Gradient Descent
Neural Networks and Computational Intelligence
Practical 3

Rogier de Weert (s1985779)
Tim Oosterhuis (s2234831)

January 2018

1 Introduction

Gradient descent (or ascent) learning is a common optimization strategy in the areas of neural networks and machine learning. In the context of training neural networks, gradient descent learning aims to minimize a cost function based on the training targets, by changing the weights of the network in the direction of the steepest negative gradient. The training error, which is some measure of difference between the output layer and the training targets, is propagated across the network weights starting from the output layer, and calculating the gradient of each previous layer, using the chain rule. This is called back-propagation of the error, because the process traverses the network in the opposite direction compared to the normal activation of the network.
In our experiment, we use gradient descent learning to train a soft committee machine consisting of two hidden units. The hidden to output weights and the gain parameter are constant. The output of the final layer $\sigma(\xi)$ for a real-valued n-dimensional input vector ξ is given by:

$$\sigma(\xi) = \tanh[w_1 \cdot \xi] + \tanh[w_2 \cdot \xi]$$

where w_1 and w_2 are the input-to-hidden weight vectors and *tanh* denotes the hyperbolic tangent activation function. The error E to be minimized, is given by:

$$E = \frac{1}{P}\frac{1}{2}\sum_{\mu=1}^{P}(\sigma(\xi^\mu) - \tau(\xi^\mu))^2$$

Where P denotes the size of the training set, and $\tau(\xi^\mu)$ denotes the continuous label for training input vector ξ^μ with index μ in the training set. In our experiment, we will use online stochastic gradient descent. At each learning step the weight vectors w_1 and w_2 are updated according to the following equations, where the learning rate η was a constant of 0.05:

$$w_1(t+1) = w_1(t) - \eta(\tanh[w_1 \cdot \xi^\mu] + \tanh[w_2 \cdot \xi^\mu] - \tau(\xi^\mu)) * (1 - \tanh^2[w_1 \cdot \xi^\mu]) * \xi^\mu$$

$$w_2(t+1) = w_2(t) - \eta(\tanh[w_1 \cdot \xi^\mu] + \tanh[w_2 \cdot \xi^\mu] - \tau(\xi^\mu)) * (1 - \tanh^2[w_2 \cdot \xi^\mu]) * \xi^\mu$$

The generalization error E_{test}, which is used to measure to which degree the network has learned the regressing problem it is trained on, is given by the following equation:

$$E_{test} = \frac{1}{Q}\frac{1}{2}\sum_{\mu=1}^{Q}(\sigma(\xi^\nu) - \tau(\xi^\nu))^2$$

Where Q denotes the size of the test set, and $\tau(\xi^\nu)$ denotes the continuous label for test input vector ξ^ν with index ν in the test set. The training set and the test set are both drawn from a larger population, with no overlap. All formulas were derived using [1, Part 4(K1)]

