Lecture 3

Christa Cuchiero
based on joint lectures with M. Gambara, J. Teichmann and H. Wutte

Institute of Statistics and Mathematics
Vienna University of Economics and Business

Part I

Gradient descent and backpropagation

Supervised learning task with neural networks

Supervised learning

Given training data $\{(x_i, y_i), i = 1, \dots N\}$ with $x_i \in \mathbb{R}^m$ and $y_i \in \mathbb{R}^d$, find a neural network g within a class of neural networks NN_Θ with a certain architecture characterized by parameters $\theta \in \Theta$, such that

$$g \in \underset{NN_\Theta}{\operatorname{argmin}} \sum_{i=1}^{N} \mathcal{L}(g(x_i), y_i),$$

where $\mathcal{L}$ is a loss function: $C(\mathbb{R}^m, \mathbb{R}^{\widetilde{d}}) \times \mathbb{R}^d \to \mathbb{R}_+$. Note that the input dimension of the neural network is m and the output dimension $\widetilde{d}$.

Since g is determined by the parameters θ, the above optimization corresponds to searching the minimum in the parameter space Θ which is nothing else than the collection of $(A_t, b_t)_{t=1,\dots,n}$ (if we have n hidden layers) and the readout map R.

Examples

- Example MNIST classification: $x_i \in \mathbb{R}^{28\times 28}$, i.e. $m = 28 \times 28$ and $y \in \mathbb{R}$, i.e. $d = 1$. The output dimension of the neural network is $\widetilde{d} = 10$. The loss function is given by

$$\mathcal{L}(g(x), y) = \sum_{k=1}^{10} 1_{\{y=k-1\}} \log((g(x))_k).$$

- Example classical regression with L^2 loss:

$$\mathcal{L}(g(x), y) = \|g(x) - y\|^2.$$

Here the output dimension of the neural network $\widetilde{d}$ is equal to d.

But how...?

- ... to deal with a non-linear, non-convex optimization problem and with around 600 000 parameters, as it is the case for the MNIST data set?

Gradient descent: the simplest method

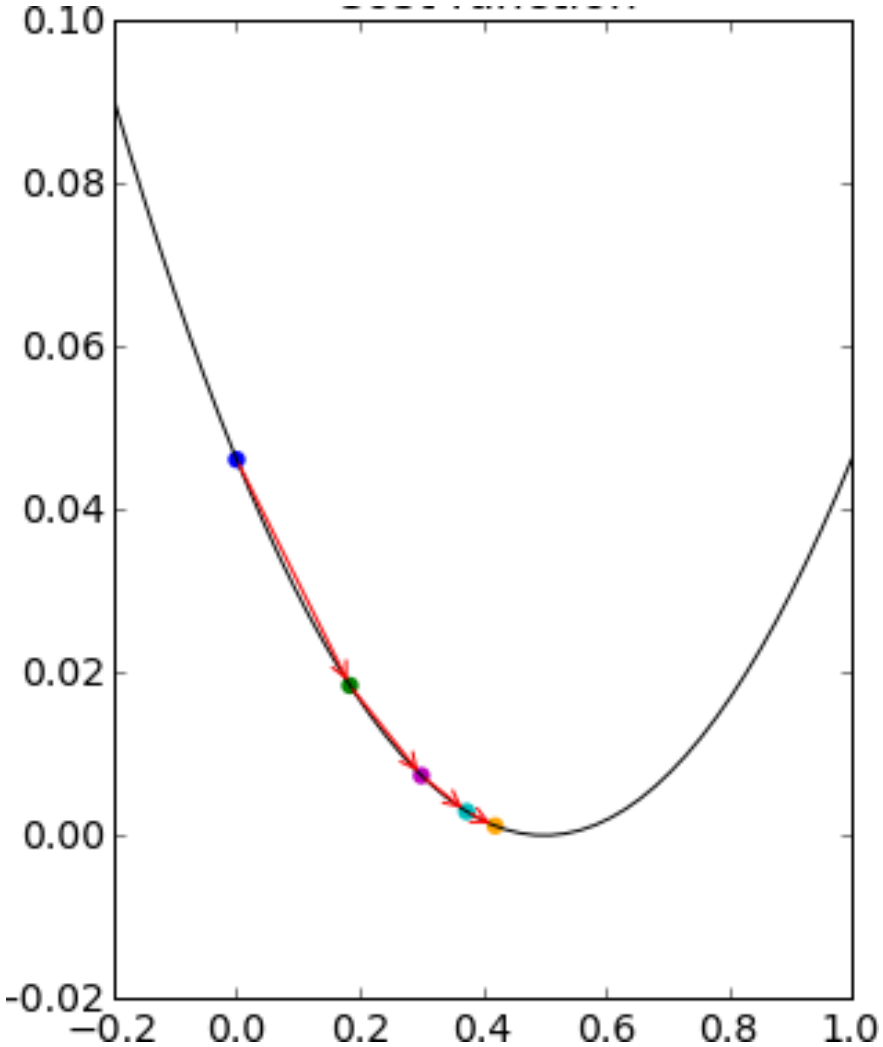
- The gradient of a function $F(\theta) : \mathbb{R}^M \to \mathbb{R}$ is given by

