Linear Regression

Lecture Series „Machine Learning“

Niels Landwehr

Research Group „Data Science“
Institute of Computer Science
University of Hildesheim

Agenda

- Linear regression model and normal equations
- Gradient descent for parameter estimation

Agenda

- Linear regression model and normal equations
- Gradient descent for parameter estimation

Review: Supervised Learning

- Review: in **supervised learning**, the goal is to make predictions about objects

Input: an **instance** (object) $\mathbf{x}$ that lives in an instance space $\mathcal{X}$

$$\mathbf{x} \mapsto y$$

$$\mathbf{x} \in \mathcal{X} \qquad y \in \mathcal{Y}$$

Output: a **label** or **target** y that lives in a target space $\mathcal{Y}$

- To obtain predictions, we are looking for a **model** f that produces a prediction $f(\mathbf{x}) \in \mathcal{Y}$ for an input instance $\mathbf{x}$

$$f : \mathcal{X} \to \mathcal{Y}$$

Input: instance $\mathbf{x}$

$$\mathbf{x} \mapsto f(\mathbf{x})$$

Output: prediction $f(\mathbf{x})$

- Model will be inferred from **training data**: a set of instances with observed targets

$$\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$$

Training instances $\mathbf{x}_n \in \mathcal{X}$: observed objects in training data, for example flowers, images of digits, or emails

Observed labels or targets $y \in \mathcal{Y}$ in training data, for example classes of flowers, digits 0...9, or spam/legitimate classifications

Models as Parameterized Functions

- **How does the model f look like?**
- An important and frequently used class of models are **parameterized functions**:
  - The model, often written as $f_{\boldsymbol{\theta}}$, is a function parameterized by a vector of parameters $\boldsymbol{\theta} \in \mathbb{R}^D$
  - The model is determined by the structure of the function and its parameters $\boldsymbol{\theta}$
  - The structure of the function is chosen a priori, the parameters are chosen during learning based on the training data

- In this setting, there are basically two ingredients to a machine learning approach:
  - We need to choose the set of models under consideration. This is determined by the structure of the function: it is given by the set

    $$\mathcal{F} = \{f_{\boldsymbol{\theta}} \mid \boldsymbol{\theta} \in \mathbb{R}^D\}$$

    which is also called the model space
  - Given the model space, we need to determine how we choose a particular model $f_{\boldsymbol{\theta}^*} \in \mathcal{F}$ based on the training data. This is the actual learning algorithm.

Linear Regression

- A simple but important instance of parameterized functions in machine learning are so-called **linear models**, for regression (this lecture) and classification (next lecture)

- **Linear Regression:** a simple but widely used model for regression tasks
  - As discussed above, assume that instances are represented by vectors

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_M \end{pmatrix} \in \mathbb{R}^M$$

  - In regression tasks, the targets are continuous, that is, we assume $y \in \mathbb{R}$
  - The parameterized function is given by

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots \theta_M x_M$$

Function computes real-valued output based on instance $\mathbf{x} \in \mathbb{R}^M$

Function is **linear** in input: the output is obtained by multiplying each feature x_m in the instance $\mathbf{x}$ by a corresponding model parameter θ_m

The model parameter vector is given by

$$\boldsymbol{\theta} = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \dots \\ \theta_M \end{pmatrix} \in \mathbb{R}^{M+1}$$

Linear Regression

- To write the linear regression model more compactly, we can replace the constant term in the regression function by an additional constant element in the input vector:
  - Assume we are including an artificial constant attribute x_0 in the input vector $\mathbf{x}$:

We extend the old feature vector by an extra feature that is constantly one

Extra feature that is constantly one

$$\mathbf{x} := \begin{pmatrix} x_0 \\ x_1 \\ \dots \\ x_M \end{pmatrix} \in \mathbb{R}^{M+1} \qquad \text{where} \quad x_0 = 1$$

Old feature vector

  - Then, we can write the linear regression compactly as a dot product:

$$\begin{aligned} f_{\boldsymbol{\theta}}(\mathbf{x}) &= \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_M x_M \\ &= \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_M x_M \\ &= \mathbf{x}^{\mathrm{T}} \boldsymbol{\theta} \end{aligned}$$

- In this notation, the constant term θ_0 is a model parameter like any other θ_m, only that it happens to be in front of a constant feature

Learning by Optimizing a Loss Function

- For linear regression, the linear regression equation defines the model space: it contains all functions that have this form, that is,

$$\mathcal{F} = \{f_{\boldsymbol{\theta}} \mid f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{x}^{\mathrm{T}}\boldsymbol{\theta},\ \boldsymbol{\theta} \in \mathbb{R}^D\}$$

- It remains to specify how we choose the model parameters $\boldsymbol{\theta}$ (and thereby a specific model out of the model space) based on the training data $\mathcal{D} = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)\}$

- How do we choose model parameters $\boldsymbol{\theta}$ based on training data?
  1. Need to quantify how good a model parameter vector $\boldsymbol{\theta}$ is, based on training data: this is accomplished by defining a **loss function**
  2. Need to find model parameters that are good according to loss function: this results in an **optimization problem** that has to be solved by the machine learning algorithm

Loss Functions

- **Loss function**: measure how well a model $f_{\boldsymbol{\theta}}$ fits the training data
- Intuitively, the loss measures how close the predictions of the model on the training instances, $f_{\boldsymbol{\theta}}(\mathbf{x}_n)$, are to the observed targets, y_n
  - Low loss: predictions of the model are close to target and the model captures the underlying $(\mathbf{x}, y)$-relationship in the data well
  - High loss: predictions of the model are far away from target and model does not capture underlying $(\mathbf{x}, y)$-relationship in the data
  - The loss function is also sometimes called cost function or objective function

- Loss on a single training example $(\mathbf{x}_n, y_n)$ is defined by an instance-level loss function

$$\ell(f_{\boldsymbol{\theta}}(\mathbf{x}_n), y_n)$$

- Loss on entire data set is the average of the losses on all instances and defined by

Here, we make the dependence on the model parameters $\boldsymbol{\theta}$ explicit, because we want to optimize the loss in $\boldsymbol{\theta}$

$$L(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N} \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_n), y_n)$$

Loss Functions

- Example: squared and absolute loss for regression
  - Most common loss function for regression is the **squared loss**:

$$\ell(f_{\boldsymbol{\theta}}(\mathbf{x}_n), y_n) = (f_{\boldsymbol{\theta}}(\mathbf{x}_n) - y_n)^2$$

Squared difference between prediction $f_{\boldsymbol{\theta}}(\mathbf{x}_n)$ and target y_n. Penalizes outliers disproportionately.

  - Alternatively, the **absolute loss** can be used:

$$\ell(f_{\boldsymbol{\theta}}(\mathbf{x}_n), y_n) = | f_{\boldsymbol{\theta}}(\mathbf{x}_n) - y_n |$$

Absolute difference between prediction $f_{\boldsymbol{\theta}}(\mathbf{x}_n)$ and target y_n.

