Learning by gradient descent

Neural Networks and Computational Intelligence - Practical Assignment III

Samuel Giacomelli
Student Number: S3546330
s.giacomelli@student.rug.nl

Davide Pedranz
Student Number: S3543757
d.pedranz@student.rug.nl

***Abstract*—Feed-forward neural networks are powerful devices used to solve regression problems. Since a network is formed by many units and the output is a continuous value, the algorithms designed for the perceptron can not be used for the training. A possible approach to solve the problem is to define a cost function on the training examples and minimize it using numerical optimization techniques. In this assignment, we implement Stochastic Gradient Descent as a numerical optimization method to train a feed-forward network with 1 hidden layer of 2 units.**

I. Introduction

Feed-forward neural networks are useful tools to solve regression problems. They consist in a group of neurons organized in layers: the neurons in one layer are connected to the ones in the following layer with weighted directed connection. The first layer of the network is called input layer and represents the input data for the network: the dimension of the input layer is equal to the number of dimensions of the dataset. The output layer has one unit for each continue value to predict. Between the input and the output layer, the network can have a variable number of hidden layers: each layer can have a different number of neurons, usually called hidden units. The output of a hidden unit is usually given by some non-linear function of the weighted sum of its input.

A way to train feed-forward neural networks is to define a cost function and optimize it using numerical optimization methods. This idea is captured in the Backpropagation training algorithm [1]: given a neural network, backpropagation defines a cost (or error) function and uses Gradient Descent to minimize it with respect to the weights parameters.

In this assignment, we implement and train a feed-forward neural network with 1 hidden layer and 2 hidden unit for a simple regression problem using Stochastic Gradient Descent and study its learning curves for different-sized training datasets and learning rate policies.

Section II introduces the Stochastic Gradient Descent training algorithm. Section III describes the implementation of the various experiments. Section IV and Section V discuss and summarize the obtained results.

II. Theory

A. Gradient Descent

The aim of Stochastic Gradient Descent is to minimize the neural network' cost function with respect to its weights. The algorithm initializes the weights to random values. Then, it iteratively select some training example ν from the training dataset $\mathbb{D}_{train}$ and updates the weights by some fraction η (usually called learning rate) of the gradient $\nabla_{w_j} e^{\nu}$ of the cost function e^{ν} with respect to the network's weights w_j:

$$w_j \leftarrow w_j - \eta \cdot \nabla_{w_j} e^{\nu}. \tag{1}$$

In order to use this iterative procedure, we need to define a proper cost function for the regression problem. A common choice is to use the squared error for the current example ε^{ν}:

$$e^{\nu} = \frac{1}{2}(\sigma(\varepsilon^{\nu}) - \tau(\varepsilon^{\nu}))^2, \tag{2}$$

where $\tau(\varepsilon^{\nu}))$ denotes the correct prediction for the current example.

Another possibility is to use all training examples (or a batch of them) to perform a single update, i.e. using the following cost function instead of e:

$$E = \frac{1}{2}\frac{1}{M}\sum_{\mu=1}^{M}(\sigma(\varepsilon^{\mu}) - \tau(\varepsilon^{\mu}))^2. \tag{3}$$

In this case, we usually talk of Gradient Descent (or Batch Gradient Descent) instead of Stochastic Gradient Descent.

B. Gradient Computation

The hidden units in out network take as input the training examples ε and use the hyperbolic tangent as the activation function:

$$g(\varepsilon) = tanh(w \cdot \varepsilon) \tag{4}$$

Since we have only 2 hidden units and the output unit simply computes the sum of the outputs of the hidden layer, the final predicted value is computed as:

$$\sigma(\varepsilon) = tanh(w_1 \cdot \varepsilon) + tanh(w_2 \cdot \varepsilon), \tag{5}$$

where w_1 and w_2 are the weights' vectors.

