

# **Data, Models, Optimization**

Advanced Computer Vision

Niels Landwehr

# Overview: Lectures so Far

- Introduction: Computer Vision
- Data, Models, Optimization

# Supervised Learning: Instances

- In the lecture, we will mostly be concerned with **supervised machine learning**
- Objects that we want to classify or otherwise reason about are called **instances**, typically denoted by  $\mathbf{x}$
- Instances live in an instance space  $\mathcal{X}$ , that is,  $\mathbf{x} \in \mathcal{X}$
- For images:

$$\mathcal{X} = \mathbb{R}^{m \times l \times d} \quad \begin{array}{l} m = \text{image height} \\ l = \text{image width} \\ d = \text{number of channels} \end{array}$$

- Notation:  $\mathbb{R}$  are the real numbers.  $\mathbb{R}^{m \times l \times d}$  is the space of all three-dimensional tensors of size  $m$  by  $l$  by  $d$ .

vector,  $\mathbb{R}^3$

$$\begin{pmatrix} 0.5 \\ 0.2 \\ 0.7 \end{pmatrix}$$

matrix,  $\mathbb{R}^{3 \times 3}$

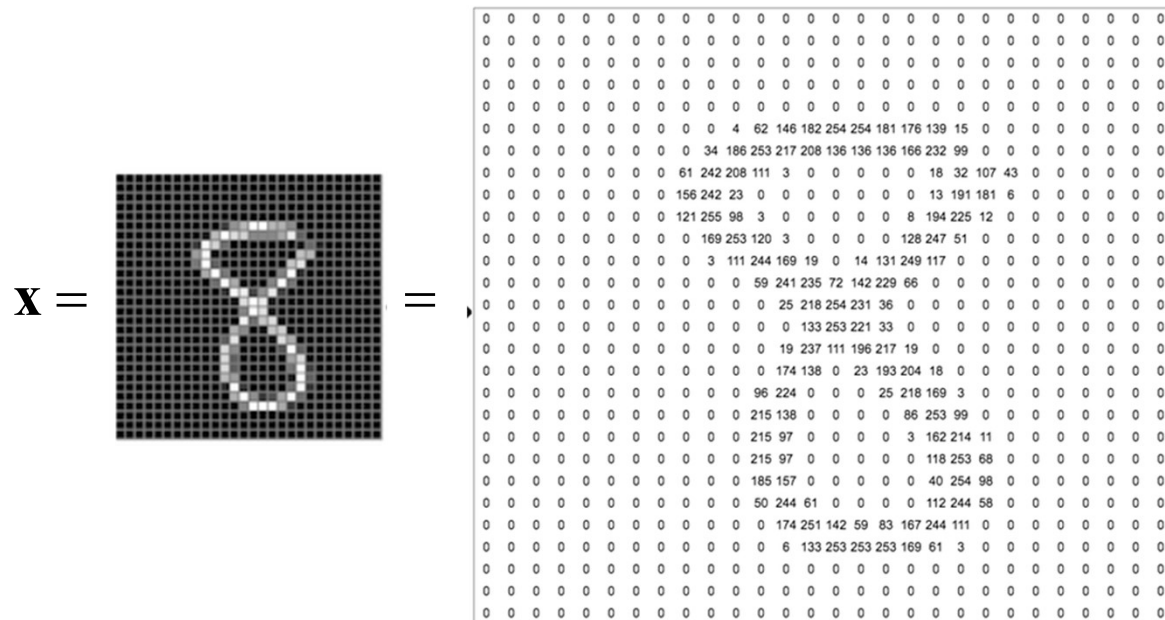
$$\begin{pmatrix} 0 & 0.3 & 1 \\ 1 & 0.7 & 0.5 \\ 0.2 & 0.5 & 0.1 \end{pmatrix}$$

3D-tensor,  $\mathbb{R}^{3 \times 3 \times 3}$

$$\begin{pmatrix} \begin{pmatrix} 0 & 0.3 & 1 \end{pmatrix} \\ \begin{pmatrix} 0 & 0.3 & 1 \end{pmatrix} \\ \begin{pmatrix} 1 & 0.7 & 0.5 \end{pmatrix} \\ \begin{pmatrix} 0.2 & 0.5 & 0.1 \end{pmatrix} \end{pmatrix} \begin{matrix} 1 \\ 5 \\ 1 \\ 1 \end{matrix}$$

# Image Instances

- Number of channels can be 1 (greyscale), 3 (color), or >3 (multispectral)
- Requirement that height and width are fixed can be relaxed
- **Example:** MNIST data set, 28 x 28 greyscale images,  $\mathcal{X} = \mathbb{R}^{28 \times 28 \times 1}$

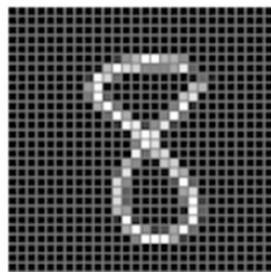


# Label or Target

- In supervised learning, a model maps instances to a **target variable** or **label**

$$\begin{array}{c} \text{Model} \\ \mathbf{x} \mapsto y \end{array}$$

- The target variable lives in a label or target space,  $y \in \mathcal{Y}$
- Different target spaces depending on the learning setting
  - Classification:  $\mathcal{Y} = \{c_1, \dots, c_k\}$ , where the  $c_i$  are different classes
  - Regression:  $\mathcal{Y} = \mathbb{R}$ , trying to estimate a numeric value
  - Many advanced learning settings where  $\mathcal{Y}$  can take different forms
- Example for classification:



$$\begin{array}{c} \text{Model} \\ \mapsto \text{„8“} \end{array}$$

# Models

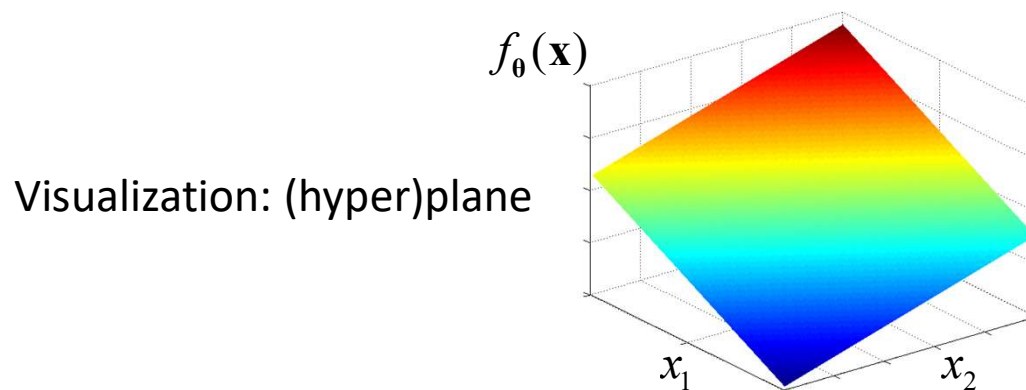
- A **model** is a function  $f : \mathcal{X} \rightarrow \mathcal{Y}$  that maps instances to labels:

$$\begin{array}{c} f \\ \mathbf{x} \mapsto y \\ \begin{array}{c} \text{8} \\ \mapsto \text{„8“} \end{array} \end{array}$$

- Most models in computer vision are parameterized functions  $f_{\theta}$ , where  $\theta$  is a vector of parameters or more generally a structured set of parameters
- The function is determined by its structure and its parameters  $\theta \in \Theta$ , where  $\Theta$  is a space of parameters
- The structure of the function is chosen a priori, the parameters are chosen during learning based on the data

# Linear Models: Regression

- Example: linear model for regression
  - Assume  $\mathcal{X} = \mathbb{R}^d$  (for images, could „flatten“ the image into a long vector of numbers)
  - Regression:  $\mathcal{Y} = \mathbb{R}$
  - Linear model:
$$f_{\theta}(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b \quad \mathbf{w} \in \mathbb{R}^d, \quad b \in \mathbb{R}, \quad \theta = (\mathbf{w}, b)$$
- Parameter vector  $\mathbf{w}$  is called weight vector,  $b$  is called bias
- If  $d$  is large, even this simple model can be relatively expressive (e.g. for text)



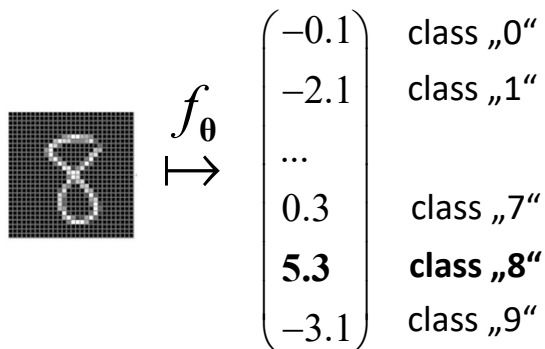
# Linear Models: Classification

- Example: linear model for classification („logistic regression“)
  - Assume  $\mathcal{X} = \mathbb{R}^d$  (for images, could „flatten“ the image into a long vector of numbers)
  - Classification:  $\mathcal{Y} = \{c_1, \dots, c_k\}$
  - Linear model:

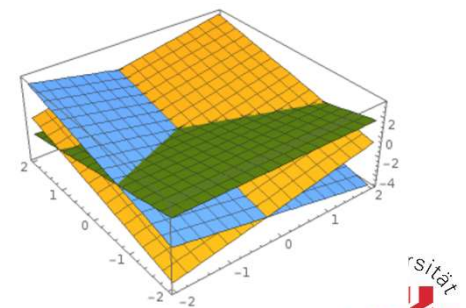
$$f_{\theta}(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}$$

$$\mathbf{W} \in \mathbb{R}^{k \times d}, \quad \mathbf{b} \in \mathbb{R}^k, \quad \theta = (\mathbf{W}, \mathbf{b})$$

- The model  $f_{\theta}$  returns a  $k$ -dimensional vector, whose elements are interpreted as class scores. The prediction is the class with highest score



Rows of  $\mathbf{W}$ :  
planes that partition  
 $\mathbf{x}$ -space into classes











# Learning: Data

- So far: defined parametric models  $f_{\theta}$  that can make predictions for instances
- Model determined by parameter vector  $\theta$ . How do we find a good  $\theta$ ?
- Models are learned from training data: a set of instances together with (presumably) correct labels
- We will denote the set of training instances by  $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_n)$ , and the set of corresponding training labels by  $\mathbf{y} = (y_1, \dots, y_n)$
- Assume that training data is sampled independently from joint distribution:

$$\mathbf{x}_i, y_i \sim p(\mathbf{x}, y)$$

- Example:

$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{x}_4$	$\mathbf{x}_5$	...	$\mathbf{x}_n$
						
„2“	„7“	„1“	„5“	„2“		„0“
$y_1$	$y_2$	$y_3$	$y_4$	$y_5$		$y_n$

# Learning: Loss Function

- How to find good model parameters  $\theta$  based on training data  $\mathbf{X}, \mathbf{y}$  ?
  - Need to quantify how good a set of model parameters is: **loss function**
  - Need to find model parameters that are good according to loss function: **optimization**
- A **loss function** measures how well a model  $f_{\theta}$  performs on training data
- The lower the loss, the better the model parameters  $\theta$
- Also sometimes called cost function or objective function
- Loss on a single example  $\mathbf{x}_i, y_i$ :  $\ell(f_{\theta}(\mathbf{x}_i), y_i)$
- Measures the error/disagreement between prediction  $f_{\theta}(\mathbf{x}_i)$  and label  $y_i$
- Loss on entire data set is the average of losses on all instances:

$$L(\theta) = \frac{1}{n} \sum_{i=1}^n \ell(f_{\theta}(\mathbf{x}_i), y_i)$$

# Learning: Loss Function

- **Example: absolute and squared loss (regression)**

- absolute loss  $\ell_1(f_\theta(\mathbf{x}), y) = |f_\theta(\mathbf{x}) - y|$
- squared loss  $\ell_2(f_\theta(\mathbf{x}), y) = (f_\theta(\mathbf{x}) - y)^2$

- **Example: cross-entropy loss (classification)**

- Model outputs class scores. Idea: transform to probabilities

$$p(y = c_j | \mathbf{x}_i, \boldsymbol{\theta}) = \frac{\exp(f_\theta(\mathbf{x}_i)_j)}{\sum_{j'=1}^k \exp(f_\theta(\mathbf{x}_i)_{j'})}$$

Diagram annotations:

- A callout box labeled "predicted probability for class  $j$ " points to the left side of the equation.
- A callout box labeled "score for class  $j$ " points to the numerator  $\exp(f_\theta(\mathbf{x}_i)_j)$ .