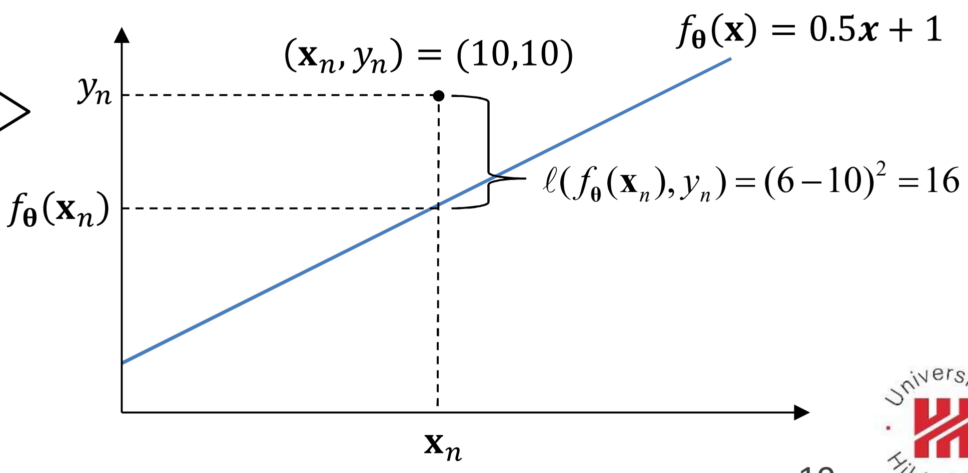
- Example: squared loss for linear regression $f_{\boldsymbol{\theta}}(\mathbf{x}) = 0.5\mathbf{x} + 1$ with $\mathbf{x} \in \mathbb{R}$

One-dimensional linear regression model $f_{\boldsymbol{\theta}}(\mathbf{x}) = 0.5\mathbf{x} + 1$ given by blue line in graph.

Loss is computed at $\mathbf{x}_n = 10$.

Function value at this point is $f_{\boldsymbol{\theta}}(\mathbf{x}_n) = 6$, target at this point is $y_n = 10$

Squared loss is $\ell(f_{\boldsymbol{\theta}}(\mathbf{x}_n), y_n) = (6-10)^2 = 16.$



Learning: Minimizing the Loss Function

- The loss function defines how well a model fits the training data, and thereby how well the model captures the underlying $(\mathbf{x}, y)$-relationship in the data
- Given the definition of a loss function, we can view the problem of learning the model $f_{\boldsymbol{\theta}}$ from data as an optimization problem: find the model parameters with the lowest loss on the training data

Learning as optimization:

$$\boxed{\begin{aligned}\boldsymbol{\theta}^* &= \arg\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) \\ &= \arg\min_{\boldsymbol{\theta}} \frac{1}{N}\sum_{n=1}^{N} \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_n), y_n)\end{aligned}}$$

- The model $f_{\boldsymbol{\theta}^*}$ with loss-minimizing parameters $\boldsymbol{\theta}^*$ is the learned model

- **Remark**: in addition to simply minimizing the loss on the training data, in practice often need to augment the optimization criterion with a regularization term to prevent overfitting. We will come back to this in a later lecture.

Training Data in Matrix Form

- How do we solve the optimization problem $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$?
  - Approximately or exactly?
  - Computational efficiency?
  - Numerical stability?

- To simplify notation, let us write the training data $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$ in matrix form:

First instance $\mathbf{x}_1 \in \mathbb{R}^M$ (as row vector)

All labels summarized as row vector $\mathbf{y} \in \mathbb{R}^N$

All instances are summarized in a matrix of size (number of instances N) x (number of features M)

$$\mathbf{X} = \begin{pmatrix} x_{1,1} & \ldots & x_{1,M} \\ \ldots & \ldots & \ldots \\ x_{N,1} & \ldots & x_{N,M} \end{pmatrix} \in \mathbb{R}^{N \times M} \qquad \mathbf{y} = \begin{pmatrix} y_1 \\ \ldots \\ y_N \end{pmatrix} \in \mathbb{R}^N$$

Last instance $\mathbf{x}_N \in \mathbb{R}^M$ (as row vector)

Training Data in Matrix Form

- Let $\hat{y}_n = f_{\boldsymbol{\theta}}(\mathbf{x}_n)$ denote the prediction of the model on the *n*-th instance
- The predictions of a linear model $f_{\boldsymbol{\theta}}(\mathbf{x}) = \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_M x_M$ on the entire training data can now be written as

$$\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta} \in \mathbb{R}^N$$

Result of multiplying *N* x *M* matrix $\mathbf{X}$ with *M*-dimensional vector $\boldsymbol{\theta}$ is *N*-dimensional vector $\hat{\mathbf{y}}$. By construction of the matrix $\mathbf{X}$, n-th element of result vector is $\theta_1 x_{n,1} + \theta_2 x_{n,2} + \ldots + \theta_M x_{n,M}$

- Here, we have left out the constant model term θ_0 because it can be incorporated into an additional constant feature as described above
- Define the L2-norm of a vector $\mathbf{z} \in \mathbb{R}^N$ as

$$\|\mathbf{z}\|_2 = \sqrt{z_1^2 + \ldots + z_N^2}$$

- The squared loss of the linear model $f_{\boldsymbol{\theta}}(\mathbf{x})$ can be computed as

$$L(\boldsymbol{\theta}) = \frac{1}{N}\|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2$$

Here, the squared L2-norm sums up the squared differences between the targets in $\mathbf{y}$ and the predictions in $\mathbf{X}\boldsymbol{\theta}$

Normal Equations for Linear Regression

- The model parameters minimizing the squared loss,

$$\begin{aligned}\boldsymbol{\theta}^* &= \arg\min_{\boldsymbol{\theta}} \frac{1}{N}\|\mathbf{y}-\mathbf{X}\boldsymbol{\theta}\|_2^2 \\ &= \arg\min_{\boldsymbol{\theta}} \|\mathbf{y}-\mathbf{X}\boldsymbol{\theta}\|_2^2\end{aligned}$$

constant 1/N does not affect minimum

  are also called the least squares estimates, because they minimize the squared errors
- The least squares estimates can be computed by solving for $\boldsymbol{\theta}$ the so-called normal equations:

$$\mathbf{X}^{\mathrm{T}}\mathbf{X}\boldsymbol{\theta} = \mathbf{X}^{\mathrm{T}}\mathbf{y}$$

- Solving the normal equations means solving a system of M linear equations: solve

$$\mathbf{A}\boldsymbol{\theta} = \mathbf{b}$$

  with $\mathbf{A} = \mathbf{X}^{\mathrm{T}}\mathbf{X} \in \mathbb{R}^{M\times M}$ and $\mathbf{b} = \mathbf{X}^{\mathrm{T}}\mathbf{y} \in \mathbb{R}^{M}$

- Different algorithmic methods available: e.g. Gaussian elimination, Cholesky decomposition, QR decomposition

Normal Equations for Linear Regression

- Why does solving the normal equations yield the least squares model parameters?
- For two vectors $\mathbf{a},\mathbf{b}\in\mathbb{R}^N$ let $\langle\mathbf{a},\mathbf{b}\rangle=\sum_{n=1}^{N}a_n b_n$ denote their dot product
- Note that $\|\mathbf{a}\|_2^2=\langle\mathbf{a},\mathbf{a}\rangle$, namely the sum of squared vector elements
- The squared loss of the model $f_{\boldsymbol\theta}(\mathbf{x}_n)$ can be written as

$$\frac{1}{N}\|\mathbf{y}-\mathbf{X}\boldsymbol\theta\|_2^2=\frac{1}{N}\langle\mathbf{y}-\mathbf{X}\boldsymbol\theta,\mathbf{y}-\mathbf{X}\boldsymbol\theta\rangle$$

Like the squared norm, the dot product also sums up the the squared vector elements

- To find the minimum, we can set the derivative to zero

Derived by chain rule: we have an expression of the form $\langle\mathbf{z},\mathbf{z}\rangle$, with $\mathbf{z}=\mathbf{y}-\mathbf{X}\boldsymbol\theta$. The outer derivative is $\frac{\partial}{\partial\mathbf{z}}\langle\mathbf{z},\mathbf{z}\rangle=2\mathbf{z}=2(\mathbf{y}-\mathbf{X}\boldsymbol\theta)$, similarly as in scalar calculus $\frac{\partial}{\partial x}x^2=2x$. The inner derivative is $\frac{\partial}{\partial\boldsymbol\theta}\mathbf{z}=-\mathbf{X}$. The two are multiplied resulting in $(-\mathbf{X})^{\mathrm{T}}2(\mathbf{y}-\mathbf{X}\boldsymbol\theta)$.

