Lecture 4

- ❖ **Multi-layer Perceptrons**
  - ✓ **Linear Regression**
  - ✓ **Multilinear Regression**
  - ✓ **Logistic Regression**
  - ✓ **Gradient Descent**
  - ✓ **Overview of Backpropogation Process**

CENG 632- Computational Intelligence, 2024-2025, Spring

Assist. Prof. Dr. Osman GÖKALP

Adapted from lecture notes of Christian Borgelt (https://computational-intelligence.eu/Lecture+Material.html)

Summary of Key Takeaways from Last Week

- Activation functions
  - **Hidden** neurons: **sigmoid**
  - **Output** neurons: **sigmoid** or **linear**
- **Unipolar** sigmoid functions' range: 0 to 1

  Bipolar sigmoid functions' range: -1 to +1 (e.g., hyperbolic tangent)
- **ReLU** is a popular non-linear activation function in deep learning.
- **Leaky ReLU** prevents **dying ReLU** problem.
- If **all** the activations functions are **linear**, any MLP can be **reduced** to a **two-layer perceptron**. So it cannot solve non-linear problems.

Summary of Key Takeaways from Last Week

- Any **Riemann-integrable** function can be approximated with arbitrary accuracy by a **four-layer perceptron** using **step** functions. (using relative step heights **one layer can be saved**).
- **Universal approximation theorem**: Any **continuous** function can be approximated by **3-layer perceptron** that has **continuous, bounded and non-constant activation functions**.

Mathematical Background: Regression

Mathematical Background: Linear Regression

Training neural networks is closely related to regression.

Given:
- A dataset $((x_1, y_1), \ldots, (x_n, y_n))$ of n data tuples and
- a hypothesis about the functional relationship, e.g. $y = g(x) = a + bx$.

Approach: Minimize the sum of squared errors, that is,

$$F(a, b) = \sum_{i=1}^{n} (g(x_i) - y_i)^2 = \sum_{i=1}^{n} (a + bx_i - y_i)^2.$$

F is a function that calculates error (squared) given a and b.

Necessary conditions for a minimum
(a.k.a. Fermat's theorem, after Pierre de Fermat, 1601–1665):

$$\frac{\partial F}{\partial a} = \sum_{i=1}^{n} 2(a + bx_i - y_i) = 0 \quad \text{and}$$

$$\frac{\partial F}{\partial b} = \sum_{i=1}^{n} 2(a + bx_i - y_i)x_i = 0$$

Partial derivative of the error function F w.r.t. a and b.

Setting derivatives to zero minimizes the sum of squared errors function because it is convex.

Mathematical Background: Linear Regression

Result of necessary conditions: System of so-called **normal equations**, that is,

$$na + \left(\sum_{i=1}^{n} x_i\right) b = \sum_{i=1}^{n} y_i,$$

$$\left(\sum_{i=1}^{n} x_i\right) a + \left(\sum_{i=1}^{n} x_i^2\right) b = \sum_{i=1}^{n} x_i y_i.$$

- Two linear equations for two unknowns a and b.
- System can be solved with standard methods from linear algebra.
- Solution is unique unless all x-values are identical.
- The resulting line is called a **regression line**.

Linear Regression: Example

Given the data (x,y), calculate the regression line (best fit line).

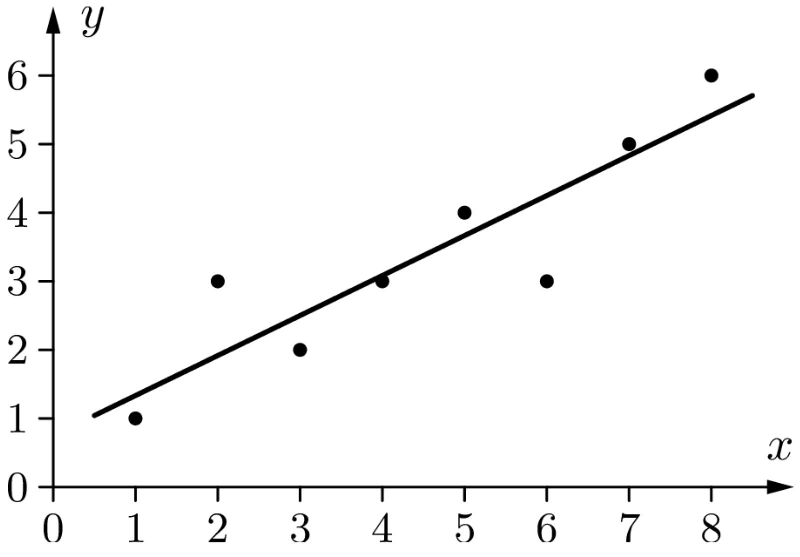
1 input (x), n=8 samples.

