Machine Learning — Statistical Methods for Machine Learning

Neural networks and deep learning

Instructor: *Nicolò Cesa-Bianchi* version of June 1, 2023

Neural networks (NNs) are a large and complex class of predictors applicable to a variety of tasks, including categorization, regression, and sequence prediction. Typically, NNs are obtained through the combination of simple predictors of the form $g(\boldsymbol{x}) = \sigma\big(\boldsymbol{w}^\top \boldsymbol{x}\big)$. The function $\sigma : \mathbb{R} \to \mathbb{R}$, which is almost always nonlinear, is known as **activation function**. We focus on **feedforward** NNs, computing functions of the form $\boldsymbol{f} : \mathbb{R}^d \to \mathbb{R}^n$. Hence, we do not consider more complex network models, such as recurrent NNs.

The structure of a feedforward NN is a directed acyclic graph $G = (V, E)$ where each node j (except for the input nodes, see later) computes a function $g(\boldsymbol{v})$ whose argument $\boldsymbol{v}$ is the value of the nodes i such that $(i, j) \in E$. The nodes in V are partitioned in three subsets: $V = V_{\text{in}} \cup V_{\text{hid}} \cup V_{\text{out}}$, where V_{in} (with $|V_{\text{in}}| = d$) are the **input nodes** which have no incoming edges, V_{out} (with $|V_{\text{out}}| = n$) are the **output nodes** which have no outgoing edge, and V_{hid} are the **hidden nodes**, which have both incoming and outgoing edges. The set of input nodes and the set of output nodes are respectively called the input layer and the hidden layer.

The simplest form of feedforward NN is a **multilayered NN**, in which the nodes of V can be partitioned in a sequence of layers such that each node of a layer has incoming edges only from nodes in the previous layer and outgoing edges only to nodes of the next layer (see Figure 1). The layers containing the hidden nodes are called hidden layers.

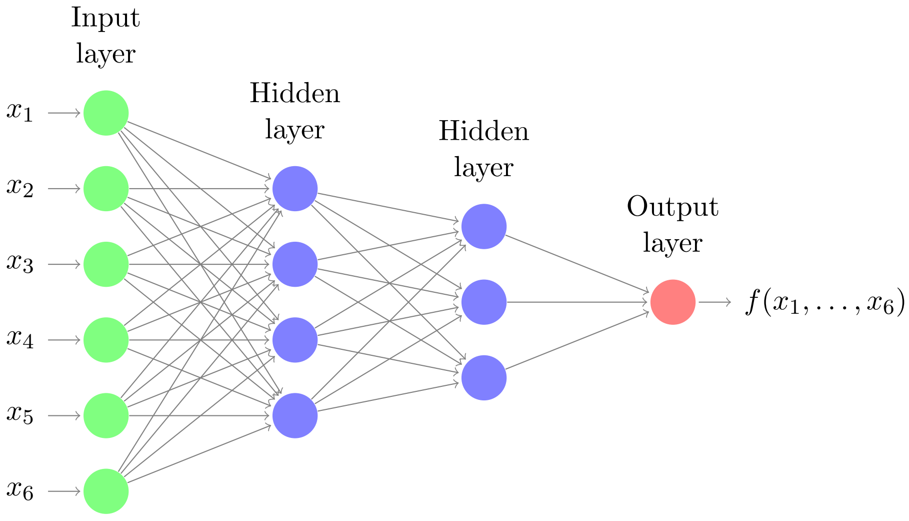

Figure 1: A multilayered NN with single output node and two hidden layers. In this example, the connectivity between layers is complete (there are no missing edges).

A parameter $w_{i,j} \in \mathbb{R}$ (called weight) is associated with every edge $(i, j) \in E$. We use W to denote the $|V| \times |V|$ **weight matrix**, where $(i, j) \notin E$ implies $w_{i,j} = 0$. The graph G, the weight matrix W, and the activation function σ define the function $\boldsymbol{f} = \boldsymbol{f}_{G,W,\sigma}$ computed by the network. In order to compute the value of $\boldsymbol{f}(\boldsymbol{x})$ on an input $\boldsymbol{x} \in \mathbb{R}^d$, we assign the value $v_i \in \mathbb{R}$ to each node $i \in V$ as follows.

