Assignment 3

Computational Intelligence, SS2020

Team Members		
Last name	First name	Matriculation Number
Blöcher	Christian	01573246
Bürgener	Max	01531577

1 Regression with Neural Networks

1.1 Simple Regression with Neural Networks

1.1.1 Learned Function

Figure 1 shows "typical" (slight variations occur due to non-fixed random seeds) results of functions learned through regression with $n_h = 2, 5, 50$ neurons in one hidden layer. Using only 2 neurons causes underfitting and leads to a poorly approximated function for both training and test sets. With $n_h = 5$ yields better results for both training and testing sets, but especially for the testing set the approximated function is not entirely accurate. Choosing $n_h = 50$ leads to overfitting: the output function fits the training data almost perfectly but barely resembles the testing set.

1.1.2 Variability of the performance of deep neural networks

Table 1 shows the minimum, maximum, mean, and standard deviation of the MSE obtained on the training set for $n_h = 5$ and seeds $0 \dots 9$. The minimum MSE on the training set is achieved with seed 3, for the testing set however the best seed is 7. With a validation set one could run the fitting process for multiple seeds, then choose the seed with the smallest MSE on the validation set. Now that the risk of overfitting has been reduced it is possible to make a better prediction on the testing set.
The variability of the MSE across seeds is expected, because a non-linear activation function (sigmoid $\sigma(x)$) is used. This causes the cost function to have multiple (local) minima instead of just one global minimum. Which of the minima the algorithm converges on depends on the random weight initialization, leading to different MSEs. This occurs for both SGD and standard GD. Another source of randomness introduced by SGD is linked to its weight update: instead of using all training samples for each weight update only one randomly chosen sample is used for each update. That means SGD typically needs more iterations to converge because the path to the minimum is not as direct as with GD, but because only one sample is used per iteration the computational cost is greatly reduced, leading to faster training.

min	max	μ	σ
0.032	0.263	0.153	0.068

Table 1: Minimum, maximum, mean, and standard deviation of training set MSEs with $n_h = 5$.

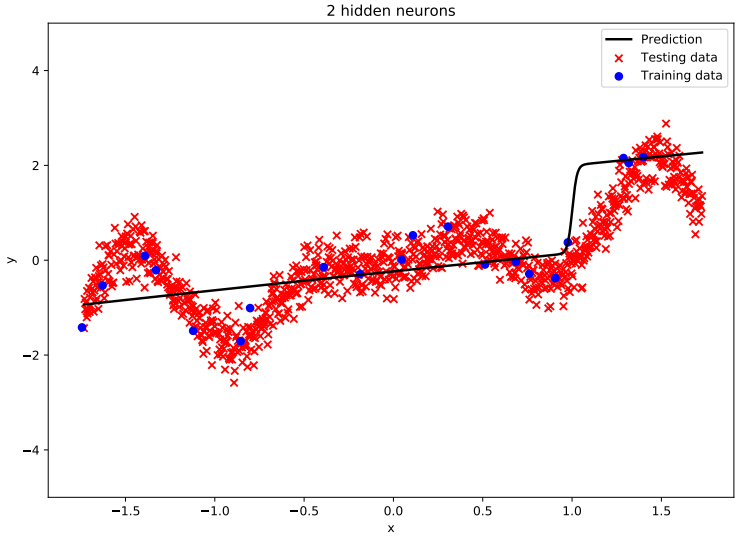
1.1.3 Varying the number of hidden neurons

Figure 2 shows the mean and standard deviation of the MSEs for both training and test data. There are two notable peaks in the testing MSE at $n_h = 6$ and $n_h = 12$, which are caused by discontinuities or large peaks in the approximated functions between training samples (e.g. s. figure 3b for worst learned function with $n_h = 6$) and which will be disregarded further on. With increasing number of hidden neurons the training cost decreases, as the training data gets better approximated. This leads to underfitting and a higher testing MSE for small values of n_h. The best result on the testing set averaged over the runs with different random seeds is achieved with $n_h = 4$ (s. figure 3a for best learned function with $n_h = 4$). For larger values of n_h the testing cost increases again due to overfitting.

