# Neural Networks: Assignment 2

Pepijn van Heiningen
pvheinin@liacs.nl

Michiel Vos
msvos@liacs.nl

November 15, 2013

## 1 Task 1: Function Optimization

### 1.1 Problem Description

The Rosenbrock function is a function that is used as a performance test for optimization algorithms. The equation can be found in figure 1. It has a global minimum at the point (1,1), where the value of $f = 0$. This is visualized in figure 2.

$$f(x, y) = 100 * (y - x^2)^2 + (1 - x)^2$$

Figure 1: The Rosenbrock's function

We were given the task to optimize the Rosenbrock function using five different algorithms, and subsequently compare their performance, in order to get an insight into the advantages, disadvantages and limitations of the different algorithms.

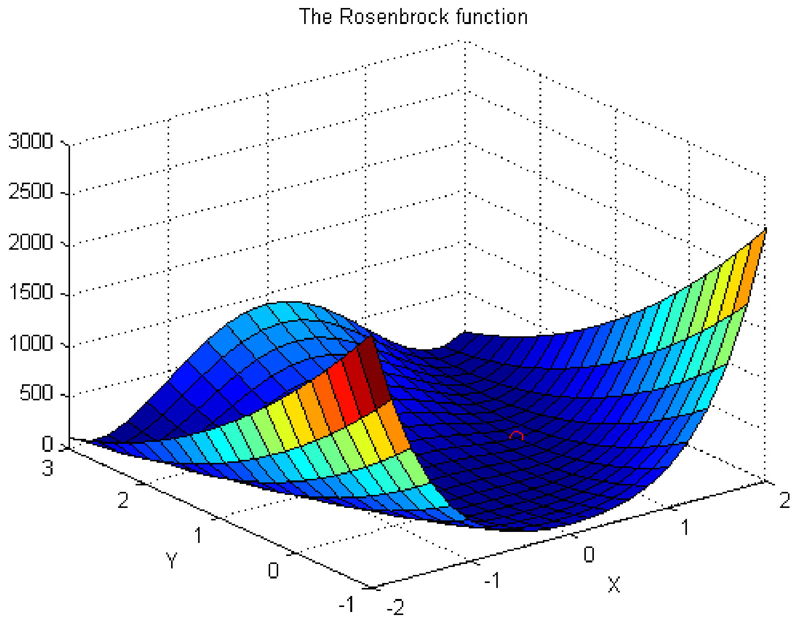


Figure 2: The Rosenbrock function, with minimal point

The 5 different algorithms we tested are:

- Gradient descent
- Gradient descent with line search
- Scaled conjugate gradient
- Conjugate gradient
- Quasi-Newton

To get a good comparison between the algorithms, each algorithm was run 100 times. We first generated 100 random points around (-1,1) as starting points. Each algorithm was started from the same random point.

There were 4 different measures to compare the algorithms with:

- The average number of evaluations of $f$
- The average number of evaluations of the gradient of $f$
- The average run time of the algorithm
- The average "success rate".

Together these measures should provide a decent indication which algorithm performs better. Because some functions might be a lot more computationally expensive to evaluate than the Rosenbrock function, the number of evaluations of both the function and the gradient should be as low as possible. A run obviously shouldn't take too long, and it should have a high success rate.

The success rate is measured as reaching the minimum with an accuracy of 0.0001. This means that when the optimal point found by the algorithm is evaluated, the value of $f$ is smaller than 0.0001. Of course we would like to have an optimizer that gets a 100% accuracy every time the algorithm is ran, but that is not always possible.