Given $j \in V \setminus V_{\text{in}}$, we denote with:

- $\boldsymbol{w}(j)$ the vector whose components are the weights $w_{i,j}$ for every $i \in V$ such that $(i, j) \in E$,
- $\boldsymbol{v}(j)$ the vector whose components are the values v_i for every $i \in V$ such that $(i, j) \in E$.

The vector $\boldsymbol{f}(\boldsymbol{x}) = (f_1(\boldsymbol{x}), \dots, f_n(\boldsymbol{x}))$ is then computed as follows:

1. each input node $i \in V_{\text{in}}$ takes value $v_i = x_i$,
2. each node $j \in V \setminus V_{\text{in}}$ takes value $v_j = \sigma\big(\boldsymbol{w}(j)^\top \boldsymbol{v}(j)\big)$,
3. $f_k(\boldsymbol{x}) = v_k$, where k is the k-th output node.

Given d (dimensionality of input) and n (dimensionality of output), we introduce the class $\mathcal{F}_{G,\sigma}$ of predictors $\boldsymbol{f} : \mathbb{R}^d \to \mathbb{R}^n$ such that $\boldsymbol{f} = \boldsymbol{f}_{G,W,\sigma}$ for some weight matrix W. These are all the predictors computed by a feedforward NN with graph G and activation function σ. Note that when $n = 1$ (one output node) and $|V_{\text{hid}}| = 0$ (no hidden nodes), then $\mathcal{F}_{G,\text{sgn}}$ only contains linear classifiers of the form $f(\boldsymbol{x}) = \text{sgn}\big(\boldsymbol{w}^\top \boldsymbol{x}\big)$.

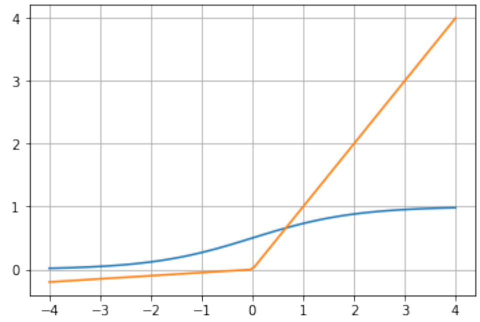

Figure 2: Two examples of activation function: the standard sigmoid $\sigma(z) = 1\big/\big(1 + e^{-z}\big)$ and the leaky ReLU $\sigma(z) = \big(\alpha\,\mathbb{I}\{z < 0\} + \mathbb{I}\{z \ge 0\}\big)z$, where α is typically of the order of 10^{-2}. The leaky ReLU can speed up the training of deep NN (feedforward NN with many hidden layers).

Choice of the activation function. Although many theoretical results hold for a large class of activation functions, in practice the activation function is carefully chosen depending on the learning problem at hand, see Figure 2 for some examples. Also, each layer of the network may use a different activation function. Our analysis does not rely on any specific property of the activation function, although—for the sake of concreteness—we refer to the sigmoid with range rescaled in $[-1, 1]$,

$$\sigma(z) = \frac{1 - e^{-z}}{1 + e^{-z}} \in [-1, 1] \ . \tag{1}$$

Representation power of NN. We now prove a rather surprising result: when data points have binary features, $\boldsymbol{x} \in \{-1, 1\}^d$, then a multilayered NN with a single hidden layer can compute every binary classifier $f : \{-1,1\}^d \to \{-1,1\}$. Note that the restriction to binary inputs is realistic, as each value x_i of a data point $\boldsymbol{x}$ is represented in a computer using a constant number of bits.

