Artificial Neural Networks

1 Multi-layer perceptron

In the last lecture, we discussed supervised learning with a linear hypothesis class of the form

\[ y = w^T x + b \]

parametrized by \( n \) weights \( w = (w_0, w_1, \ldots, w_n) \) and a bias \( b \). In the machine learning literature, this family of functions (or “architecture” as we shall call it in the sequel) is known as (linear) perceptron.

We have seen that in the case of binary logistic regression (which, despite the name, is a binary classification problem) the scalar output \( y \) of the hypothesis was further fed into the logistic (a.k.a. sigmoid function

\[ \psi(t) = \frac{e^t}{1 + e^t}. \]

This can be viewed as a two-dimensional output of the form

\[ y = \left( \frac{e^{w_1^T x + b}}{1 + e^{w_1^T x + b}}, \frac{1}{1 + e^{w_1^T x + b}} \right) \]

which can be interpreted as the vector of probabilities of the instance \( x \) belonging to each of the two classes.

Using this perspective, the linear perceptron model can be generalized to the \( k \) class cases according to

\[ Y = \frac{e^{W^T x + b}}{1^T e^{W^T x + b}} = \left( \frac{e^{w_1^T x + b_1}}{\sum_{i=1}^{k} e^{w_i^T x + b_i}}, \ldots, \frac{e^{w_n^T x + b_n}}{\sum_{i=1}^{k} e^{w_i^T x + b_i}} \right), \]

where \( W \) is a \( k \times n \) weight matrix whose rows are denoted as \( w_i \), \( b \) is a \( k \)-dimensional bias vector, and \( 1 \) is an appropriately-sized vector of ones. This generalization of the logistic function used to normalize the output into the form of a vector of probabilities is known as softmax. Softmax is a function of the form

\[ \psi(z) = \frac{e^z}{1^T e^z} \]

that highlights the maximal value in the vector \( z \) and suppresses other elements that are significantly lower than the maximum.
1.1 Adding layers

The linear perceptron model is rather limited due to its linearity. For example, it cannot produce the XOR function. A much more powerful family of functions is obtained by applying a non-linearity to the output of a linear perceptron and concatenating several such models. We define the \( i \)-th layer as

\[
y_i = \varphi_i(W_i y_{i-1} + b_i),
\]

for \( i = 1, \ldots, L \), where \( y_{i-1} \) is an \( n_{i-1} \)-dimensional input, \( y_i \) is an \( n_i \)-dimensional output, \( W_i \) is an \( n_i \times n_{i-1} \) matrix of weights (whose columns are denoted as \( w_i^1, \ldots, w_i^{n_i-1} \)), \( b_i \) is an \( n_i \)-dimensional bias vector, and \( \varphi_i : \mathbb{R} \to \mathbb{R} \) is a non-linear function applied element-wise. Setting \( y = y_L \) and \( y_0 = x \), a multi-layer perceptron (MLP) with \( L \) layers is obtained. MLP can be described by the following input-to-output map

\[
y = \varphi_L(W_L \varphi_{L-1}(W_{L-1} \varphi_{L-2}(\cdots \varphi_1(W_1 x) \cdots))).
\]

parametrized by the weight matrices \( \{W_1, \ldots, W_L\} \) and bias vectors \( \{b_1, \ldots, b_L\} \) which we will collectively denote as a pseudo-vector \( \Theta \).

Graphically, the \( i \)-layer can be thought of a weighted directed graph connecting each of the \( n_{i-1} \) inputs to \( n_i \) sum nodes with the weights given by elements of \( W_i \). The output of each sum node undergoes a non-linearity and together the \( n_i \) outputs form the input of the following layer. Because of its (deliberate) resemblance to biological neural networks, MLP is called an (artificial) neural network. In the jargon of artificial neural networks, each sub-graph of the form \( y_{j}^i = \varphi_i(y_{j-1}^i w_j^i + b_j) \) is called a neuron (the \( j \)-th neuron in \( i \)-th layer), its non-linearity \( \varphi_i \) is called an activation function, and its output \( y_{j}^i \) an activation. MLP is a feedforward neural network, since the graph is acyclic – the data flow forward from the input to the output without feedback loops.

Unlike their single-layered linear counterparts, MLPs constitute a potent hypothesis class. In fact, even with just two layers, MLPs were shown to be universal approximators – their weights can be selected to approximate any function under mild technical conditions, provided they have enough degrees of freedom (sufficiently large number of weights).