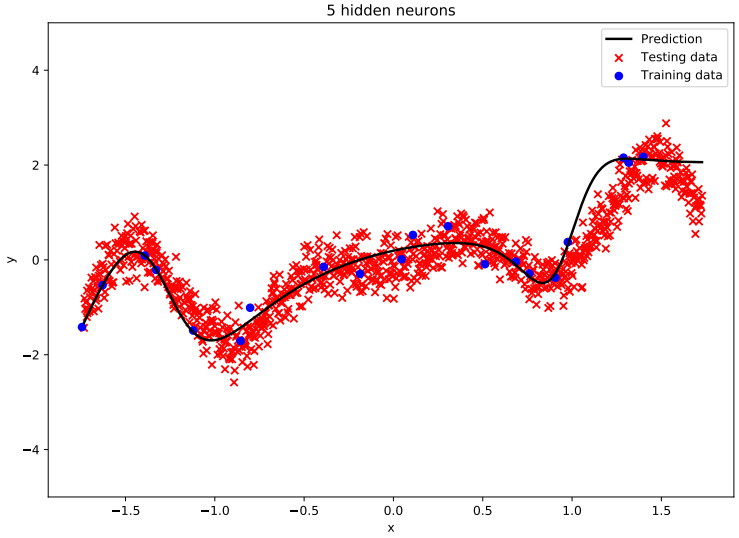
1.1.4 Change of MSE during the course of training

Figure 4 shows the MSE during the course of training for $n_h = 2, 5, 50$. Because of the random initialization all maximum MSEs for training and testing data are at the very beginning of the update loop. As the weights get fitted on the training data all training MSEs decrease subsequently and converge at around 0.2, 0.05, and 0.3 respectively. After an initial drop the testing MSEs rise to a local maximum again, which is reached the faster the smaller n_h. Then the testing MSEs decrease again until they converge at around 0.37, 0.23, and 0.5 respectively, making $n_h = 5$

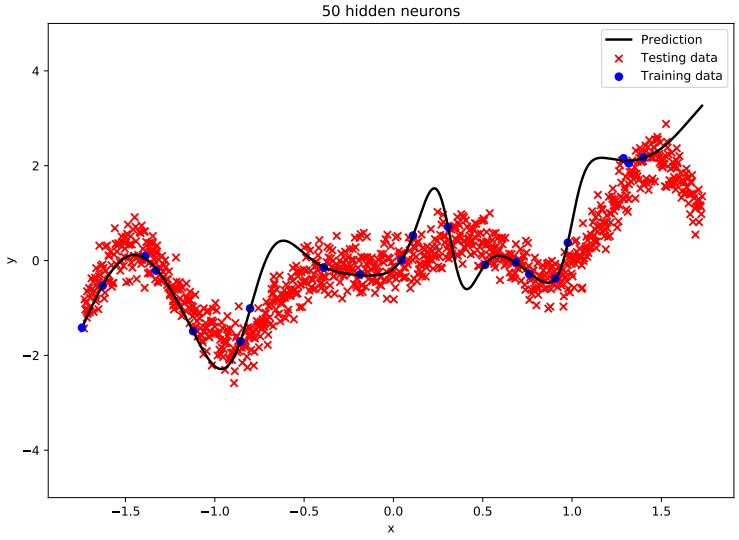
the best number of hidden neurons. As the testing MSE is the largest with $n_h = 50$ the risk of overfitting increases with the number of hidden neurons.
By using the SGD one can reduce overfitting: the random selection of the samples for the weight update causes the path to the minimum of the cost function to be noisy. Additionally by reducing the computational cost one can increase the size of the training set, which generally reduces the risk of overfitting.



(a) $n_h = 2$



(b) $n_h = 5$



(c) $n_h = 50$

Figure 1: Results of Regression with varying number of neurons n_h.