The gradient of the cost function defined in Equation (2) with respect to the weights w_j for a single hidden unit j can be computed as follows:

$$\begin{aligned}\nabla_{w_j} e^{\nu} = (tanh(w_1 \cdot \varepsilon^{\nu}) + tanh(w_2 \cdot \varepsilon^{\nu}) - \tau(\varepsilon^{\nu})) \cdot \\ \cdot (1 - tanh^2(w_j \cdot \varepsilon^{\nu})) \cdot \varepsilon^{\nu}\end{aligned} \tag{6}$$

III. Implementation

A. Training and Test Set

In different experiments we use a different number of training examples. Given the original dataset $\mathbb{D} = \{\varepsilon_\mu, \tau(\varepsilon_\mu)\}_{\mu=1}^{M}$ (with $M = 5000$), we create the training $\mathbb{D}_{train}$ and test set $\mathbb{D}_{test}$ as follows:

$$\mathbb{D}_{train} = \{\varepsilon_\mu, \tau(\varepsilon_\mu)\}_{\mu=1}^{P},$$
$$\mathbb{D}_{test} = \{\varepsilon_\mu, \tau(\varepsilon_\mu)\}_{\mu=Q}^{M},$$

with $Q > P$. Since we choose $P \in [1, 2000]$ in different experiments, we fix $Q = 2001$ for all experiments, in order to always test on the same dataset.

B. Stochastic Gradient Descent

The first step of our implementation of Stochastic Gradient Descent is to initialize the weights' vectors with random values and then normalize them to have unit norm $||w_j|| = 1$. It is important to initialize the weights randomly in order to avoid symmetry problems: since the weights gradient and the update rule is the same for all hidden unit, the updates at each epoch are also the same, which causes all units to learn the same final weights and reduces the representation power of the network. Then, we perform a given number of updates: at each iteration, we randomly select an example ν from the training dataset, compute the gradient with respect to the weights (see Section A-A) and update them according to Equation (1).

At regular intervals (i.e. after a fixed number of iterations), we compute the error on both the entire training and the test sets, as shown in Equation (3).

IV. Evaluation

To be able to compare the results of different experiments, we have fixed the number of iterations of Stochastic Gradient Descent to a constant value of 20000.

A. Train Error

Figure 1 shows the train and test error for the network trained using Stochastic Gradient Descent on the first 1000 examples. Both the train error and test error drop very quickly during approximately the first 1000 iterations, then remain almost constant for the rest of the training. The test error is slightly bigger than the train error: at the end of the training, the train error is around 0.10 and the test error around 0.12. Since the test error never increases, the model does not seem to overfit the training data.

B. Learned Weights

Figure 2 shows the weights learned from the hidden units of the networks after the training for $P = 1000$. As expected, the units learn different weights thanks to the random initialization of the weights.

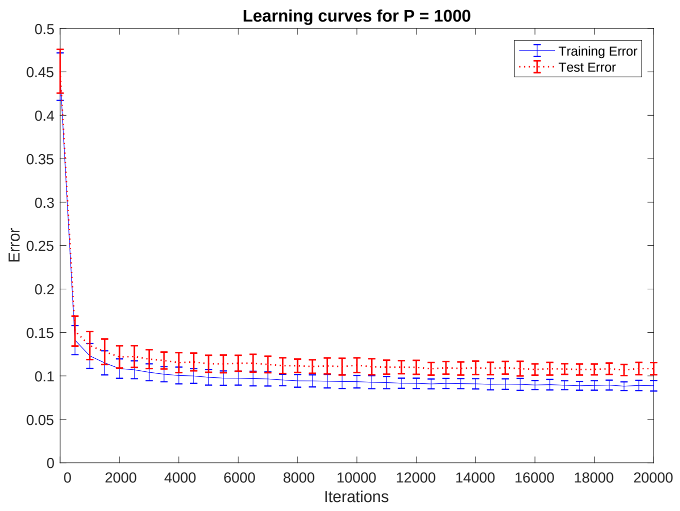

Figure 1. Train and test error of the network for $P = 1000$. The training is done with a fixed learning rate $\eta = 0.05$.

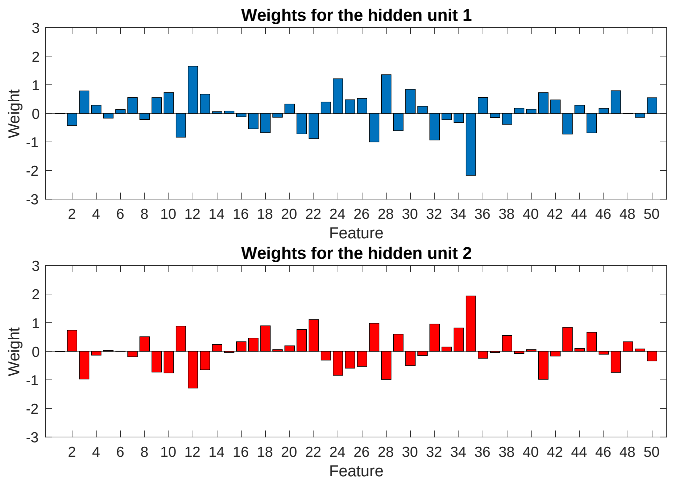

Figure 2. Weights of the hidden units after the training for $P = 1000$.