1.2 Non-linearity

Various functions can be used as the element-wise nonlinearities (activation function) of the MLP. Older neural networks used the logistic function (a.k.a. sigmoid)

\[
\varphi(t) = \frac{1}{1 + e^{-t}}
\]

saturating the input in \( \mathbb{R} \) between 0 and 1, or its shifted and scaled version

\[
\varphi(t) = \frac{e^t - e^{-t}}{e^t + e^{-t}} = \tanh t.
\]
The arctangent function also has a sigmoid-like behavior.

However, due to numerical issues that will be discussed in the sequel, these functions were nowadays almost universally replaced by the rectifier function (a.k.a. rectified linear unit or ReLU)

\[ \varphi(t) = [t]_+ = \max\{t, 0\}. \]

Note that this function has the derivative of exactly 0 on \((-\infty, 0)\), exactly 1 on \((0, \infty)\), and is non-smooth at 0. These facts justifying its choice will be discussed in the sequel.

In addition to element-wise non-linearities, modern neural networks sometimes use “horizontal” non-linearities acting on the entire activation vector. One typical choice of such a non-linearity adopted in classification networks is a softmax function applied to the activation of the last (output) layer. Other non-linearities of this kind are pooling operations that will be discussed in the sequel.

## 2 Supervised training

Now equipped with a new richer hypothesis class, let us zoom out to see the whole picture. In the supervised learning problem, we are given a finite sample of labeled training instances \(\{(x_i, y_i)\}_{i=1}^{N}\). We then select a hypothesis that minimizes the empirical (in-sample) loss function,

\[
 h^* = \arg \min_{h \in H} \frac{1}{N} \sum_{i=1}^{N} \ell(h(x_i), y_i).
\]

In our terms, this minimization problem can be written as

\[
 \Theta^* = \arg \min_{\Theta} \frac{1}{N} \sum_{i=1}^{N} \ell_i(h_{\Theta}(x_i))
\]

where \(h_{\Theta}\) is the MLP parametrized by the pseudo-vector \(\Theta\). Note that to simplify notation we dropped the dependence of the \(i\)-th pointwise loss term on \(y_i\), denoting it by \(\ell_i\). We will henceforth denote the loss function as

\[
 L(\Theta) = \frac{1}{N} \sum_{i=1}^{N} \ell_i(h_{\Theta}(x_i))
\]

emphasizing that we are interested in its dependence on the model parameters \(\Theta\). Let us now discuss how to minimize it with respect to \(\Theta\).

### 2.1 Global and local minima

Let us assume that \(L\) is a function of an \(m\)-dimensional argument \(\theta\) defined on all \(\mathbb{R}^m\) (we can always parse all the degrees of freedom of our neural network into an \(m\)-dimensional vector). A point \(\theta^*\) is called a global minimizer of \(L\) if for any \(\theta\), \(L(\theta) \geq L(\theta^*)\). The
corresponding value of the function, $L(\theta^*)$, is called a **global minimum**. The latter term is often (strictly speaking; erroneously) used to denote the minimizer as well. A point $\theta^*$ is called a **local minimizer** of $L$ if there exists $\epsilon > 0$ such that $\theta^*$ is a global minimizer of $L$ on the ball $B_\epsilon(\theta^*)$.

Unless $L$ satisfied special properties (such as convexity), finding its global minimizer is an unsolvable problem. On the other hand, finding a local minimizer is a much easier task, since local minimizers can be characterized using local information (i.e., derivatives). Assuming $L$ is $C^1$, from elementary multivariate calculus we should recollect the first-order necessary condition for $\theta^*$ being a local minimizer:

$$\nabla_\theta L(\theta^*) = 0.$$ 

Obviously, this is not a sufficient condition – in fact, a local maximum and a saddle point also satisfy it. However, the latter two types of extremal points (characterized by negative curvature) are **unstable**, which will allow methods such as stochastic gradient descent not to remain stuck at such points.

As a reminder, the **gradient** of a multi-variate function is an operator $\nabla L : \mathbb{R}^m \rightarrow \mathbb{R}^m$. At a given point $\theta$, it produces a vector $\mathbf{g} = \nabla L(\theta)$ satisfying

$$dL = \langle \mathbf{g}, d\mathbf{x} \rangle = \mathbf{g}^T d\theta;$$

in other words, an inner product of the argument change $d\theta$ with the gradient yields the differential $dL$.

