Optimization for Computer Science
Assignment 3

November 8, 2016

Submission: Upload your implementation and report zipped (`MatrNr.zip`) to the TU-Graz TeachCenter using the MyFiles extension.
Deadline: November 22, 2016 at 23:58h.

1 Regression with Neural Networks

In this exercise we want to learn a complicated non-linear mapping with a neural network. This problem is an instance of *function approximation*, more specifically, we are interested in constructing a function $y = f(x)$ that best explains a set of data samples $\{x^c, y^c,\ c = 1 \dots C\}$. Neural networks have shown to be very effective for this kind of task.

We will use a non-linear architecture called multilayer perceptron (MLP). A MLP is a feed-forward network with one hidden layer[1], which uses non-linear activation functions. More specifically, each unit consists of a non-linear function $\varphi\ :\ \mathbb{R} \to \mathbb{R}$ that operates on a linear transformation of the input.

Let $x \in \mathbb{R}^n$ be an input vector, a hidden unit in the MLP is given by

$$\tilde{x}_i = \varphi\left(b_i^s + \sum_{j=1}^{n} w_{ij}^s x_j\right), \tag{1}$$

where w^s, b^s are the so-called weights and biases of the s-th layer (or stage). The non-linear function φ is typically chosen as the logistic function

$$\varphi(u) = \frac{1}{1 + e^{-u}} \tag{2}$$

1.1 Network Architecture

Let $y \in \mathbb{R}^2$ and $x \in \mathbb{R}^3$. We are interested in regressing a function $f\ :\ \mathbb{R}^3 \to \mathbb{R}^2$, i.e. to learn the optimal dimensionality-reduction mapping from a set of data samples $\{x^c, y^c\}$.

We will use a fully connected network with one hidden layer as depicted in fig. 1. The input layer takes a vector $x \in \mathbb{R}^3$. Each hidden unit computes a component of the vector $\tilde{x}$ according to eq. (1). The output is a vector $y \in \mathbb{R}^2$, hence the output layer has 2 units.

[1] "hidden" refers to the units of the network not directly connected to the inputs or the outputs.

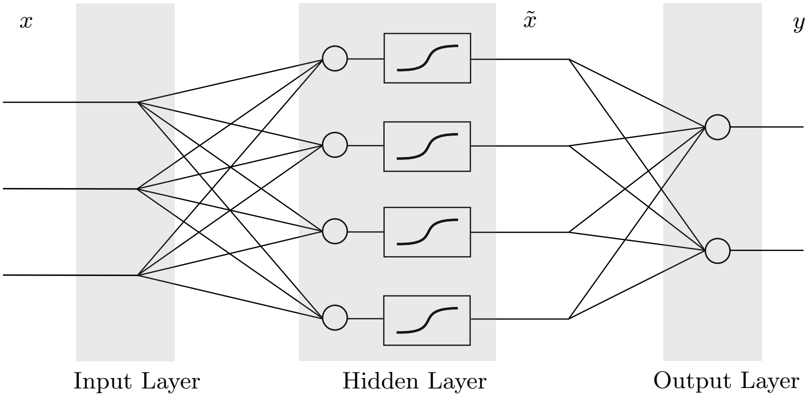

Figure 1: A network with one hidden layer consisting of 4 units.

Because we are interested in regression, the output units do not have the non-linear activation function, but apply just a linear transformation on their input

$$y_l = b_l^s + \sum_{k=1}^{i} w_{lk}^s \tilde{x}_k \tag{3}$$

Note that the dimensionality of the intermediate vector $\tilde{x}$ depends on the number of units in the hidden layer. In our approach, this will be a free hyper-parameter of the architecture.

Clearly if we have an input vector x_0 and a fixed set of parameters $p = \{w^s, b^s,\ s = 1, 2\}$ (i.e. a vector of all weights and biases in the network), we can describe the network by $y = h(x_0, p)$. Given a number of input-output pairs $\{z^c, t^c,\ c = 1 \dots C\}$, the training process consists of adapting the parameters p such that the output of the network matches t^c for an input z^c. A common way to do this is to minimize the sum of the squared errors (or "loss")

$$L(p; z, t) = \frac{1}{C} \sum_{c=1}^{C} \frac{1}{2} \|e^c(p; z^c, t^c)\|^2 = \frac{1}{C} \sum_{c=1}^{C} \frac{1}{2} \|h(z^c, p) - t^c\|^2 \tag{4}$$

over the parameters p. Here, $e^c(p; z^c, t^c) = h(z^c, p) - t^c$ is the error of the c-th data sample. For the next section we are interested in the loss as a function of the parameters, hence we will drop the inputs z, t and denote the loss function and the error as $L(p)$ and $e^c(p)$ respectively to avoid clutter.

1.2 Training

We will train the network by means of gradient descent (see Alg. 1). For this purpose, we have to compute a descent direction d^k. The most simple choice is to set

$$d^k = -\frac{1}{C} \sum_{c=1}^{C} \nabla L^c(p^k), \tag{5}$$

i.e. the descent direction is the normalized negative gradient over all training samples. This choice results in the well-known steepest descent scheme. The basic version of steepest descent is prone to slow convergence, especially if the dimensionality of the feature space is

Data: Choose $p^0 \in \mathbb{R}^n$ and iterate for $k \geq 0$ and a stepsize $\alpha > 0$

while *True* **do**
    compute a descent direction d^k;
    $p^{k+1} = p^k + \alpha d^k$;
    if *stopping criterion* **then**
        exit
    end
end

Algorithm 1: Gradient descent for training the network.

large and the energy is badly behaved (e.g. elongated level lines). A simple improvement is the Gauss-Newton method, which is obtained by choosing

$$d^k = -\frac{1}{C} \sum_{c=1}^{C} \left(\nabla e^c(p^k) \nabla e^c(p^k)^T \right)^{-1} \nabla e^c(p^k) e^c(p^k) \tag{6}$$

1.3 Tasks

- Generate a number of data samples (at least 100) using the provided functionality in the framework and divide them into a training set and test set.
- Compute the gradient $\nabla L(p)$ of the loss function[2]. Describe all your steps in the report.
- Implement Algorithm 1 and train the network with steepest descent. Use the Armijo rule to determine the stepsize α. Run a sufficient number of iterations such that the training converges. What would be a good stopping criterion?
- Validate the trained network on the test set. Report the loss (eq. (4)) on the training set as well as on the test set in your report.
- Investigate the different possibilities for computing the descent direction d^k. Which one performs better? Include your findings in the report!
- Run the better algorithm with different numbers of units in the hidden layer (e.g. 3, 5, 15, 25). What do you find?

1.4 Framework

We provide a framework with the basic structure. It contains functionality to generate training data as well as a few function definitions you might find useful. You are free to change the script to you liking, as long as you use the same function for generating data samples.

You must not import any additional python modules besides the ones that are already present!

Submission Upload your implementation and report zipped (filename: `MatrNr.zip`) to the TeachCenter. Don't forget to delete your old submission.

[2] Use the chain rule.