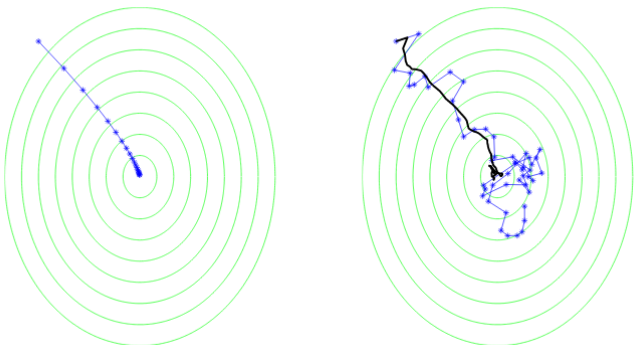
Deep Learning

Basic Training



Learning goals

- Gradient Descent
- Stochastic Gradient Descent
- Minibatch Gradient Descent
- Learning Rates and (S)GD specifics

TRAINING NEURAL NETWORKS

Training of neural nets is composed of two iterative steps:

1. **Forward pass:** The information of the inputs flows through the model to produce a prediction. Based on this prediction, the empirical risk is computed.

2. **Backward pass:** The information of the prediction error flows backward through the network to update the weights in a way that the error reduces.

Recall: The error is calculated via a loss function $L(y, f(\mathbf{x}, \theta))$, where y and $f(\mathbf{x}, \theta)$ are the true target and the network outcome, respectively.

TRAINING NEURAL NETWORKS

- For regression, the L2 loss is typically used:

$$L(y, f(\mathbf{x})) = \frac{1}{2}(y - f(\mathbf{x}))^2$$

- For binary classification, the binary cross entropy:

$$L(y, f(\mathbf{x})) = -(y \log f(\mathbf{x}) + (1 - y) \log(1 - f(\mathbf{x})))$$

Note: Evaluating the loss on the data, the **empirical risk function** is computed:

$$\mathcal{R}_{\text{emp}} = \frac{1}{n} \sum_{i=1}^{n} L\left(y^{(i)}, f\left(\mathbf{x}^{(i)}\right)\right)$$

- To minimize the empirical risk, the **gradient descent** (GD) method can be used.

GRADIENT DESCENT

- Let $\mathcal{R}_{\text{emp}} : \mathbb{R}^m \to \mathbb{R}$ be an arbitrary, differentiable, unrestricted function (of $\boldsymbol{\theta} \in \mathbb{R}^m$).

  In the context of deep learning, $\mathcal{R}_{\text{emp}}$ represents the empirical risk function and $\boldsymbol{\theta}$ represents the weights (and biases) of the network. For simplification, we assume $\boldsymbol{\theta} = (\theta_1, ..., \theta_m)$.
- We want to minimize this function by gradient descent (GD).
- The negative gradient

$$-\mathbf{g} = -\nabla \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = -\left(\frac{\partial \mathcal{R}_{\text{emp}}}{\partial \theta_1}, \ldots, \frac{\partial \mathcal{R}_{\text{emp}}}{\partial \theta_m} \right)^\top$$

  points in the direction of the **steepest descent**, since $\nabla \mathcal{R}_{\text{emp}}$ always points in the direction of the steepest ascent.

GRADIENT DESCENT

- "Standing" at a point $\boldsymbol{\theta}^{[t]}$ during minimization, we improve by performing the following update:

$$\boldsymbol{\theta}^{[t+1]} = \boldsymbol{\theta}^{[t]} - \alpha \nabla \mathcal{R}_{\text{emp}} \left(\boldsymbol{\theta}^{[t]} \right),$$