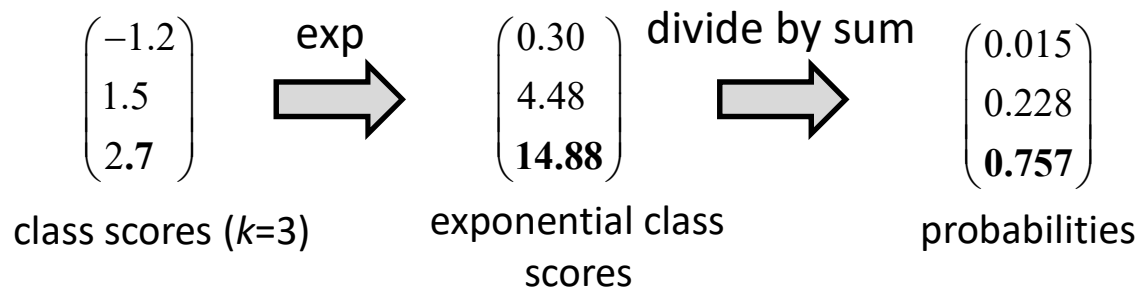
**Example ( $k=3$ ):**

$$\begin{array}{ccccc}
 \begin{pmatrix} -1.2 \\ 1.5 \\ 2.7 \end{pmatrix} & \xrightarrow{\text{exp}} & \begin{pmatrix} 0.30 \\ 4.48 \\ 14.88 \end{pmatrix} & \xrightarrow{\text{divide by sum}} & \begin{pmatrix} 0.015 \\ 0.228 \\ 0.757 \end{pmatrix} \\
 f_\theta(\mathbf{x}_i) & & \exp(f_\theta(\mathbf{x}_i)) & & p(y = c_1 | \mathbf{x}_i, \boldsymbol{\theta}) \\
 & & & & p(y = c_2 | \mathbf{x}_i, \boldsymbol{\theta}) \\
 & & & & p(y = c_3 | \mathbf{x}_i, \boldsymbol{\theta}) \\
 & & & & \text{probabilities}
 \end{array}$$

# Learning: Loss Function

- **Example, continued: cross-entropy loss (classification)**
  - Probability according to model for observed true class for instance  $i$ :  
 $p(y = y_i | \mathbf{x}_i, \boldsymbol{\theta})$
  - Cross-entropy loss: negative log-probability of observed class  
 $\ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) = -\log p(y = y_i | \mathbf{x}_i, \boldsymbol{\theta})$

**Example ( $k=3$ ):** Assume true class is  $y_i = c_2$



$$\ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) = -\log p(y = c_2 | \mathbf{x}_i, \boldsymbol{\theta}) = -\log(0.228) \approx 1.48$$

natural logarithm

# Learning: Regularization

- Loss function expresses preference between model parameters: parameters  $\theta$  that match the data better are preferable

$$L(\theta) = \frac{1}{n} \sum_{i=1}^n \ell(f_{\theta}(\mathbf{x}_i), y_i)$$

how well does model predict i-th label?

- Often, a regularization term is added to loss function

$$L(\theta) = \underbrace{\frac{1}{n} \sum_{i=1}^n \ell(f_{\theta}(\mathbf{x}_i), y_i)}_{\text{Loss}} + \underbrace{\lambda R(\theta)}_{\text{Regularization}}$$

**Loss:** model predictions should match training data

**Regularization:** prevent model from doing **too** well on the training data, prefer simpler models

- Common simple regularizers:

L2-regularizer  $R(\theta) = \sum_j \theta_j^2$

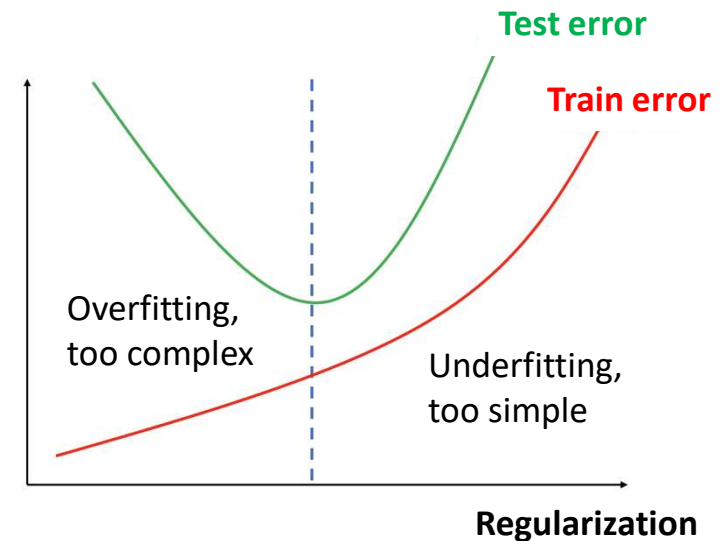
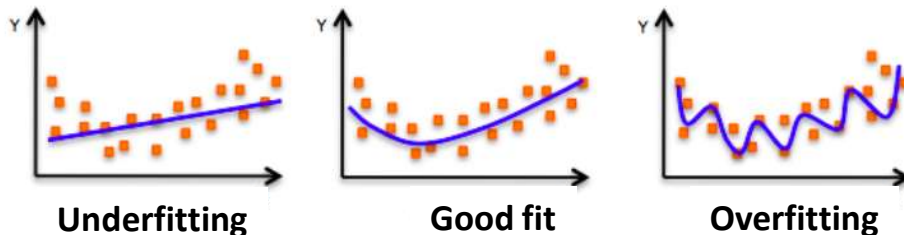
L1-regularizer  $R(\theta) = \sum_j |\theta_j|$