$$\nabla F(\theta) = (\partial_{\theta_1} F(\theta), \ldots, \partial_{\theta_M} F(\theta)).$$

- Gradient descent:
  starting with an initial guess $\theta^{(0)}$, one iteratively defines for some learning rate η_k

$$\theta^{(k+1)} = \theta^{(k)} - \eta_k \nabla F(\theta^{(k)})$$

Gradient descent: the simplest method



Classical convergence result

Theorem

Suppose the function $F : \mathbb{R}^M \to \mathbb{R}$ *is convex and differentiable, and that its gradient is Lipschitz continuous with constant* $L > 0$, *i.e. we have that* $\|\nabla F(\theta) - \nabla F(\beta)\| \leq L\|\theta - \beta\|$ *for any* $\theta, \beta \in \mathbb{R}^M$. *Then if we run gradient descent for* k *iterations with a fixed step size* $\eta \leq 1/L$, *it will yield a solution* $F(\theta^{(k)})$ *which satisfies*

$$F(\theta^{(k)}) - F(\theta^*) \leq \frac{\|\theta^{(0)} - \theta^*\|^2}{2\eta k},$$

where $F(\theta^*)$ *is the optimal value. Intuitively, this means that gradient descent is guaranteed to converge and that it converges with rate* $O(1/k)$.

In practice, the convexity condition is often not satisfied. Moreover, the solution depends crucially on the inital value.

How to compute the gradient

- Nevertheless all optimization algorithms build on the classical idea of gradient descent usually in its enhanced form of stochastic gradient descent.
- How to compute the gradient in our case of supervised learning, where

$$F(\theta) = \sum_{i=1}^{N} \mathcal{L}(g(x_i|\theta), y_i)$$

and θ corresponds to $(A_t, b_t)_{t=1,\dots,n}$ (if we have n hidden layers) and the readout map R? We here indicate the dependence of the neural network on θ.

- We suppose here for simplicity that the readout map R is linear, i.e.

$$R(x) = A_{n+1}x + b$$

where A_{n+1} has $\widetilde{d}$ rows and $b \in \mathbb{R}^{\widetilde{d}}$, so that $\theta = \{(A_t, b_t)_{t=1,\dots,n+1}\}$.

Backpropagation

- Since

$$\nabla_\theta F(\theta) = \sum_{i=1}^{N} \nabla_\theta \mathcal{L}(g(x_i|\theta), y_i),$$

we need to determine $\partial_{A_t,kl}\mathcal{L}(g(x|\theta), y)$ and $\partial_{b_t,k}\mathcal{L}(g(x|\theta), y)$.