x	1	2	3	4	5	6	7	8
y	1	3	2	3	4	3	5	6

$$8a + 36b = 27$$
$$36a + 204b = 146$$

$$a = 3/4$$
$$b = 7/12$$

$$y = \frac{3}{4} + \frac{7}{12}x.$$



Instead of the line, now we calculate the best fit polynomial (regression polynomial).

Generalization to polynomials

$$y = p(x) = a_0 + a_1 x + \ldots + a_m x^m$$

Approach: Minimize the sum of squared errors, that is,

$$F(a_0, a_1, \ldots, a_m) = \sum_{i=1}^{n} (p(x_i) - y_i)^2 = \sum_{i=1}^{n} (a_0 + a_1 x_i + \ldots + a_m x_i^m - y_i)^2$$

Necessary conditions for a minimum: All partial derivatives vanish, that is,

$$\frac{\partial F}{\partial a_0} = 0, \quad \frac{\partial F}{\partial a_1} = 0, \quad \ldots \quad , \frac{\partial F}{\partial a_m} = 0.$$

Mathematical Background: Polynomial Regression

System of normal equations for polynomials

$$
\begin{array}{rcrcrcl}
n a_0 & + & \left(\sum_{i=1}^{n} x_i\right) a_1 & + \ldots + & \left(\sum_{i=1}^{n} x_i^m\right) a_m & = & \sum_{i=1}^{n} y_i \\
\left(\sum_{i=1}^{n} x_i\right) a_0 & + & \left(\sum_{i=1}^{n} x_i^2\right) a_1 & + \ldots + & \left(\sum_{i=1}^{n} x_i^{m+1}\right) a_m & = & \sum_{i=1}^{n} x_i y_i \\
\vdots & & & & & & \vdots \\
\left(\sum_{i=1}^{n} x_i^m\right) a_0 & + & \left(\sum_{i=1}^{n} x_i^{m+1}\right) a_1 & + \ldots + & \left(\sum_{i=1}^{n} x_i^{2m}\right) a_m & = & \sum_{i=1}^{n} x_i^m y_i,
\end{array}
$$

- $m+1$ linear equations for $m+1$ unknowns $a_0, \ldots, a_m$.
- System can be solved with standard methods from linear algebra.
- Solution is unique unless the points lie exactly on a polynomial of lower degree.

Mathematical Background: Multilinear Regression

Multilinear regression: finding a best fitting linear function with multiple arguments.

Generalization to more than one argument

$$z = f(x, y) = a + bx + cy$$

2-argument (input) example.

Approach: Minimize the sum of squared errors, that is,

$$F(a, b, c) = \sum_{i=1}^{n} (f(x_i, y_i) - z_i)^2 = \sum_{i=1}^{n} (a + bx_i + cy_i - z_i)^2$$

Necessary conditions for a minimum: All partial derivatives vanish, that is,

$$\begin{aligned}
\frac{\partial F}{\partial a} &= \sum_{i=1}^{n} 2(a + bx_i + cy_i - z_i) &= 0,\\
\frac{\partial F}{\partial b} &= \sum_{i=1}^{n} 2(a + bx_i + cy_i - z_i)x_i &= 0,\\
\frac{\partial F}{\partial c} &= \sum_{i=1}^{n} 2(a + bx_i + cy_i - z_i)y_i &= 0.
\end{aligned}$$

Mathematical Background: Multilinear Regression

System of normal equations for several arguments

$$
\begin{aligned}
na \;+\; \left(\sum_{i=1}^n x_i\right) b \;+\; \left(\sum_{i=1}^n y_i\right) c &= \sum_{i=1}^n z_i \\
\left(\sum_{i=1}^n x_i\right) a \;+\; \left(\sum_{i=1}^n x_i^2\right) b \;+\; \left(\sum_{i=1}^n x_i y_i\right) c &= \sum_{i=1}^n z_i x_i \\
\left(\sum_{i=1}^n y_i\right) a \;+\; \left(\sum_{i=1}^n x_i y_i\right) b \;+\; \left(\sum_{i=1}^n y_i^2\right) c &= \sum_{i=1}^n z_i y_i
\end{aligned}
$$

- 3 linear equations for 3 unknowns a, b, and c.
- System can be solved with standard methods from linear algebra.
- Solution is unique unless all data points lie on a straight line.