$$\frac{\partial}{\partial\boldsymbol\theta}\frac{1}{N}\langle\mathbf{y}-\mathbf{X}\boldsymbol\theta,\mathbf{y}-\mathbf{X}\boldsymbol\theta\rangle=\frac{1}{N}(-\mathbf{X})^{\mathrm{T}}2(\mathbf{y}-\mathbf{X}\boldsymbol\theta)=-\frac{2}{N}(\mathbf{X}^{\mathrm{T}}\mathbf{y}-\mathbf{X}^{\mathrm{T}}\mathbf{X}\boldsymbol\theta)\overset{!}{=}0$$

Resolving product

Set to zero to find minimum

$$\Rightarrow\mathbf{X}^{\mathrm{T}}\mathbf{y}=\mathbf{X}^{\mathrm{T}}\mathbf{X}\boldsymbol\theta$$

results in normal equations

Normal Equations for Linear Regression

- Learning a linear regression model using normal equations, formulated as algorithm:

```
Algorithm learn-linreg-NormEq
Input: training data $\mathcal{D} := \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$
Output: learned model parameters $\boldsymbol{\theta}$
1. $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_N)^{\mathrm{T}}$
2. $\mathbf{y} = (y_1, \ldots, y_N)^{\mathrm{T}}$
3. $\mathbf{A} = \mathbf{X}^{\mathrm{T}}\mathbf{X}$
4. $\mathbf{b} = \mathbf{X}^{\mathrm{T}}\mathbf{y}$
5. $\boldsymbol{\theta} = \text{solve-linear-equations}(\mathbf{A}, \mathbf{b})$
6. return $\boldsymbol{\theta}$
```

Summarize training instances in matrix $\mathbf{X}$ and training labels in vector $\mathbf{y}$

Compute expressions $\mathbf{A}$, $\mathbf{b}$ for normal equations

Solving system of equations (using one of the methods mentioned above) yields final model parameters

Computational Complexity

- What is the computational complexity of learning a linear regression model via the normal equations?
- Dimensions of input:

Matrix $\mathbf{X} \in \mathbb{R}^{N \times M}$ $(N =$ number of instances, $M =$ number of features$)$

Vector $\mathbf{y} \in \mathbb{R}^{N}$

- Main computational steps:

Matrix product $\mathbf{X}^{\mathrm{T}}\mathbf{X}$: $M \times N$ times $N \times M$ matrix multiplication, $\mathrm{O}(NM^2)$

Solve system of M linear equations: $O(M^3)$

- Overall runtime: $O(NM^2 + M^3)$

Linear dependence on N is ok,

but computation time rises quickly with M

M	runtime [s]
100	0.002
200	0.004
400	0.022
800	0.086
1600	0.555
3200	6.275
6400	27.902

(Intel i5-760 2.8 MHz, 2010, Python numpy)

Agenda

- Linear regression model and normal equations
- Gradient descent for parameter estimation

Motivation: Gradient Descent

- For the linear regression model, the parameter vector minimizing the squared loss can be obtained by simply solving a system of linear equations
- However, this is due to the particularly simple form of the model and the particular loss function (squared loss)
- In general, for most combinations of models and loss functions, finding the model parameters that minimize the loss function is less straightforward
- Even for linear regression, computational complexity might prevent us from using normal equations to solve for the parameters minimizing the loss

- **Gradient descent**: An alternative approach for finding model parameters that (approximately) minimize the loss
  - Iterative optimization algorithm
  - Finds global optimum if the problem of loss-minimization is convex in parameters
  - Otherwise, finds a local optimum (still good & useful in many practical setttings)
  - Very flexible for different models and losses, widely used in practice

Learning: How to Solve the Optimization Problem?

- Formally speaking, to find the loss-minimizing parameter vector $\boldsymbol{\theta}$ we need to solve an optimization problem
  - Given a model class $\mathcal{F} = \{f_{\boldsymbol{\theta}} \mid \boldsymbol{\theta} \in \mathbb{R}^D\}$
  - Given a loss function $\ell(f(\mathbf{x}), y)$
  - Given training data $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$
  - Find

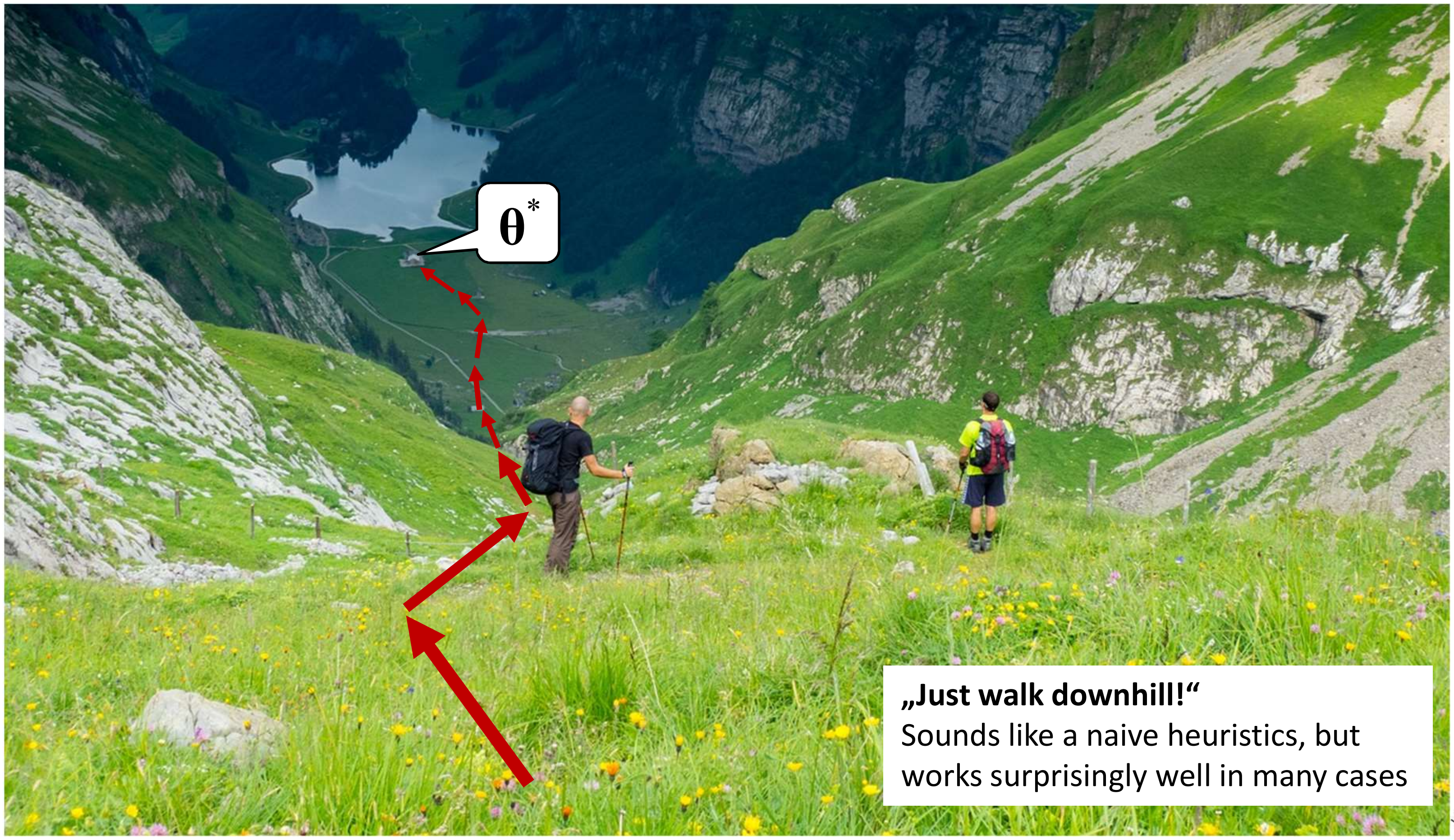
$$\boxed{\begin{aligned}\boldsymbol{\theta}^* &= \arg\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) \\ L(\boldsymbol{\theta}) &= \frac{1}{N}\sum_{n=1}^{N} \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_n), y_n)\end{aligned}}$$

