Chapter 2

Gradient based optimization

2.1 Gradient descent

Once our neural architecture f_θ and loss $\mathcal{L}_N(\cdot)$ are defined we are ready to attack the following optimization problem

$$\min_{\theta \in \mathbb{R}^D} \mathcal{L}_N(\theta)$$

where D denotes here the number of learnable parameters in our neural network. So far, we used the notation $\mathcal{L}_N$ to stress that the loss function does not only depend from the network parameters θ but also from the training data points $x_1, \ldots, x_N$ and labels $y_1, \ldots, y_N$. However, to keep the notation uncluttered, we omit the subscript N from now on.

Now, for a given θ^* we look at the following second order Taylor expansion of $\mathcal{L}(\cdot)$ around θ^*

$$\begin{aligned}\mathcal{L}(\theta) &= \mathcal{L}(\theta^*) + (\nabla\mathcal{L}(\theta^*))^T(\theta - \theta^*) + \frac{1}{2}(\theta - \theta^*)^T H\mathcal{L}(\theta^*)(\theta - \theta^*) + o(||\theta - \theta^*||^2) \\ &\approx \mathcal{L}(\theta^*) + (\nabla\mathcal{L}(\theta^*))^T(\theta - \theta^*) + \frac{1}{2}(\theta - \theta^*)^T H\mathcal{L}(\theta^*)(\theta - \theta^*),\end{aligned} \tag{2.1}$$

where the approximation is of course reasonable in a neighbourhood of θ^* and $\nabla\mathcal{L}(\cdot)$ (respectively $H\mathcal{L}(\cdot)$) is the gradient (the Heissian matrix) of $\mathcal{L}$ with respect to θ. If we seek to minimize the quadratic approximation of $\mathcal{L}$, on the left hand side of the above equality, we see that it can be done directly

by setting its gradient w.r.t. θ equal to zero:

$$\nabla\mathcal{L}(\theta^*) + H\mathcal{L}(\theta^*)(\theta - \theta^*) = 0 \tag{2.2}$$

which leads to the Newton's update

$$\theta^{**} := \theta^* - (H\mathcal{L}(\theta^*))^{-1}\nabla\mathcal{L}(\theta^*), \tag{2.3}$$

which clearly makes sense assuming that the Hessian is invertible. Then the idea would be approximate again $\mathcal{L}$ around $\theta **$ and repeat a Newton step and so on until a stationary point is reached. Indeed, if θ^* is a stationary point for $\mathcal{L}$, then $\nabla\mathcal{L}(\theta^*) = 0$ and the above equation reduce to $\theta^{**} = \theta^*$. Moreover, in that case the quadratic form on the right hand side of Eq. (2.1) reduces to

$$\mathcal{L}(\theta^*) + \frac{1}{2}(\theta - \theta^*)^T H\mathcal{L}(\theta^*)(\theta - \theta^*).$$

If we denote by $H\mathcal{L}(\theta^*) = Q\Lambda Q^T$ the spectral decomposition of the Heissian matrix at θ^* we can adopt the following change of variables $\alpha := Q^T(\theta - \theta^*)$ in order to rewrite Eq. (2.1) as

$$\mathcal{L}(\theta) \approx \mathcal{L}(\theta^*) + \sum_{j=1}^{D} \alpha_j^2 \lambda_j$$

where $\lambda_1, \ldots, \lambda_D$ denote here the eigenvalues of the Heissian matrix in decreasing order.

Exercise 2.1 Use the above equation (treat $\approx$ as en equality) to show that θ^* is a local minimum if and only if $H\mathcal{L}(\theta^*)$ is positive (semi-)definite.