$\theta_j$ : any model parameter, e.g. for linear classification model the entries in  $\mathbf{W}, \mathbf{b}$

# Learning: Regularization

- Regularization: express preference for models beyond matching training data
  - Regularization makes it harder for model to perfectly fit training data
  - Avoid **overfitting**: regularized models (sometimes) generalize better
  - Most important for small data sets

Simple versus complex models



- L1/L2 regularization only one (simple) technique, other techniques: early stopping, dropout, cutout, mixup, batch normalization, ...

# Learning: Optimization

- How to find good model parameters  $\theta$  based on training data  $\mathbf{X}, \mathbf{y}$  ?
  - Need to quantify how good a set of model parameters is: **loss function**
  - Need to find model parameters that are good according to loss function: **optimization**

$$\text{Optimization: } \theta^* = \arg \min_{\theta} L(\theta)$$

- Challenging: large models, high-dimensional  $\theta$ , large data sets
- How to solve?
  - approximately or exactly?
  - computational efficiency?
  - numerical stability?

# Optimization: Closed-Form Solution

- **Simple example: Linear regression**
- Assume that each instance is given by vector of  $d$  attributes:  $\mathcal{X} = \mathbb{R}^d$
- Training instances can be summarized in  $n$  by  $d$  matrix:

$$\mathbf{X} \in \mathbb{R}^{n \times d} \quad \text{each row describes attribute values for one instance}$$

- Training labels can be summarized in  $n$ -dimensional vector:

$$\mathbf{y} \in \mathbb{R}^n \quad \text{each entry describes label for one instance}$$

- Assume linear regression without bias (can be incorporated by additional constant attribute):

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{w}^T \mathbf{x} \quad \mathbf{w} \in \mathbb{R}^d, \quad \boldsymbol{\theta} = \mathbf{w}$$

- Assume that objective function is squared loss plus squared regularizer:

$$L(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) + \lambda \|\mathbf{w}\|_2^2$$



# Optimization: Closed-Form Solution

- **Simple example: Linear regression**
- Optimization problem:

$$\mathbf{w}^* = \arg \min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^n \ell(f_{\theta}(\mathbf{x}_i), y_i) + \lambda \|\mathbf{w}\|_2^2$$

- In this simple example, there is a closed-form solution (derived by calculating the first derivative of  $L(\mathbf{w})$  with respect to  $\mathbf{w}$  and setting to zero):

$$\mathbf{w}^* = (\mathbf{X}^T \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^T \mathbf{y} \quad \mathbf{I} = \begin{pmatrix} 1 & \dots & 0 \\ \dots & \ddots & \dots \\ 0 & \dots & 1 \end{pmatrix} \in \mathbb{R}^d \quad \text{„identity matrix“}$$

- For most practically relevant models no closed-form solutions available
  - Need to resort to iterative numerical optimization algorithm to compute  $\theta^*$
  - For some models (e.g. SVMs) possible to find exact solution, for other models (e.g. deep neural networks) „good enough“ approximations

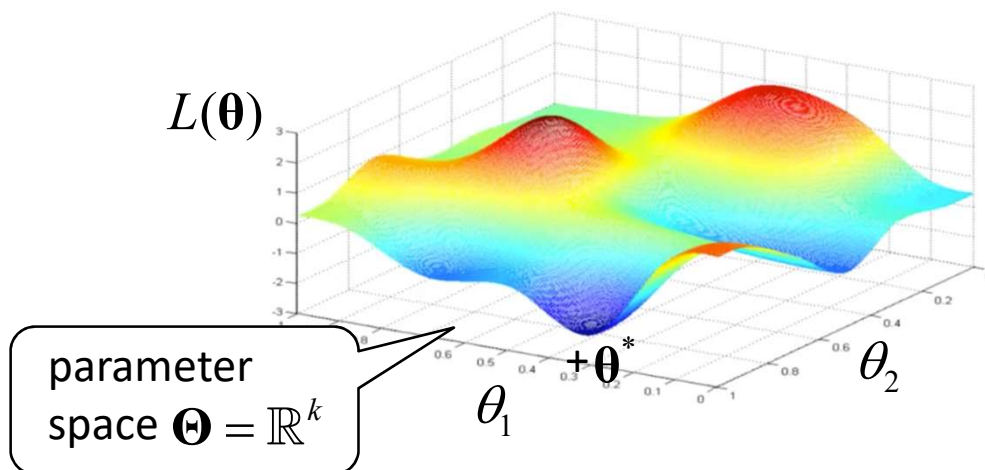
# Optimization: Gradient Descent

- For large models such as deep neural networks, often use algorithms based on **gradient descent**: follow the (local) slope of the loss function



# Loss Function Surface

- The loss function  $L(\boldsymbol{\theta})$  maps parameter vectors to a real number:  $L : \Theta \rightarrow \mathbb{R}$
- Assume  $\Theta = \mathbb{R}^k$  (flatten all parameters into a long vector)
- 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  $f: \mathbb{R}^k \rightarrow \mathbb{R}$  denote a scalar-valued differentiable function. The gradient of  $f$ , written  $\nabla f$ , is a function  $\nabla f: \mathbb{R}^k \rightarrow \mathbb{R}^k$  defined by

$$\nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1}(\mathbf{x}) \\ \frac{\partial f}{\partial x_2}(\mathbf{x}) \\ \dots \\ \frac{\partial f}{\partial x_k}(\mathbf{x}) \end{pmatrix}$$

partial derivative  
of  $f$  with respect to  $x_1$

# Gradient of Function

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

$$f : \mathbb{R}^3 \rightarrow \mathbb{R} \quad \text{with} \quad f(\mathbf{x}) = 3x_1^2x_2 + 2x_2^2 - x_1x_3 \quad \mathbf{x} = (x_1, x_2, x_3)^T$$

