NOTES: INTRODUCTION TO ARTIFICIAL NEURAL NETWORKS

MICHAŁ GNACIK

In this set of notes we provide all the elements that allow the reader to code all described neural networks from scratch. We are not providing the details regarding the convergence of certain algorithms. As the main reference on various neural networks we refer the reader to [1].

1. Feedforward Neural Networks

Let $n, m \in \mathbb{N}$ and let $x = (x_1, \ldots, x_n), y = (y_1, \ldots, y_m) \in \mathbb{R}^m$ be the input vector (one element of a training data set) and the desired output vector, respectively. Here n represents the number of features (predictors) and the y is the response (or the target). We aim to construct a network so that given the input x the output $\widehat{y}$ is obtained so that $\|y - \widehat{y}\|^2$ is as small as possible.
We consider a network or directed acyclic graph $G = (V, E)$ (V is a set of vertices or nodes and E is a set of all edges (between nodes)) with the following structure of layers (organised disjoint blocks of graph vertices/nodes/neurons), see also Figure 1.

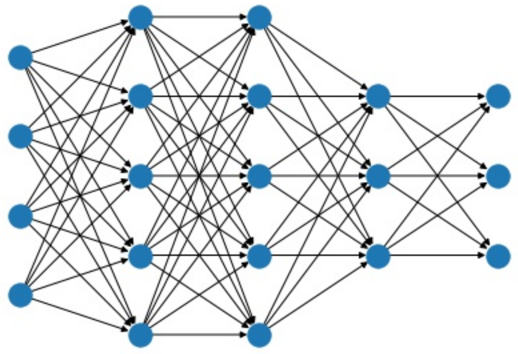

Figure 1. A feedforward neural network with input of size 3, three hidden layers and output of size 2.

(1) **Input layer**; each node of this layer corresponds to x_i for $1 \leq i \leq n$, none of the nodes within this layer are connected by edges.
(2) **Hidden layers**; first fix the number of hidden layers $n_h \geq 0$. Assume that k-th hidden layer has $r_k \in \mathbb{N}$ nodes (setting $r_0 = n$ so the input layer has n nodes and $r_{n_h+1} = m$ so that the output layer has m nodes). Each of the nodes from the input layer is connected (forward only) with the first hidden layer with specified weights, none of the nodes within a single hidden layer are connected by edges, each of the

node of the hidden layer is connected to each of the node of the next layer (forward connection only), in particular, the last hidden layer is connected forward with the output layer.

(3) **Output layer**; the number of the nodes of this layer correspond to the number of the coordinates of y vector. As mentioned earlier, each of the node is connected with each of the node of the previous (last hidden) layer, but the direction is forward, that is, one can travel from the previous layer to output layer but not vice-versa.

There are $n_h + 1$ matrices $W_k = [w_{ij}^{(k)}]$ whose entries correspond to the weights between the nodes of $k-1$ and k layer, and n_h+1 vectors $B_k = (b_1^{(k)}, \dots, b_{r_k}^{(k)})$ for $k \in \{1, \dots, n_h+1\}$ that contain the bias (constant value that shifts the activation from the origin) for the nodes.

Remark 1.1. A feedforward neural network with no hidden layers is also known a *perceptron*. One may think about the feedforward neural network as a composition of perceptrons, and refer to it as the multi-layer perceptron (MLP).

We describe how the network output $\widehat{y} \in \mathbb{R}^m$ is obtained in the next section.

1.1. **Feeding forward.**

Assume that i-th hidden layer is equipped with an activation function g_i for $i \in \{1, \dots, n_h\}$ and the output layer is equipped with the activation function $g_{n_h+1} := g_{\mathsf{out}}$. Here we list a few examples of differentiable activation functions:

Example 1.2.

(1) identity: $I(x) = x$ with range $\mathbb{R}$.
(2) sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}}$ has range $(0, 1]$.
(3) hyperbolic tangent: $g(x) = \tanh x = \frac{e^x+e^{-x}}{e^x-e^{-x}}$ has range $(-1, 1)$.
(4) SoftPlus: $f(x) = \ln(1 + e^x)$ has range $(0, \infty)$.

Note that all but first one, are non-linear. Only non-linear activation functions enable networks to compute non-trivial problems using a small number of neurons.

First note that the size of each W_k matrix is $r_{k-1} \times r_k$ for $k \in \{1, \dots, n_h + 1\}$. We will use the following notation, $W_k[:, j]$ is the j-th column of W_k matrix.