- Challenges:
  - Optimization variable $\boldsymbol{\theta}$ can be high-dimensional (depending on model class)
  - Data set can be large (in terms of number of instances and/or features)
  - $L(\boldsymbol{\theta})$ can be a complex, generally non-convex function in $\boldsymbol{\theta}$

- **How to solve optimization problem?**

Optimization: Gradient Descent

- Gradient descent: find the model parameter vector $\boldsymbol{\theta}^*$ by „walking“ in the space $\mathbb{R}^D$ of model parameters always in the direction of steepest descent

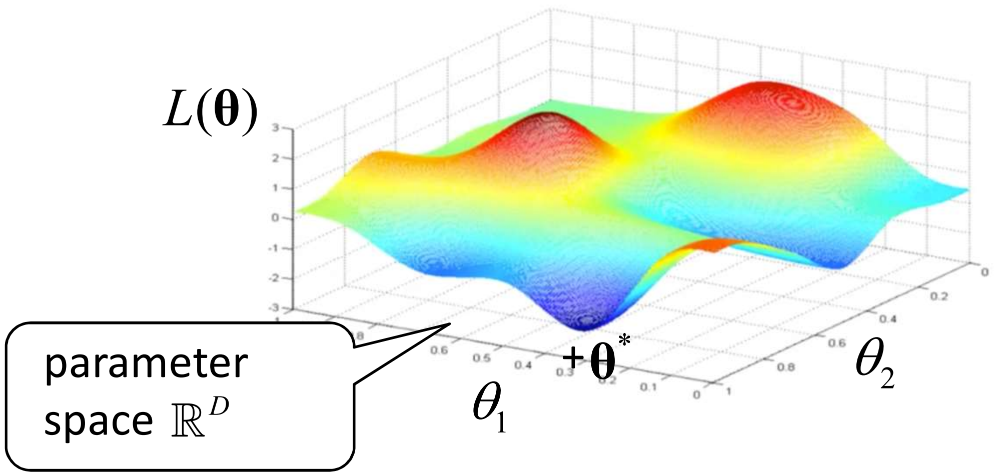


„Just walk downhill!“
Sounds like a naive heuristics, but works surprisingly well in many cases

Loss Function Surface

- More formally: the loss function defines a surface over the *D*-dimensional $\boldsymbol{\theta}$-space

- The loss function $L(\boldsymbol{\theta})$ maps parameter vectors to a real number: $L : \mathbb{R}^D \rightarrow \mathbb{R}$
- Can be visualized as a loss surface / landscape:



Want to find the „lowest point“:

$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$

- Note: parameter space is high-dimensional (often millions of parameters and therefore dimensions). 2D-visualizations can be misleading.
- Shape of loss surface will depend on loss function, structure of model, data

Gradient of Function

- Idea: follow the slope. How do we get the local slope of the loss function?
- **Gradient**:

Let $f(\mathbf{x})$ with $f:\mathbb{R}^D \to \mathbb{R}$ denote a scaler-valued differentiable function.
The gradient of f, written ∇f, is a function $\nabla f:\mathbb{R}^D \to \mathbb{R}^D$ defined by

$$\nabla f(\mathbf{x}) = \begin{pmatrix} \underbrace{\frac{\partial f}{\partial x_1}(\mathbf{x})}_{\text{partial derivative of } f \text{ with respect to } x_1} \\ \frac{\partial f}{\partial x_2}(\mathbf{x}) \\ \ldots \\ \underbrace{\frac{\partial f}{\partial x_D}(\mathbf{x})}_{\text{partial derivative of } f \text{ with respect to } x_D} \end{pmatrix}$$

Gradient of Function

- Idea: follow the slope. How do we get the local slope of the loss function?
- **Example for gradient (k=3)**:

$$f : \mathbb{R}^3 \to \mathbb{R} \quad \text{with} \quad f(\mathbf{x}) = 3x_1^2 x_2 + 2x_2^2 - x_1 x_3 \qquad \mathbf{x} = (x_1, x_2, x_3)^{\mathrm{T}}$$

Partial derivatives:

$$\frac{\partial f}{\partial x_1} = 6x_1 x_2 - x_3$$

$$\frac{\partial f}{\partial x_2} = 3x_1^2 + 4x_2$$

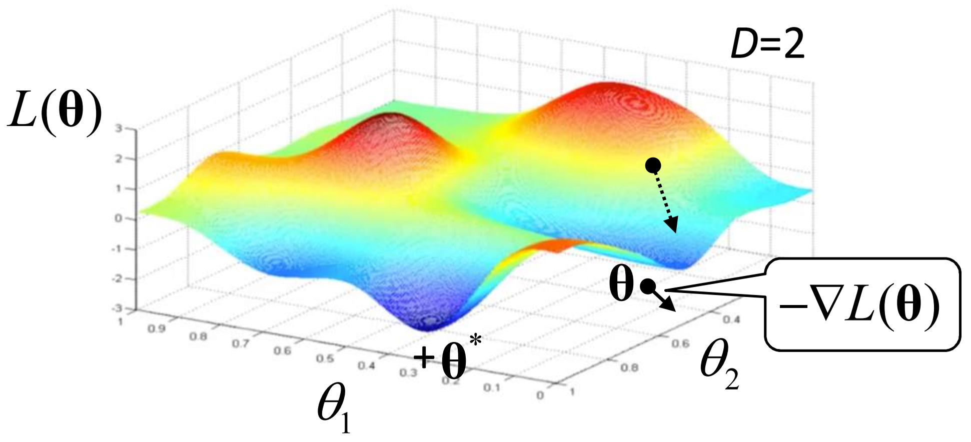
$$\frac{\partial f}{\partial x_3} = -x_1$$

Gradient at $\mathbf{x} = (1, 2, 3)^{\mathrm{T}}$:

$$\nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1}(\mathbf{x}) \\ \frac{\partial f}{\partial x_2}(\mathbf{x}) \\ \frac{\partial f}{\partial x_3}(\mathbf{x}) \end{pmatrix} = \begin{pmatrix} 12 - 3 \\ 3 + 8 \\ -1 \end{pmatrix} = \begin{pmatrix} 9 \\ 11 \\ -1 \end{pmatrix}$$

Gradient of Loss Function

- Idea: follow the slope. How do we get the local slope of the loss function?
- **Negative gradient points into the direction of steepest descent**

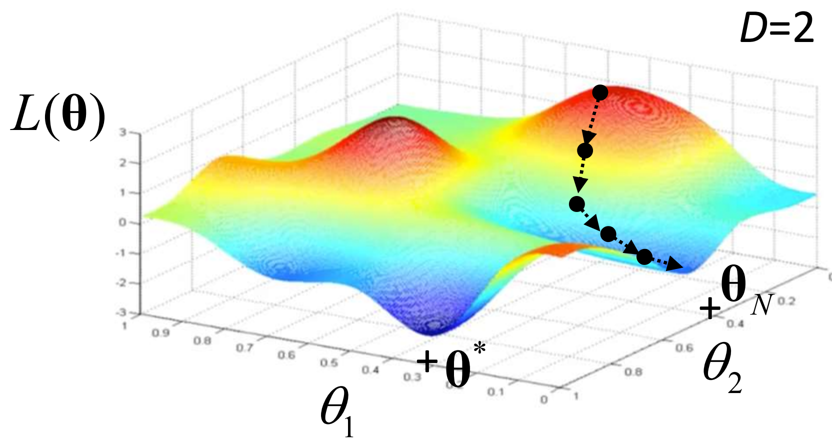


- For $\boldsymbol{\theta} \in \mathbb{R}^D$, the vector $-\nabla L(\boldsymbol{\theta}) \in \mathbb{R}^D$ points into the direction of steepest descent
- Locally moving into the direction of negative gradient will decrease loss

$$L(\boldsymbol{\theta} - \eta \nabla L(\boldsymbol{\theta})) \le L(\boldsymbol{\theta}) \qquad \text{for small enough } \eta > 0$$