Theorem 1. *For every $d \in \mathbb{N}$, there exists a $G = (V,E)$ with $d+1$ input nodes, a single hidden layer, and a single output node, such that $\mathcal{F}_{G,\mathrm{sgn}}$ contains all functions $f : \{-1,1\}^d \to \{-1,1\}$.*

Proof. Let $G = (V,E)$ have 2^d+1 hidden nodes and complete connectivity between layers. Nodes in V are numbered as follows: 0 is the output node, $1,\dots,2^d+1$ are the hidden nodes, and the remaining ones are the $d+1$ input nodes. Fix any function $f : \{-1,1\}^d \to \{-1,1\}$. We now show that there exists W such that $f_{G,W,\mathrm{sgn}} = f$. Let $\boldsymbol{x}_1,\dots,\boldsymbol{x}_N$, with $N \le 2^d$, be all and only the data points $\boldsymbol{x}_i \in \{-1,1\}^d$ such that $f(\boldsymbol{x}_i) = 1$. We assign weights so that the function g_i computed by the i-th hidden node detects whether the input is $\boldsymbol{x}_i$. In other words, $g_i(\boldsymbol{x}) = 1$ if and only if $\boldsymbol{x} = \boldsymbol{x}_i$. This is done as follows. Observe that for every $\boldsymbol{x} \in \{-1,1\}^d$ and every $i = 1,\dots,N$,

$$\begin{array}{ll} \boldsymbol{x}^\top \boldsymbol{x}_i \le d-2 & \text{if } \boldsymbol{x} \neq \boldsymbol{x}_i \\ \boldsymbol{x}^\top \boldsymbol{x}_i = d & \text{otherwise.} \end{array}$$

This implies that $\mathrm{sgn}\big(\boldsymbol{x}^\top \boldsymbol{x}_i - d + 1\big) = 1$ if and only if $\boldsymbol{x} = \boldsymbol{x}_i$. In order to design functions g_i that behave in that way, we proceed as follows: we map each data point $\boldsymbol{x} = (x_1,\dots,x_d)$ to $\widetilde{\boldsymbol{x}} = (\boldsymbol{x},1) = (x_1,\dots,x_d,1)$ using the extra input node—this is the same technique we used for linear classifiers. Then, we assign the weights $\boldsymbol{w}(i) = (\boldsymbol{x}_i, -d+1)$ to each $i = 1,\dots,N$. This implies that $g_i(\widetilde{\boldsymbol{x}}) = \mathrm{sgn}\big(\boldsymbol{w}(i)^\top \widetilde{\boldsymbol{x}}\big) = \mathrm{sgn}\big(\boldsymbol{x}^\top \boldsymbol{x}_i - d + 1\big)$, as desired. We then assign $\boldsymbol{w}(N+1) = (0,\dots,0,1)$ so that $\mathrm{sgn}\big(\boldsymbol{w}(N+1)^\top \widetilde{\boldsymbol{x}}\big) = 1$ for every $\boldsymbol{x}$. The remaining $\boldsymbol{w}(i)$ for $i = N+2,\dots,2^d+1$ are set to zero. The function f is computed by the output node 0 when the weights $\boldsymbol{w}(0)$ are set to

$$\boldsymbol{w}(0) = \big(\underbrace{1,\dots,1}_{N \text{ times}}, N-1, \underbrace{0,\dots,0}_{2^d - N \text{ times}}\big)$$

Indeed, we have

$$\boldsymbol{w}(0)^\top \big(g_1(\widetilde{\boldsymbol{x}}),\dots,g_{2^d+1}(\widetilde{\boldsymbol{x}})\big) = \sum_{i=1}^N g_i(\widetilde{\boldsymbol{x}}) + N - 1$$

which is at least 1 if and only if $g_i(\widetilde{\boldsymbol{x}}) = 1$ for some $i = 1,\dots,N$. This concludes the proof. □

Multilayered networks with a single hidden layer and sigmoidal activation function can be shown to approximate any function $f : [-1,1]^d \to [-1,1]$ with finite Lipschitz constant L (i.e., $\big|f(\boldsymbol{x}) - f(\boldsymbol{x}')\big| \le L\,\|\boldsymbol{x} - \boldsymbol{x}'\|$ for all $\boldsymbol{x},\boldsymbol{x}' \in [-1,1]^d$). More precisely, for all $\varepsilon > 0$ and for all Lipschitz functions f, there exists an network G with a single hidden layer and a weight matrix W such that $\big|f_{G,W,\sigma}(\boldsymbol{x}) - f(\boldsymbol{x})\big| \le \varepsilon$ per ogni $\boldsymbol{x} \in [-1,+1]^d$.

In all these cases, however, the size of the hidden layer is exponential in d for most target functions f. This implies that these results about the representation power of NN with a single hidden layer have no real practical significance. Unfortunately, things do not improve even if we allow an arbitrary number of hidden layers, as shown by the next result.