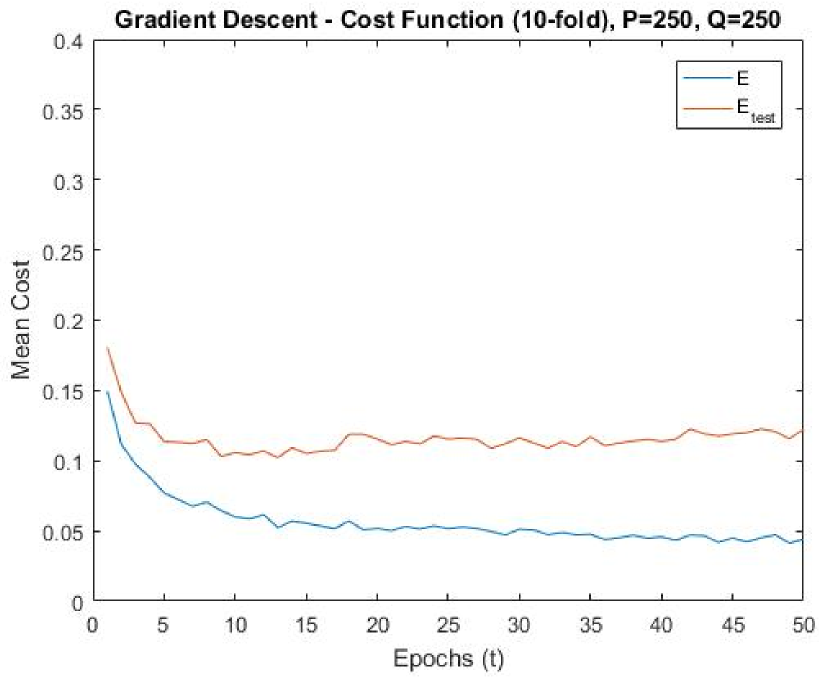
2 Implementation

The supplied data set consists of 5000 data vectors of 50 dimensions, with a continuous label for each input vector. This data was divided into 20 parts of 500 vectors for 10-fold training and testing, using 250 data vectors for training and 250 data vectors for testing in each fold. In each fold, the gradient descent algorithm was trained for 50 epochs, at which the E and E_{test} were calculated after each epoch. During each epoch P (250) training steps were performed, using a randomly picked data vector for training. This means that the algorithm did 12500 training steps per fold. At each fold the weight vectors were initialized as normalized random independent vectors. Our implementation of gradient descent with the update step, in Matlab, is shown below, in the first listing. In the second listing below, the error calculation function is shown. This function is used for calculation of both the training and test error.

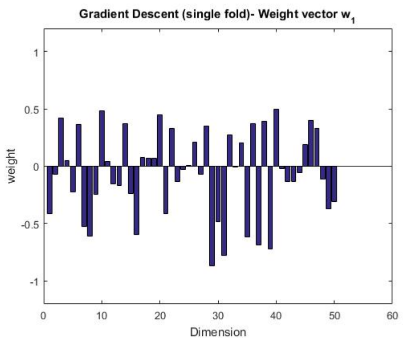
```
for t = 1:t_max % t_max = epochs
    if(time_dependent==1)
        learnrate = a/(b*t);
    end
    for k = 1:P     % actual training steps = t_max*P
        idx = randi(P);
        sigm = tanh(dot(w1,train_data(:,idx))) +
                tanh(dot(w2,train_data(:,idx)));

        %derivation with respect to w1
        sigm_der1 = 1-tanh(dot(w1,train_data(:,idx)))^2;
        sigm_der2 = 1-tanh(dot(w2,train_data(:,idx)))^2;

        % Three parts gradient
        grad1 = (sigm-train_labels(idx)) * sigm_der1 * train_data(:,idx);
        grad2 = (sigm-train_labels(idx)) * sigm_der2 * train_data(:,idx);
        % Update step using gradient
        w1 = w1 - learnrate*grad1';
        w2 = w2 - learnrate*grad2';
    end
    % Error calculation
    E(t) = calc_E(train_data, train_labels, w1, w2);
    E_test(t) = calc_E(test_data, test_labels, w1, w2);
end


function [ E ] = calc_E( data, labels, w1, w2 )
Ex = 0;
P = size(data,2);
for i = 1:P
    Ex = Ex + (tanh(dot(w1,data(:,i))) + tanh(dot(w2,data(:,i))) - labels(i))^2;
end
E = Ex/(2*P);
end
```

3 Results

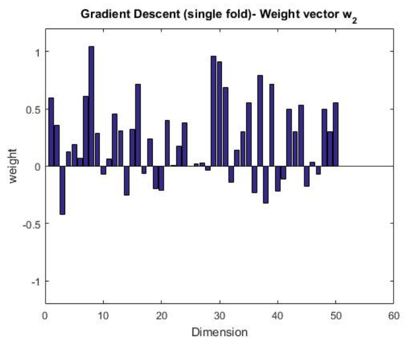
The training and generalization errors over the course of training are shown in Figure 1a, notice that these are the mean errors over 10-fold training and testing. Figure 1b and Figure 1c show the resulting weight vectors after 50 epochs of training.



