Chapter 3

Regularization and other tricks

3.1 Overfitting

The previous chapter was devoted to discuss how to solve the optimization problem

$$\hat{\theta}_N =: \arg\min_{\theta \in \mathbb{R}^D} \mathcal{L}_N(\theta), \tag{3.1}$$

for a given training dataset $(x_1, y_1), \ldots, (x_N, y_N)$, neural network f_θ with D trainable parameters and loss function $\mathcal{L}_N(\cdot)$, depending on the learning problem we face (Section 1.4). Ironically, we open this chapter by stating that *exactly* solving that problem generally is not a good idea. Indeed, given a *test* dataset $x_1^*, \ldots, x_M^*$ with (usually unknown) labels $y_1^*, \ldots, y_M^*$, a typical score we really want to minimize is the predictive Root Mean Square Error

$$\mathrm{RMSE}(\theta) = \left(\frac{1}{M} \sum_{i=1}^{M} L(y_i^*, f_\theta(x_i^*)) \right)$$

where $L(\cdot, \cdot)$ once more depends on the learning problem (by the way, the same adopted in $\mathcal{L}_N$). In other terms, we want to be good on new previously unseen data. Due to the high flexibility of neural networks (as well as to other factors) $\hat{\theta}_N$ in Eq. (3.1) might not be the best candidate to minimize the predictive RMSE. When this phenomenon occurs (for instance a trained neural net reaches a very high classification accuracy on the training dataset but a poor one on the test dataset) we say that the model **overfits**. In this chapter we discuss a few standard solutions adopted in order to avoid or reduce overfitting. In turn, all of them relates with either modifying the

objective to minimize in Eq. (3.1) or “stopping” before reaching the actual minimum $\hat{\theta}_N$ is reached.

3.2 Penalties

A first solution consists into adding a penalty term $\Omega(\theta)$ to the training loss function in order to modify the optimization problem

$$\min_{\theta \in \mathbb{R}^D} \{\mathcal{L}(\theta) + \lambda\Omega(\theta)\}$$

where $\lambda \geq 0$ is a user defined hyper-parameter and henceforth we remove the subscript N from $\mathcal{L}_N(\cdot)$ to ease the notation. Although several options exist for $\Omega(\cdot)$, we focus here on two of them, i.e. the l_2 and l_1 norms.

About the former, the minimization to solve becomes

$$\min_{\theta} \{\mathcal{L}(\theta) + \lambda\|\theta\|_2^2\}, \qquad \lambda \geq 0. \tag{I}$$

When taking the gradient of the objective in order to perform gradient descent, we see that the building block in Eq. (2.6) is modified as

$$\theta^{(t)} = \theta^{(t-1)} - 2\eta\lambda\theta^{(t-1)} - \eta\nabla\mathcal{L}(\theta^{(t-1)}).$$

The above equation is the reason why the l_2 penalty is sometimes referred to as **weight decay**.

Exercise 3.1 In the above equation, assume no evidence from the data is available (i.e. $\nabla\mathcal{L}(\cdot) = 0$) and consider the continuous counterpart in which the updates are infinitesimal. Solve the corresponding ordinary differential equation to show that $\theta(t)$ decays exponentially to zero.

Exercise 3.2 Show that, from a Bayesian perspective, the minimization problem (I) is the one be solved in order to compute the maximum at posteriori (MAP) estimate of θ following a multivariate centred isotropic Gaussian *prior* distribution.

Exercise 3.3 Show that there exists a positive η such that the minimization problems (I) and

$$\min_{\theta} \mathcal{L}(\theta) \qquad \text{s. to} \qquad \|\theta\|_2^2 \leq \eta \tag{II}$$

are equivalent.

Solution. We define $g_\eta(\theta) := \|\theta\|_2^2 - \eta$ as well as the following set

$$\mathcal{G} := \{\theta \in \mathbb{R}^D | g_\eta(\theta) \leq 0\}.$$

A solution θ^* of (II) is either internal to $\mathcal{G}$ (*inactive* constraint. i.e θ^* is also solution of (I) for $\lambda = 0$) or belongs to the boundary $\overline{\mathcal{G}}$ and in that case the constraint is *active* and $\nabla g_\eta(\theta^*)$ is orthogonal to the tangent space of $\overline{\mathcal{G}}$ at θ^* and pointing outward whereas $\nabla \mathcal{L}(\theta^*)$ (still orthogonal to the boundary) points inside. It means that