Theorem 2. *For every $d \in \mathbb{N}$, let $s(d)$ be the smallest integer such that there exists $G = (V,E)$ with $|V| = s(d)$ for which $\mathcal{F}_{G,\mathrm{sgn}}$ contains all functions of the form $f : \{-1,1\}^d \to \{-1,1\}$. Then $s(d) = \Omega(2^{d/3})$. A similar result holds when* sgn *is replaced by the sigmoidal function* (1).

This last result implies that networks of reasonable size (e.g., polynomial in d) have some bias error. As a consequence, we face the usual bias-variance trade-off: we have to choose G and σ so that the bias error of $\mathcal{F}_{G,\sigma}$ is not too large, and there exist efficient algorithms A that learn low-risk models in $\mathcal{F}_{G,\sigma}$ without using too many training examples. As the variance error typically grows with the number of edges in G (corresponding to the number of trainable parameters in W), we seek $G = (V, E)$ and σ such that $\mathcal{F}_{G,\sigma}$ contains many functions (low bias) without E having too many edges (low variance).

A large amount of empirical evidence points at **deep neural networks**, networks with many hidden layers, as models able to strike a good balance between bias and variance. In particular, some theoretical results show that the most effective way of decreasing the approximation error of $\mathcal{F}_{G,\sigma}$ is to create new layers in G as opposed to increase the size of layers that already exist.

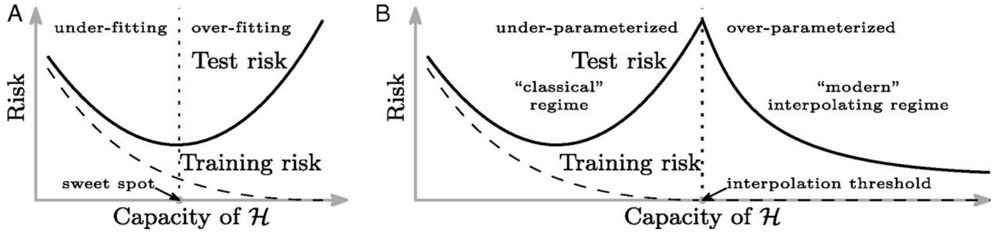

Figure 3: Curves for training risk (dashed line) and test risk (solid line). (A) The classical U-shaped risk curve arising from the bias–variance trade-off. (B) The double-descent risk curve, which incorporates the U-shaped risk curve (i.e., the "classical" regime) together with the observed behavior from using high-capacity function classes (i.e., the "modern" interpolating regime), separated by the interpolation threshold. The predictors to the right of the interpolation threshold have zero training risk. (from "Reconciling modern machine-learning practice and the classical bias–variance trade-off", by Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal.)

On the other hand, recent evidence shows that the bias-variance trade-off fails to capture the behavior of deep networks beyond the "interpolation threshold", which is attained when the number of weights is large enough to reach zero training error, see Figure 3. Under certain conditions, which are not yet fully understood, after reaching a peak at the interpolation threshold, the test error goes down again (hence "double descent") as the number of weights in the network increases. In particular, when the number of weights is much larger than the size of the training set (in the so-called "over-parameterized regime"), the test error can be significantly smaller than the minimum achieved before the interpolation threshold (i.e., in the "classical regime").

Training a neural network. The most natural approach for training a NN is ERM. Unfortunately, the next result shows that we cannot hope to run ERM efficiently even on networks with just one hidden layer of two hidden nodes and the ReLU activation function, $\sigma(z) = \max\{z, 0\}$. For any $k \in \mathbb{N}$, consider the set $\mathcal{F}_{k,\sigma}$ of predictors $f : \mathbb{R}^d \to \mathbb{R}$ of the form

$$f(\boldsymbol{x}) = \sum_{i=1}^{k} a_i \, \sigma\big(\boldsymbol{w}(i)^\top \boldsymbol{x}\big)$$

for all $a_i \in \mathbb{R}$ and $\boldsymbol{w}(i) \in \mathbb{R}^d$ for $i = 1, \dots, k$.