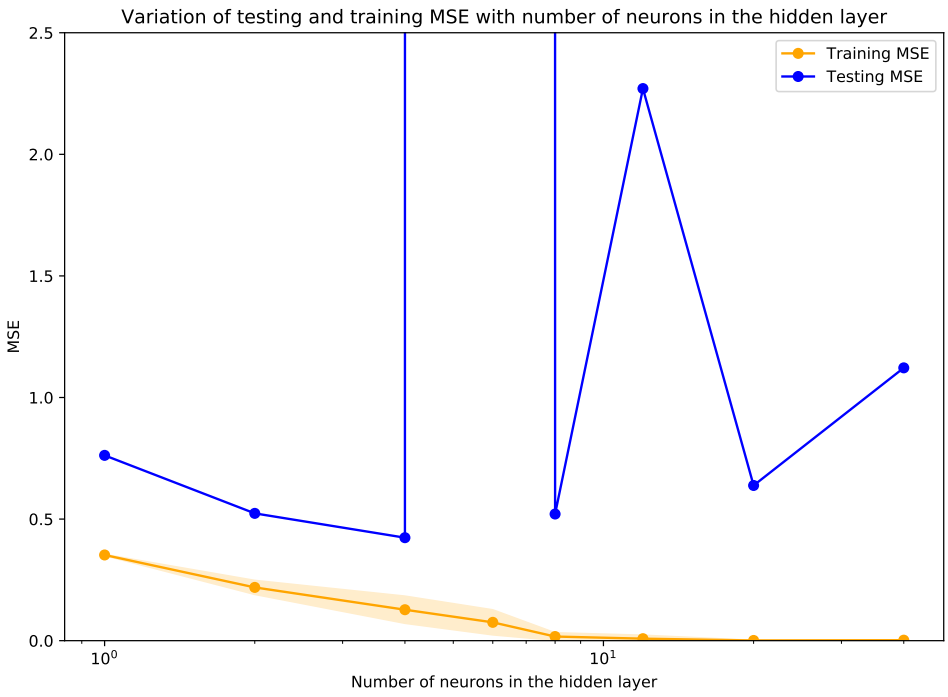
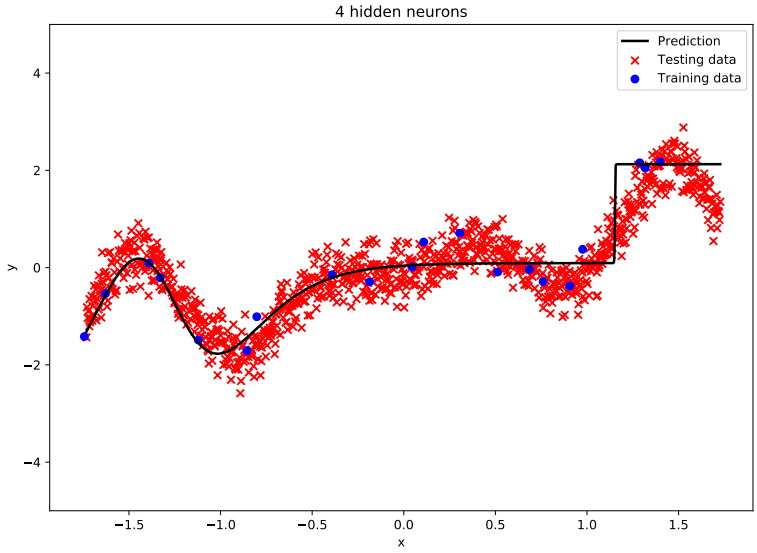
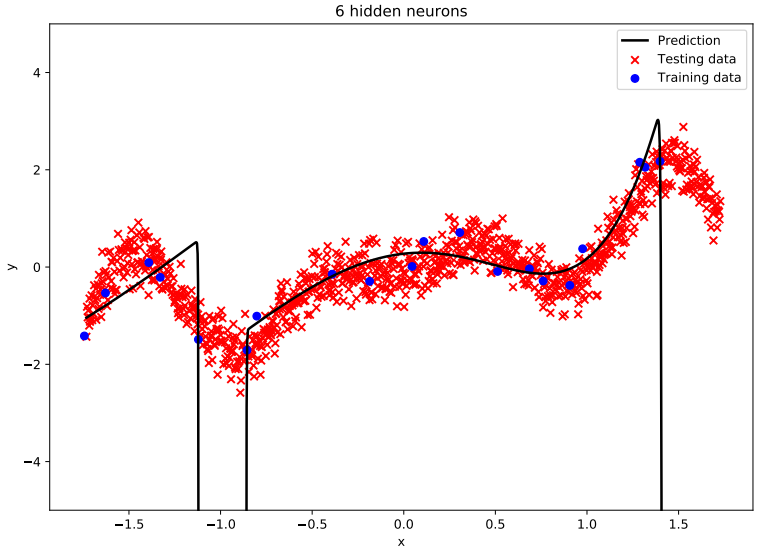


