08336 Neural, Emergent and Agent Technology

ACW Part I + II

(Spring Term 2018)

Ephraim Schott

ACW Part I

Neuron model and learning setup

In order to examine and answer the questions of this report a neuron model similar to the illustration in figure 1 has been implemented using python. The code can be seen in the attached file `neural_acw_part1.py`. The input weights were initialized with random values between 0 and 1.

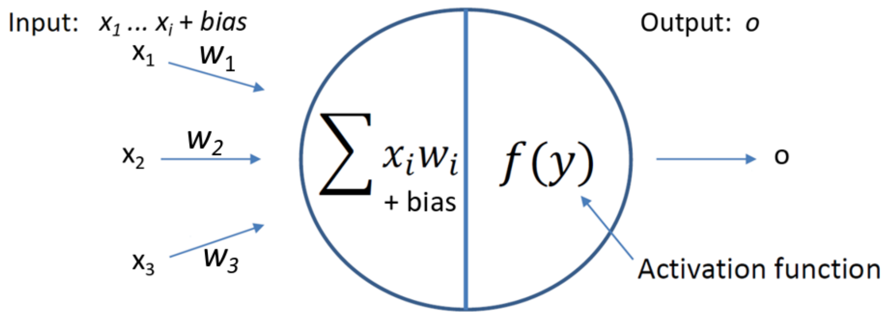


Figure 1: Model of a neuron or perceptron.

The general calculation of the output of a neuron always followed the equations $y = \sum x_i w_i + bias$ and $o = f(y)$. The weight update gets calculated by this equation $w_i = w_i * learn\_rate * (t - o) * x_i$, where t is our target value. For modelling the perceptrons activation function $f(y)$ the Heaviside step function (see figure 2) was used. The neuron uses the logistic sigmoid function $(\frac{1}{1+e^{-y}})$ as $f(y)$.

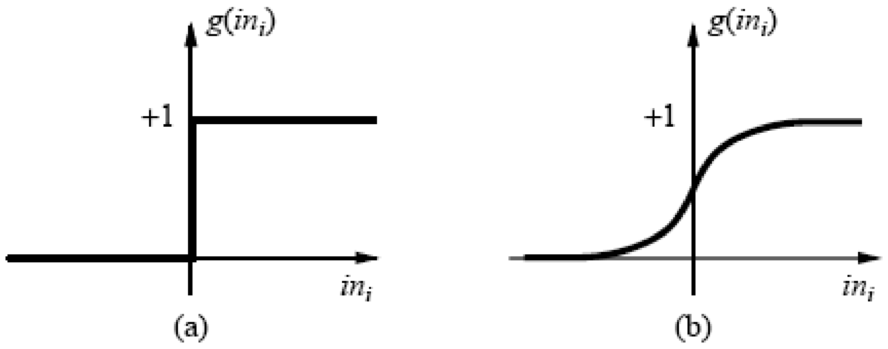


Figure 2: Used activation functions (left: step function, right: logistic sigmoid function).

Training perceptrons and neurons

As the simple perceptron uses the Heaviside step function it is only able to create binary outputs. Figure 3 shows that the output of the perceptron is not effected by learning. Indeed the weights of the perceptron get adjusted by each training iteration, but the adjustments can't effect or improve the binary output of the perceptron. A step function is not suitable for this task, because the task requires continuous outputs between 0 and 1 in order to make precise and useful predictions.