Theorem 3. *For any integer $k \geq 2$, the problem*

$$\min_{f \in \mathcal{F}_{k,\sigma}} \ell_S(f) \qquad \text{where} \qquad \ell_S(f) = \frac{1}{m} \sum_{t=1}^{m} \big(f(\boldsymbol{x}) - y_t \big)^2$$

of minimizing the training error in $\mathcal{F}_{k,\sigma}$ with respect to the square loss on a training set $S = \big\{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_m, y_m)\big\}$ is NP-hard. The problem remains NP-hard even when there exists $f \in \mathcal{F}_{k,\sigma}$ such that $f(\boldsymbol{x}_t) = y_t$ for all $t = 1, \dots, m$.

Even though the square loss $\ell(\widehat{y}) = (y - \widehat{y})^2$ is convex in $\widehat{y}$, the training error is not convex as a function of the trainable parameters $a_i \in \mathbb{R}$ and $\boldsymbol{w}(i) \in \mathbb{R}^d$ for $i = 1, \dots, k$. In general, the problem of minimizing a non-convex function is computationally intractable, and the above theorem shows that computing ERM in the class $\mathcal{F}_{k,\sigma}$ is hard even when there exists a predictor in the class achieving zero training error.

Despite these negative results, NNs are successfully trained using algorithms that reduce the training error without any mathematical guarantee on the quality of the solution. Fix a differentiable loss function ℓ. Given a generic activation function σ, a directed acyclic graph G, and a training set $S = \big\{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_m, y_m)\big\}$, we write $\ell_t(W)$ to denote the loss $\ell(y_t, \widehat{y}_t)$ where $\widehat{y}_t = f_{G,W,\sigma}(\boldsymbol{x}_t)$. The standard training algorithm for NNs is stochastic gradient descent,

$$w_{i,j} \leftarrow w_{i,j} - \eta_t \frac{\partial \ell_{Z_t}(W)}{\partial w_{i,j}} \qquad (i,j) \in E$$

where Z_t is the index of a random training example. In order to speed up convergence, the **mini-batched** version of stochastic gradient descent is often used,

$$w_{i,j} \leftarrow w_{i,j} - \eta_t \frac{1}{|S_t|} \sum_{s \in S_t} \frac{\partial \ell_s(W)}{\partial w_{i,j}} \qquad (i,j) \in E$$

where S_t is a fixed-size random subset of training examples. Because the training error is a non-convex function of W, the algorithm essentially finds only local minima of the training error. Explaining why the weights W found by stochastic gradient descent typically have a small risk is an open research problem.

The procedure to perform gradient descent on feedforward NNs is known as **error backpropagation** algorithm (backprop for short). This algorithm uses the rule for the derivation of composite functions,

$$\frac{df(g(x))}{dx} = \frac{df(g)}{dg} \frac{dg(x)}{dx} \ . \tag{2}$$

We now describe backprop for square loss on a multilayered network with a single output node and with the sigmoidal activation function (1). The adaptation to different loss and activation functions is left as an easy exercise.

Let 0 be the index of the output node. Then $f_{G,W,\sigma}(\boldsymbol{x}_t) = v_0 = \sigma(s_0)$, where we set $s_0 = \boldsymbol{w}(0)^\top \boldsymbol{v}(0)$. For any node i in the first hidden layer (the one closest to the output node), consider the update of the weight $w_{i,0}$ for $(i,0) \in E$. Applying twice the rule (2), we obtain

$$\frac{\partial \ell_t(W)}{\partial w_{i,0}} = \frac{\partial \ell(v_0, y_t)}{\partial w_{i,0}} = \frac{d\ell(v_0, y_t)}{dv_0} \frac{dv_0}{ds_0} \frac{\partial s_0}{\partial w_{i,0}} = \ell'(v_0) \frac{d\sigma(s_0)}{ds_0} \frac{\partial s_0}{\partial w_{i,0}} \ .$$

The derivative $\ell'(v_0) = \frac{d\ell(v_0,y_t)}{dv_0}$ is computed from the loss function,

$$\ell(x,y) = (x-y)^2 \qquad \text{implies} \qquad \frac{\partial \ell(x,y_t)}{\partial x} = 2(x-y) \ .$$

