Assignment 3

Numerical Optimization / Optimization for CS WS2022

Lea Bogensperger, lea.bogensperger@icg.tugraz.at
Nives Križanec, nives.krizanec@student.tugraz.at
Jakob Ederer, ederer@student.tugraz.at

December 6, 2022

Deadline: Jan 10$^{\text{th}}$, 2023 at 23:59

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 given task. 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 given task.
- `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.

1 Multi-Class Classification with Neural Networks

The task of this assignment is to classify 2D points into 3 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 feed-forward neural network and a dataset comprised of 360 training samples and 90 test samples.

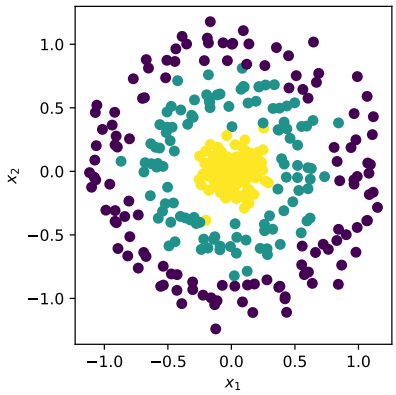

Figure 1: Training data of 2D point dataset. The 3 colors indicate the 3 classes.

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 $y^s \in \{0, 1, 2\}$ is the target label of the sth sample.

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 setup a neural network f that maps an input $\mathbf{x} \in \mathbb{R}^{N_I}$ (N_I is the input dimension) and the learnable network parameters ($\mathbf{W}^{(0)}$, $\mathbf{b}^{(0)}$, $\mathbf{W}^{(1)}$, $\mathbf{b}^{(1)}$) to the probability simplex $\{\mathbf{y} \in \mathbb{R}^{N_O} : y_i \geq 0, \sum_{j=0}^{N_O} y_i = 1\}$ (N_O is the output dimension), i.e.

$$f(\mathbf{x}, \mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(0)}, \mathbf{b}^{(0)}) \coloneqq g\left(\mathbf{W}^{(1)} h\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right) + \mathbf{b}^{(1)}\right).$$

The activation function h applies the well-known sigmoid function which is applied to each element of the pre-activation of the input layer $\mathbf{z}^{(1)} = \mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)} \in \mathbb{R}^{N_H}$ (N_H is the hidden dimension)

$$\mathbf{a}^{(1)} = h(\mathbf{z}^{(1)}) \iff a_i^{(1)} = \frac{1}{1 + \exp(-z_i^{(1)})}.$$

The output of the network $\widetilde{\mathbf{y}} \in \mathbb{R}^{N_O}$ is eventually computed by applying the softmax function to the pre-activation of the output layer $\mathbf{z}^{(2)} = \mathbf{W}^{(1)}\mathbf{a}^{(1)} + \mathbf{b}^{(1)} \in \mathbb{R}^{N_O}$

$$\widetilde{\mathbf{y}} = g(\mathbf{z}^{(2)}) \iff \widetilde{y}_i = \frac{\exp(z_i^{(2)})}{\sum_{j=1}^{N_O} \exp(z_j^{(2)})}.$$

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

$$\widetilde{y} = \arg\max_i \widetilde{y}_i,$$

i.e. the most likely label. See Figure 2 for a complete overview of the network structure.

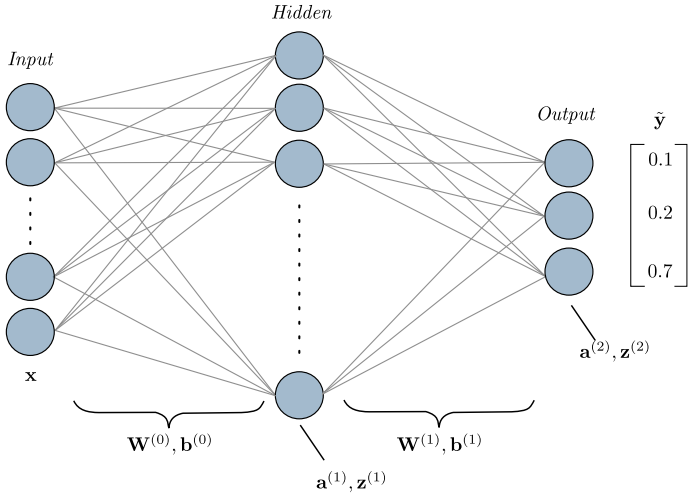

Figure 2: Structure of a feed-forward neural network with one hidden layer.

3 Optimization

Assume we have given a set of S input-output samples $\{(\mathbf{x}^s, y^s) : s = 1, \ldots, S\}$, where $\mathbf{x}^s \in \mathbb{R}^2$ is an input sample and $y^s \in \{0, 1, 2\}$ is the corresponding ground-truth label. First, we convert each ground-truth label y^s into a discrete target probability distribution $\mathbf{y}^s$. For example let $y^s = 2$, then $\mathbf{y}^s = (0, 0, 1)^\top$. Then, the training process consists of adapting the parameters of the network $\{\mathbf{W}^{(0)}, \mathbf{b}^{(0)}, \mathbf{W}^{(1)}, \mathbf{b}^{(1)}\}$ such that the output of the network $\widetilde{\mathbf{y}}^s = f(\mathbf{x}^s, \mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(0)}, \mathbf{b}^{(0)})$ is as close as possible to the ground-truth distribution $\mathbf{y}^s$ for all samples. This

can be achieved by minimizing a cost function through back-propagation. A common loss function in multi-class classification is the cross-entropy loss which is in our case defined as

$$l(\widetilde{\mathbf{y}}^s, \mathbf{y}^s) = -\sum_{i=1}^{N_O} y_i^s \ln(\widetilde{y}_i^s)\,.$$

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

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

In each iteration of the *training* process 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 L is computed for the all S samples. In the backward pass, all network parameters are updated through a gradient method based on the loss L. This is referred to as *back-propagation*.