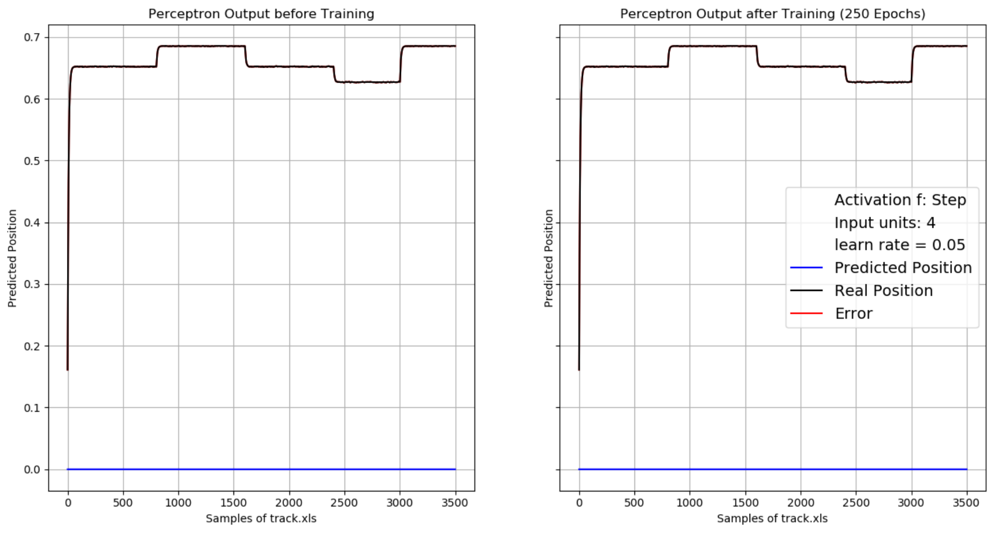
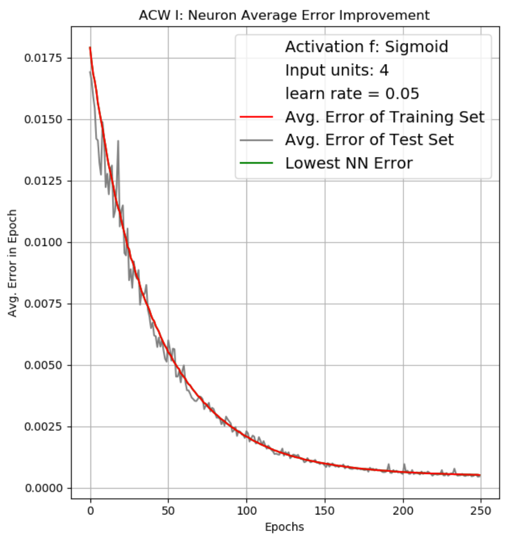
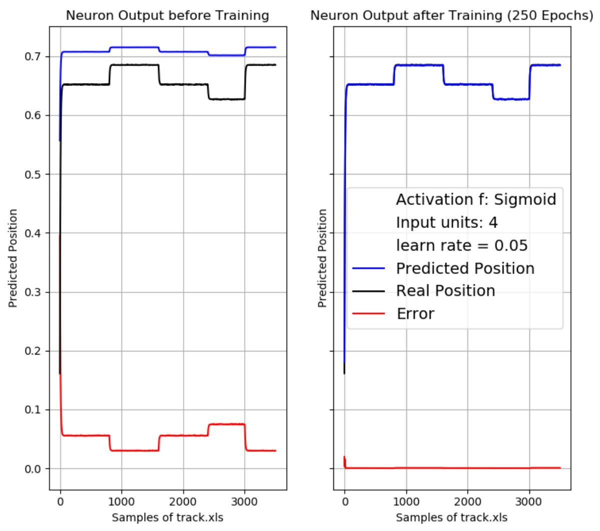


Figure 3: Output of a perceptron with a step function before and after learning.

The prediction results change when we replace the perceptron by a neuron which utilizes the logistic sigmoid as an activation function. The graphs of figure 4(b) show that the trained neuron is able to predict positions with a small error. The neuron is able to output continuous numbers that depend on, and vary according to the inputs. Through the continuous output we can obtain different delta errors in each iteration, through which we can update the weights. As one can see in figure 4(a) the average error of the neuron is decreasing through the training, which indicates that the neuron is now capable of learning.



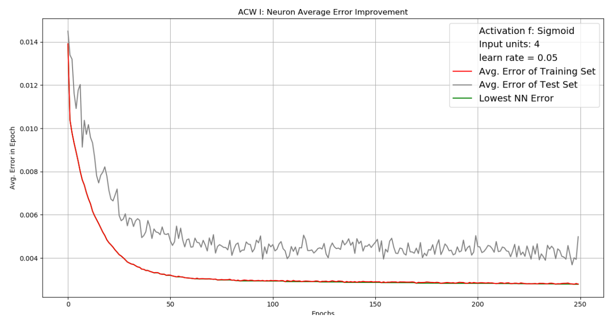
(a) The neurons average error is decreasing over all samples.



