Computational Intelligence WS24/25

Exercise Sheet 4 (Solution) — December 10th, 2024

Thomas Gabor, Maximilian Zorn, Claudia Linnhoff-Popien

1 Neural Networks

Consider the small neural network in Fig. 1, which we want to train for the *addition of any two positive inputs*, i.e., $\mathcal{N}(x) = \sum_i x_i$, where $x \in \mathbb{R}^2_+$. As the loss function we will utilize the error $SSE(x) = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$ for network output $\hat{y}_i$ given network input x_i and training data $\{(x_1, y_1), ..., (x_n, y_n)\}$. Assume that the first layer is activated with the *Rectified Linear* (ReL) function $ReL(x) = \max\{0, x\}$ and the second layer with the *linear* function $f(x) = x$.

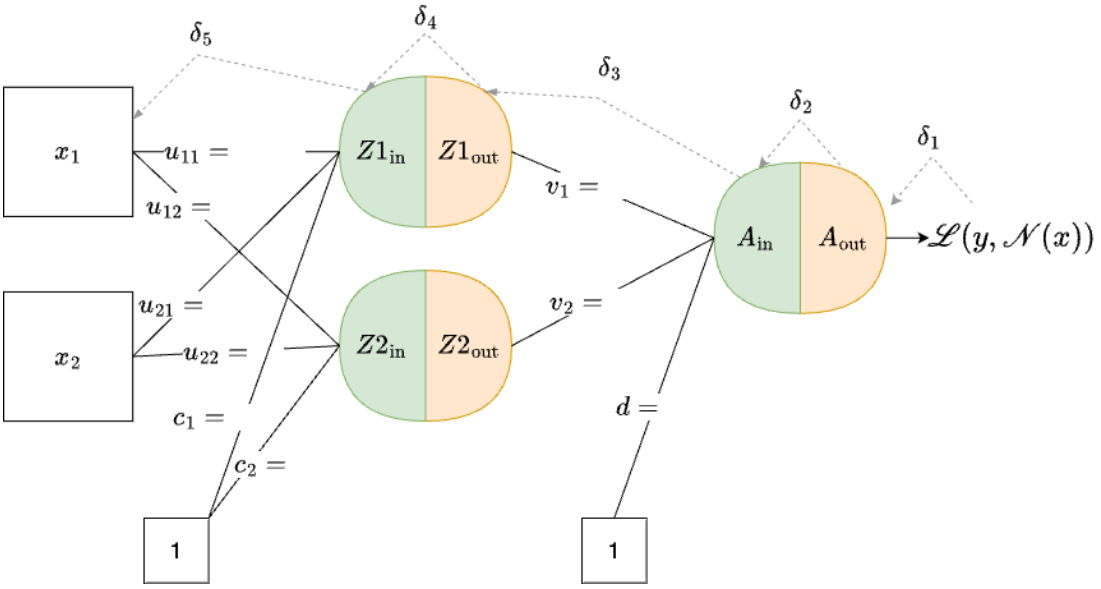


Figure 1: A simple shallow neural network with one output neuron.

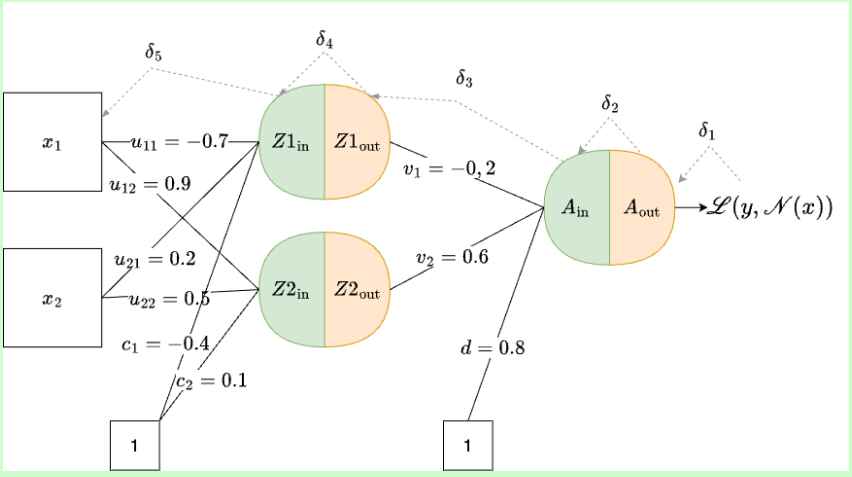
(i) Clarify the meaning of the variables, $x, y, u, v, c, d, p, q, Z, A$ and $\mathcal{L}$. Of which dimension are u, v, c, d in the above case?