(a) Cost function over time



(b) Resulting w_1 vector after 1 fold



(c) Resulting w_2 vector after 1 fold

Figure 1: Gradient Descent results

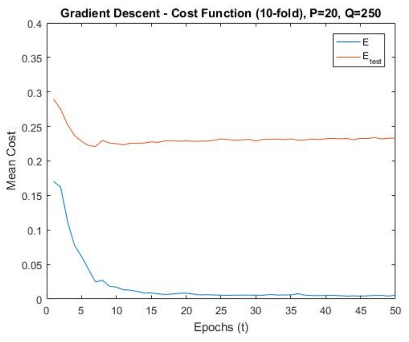
4 Discussion

When looking at the training error and generalization error as shown in figure 1a, we can see that the generalization error is notably higher than the training error. We can conclude from this that the network over-fitted to the training data, at least to some degree, and was not able to learn the more general pattern which underlies both the training and test data combined, based on just the training data. However, there is also a notable drop in the generalization error compared to its initial value, which implies that the network did learn some general features of the data set. We can also see however that both the training and test error fluctuates, leading us to believe that the network has not fully converged yet. These fluctuations also lead us to believe that the learning rate is too high, so that no convergence will occur.

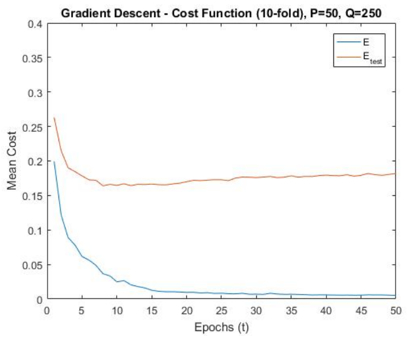
It's difficult to draw any major conclusions based on the final weight vectors, shown in figures 1b and 1b. We can see some dimensions, for example dimension 25, which has a weight of nearly zero, for both weight vectors. This means variance within the input data along these dimensions, will barely influence the output activation. For a large machine learning project with high dimensional data, such information might be relevant, because it could be used for data pruning. In this case the great variability in the resulting weight vectors shows that the gradient descent does not yield sensible weight vectors.

5 Bonus: training sample size

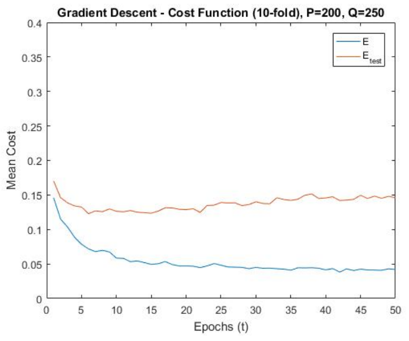
Investigating the influence of the size of the training data on the resulting error we repeated the experiment as described above with different values for P. Each experiment was performed with a more traditional way of folding. For each fold, P data vectors were randomly picked from the complete data set, after which Q (which was kept constant at 250) data vectors were also randomly picked from the remaining data vectors. The newly tested values for the training set size were: 20, 50, 200, 500, 1000 and 2000. The resulting error rates during training can be found in Figure 2. This figure shows that for smaller P, the resulting training error is very small, even approaching zero for P=20 and P=50. The resulting test error is however much larger for the same training set sizes used. When the training size is increased, the resulting training error also increases. The resulting test error however decreases with increasing training set size. When using a training set size of 2000, the training and test error almost resemble each other after full training. This effect shows that when training is done with a small training set size the gradient descent algorithm greatly over-fits, having a very small training error but a very high test error. Training with a larger training set size leads to slightly worse training error, but the test error greatly resembles the training error and is significantly lower than the test errors of that of the algorithm trained with a smaller training set size. This leads us to conclude that for better generalization, a large training set size should be used. These results show that a training set size of 2000 leads to the best generalization. We however still notice fluctuations in the resulting training and test error rates. We again hypothesize this is because the learning rate of 0.05 is too high to ensure convergence