Similarly, the derivative $\frac{d\sigma(s_0)}{ds_0} = \sigma'(s_0)$ is computed from the activation function,

$$\sigma(z) = \frac{1-e^{-z}}{1+e^{-z}} \qquad \text{implies} \qquad \sigma'(z) = \frac{\bigl(1-\sigma(z)\bigr)^2}{2} \ .$$

Finally, recalling that $s_0 = \boldsymbol{w}(0)^\top \boldsymbol{v}(0)$, we obtain

$$\frac{\partial s_0}{\partial w_{i,0}} = v_i = \sigma\bigl(\boldsymbol{w}(i)^\top \boldsymbol{v}(i)\bigr) \ .$$

We can then compute the gradient as

$$\frac{\partial \ell_t(W)}{\partial w_{i,0}} = \ell'(v_0)\,\sigma'(s_0)\,v_i \ .$$

We now compute $\frac{\partial \ell_t(W)}{\partial w_{i,j}}$ for all nodes i in the second hidden layer. For each $(i,j) \in E$, we look at the path $(i,j) \to (j,0)$. Using $s_i = \boldsymbol{w}(i)^\top \boldsymbol{v}(i)$ we compute the gradient as follows

$$\frac{\partial \ell_t(W)}{\partial w_{i,j}} = \underbrace{\frac{d\ell(v_0,y_t)}{ds_0}}_{\ell'(v_0)\sigma'(s_0)} \underbrace{\frac{\partial s_0}{\partial v_j}}_{w_{j,0}} \underbrace{\frac{dv_j}{ds_j}}_{\sigma'(s_j)} \underbrace{\frac{\partial s_j}{\partial w_{i,j}}}_{v_i} = \ell'(v_0)\,\sigma'(s_0) w_{j,0}\,\sigma'(s_j) v_i \ .$$

Starting from the nodes i of the third hidden layer, the computation of the gradient $\frac{\partial \ell_t(W)}{\partial w_{i,j}}$ is a bit more complex because we have to take into account all the paths $(i,j) \to (j,k) \to (k,0)$. Specifically, we need to compute $\frac{\partial \ell_t(W)}{\partial s_j} = \frac{\partial \ell(v_0,y_t)}{\partial s_j}$ where v_0 depends on s_j through all nodes k in the paths $(i,j) \to (j,k) \to (k,0)$. Note that there exists functions $g, h_1, \dots, h_r$ such that $v_0 = g\bigl(h_1(s_j), \dots, h_r(s_j)\bigr)$ where g computes v_0 as a function of $s_k = h_k(s_j)$ for each k in the path, and h_k computes s_k as a function of s_j. The rule for the derivation of the multivariate composite functions then says that

$$\frac{dg}{ds_j} = \sum_{k=1}^{r} \frac{\partial g}{\partial h_k} \frac{dh_k}{ds_j}$$

In our case, we then have

$$\frac{\partial \ell(v_0,y_t)}{\partial s_j} = \sum_{k\,:\,(j,k)\in E} \frac{\partial \ell(v_0,y_t)}{\partial s_k} \frac{\partial s_k}{\partial s_j} = \sum_{k\,:\,(j,k)\in E} \frac{\partial \ell(v_0,y_t)}{\partial s_k} \frac{\partial s_k}{\partial v_j} \frac{dv_j}{s_j} = \sum_{k\,:\,(j,k)\in E} \frac{\partial \ell(v_0,y_t)}{\partial s_k} w_{j,k}\,\sigma'(s_j) \ .$$

Hence, introducing the recursive definition,

$$\delta_j = \frac{\partial \ell_t(W)}{\partial s_j} = \begin{cases} \ell'(v_0)\,\sigma'(s_0) & \text{if } j = 0 \\ \sigma'(s_j) \displaystyle\sum_{k\,:\,(j,k)\in E} \delta_k\, w_{j,k} & \text{otherwise} \end{cases}$$

we can finally write the partial derivative for any $(i,j) \in E$ as

$$\frac{\partial \ell_t(W)}{\partial w_{i,j}} = \frac{\partial \ell_t(W)}{\partial s_j} \frac{\partial s_j}{\partial w_{i,j}} = \delta_j\, v_i \ .$$