$x \to$ input, $[x_1, x_2] \in \mathbb{R}^p$ – input dim $p = 2$
$y \to$ output, $[y_1] \in \mathbb{R}^q$ – output dim $q = 1$
$u, v \to$ weights for layers 1 and 2, where $u \in \mathbb{R}^{p \times |z|}$, $v \in \mathbb{R}^{|z| \times q}$
$c, d \to$ biases for layer 1 and 2, where $c \in \mathbb{R}^{|z|}$, $d \in \mathbb{R}^q$
$Z_{1,out} = ReL(Z_{1,in})$, where $z_{in} = \sum_{i=1}^{p} w_i \cdot x_i + c$
$A_{out} = A_{in} =$ output layer (with linear activation), $A_{in} = \sum_{i=1}^{|z|} u_i \cdot z_{out,i} + c$
$\mathcal{L} \to$ SSE loss (i.e., target) function $(Y - A_{out})^2$

(ii) Complete the neural network in Fig. 1 above with the given parameterization, then give $\mathbf{w}, \mathbf{b}, \overline{\mathcal{N}}$, $|L_1|$, and $|L_2|$. For notation refer to **Definition 8** in the lecture slides or definition sheet.

$$\mathbf{c} = \langle -0.4, 0.1 \rangle$$
$$\mathbf{d} = \langle 0.8 \rangle$$
$$E_{1,1,1} = -0.7,\ E_{1,1,2} = 0.2$$
$$E_{1,2,1} = 0.9,\ E_{1,2,2} = 0.5$$
$$E_{2,1,1} = -0.2\ E_{2,1,2} = 0.6$$

The parameterized network:



Per our Notation $|L_1| = 2$ and $|L_2| = 1$ and $\mathbf{w} = (u +\!\!+ v)$ and $\mathbf{b} = (c +\!\!+ d)$ so that

$$\overline{\mathcal{N}} = \mathbf{w} +\!\!+ \mathbf{b} = (u +\!\!+ v) +\!\!+ (c +\!\!+ d) = \langle -0.7, 0.9, 0.2, 0.5, -0.2, 0.6, -0.4, 0.1, 0.8 \rangle$$

(iii) Compute a forward pass for the input values $x_1 = x_2 = 1$ and the weights/biases you have added in Fig. 1. What is the loss for this input?

$$
\begin{aligned}
&Z_{in} = \mathbf{u}^\top \mathbf{x} + c \\
&= \begin{bmatrix} -0.7 & 0.2 \\ 0.9 & 0.5 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} -0.4 \\ 0.1 \end{bmatrix} = \begin{bmatrix} -0.7 \cdot 1 + 0.2 \cdot 1 + (-0.4) \\ 0.9 \cdot 1 + 0.5 \cdot 1 + 0.1 \end{bmatrix} = \begin{bmatrix} -0.9 \\ 1.5 \end{bmatrix}
\end{aligned}
$$

$$
\begin{aligned}
&Z_{out} = ReL(Z_{in}) \\
&= \max\{0, Z_{in}\} = \begin{bmatrix} ReL(-0.9) \\ ReL(1.5) \end{bmatrix} = \begin{bmatrix} 0 \\ 1.5 \end{bmatrix}
\end{aligned}
$$

$$
\begin{aligned}
&A_{in} = \mathbf{v}^\top \mathbf{Z_{out}} + d \\
&= \begin{bmatrix} -0.2 & 0.6 \end{bmatrix} \begin{bmatrix} 0 \\ 1.5 \end{bmatrix} + 0.8 = [(-0.2) \cdot 0 + 0.6 \cdot 1.5 + 0.8] = 1.7
\end{aligned}
$$

$$
\begin{aligned}
&A_{out} = \text{linear}(A_{in}) \\
&= 1.7
\end{aligned}
$$

$$\mathcal{L} = (2 - 1.7)^2 \approx 0.09$$

Now, to improve the network parameters we need to update the weight and bias values such that the next computation is closer to the target value. We could do this, e.g., via the *gradient descent* optimization after applying a step of the *backpropagation* algorithm, which we have covered in the lecture.