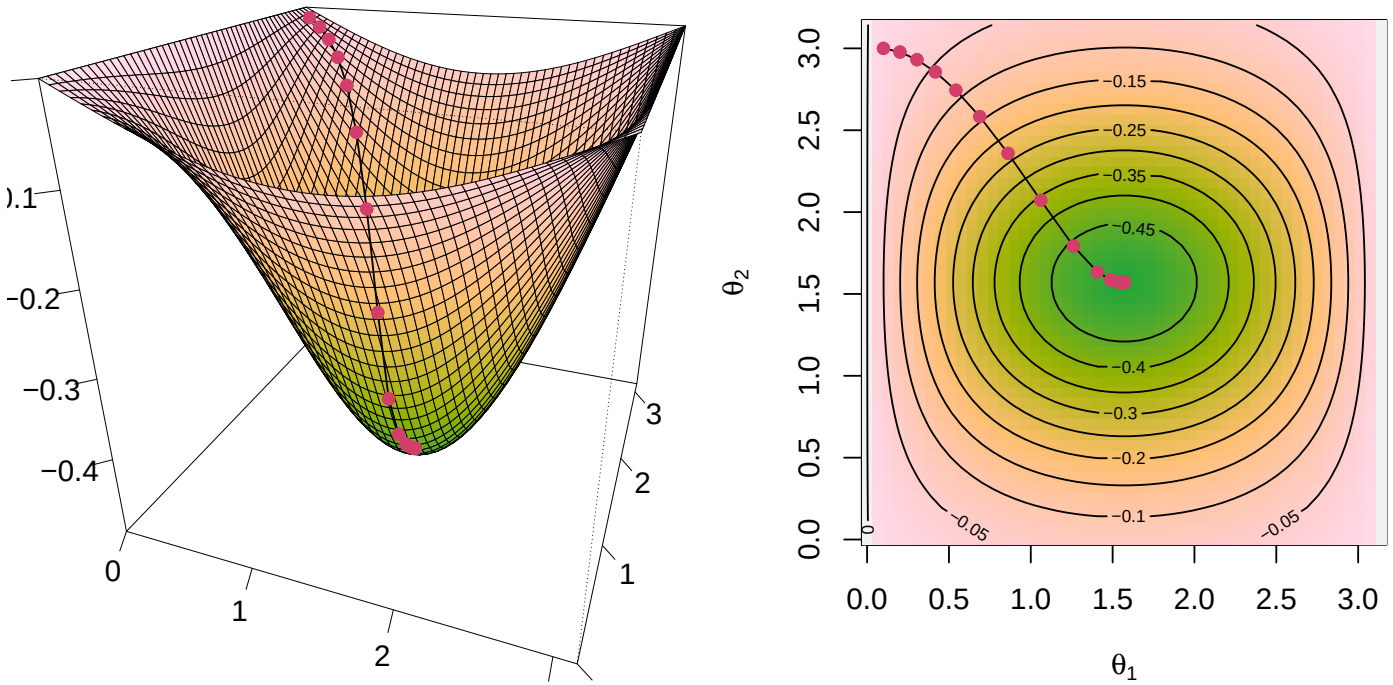
  which implies (for sufficiently small α),

$$\mathcal{R}_{\text{emp}}(\boldsymbol{\theta}^{[t+1]}) \leq \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}^{[t]})$$

- α determines the length of the step and is called **step size** or, in risk minimization, **learning rate**.

EXAMPLE: GRADIENT DESCENT

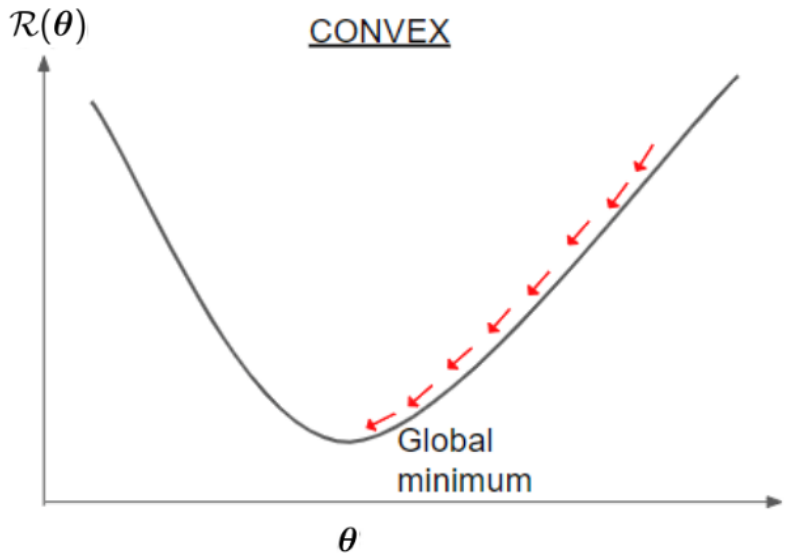
$$\mathcal{R}_{\text{emp}}(\theta_1, \theta_2) = -\sin(\theta_1) \cdot \frac{1}{2\pi} \exp\left((\theta_2 - \pi/2)^2\right)$$