Multilinear Regression

General multilinear case:

General case: m-argument (input) instead of 2.

$$y = f(x_1, \ldots, x_m) = a_0 + \sum_{k=1}^{m} a_k x_k$$

Approach: Minimize the sum of squared errors, that is,

Error function: (Xa-y) is multiplied by itself to find square error. Note that the transpose is needed here.

$$F(\vec{a}) = (\mathbf{X}\vec{a} - \vec{y})^\top (\mathbf{X}\vec{a} - \vec{y}),$$

where

Parameters vector that needs be calculated by multilinear regression.

Input data matrix:
m inputs
n samples

$$\mathbf{X} = \begin{pmatrix} 1 & x_{11} & \ldots & x_{m1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{1n} & \ldots & x_{mn} \end{pmatrix}, \qquad \vec{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}, \qquad \text{and} \qquad \vec{a} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{pmatrix}$$

Output data vector of n samples

Necessary conditions for a minimum:

$$\vec{\nabla}_{\vec{a}}\, F(\vec{a}) = \vec{\nabla}_{\vec{a}}\, (\mathbf{X}\vec{a} - \vec{y})^\top (\mathbf{X}\vec{a} - \vec{y}) = \vec{0}$$

Multilinear Regression

- $\vec{\nabla}_{\vec{a}}\, F(\vec{a})$ may easily be computed by remembering that the differential operator

$$\vec{\nabla}_{\vec{a}} = \left(\frac{\partial}{\partial a_0}, \ldots, \frac{\partial}{\partial a_m}\right)$$

  behaves formally like a vector that is "multiplied" to the sum of squared errors.

- Alternatively, one may write out the differentiation componentwise.

With the former method we obtain for the derivative:

$$\begin{aligned}
\vec{\nabla}_{\vec{a}} F(\vec{a}) &= \vec{\nabla}_{\vec{a}}\left((\mathbf{X}\vec{a} - \vec{y})^\top (\mathbf{X}\vec{a} - \vec{y})\right) \\
&= \left(\vec{\nabla}_{\vec{a}}\left(\mathbf{X}\vec{a} - \vec{y}\right)\right)^\top (\mathbf{X}\vec{a} - \vec{y}) + \left((\mathbf{X}\vec{a} - \vec{y})^\top \left(\vec{\nabla}_{\vec{a}}\left(\mathbf{X}\vec{a} - \vec{y}\right)\right)\right)^\top \\
&= \left(\vec{\nabla}_{\vec{a}}\left(\mathbf{X}\vec{a} - \vec{y}\right)\right)^\top (\mathbf{X}\vec{a} - \vec{y}) + \left(\vec{\nabla}_{\vec{a}}\left(\mathbf{X}\vec{a} - \vec{y}\right)\right)^\top (\mathbf{X}\vec{a} - \vec{y}) \\
&= 2\mathbf{X}^\top (\mathbf{X}\vec{a} - \vec{y}) \\
&= 2\mathbf{X}^\top \mathbf{X}\vec{a} - 2\mathbf{X}^\top \vec{y} \quad = \quad \vec{0}
\end{aligned}$$

Multilinear Regression

Necessary condition for a minimum therefore:

$$\begin{aligned}\vec{\nabla}_{\vec{a}} F(\vec{a}) &= \vec{\nabla}_{\vec{a}} (\mathbf{X}\vec{a} - \vec{y})^\top (\mathbf{X}\vec{a} - \vec{y}) \\ &= 2\mathbf{X}^\top \mathbf{X}\vec{a} - 2\mathbf{X}^\top \vec{y} \stackrel{!}{=} \vec{0}\end{aligned}$$

As a consequence we obtain the system of **normal equations**:

$$\mathbf{X}^\top \mathbf{X}\vec{a} = \mathbf{X}^\top \vec{y}$$

This system has a solution unless $\mathbf{X}^\top \mathbf{X}$ is singular. If it is regular, we have

$$\vec{a} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \vec{y}.$$

$(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top$ is called the (Moore-Penrose-)**Pseudoinverse** of the matrix $\mathbf{X}$.