## 1.2 Implementation

As described in the assignment, we run the algorithms using 100 randomly generated starting points from the distribution `[-1, 1] + 0.5 * randn(1,2)`.
We set the options of the algorithm to the default options, with a few changes, we set the tolerance of the function value to $1 * 10^{-8}$, the iterations to 100, and we use a different learning rate for the two gradient descent algorithms. The reasoning behind the value is described in section 1.3.
In figure 3, we show the partial derivatives of the Rosenbrock function.

$$\frac{\rho f}{\rho x} = 400x^3 + 2x - 400 * yx - 2$$

$$\frac{\rho f}{\rho y} = 200y - 200 * x^2$$

Figure 3: Partial derivatives of $f$

In figure 4 you can see the pseudo-code of the implemented algorithm.

```
starting_point = repmat([-1,1],100,1) + 0.5*randn(100,2);
options = foptions;         % Standard options
options(1) = -1;            % Turn off printing completely
options(3) = 1e-8;          % Tolerance in value of function
options(14) = 100;          % Max. 100 iterations of algorithm
options(18) = 0.001;        % Learning rate
default_options = options;
functionList = {@graddesc, @graddesc, @scg, @conjgrad, @quasinew};
for i = 1:100
    for j = 1:5
        options = default_options;
        if(j==1 || j==2)
            options(18) = 0.008;
        end
        if(j==2)
            options(7) = 1;
        end
        tic;
        [dump, options, dump, dump] = functionList{j}(’rosen’, starting_point(i,:),
                                                                    options, ’rosegrad’);
        time = toc;
        results(i,j,1) = options(10);
        results(i,j,2) = options(11);
        results(i,j,3) = time;
        results(i,j,4) = options(8);
    end
end
```

Figure 4: Pseudo-code of task 1.

## 1.3 Experiments

In task 1.3 we were asked to tune both gradient descent algorithms manually. We first kept the number of iterations to 100, and tested different values of the learing rate. The results are in table 1.

| Iterations | Learning rate | GD | GD /w linesearch |
|---|---|---|---|
| 100 | 0.05 | 0 % | 5.4 % |
| 100 | 0.01 | 0 % | 5.8 % |
| 100 | 0.008 | 0 % | 5.8 % |
| 100 | 0.005 | 0 % | 5.6 % |
| 100 | 0.001 | 0 % | 3.8 % |

| Iterations | Learning rate | GD | GD /w linesearch |
|---|---|---|---|
| 100.000 | 0.008 | 0 % | 100 % |

Table 1: Optimizing the learning rate

The values are averages over 500 runs. We then tested the best value on 100 runs with 100.000 iterations, and the gradient descent algorithm with line search reached an accuracy of 100%. The standard gradient descent still got 0%. As you can see the optimal error is somewhere around a learning rate of 0.008, which is why we set the learning rate for the experiment to that value.

In order to acquire additional insights into how the algorithms work, we plotted the points that were found after each iteration in the following plot.

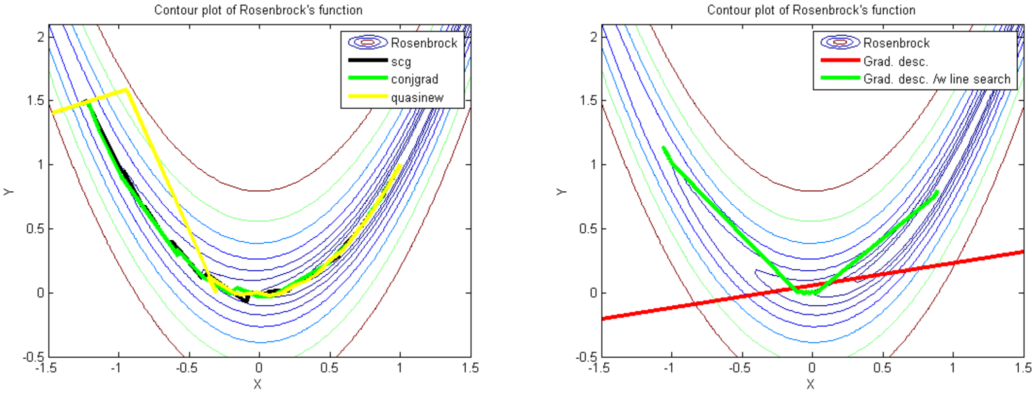


Figure 5: Contour plots of Rosenbrock's function

| Function | Evaluation | Gradient evaluation | Runtime | Successrate |
|---|---|---|---|---|
| Gradient descent | 101 | 100 | 0.003 | 0% |
| Gradient descent /w line search | 1836.4 | 98.8 | 0.090 | 4% |
| Scaled conjugate gradient | 45.38 | 69.64 | 0.002 | 100% |
| Conjugate gradients | 274.15 | 18.09 | 0.014 | 100% |
| Quasi-Newton | 92.39 | 27.41 | 0.006 | 100% |

Table 2: Results of the 5 algorithms. All values are averages.