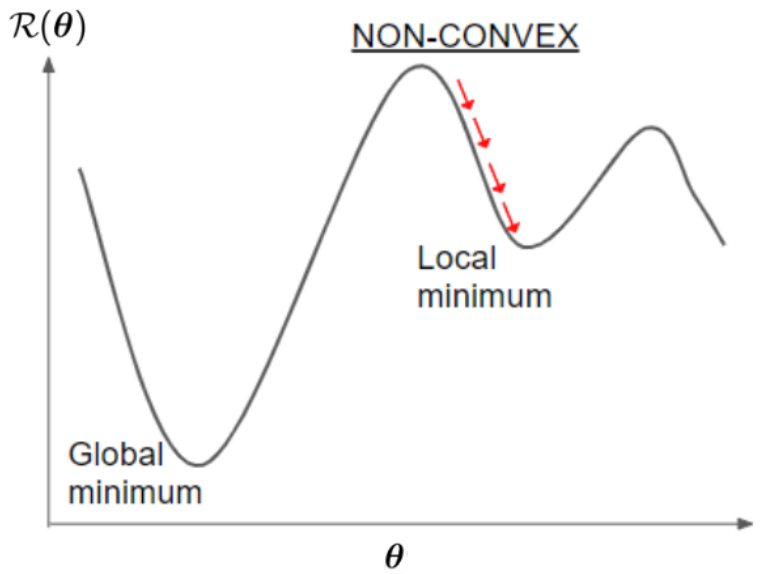


"Walking down the hill, towards the valley."

GRADIENT DESCENT AND OPTIMALITY

- GD is a greedy algorithm: In every iteration, it makes locally optimal moves.
- If $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$ is **convex** and **differentiable**, and its gradient is Lipschitz continuous, GD is guaranteed to converge to the global minimum (for small enough step-size).
- However, if $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$ has multiple local optima and/or saddle points, GD might only converge to a stationary point (other than the global optimum), depending on the starting point.





GRADIENT DESCENT AND OPTIMALITY

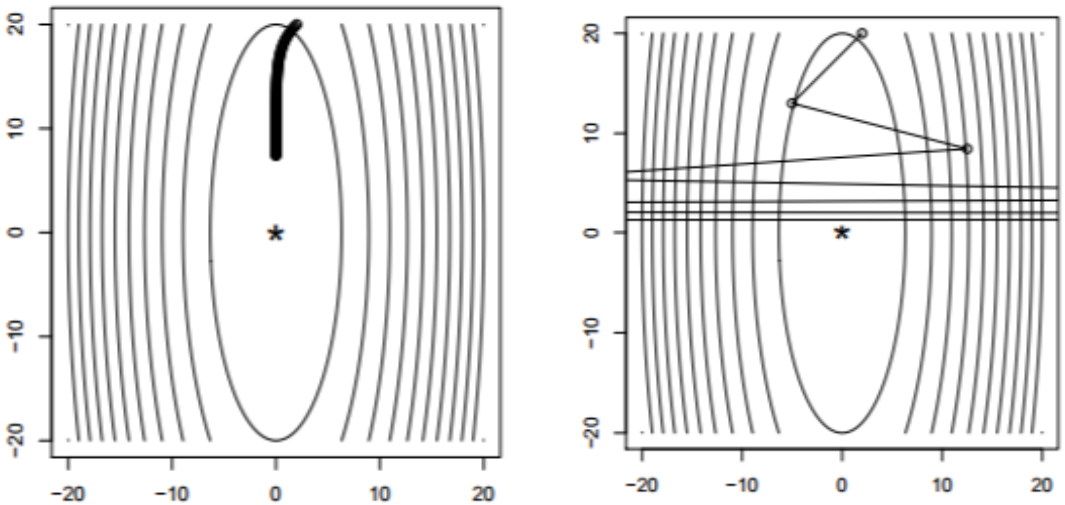
Note: It might not be that bad if we do not find the global optimum:

- We do not optimize the actual quantity of interest, i.e. the (theoretical) risk, but only an approximate version, i.e. the empirical risk.
- If the model class is very flexible, it might be disadvantageous to optimize too aggressively and increase the risk of overfitting.
- Early-stopping the optimization might even increase generalization performance.