(iv) Give (not compute) the backpropagation chain for the weight u_{11} in the Leibniz notation of the chain rule

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx},$$

for a variable z that depends on the variable y, which itself depends on the variable x. In other words: Formulate the backpropagation chain $\delta_1, \ldots, \delta_5$ in Fig. 1 in terms of the network building blocks ($\mathcal{L}, A_{out}, A_{in}, \ldots$).

For the gradient of weight u_{11} the backpropagation chain of differentiation extends to:

$$\frac{\delta \mathcal{L}(y, \mathcal{N}(x))}{\delta u_{11}} = \frac{\delta \mathcal{L}(y, \mathcal{N}(x))}{\delta A_{out}} \cdot \frac{\delta A_{out}}{\delta A_{in}} \cdot \frac{\delta A_{in}}{\delta Z_{1,out}} \cdot \frac{\delta Z_{1,out}}{\delta Z_{1,in}} \cdot \frac{\delta Z_{1,in}}{\delta u_{11}},$$

(v) (Bonus) Think about why this algorithm is called backpropagation? What makes

this approach advantageous? Give a brief argument. (Hint: Think about the order of computation and efficiency of re-computation.)

The big advantage of computing the gradients in reverse is the fact that you can (a) cache all computation results of the forward pass to use for the differentiation and (b) you can also cache and reuse the gradient results of previously differentiated layers/neurons. Take, for example, the differentiation chain for weight w_{11}

$$\frac{\delta \mathcal{L}(y, \mathcal{N}(x))}{\delta u_{11}} = \frac{\delta \mathcal{L}(y, \mathcal{N}(x))}{\delta A_{out}} \cdot \frac{\delta A_{out}}{\delta A_{in}} \cdot \frac{\delta A_{in}}{\delta Z_{1,out}} \cdot \frac{\delta Z_{1,out}}{\delta Z_{1,in}} \cdot \frac{\delta Z_{1,in}}{\delta u_{11}},$$

which already computes most of the gradient for weight u_{21}

$$\frac{\delta \mathcal{L}(y, \mathcal{N}(x))}{\delta u_{21}} = \frac{\delta \mathcal{L}(y, \mathcal{N}(x))}{\delta A_{out}} \cdot \frac{\delta A_{out}}{\delta A_{in}} \cdot \frac{\delta A_{in}}{\delta Z_{1,out}} \cdot \frac{\delta Z_{1,out}}{\delta Z_{1,in}} \cdot \frac{\delta Z_{1,in}}{\delta u_{21}},$$

and so on, resulting in substantial gains for big ('deep') NNs. This simple algorithm was first applied to neural networks way back in 1985 and is *still* one of the biggest breakthroughs in deep learning.

2 PyTorch

The `pytorch`[1] Python library focuses on *differentiable computing* of neural networks through *tensors*, two key components which enable its application towards modern *machine learning* (ML). In this exercise we will work with `pytorch` to realize and expand the above neural network formalization.

(i) After installing the `pytorch` library, implement and train the simple addition network from the previous task to correctly add any two positive input floats. The `addition_net.py` file will help you get started by providing TODOs for the following ML primitives:

1. Creating a *neural network* (in this case with the structure from above)
2. Initializing the `SDG` *optimizer* for the parameter of this network
3. Computing the *forward pass* of an input through the network
4. Using the output to calculate a *loss* and *backward pass* for the given input
5. Using an *optimizer step* to *update* the network's parameter

Hint: All required classes are already imported.

[1] For installation see `https://pytorch.org/get-started/locally/`. You can refer to `https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html` for a simple tutorial.

(ii) Like the learning rate α there are many *hyperparameters* that could be adjusted to improve (or worsen) the training process. Name a few. For instance, how do you observe the learning process to proceed with other values for α? Briefly summarize your observations.

Example implementation of the TODOs and a short ablation of the α parameter can be found in `addition_net_solution.py`.

Here we can observe that the value of α alone can determine the convergence of the training process, from $\alpha = 0.1$ (almost instantly), $\alpha = 0.5$ (very unsteady, slow) to $\alpha = 0.9$ (very slow progress, almost random).

(iii) We now have all the tools to replicate the data classifier we have seen on the *Tensorboard Playground* in the lecture. With the ML building blocks we have constructed in task 2 (i), fill in the TODOs in the `classifier_net.py` to learn to classify the "double ring" dataset.

Example implementation of the TODOs see `classifier_net_solution.py`.