In table 2 you can find the results of the different algorithms. All 5 algorithms were run on an Intel Core i5-3470, the values are averages over 100 runs. The parameters used for the algorithms are described in the section 'Implementation'. As you can see the gradient descent algorithm, even after optimization, still achieves a successrate of 0% on the data. If we look at the output-coordinates of the algorithm, it shows that this setting of the learning rate is too high, because the coordinates keep growing until they reach infinity. Gradient descent with line search already works a lot better, but it costs more function evaluations. The other three methods clearly achieve better results on all measured objectives, with less evaluations, a shorter runtime and higher successrates.

## 1.4 Conclusions

It is difficult to select the best algorithm. Both the Scaled Conjugate Gradient approach and the Quasi-Newton algorithm perform well on this particular problem. The Scaled Conjugate Gradient uses less function evaluations than Quasi-Newton, but Quasi-Newton uses less gradient evaluations. The runtime of both algorithms is almost equal, and they both achieve a successrate of 100%. It will depend on the computational difficulty of the problem to determine which algorithms is better.

The worst algorithm is clearly gradient descent, since it doesn't find good solutions, even in 100 runs there are no successful solutions. Gradient descent with line search is the slowest algorithm of them all, but because it does produce some useful solutions it is better than the original gradient descent algorithm. The reason for the very poor performance of the gradient descent algorithm is probably that if the learning rate is set too high, the algorithms best coordinates will 'explode' into infinity. If the learning rate is set to a lower rate, 100 iterations are not enough to get to the optimal point. Because the Rosenberg function has such a narrow valley, finding the optimal value is very difficult.

# 2 Task 2

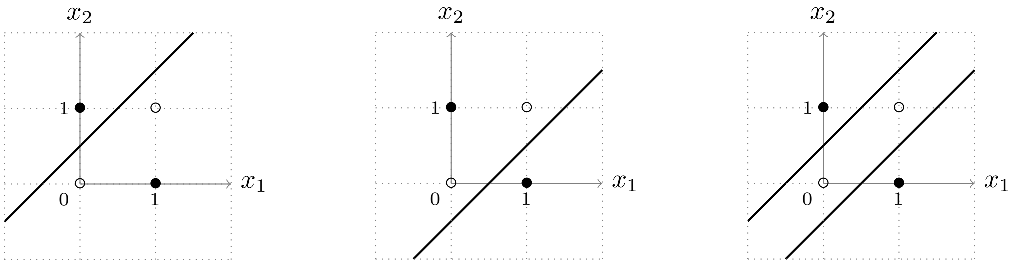
## 2.1 Problem Description

Instead of trying to find the minimum of the Rosenbrock's function in task 1, we were asked to solve the "exclusive or" (xor) problem in task 2. The problem exists out of two input bits which should produce their associated value of the (only) output bit. Please see table 3 for the truth table.

| $x_1$ | $x_2$ | $y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Table 3: The truth table. When only one of the two input bits is true, the output bit should be true.

A perceptron cannot solve this problem, because the classes (different values of y) could not be separated linearly with one line in twodimensional space. This is shown if figure 6.



(a) (b) (c)

Figure 6: 3 trials to separate the classes. The filled circles are true cases and the empty circles are false cases.

As seen in figure 6 (c) it is possible to solve the xor problem, but not with a perceptron. We need something more advanced, like a neural network with a hidden layer.