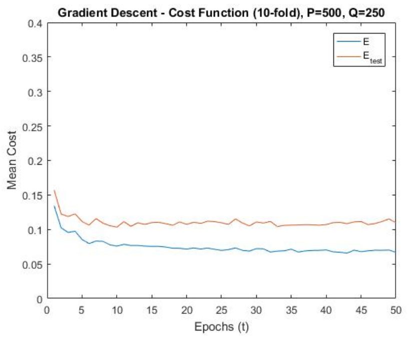
(a) $P = 20$



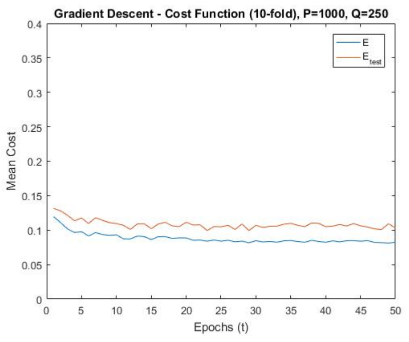
(b) $P = 50$



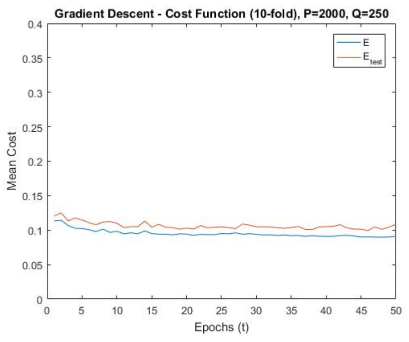
(c) $P = 200$



(d) $P = 500$



(e) $P = 1000$



(f) $P = 2000$

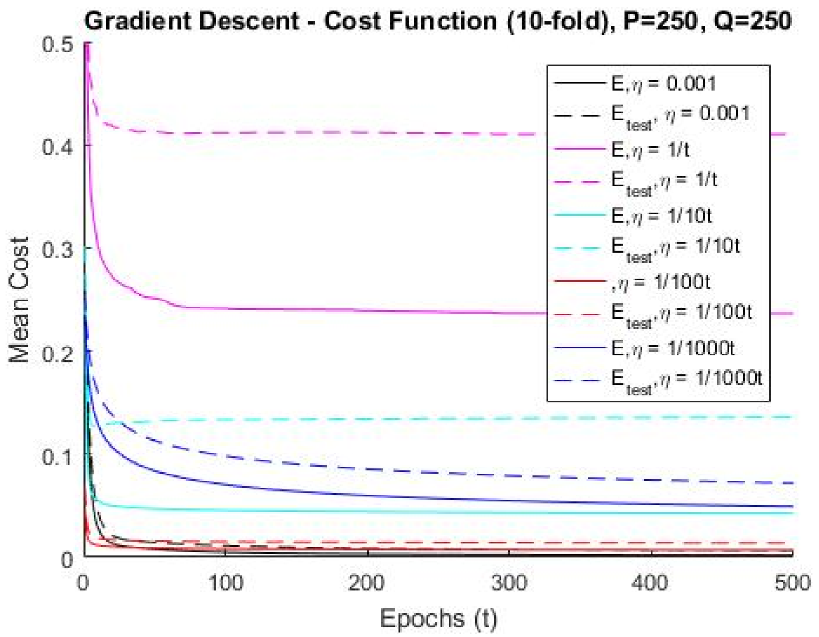
Figure 2: Gradient Descent cost functions - different training sizes

6 Bonus: time-dependent learning rate

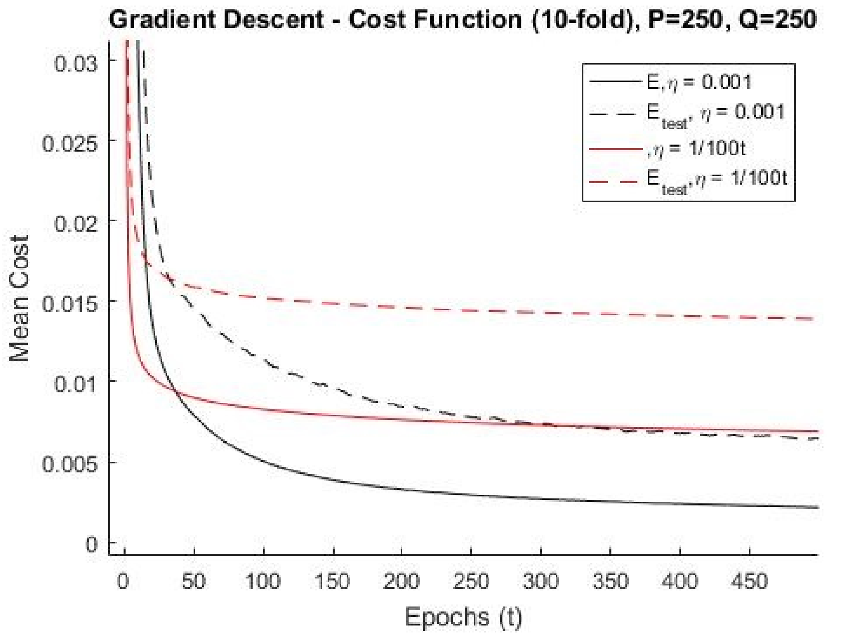
Instead of a constant learning rate, a time-dependent learning rate can be used. When using a time-dependent learning rate, the learning rate decreases as time increases. In our implementation we took the original 10-fold experiment with $P = 250$ and $Q = 250$, changed t_{max} to 500 and tested several time-dependent learning rates. The general formula for a time-dependent learning rate is:

$$\eta = a/b * t$$

In this experiment we kept a at a constant of 1, and tested the values 1, 10, 100 and 1000 for b. The learning rate was adapted after each epoch, not after each single training step. Figure 3a shows the averaged resulting training and testing errors during the gradient descent training using the original data set over 10 folds. The solid lines represent the training error, where the dashed lines of the same color represent the test error of the same experiment. Note that the black lines represents a control experiment using a small constant value for the learning rate $\eta = 0.001$. Firstly, we observe that, for all values of b, gradient descent learning with a time-dependent η produces smooth error curves, which flatten to a near plateau relatively quickly, compared to the control and to the error curves seen in Figures 1 and 2. Secondly it seems that a higher value for b leads to smaller errors and better generalization, up to a certain point. When looking at the error rates for gradient descent learning with $\eta = 1/t$, we notice that the training error is much larger than with the fixed learning rate, or even with the higher fixed learning rate of $\eta = 0.05$, as seen in figure 1. The test error is also much higher than the training error, suggesting over fitting takes place, in spite of the relatively high cost during training. This over fitting seems to disappear as b increases, leading to good results for $\eta = 1/100t$. On the first epoch this means the learning rate is 0.01, decreasing with each epoch. For $b = 1000$, the results are notably worse. This indicates that the learning rate should be small enough to ensure convergence, but also as large as possible, to facilitate learning within a reasonable time frame. A counter argument against using a time-dependent learning rate in this case can be made, based on the results of this experiment. The learning rate becomes exponentially smaller with each epoch. This makes the gradient descent so slow, that it seems not to lead to convergence to the global minimum, within a reasonable time frame. To illustrate this, we look more closely at the results of the best time-dependent learning rate experiment, $\eta = 1/100t$, compared to the control experiment, with $\eta = 0.001$. Notice, that this is a zoom in of the bottom part of Figure 3b, with the scale of the y-axis adjusted. Here we can clearly see that the (generalization) error rate for gradient descent learning using a constant learning rates of $\eta = 0.001$ is lower than that of the time-dependent learning rate of $\eta = 1/100t$. The algorithm seems to converge so slowly, that it almost seems like it has reached a sub-optimal plateau region when using the time-dependent learning rate. When using a constant learning rate, the decrease is more gradual, which leads to a more optimal result in this case. Seeing as $\eta = 1/100t$ was the optimal time-dependent learning rate we found, after doing a parameter sweep on b from 1 till 1000, we hypothesize that a small constant learning rate of for example $\eta = 0.001$ or less, would lead to a better convergence to the minimum than a time-dependent learning rate, for this data, when considering only the parameter b.



(a) error comparison



(b) zoom of (a) to highlight the difference between $\eta = 0.001$ and $\eta = 1/100t$

Figure 3: Gradient Descent - time dependent learning rates

7 Conclusion

The performance of using gradient descent to predict the output labels of the supplied data depends on multiple factors. Two of the factors that were studied are the learning rate and the training set size. As described above taking a larger training set size increases the training error when compared to lower training set sizes and decreases the test error to almost that of the training error. This means that using a higher training set size leads to better generalization. Using 10-fold training with a training set size of 2000 and a learning rate of 0.05 leads to a test error of approximately 0.12 after 50 epochs. Our second conclusion is that using a time-dependent learning rate does not outperform a fixed learning rate of 0.001. This fixed learning rate is small enough to ensure convergence (where a fixed learning rate of 0.05 did not converge optimally), large enough for fast learning and leads to an even better test error (below 0.01) using only a training set size of 250 than in the first experiment. Future work could be done, in search of the best results combining these findings by training using a small fixed learning rate of 0.001 and a large training set size of 2000. It could be even better, to train using 80% of the data, while using the remaining 1000 examples as the test set.

References

[1] Michael Biehl. Lecture notes on Neural Networks and Computational Intelligence, , 2017-2018.