Assignment 3

Numerical Optimization / Optimization for CS WS2023

Lea Bogensperger, `lea.bogensperger@icg.tugraz.at`
Alexander Falk, `alexander.falk@icg.tugraz.at`
Clemens Krenn, `clemens.krenn@student.tugraz.at`
Fabian Ofner, `f.ofner@student.tugraz.at`

December 5, 2023

Deadline: Jan 9th, 2024 at 12:00

Submission: Upload your report and your implementation to the TeachCenter. Please use the provided framework-file for your implementation. Make sure that the total size of your submission does not exceed 50MB. Include **all** of the following files in your submission:

- `report.pdf`: This file includes your notes for the individual tasks. Keep in mind that we must be able to follow your calculation process. Thus, it is not sufficient to only present the final results. You are allowed to submit hand written notes, however a compiled LaTeX document is preferred. In the first case, please ensure that your notes are well readable.
- `main.py`: This file includes your python code to solve the different tasks. Please only change the marked code sections. Also please follow the instructions defined in `main.py`.
- `figures.pdf`: This file is generated by running `main.py`. It includes a plot of all mandatory figures on separate pdf pages. Hence, you do not have to embed the plots in your report.
- `model.json`: This file includes your learned parameters θ which are necessary to reproduce the obtained results.

1 Classification with Neural Networks

The task of this assignment is to classify 2D points into $C = 5$ classes, which are distributed as shown in Figure 1 for the training data. As these classes are non-linearly separable, we can solve this using a small feedforward neural network and a dataset comprised of 400 training samples and 100 test samples. Each sample is given as a tuple $(\mathbf{x}^s, y^s)$, where $\mathbf{x}^s = (x_1^s \; x_2^s)^\top \in \mathbb{R}^2$ is the input and $y^s \in \{0, 1, \dots, C-1\}$ is the target label of the sth sample.

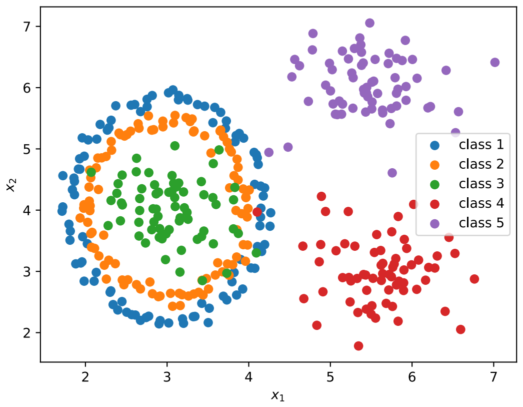

Figure 1: 2D training data set with $C = 5$ classes.

Layer	**Neurons**	**Activation**
Input	N_I=?	-
Hidden	N_H=12	Softplus
Output	N_O=?	Softmax

Table 1: The architecture of the network.

2 Network Architecture

To achieve this task, we use a fully connected neural network with 1 hidden layer. In neural networks, the term fully connected means that every neuron from the previous layer is connected to every neuron of the subsequent layer. Using this approach, classifying multiple classes is typically performed in a lifted space that represents discrete probability distributions over the labels. Thus, the network predicts for *each* label how likely it is.

To achieve this goal, we set up a neural network f that maps an input $\mathbf{x} \in \mathbb{R}^{N_I}$ (N_I is the input dimension) to the probability simplex $\{\mathbf{y} \in \mathbb{R}^{N_O} : y_i \geq 0, \sum_{j=1}^{N_O} y_i = 1\}$ (N_O is the output dimension). Note that N_O for classification is governed by the number of classes C as each output node should contain the probability that an input sample $\mathbf{x}^s$ belongs to the respective class. The hidden layer has N_H neurons. The function $f(\mathbf{x}, \theta)$ uses a set of weights and biases i.e. the *learnable* parameters that have to be optimized such that a meaningful mapping can be learned. The learnable network parameters are $\theta = \{\mathbf{W}^{(0)}, \mathbf{b}^{(0)}, \mathbf{W}^{(1)}, \mathbf{b}^{(1)}\}$. The details of the architecture can be found in Table 1.