Figure 2: Variation of testing and training MSE for varying n_h.



(a) Best result: $n_h = 4$



(b) Worst result: $n_h = 6$

Figure 3: Best and worst result of regression with varying n_h.

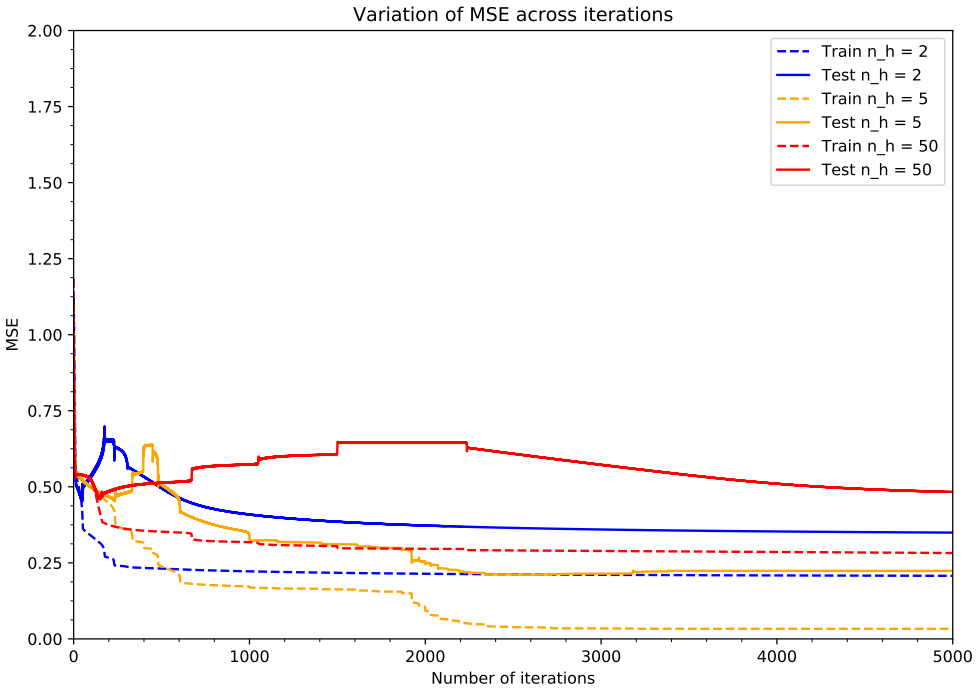


Figure 4: Variation of testing and training MSE across iterations with varying n_h.

