Deep Learning **Summer Term 2024**

http://ml3.leuphana.de/lectures/summer24/DL
Machine Learning Group, Leuphana University of Lüneburg

Soham Majumder (soham.majumder@leuphana.de)

Exercise 2

Due on: Thursday, 02.05.2024

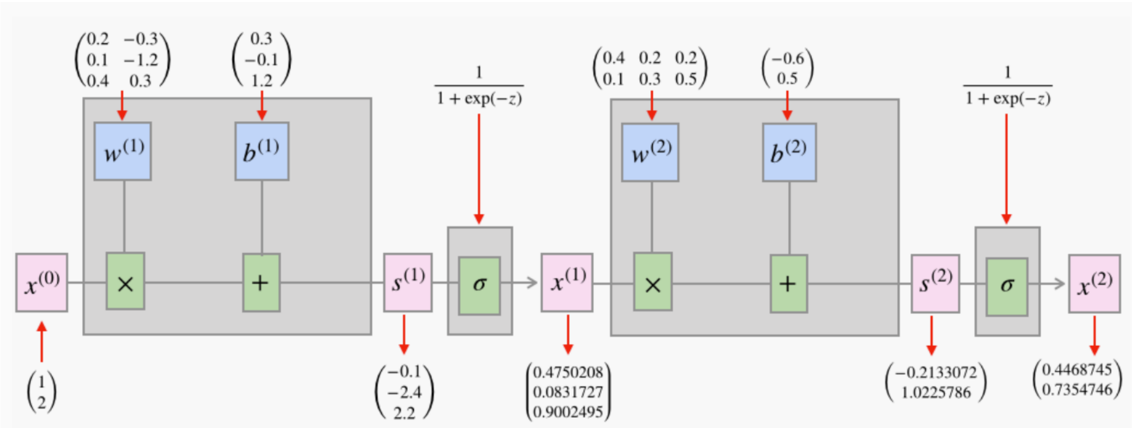

Figure 1: A simple neural network.

Task 5 A Simple Neural Network

Let $x^{(0)} = (1, 2)^\top \in \mathbb{R}^2$ be a data point and $y = (0, 1)^\top \in \mathbb{R}^2$ be the target. Consider the neural network with sigmoid activation functions depicted in Figure 1.

(i) Compute the forward pass of $x^{(0)}$ by hand. What is the value of $x^{(2)}$?

(ii) Compute the gradients with respect to all parameters in the network using backpropagation. For simplicity, you can use the squared loss function.

Solution

(i) Figure 1 now contains the (intermediate) result(s).

(ii) Let $E(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$ be the squared loss function. Its derivative is given by $\frac{\partial E}{\partial \hat{y}} = \hat{y} - y$. Here, $\hat{y}$ is the prediction, hence $x^{(2)}$ in our case. Remember, that for a sigmoid activation function, the derivative is $\sigma'(\cdot) = \sigma(\cdot)(1 - \sigma(\cdot))$.