Partial derivatives:

$$\frac{\partial f}{\partial x_1} = 6x_1x_2 - x_3$$

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

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

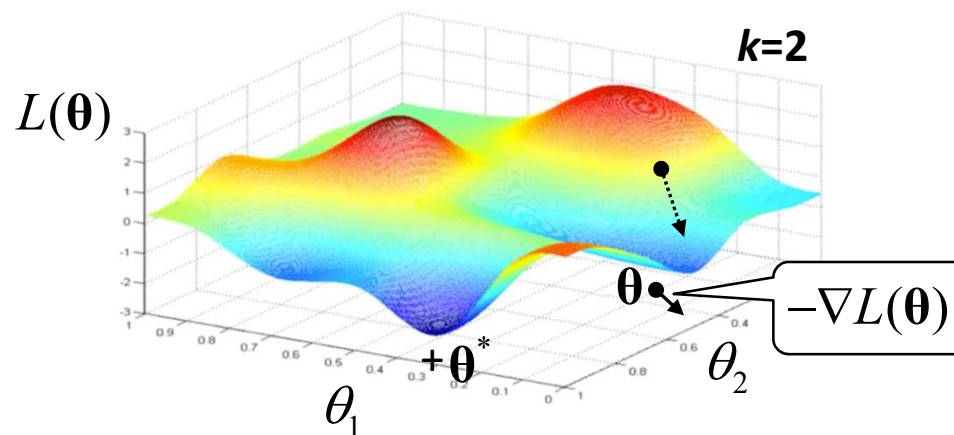
Gradient at  $\mathbf{x} = (1, 2, 3)^T$ :

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



# 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**

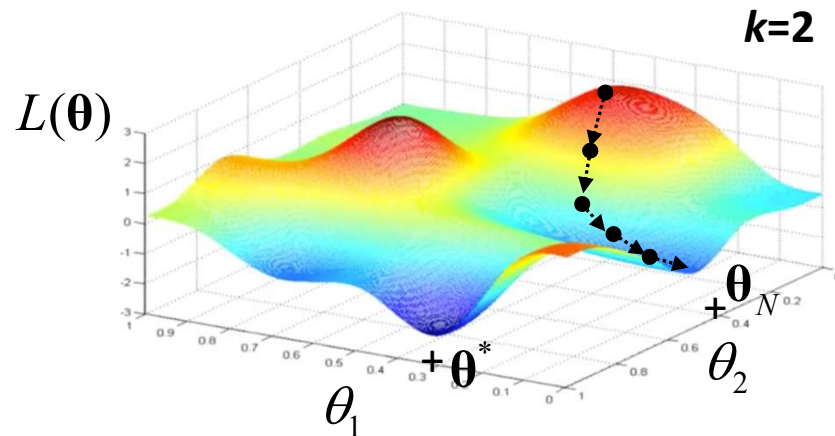


- For  $\theta \in \Theta$ , the vector  $-\nabla L(\theta) \in \Theta$  points into the direction of steepest descent
- Locally moving into the direction of negative gradient will decrease loss

$$L(\theta - \eta \nabla L(\theta)) < L(\theta) \quad \text{for small enough } \eta > 0 \text{ and } \nabla L(\theta) \neq 0$$

# Gradient Descent Algorithm

- 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  $\eta$
  - For small enough  $\eta$ , will converge to local optimum



## Gradient descent algorithm

1.  $\theta_0 = \text{randomInitialization}()$
2. for  $i = 0, \dots, N$ :  
$$\theta_{i+1} = \theta_i - \eta \nabla L(\theta_i)$$
3. return  $\theta_N$

# How to Compute the Gradient?

- Crucial step in gradient descent algorithm: compute  $\nabla L(\boldsymbol{\theta})$
- The function  $L(\boldsymbol{\theta})$  is typically very complex
  - parameter vector  $\boldsymbol{\theta} \in \mathbb{R}^k$  with often  $k > 1.000.000$
  - computation of  $L(\boldsymbol{\theta})$  involves computing model predictions, the loss function, and a sum over data instances

$$L(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) + \lambda R(\boldsymbol{\theta})$$

Enough to derive  
gradient for each  
element in sum

Model can  
be very complex!

Typically  
not so bad

- **How to compute gradients efficiently and robustly?**



# Numerical Gradients

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

Let  $\boldsymbol{\theta} = (\theta_1, \dots, \theta_k)^T \in \mathbb{R}^k$ , that is,  $L: \mathbb{R}^k \rightarrow \mathbb{R}$ .

$$\text{Gradient is } \nabla L(\boldsymbol{\theta}) = \left( \frac{\partial L}{\partial \theta_1}(\boldsymbol{\theta}), \dots, \frac{\partial L}{\partial \theta_k}(\boldsymbol{\theta}) \right)^T$$

Partial derivative is limes of difference quotient:

$$\frac{\partial L}{\partial \theta_i}(\boldsymbol{\theta}) = \lim_{h \rightarrow 0} \frac{L(\boldsymbol{\theta} + h\mathbf{u}_i) - L(\boldsymbol{\theta})}{h} \quad \mathbf{u}_i = (0, 0, \dots, 0, \underset{\substack{\uparrow \\ \text{i-th position}}}{1}, 0, \dots, 0)^T \in \mathbb{R}^k$$

Approximate i-th entry in gradient by

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

# Numerical Gradients

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

$$\frac{\partial L}{\partial \theta_i}(\boldsymbol{\theta}) \approx \frac{L(\boldsymbol{\theta} + h\mathbf{u}_i) - L(\boldsymbol{\theta})}{h} \quad \text{for each } i \in \{1, \dots, k\}.$$