The evaluation of the network at a given point $\mathbf{x}$ is referred to as the so-called *forward pass*. In our setting it is given as

$$f(\mathbf{x}, \mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(0)}, \mathbf{b}^{(0)}) = \phi\left(\mathbf{W}^{(1)}\sigma\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right) + \mathbf{b}^{(1)}\right). \tag{1}$$

By convention, we will denote the linear output of a layer l by $\mathbf{z}^{(l)}$, whereas applying a non-linear function to this output (the activation function) yields $\mathbf{a}^{(l)}$ (see Figure 3). Also, we use $\mathbf{x} = \mathbf{z}^{(0)} = \mathbf{a}^{(0)}$ for the input. Thus, these two entities can be established for all layers. See also Figure 2

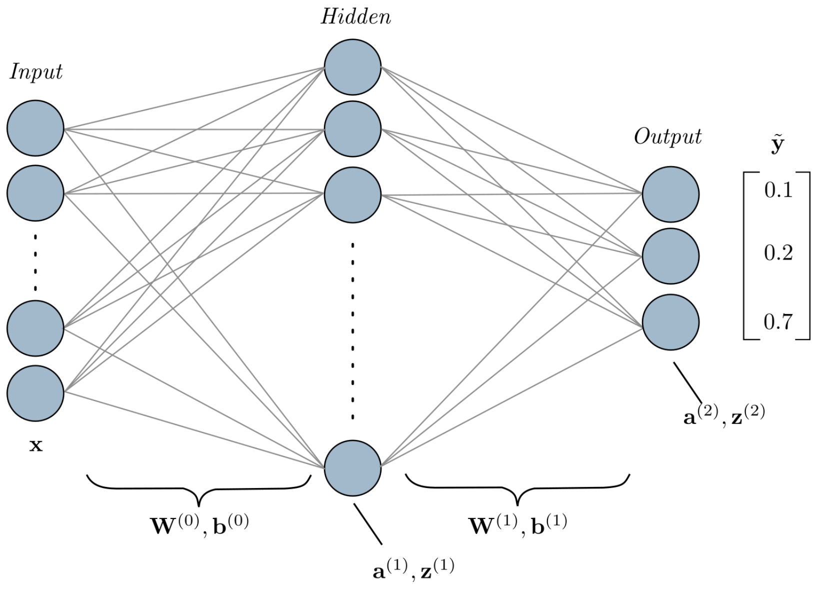
Hidden
Input
Output
$\tilde{\mathbf{y}}$
0.1
0.2
0.7
$\mathbf{a}^{(2)}, \mathbf{z}^{(2)}$
$\mathbf{x}$
$\mathbf{W}^{(0)}, \mathbf{b}^{(0)}$
$\mathbf{W}^{(1)}, \mathbf{b}^{(1)}$
$\mathbf{a}^{(1)}, \mathbf{z}^{(1)}$

Figure 2: Fully connected neural network with 1 hidden layer.

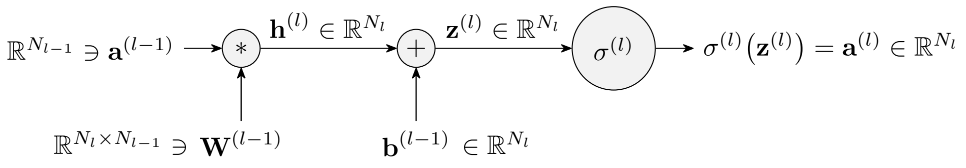
$\mathbb{R}^{N_{l-1}} \ni \mathbf{a}^{(l-1)}$
$*$
$\mathbf{h}^{(l)} \in \mathbb{R}^{N_l}$
$+$
$\mathbf{z}^{(l)} \in \mathbb{R}^{N_l}$
$\sigma^{(l)}$
$\sigma^{(l)}(\mathbf{z}^{(l)}) = \mathbf{a}^{(l)} \in \mathbb{R}^{N_l}$
$\mathbb{R}^{N_l \times N_{l-1}} \ni \mathbf{W}^{(l-1)}$
$\mathbf{b}^{(l-1)} \in \mathbb{R}^{N_l}$

Figure 3: Forward pass in arbitrary fully connected layer l with nonlinear activation function $\sigma^{(l)}$.