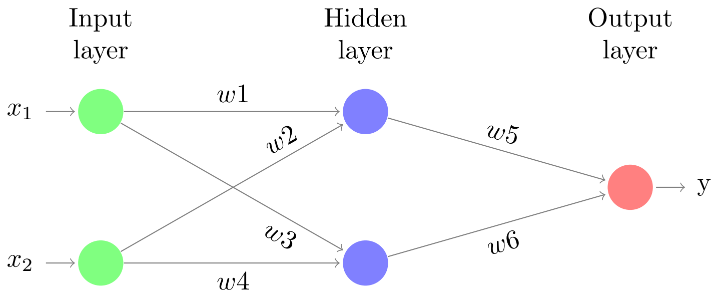


Figure 7: The xor neural network.

## 2.2 Solution

We made a multilayer neural network with three nodes and six weights [1]. Please see Figure 7 for an illustration. The network is trained with the data of the truth table and the weights are updated with the algorithms of task 1. The algorithms tries to steer the weights in such a way the errors of the predicted classes and the actual classes are minimized.

## 2.3 Implementation

```
    function [y] = xornet(x1, x2, w)
        net1 = w(1) * x1 + w(2) * x2;
        y1 = phi(net1);
        net2 = w(3) * x1 + w(4) * x2;
        y2 = phi(net2);
        net = w(5) * y1 + w(6) * y2;
        y = phi(net);
    end
```

Listing 1: The computation of the output value, based on the input values and the weights.

The xornet function computes the output value. The ($\phi$) sigmoid function is used as the activation function, so the output value will always lay between 0 and 1. The mysse function calculates the sum squared errors of the network. For every scenario of input values (all four combinations of $x_1$ and $x_2$) the computed value is substracted with the actual value (which is found in the truth table). These values are squared and cumulated. The lower the result the better. The dmysse calculates the derivatives, which are used by the 5 algorithms. In order to check the correctness of the function we used the gradchek function of the Netlab toolbox. The results are in table 4. The values of the analytic column are computed with the functions mysse (Listing 2) and dmysse (Listing 3). The values of the diffs column are approximated with a central difference formula with a very small step size. The deltas are the difference between the two methods and are very near to 0. The reason for this is because the gradcheck function produces an approximation.

$$\phi(x) = \frac{1}{1+e^{-x}} \tag{1}$$

$$\phi'(x) = \phi(x)(1-\phi(x)) \tag{2}$$

Figure 8: The sigmoid function (1) and the derivative (2).

```
function [d] = mysse(w)
    d = power(xornet(0, 0, w) - 0, 2) +
        power(xornet(0, 1, w) - 1, 2) +
        power(xornet(1, 0, w) - 1, 2) +
        power(xornet(1, 1, w) - 0, 2);
end
```

Listing 2: The calculation of the sum squared error of the weights.

[1] The network could be trained much faster when bias nodes are used, but in sight of the assignment there is chosen to do it the hard way and make the problem more difficult.

```
function [d] = dmysse(w)
    d = zeros(1, 6);

    input = [0, 0; 0, 1; 1, 0; 1, 1];
    target = [0, 1, 1, 0];

    for i = 1:4
        net1 = w(1) * input(i, 1) + w(2) * input(i, 2);
        y1 = phi(net1);
        net2 = w(3) * input(i, 1) + w(4) * input(i, 2);
        y2 = phi(net2);
        net = w(5) * y1 + w(6) * y2;
        y = phi(net);

        d(1) = d(1) + (y - target(i)) * phiprime(net) * w(5) * phiprime(net1) * input(i, 1);
        d(2) = d(2) + (y - target(i)) * phiprime(net) * w(5) * phiprime(net1) * input(i, 2);
        d(3) = d(3) + (y - target(i)) * phiprime(net) * w(6) * phiprime(net2) * input(i, 1);
        d(4) = d(4) + (y - target(i)) * phiprime(net) * w(6) * phiprime(net2) * input(i, 2);
        d(5) = d(5) + (y - target(i)) * phiprime(net) * y1;
        d(6) = d(6) + (y - target(i)) * phiprime(net) * y2;
    end
    d = d * 2;
end
```