1.2 Regularized Neural Networks: Weight Decay

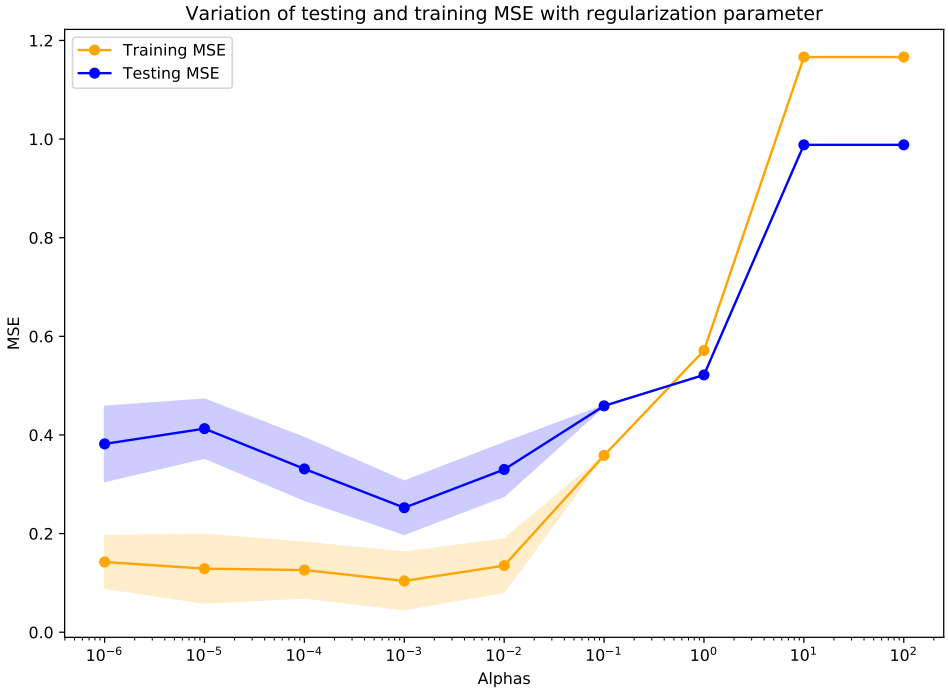


Figure 5: Variation of testing and training MSE with regularization parameter α.

2 Classification with Neural Networks: Fashion MNIST

- The confusion matrix

	T-Shirt/top	Trousers	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot
T-Shirt/top	841	3	12	25	4	1	108	0	6	0
Trousers	7	960	2	21	5	0	4	0	1	0
Pullover	19	0	835	15	52	0	78	0	1	0
Dress	21	4	14	904	23	0	31	0	3	0
Coat	2	1	104	30	797	0	65	0	1	0
Sandal	1	0	0	1	0	955	0	23	2	18
Shirt	104	1	75	26	45	1	741	0	7	0
Sneaker	0	0	0	0	0	15	0	958	0	27
Bag	5	0	5	6	2	3	4	4	970	1
Ankle boot	1	0	0	0	0	9	0	3	0	960

Table 2: Confusion matrix for Fashion MNIST Classification

The rows of our confusion matrix correspond to the true classes and the columns to the predicted classes. The global accuracy for all classes is fairly good with $89,2\%$. Apparently it was difficult for the code to distinguish t-shirts/tops from shirts and coats from pullovers. Also remarkable is that all top clothes were often mistakenly recognized as shirts.

- Are particular regions of the images get more weights than others?

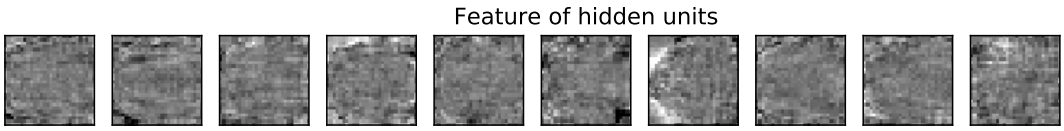


Figure 6: Weights per pixel for 10 samples

In figure 7 are the weights per features visualized. As for the features these are referring to one pixel of a 28 x 28 pixel conversion from each picture. The weights are scaled from important regions (white) to regions with less information (black). It makes sense and can be seen in the pictures that that the object edges are important since all picture have the black background in common and the object outlines vary a lot.