LEARNING RATE

The step-size α plays a key role in the convergence of the algorithm.

If the step size is too small, the training process may converge **very** slowly (see left image). If the step size is too large, the process may not converge, because it **jumps** around the optimal point (see right image).

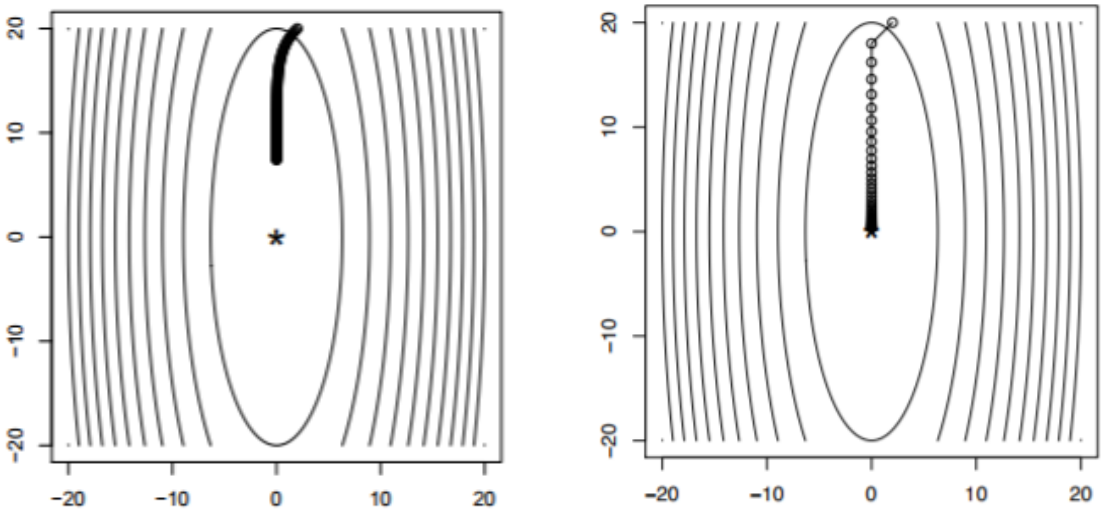


LEARNING RATE

So far we have assumed a fixed value of α in every iteration:

$$\alpha^{[t]} = \alpha \quad \forall t = \{1, \ldots, T\}$$

However, it makes sense to adapt α in every iteration:



Steps of gradient descent for $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = 10\,\theta_1^2 + 0.5\,\theta_2^2$. Left: 100 steps for with a fixed learning rate. Right: 40 steps with an adaptive learning rate.

WEIGHT INITIALIZATION

- The weights (and biases) of a neural network must be assigned some initial values before training can begin.
- It's important to initialize the weights randomly in order to "break symmetry". If two neurons (with the same activation function in a fully connected network) are connected to the same inputs and have the same initial weights, then both neurons will have the same gradient update in a given iteration and they'll end up learning the same features.
- Weights are typically drawn from a uniform a Gaussian distribution (both centered at 0 with a small variance).
- Two common initialization strategies are 'Glorot initialization' and 'He initialization' which tune the variance of these distributions based on the topology of the network.

STOCHASTIC GRADIENT DESCENT

Let us consider GD for empirical risk minimization. The updates are:

$$\boldsymbol{\theta}^{[t+1]} = \boldsymbol{\theta}^{[t]} - \alpha \cdot \frac{1}{n} \cdot \sum_{i=1}^{n} \nabla_{\theta} L \left(y^{(i)}, f(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}^{[t]}) \right)$$

- Optimization algorithms that use the entire training set to compute updates in one huge step are called **batch** or **deterministic**. This is computationally very costly or often impossible.
- **Idea:** Instead of letting the sum run over the whole dataset (**batch mode**) one can also let it run only over small subsets (**minibatches**), or only over a single example i.
- If the index i of the training example is a random variable with uniform distribution, then its expectation is the batch gradient $\nabla_{\theta} \mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$