The activation function σ is identified with the component-wise application of the well-known *Softplus function* to the pre-activation of the input layer $\mathbf{z}^{(1)}$

$$\mathbf{a}^{(1)} = \sigma\big(\mathbf{z}^{(1)}\big) \iff a_i^{(1)} = \ln\Big(1 + \exp z_i^{(1)}\Big).$$

The output of the network $\mathbf{a}^{(2)}$ is computed by applying the *Softmax function* to elements of the pre-activation of the output layer $\mathbf{z}^{(2)}$

$$\mathbf{a}^{(2)} = \phi\big(\mathbf{z}^{(2)}\big) \iff a_i^{(2)} = \frac{\exp z_i^{(2)}}{\sum_{j=1}^{N_O} \exp z_j^{(2)}}\,.$$

The elements of the output of the softmax function are all positive and sum up to one. Thus, the output of the neural network $\mathbf{a}^{(2)}$ can indeed be interpreted as a discrete probability distribution over all labels. The actual prediction of the network (the most likely label) is then defined as

$$\widetilde{y} = \arg\max_i a_i^{(2)}.$$

3 Optimization with Gradient Descent

Assume we have given a training data set as described in Section 1. First, we convert each ground-truth label y^s into a discrete target probability distribution $\mathbf{y}^s$. For example let $y^s = 3$, then $\mathbf{y}^s = (0,0,0,1,0)^\top$. Then, the training process consists of adapting the network parameters θ such that the output of the network $\mathbf{a}^{(2)} = f(\mathbf{x}^s, \theta)$ is as close as possible to the ground truth $\mathbf{y}^s$ for all samples. This can be achieved by minimizing a cost function through backpropagation. A common loss function in multi-class classification is the cross-entropy loss which is defined as

$$l(\mathbf{a}^{(2)s}, \mathbf{y}^s) = -\sum_{i=1}^{N_O} y_i^s \ln(a_i^{(2)s})\,. \tag{2}$$

The total loss for the whole dataset is simply the average of each sample-loss l across the dataset , i.e.

$$\mathcal{L}(\theta) = \frac{1}{S}\sum_{s=1}^{S} l(\mathbf{a}^{(2)s}, \mathbf{y}^s)\,.$$

In each *training* iteration the objective is given by minimizing the cost function $\mathcal{L}$. Each iteration is comprised of a *forward* and a *backward* pass. During the forward pass the total loss $\mathcal{L}$ is computed for all S samples. In the backward pass, all network parameters are updated through a gradient method based on the loss $\mathcal{L}$. This is referred to as *backpropagation*.

The simplest option to perform the backpropagation step is to apply the steepest descent algorithm, where you need to compute the gradient of the total loss w.r.t. to each learnable parameter. Let $\mathbf{p}$ represent a network parameter, $\mathbf{p} \in \theta$. The k^{th} update step of the steepest descent algorithm for the parameter $\mathbf{p}$ with step size t^k reads as

$$\mathbf{p}^{k+1} = \mathbf{p}^k - t^k \nabla\mathcal{L}(\mathbf{p}^k),$$

where $\nabla\mathcal{L}(\mathbf{p}^k)$ is the gradient of the total loss w.r.t. $\mathbf{p}$ at the position $\mathbf{p}^k$

$$\nabla\mathcal{L}(\mathbf{p}^k) = \left.\frac{\partial\mathcal{L}}{\partial\mathbf{p}}\right|_{\mathbf{p}^k}.$$

4 Computing Gradients using Automatic Differentiation (10 P.)

With Pen & Paper:

1. Complete the two missing entries in Table 1.
2. Given the architecture described in Table 1 and the forward pass in Eq. (1), state the dimensions of the learnable parameters θ and the total number of learnable parameters.
3. Using the convention of $\mathbf{z}^{(l)}$ and $\mathbf{a}^{(l)}$, state them for the layers $l = 0$ (input), $l = 1$ (hidden) and $l = 2$ (output).