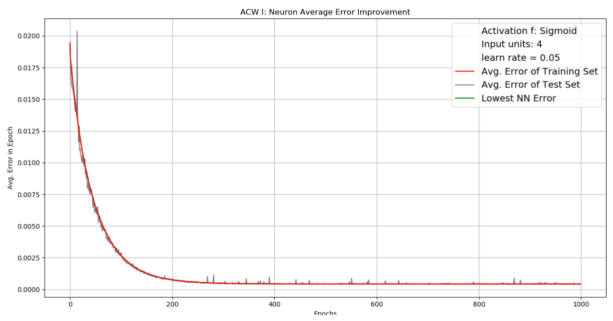
(b) The predicted position of the neuron is close to the real position of the sample data after training.

Figure 4: Learning curve and outputs of a neuron with a sigmoid activation function.

In general the predictions of a neuron for a training set become more accurate the longer one trains it with the same set. In the beginning of this training the neuron will also improve its predictions for data related to the training set. With each training epoch the weights will become better in predicting particular data while their ability to generalise will decrease. By partitioning our data into trainings and test sets we can validate how good the fit of our model is and if it is *over-* or *underfitting* our data. When creating different data sets it is important to add some form of randomization. Figure 5 shows the cross validation output of a neuron which got trained with the first 2000 samples and one that got trained with shuffled data.



(a) Trainings set not shuffled



(b) Shuffled trainings set

Figure 5: Effect of shuffling the data set before cross validation learning.

Adapting the neuron to data and velocity

The system that the neuron is predicting seems to switch between three different states. Depending on the number of inputs and the training one can observe that the predictions of the neuron are some samples behind the real output. The reason might be that the neuron realises the state transition of the system only step by step.

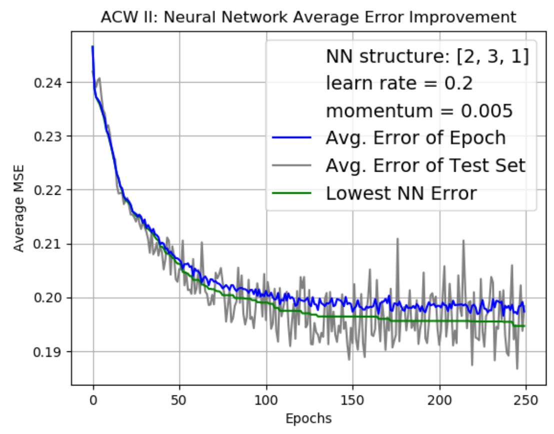
In order to additionally predict the velocity one need another neuron for the extra output. As the velocity depends on the position it could be useful to give the position output as an input to the velocity neuron. A multi-layered network should be helpful in this case, as it could model the dependency of position and velocity.

ACW Part II

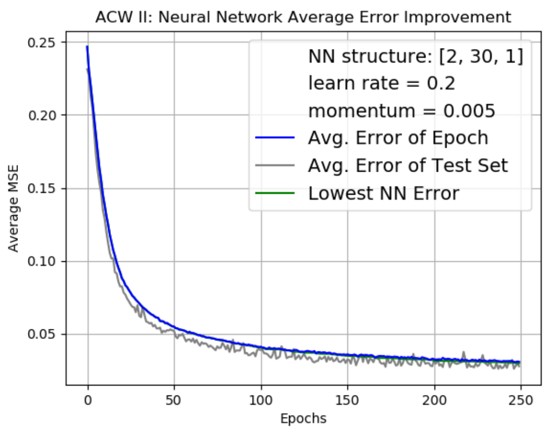
For the second part of this report a neural network with back propagation has been implemented using python. The code can be seen in the attached file `neural_acw_part2.py`. The input weights were initialized with normal distributed random values between 0 and 1.