With the matrix-vector representation of the regression problem
an extension to **multipolynomial regression** is straighforward:
Simply add the desired products of powers to the matrix $\mathbf{X}$.

Mathematical Background: Logistic Regression

Generalization to non-polynomial functions

$$\text{Simple example:} \qquad y = ax^b$$

Idea: Find transformation to linear/polynomial case.

$$\text{Transformation for the above example:} \qquad \ln y = \ln a + b \cdot \ln x.$$

Special case: **logistic function**

$$y = \frac{Y}{1 + e^{a+bx}} \qquad \Leftrightarrow \qquad \frac{1}{y} = \frac{1 + e^{a+bx}}{Y} \qquad \Leftrightarrow \qquad \frac{Y - y}{y} = e^{a+bx}.$$

Result: Apply so-called **Logit-Transformation**

$$\ln\left(\frac{Y - y}{y}\right) = a + bx.$$

Logistic Regression: Example

x	1	2	3	4	5
y	0.4	1.0	3.0	5.0	5.6

Transform the data with

$$z = \ln\left(\frac{Y - y}{y}\right), \qquad Y = 6.$$

The transformed data points are

x	1	2	3	4	5
z	2.64	1.61	0.00	-1.61	-2.64

The resulting regression line and therefore the desired function are

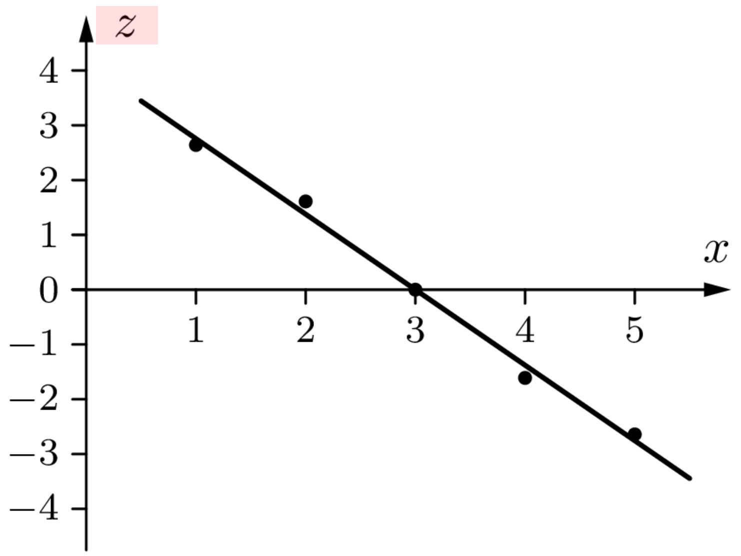
$$z \approx -1.3775x + 4.133 \qquad \text{and} \qquad y \approx \frac{6}{1 + e^{-1.3775x + 4.133}}.$$

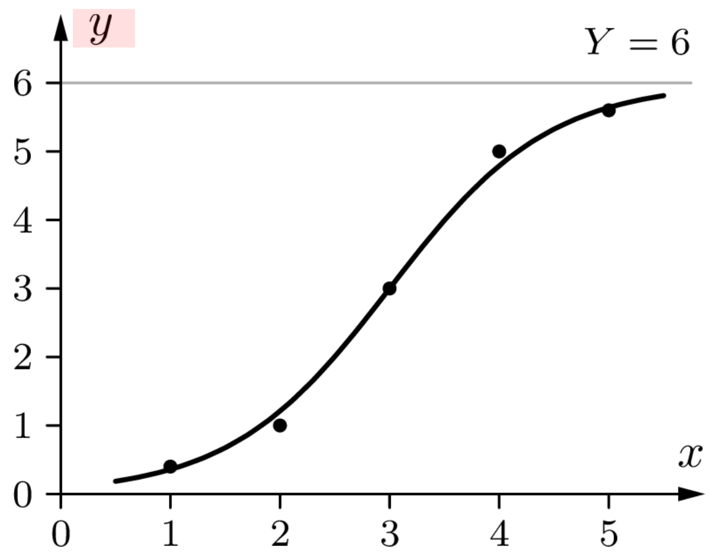
Attention: Note that the error is minimized only in the transformed space!
Therefore the function in the original space may not be optimal!