$$
\begin{aligned}
\frac{\partial E}{\partial w^{(2)}} &= \underbrace{\frac{\partial E}{\partial x^{(2)}} \frac{\partial x^{(2)}}{\partial s^{(2)}}}_{\texttt{tmp}} \frac{\partial s^{(2)}}{\partial w^{(2)}} \\
&= \left(x^{(2)} - y\right) \sigma\left(s^{(2)}\right)\left(1 - \sigma\left(s^{(2)}\right)\right) x^{(1)\top} \\
&= \begin{pmatrix} 0.45 \\ -0.26 \end{pmatrix} \begin{pmatrix} 0.25 \\ 0.19 \end{pmatrix} \begin{pmatrix} 0.48 & 0.08 & 0.90 \end{pmatrix} \\
&= \begin{pmatrix} 0.05 & 0.01 & 0.10 \\ -0.02 & -0.00 & -0.05 \end{pmatrix} \\
\frac{\partial E}{\partial b^{(2)}} &= \underbrace{\frac{\partial E}{\partial x^{(2)}} \frac{\partial x^{(2)}}{\partial s^{(2)}}}_{\texttt{tmp}} \frac{\partial s^{(2)}}{\partial b^{(2)}} \\
&= \left(x^{(2)} - y\right) \sigma\left(s^{(2)}\right)\left(1 - \sigma\left(s^{(2)}\right)\right) 1 \\
&= \begin{pmatrix} 0.45 \\ -0.26 \end{pmatrix} \begin{pmatrix} 0.25 \\ 0.19 \end{pmatrix} \\
&= \begin{pmatrix} 0.11 \\ -0.05 \end{pmatrix} \\
\frac{\partial E}{\partial w^{(1)}} &= \underbrace{\frac{\partial E}{\partial x^{(2)}} \frac{\partial x^{(2)}}{\partial s^{(2)}}}_{\text{previous } \texttt{tmp}} \frac{\partial s^{(2)}}{\partial x^{(1)}} \frac{\partial x^{(1)}}{\partial s^{(1)}} \frac{\partial s^{(1)}}{\partial w^{(1)}} \\
&= \underbrace{\texttt{tmp} \frac{\partial s^{(2)}}{\partial x^{(1)}} \frac{\partial x^{(1)}}{\partial s^{(1)}}}_{\text{new } \texttt{tmp}} \frac{\partial s^{(1)}}{\partial w^{(1)}} \\
&= \texttt{tmp} w^{(2)\top} \sigma\left(s^{(1)}\right)\left(1 - \sigma\left(s^{(1)}\right)\right) x^{(0)\top} \\
&= w^{(2)\top} \texttt{tmp} \sigma\left(s^{(1)}\right)\left(1 - \sigma\left(s^{(1)}\right)\right) x^{(0)\top} \\
&= \begin{pmatrix} 0.4 & 0.1 \\ 0.2 & 0.3 \\ 0.2 & 0.5 \end{pmatrix} \begin{pmatrix} 0.11 \\ -0.05 \end{pmatrix} \begin{pmatrix} 0.25 \\ 0.08 \\ 0.09 \end{pmatrix} \begin{pmatrix} 1.00 & 2.00 \end{pmatrix} \\
&= \begin{pmatrix} 0.010 & 0.019 \\ 0.001 & 0.001 \\ 0.000 & -0.001 \end{pmatrix} \\
\frac{\partial E}{\partial b^{(1)}} &= \underbrace{\frac{\partial E}{\partial x^{(2)}} \frac{\partial x^{(2)}}{\partial s^{(2)}}}_{\text{previous } \texttt{tmp}} \frac{\partial s^{(2)}}{\partial x^{(1)}} \frac{\partial x^{(1)}}{\partial s^{(1)}} \frac{\partial s^{(1)}}{\partial b^{(1)}} \\
&= \frac{\partial E}{\partial x^{(2)}} \frac{\partial x^{(2)}}{\partial s^{(2)}} \frac{\partial s^{(2)}}{\partial x^{(1)}} \frac{\partial x^{(1)}}{\partial s^{(1)}} 1 \\
&= \begin{pmatrix} 0.010 \\ 0.001 \\ 0.000 \end{pmatrix}
\end{aligned}
$$

Task 6 A Neural Network Implementation

We now want to implement a neural network as a class in Python. Check the template `code2.py`.

Part 1

(i) Implement the ReLU and sigmoid activation functions and their derivatives.

(ii) The class `NeuralNetwork` already has two member functions `__init__` and `forward`. Fix the TODOs and

(a) initialize the weights via $w_{ij} \sim \mathcal{N}(0, \sigma^2)$, where $\sigma = \sqrt{\frac{2}{d_{\text{in}} + d_{\text{out}}}}$, d_{in} denotes the number of input neurons, and d_{out} denotes the number of output neurons of that layer

(b) initialize the biases as zeros

(c) implement the forward pass

Make sure to also save the intermediate results (e.g. $x^{(\ell)}$, $s^{(\ell)}$, see Figure 1) when computing the forward pass. We will need those later on.

Hint: You can test your implementation by overwriting the weights and biases with the values from Task 5 (i) and compare all intermediate results.