Varying the number of hidden units and layers

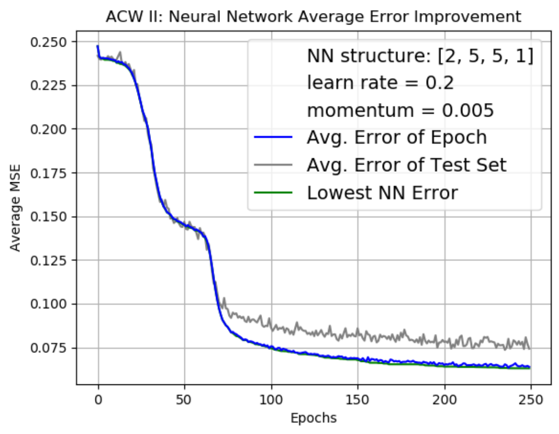
Changing the number of units in a hidden layer has an impact on the time needed for training a network, on the improvement through learning and on the probability of learning at all. Figure 6(a) and 6(b) show that in average networks with 30 hidden units learn and perform better than networks with 3 units. The networks with more units decreased their average error by ∼0.23 over 250 epochs while the networks with 3 units decreased their error only by ∼0.04. Training a lot of different weights might take a longer time, but seems more efficient as the gradient of many different weights are more likely to lead into the right direction. This could be verified by all examples that compared networks with different numbers of hidden units (see figure 6(c) and 6(d)). Having less weights means that we have fewer modifiable options and are more likely to get stuck in a minimum. Networks with to few hidden neurons don't have the capacity to learn enough of the underlying patterns. At one point networks with too many nodes (> ∼130 units) didn't improve any further.



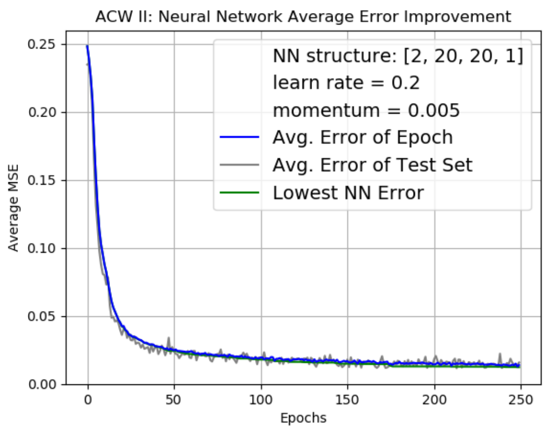
(a) Avg. outputs of 5 NN with 3 hidden units.



(b) Avg. outputs of 5 NN with 30 hidden units



(c) Avg. outputs of 5 NN with 2 x 5 hidden units.

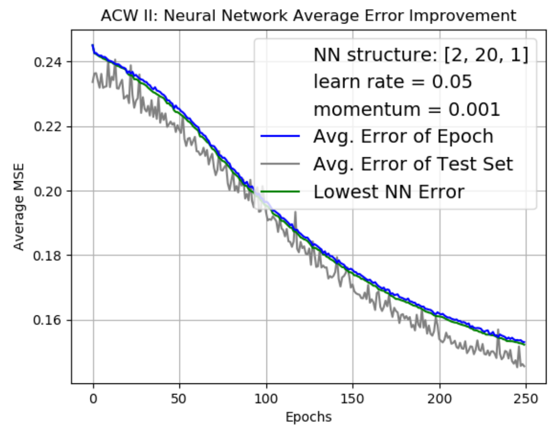


(d) Avg. outputs of 5 NN with 2 x 20 hidden units

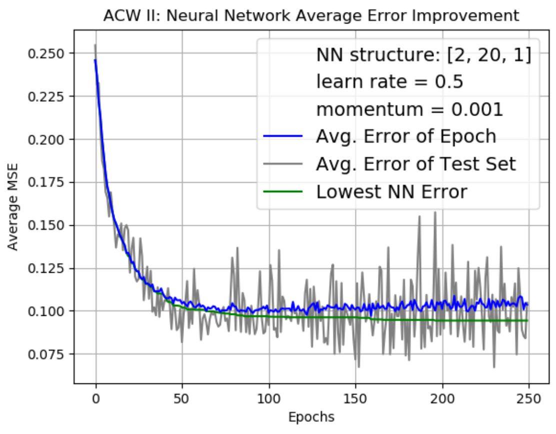
Figure 6: Averaged error improvement of networks with different hidden units.