Gradient Descent Algorithm (Fixed Step Size)

- Idea: follow the slope. How do we get the local slope of the loss function?
- **Gradient descent: iterative small steps in direction of negative gradient**
  - Step size: how far to move along direction of gradient? Parameter η
  - For small enough η, will converge to local optimum



Gradient descent algorithm

```
1. $\boldsymbol{\theta}_0 = \text{randomInitialization}()$
2. for $i = 0,\dots,i_{max}$:
3.         $\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta \nabla L(\boldsymbol{\theta}_i)$
4.         if $L(\boldsymbol{\theta}_i) - L(\boldsymbol{\theta}_{i+1}) < \epsilon$:
5.                 return $\boldsymbol{\theta}_{i+1}$
6. raise Exception("Not converged in $i_{max}$ iterations")
```

Computing the Gradient

- To carry out the gradient descent algorithm, we need to compute the gradient $\nabla L(\boldsymbol{\theta})$ for any given $\boldsymbol{\theta} \in \mathbb{R}^D$
- The function $L(\boldsymbol{\theta})$ can be quite complex
  - parameter vector $\boldsymbol{\theta} \in \mathbb{R}^D$ can be high-dimensional
  - computation of $L(\boldsymbol{\theta})$ involves computing model predictions, the loss function, and a sum over data instances

$$L(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_n), y_n)$$

- How can we compute the gradient in practice?

Numerical Gradients

- **First idea:** numerical gradient based on difference quotient

Let $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_D)^{\mathrm{T}} \in \mathbb{R}^D$, that is, $L : \mathbb{R}^D \to \mathbb{R}$.

Gradient is $\nabla L(\boldsymbol{\theta}) = \left(\frac{\partial L}{\partial \theta_1}(\boldsymbol{\theta}), \ldots, \frac{\partial L}{\partial \theta_D}(\boldsymbol{\theta}) \right)^{\mathrm{T}}$

Partial derivative is limes of difference quotient:

$$\frac{\partial L}{\partial \theta_d}(\boldsymbol{\theta}) = \lim_{h \to 0} \frac{L(\boldsymbol{\theta} + h\mathbf{u}_d) - L(\boldsymbol{\theta})}{h} \qquad \mathbf{u}_d = (0, 0, \ldots, 0, \underset{\uparrow}{1}, 0, \ldots, 0)^{\mathbf{T}} \in \mathbb{R}^D$$

d-th position

Approximate d-th entry in gradient by

$$\frac{\partial L}{\partial \theta_d}(\boldsymbol{\theta}) \approx \frac{L(\boldsymbol{\theta} + h\mathbf{u}_d) - L(\boldsymbol{\theta})}{h}$$ where *h* is a small number (e.g. $h = 10^{-4}$)

Numerical Gradients

- **First idea:** numerical gradient based on difference quotient
- To compute entire gradient, have to compute the approximation

$$\frac{\partial L}{\partial \theta_d}(\boldsymbol{\theta}) \approx \frac{L(\boldsymbol{\theta} + h\mathbf{u}_d) - L(\boldsymbol{\theta})}{h} \quad \text{for each } d \in \{1, \ldots, D\}$$

- As number of parameters D can be large this is usually not efficient: Would have to compute the entire loss many times to get a single gradient
- Also, solution is only approximate and not always numerically stable

- However, numerical gradients are easy to implement and can be used to verify (debug) other, more efficient implementations of gradient computations

Analytic Gradients

- **Second idea**: analytically derive gradient

Example from above:

$$f : \mathbb{R}^3 \to \mathbb{R} \quad \text{with} \quad f(\mathbf{x}) = 3x_1^2 x_2 + 2x_2^2 - x_1 x_3$$

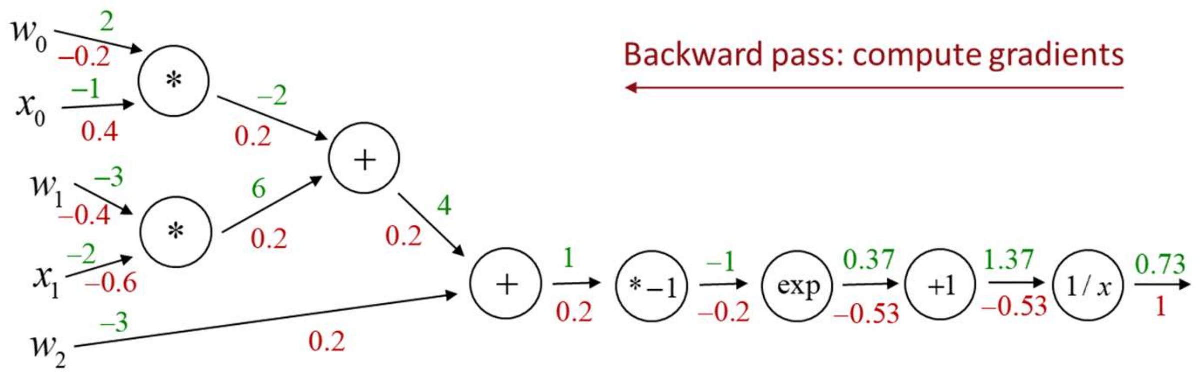
$$\frac{\partial f}{\partial x_1} = 6x_1 x_2 - x_3 \qquad \frac{\partial f}{\partial x_2} = 3x_1^2 + 4x_2 \qquad \frac{\partial f}{\partial x_3} = -x_1$$

- Advantages: exact solution, potentially faster

- This is a widely used approach for model classes and loss functions that are not too complex, such that corresponding closed-form solutions for the gradient can still be derived
- Will see examples below and in further lectures

Automatic Differentiation

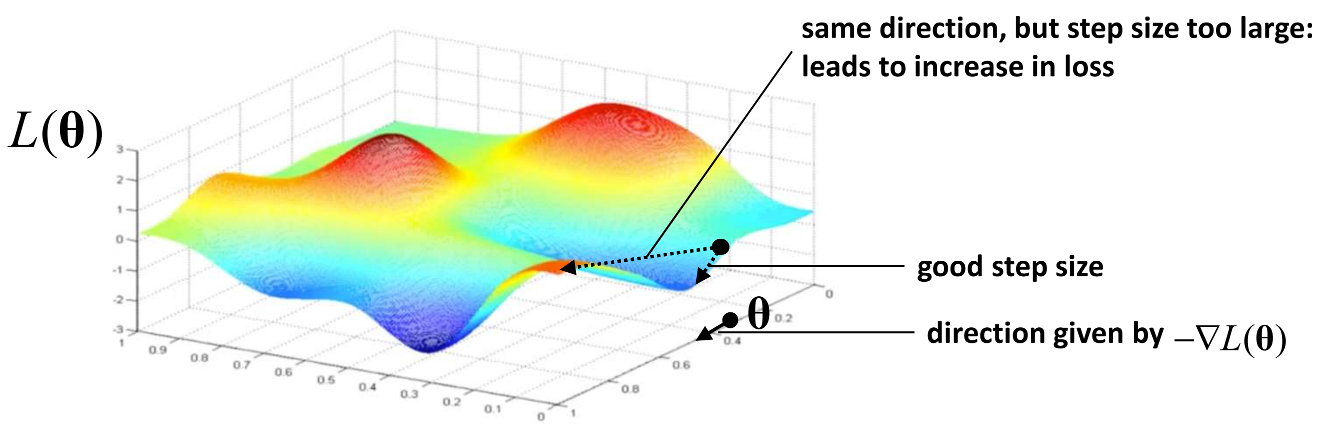
- **Alternative for models for which deriving analytic gradients is not (easily) possible: automatic differentiation**
  - algorithmic approach to compute gradient vector given model, loss, data
  - look at the overall expression defining $L(\boldsymbol{\theta})$ as a graph of elementary operations („computation graph“ or „data flow graph“)
  - if we know the derivates of the individual operations, we can efficiently compute the overall derivative by the chain rule
  - widely used for neural networks



