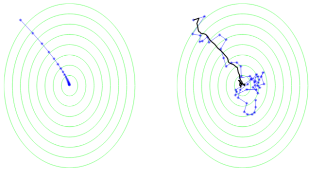
Deep Learning

Basic Training



Learning goals

- Gradient Descent
- Mini-batch Gradient Descent
- Stochastic 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:

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

- For binary classification, the binary cross entropy:

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

Note: Evaluating the loss on the data, the **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 risk, the **gradient descent** (GD) method can be used.

GRADIENT DESCENT

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

  In the context of deep learning, $\mathcal{R}$ 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

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

  points in the direction of the **steepest descent** since $\nabla\mathcal{R}$ 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:

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

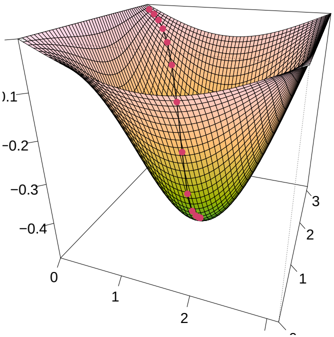
  which implies (for sufficiently small α),

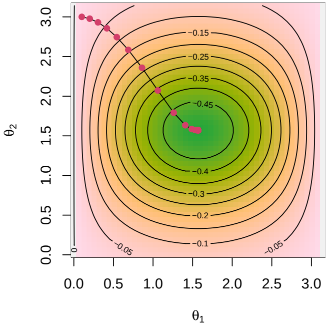
$$\mathcal{R}(\boldsymbol{\theta}^{[t+1]}) \leq \mathcal{R}(\boldsymbol{\theta}^{[t]})$$

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

EXAMPLE: GRADIENT DESCENT

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

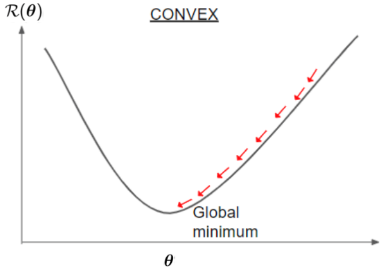


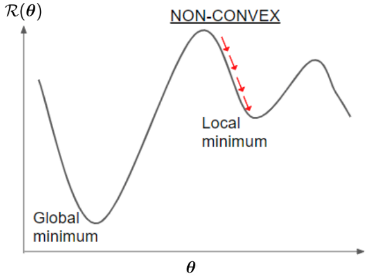


"Walking down the hill, towards the valley."

GRADIENT DESCENT AND OPTIMALITY

- GD is a greedy algorithm: In every iteration, it makes locally optimal moves.
- If $\mathcal{R}(\boldsymbol{\theta})$ 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 $\mathcal{R}(\boldsymbol{\theta})$ 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

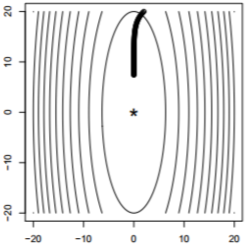
Note: It might not be that bad if we do not find the global optimum:

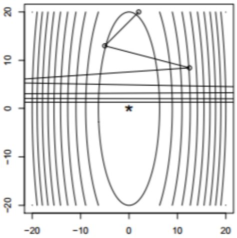
- We do not optimize the actual quantity of interest, i.e. the (theoretical) risk, but only an approximate version, i.e. the empirical risk.
- If the model class is very flexible, it might be disadvantageous to optimize too aggressively and increase the risk of overfitting.
- Early-stopping the optimization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



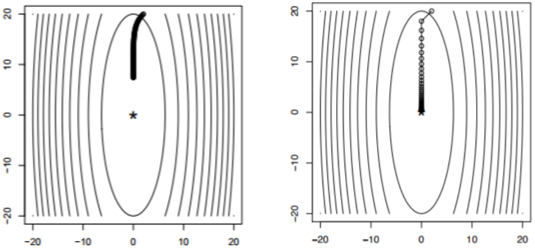


LEARNING RATE

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

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

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



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

WEIGHT UPDATES WITH BACKPROPAGATION

- To update each weight $w \in \boldsymbol{\theta}$ in the network, we need their gradients with regards to the risk.

- Since weights are stacked in layers inside the network, we need to repeatedly apply the “chain rule of calculus”. This process is called **backpropagation**.

- After obtaining the gradients, the weights can be updated by GD:

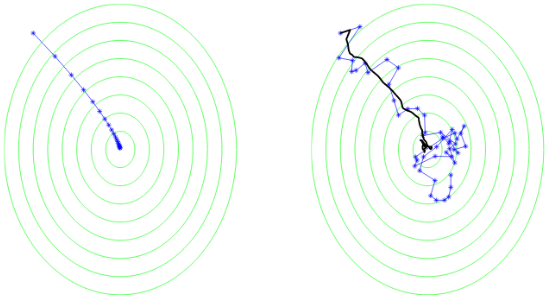
$$\boldsymbol{\theta}^{[t+1]} = \boldsymbol{\theta}^{[t]} - \alpha \cdot \frac{1}{n} \cdot \sum_{i=1}^{n} \nabla_{\theta} L\left(y^{(i)}, f\big(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}^{[t]}\big)\right)$$

STOCHASTIC GRADIENT DESCENT

- 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.
- Instead of running the sum over the whole dataset (**batch mode**), one can run over small subsets (**minibatches**) of size m.
- With minibatches of size m, a full pass over the training set (called an **epoch**) consists of $\frac{n}{m}$ gradient updates.
- This stochastic version of the batch gradient is known as **Stochastic Gradient Descent** (SGD).

STOCHASTIC GRADIENT DESCENT

An illustration of the SGD algorithm: on the left is GD and on the right is SGD (to minimize the function $1.25(x_1 + 6)^2 + (x_2 - 8)^2$). The black line depicts the averaged value of $\boldsymbol{\theta}$.



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

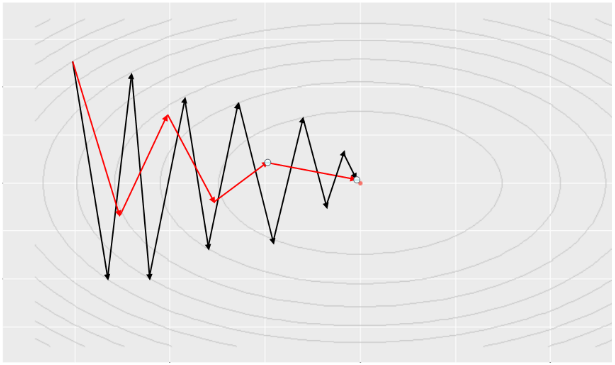
STOCHASTIC GRADIENT DESCENT

Algorithm Basic SGD pseudo code

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

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

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.