Listing 3: The computation of the derivatives of the sum squared error of the weights.

| Weight | Analytic | Diffs | Delta |
|---|---|---|---|
| $w_1$ | -0.0050 | -0.0050 | $5.4682 * 10^{-11}$ |
| $w_2$ | -0.0032 | -0.0032 | $1.1398 * 10^{-10}$ |
| $w_3$ | 0.06435 | 0.06435 | $-1.8951 * 10^{-11}$ |
| $w_4$ | 0.04339 | 0.04339 | $-3.2260 * 10^{-11}$ |
| $w_5$ | 0.18566 | 0.18566 | $6.0969 * 10^{-11}$ |
| $w_6$ | 0.18131 | 0.18131 | $1.0486 * 10^{-10}$ |

Table 4: Results of the gradchek function.

## 2.4 Experiments

We ran all 5 algorithms for 100 iterations, starting from a random point. This is done 100 times to get the averages of the function evaluations, gradient evaluation and the successrate. The relative amount of times an algorithm is successful after 100 iterations is the successrate. An algorithm is succesful when it can predict all four scenarios correctly. Listing 4 gives more details about this definition. The results are in Table 5.

```
success =   round(xornet(0, 0, w)) == 0 & ...
            round(xornet(1, 1, w)) == 0 & ...
            round(xornet(0, 1, w)) == 1 & ...
            round(xornet(1, 0, w)) == 1;
```

Listing 4: Success condition of a set of weights.

| Function | Evaluation | Gradient evaluation | Runtime (sec) | Successrate |
|---|---|---|---|---|
| Gradient descent | 101 | 100 | 2.0 | 0% |
| Gradient descent (line search) | 1700 | 100 | 5.1 | 0% |
| Scaled conjugate gradient | 67 | 121 | 2.3 | 0% |
| Conjugate gradients | 1691 | 72 | 4.6 | 10% |
| Quasi-Newton | 253 | 82 | 2.0 | 13% |

Table 5: Results of the 5 algorithms. All values are averages and rounded.

Another experiment was to see how well an naive produce will perform. We generated 10 million random weights sets and tested how many of the sets were successful. Only 8 were successful.

### 2.5 Conclusion

Neural networks can solve the xor problem. The update function of the weights are important, because they have different performances. The successrate of the Quasi-Newton method is the highest, but how important is the successrate? We only need one solution to solve the problem and one successful set of weights is not better compared to another successful set of weigths. Gradient descent and scaled conjugate cannot solve this problem within the chosen boundaries. The naive approach of generating random solutions can solve the problem, it only takes a while. Note, probably all functions will perform better if we used bias nodes in the neural network.

# 3 Task 3: Handwritten Digit Recognition with MLP

In task 3 we had to compare the Multilayer Perceptron networks to the single layer networks we have investigated in Assignment 1.

## 3.1 Problem Description

We were given the handwritten digits dataset from Assignment 1 again, and were now asked to implement a multilayer perceptron network, and train it to recognize handwritten digits. The goal is of course to achieve the highest classification rate posssible.

## 3.2 Implementation

We built a Multi-Level perceptron with 256 inputs and 10 outputs. Each output can be seen as a boolean value that is true if the input was the corresponding digit.

```
function task_3()
    %Create the net
    net = mlp(256,75,10,func);
    %Train the nets
    net = mlptrain(net, training’, target_training, 100);
    %Test errors on training
    output_training = mlpfwd(net, training’);
    %Test errors on testset
    output_test = mlpfwd(net, testdata’);
    %create confusion matrix for training
    %create confusion matrix for testset
end
```

Figure 9: Pseudo-code of task 1.

### 3.3 Experiments

We first tried to find the best number of hidden nodes. We set the number of training iterations to 100, and used the Scaled Conjugate Gradient as a training method. In the plot of figure 10 you can see that the softmax activation function yields the best results.