- No details here, will discuss this in more detail when talking about neural networks

Step Sizes in Gradient Descent

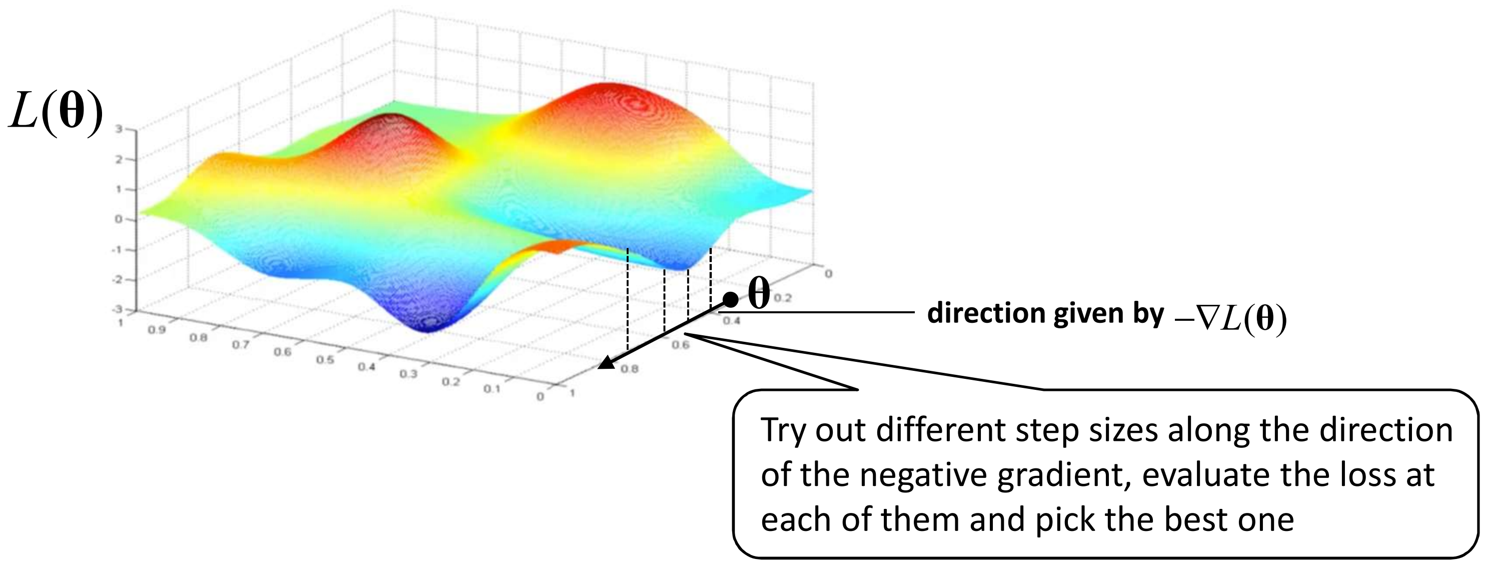
- For the gradient descent algorithm, we need to specify a step size η
- A step size is needed, because the negative gradient $-\nabla L(\boldsymbol{\theta})$ only points into the direction of steepest descent in an infinitisimal neighborhood of $\boldsymbol{\theta}$
- Therefore, a decrease in loss is only guaranteed for small enough η


same direction, but step size too large: leads to increase in loss
$L(\boldsymbol{\theta})$
good step size
$\boldsymbol{\theta}$
direction given by $-\nabla L(\boldsymbol{\theta})$

- If step size is too large, algorithm might not find any good $\boldsymbol{\theta}$
- If step size is too small, algorithm will take very long to find good $\boldsymbol{\theta}$

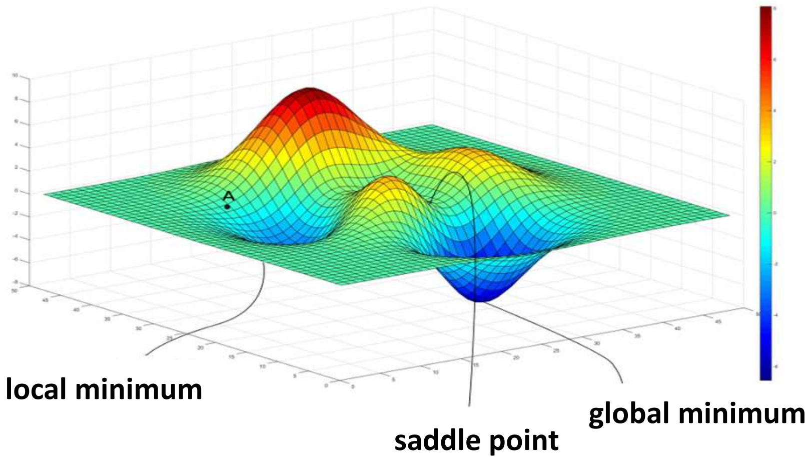
Selecting Step Sizes in Gradient Descent

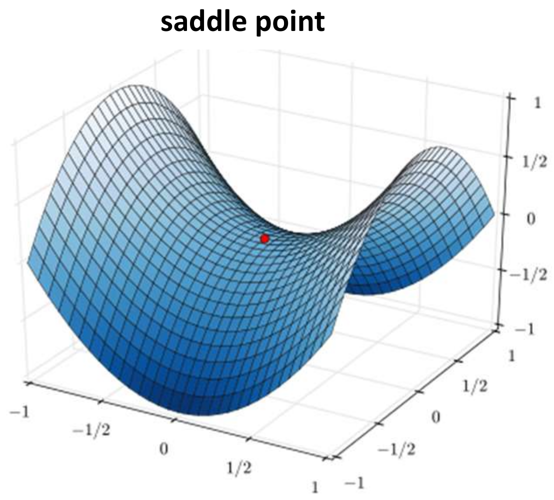
- How to select step size?
- Option 1: fixed step size which is a tunable hyperparameter (that is, have to empirically test different step sizes and pick the one that works best)
- Option 2: policy to reduce step size over time or adapt it heuristically (e.g. by monitoring how much parameters change at each iteration)
- Option 3: rather than using a fixed step size, perform a **line search** along the direction of the negative gradient


$L(\boldsymbol{\theta})$
$\boldsymbol{\theta}$
direction given by $-\nabla L(\boldsymbol{\theta})$
Try out different step sizes along the direction of the negative gradient, evaluate the loss at each of them and pick the best one

Local Minima

- In general, the function $L(\boldsymbol{\theta})$ can have several different **local minima**, that is, parameters $\boldsymbol{\theta}$ with $\nabla L(\boldsymbol{\theta}) = 0$
- Gradient descent will find one of the local minima, which is not necessarily globally optimal
- The gradient can also become zero at a saddle point, and (simple) gradient descent can then get „stuck“