### 2.2 Gradient descent

We can therefore suggest a very simple iterative strategy for finding a local minimum, which can be summarized as the following “algorithm”:

Starting with some *initial guess* $\theta_0$, repeat for $k = 1, 2, \ldots$

1. Select a **descent direction** $\mathbf{d}_k$
2. Select a **step size** $\eta_k$
3. Update $\theta_k = \theta_{k-1} + \eta_k \mathbf{d}_k$
4. Check optimality condition at $\theta_k$ and stop if minimum is reached

(In practice, rather than checking the optimality condition, we will run the algorithm for a fixed number of iterations and stop it prematurely based on the value of cross-validation loss – these details will be discussed further in the course.)

The main ingredient of the above “algorithm” is the choice of the descent direction, i.e., a direction a (small) step in which decreases the value of the function. Let $\theta$ be our current
iterate (we drop the iteration subscript) and let $d$ be a direction. Once a direction is chosen, we can consider a one-dimensional “section” of the $m$-dimensional function $L$,

$$f(\eta) = L(\theta + \eta d).$$

The quantity

$$f'(0) = \left. \frac{dL(\theta + \eta d)}{d\eta} \right|_{\eta=0} = \nabla L(\theta)^T d$$

is known as the directional derivative of $L$ at point $\theta$ in the direction $d$. A negative directional derivative indicates that a small step in the direction $d$ decreases the value of the function. Geometrically, this means that a descent direction forms an obtuse angle with the gradient (or an acute angle with the negative gradient).

Let us now approximate our function linearly around $\theta$,

$$L(\theta + d) \approx L(\theta) + \nabla L(\theta)^T d$$

and ask ourselves what direction minimizes the difference $L(\theta + d) - L(\theta) \approx \nabla L(\theta)^T d$ – we could call such a direction the steepest descent direction. Obviously, this linear approximation is unbounded, so we need to normalize the length of $d$. Different choices of the norm lead to different answers (so there are many steepest directions); in the $\ell_2$ sense we obtain

$$d = -\nabla d.$$

This choice of the descent direction leads to a family of algorithms known as gradient descent.

Our next goal is to select the step size $\eta$. Ideally, once we have the direction $d$, we would like to solve for

$$\eta = \arg \min_\eta L(\theta + \eta d).$$

While there exist various methods known as line search to solve such a one-dimensional minimization problem, usually they come at the expense of unaffordable extra complexity. In deep learning, a much more common choice is to use a vanishing sequence of weights that start with some initial $\eta_0$ which is kept for a certain number of iterations and then gradually reduced as $1/k$. Using the statistical mechanics metaphor, such a reduction in the step size resembles a decrease in temperature and is therefore referred to as annealing.

Gradient descent can be thus summarized as

Starting with some initial guess $\theta_0$, repeat for $k = 1, 2, \ldots$

1. Select a step size $\eta_k$

2. Update $\theta_k = \theta_{k-1} - \eta_k \nabla L(\theta_{k-1})$

3. Check optimality condition at $\theta_k$ and stop if minimum is reached

We will discuss variants of the gradient descent algorithm that are used in practice in the sequel.
2.3 Error backpropagation

The main computation ingredient in the gradient descent algorithm is the gradient of the loss function w.r.t. the network parameters $\theta$. Obviously, since an MLP is just a composition of multi-variate functions, the gradient can be simply computed invoking the chain rule. However, recall that the output of the network is usually a $k$-dimensional vector, whereas the parameters are a collection of $n_i \times n_{i-1}$ weight matrices and $n_i$-dimensional bias vectors. The gradient of a vector with respect to a matrix (formally termed the Jacobian) is a third-order tensor, which is not exactly nice to work with.

A much more elegant approach to apply the chain rule takes advantage of the layered structure of the network. As an illustration, we start with a two-layer MLP of the form

$$y = \varphi(A\phi(Bx)),$$

where $\varphi$ and $\phi$ are the two non-linearities, and $A$ and $B$ are the two weight matrices. We are ignoring the bias terms for the sake of exposition clarity.

To analyze the influence of the last (second) layer, we denote its input as $y' = \phi(Bx)$, and the input to the second layer activation function as $z = Ay'$. In this notation, we have $y = \varphi(Ay')$. According to the chain rule,

$$\frac{\partial L}{\partial A} = \frac{\partial y}{\partial A} \frac{\partial L}{\partial y} = \sum_{j=1}^{k} \frac{\partial y_j}{\partial A} \frac{\partial L}{\partial y_j}.$$