- By the chain rule this is given by

$$\partial_{A_t,kl}\mathcal{L}(g(x|\theta), y) = \langle \partial_g \mathcal{L}(g(x|\theta), y), \partial_{A_t,kl} g(x|\theta) \rangle$$
$$\partial_{b_t,k}\mathcal{L}(g(x|\theta), y) = \langle \partial_g \mathcal{L}(g(x|\theta), y), \partial_{b_t,k} g(x|\theta) \rangle.$$

- Output Layer:

$$\partial_{A_{n+1},kl}\mathcal{L}(g(x|\theta), y) = (\partial_g \mathcal{L}(g(x|\theta), y))_k (\underbrace{\sigma(A_n x(n-1) + b_n)}_{x(n)})_l$$

$$\partial_{b_{n+1},k}\mathcal{L}(g(x|\theta), y) = (\partial_g \mathcal{L}(g(x|\theta), y))_k$$

Backpropagation: second last layer

- Recall $x(t+1) = \sigma(z_{t+1})$ where $z_{t+1} = A_{t+1}x(t) + b_{t+1}$ and $g = z_{n+1} = A_{n+1}x(n) + b_{n+1}$.
- To continue with the second last layer, we use the chain rule again

  Note that $\mathcal{L}(g, y) = \mathcal{L}(z_{n+1}, y) = \mathcal{L}(A_{n+1}x(n) + b_n, y)$. Hence...

$$\begin{aligned}\partial_{A_n,kl}\mathcal{L} &= \langle \partial_{x(n)}\mathcal{L}, \partial_{A_n,kl}x(n)\rangle = \langle A_{n+1}\partial_g\mathcal{L}, \partial_{A_n,kl}x(n)\rangle \\ &= \langle A_{n+1}\partial_g\mathcal{L}, \mathrm{diag}(\sigma'(z_n)) \underbrace{\partial_{A_n,kl}z_n}_{\text{similar as in the last layer}} \rangle\end{aligned}$$

$$\begin{aligned}\partial_{b_n,k}\mathcal{L} &= \langle \partial_{x(n)}\mathcal{L}, \partial_{b_n,k}x(n)\rangle = \langle A_{n+1}\partial_g\mathcal{L}, \partial_{b_n,k}x(n)\rangle \\ &= \langle A_{n+1}\partial_g\mathcal{L}, \mathrm{diag}(\sigma'(z_n)) \underbrace{\partial_{b_n,k}z_n}_{\text{similar as in the last layer}} \rangle\end{aligned}$$

Backpropagation - matrix notation

- Output Layer:

$$\partial_{A_{n+1}} \mathcal{L} = \underbrace{\partial_g \mathcal{L}}_{\delta_{n+1}} (\underbrace{\sigma(A_n(\cdots) + b_n)}_{x(n)})^T$$

$$\partial_{b_{n+1}} \mathcal{L} = \partial_g \mathcal{L}$$

- All other layers:

$$\partial_{A_t} \mathcal{L} = \underbrace{\text{diag}(\sigma'(z_t)) A_{t+1} \partial_g \mathcal{L}}_{\delta_t} (\underbrace{\sigma(A_{t-1} x(t-2) + b_{t-1})}_{x(t-1)})^T$$

$$\partial_{b_t} \mathcal{L} = \text{diag}(\sigma'(z_t)) A_{t+1} \partial_g \mathcal{L}$$

The Backpropogation Algorithm

1. Calculate $z_t, x(t)$ for $t = 1, \ldots, n+1$ (forward pass)
2. Set $\delta_{n+1} = \partial_g \mathcal{L}$
3. Then $\partial_{A_{n+1}} \mathcal{L} = \delta_{n+1} x(n)$ and $\partial_{b_{n+1}} \mathcal{L} = \delta_{n+1}$
4. **for** t from n to 1 **do:**
   - $\delta_t = \text{diag}(\sigma'(z_t)) A_{t+1} \partial_g \mathcal{L}$
   - Then $\partial_{A_t} \mathcal{L} = \delta_t x(t-1)$ and $\partial_{b_t} \mathcal{L} = \delta_t$
5. **return** $\partial_{A_t} \mathcal{L}$ and $\partial_{b_t} \mathcal{L}$ for $t = 1, \ldots, n+1$