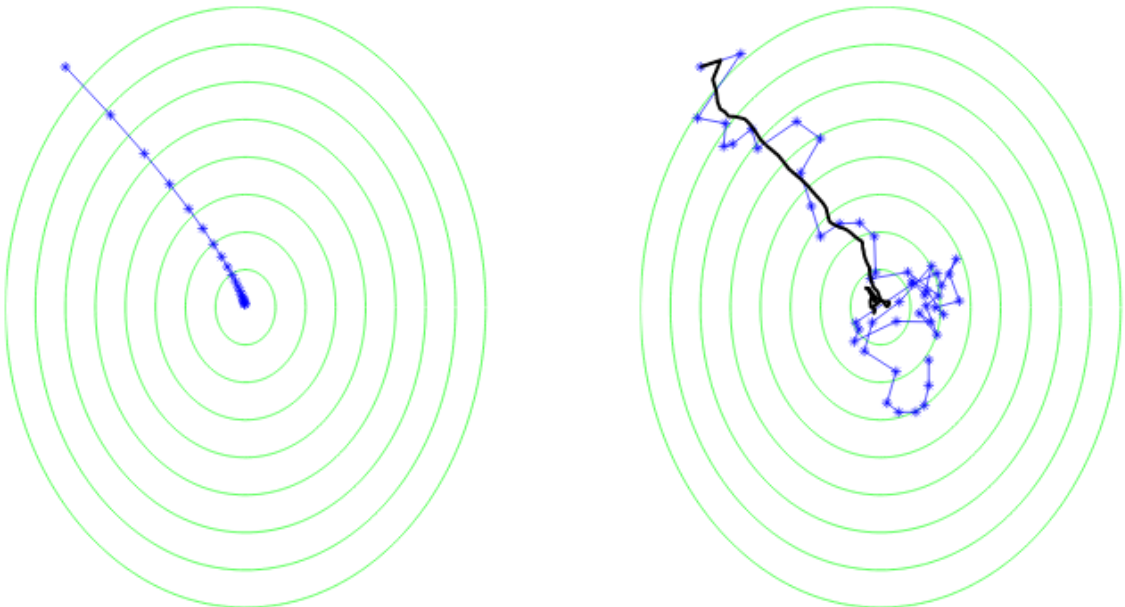
$\rightarrow$ We have a **stochastic**, noisy version of the batch gradient

STOCHASTIC GRADIENT DESCENT

- The gradient w.r.t. a single training observation is fast to compute but not reliable. It can be used simply as a computational trick to deal with large data or to operate on real streams of online data in online learning.
- In contrast, the full batch gradient is costly (or even impossible, e.g., when data does not even fit into memory) to compute, particularly in DL, but it averages out all the noise from sub-sampling.
- Minibatches are in between. The batch size decides upon the compromise between speed and averaging (smoothing).
- In summary: SGD computes an unbiased estimate of the gradient by taking the average gradient over a minibatch (or one sample) to update the parameter θ in this direction.

STOCHASTIC GRADIENT DESCENT

An illustration of the SGD algorithm (to minimize the function $1.25(x_1 + 6)^2 + (x_2 - 8)^2$).