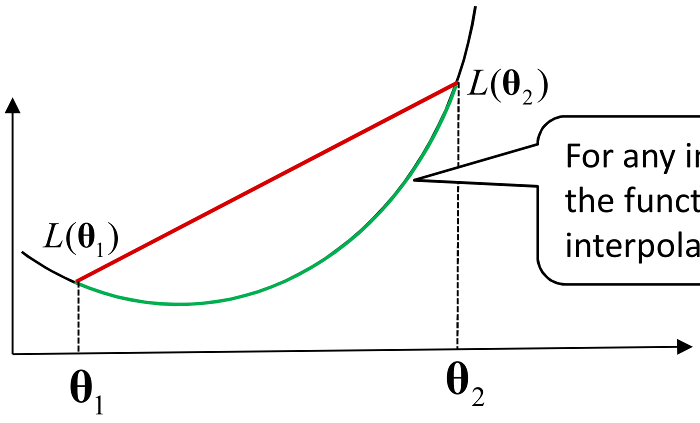




Convexity

- A function $L : \mathbb{R}^D \to \mathbb{R}$ is called **convex** if for any $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2 \in \mathbb{R}^D$ and any $t \in [0,1]$

$$L(t\boldsymbol{\theta}_1 + (1-t)\boldsymbol{\theta}_2) \leq tL(\boldsymbol{\theta}_1) + (1-t)L(\boldsymbol{\theta}_2)$$



For any interpolated point between $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$, the function value is below the linear interpolation of $L(\boldsymbol{\theta}_1)$ and $L(\boldsymbol{\theta}_2)$

- For a convex function, any local minimum is also a global minimum
- For a convex function, gradient descent will thus find a global optimum, given sufficiently small step sizes

Convexity

- If the function $L:\mathbb{R}^D \rightarrow \mathbb{R}$ is twice differentiable, it is convex if the Hessian matrix

$$\mathbf{H} = \begin{pmatrix} \frac{\partial L}{\partial\theta_1\partial\theta_1} & \dots & \frac{\partial L}{\partial\theta_1\partial\theta_D} \\ \dots & \dots & \dots \\ \frac{\partial L}{\partial\theta_D\partial\theta_1} & \dots & \frac{\partial L}{\partial\theta_D\partial\theta_D} \end{pmatrix}$$

  is positive semidefinite for all $\boldsymbol{\theta} \in \mathbb{R}^D$ (remember that $\mathbf{H}$ depends on $\boldsymbol{\theta}$)
- The matrix $\mathbf{H}$ is positive semidefinite if for all $\overline{\boldsymbol{\theta}} \in \mathbb{R}^D$ it holds that $\overline{\boldsymbol{\theta}}^{\mathrm{T}}\mathbf{H}\overline{\boldsymbol{\theta}} \geq 0$
- For any matrix $\mathbf{A} \in \mathbb{R}^{N\times M}$, the matrix $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ is positive semidefinite
- For linear regression with squared loss, the Hessian of the loss of the model in the model parameters is $(2/N)\mathbf{X}^{\mathrm{T}}\mathbf{X}$ (can be derived similarly as for normal equations)
  – Loss is therefore convex and gradient descent will find global optimum

Learning Linear Regression by Gradient Descent

- Instead of solving the normal equations, we can also use gradient descent to learn a linear regression model
- For linear regression, the gradient of the loss function is (result copied from derivation of normal equations above):

$$\nabla L(\boldsymbol{\theta}) = \frac{\partial}{\partial \boldsymbol{\theta}} \frac{1}{N} \left\| \mathbf{X}\boldsymbol{\theta} - \mathbf{y} \right\|_2^2 = \frac{1}{N} (-\mathbf{X})^{\mathrm{T}} 2(\mathbf{y} - \mathbf{X}\boldsymbol{\theta})$$

- Can be computed readily from training data and current parameter vector
- Pick learning rate η and run gradient descent algorithm:

Gradient descent algorithm

```
1.  $\boldsymbol{\theta}_0 = \mathrm{randomInitialization()}$
2.  for $i = 0, \ldots, i_{max}$:
3.      $\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta \nabla L(\boldsymbol{\theta}_i)$
4.      if $L(\boldsymbol{\theta}_i) - L(\boldsymbol{\theta}_{i+1}) < \epsilon$:
5.          return $\boldsymbol{\theta}_{i+1}$
6.  raise Exception("Not converged in $i_{max}$ iterations")
```

Sparsity in Linear Regression

- In some application domains, instance vectors $\mathbf{x} \in \mathbb{R}^M$ will be high-dimensional but sparse, meaning that most of the entries in a vector are typically zero
- For example, using a word indicator representation for spam classification: there are tens of thousands of words, but only a few of them appear in the average message

- In gradient descent, we can take advantage of this sparsity:

Multiplication of sparse matrix with dense vector is efficient: only go over non-zero entries in matrix

$$\nabla L(\boldsymbol{\theta}) = \frac{1}{N}(-\mathbf{X})^{\mathrm{T}} 2(\mathbf{y} - \mathbf{X}\boldsymbol{\theta})$$

Again, multiplication of sparse matrix with dense vector

- Computational complexity of gradient descent is then $O(I(NM_{nz} + M))$
  - I is number of iterations, M_{nz} average number of nonzero elements in row of $\mathbf{X}$
- In contrast, approaches to solving the normal equations cannot easily take advantage of sparsity

Example: Linear Regression

- **Example: gradient descent for linear regression**

One-dimensional model $f:\mathbb{R}\rightarrow\mathbb{R}$, using a constant attribute $x_2 = 1$:

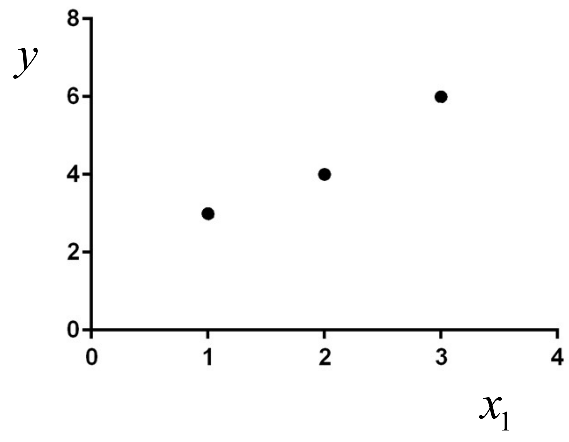
$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \theta_1 x_1 + \theta_2 x_2 = \theta_1 \underset{\substack{\uparrow\\\text{input}}}{x_1} + \theta_2$$

Assume three training instances. The training data can be given in matrix form with $\mathbf{X}\in\mathbb{R}^{3\times 2}$ and $\mathbf{y}\in\mathbb{R}^3$:

$$\mathbf{X} = \begin{pmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \end{pmatrix} \qquad \mathbf{y} = \begin{pmatrix} 3 \\ 4 \\ 6 \end{pmatrix}$$

inputs x_1 $\uparrow$ $\uparrow$ constant attribute x_2

targets y $\uparrow$



Example: Linear Regression

- **Example: gradient descent for linear regression**

  We can train the model using a squared loss function and gradient descent:

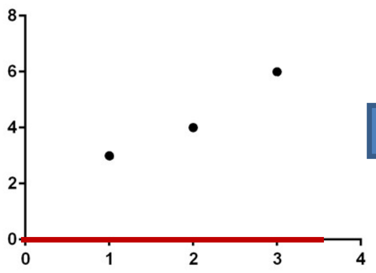
$$\boxed{\begin{aligned}\boldsymbol{\theta}^* &= \arg\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) \\ L(\boldsymbol{\theta}) &= \frac{1}{3}\sum_{n=1}^{3}(f_{\boldsymbol{\theta}}(\mathbf{x}_n) - y_n)^2\end{aligned}}$$

Gradient descent algorithm

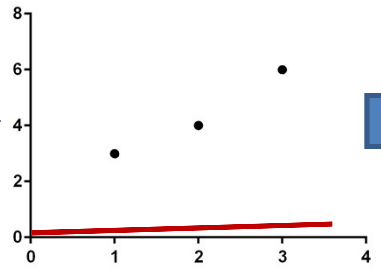
```
1. $\boldsymbol{\theta}_0$ = randomInitialization()
2. for $i = 0,\dots,i_{max}$:
3.     $\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta \nabla L(\boldsymbol{\theta}_i)$
4.     if $L(\boldsymbol{\theta}_i) - L(\boldsymbol{\theta}_{i+1}) < \epsilon$ :
5.         return $\boldsymbol{\theta}_{i+1}$
3. raise Exception("Not converged in $i_{max}$ iterations")
```