- Boxplot of classifications with different inital weight vector values

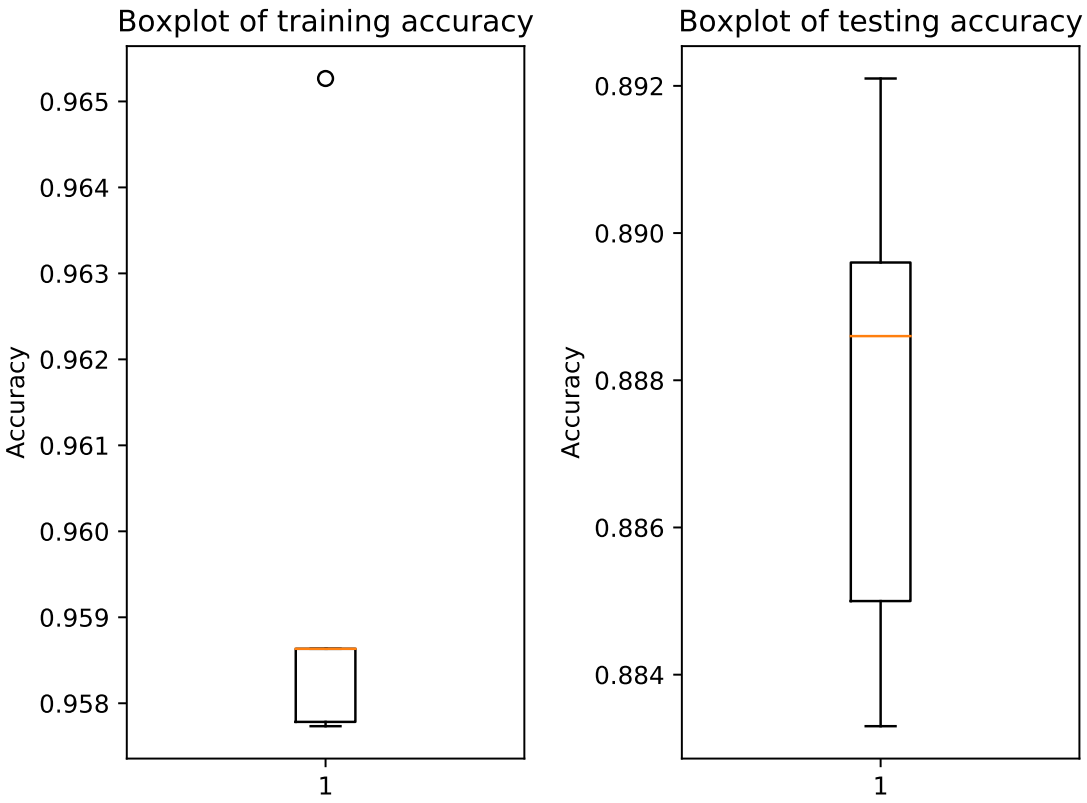
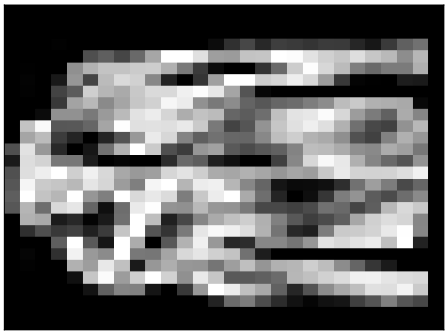


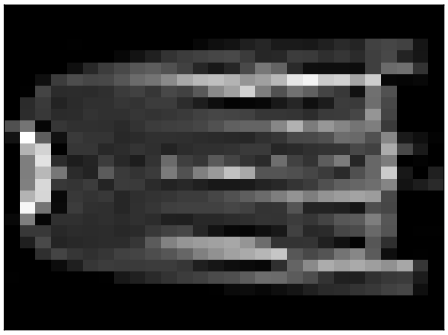
Figure 7: Accuracy of classifications using different inital **w**

After choosing the optimal amount of hidden layers the boxplots visualize the accuracy of classifications which started with different weight vectors **w**. The orange coloured line shows the median of our calculated numbers. The lower border of the box is enclosing the median of all numbers which are lower than our global median. The upper border is enclosing all numbers till the median of all higher numbers. The last section is showing the global minimum and maximum. Any outliers are marked with a spot.