C. Train Dataset

Figure 3 shows the train and test error for different values of P, i.e. for a network trained on training dataset of different dimensions. For very small training datasets ($P = 20$, $P = 50$), the train error drops very quickly and stabilizes close to 0; the test error is high and even increases during the training for $P = 50$. It seems like the training set is too small for the network to generalize properly to new data, so it tends to overfit the train set.

For $P = 200$, the train error is higher, but the test error gets significantly smaller. Still, the test error increases during the training, which indicates overfitting.

For $P = 500$, the train and test error get closer and remain constant during most of the training. The results for higher values of P are very similar and are thus not shown here.

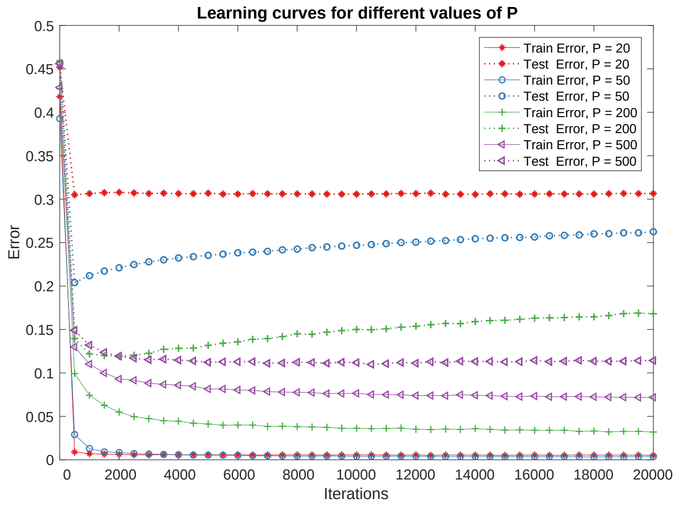

Figure 3. Train and test error of the network for different numbers of training examples P. The training is done with a fixed learning rate $\eta = 0.05$.

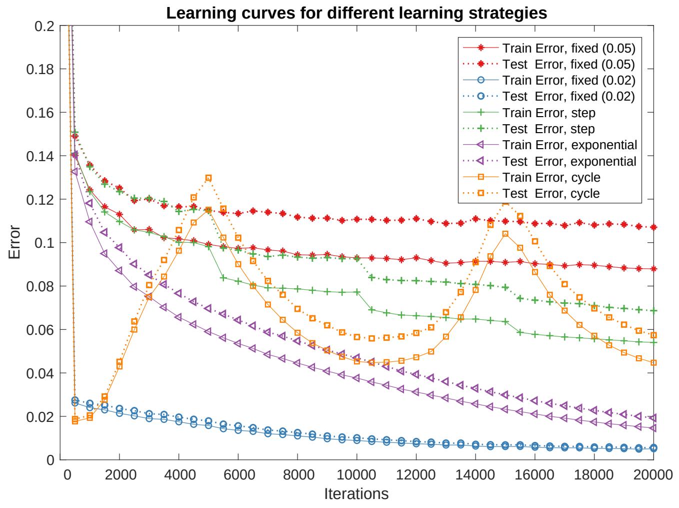

Figure 5. Train and test error for different learning strategies.

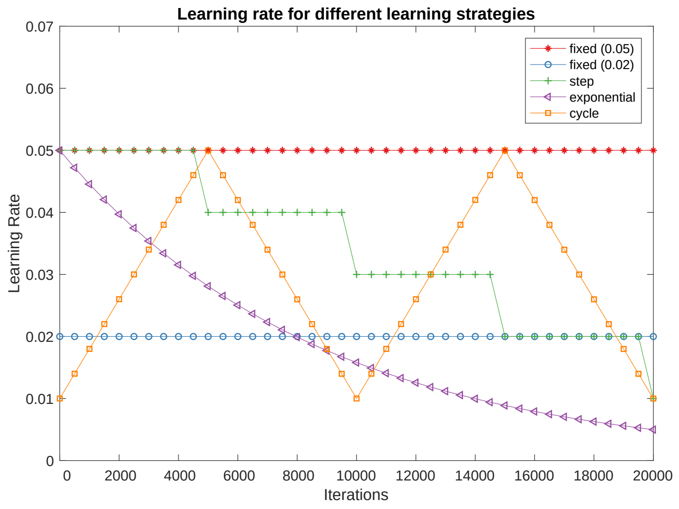

Figure 4. Learning rate evolution over iterations for different learning strategies.

