0, 0]$, $Parrot \rightarrow [0, 1, 0, 0]$, $Cat \rightarrow [0, 0, 1, 0]$ $Carp \rightarrow [0, 0, 0, 1]$
- Optionally, we can normalize the values:
  - $Red \rightarrow [1, -1, -1]$, $Blue \rightarrow [-1, 1, -1]$, $Green \rightarrow [-1, -1, 1]$

Embeddings

- Used for more complex relationships between categorical values, e.g., representing words and sentences in natural language processing

Feature Engineering

Motivating Example: Multi-Class Classification with Categorical Features

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

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

Example: categorical_values.ipynb

Feature Engineering

Example: Multi-Class Classification with Categorical Features

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

How do we train the model?

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

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

Feature Engineering

Example: Multi-Class Classification with Categorical Features

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

How do we train the model?

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