- As number of parameters  $k$  is often large ( $k > 1.000.000$ ), not feasible: Would have to compute the entire loss millions of 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:

$$f : \mathbb{R}^3 \rightarrow \mathbb{R} \quad \text{with} \quad f(\mathbf{x}) = 3x_1^2x_2 + 2x_2^2 - x_1x_3$$

$$\frac{\partial f}{\partial x_1} = 6x_1x_2 - x_3$$

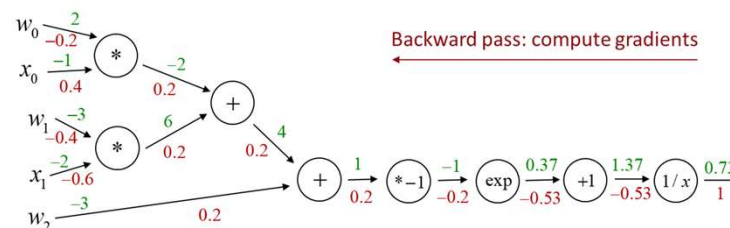
$$\frac{\partial f}{\partial x_2} = 3x_1^2 + 4x_2$$

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

- Advantages: exact solution, potentially faster
- Do this manually?
  - Writing down  $L(\boldsymbol{\theta})$  as a closed-form expression? For complex models, hundreds of millions of variables, billions of mathematical operations
  - For any new model structure or loss function, would have to redo this
- Manually deriving derivatives and optimization algorithms is possible for some classes of models (e.g. SVMs), but not for others, e.g. deep neural networks

# Automatic Differentiation

- **Widely used practical solution: automatic differentiation**
  - analytic approach to gradient computation, but not manual
  - algorithmic approach to compute gradient vector given model, loss, data
  - do not have to explicitly write down large closed-form expressions
- **Idea of automatic differentiation:**
  - look at the overall expression defining  $L(\theta)$  as a graph of elementary operations („computation graph“ or „data flow graph“)
  - if we know the derivatives of the individual operations, we can efficiently compute the overall derivative by the chain rule
  - more details later („backpropagation“ algorithm for neural networks)



# Stochastic Gradient Descent

- Gradient descent: in each iteration, change the parameter vector  $\theta$  in the direction of the negative gradient of the loss function
- The loss function and thus the gradient are defined by a sum over data set:

$$\nabla L(\theta) = \nabla \left( \frac{1}{n} \sum_{i=1}^n \ell(f_{\theta}(\mathbf{x}_i), y_i) \right) + \nabla (\lambda R(\theta))$$

- Computing the gradient scales linearly with the size of the data set. If the data set is large, even computing a single parameter update will be expensive
- **Idea:** do not compute gradient on all of the data, but on (small) subset:

$$\nabla L(\theta) \approx \nabla \left( \frac{1}{m} \sum_{j=1}^m \ell(f_{\theta}(\mathbf{x}_{i_j}), y_{i_j}) \right) + \nabla (\lambda R(\theta)) \quad \{i_1, \dots, i_m\} \subset \{1, \dots, n\}, \quad m \ll n$$

- Gradient approximated on a small random subset: noisy/stochastic gradient
- This is called **Stochastic Gradient Descent**, or **SGD**.

# Stochastic Gradient Descent

- What happens if we compute the gradient on random subset?
  - Individual gradients will be „noisy“ in the sense that they have a strong random component
  - However, in expectation, gradient will still be identical to „full“ gradient computed on the entire data set:

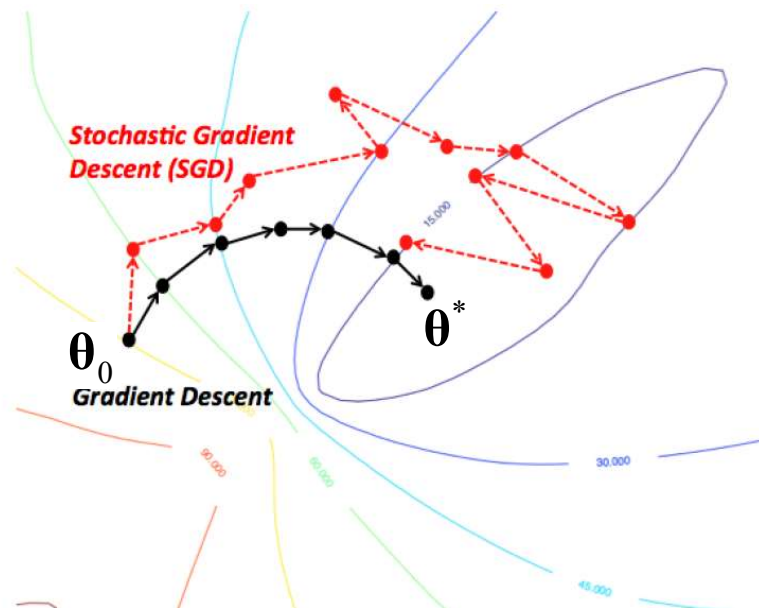
$$\mathbb{E} \left[ \nabla \left( \frac{1}{m} \sum_{i=1}^m \ell(f_{\theta}(\mathbf{x}_{i_m}), y_{i_m}) \right) \right] = \nabla \left( \frac{1}{n} \sum_{i=1}^n \ell(f_{\theta}(\mathbf{x}_i), y_i) \right)$$

Expectation over draws of random subsets  $\{i_1, \dots, i_m\} \subset \{1, \dots, n\}$