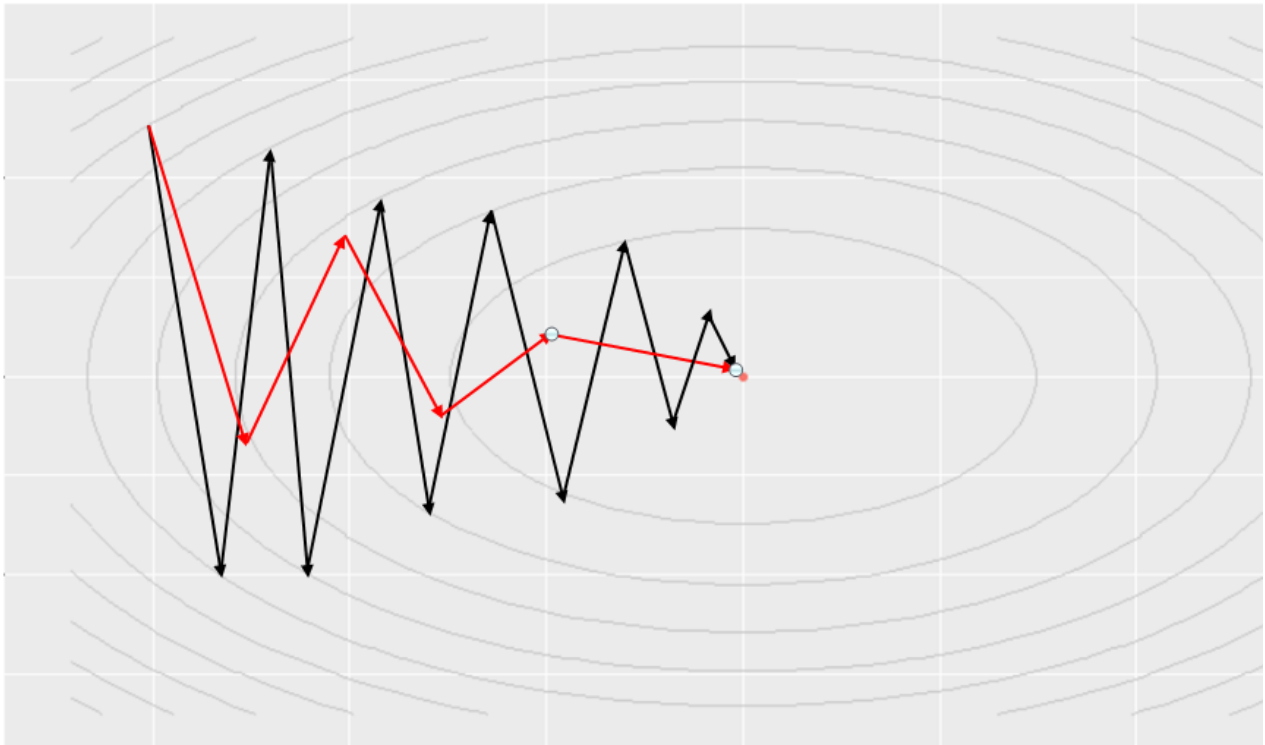


Source : Shalev-Shwartz and Ben-David. Understanding machine learning: From theory to algorithms. Cambridge University Press, 2014.

Figure: On the left is GD and on the right is SGD. The black line depicts the averaged value of θ.

SGD WITH MOMENTUM

- While SGD remains a popular optimization strategy, learning with it can sometimes be slow.
- Momentum is designed to accelerate learning, by accumulating an exponentially decaying moving average of past gradients.



GD (black) versus momentum (red) when dealing with ravines

Algorithm Basic SGD pseudo code

Initialize parameter vector $\boldsymbol{\theta}^{[0]}$
$t \leftarrow 0$
while stopping criterion not met **do**
    Randomly shuffle data and partition into minibatches $J_1, \ldots, J_K$ of size m
    for $k \in \{1, \ldots, K\}$ **do**
        $t \leftarrow t + 1$
        Compute gradient estimate with J_k: $\hat{g}^{[t]} \leftarrow \frac{1}{m} \sum_{i \in J_k} \nabla_\theta L(y^{(i)}, f(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}^{[t-1]}))$
        Apply update: $\boldsymbol{\theta}^{[t]} \leftarrow \boldsymbol{\theta}^{[t-1]} - \alpha \hat{g}^{[t]}$
    end for
end while

- With minibatches of size m, a full pass over the training set (called an **epoch**) consists of $\frac{n}{m}$ gradient updates (**iterations**).
- SGD and its modifications are the most used optimization algorithms for ML in general and for deep learning in particular.
- SGD (with one or a few samples per batch) updates have a high variance, even though they are unbiased. Because of this variance, the learning rate α is typically much smaller than in the full-batch scenario.
- SGD with minibatches reduces the variance of the parameter updates and utilizes highly optimized matrix operations to efficiently compute gradients.

Note: Sometimes, SGD with minibatches is referred to as "Minibatch gradient descent" with the term SGD being reserved for updates using a single example.

- Minibatch sizes are typically between 50 and 1000.
- When the learning rate is slowly decreased, SGD converges to a local minimum.
- Recent results indicate, that SGD often leads to better generalizing models then GD, and thus may perform some kind of indirect regularization.
- Now we know how we can optimize/train a neural network based on the gradient, but how do we compute the gradient? We will learn more about this in the chapter on backpropagation, a method to efficiently compute gradients.