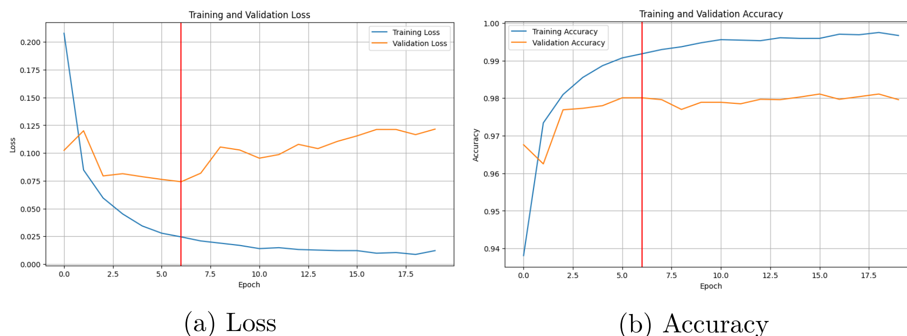
$$\nabla \mathcal{L}(\theta^*) = -\lambda \nabla g_\eta(\theta^*), \qquad \exists \lambda > 0.$$

Since $\nabla g_\eta(\theta) = \nabla \|\theta\|_2^2$ also in this case θ^* is a solution for (I) with its λ. More formally, a solution θ^* of (II) satisfies the KKT conditions

$$\begin{cases} \nabla \mathcal{L}(\theta) = -\lambda \nabla g_\eta(\theta) \\ \lambda \geq 0 \\ \lambda g_\eta(\theta) = 0 \\ g_\eta(\theta) \leq 0 \end{cases} \tag{3.2}$$

and (η is fixed here!) we saw that, together with its λ, that solution also solves (I). Now, we take the opposite point of view and assume $\lambda > 0$ to be fixed, θ^* to be solution of (I) and set $\eta = \|\theta^*\|_2^2$ in (3.2). If (3.2) admitted another solution $\hat{\theta}$ such that $\|\hat{\theta}\|_2^2 < \eta$, this would contradict the fact that θ^* is a solution of (I) because of what previously said, so θ^* is the only solution of (3.2). In this sense we proved that the above minimization problems are indeed equivalent.

An in depth analysis of the estimator of θ obtained by solving Eq (I) is outside the scope of this course. Here we just mention that the intuitive effect of adding the (squared) l_2 penalty is to shrink toward zero the parameters with respect to whom the output is less sensitive, while maintaining the objective function fully differentiable. In contrast, setting $\Omega(\theta) = \|\theta\|_1$ has the effect to push the same parameters exactly to zero. However, the loss function will no longer be differentiable and more complex optimization method are required. When the loss function is quadratic in θ (e.g. in linear regression) the l_2 and l_1 regularised regression models are known as Ridge and Lasso and their properties are inspected in some detail, for instance, in James et al., 2013, Chapter 6.2.



(a) Loss

(b) Accuracy

Figure 3.1: An illustration of overfitting when training a dense neural net on the MNIST dataset. The vertical red bar is the epoch where training should be stopped.

3.3 Early stopping

Early stopping formalizes the idea that while performing gradient descent in order to solve Eq. (3.1) it might be better to stop *before* reaching a stationary point. The easiest way to implement early stopping is two split the entire data set into **training** and **validation** (usually 10% or 20% of training observations are left out for validation) and monitor the loss function on both. Whereas the training loss will continue to decrease up to convergence to a stationary point, the validation loss might stop decreasing up to a certain number of epochs: from that point onward the model starts to overfit the data. There is a number of more sophisticated ways to perform early stopping than simply making *one* split of the available data and this is related with an area of statistics and machine learning called cross-validation[1] and being outside the scope of this course. However, early stopping is automatically implemented in PyTorch and Keras/Tensorflow, we'll see how it works in a dedicated practical section.

3.4 Dropout

Dropout is a very simple yet powerful technique to avoid overfitting. Roughly speaking, it consists into removing a number of links from the neural net,

[1] https://en.wikipedia.org/wiki/Cross-validation_(statistics)

uniformly at random, at any time a training data point undergoes a forward pass during training. More formally, looking at the hidden neurons $h_i^{(l)}$ in Eq. (1.4), adding a Dropout layer after the l-th layer ($l \in \{0, \dots, L-1\}$) means multiplying vector $h_i^{(l)}$ by the mask $R_i := (R_{i1}, \dots, R_{iP})$ elementwise, with $R_{i1}, \dots, R_{iP}$ being i.i.d. Bernoulli random variables with probability of success ρ (typically 30% or 50%) and P is the number of hidden neurons in the layer. The main intuition behind droptout is the following: for each mini-batch of training data we are actually training a different neural net although several weights are shared from one net to another. As such, the final prediction on the test data point can be seen as an ensemble prediction[2] coming from many networks.