For convenience, let us adopt the standard deep learning notation, according to which the derivative of the loss w.r.t. to a parameter $*$ is denoted as $\delta *$. In our case,

$$\delta y = \frac{\partial L}{\partial y} = \left( \frac{\partial L}{\partial y_1}, \ldots, \frac{\partial L}{\partial y_k} \right)^T$$

is the gradient of the loss w.r.t. its input, and $\delta A$ is a matrix whose elements are $\frac{\partial L}{\partial a_{ij}}$, etc. In this notation, we can rewrite

$$\delta A = \sum_{j=1}^{k} \frac{\partial y_j}{\partial A} \delta y_j.$$

We can write $\frac{\partial y_j}{\partial A}$ as a matrix of the size of $A$, filled with zeros except the $j$-th row, which is given by $\varphi'(z_j)y'^T$. Substituting this result into the former sum yields

$$\delta A = \begin{pmatrix} \delta y_1 \varphi'(z_1)y'^T \\ \vdots \\ \delta y_k \varphi'(z_k)y'^T \end{pmatrix} = \text{diag}\{\delta y\} \text{diag}\{\varphi'(z)\} \begin{pmatrix} y'^T \\ \vdots \\ y'^T \end{pmatrix} = \text{diag}\{\delta y\} \text{diag}\{\varphi'(z)\} 1y'^T.$$
To analyze the influence of the first layer, we denote $z' = Bx$. To derive the gradient of the loss w.r.t. the first layer parameter $B$, we again invoke the chain rule

$$\frac{\partial L}{\partial B} = \frac{\partial y'}{\partial B} \frac{\partial L}{\partial y'} = \sum_{j=1}^{\partial y'_j} \frac{\partial y'_j}{\partial B} \delta y'_j.$$

As before, $\frac{\partial y'_j}{\partial B}$ is a matrix of the size of $B$, filled with zeros except the $j$-th row, which is given by $\phi'(z'_j)x^T$, so

$$\delta B = \text{diag}\{\delta y'\} \text{diag}\{\phi'(z')\}1x^T.$$

It remains to derive

$$\frac{\partial y'}{\partial y'} = \frac{\partial L}{\partial y'} = \frac{\partial y}{\partial y'} \frac{\partial L}{\partial y}.$$

From $y = \varphi(Ay')$, we have

$$\frac{\partial y}{\partial y'} = \text{diag}\{\varphi'(z)\} A^T,$$

from where

$$\delta y' = \text{diag}\{\varphi'(z)\} A^T \delta y.$$

We can therefore summarize the chain rule in our two-layer MLP as follows: First, we propagate the data forward through the network, computing

$$z' = Bx$$
$$y' = \varphi(z')$$
$$z = Ay'$$
$$y = \varphi(z).$$

Then, we propagate the derivatives backward through the network:

$$\delta y = \nabla L(y)$$
$$\delta A = \text{diag}\{\delta y\} \text{diag}\{\varphi'(z)\}1y'^T$$
$$\delta y' = \text{diag}\{\varphi'(z)\} A^T \delta y$$
$$\delta B = \text{diag}\{\delta y'\} \text{diag}\{\phi'(z')\}1x^T.$$

The entire procedure, known as error backward propagation or backpropagation for short can be applied recursively for any number of layers.

**Forward pass:** starting with $y_0 = x$, compute for $k = 1, \ldots, L$

- $z_k = W_k y_{k-1}$
- $y_k = \varphi_k(z_k)$

and output $y = y_L$. 

Backward pass: starting with $\delta y_L = \nabla L(y)$, compute for $k = L, L - 1, \ldots, 1$

- $\delta W_k = \text{diag}\{\delta y_k\} \text{diag}\{\phi_k'(z_k)\} y^T_{k-1}$
- $\delta b_k = \text{diag}\{\delta y_k\} \phi_k'(z_k)$
- $\delta y_{k-1} = \text{diag}\{\phi_k'(z_k)\} W^T_k \delta y_k$

We remind that $\delta W_k$ and $\delta b_k$ are blocks of coordinates of the gradient of the loss $L$ with respect to the network parameters.

### 2.4 Exploding and vanishing gradients

Backpropagation allows a recursive calculation of the loss gradient w.r.t. the parameters of the network without the need to ever construct the Jacobian matrices of each layer’s output w.r.t. its input. Note, however, that in order to compute the gradient w.r.t. the first layer, $\delta W_1$, one need to compute the product of $\phi_L'(z_L), \ldots, \phi_1'(z_1)$. This may lead to numerical instabilities. For example, in a network with $L = 20$ layers, a slope of $\phi' = 2$ in each activation function would be amplified by $10^6$. Similarly, a slope of $\phi' = 0.5$ would diminish to $10^{-6}$ – practically to zero. This problem is known as vanishing and exploding gradients, and it prevented end-to-end supervised training of deep neural networks from random initialization.