Part 2

(i) Implement backpropagation as a member function `backprop(self, x, y)`. It should compute the gradients of all parameters.

(ii) Implement a training function `train(self, x, y, eta=.01, iterations=100)` that implements (batch) gradient descent. You can use the squared loss if not stated otherwise.

(iii) Test your training procedure by fitting a binary classification problem on toy data and visualize the prediction appropriately. Make sure to use a sigmoid activation and a single neuron in the output layer.

Hint: You can test your implementation by comparing the results with the values from Task 5 (ii).

Solution

The code is provided as `solution2.py`.

Task 7 Approximating a Function Using ReLUs

In this task, we aim to approximate a function $f : [a, b] \to \mathbb{R}$ arbitrarily well using a neural network with a single hidden layer consisting of n neurons based on ReLU activations.

Consider a grid of n equidistant points $x_1, x_2, x_3, \dots, x_n$, with $a = x_1$ and $b = x_n$. Furthermore, the grid has a constant spacing of $h = x_{i+1} - x_i$ for $i = 1, 2, \dots, n-1$. We want to approximate f using a linear combination of ReLU functions given by

$$\mathrm{NN}(x) = \sum_{i=1}^{n} \lambda_i \cdot \mathrm{ReLU}(x - b_i),$$

where $\lambda_i \in \mathbb{R}$ are weights and the biases are given by $b_1 = a - h$, $b_2 = a$, $b_3 = a + h$, $\dots$, $b_n = b - h$.

Prior to b_1, all ReLU functions $\max(0, x - b_i)$ are zero. Between $b_1 = a - h$ and $b_2 = a = x_1$, only the first ReLU function $\max(0, x - b_1)$ can be non-zero. The function $\lambda_1 \cdot \max(0, x - b_1)$ is supposed to approximate f at x_1. If we subtract that function from f, we obtain a new function which the next ReLU function can approximate, and iterate to the end. Hence, we can approximate the function f on the interval $[a, b]$ using a sum of n ReLU functions.

To learn the parameters λ_i we can construct a linear system of equations

$$f(x_i) = \sum_{j=1}^{n} \lambda_j \cdot \mathrm{ReLU}(x_i - b_j) \quad \text{for} \quad i = 1, \dots, n.$$

By using $x_i = a + (i-1)h$ and $b_j = a + (j-2)h$, we obtain $x_i - b_j = (i - j + 1)h$. Considering i and j where the ReLU is positive,

$$f(x_i) = \sum_{j=1}^{i} \lambda_j (i - j + 1)h \quad \text{for} \quad i = 1, \dots, n. \tag{1}$$

Write down the linear system determined by Equation (1). Then, implement this approximation method, solve for $\lambda_1, \lambda_2, \dots, \lambda_n$, and experiment with various values of n. Check `code2.py`.

Solution

Equation (1) yields

$$\begin{aligned}
f(x_1) &= 1h\lambda_1 && = h(1\lambda_1) \\
f(x_2) &= 2h\lambda_1 + 1h\lambda_2 && = h(2\lambda_1 + 1\lambda 2) \\
f(x_3) &= 3h\lambda_1 + 2h\lambda_2 + 1h\lambda_3 && = h(3\lambda_1 + 2\lambda_2 + 1\lambda_3) \\
f(x_4) &= 4h\lambda_1 + 3h\lambda_2 + 2h\lambda_3 + 1h\lambda_4 && = h(4\lambda_1 + 3\lambda_2 + 2\lambda_3 + 1\lambda_4)
\end{aligned}$$

for the first values of i which can be written in the following linear system of equations:

$$h \begin{pmatrix} 1 & & & \\ 2 & 1 & & \\ \vdots & \ddots & \ddots & \\ n & \dots & 2 & 1 \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{pmatrix}$$

where $f_i = f(x_i)$ for $i = 1, \dots, n$. The parameters λ_i can be obtained by solving the system. Note that this approach can be generalized to non-equidistant data points. The approximation improves by increasing the number of functions n.

The code is provided as `solution2.py`.