The function y is an approximation to this data.

Nevertheless, this approach usually yields very good results.

Logistic Regression: Example





The logistic regression function can be computed by a single neuron with

- network input function $f_{\text{net}}(x) \equiv wx$ with $w \approx -1.3775$,
- activation function $f_{\text{act}}(\text{net}, \theta) \equiv \left(1 + e^{-(\text{net} - \theta)}\right)^{-1}$ with $\theta \approx 4.133$ and
- output function $f_{\text{out}}(\text{act}) \equiv 6\,\text{act}$.

Limitations of Logistic Regression Approach

- Since the sum of the squared errors can be determined **only for output neurons**, this method is **limited to two-layer** perceptrons (no hidden layer).
- This limitation reminds that we cannot transfer the delta rule for the training threshold logic units.
- Therefore we consider next a **different method called gradient descent**.

Training Multi-layer Perceptrons

Training Multi-layer Perceptrons: Gradient Descent

- Problem of logistic regression: Works only for two-layer perceptrons.
- More general approach: **gradient descent**.
- Necessary condition: **differentiable activation and output functions**.

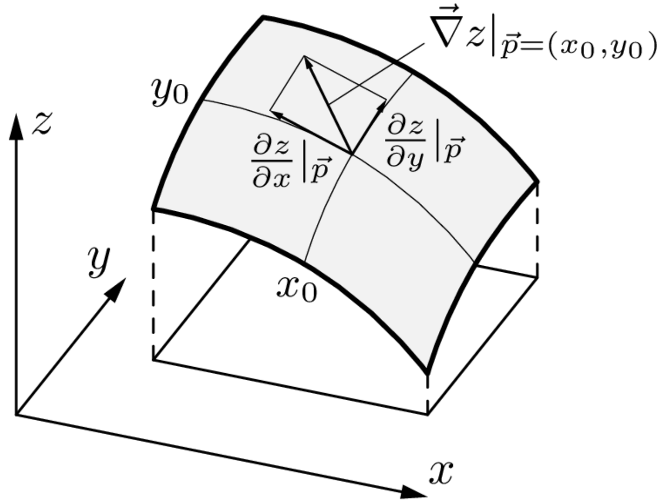


Illustration of the gradient of a real-valued function $z = f(x, y)$ at a point (x_0, y_0).

It is $\vec{\nabla} z|_{(x_0,y_0)} = \left(\frac{\partial z}{\partial x}|_{x_0}, \frac{\partial z}{\partial y}|_{y_0}\right)$. ($\vec{\nabla}$ is a differential operator called "nabla" or "del".)

Gradient Descent (Ascent) Algorithm

1. Choose a (random) starting point $\boldsymbol{u}^{(0)} = \left(u_1^{(0)}, \ldots, u_n^{(0)}\right)^\top$
2. Compute the gradient of the objective function f at the current point $\boldsymbol{u}^{(i)}$:

$$\nabla_{\boldsymbol{u}} f(\boldsymbol{u})\Big|_{\boldsymbol{u}^{(i)}} = \left(\frac{\partial}{\partial u_1} f(\boldsymbol{u})\Big|_{u_1^{(i)}}, \ldots, \frac{\partial}{\partial u_n} f(\boldsymbol{u})\Big|_{u_n^{(i)}}\right)^\top.$$

3. Make a small step in the direction (or against the direction) of the gradient:

$$\boldsymbol{u}^{(i+1)} = \boldsymbol{u}^{(i)} \pm \eta\, \nabla_{\boldsymbol{u}} f(\boldsymbol{u})\Big|_{\boldsymbol{u}^{(i)}}. \qquad \begin{array}{l} + : \text{gradient ascent} \\ - : \text{gradient descent} \end{array}$$

   η is a step width parameter ("learning rate" in artificial neuronal networks).
4. Repeat steps 2 and 3, until some termination criterion is satisfied (e.g., a certain number of steps has been executed, current gradient is small).