Part II

Stochastic gradient descent

Complexity of the standard gradient descent

- Recall that one gradient descent step requires the calculation of

$$\nabla_\theta F(\theta) = \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \mathcal{L}(g(x_i|\theta), y_i).$$

  and each of the summands requires one backpropagation run. (We normalize the loss function by N). Thus, the total complexity of one gradient descent step is equal to

$$N * \text{complexity(backprop)}$$

- The complexity of one backpropagation run corresponds to the number of parameters of the neural network

$$\sum_{t=1}^{n}(m_t \times m_{t+1} + b_t).$$

- In the case of the standard algorithm for the MNIST data set that would be $600000 * 60000 = 36 * 10^9$ flops (Floating Point Operations Per Seconds) and memory units!

Stochastic gradient descent

- Key insight for deep learning: stochastic gradient descent
- $\nabla_\theta F$ is approximated/replaced by the gradient of a subsample: starting with an initial guess $\theta^{(0)}$, one iteratively defines for some learning rate η_k

$$\theta^{(k+1)} = \theta^{(k)} - \eta_k \nabla \mathcal{L}^{(k)}(\theta^{(k)})$$

with

$$\mathcal{L}^{(k)}(\theta) = \frac{1}{N_{\text{batch}}} \sum_{i=1}^{N_{\text{batch}}} \mathcal{L}(g(x_{i+kN_{\text{batch}}}|\theta), y_{i+kN_{\text{batch}}})$$

$$k \in \{0, 1, ..., \lfloor N/N_{\text{batch}}, \rfloor - 1\}$$

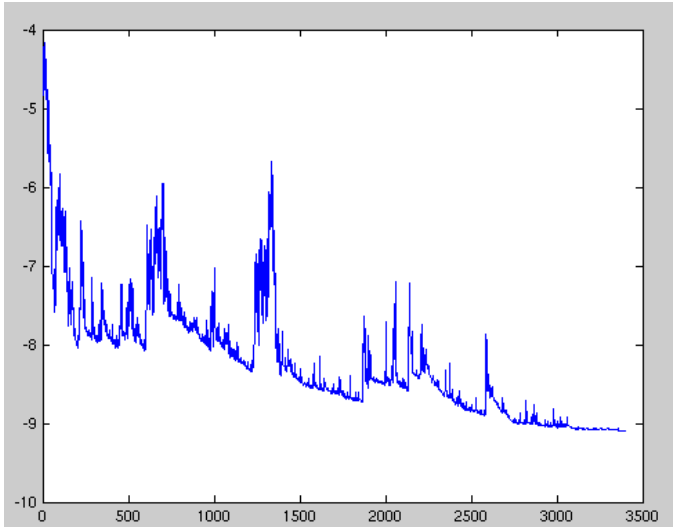
meaning that the training data is batched in packages (mini batches) of size N_{batch}.

- In the most extrem case N_{batch} can be equal to 1. Then the true gradient of F is approximated by a gradient of a single (randomly chosen) sample k, i.e. $\nabla \mathcal{L}((g(x_k|\theta), y_k)$. The algorithm then sweeps through the training set to perform the above update of θ.