Varying the learning rate

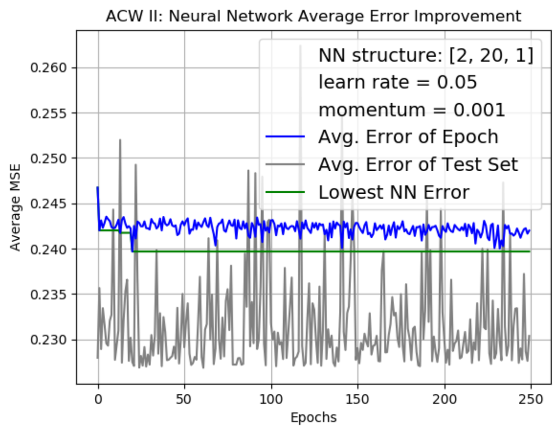
The learning rate defines the step size with which we exploit the calculated gradient. If the step size is very small we converge slowly towards a minimum and might not use our full learning potential. It is also more likely that we get stuck in a small local minima. If the step size is too large we might overshoot the minimum (see figure **??**)and might start oscillating. Figure **??**



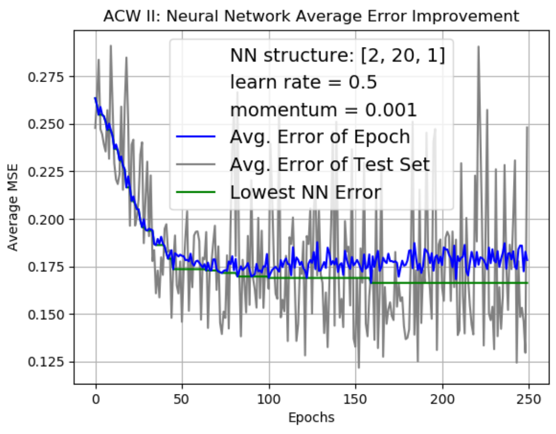
(a) Avg. error of a 5 NN with learning rate of 0.05



(b) Avg. error of a 5 NN with learning rate of 0.5



(c) Example with low learning rate that got stuck in a local minima.

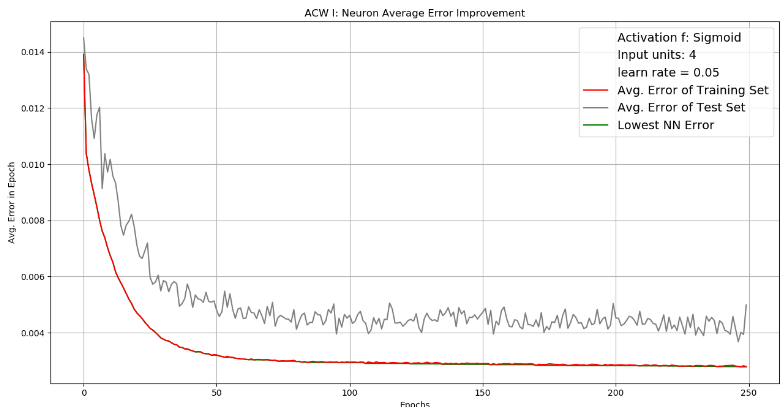


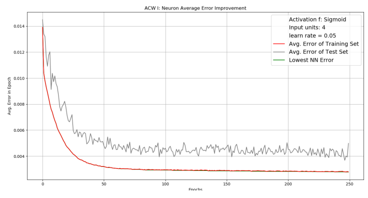
(d) Example with high learning rate that starts oscillating around the minimum.

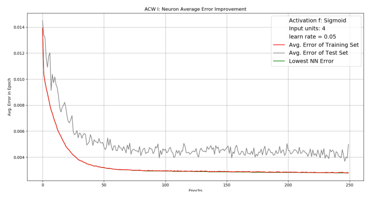
Figure 7: Aver.

The effect of momentum

By adding momentum to our back propagation we
discussion on the effect of a momentum term
the nature of the error as you vary the number of hidden nodes
discussion and Analysis (e.g Compare the performance of the best and worst networks)
Suggestions for further improving performance (e.g use RBFs and sample result)


ACW I: Neuron Average Error Improvement
Activation f: Sigmoid
Input units: 4
learn rate = 0.05
Avg. Error of Training Set
Avg. Error of Test Set
Lowest NN Error
Avg. Error in Epoch
Epochs


ACW I: Neuron Average Error Improvement
Activation f: Sigmoid
Input units: 4
learn rate = 0.05
Avg. Error of Training Set
Avg. Error of Test Set
Lowest NN Error


ACW I: Neuron Average Error Improvement
Activation f: Sigmoid
Input units: 4
learn rate = 0.05
Avg. Error of Training Set
Avg. Error of Test Set
Lowest NN Error