Gradient Descent Example

can be used to find the minimum of a polynomial function, here specifically

$$f(u) = \frac{5}{6}u^4 - 7u^3 + \frac{115}{6}u^2 - 18u + 6.$$

the derivative of the polynomial target function, that is,

$$f'(u) = \frac{10}{3}u^3 - 21u^2 + \frac{115}{3}u - 18,$$

The computations then proceed according to the scheme

$$u_{i+1} = u_i + \Delta u_i \qquad \text{with} \qquad \Delta u_i = -\eta f'(u_i),$$

Gradient Descent Example

i	u_i	$f(u_i)$	$f'(u_i)$	Δu_i
0	0.200	3.112	−11.147	0.111
1	0.311	2.050	−7.999	0.080
2	0.391	1.491	−6.015	0.060
3	0.451	1.171	−4.667	0.047
4	0.498	0.976	−3.704	0.037
5	0.535	0.852	−2.990	0.030
6	0.565	0.771	−2.444	0.024
7	0.589	0.716	−2.019	0.020
8	0.610	0.679	−1.681	0.017
9	0.626	0.653	−1.409	0.014
10	0.640	0.635		

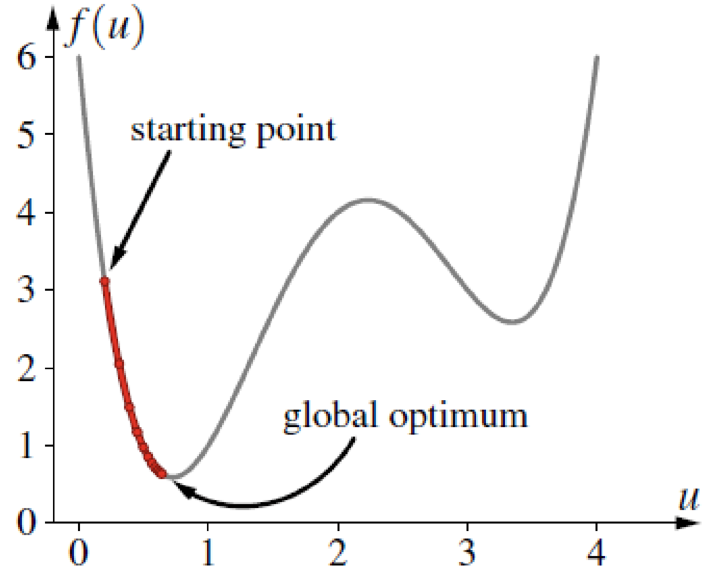


Fig. 5.21 Gradient descent with initial value 0.2 and step width parameter 0.01

i	u_i	$f(u_i)$	$f'(u_i)$	Δu_i
0	0.200	3.112	−11.147	0.011
1	0.211	2.990	−10.811	0.011
2	0.222	2.874	−10.490	0.010
3	0.232	2.766	−10.182	0.010
4	0.243	2.664	−9.888	0.010
5	0.253	2.568	−9.606	0.010
6	0.262	2.477	−9.335	0.009
7	0.271	2.391	−9.075	0.009
8	0.281	2.309	−8.825	0.009
9	0.289	2.233	−8.585	0.009
10	0.298	2.160		