3.1 Steepest Descent Algorithm

The simplest option to perform the back-propagation 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 network parameter as presented in the exercise lecture. Let $\mathbf{p} \in \{\mathbf{W}^{(0)}, \mathbf{b}^{(0)}, \mathbf{W}^{(1)}, \mathbf{b}^{(1)}\}$ represent a network parameter. 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 $\mathbf{p}^k$

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

3.2 Step Size Selection

An obvious choice for t^k would be setting it to a *constant*, i.e. $t^k = \alpha$ for all $k = 0, 1, ..., K-1$.

A more sophisticated step size selection procedure, however, is the *Armijo rule*, which uses backtracking to compute t^k in each iteration. Given an initial step size α, a very small $\sigma \in [10^{-5}, 10^{-1}]$ and a reduction factor $\beta \in [0.1, 0.5]$, we increase $m_k = 0, 1, ...$ leading to the step size $t^k = \alpha\beta^{m_k}$ until

$$\mathcal{L}(\mathbf{p}^k) - \mathcal{L}(\mathbf{p}^k - t^k\nabla\mathcal{L}(\mathbf{p}^k)) \geq \sigma t^k \nabla\mathcal{L}(\mathbf{p}^k)^T \nabla\mathcal{L}(\mathbf{p}^k).$$

3.3 Weight Decay

When training neural networks, we sometimes encounter the issue of over-fitting, indicating that the network tends to produce very good predictions on the training set while net being able to generalize well to unseen test data. This can be alleviated using additional regularization terms such as weight decay, where the loss function is augmented with

$$\mathcal{L}_w = \frac{1}{S}\sum_{s=1}^{S} l(\widetilde{\mathbf{y}}^s, \mathbf{y}^s) + \lambda(\|\mathbf{W}^{(0)}\|_2^2 + \|\mathbf{W}^{(1)}\|_2^2)$$

where $\lambda \in \mathbb{R}^+$ is a very small, positive parameter to balance the loss terms.

4 Tasks (25 P.)

Pen & Paper:

- For the given dataset, what is the size of the matrices $\mathbf{W}^{(0)}$, $\mathbf{W}^{(1)}$, and the bias terms $\mathbf{b}^{(0)}$ and $\mathbf{b}^{(1)}$? Specify the number of learnable parameters of the neural network as a function of N_H.

- Compute the derivative of the total loss w.r.t. to all model parameters using the results from the practical exercise session.

In Python:

- Initialize the parameters using a uniform distribution with $\mathbf{p}^i \sim \mathcal{U}(-v, v)$, where $v = \frac{1}{\sqrt{N_I}}$ for $\mathbf{W}^{(0)}$, $\mathbf{b}^{(0)}$ and $v = \frac{1}{\sqrt{N_H}}$ for $\mathbf{W}^{(1)}$, $\mathbf{b}^{(1)}$.
- Implement the forward pass of the network.
- Implement the backward pass by computing the gradients w.r.t. the model parameters.
- With one data sample verify your gradients using scipy's `approx_fprime()`[1].
- Implement the steepest descent algorithm. Experiment with 2 different choices for the step size parameter using:
  - a constant step size,
  - backtracking using Armijo's rule.
- For both versions: train the neural network using the provided *training data* for 5000 iterations with the steepest descent algorithm for $N_H = 12$. For the constant step size $t^k = 1$ is a good starting point, whereas for Armijo's rule, set the initial step size to $\alpha = 10$.
- Compare the training error and the training accuracy over the iterations in a separate plot each. Use a y-logarithmic scale for the loss plot[2]. The accuracy 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.
- Apply the learned network to the *test data* and report the accuracy and the test error to compare step size versions in the gradient method.
- Discuss the plots regarding the convergence rate of both gradient methods. State and interpret the final accuracy w.r.t. to the test set.
- For both methods, plot the decision boundary in an evenly spaced grid covering the range space of the training data. Use `numpy.meshgrid()` for this, and evaluate the class prediction for each point.
- **Bonus Task: Weight Decay (+2.5 points)**
  - Compute the derivative $\mathcal{L}_w$ w.r.t. the model parameters and adapt the gradient update in the code. Set $\lambda = 0.001$.
  - Retrain your model using both the constant and Armijo step size selection. Report the test errors and accuracies and plot the decision boundaries again. Discuss your findings.

5 Framework

Use the provided python file `assignment3.py` for your implementation. It already provides the functionality to load the train and test data. For this task, you are only allowed to use the `numpy`, `scipy`, and `matplotlib` packages.

Show and describe all your results in the report!

[1] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.approx_fprime.html`
[2] `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.semilogy.html`