Now although Eq. (2.3) is likely to converge rapidly (the Newton method converges at least quadratically in the number of iterations), unfortunately it is not adapted to deep learning models, where D is usually very high (often higher than N), without mentioning the fact that we need to **invert** the Heissian matrix. This is why what we commonly see the following variant of Eq. (2.3)

$$\theta^{**} := \theta^* - \eta\nabla\mathcal{L}(\theta^*) \tag{2.4}$$

where η is a user fixed positive learning rate. What can we say about η? Intuitively higher values of η should lead to a faster convergence but things

are definitely more complicated. Assume the current value of θ is θ_1, in a neighborhood of the local minimum θ^*. Then, the following update of θ is

$$\theta_2 = \theta_1 - \eta \nabla \mathcal{L}(\theta_1).$$

Relying on Eq. (2.2) and recalling that θ^* is a local minimum, we have $\nabla \mathcal{L}(\theta_1) \approx Q\Lambda\alpha_1$. Plugging in into the above equation and subtracting θ^* from both sides of the equality we have that $Q\alpha_2 = Q\alpha_1 - \eta Q\Lambda\alpha_1$ which in turn leads to $\alpha_2 = (I_D - \eta\Lambda)\alpha_1$. After U steps we have

$$\alpha_U = (I_D - \eta\Lambda)^U \alpha_1,$$

which converges to zero (hence $\theta_\infty = \theta^*$) as long as $|1 - \eta\lambda_1| < 1$, which requires $\eta < 2/\lambda_1$. Although the above formula has the main theoretical interest of showing that, in a neighbourhood of θ^*, gradient descent converges linearly (**Exercise 2.2.**), in practice it is not useful since we don't know λ_1.

2.2 Stochastic gradient descent

It is important to realize that the update rule in Eq. (2.4) involves at each iteration the calculation $\nabla \mathcal{L}$ (and later its evaluation at θ^*), which depends on the whole data set $(x_1, y_1), \ldots, (x_N, y_N)$. This is why this routine is known as **full batch** gradient descent. Needless to say, when working with large datasets full batch gradient descent is too costly from a computational point of view. Stochastic gradient descent (SGD, Bottou, 2010) allows one to reduce the computational burden and make deep learning on large scale datasets. Let us rewrite the loss function as

$$\mathcal{L}_N(\theta) = \sum_{i=1}^{N} L(y_i, f_\theta(x_i)),$$

where $L(\cdot, \cdot)$ can take different forms depending on the supervised learning problem we address (Section 1.4). Now introduce a random variable I equipped with the following probability measure $\pi\{I = i\} = 1/N$ for all $i \leq N$ and define

$$l_I(\theta) := NL(y_I, f_\theta(x_I)). \tag{2.5}$$

Algorithm 1 Stochastic gradient descent

Require: training data $\{(x_i, y_i)\}_{i \leq N}$ and learning rate $\eta > 0$
Ensure: a local minimum θ^*
1: **while** Not convergence **do**
2: sample $I \sim \pi$ ▷ sampling one obs. uniformly at random
3: $\theta \leftarrow \theta - \eta \nabla l_I(\theta)$
4: **end while**
5: $\theta^* = \theta$

Exercise 2.3. Show that $\nabla l_I(\theta)$ is an unbiased estimator of $\nabla \mathcal{L}_N(\theta)$ (under the probability measure π).

The main idea of SGD is to use $l_I(\cdot)$ in place of $\mathcal{L}_N(\cdot)$ for the gradient computation, as illustrated in Algorithm 1. In words: one observation (y_i, x_i) in the training data set is chosen uniformly at random, with re-injection, and the gradient of the loss function is computed based on that observation alone and used to update θ. A word of caution is needed about the condition "Not convergence". There are several ways to assess whether the learning routine should be stopped or not. A few of them will be discussed later in this notes. For the moment, we just say that once all the training data points have been sampled at least once we say that one **epoch** passed. So the easiest way to quit the while loop in Algorithm 1 is to fix a maximal number of epochs to visit.

The estimator of the loss function in Eq. (2.5) is in general too volatile (high variance), this is why usually SGD relies on **mini-batches** of more that one observations. If we denote by B a set of $|B|$ distinct indices extracted from $\{1, \ldots, N\}$, uniformly at random, the estimator in Eq. 2.5 is replaced by

$$l_B(\theta) := \binom{N}{|B|} \sum_{i \in B} L(y_i, f_\theta(x_i))$$

and line 3 in Algorthm 1 by

$$\theta \leftarrow \theta - \eta \nabla l_B(\theta)$$

2.3 Momentum and adaptive gradient descent

So far, we saw that the building block of gradient descent optimization is

$$\theta^{(t)} = \theta^{(t-1)} - \eta \nabla \mathcal{L}(\theta^{(t-1)}) \tag{2.6}$$

possibly reformulated in mini-batch mode. A lot of research in the last fifteen years focused on how to modify the above update rule in order to obtain a better/faster convergence and an extensive review of the existing improvements is outside the scope of this notes[1]. Here, we just mention a couple of very known improvements the student should be aware of.