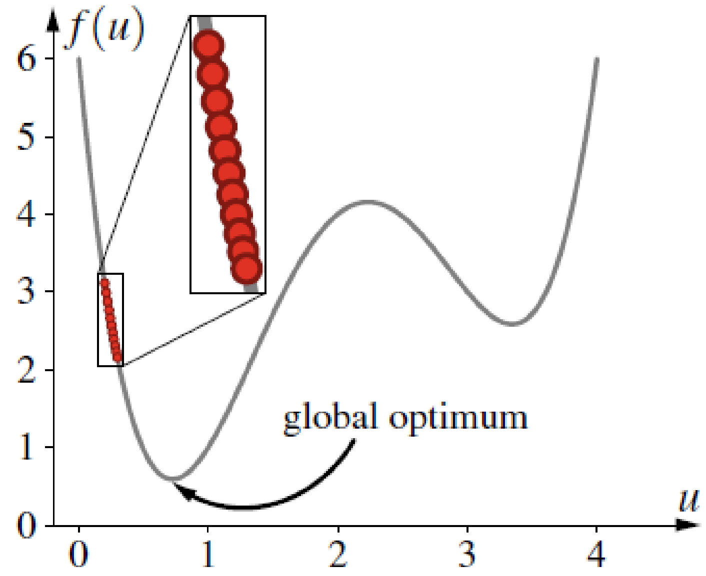


Fig. 5.22 Gradient descent with initial value 0.2 and step width parameter 0.001

Gradient Descent Example

i	u_i	$f(u_i)$	$f'(u_i)$	Δu_i
0	1.500	2.719	3.500	−0.875
1	0.625	0.655	−1.431	0.358
2	0.983	0.955	2.554	−0.639
3	0.344	1.801	−7.157	1.789
4	2.134	4.127	0.567	−0.142
5	1.992	3.989	1.380	−0.345
6	1.647	3.203	3.063	−0.766
7	0.881	0.734	1.753	−0.438
8	0.443	1.211	−4.851	1.213
9	1.656	3.231	3.029	−0.757
10	0.898	0.766		

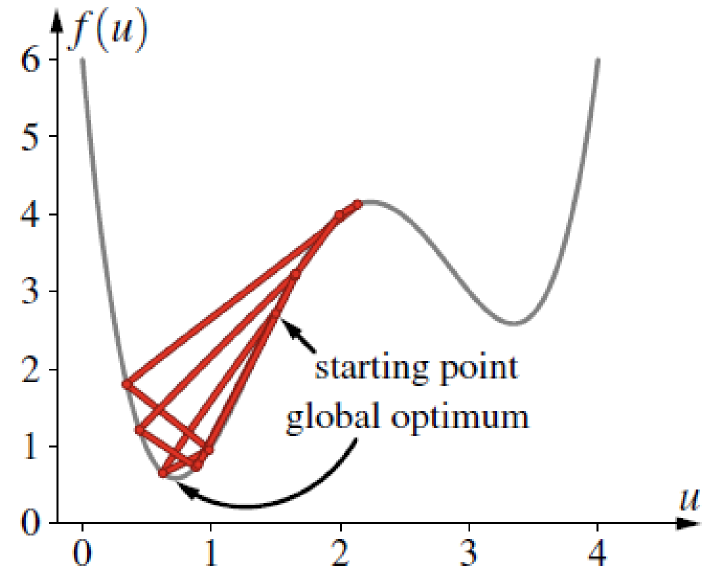


Fig. 5.23 Gradient descent with initial value 1.5 and step width parameter 0.25

i	u_i	$f(u_i)$	$f'(u_i)$	Δu_i
0	2.600	3.816	−1.707	0.085
1	2.685	3.660	−1.947	0.097
2	2.783	3.461	−2.116	0.106
3	2.888	3.233	−2.153	0.108
4	2.996	3.008	−2.009	0.100
5	3.097	2.820	−1.688	0.084
6	3.181	2.695	−1.263	0.063
7	3.244	2.628	−0.845	0.042
8	3.286	2.599	−0.515	0.026
9	3.312	2.589	−0.293	0.015
10	3.327	2.585		

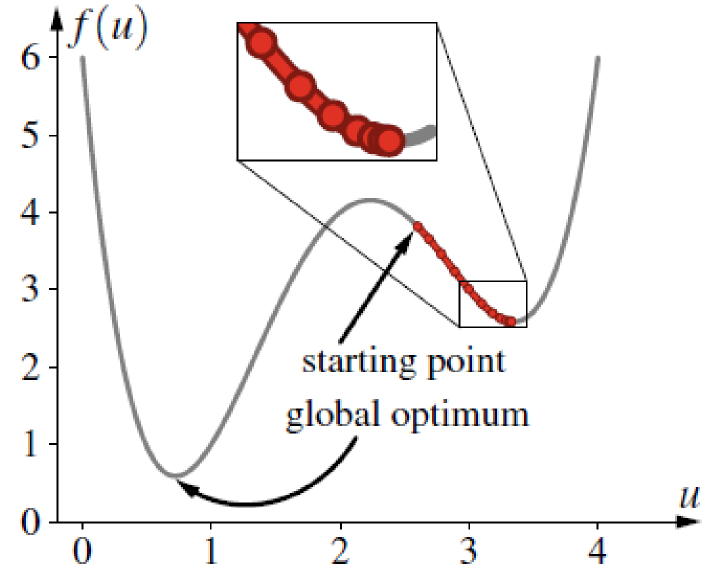


Fig. 5.24 Gradient descent with initial value 2.6 and step width parameter 0.05

Gradient Descent for the Logistic Function

1. Logistic Function

The logistic (sigmoid) function is:

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

where $z = w^T x + b$.