Obtaining $\widehat{y}$ is achieved by the following iterative process: for brevity set $a^{(0)} := x$ so that $a^{(0)} = (a_1^{(0)}, \dots, a_n^{(0)})$. At each neuron (node) of the first hidden layer we arrive at the following quantities:

$$\begin{aligned} z_k^{(1)} :=& \left\langle a^{(0)}, W_1[:, k] \right\rangle + b_k^{(1)} = \sum_{j=1}^{r_0} a_j^{(0)} w_{jk}^{(1)} + b_k^{(1)}, \\ a_k^{(1)} :=& g_1(z_k^{(1)}), \end{aligned}$$

the latter is referred to as the output of that node. Also denote $z^{(1)} = (z_1^{(1)}, z_2^{(1)}, \dots, z_{r_1}^{(1)})$ and $a^{(1)} = (a_1^{(1)}, a_2^{(1)}, \dots, a_{r_1}^{(1)})$. Alternatively, one may obtain $z^{(1)}$ via matrix operations

$$z^{(1)} = (a^{(0)})^T W_1 + B_1,$$

where T denotes the transpose.
We iterate so that at l-th layer ($l \in \{2, \ldots, n_h + 1\}$) we obtain

$$\begin{aligned} z_k^{(l)} &:= \langle a^{(l-1)}, W_l[:, k] \rangle + b_k^{(l)} = \sum_{j=1}^{r_{l-1}} a_j^{(l-1)} w_{jk}^{(l)} + b_k^{(l)}, \\ a_k^{(l)} &:= g_l(z_k^{(l)}) \end{aligned}$$

and similarly, denoting $z^{(l)} = (z_1^{(l)}, z_2^{(l)}, \ldots, z_{r_l}^{(l)})$, $a^{(l)} = (a_1^{(l)}, a_2^{(l)}, \ldots, a_{r_l}^{(l)})$. Alternatively, one may find $z^{(l)}$ via matrix operations

$$z^{(l)} = \left(a^{(l-1)}\right)^T W_l + B_l.$$

Finally we obtain the output from our network as

$$\widehat{y}_k = a_k^{(n_h+1)} \quad (k = 1, \ldots, m).$$

Remark 1.3.

- When initilising the network (before we train it with data) we set up the network with random weights (e.g. drawn from uniform distribution on $[-1, 1]$) and biases and adjust them later via, e.g., gradient descent so that prediction error is minimal (here one needs to run cross validation to avoid overfitting).
- Note that the range of activation function g_{n_h+1} needs to be consistent with the values of y.

1.2. **Backpropagation.**

Define the cost function for a single iteration K-th iteration

$$C_K = \sum_{i=1}^{m} (\widehat{y}_i - y_i)^2.$$

For brevity, in this section, we drop the subscript K so write $C \equiv C_K$. We wish to minimise C treated as a function of weights and biases. In order to that one may employ, e.g., gradient descent method, for that we would require to find $\frac{\partial C}{\partial w_{ij}^{(k)}}$ and $\frac{\partial C}{\partial b_i^{(k)}}$ for each $i \in \{1, \ldots, r_{k-1}\}$, $j \in \{1, \ldots, r_k\}$ and $k \in \{1, \ldots, n_h + 1\}$. The process of finding the above derivatives is referred to as backpropagation, as it is easy to find the derivatives with respect to weights and biases in the last layer, and then consider the ones from the previous layer and so on.

We use the chain rule to obtain

$$\frac{\partial C}{\partial w_{ij}^{(n_h+1)}} = \frac{\partial C}{\partial z_j^{(n_h+1)}} \frac{\partial z_j^{(n_h+1)}}{\partial w_{ij}^{(n_h+1)}}.$$

Note that since C is a multi-variable function, one would expect more terms in the chain rule, however these vanish as

$$\frac{\partial z_l^{(n_h+1)}}{\partial w_{ij}^{(n_h+1)}} = \begin{cases} a_i^{(n_h)} & \text{if } l = j \\ 0 & \text{if } l \neq j \end{cases}.$$

Let us denote

$$\delta_j^{(k)} := \frac{\partial C}{\partial z_j^{(k)}}.$$

We find that

$$\delta_j^{(n_h+1)} = 2(\widehat{y}_j - y_j)g'_{\mathsf{out}}(z_j^{(n_h+1)}).$$

Hence,

$$\frac{\partial C}{\partial w_{ij}^{(n_h+1)}} = 2a_i^{(n_h)}(\widehat{y}_j - y_j)g'_{\mathsf{out}}(z_j^{(n_h+1)}).$$

For all $k \in \{1, \dots, n_h\}$ we obtain

$$\begin{aligned}\frac{\partial C}{\partial w_{ij}^{(k)}} =& \frac{\partial C}{\partial z_j^{(k)}} \frac{\partial z_j^{(k)}}{\partial w_{ij}^{(k)}} \\ =& \delta_j^{(k)} \frac{\partial z_j^{(k)}}{\partial w_{ij}^{(k)}}.\end{aligned}$$

We have

$$\frac{\partial z_j^{(k)}}{\partial w_{ij}^{(k)}} = a_i^{(k-1)}$$

For $\delta_j^{(k)}$ we apply the chain rule, thus

$$\begin{aligned}\delta_j^{(k)} =& \sum_{l=1}^{r_{k+1}} \frac{\partial C}{\partial z_l^{(k+1)}} \frac{\partial z_l^{(k+1)}}{\partial z_j^{(k)}}, \\ =& \sum_{l=1}^{r_{k+1}} \delta_l^{(k+1)} \frac{\partial z_l^{(k+1)}}{\partial z_j^{(k)}}\end{aligned}$$

and

$$\frac{\partial z_l^{(k+1)}}{\partial z_j^{(k)}} = g'_k(z_j^{(k)}) w_{jl}^{(k+1)}.$$