A first one consists into leaving the learning rate η unchanged (with respect to t) while accounting for past values of the gradient of the loss on the right hand side of the above equation. This can be done by adding a **momentum** term

$$\begin{cases} \Delta\theta^{(t-1)} = -\eta \nabla \mathcal{L}(\theta^{(t-1)}) + \mu \Delta\theta^{(t-2)} \\ \theta^{(t)} = \theta^{(t-1)} + \Delta\theta^{(t-1)} \end{cases} \tag{2.7}$$

where μ is a positive parameters in $(0, 1)$.

Exercise 2.4 Make explicit the recursion in the first equation in (2.7) to show the impact of the past gradients on the update of θ. Then, assume the loss function has no curvature in a neighbourhood of $\theta^{(0)}$, so that the gradient can be treated as constant at successive iterations. Show that, in the limit $t \to \infty$ the momentum increases the effective learning rate.

Another option consists into making the learning rate time dependent $\eta^{(t)}$ in such a way to make it decrease as long as we approach the stationary point θ^*. In PyTorch, this can be done directly by adopting linear, power law or exponential decays, or indirectly by dedicated learning algorithms. For instance, **RMSProp** modifies Eq. (2.6) as

$$\begin{cases} v^{(t)} = \beta v^{(t-1)} + (1-\beta) \left(\nabla \mathcal{L}(\theta^{(t-1)}) \odot \nabla \mathcal{L}(\theta^{(t-1)}) \right) \\ \theta^{(t)} = \theta^{(t-1)} - \frac{\eta}{\sqrt{(v^{(t)}) + \delta}} \odot \nabla \mathcal{L}(\theta^{(t-1)}) \end{cases} \tag{2.8}$$

[1]The reader can take a look here https://docs.pytorch.org/docs/stable/optim.html to have an idea of how vast is the panorama.

where $v^{(t)}$ is a vector of the same dimension of the gradient, $\odot$ denotes the element-wise multiplication and, in the second equation, the division, the square root and the sum are intended element-wise. The main intuition of RMSProp is to allow for a faster reduction of the learning rate for those parameters associated with initial highest curvatures (via the the accumulated sum of squares of their derivatives) and slower reduction for parameters associated with small initial curvatures. The weighted average on the first equation ($\beta = 0.9$ usually) avoids the learning rate to become too slow too soon. The positive (very small) hyper-parameters δ avoids division by zero. One of the most popular extension of RMSProp, including an additional momentum term is **Adam** (Kingma and Ba, 2014).

2.4 Automatic differentiation

Independently from the particular GD based algorithm used to train the neural net, from the previous section we see that that one needs to be able to *efficiently* compute $\nabla \mathcal{L}_N(\theta)$, namely the gradient of the loss function $\mathcal{L}$ with respect to the network trainable parameters θ. This can be done by the so called **backpropagation** algorithm, which is illustrated in Appendix 2.A. Backpropagation is a particular case of a more general framework: **reverse mode automatic differentiation**. Another type of automatic differentiation (forward mode) exists but its description as well as an in depth dive into automatic differentiation in general is outside the scope of this notes[2]. The main idea in automatic differentiation is to exploit the recursive nature of the chain rule of calculus in order to make the computer evaluate gradients. The main ingredients are i) pre-computed derivatives with respect to arithmetic operators $(+, -, \times, /)$, transcendental and trigonometric functions and ii) a **computational graph**. We now illustrate the last notion with an example. Consider the following function

$$f(x, y) = xy - \log(1 + \exp(xy)) \tag{2.9}$$

[2]A more extensive treatment of the topic can be found in a very nice blog post at https://huggingface.co/blog/andmholm/what-is-automatic-differentiation.