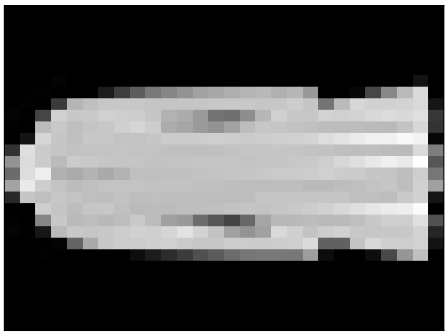
- Missclassified items



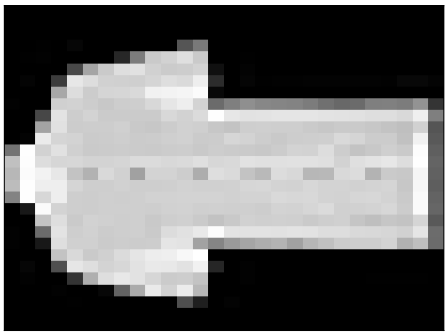
(a) Missclassified image 1



(b) Missclassified image 2



(c) Missclassified image 3



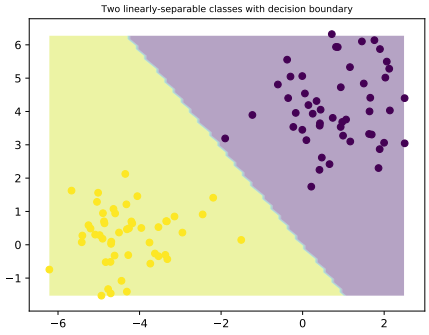
(d) Missclassified image 4

Figure 8: Missclassified items from our test set

3 Bonus: Implementation of a Perceptron

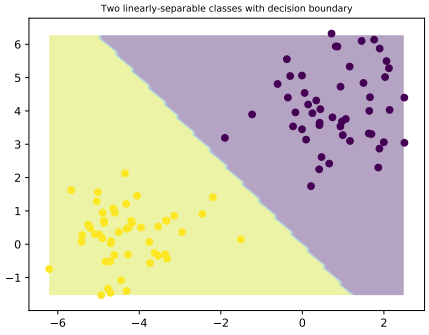
- Linear separable dataset:

(a) self implementation

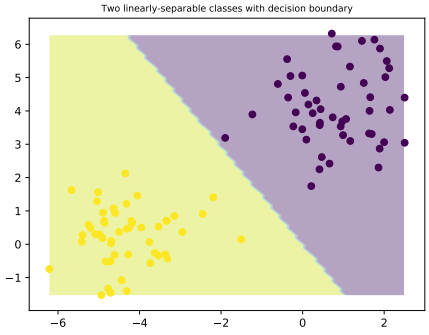


(b) scikit-learn's implementation

Figure 9: Classification using eta = $0,01$ and max. iterations of 2

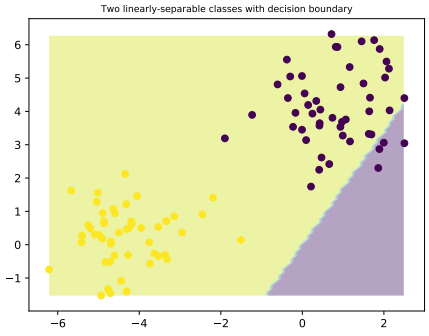


(a) self implementation

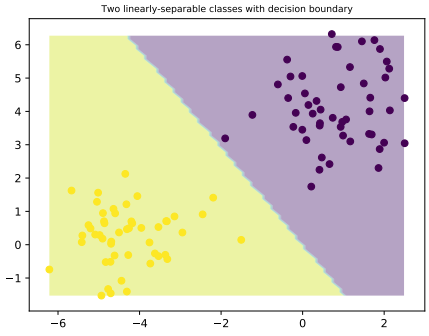


(b) scikit-learn's implementation

Figure 10: Classification using eta = $0,01$ and max. iterations of 5



(a) self implementation



(b) scikit-learn's implementation

Figure 11: Classification using eta = $0,001$ and max. iterations of 5

The classification of the dataset was similar for both implementations. Only using small eta's $< 0,001$ our code began to misclassify the whole dataset.

- How many training iterations does it take for the perceptron to learn to classify all training samples perfectly?

  Our own implementation needs in average 2 iterations to classify the linear dataset perfectly. This make sense since we're using random initial weights for just two different states.

- The misclassification rate for both datasets

Training MSE:	0,475
Test MSE:	0,5

Training MSE:	0,475
Test MSE:	0,5

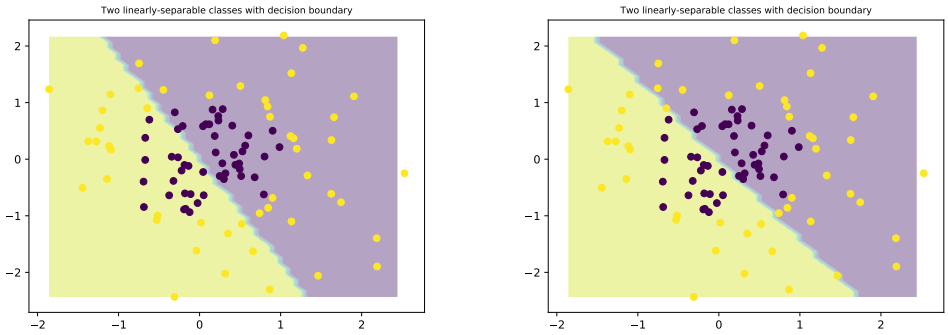
Table 3: MSEs using eta $= 0,01$ and 5 iterations self impl.(l.) skl.impl.(r.)

Training MSE:	0,525
Test MSE:	0,55

Training MSE:	0,475
Test MSE:	0,5

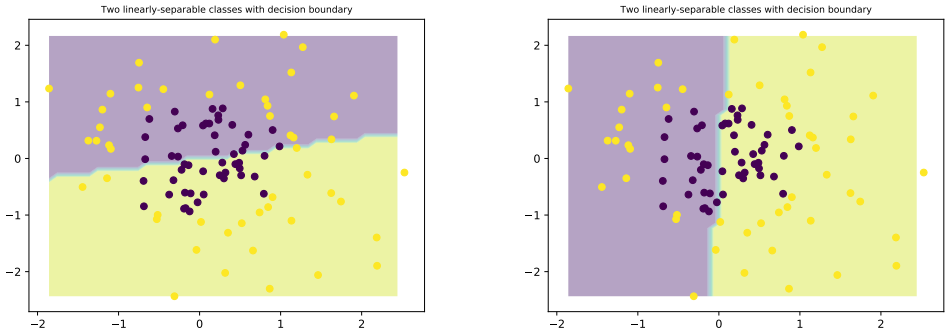
Table 4: MSEs using eta $= 0,1$ and 100 iterations self impl.(l.) skl.impl.(r.)

- Non linear seperable dataset:



(a) self implementation (b) scikit-learn's implementation

Figure 12: Classification using eta $= 0,01$ and max. iterations of 5



(a) self implementation (b) scikit-learn's implementation

Figure 13: Classification using eta $= 0,1$ and max. iterations of 100

Both implementations can not classify the non linear dataset.

- How would you learn a non-linear decision boundary using a single perceptron?

  We should use a non linear activation function i.e. the sigmoid function with logistic regression as we used it in our homework 2. Using a suiting degree and learning rate the classification could be much better.