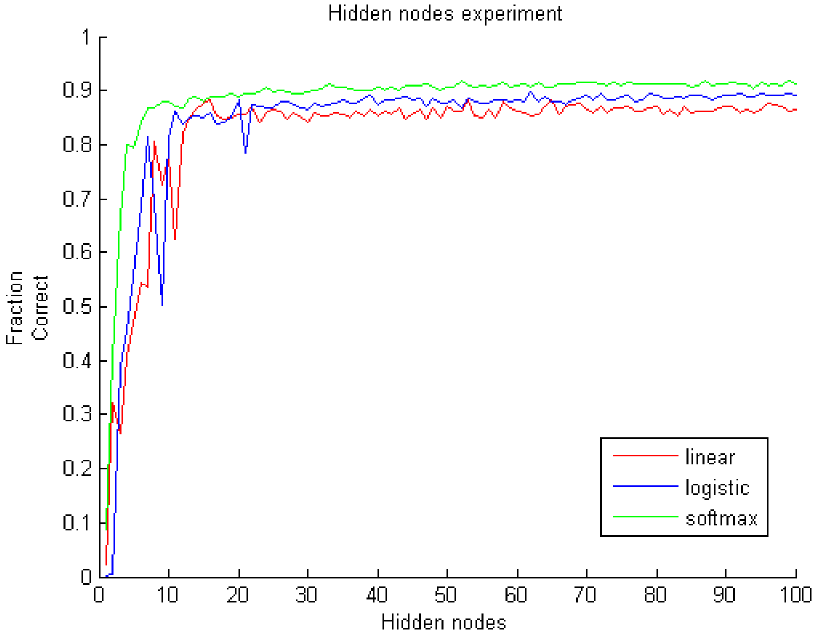


Figure 10: Hidden nodes

Because there is no improvement after about 30 hidden nodes, we set the number of nodes to 75. To check whether there are differences between the Scaled Conjugate Gradient method, Quasi-Newton and Backpropagation, we show the classification rates and runtime in table 6.

| Training method | Runtime | Classification rate |
|---|---|---|
| SCG | 2.3550 | 93.84 |
| Conjugate Gradient | 7.2546 | 93.42 |
| Backpropagation | 1.1860 | 65.74 |

Table 6: Comparison of training methods

The values are averages over 20 runs. Quasi-Newton was not tested, because it used too much RAM for our pc (which has 8GB).

Finally we tried to find the best value for the number of training iterations. (See table 7)

| Training iterations | Runtime | Classification rate |
|---|---|---|
| 1 | 0.0567 | 6.7000 |
| 11 | 0.2825 | 90.6000 |
| 21 | 0.4920 | 92.4000 |
| 31 | 0.6624 | 94.7000 |
| 41 | 0.8748 | 93.4000 |
| 51 | 1.0392 | 94.9000 |
| 61 | 1.2038 | 93.7000 |
| 71 | 1.3936 | 93.9000 |
| 81 | 1.6338 | 93.4000 |
| 91 | 1.7814 | 94.0000 |

Table 7: Comparison of training iterations

As we expected there is a minimum of training iterations necessary to train the network properly, but after that threshold is reached, no more improvement is found. This is why we set the training iterations to 50.

### 3.3.1 Activation functions

In figure 11 you can see the different functions and the way their error values are calculated.

$$error = y - t \tag{3}$$

$$error = t * log(y) + (1 - t) * log(1 - y) \tag{4}$$

$$error = t * log(y) \tag{5}$$

Figure 11: Error measurements in multiple activation functions. (1) Linear (2) Logistic (3) Softmax

## 3.4 Conclusions

The final algorithm used a network with 75 hidden nodes, 50 training iterations and the Scaled Conjugate Gradient method for training the network. Finally we ran the algorithm 100 times, mixing up the test and trainingset each run, checking the algorithm only on input digits that the it hasn't seen before. We achieved a final accuracy of 93.72%.