The introduction of ReLU activations mitigated this problem. In ReLU, the derivative is 1 for positive arguments and 0 for negative ones. This implies that depending on the path through the network from the output back to the inputs, the product of the activation derivatives will always be either 0 or 1. The 0 derivative for negative arguments could still lead to vanishing gradients, but practice shows that, on the contrary, it helps optimization and promotes sparse solutions.

ReLU was probably one of the few significant algorithmic changes in the classical neural networks that enabled deep learning.

### 3 Convolutional neural networks

The layers on MLP described so far are termed *fully connected* in the deep learning literature, due to the fact that every layer input is connected (through some weight) to every output. For large input and output dimensions, such an architecture results in a vast number of degrees of freedom, which increases the network complexity and requires more data to train.

#### 3.1 Weight sharing and shift invariance

*Weight sharing* is a strategy aiming at reducing the layer complexity by reusing the same weights at different parts of the input. For the sake of the following discussion, we assume
the input to be discrete and infinitely supported (i.e., a sequence \( x = \{ x_i \}_{i \in \mathbb{Z}} \)). The output is also assumed to be a sequence, \( y = \{ y_i \}_{i \in \mathbb{Z}} \). Let us consider the output of the \( i \)-th neuron, \( y_i = \varphi \left( \sum_{j \in \mathbb{Z}} w_{ij} x_j + b + i \right) \).

In many cases such as audio signals, images, etc., it is reasonable to assume that the same operation is valid at different parts of the signal. Mathematically, this can be expressed by asserting that the action of the neuron commutes with the action of a translation group. This leads to demanding

\[
\varphi \left( \sum_{j \in \mathbb{Z}} w_{i-m,j} x_j + b_{i-m} \right) = \varphi \left( \sum_{j \in \mathbb{Z}} w_{ij} x_{j-m} + b_i \right)
\]

for every input \( x \). Since the non-linearity is applied element-wise, the equivalent condition holds on its arguments as well,

\[
\sum_{j \in \mathbb{Z}} w_{i-m,j} x_j + b_{i-m} = \sum_{j \in \mathbb{Z}} w_{ij} x_{j-m} + b_i = \sum_{j' \in \mathbb{Z}} w_{i,j'+m} x_{j'} + b_i.
\]

This implies \( b_i = \text{const} \) and \( w_{i-m,j} = w_{i,j+m} \); in other words, if we consider \( w_{ij} \) to be the elements of an infinite weight matrix, it will have equal elements on each of its diagonals. Another way to express is is by saying that \( w_{ij} \) is a function of \( i - j \).

### 3.2 Toeplitz operators and convolution

A linear operator exhibiting the above structure is called *Toeplitz*. The output of a shift-invariant (Toeplitz) neuron can be written as

\[
y_i = \varphi \left( \sum_{j \in \mathbb{Z}} w_{i-j} x_j + b \right).
\]

Note that the weights \( w \) can now be considered as a window that is applied to the input at a certain location to produce an output at the same location, and then is slid to a different input location to produce the corresponding output. This operation (the application of the Toeplitz operator) called *convolution*, denoted as

\[
(w * x)_i = \sum_{j \in \mathbb{Z}} w_{i-j} x_j = \sum_{j \in \mathbb{Z}} w_j x_{i-j} = (x * w)_i.
\]

In this notation, the action of our layer can be written as

\[
y = \varphi (w * x + b).
\]

In the signal processing jargon, we can say that the input signal \( x \) is filtered by a filter with the impulse response \( w \).
3.3 Convolutional layer

Neural networks making use of shift-invariant linear operations are called convolutional neural networks (CNNs). A convolutional layer accepts an $m$-dimensional vector-valued infinitely supported signal $x = (x^1, \ldots, x^m) = \{(x_i^1, \ldots, x_i^m)\}_{i \in \mathbb{Z}}$; each input dimension is called a channel or feature map. The layer produces an $n$-dimensional infinitely supported signal $y = (y^1, \ldots, y^n) = \{(y_i^1, \ldots, y_i^n)\}_{i \in \mathbb{Z}}$ by applying a bank of filters,

$$y^j = \varphi \left( \sum_{i=1}^{m} w^{ij} * x^i \right),$$

or, explicitly,

$$y^j_k = \varphi \left( \sum_{i=1}^{m} \sum_{p} w^{ij}_p x_{k-p}^i \right).$$

In practice, each filter $w^{ij}$ is supported on some small fixed domain.