D. Train Policy

The learning rate influences the number of iterations the training algorithm needs to converge to the optimal weights. A high learning rate causes the process to be unstable, while a low one takes too long time to converge. For this reason, we implement different schedulers for the learning rate able to change its value over time as a function of the current iteration: *fixed*, *step*, *exponential* and *cycle*. Figure 4 shows the learning rate change over time for the different strategies. From the application of these learning rate policies (LRPs) we expect a change in the behavior of the error function. In most LRPs the learning rate decreases over time, so the error should get more stable.

Figure 5 shows that different LRPs have different effects on the error trend. Overall, all implemented LRPs perform better than the *fixed*(0.05). The next sections discuss them in more detail.

1) Fixed learning rate policy (LRP): As the name suggests this LRP leaves the learning rate fixed during iterations. The error to drops down after around 500 iterations, then decreases very slowly. On our dataset, this strategy yields the best performances for a learning rate $\eta = 0.02$.

2) Step LRP: This strategy consists in decreasing the learning rate of a defined quantity (*drop*) after a certain number of iterations (*step_size*). As shown in Figure 4, we chose *step_size* = 5000 and *drop* = 0.01. Figure 5 shows a drop in both train and test error close to the step in the learning rate.

3) Exponential LRP: This strategy consist in decreasing the learning rate in an exponential way, i.e. in dividing it by a constant value at each iteration. This LRP lets the network learn quickly at the beginning, when the currently weights are likely to be far from optimal one, and then slowly, when they are getting closer to it. Figure 5 shows a smoother decrease of both the train and test errors, which reaches a value smaller than 0.1 after just 3000 iterations.

4) Cycle: This strategy consists in increasing and decreasing the learning rate linearly over the number of iterations, creating a sawteeth plot, as shown in Figure 4. The aim of this LRP is to avoid local minima letting the value of the learning rate to grow again when it reaches its minimum value. Figure 5 shows that this strategy performs well as far as the value of the learning rate is smaller than 0.02, then the errors increase following the increase of the learning rate.

V. Conclusion

Stochastic Gradient Descent is an effective algorithm to learn the parameters of a feed-forward neural network. The algorithm is iterative and takes as an input the number of iterations and a learning rate: it is important to choose an appropriate value for both parameters. If the number of iterations is too small, the algorithms does not manage to learn the optimal parameters for the network; if the number is too big, the model may overfit the training data and have bad performances on new data. Similarly, if the learning rate is too small, the training may take too long or even stuck in local minima; if it is too big, the updates may be too big and the model may never reach the optimal weights.

In general, it is difficult to choose the appropriate learning rate. In some cases it may be more effective to used a time-dependent one: for example, one may set a big learning rate at the beginning and then reduce it to make the network converge easier. We discussed some possible strategies and their effect on our regression problem. However, a complete discussion of the possible learning rate policies is outside the scope of this document.

VI. Individual Workload

The workload of this assignment was divided as follows.

A. *Together*

- Implementation of Stochastic Gradient Descent with a fixed learning rate (pair-programming).
- Report.

B. *Individually*

1) *Samuel Giacomelli:*

- Extension of Stochastic Gradient Descent to accept custom learning policies.

2) *Davide Pedranz:*

- Training of the neural network for different dimensions of the train dataset.

References

[1] Y. Chauvin and D. E. Rumelhart, *Backpropagation: theory, architectures, and applications*. Psychology Press, 1995.

Appendix A
Matlab code

A. *Weights' Gradients*

```
function [g1, g2] = gd(example, tau, w1, w2)
    %GD Compute the gradients of the error function for the given example
    %with respect to w1 and w1 (weights of the 2 hidden units).

    tanh_w1 = tanh(example * w1);
    tanh_w2 = tanh(example * w2);
    sigma = tanh_w1 + tanh_w2;
    g_comp = (sigma - tau) * example';
    g1 = g_comp * (1 - (tanh_w1 ^ 2));
    g2 = g_comp * (1 - (tanh_w2 ^ 2));
end
```