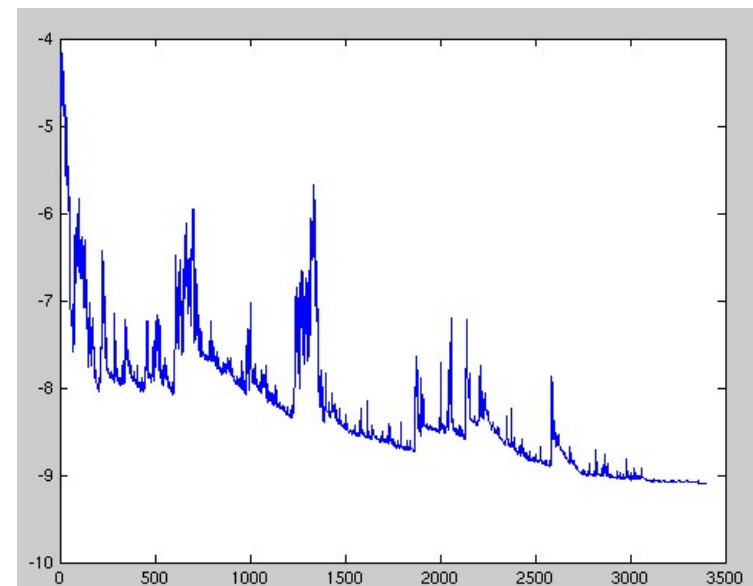
- This means that in expectation gradient descent will move towards minimum
- Typically, small subsets ("mini-batches") are used, e.g. 32, 64, 128, or 256 instances (while size of data set can be 1.000.000+ instances).
- Gradient steps orders of magnitude faster, can run many steps

# Stochastic Gradient Descent

- Trajectory to the minimum will be more noisy/random
- Loss on all of the data not guaranteed to fall at every SGD step, but will still fall on average



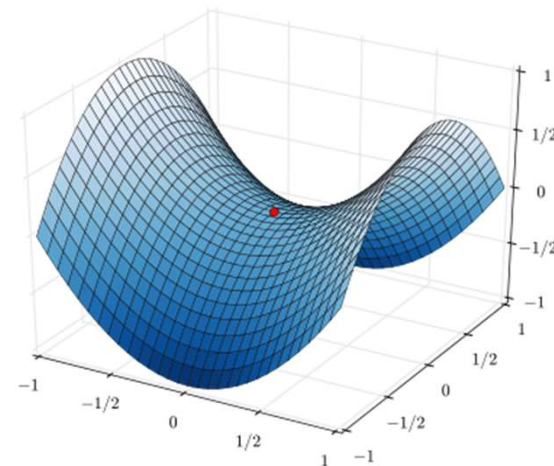
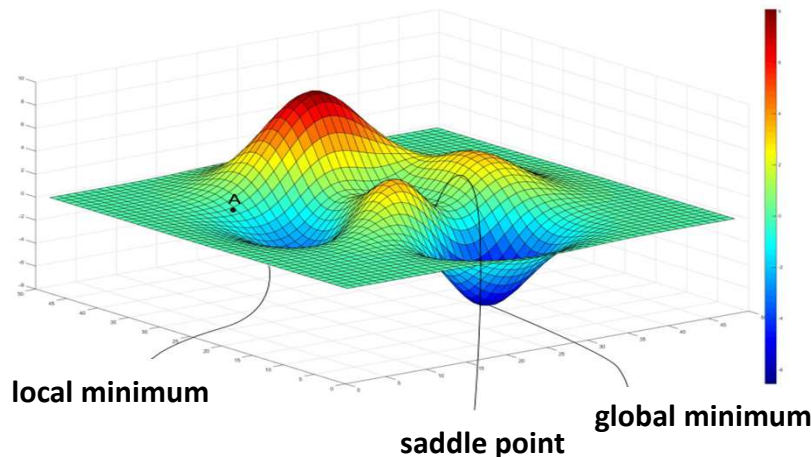
Path in parameter space SGD vs full gradient descent



Loss on complete data set over SGD iterations

# Local Minima and Saddle Points

- Gradient descent follows the (local) slope of the loss function
- What about local minima and saddle points?
  - local minimum: gradient converges towards a point that has lower loss than the immediate surroundings, but higher than the global minimum
  - saddle point: region where the gradient becomes zero and gradient descent can get stuck

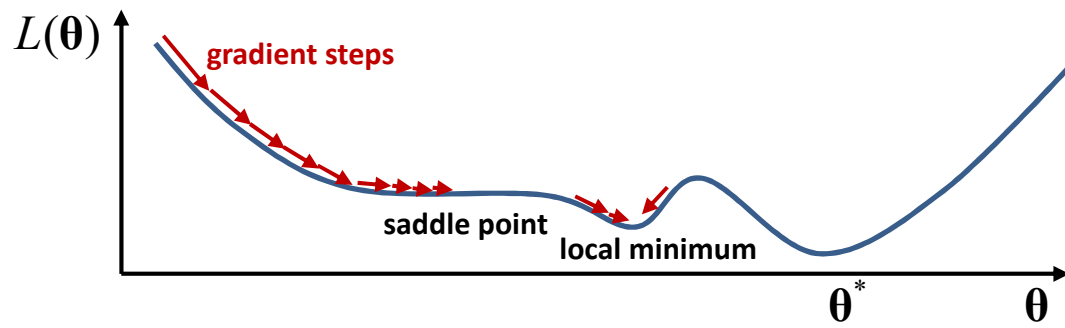


- Gradient descent will not generally find the global minimum.



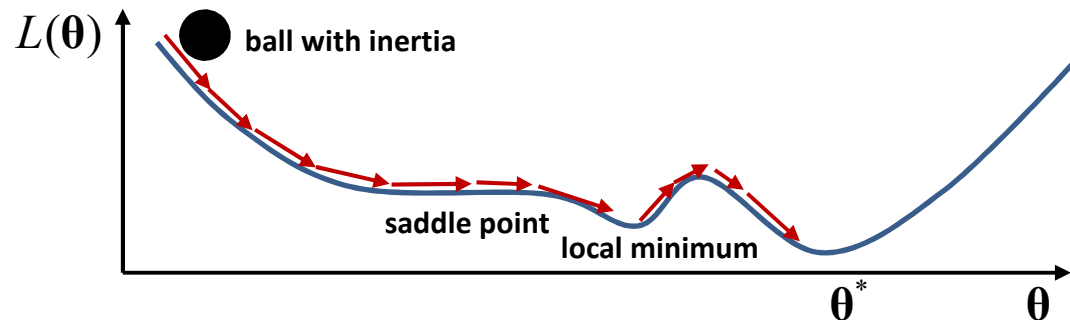
# Momentum in Gradient Descent

- One way of improving convergence of SGD is to use so-called **momentum**



## Without momentum:

step is local negative slope.  
Gets stuck at saddle point  
or in local minimum



## With momentum:

Imagine rolling a ball (with inertia) down the loss surface. Inertia carries the ball through saddle point and over local minimum