For example, initialize to zeros

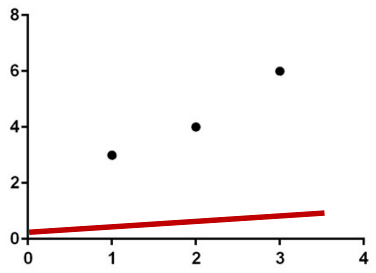
i=0



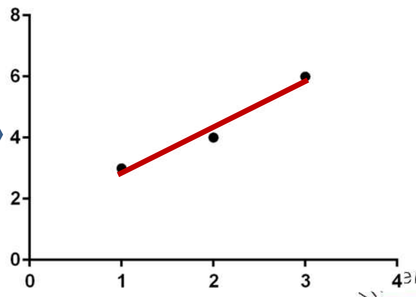
i=1



i=2



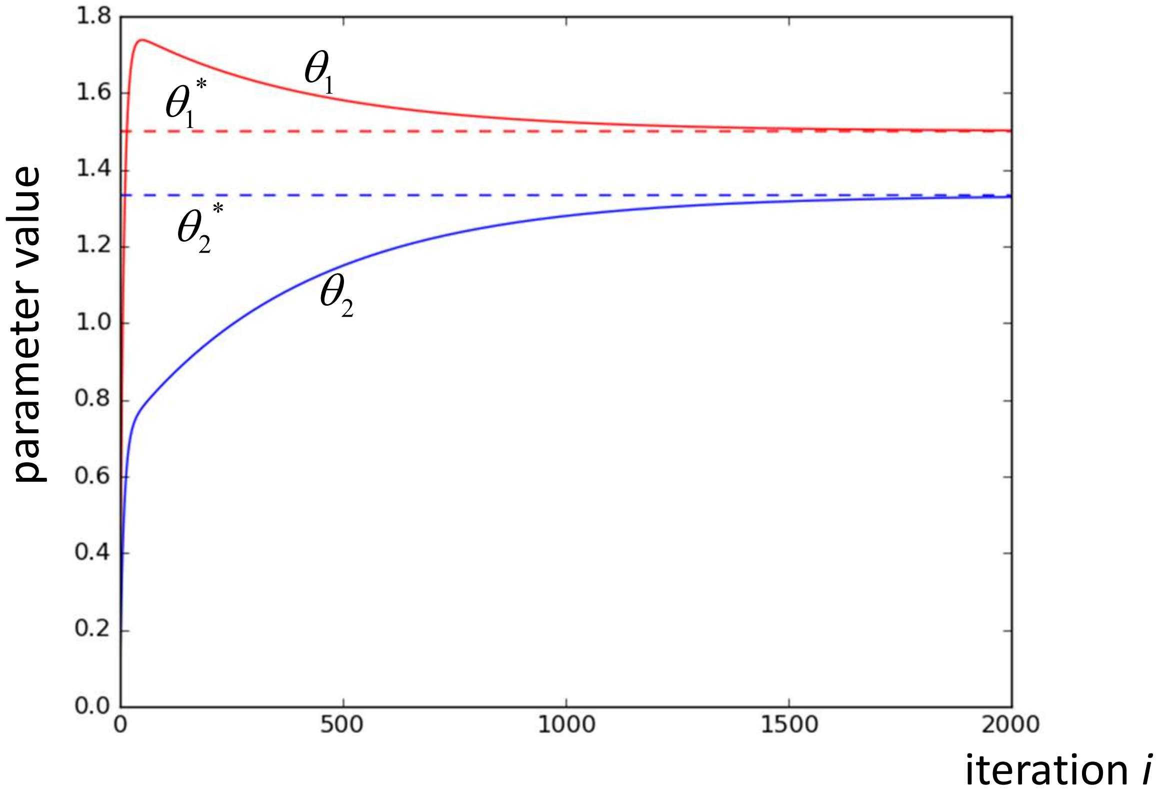
i=2000



Gradient Descent: Example

- **Example: gradient descent for linear regression**

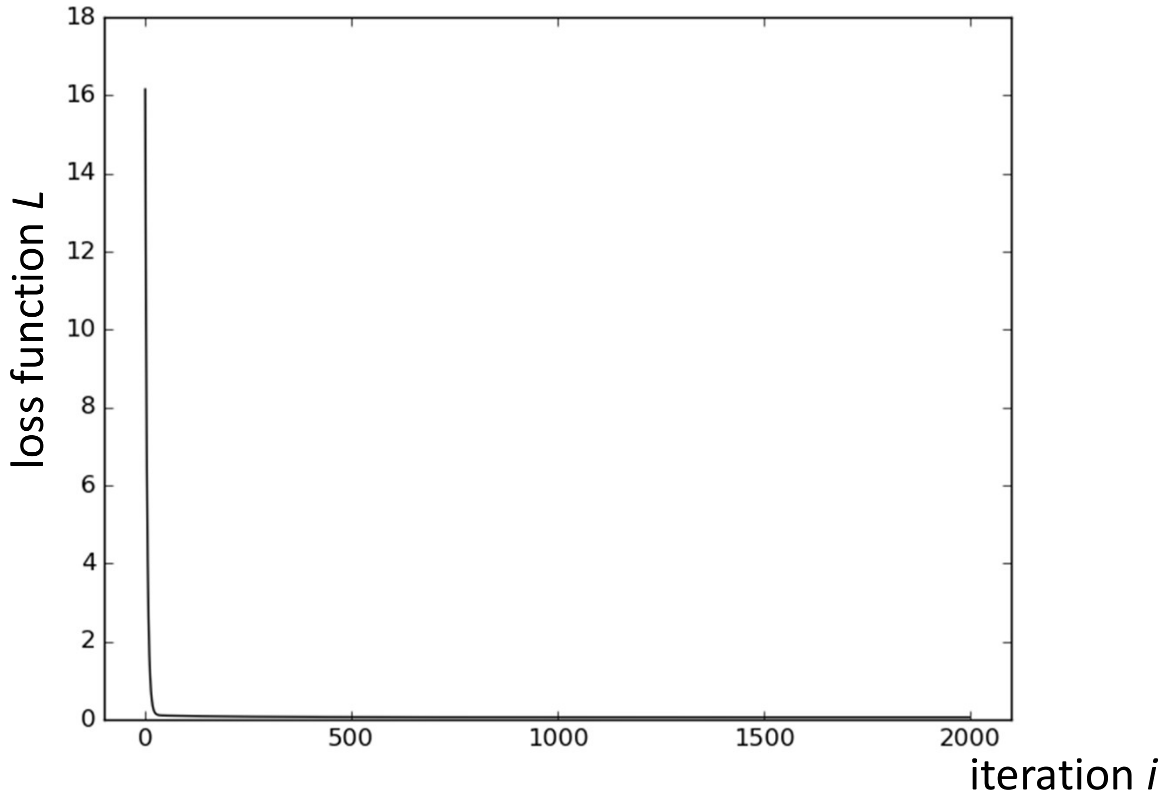
Development of model parameters θ_1, θ_2 during gradient descent



Gradient Descent: Example

- **Example: gradient descent for linear regression**

Development of loss function during gradient descent



Summary

- **Linear regression** is a simple but widely used model for regression problems
- The model can be learned from data by minimizing a **loss function**, e.g. squared loss
- Minimizing the loss function is an optimization problem, which for linear regression can be solved by solving the normal equations (system of linear equations)
- Alternatively, **gradient descent** is a flexible iterative optimization method that can be used to learn linear regression and also many other models
  - start with random parameter vector
  - repeatedly update the parameter vector by following the direction of steepest descent, given by the negative gradient