There is a subtle link between dropout and penalization (Section 3.2). In order to have an intuition about it, consider the linear regression model with loss function $\mathcal{L}(\beta) := \|Y - X\beta\|_2^2$, where $Y \in \mathbb{R}^N$ (response), $X \in \mathbb{R}^{N\times D}$ (feature matrix) and β in $\mathbb{R}^D$ (parameters to estimate). Dropout can be implemented by introducing a random mask $R \in \{0,1\}^{N\times D}$ such that R_{nd} are i.i.d. Bernoulli i.i.d random variables. with parameter ρ. The random mask multiplies element-wise X, so that the corresponding loss reads $\mathcal{L}_R(\beta) := \|Y - Z\beta\|_2^2$, where $Z = X \otimes R$. When averaging over R, the expected loss reads

$$\mathbb{E}_R\left[\mathcal{L}_R(\beta)\right] = \mathcal{L}(\rho\beta) + \rho(1-\rho)\|\beta\|_{M_X}^2, \tag{3.3}$$

where $\|\beta\|_{M_X}^2 = \beta^T M_X \beta$ and $M_X = \mathrm{diag}(\|X^1\|_2^2, \dots, \|X^N\|_2^2)$, with X^j denoting the j-th column of X. As it can be seen, a quadratic penalty term involving β appears and it is easy to compute

$$\hat{\beta}_\rho := (\rho X^T X + (1-\rho)M_X)^{-1} X^T Y$$

Exercise 3.4 Prove Eq. (3.3).

3.5 Residual blocks and batch normalization

Apart from preventing overfitting, one other reason one might wish to regularize (i.e. smooth) the error function $\mathcal{L}(\theta)$ is to avoid *gradient vanishing* or *exploding* problems, so easing the optimization procedure. Two useful

[2] https://en.wikipedia.org/wiki/Ensemble_learning

weapons against this pathology rely onto a direct modification of the network's architecture. The former consists into adding *residual* connections to one or more layers. Still having in mind Eq. (1.4), the l-th hidden layer can be converted into a residual block as following

$$\begin{aligned} h_i^{(l)} &= \sigma_l \left(W_l h_i^{(l-1)} + b_l \right) + h_i^{(l-1)} \\ &=: F_l(h_i^{(l-1)}) + h_i^{(l-1)}. \end{aligned}$$

Combining pre-existing layers with residual block, the first **ResNet** was introduced by He et al., 2016 and residual or skip-layer connections (skipping more then one layer) are very common in many popular architectures since then. Moreover, residual connections allowed researchers to make interesting connections between networks depth and ordinary differential equations. Indeed, if we read $\{0, \ldots, L\}$ as discrete *time* points, the above equation appears to be the Euler scheme of the following ODE

$$\dot{h_i}(l) = F_l(h_i(l))$$

which is the port of entry in the realm of neural ODEs (Chen et al., 2018).

Similarly to residual connections, also batch normalization acts on the network architecture and, as the name suggests, it comes into play when performing mini-batch gradient descent (Section 2.2). To a batch B of $|B|$ observations $x_1, \ldots, x_{|B|}$, extracted uniformly at random from the training data set, will correspond l-th hidden units $h_1^{(l)}, \ldots, h_{|B|}^{(l)}$, each in dimension P via the forward pass in Eq. (1.4). Batch normalization modifies each unit via

$$\begin{aligned} \mu &:= \frac{1}{|B|} \sum_{i \in B} h_i^{(l)} \\ S &:= \frac{1}{|B|} \sum_{i \in B} \operatorname{diag} \left((h_i^{(l)} - \mu)(h_i^{(l)} - \mu)^T \right) \\ \hat{h}_i^{(l)} &:= \sqrt{S_\epsilon}(h_i^{(l)} - \mu) \end{aligned}$$

where (with a slight abuse of notation) we denote by $\sqrt{S_\epsilon}$ a $P \times P$ diagonal matrix whose (j, j)-th element is $\sqrt{S_{jj} + \epsilon}$, with $\epsilon > 0$, small. Roughly speaking, each entry in the i-th hidden unit is centred and scaled at the batch level. In order to avoid loosing to much flexibility, a further modification is

$$\tilde{h}_i^{(l)} = \Gamma \hat{h}_i^{(l)} + \beta,$$

with Γ diagonal matrix of order P and $\beta \in \mathbb{R}^P$ to be learned by gradient descent.