- Momentum in SGD: take a step into the direction of the local negative slope plus an exponentially decaying average of earlier steps

# Momentum in Gradient Descent

- Momentum in gradient descent: take a step into the direction of the local negative slope plus an exponentially decaying average of earlier steps
- Typically, use SGD with momentum

**Gradient descent  
with momentum**

1.  $\theta_0 = \text{randomInitialization}()$

2.  $v_0 = 0$

3. for  $i = 0, \dots, N$ :

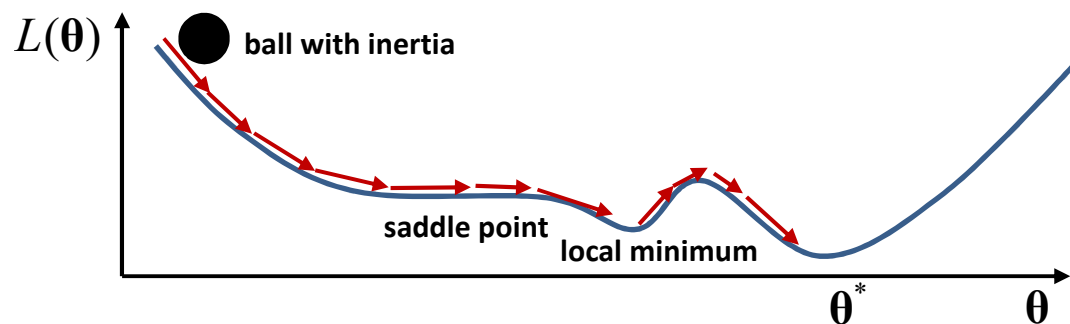
next update: negative slope  
plus discounted old update

$$v_{i+1} = -\eta \nabla L(\theta_i) + \gamma v_i$$

$$\theta_{i+1} = \theta_i + v_{i+1}$$

4. return  $\theta_N$

$\gamma \approx 0.9$



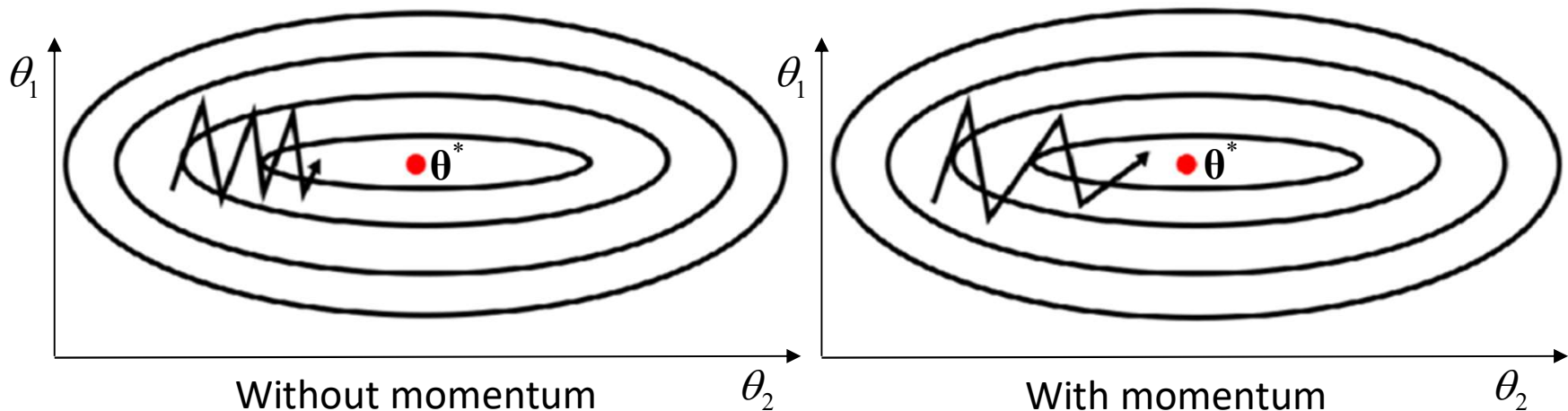
**With momentum:**

Imagine rolling a ball (with inertia) down the loss surface. Inertia carries the ball through saddle point and over local minimum

# Momentum in Gradient Descent

- Momentum also helps with convergence speed along long „valleys“:

Contour plot of loss function



Standard gradient will oscillate along  $\theta_1$  axis and only make slow progress along  $\theta_2$  axis.

Gradient with momentum builds up momentum along  $\theta_2$  axis over time

# Local Minima / Saddle Points in Practice

- Stochastic gradient descent helps: randomness in the individual steps can help the optimizer to „escape“ (small) local minima and saddle points
- Momentum helps
- More advanced optimization schemes based on SGD with momentum: AdaGrad, RMSProp, AdaDelta, Adam, ...
- Good initialization schemes are also important (how to set initial model parameters  $\theta_0$  before optimization starts)
- Empirical observation for training of deep neural networks: optimization does converge to different local optima, but all of them are similarly good (in terms of loss and also performance on test data)

# Optimization Summary

- Optimization problem: find model with lowest loss/objective

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

- For simple models, sometimes closed-form solution, e.g. linear regression

$$\mathbf{w}^* = (\mathbf{X}^T \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^T \mathbf{y}$$

- For large models such as deep neural networks almost always optimization based on stochastic gradient descent
- Not covered: **second order methods** (e.g. Newton method)
  - Use information from second derivative to speed up convergence
  - Widely used for some model classes, but not currently in deep learning