Hence, for all $k \in \{1, \dots, n_h\}$ we have

$$\begin{aligned}\frac{\partial C}{\partial w_{ij}^{(k)}} =& g'_k(z_j^{(k)}) a_i^{(k-1)} \sum_{l=1}^{r_{k+1}} \delta_l^{(k+1)} w_{jl}^{(k+1)} \\ \frac{\partial C}{\partial w_{ij}^{(n_h+1)}} =& \underbrace{2(\widehat{y}_j - y_j)g'_{\mathsf{out}}(z_j^{(n_h+1)})}_{\delta_j^{(n_h+1)}} a_i^{(n_h)}.\end{aligned}$$

Similarly, taking into account that

$$\frac{\partial z_j^{(k)}}{\partial b_i^{(k)}} = 1$$

we obtain

$$\frac{\partial C}{\partial b_i^{(k)}} = g_k'(z_j^{(k)}) \sum_{l=1}^{r_{k+1}} \delta_l^{(k+1)} w_{jl}^{(k+1)},$$
$$\frac{\partial C}{\partial b_{ij}^{(n_h+1)}} = \underbrace{2(\widehat{y}_j - y_j) g_{\mathsf{out}}'(z_j^{(n_h+1)})}_{\delta_j^{(n_h+1)}}.$$

for all $k \in \{1, \dots, n_h\}$.

1.3. **Training the network.**

1.3.1. *Training via gradient descent.* Given a training data set (X, Y), where $X = (X_1, \dots, X_N)$, $Y = (Y_1, \dots, Y_N)$ and each $X_k \in \mathbb{R}^n$, $Y_k \in \mathbb{R}^m$ for all k. Let us denote the output of our network given input X_k to be $\widehat{y}_k \in \mathbb{R}^m$. Then in order to minimise

$$C(N) = \frac{1}{N} \sum_{i=1}^{N} \|\widehat{y}_i - Y_i\|^2 = \frac{1}{N} \sum_{i=1}^{N} C_i,$$

where $\|\cdot\|$ is a Euclidean norm in $\mathbb{R}^m$, we perform gradient descent (GD) and so use the results from backpropagation section, so that each we update the weights and biases as follows

$$w_{ij}^{(k),\star} = w_{ij}^{(k)} - \frac{\alpha}{N} \sum_{l=1}^{N} \frac{\partial C_l}{\partial w_{ij}^{(k)}},$$
$$b_i^{(k),\star} = b_i^{(k)} - \frac{\alpha}{N} \sum_{l=1}^{N} \frac{\partial C_l}{\partial b_i^{(k)}},$$

where $\alpha > 0$ is so-called learning rate, and $w_{ij}^{(k),\star}$, $b_i^{(k),\star}$, represent the new weights and biases that we should consider in our neural network.

We can iterate this process to get to the minimum as close as possible. We refer to each iteration of the above process as an epoch. To be more precise a number of epochs describes how man times the algorithm has gone through the entire training data set (full-batch)

1.3.2. *Training via stochastic gradient descent.* In stochastic gradient descent (SGD), rather than the gradient of the full-batch we use a gradient of a single (or a mini-batch of) the training data set and proceed as follows:

(1) randomly shuffle your training data set
(2) for l in $\{1, \dots, N\}$ update the weights and biases as follows:

$$w_{ij}^{(k),\star} = w_{ij}^{(k)} - \alpha \frac{\partial C_l}{\partial w_{ij}^{(k)}},$$
$$b_i^{(k),\star} = b_i^{(k)} - \alpha \frac{\partial C_l}{\partial b_i^{(k)}}.$$

An epoch is completed when you finish all N iterations. A mini-batch version of SGD proceeds as follows

(1) randomly shuffle your training data set.
(2) Divide your data set into mini-batches B_l of size K (assume that $K|N$) then for l in $\{1, \dots, \frac{N}{K}\}$ update the weights and biases as follows:

$$\begin{aligned} w_{ij}^{(k),\star} &= w_{ij}^{(k)} - \frac{\alpha}{|B_l|} \sum_{r \in B_l} \frac{\partial C_r}{\partial w_{ij}^{(k)}}, \\ b_i^{(k),\star} &= b_i^{(k)} - \frac{\alpha}{|B_l|} \sum_{r \in B_l} \frac{\partial C_r}{\partial b_i^{(k)}}. \end{aligned}$$

An epoch is completed when you finish all $\frac{N}{K}$ iterations.

SGD often converges much faster than GD however the cost function may not be minimised as well as in the GD. In modern ML due to faster computation, SGD is a preferred method.

References

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.