4. The task is now to draw the complete *computational graph* associated with the loss in (2) [1]. It receives inputs $\mathbf{x}$ and target labels y from the training data set and leaf variables θ (i.e. the learnable parameters) such that it can then output the loss in (2). All intermediate nodes are thus constructed of the "basic" operations of the forward pass – where we define "basic" as

   - matrix-vector multiplications,
   - vector additions/subtractions,
   - applications of non-linear functions (activations and loss function).

   Helper variables $\mathbf{h}^{(l)} = \mathbf{W}^{(l-1)}\mathbf{a}^{(l-1)}$ for $l = 1$ and $l = 2$ should be used, such that *each node* has a corresponding variable assigned to it.

5. Compute the gradient of the loss function (the output node of your computational graph) w.r.t. θ using the reverse mode of automatic differentiation. Follow the exercise notes for this. Check and state all dimensions of the resulting expressions.

In Python:

6. Implement the computed gradients and check all dimensions.
7. Implement the forward pass of the neural network.
8. Perform a gradient check of your implemented gradient using `scipy.optimize.approx_fprime`.

5 Training the Neural Network (10 P.)

In this section we explicitly work only with *training* data that is already loaded in the framework. The task is to use the training data to learn the parameters of the neural network which is later evaluated on the *test* set.

In Python:

1. Initialize the parameters using a uniform distribution with $\mathbf{p}^i \sim \mathcal{U}(-v, v)$, where $v = \frac{1}{\sqrt{N_I}}$ for $l = 0$ and $v = \frac{1}{\sqrt{N_H}}$ for $l = 1$.
2. Select a suitable, constant learning rate and report it. Train the network with steepest descent for a chosen, fixed number of iterations and log the training error. Note that this might require a bit of manual tuning.
3. Plot the training error over the iterations in a separate plot. Use a y-logarithmic scale for the loss plot[2]. Moreover, state the final accuracy obtained on the training data, which is defined as the average number of correctly classified samples

   $$\mathcal{A} = \frac{1}{S}\sum_{s=1}^{S}\delta(y^s, \widetilde{y}^s) \quad \text{with } \delta(u, v) = \begin{cases} 1 & \text{if } u = v \\ 0 & \text{else,} \end{cases}$$

   where δ is the indicator function.
4. Save your learned parameters θ and upload them.

6 Inference (5 P.)

Finally, we work also with *test* data to evaluate the performance of the neural network on unseen data.

In Python:

1. Apply the learned network to the *test data* and report the accuracy and the test error.
2. Another tool to evaluate the accuracy of a classifier is to set up a confusion matrix. For a multi-class setting this yields a matrix $M \in \mathbb{R}^{C\times C}$, where each element m_{ij} reports the number of observation for which a sample

[1] Note that this differs from Figure 2, which only displays the schematic of the type of neural network that we are using.
[2] `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.semilogy.html`

of ground truth class i is classified as class j, thus it is defined by

$$m_{ij} = \sum_{s=1}^{S} \delta(y^s = i, \widetilde{y}^s = j).$$

3. Plot the decision regions of the neural network by evaluating its prediction in an evenly spaced grid covering the range space of the training data. Use `numpy.meshgrid()` for this, and utilize `matplotlib.pyplot.imshow()` for visualization. Each pixel should be colored according to the predicted class and pixelwise alpha blending values should be taken from the network's output $\mathbf{a}^{(2)}$. As interpolation mode, choose `"nearest"`.