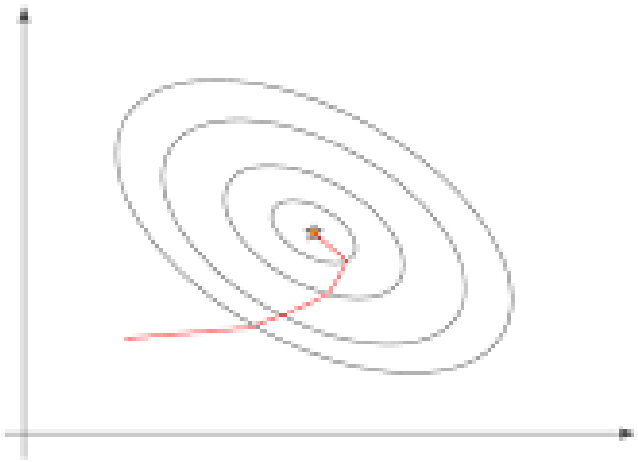
Stochastic gradient descent - algorithm

- Choose an initial vector of parameters θ and learning rate η
- Repeat until an approximate minimum is obtained:
  - Randomly shuffle examples in the training set.
  - **for** $k = 1, 2, \ldots, N$ **do:**
    - ★ $\theta^{(k)} := \theta^{(k-1)} - \eta \nabla \mathcal{L}((g(x_k|\theta^{k-1}), y_k)$

The following picture illustrated the fluctuations in the loss function as gradient steps with respect to mini-batches are taken.



"Comparison between SGD and GD"



Gradient Descent

Stochastic Gradient Descent

Stochastic approximations - Robbins-Monro algorithm

- Stochastic gradient descent can be traced back to the Robbins-Monro algorithm (1951).
- This is a methodology for solving a root finding problem, where the function is represented as an expected value.
- Assume that we have a function $M(\theta)$ and a constant α , such that the equation $M(\theta) = \alpha$ has a unique root at θ^*, i.e. $M(\theta^*) - \alpha = 0$.
- It is assumed $M(\theta) = \mathbb{E}[N(\theta)]$.
- The structure of the algorithm is to then generate iterates of the form:

$$\theta^{(k+1)} = \theta^{(k)} - \eta_k(N(\theta^{(k)}) - \alpha)$$

A convergence result

Theorem (Robbins-Monoroe '51, Blum '54)

Assume that

- *the random variable $N(\theta)$ is uniformly bounded,*
- *$M(\theta)$ is nondecreasing,*
- *$M'(\theta^*)$ exists and is positive,*
- *the sequence η_k satisfies the following requirements*

$$\sum_{k=0}^{\infty} \eta_k = \infty \quad \text{and} \quad \sum_{k=0}^{\infty} \eta_k^2 < \infty.$$

Then $\theta^{(k)}$ converges in L^2 and almost surely to θ^*.

Application in Stochastic Optimization

- Suppose we want to solve the following stochastic optimization problem

$$\min_{\theta \in \Theta} \mathbb{E}[Q(\theta)]$$

with a stochastic objective function $Q : \Omega \times \Theta \to \mathbb{R}$, $(\omega, \theta) \mapsto Q(\theta)(\omega)$.

- If $\theta \mapsto \mathbb{E}[Q(\theta)]$ is differentiable and convex, then this problem is equivalent to find the root θ^* of $\nabla \mathbb{E}[Q(\theta)] = 0$.
- We can apply the Robbins Monro algorithm whenever we find $N(\theta) = N(\theta)(\omega)$ such that

$$\nabla \mathbb{E}[Q(\theta)] = \mathbb{E}[N(\theta)],$$

i.e., $N(\theta)$ needs to be an unbiased estimator of $\nabla \mathbb{E}[Q(\theta)]$. If we can interchange differentiation and expectation,

$$N(\theta) = \nabla Q(\theta)$$

is clearly a candidate.

Convergence and application to supervised learning

- The above first three conditions for convergence translate to strict convexity of $\theta \mapsto \mathbb{E}[Q(\theta)]$ and uniform boundedness of $\nabla Q(\theta)$.
- In the case of supervised learning we deal exactly with such problems. The stochastic objective function $Q(\theta)(\omega)$ then corresponds to

$$\mathcal{L}(g(x|\theta), y)$$

where (x, y) corresponds to ω and we deal with the empirical measure to compute the mean.