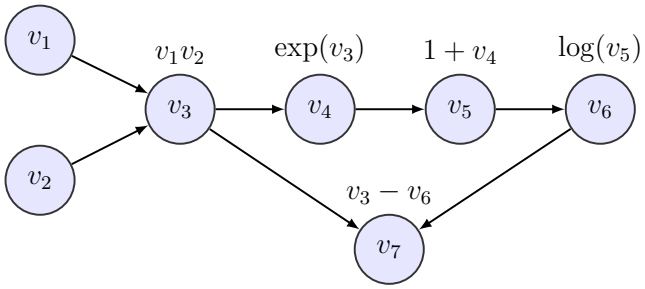

Figure 2.1: Computational graph of f.

in two real variables x and y. That function can be rewritten by means of the following *instrumental* variables

$$\begin{aligned} v_1 &:= x \\ v_2 &:= y \\ v_3 &:= v_1 v_2 \\ v_4 &:= \exp(v_3) \\ v_5 &:= 1 + v_4 \\ v_6 &:= \log(v_5) \\ f(x,y) = v_7 &:= v_3 - v_6 \end{aligned} \tag{2.10}$$

and collected in the directed acyclic graph (DAG) in Figure 2.1. Now, for each variable/node v_i an a*adjoint* variable $\overline{v}_i := \frac{\partial f}{\partial v_i}$ is also computed, for all $i \in \{1, \ldots, 7\}$. Adjoint variables are computed recursively **backward** by making use of the chain rule as follows

$$\overline{v}_i = \sum_{j \in \mathrm{ch}(i)} \frac{\partial f}{\partial v_j} \frac{\partial v_j}{\partial v_i} = \sum_{j \in \mathrm{ch}(i)} \overline{v}_j \frac{\partial v_j}{\partial v_i}$$

where $\mathrm{ch}(i)$ denotes the set of the childrens of v_i on the computational graph (i.e. the nodes v_i sends a link to). Hence, by applying the above formula we

obtain (**Exercise 2.5**)

$$
\begin{aligned}
\overline{v}_7 &= 1 \\
\overline{v}_6 &= -1 \\
\overline{v}_5 &= -\frac{1}{v_5} \\
\overline{v}_4 &= -\frac{1}{v_5} \\
\overline{v}_3 &= -\frac{e^{v_3}}{v_5} + 1 \\
\overline{v}_2 &= \left(1 - \frac{e^{v_3}}{v_5}\right) v_1 \\
\overline{v}_1 &= \left(1 - \frac{e^{v_3}}{v_5}\right) v_2
\end{aligned}
\tag{2.11}
$$

By replacing in the above equations the definitions of the instrumental variables in Eqs. (2.10) we finally have

$$
\overline{v}_1 = \frac{\partial f}{\partial x} = \left(1 - \frac{e^{xy}}{1 + e^{xy}}\right) y \qquad \text{and} \qquad \overline{v}_2 = \frac{\partial f}{\partial y} = \left(1 - \frac{e^{xy}}{1 + e^{xy}}\right) x
$$

where two things should be noticed: i) at each step in Eqs. (2.11) we only needed to compute elementary derivatives of transcendental/trigonometric functions possible under arithmetic operators and ii) in one backward pass we can evaluate ∇f at (x, y).

Algorithm 2 Backpropagation

Require: A neural net f_θ, with L layers evaluated at x (forward pass).
Ensure: The gradients $\{\frac{\partial \mathcal{L}}{\partial W^{(l)}}(\theta)\}_l$, for all $l \leq L$

1: Compute $\nabla_{a^{(L)}}\mathcal{L}$
2: Compute and store $\frac{\partial \mathcal{L}}{\partial W^{(L)}} = (\nabla_{a^{(L)}}\mathcal{L})z_L^T$
3: **for** l in $(L-1), \ldots, 1$ **do**
4: Compute $\nabla_{a^{(l)}}\mathcal{L}$ via Eq. (2.18)
5: Compute and store $\frac{\partial \mathcal{L}}{\partial W^{(l)}} = (\nabla_{a^{(l)}}\mathcal{L})z_l^T$
6: **end for**