- w is the weight vector.
- x is the feature vector.
- b is the bias (intercept).
- $\sigma(z)$ is the predicted probability.

Gradient Descent for the Logistic Function

2. Sum of Squared Errors (SSE)

Instead of log-loss, we define the **error function** as the sum of squared errors:

$$F(w, b) = \frac{1}{2} \sum_{i=1}^{m} (\sigma(z_i) - y_i)^2$$

where:

- m is the number of training samples,
- y_i is the true label (0 or 1),
- $\sigma(z_i)$ is the predicted probability.

This is different from the standard **log-loss**, but we can still minimize it using gradient descent.

Gradient Descent for the Logistic Function

3. Partial Derivatives for Gradient Descent

To minimize $F(w, b)$, we compute the **gradients w.r.t.** w and b.

Derivative w.r.t. w**:**

Using the chain rule:

$$\frac{\partial F}{\partial w} = \sum_{i=1}^{m} (\sigma(z_i) - y_i) \cdot \frac{\partial \sigma(z_i)}{\partial z_i} \cdot \frac{\partial z_i}{\partial w}$$

- The derivative of the sigmoid function:

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

Gradient Descent for the Logistic Function

- Since $z_i = w^T x_i + b$, we get:

$$\frac{\partial z_i}{\partial w} = x_i$$

Thus, the gradient of the SSE function w.r.t. w is:

$$\frac{\partial F}{\partial w} = \sum_{i=1}^{m} (\sigma(z_i) - y_i) \cdot \sigma(z_i)(1 - \sigma(z_i)) \cdot x_i$$

Derivative w.r.t. b:

$$\frac{\partial F}{\partial b} = \sum_{i=1}^{m} (\sigma(z_i) - y_i) \cdot \sigma(z_i)(1 - \sigma(z_i))$$

Gradient Descent for the Logistic Function

4. Gradient Descent Update Rules

Using gradient descent:

$$w := w - \eta \sum_{i=1}^{m} (\sigma(z_i) - y_i) \cdot \sigma(z_i)(1 - \sigma(z_i)) \cdot x_i$$

$$b := b - \eta \sum_{i=1}^{m} (\sigma(z_i) - y_i) \cdot \sigma(z_i)(1 - \sigma(z_i))$$

where η is the **learning rate**.

Gradient Descent for the Logistic Function (Binary Classification)

2. Cost Function (Log-Loss)

The loss function for **logistic regression** is the **log-likelihood function** (negative log-likelihood):

$$J(w,b) = -\frac{1}{m}\sum_{i=1}^{m}\left[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right]$$

where:

- m = number of training samples,
- y_i = true label (0 or 1),
- $\hat{y}_i = \sigma(w^T x_i + b)$ = predicted probability.

Gradient Descent for the Logistic Function (Binary Classification)

3. Compute Gradients

To minimize the cost, compute the **partial derivatives**:

$$\frac{\partial F}{\partial w} = \frac{1}{m} \sum_{i=1}^{m} (\sigma(z_i) - y_i) x_i$$

$$\frac{\partial F}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (\sigma(z_i) - y_i)$$

Gradient Descent for the Logistic Function (Binary Classification)

4. Gradient Descent Update Rules

The parameters are updated using gradient descent:

$$w := w - \eta \frac{\partial F}{\partial w}$$

$$b := b - \eta \frac{\partial F}{\partial b}$$

where η is the learning rate.

Overview of the Backpropogation Process

1. **Forward Pass**:
    - The input data passes through the network (input → hidden layers → output).
    - Each neuron applies weights and biases to the inputs and calculates an output using an activation function (e.g., sigmoid).
2. **Calculate Error**:
    - Compare the network's output with the true target (the actual answer you want).
    - The **error** is the difference between the predicted output and the actual output.
3. **Backward Pass**:
    - The error is "backpropagated" through the network to adjust the weights.
    - This is done layer by layer, starting from the output layer back to the input layer.
4. **Use Chain Rule**:
    - To update weights, we calculate how much each weight contributed to the error. This is done using **partial derivatives** (from calculus).
    - The **chain rule** is used to calculate these derivatives for each layer's weights and biases.
5. **Update Weights**:
    - Adjust the weights to minimize the error. The update is done by subtracting a small step (called the learning rate) times the derivative (gradient) for each weight.