2.A Backpropagation

Backpropagation is a recursive algorithm allowing computer scientists to efficiently compute the gradient of the loss function $\mathcal{L}_N(\theta)$ with respect to all the neural-net trainable parameters. The main ingredient in backpropagation is nothing but chain rule of calculus. Let us consider here the example of a two layer neural network trained (say) for multiclass classification

$$f_\theta(x) = \sigma_2(W^{(2)}\sigma_1(W^{(1)}x)) \tag{2.12}$$

where $x \in \mathbb{R}^D$ is the input feature, $W^{(1)} \in \mathbb{R}^{M\times D}$ and $W^{(2)} \in \mathbb{R}^{K\times M}$ are the learnable parameters collected in θ, M hidden units and K the number of classes. The final activation σ_2 is a softmax whereas σ_1 needs not to be specified. Also, the intercept is included in the matrix notation used above. It is useful to introduce the intermediate notation

$$\begin{cases} z = \sigma_1(W^{(1)}x) \\ f_\theta(x) = \sigma_2(W^{(2)}z) \end{cases} \tag{2.13}$$

as well as to denote by $a^{(2)} := W^{(2)}z$ and $a^{(1)} := W^{(1)}x$ the pre-activation units. We only consider the error function for one observation $\mathcal{L} := \mathcal{L}(y, f_\theta(x))$ where y is the true label. Now, having in mind the definitions of $a^{(1)}$ and $a^{(2)}$ it is straightforward to obtain

$$\frac{\partial a_j^{(2)}}{\partial W_{ji}^{(2)}} = z_i \qquad \text{and} \qquad \frac{\partial a_i^{(1)}}{\partial W_{id}^{(1)}} = x_d \tag{2.14}$$

for all $j \leq K$, $i \leq M$ and $d \leq D$. Moreover

$$\frac{\partial \mathcal{L}}{\partial W_{ji}^{(2)}} = \frac{\partial \mathcal{L}}{\partial a_j^{(2)}} \frac{\partial a_j^{(2)}}{\partial W_{ji}^{(2)}} = \delta_j^{(2)} z_i, \tag{2.15}$$

where $\delta_j^{(2)} := \frac{\partial \mathcal{L}}{\partial a_j^{(2)}}$. Now the main equation of backpropagation is the following one

$$\delta_i^{(1)} := \frac{\partial \mathcal{L}}{\partial a_i^{(1)}} = \sum_{j=1}^{K} \delta_j^{(2)} \frac{\partial a_j^{(2)}}{\partial a_i^{(1)}} = \sum_{j=1}^{K} \delta_j^{(2)} W_{ji}^{(2)} \sigma_1'(a_i^{(1)}), \tag{2.16}$$

where $\sigma_1'(\cdot)$ is the first derivative of the sigmoid activation. The last piece we need is

$$\frac{\partial \mathcal{L}}{\partial W_{id}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_i^{(1)}} \frac{\partial a_i^{(1)}}{\partial W_{id}^{(1)}} = \delta_i^{(1)} x_d. \tag{2.17}$$

Apart from being everything we need, the last three equations clearly show a recursive path which becomes even clearer when moving to matrix notation. Indeed, by noting that $\delta^{(2)} := (\delta_1^{(2)}, \ldots, \delta_K^{(2)})^T$ is nothing but $\nabla_{a^{(2)}} \mathcal{L}$, Eq. (2.16) can be rewritten as

$$\frac{\partial \mathcal{L}}{\partial a_i^{(1)}} = \langle \nabla_{a^{(2)}} \mathcal{L}, W_{\cdot,i}^{(2)} \rangle \sigma_1'(a_i^{(1)})$$

and finally

$$\nabla_{a^{(1)}} \mathcal{L} = A^{(1)} (W^{(2)})^T (\nabla_{a^{(2)}} \mathcal{L}), \tag{2.18}$$

where $A^{(1)} := \operatorname{diag}(\sigma_1'(a_1^{(1)}), \ldots, \sigma_1'(a_M^{(1)}))$. Equations (2.15) and (2.17) finally look

$$\frac{\partial \mathcal{L}}{\partial W^{(2)}} = (\nabla_{a^{(2)}} \mathcal{L})\, z^T \qquad \frac{\partial \mathcal{L}}{\partial W^{(1)}} = (\nabla_{a^{(1)}} \mathcal{L}) x^T$$

The general backpropagation algorithm for a feed-forward neural net of depth L in sketched in Algorithm 2, where $z_2, \ldots, z_L$ are the hidden (post) activation units and $z_1 = x$