Assignment 3

Computational Intelligence, SS2020

Team Members		
Last name	First name	Matriculation Number
Reindl	Hannes	01532129
Samer	Philip	01634718
Rösel	David	01634719

Contents

1 Regression with Neural Networks

1.1 Simple Regression with Neural Networks

1.1.1 Learned function

The goal of this exercise was to show the regression function in comparison to the number of neurons used, which in this task was either 2, 5 or 50. Depending on the number of neurons there can be the problem of over or under fitting which should also be explored in this context. The training of the neural network should be done with a random seed.

Overfitting occurs when using a too complex model to rebuild a simple function (in this case the model with 50 neurons). This will still benefit the performance on the training set, but increase the error on the test set, since it will try to fit the curve to each training sample.

Underfitting occurs when the model complexity to fit a function to given data is too simple (in this case the model with 2 neurons). For example if you have a non linear equation and try to model it with a linear model the error will be very high.

Include demonstrative plots of the learned function and the actual function for all values of n_h. The following figure shows a typical good and bad fit for $n_h = 2$.

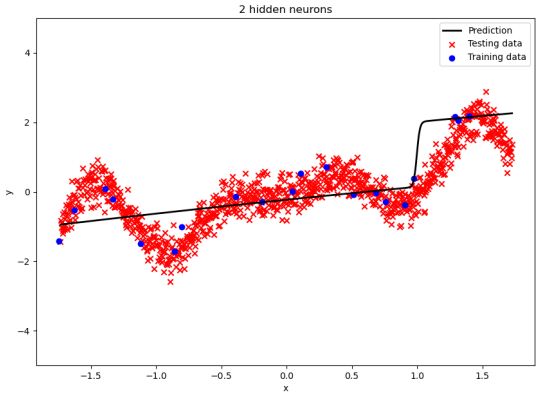

Figure 1: poorly fitted function for 2 neurons

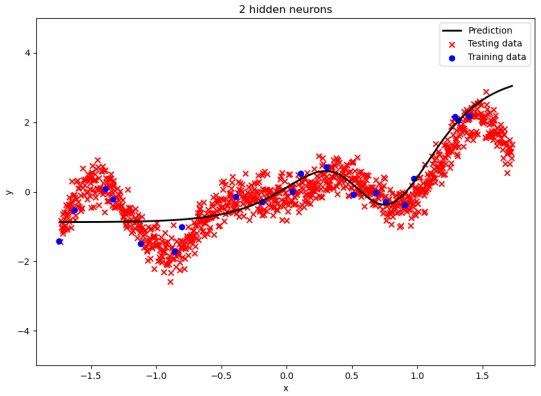

Figure 2: good fitted function for 2 neurons

As one can see this is a perfect example of under fitting. The fitted function can not follow the overall trend - the model complexity is too low.

The following figure shows a typical good and bad fit for $n_h = 5$.

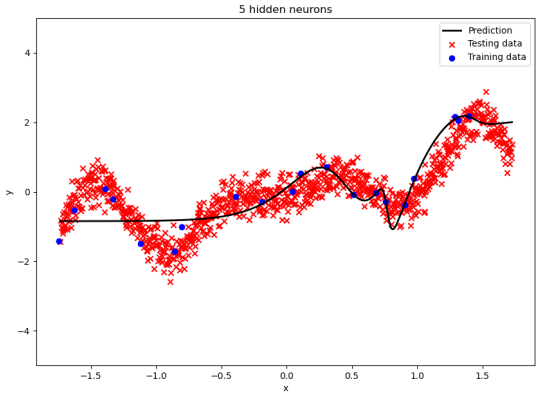

Figure 3: poorly fitted function for 5 neurons

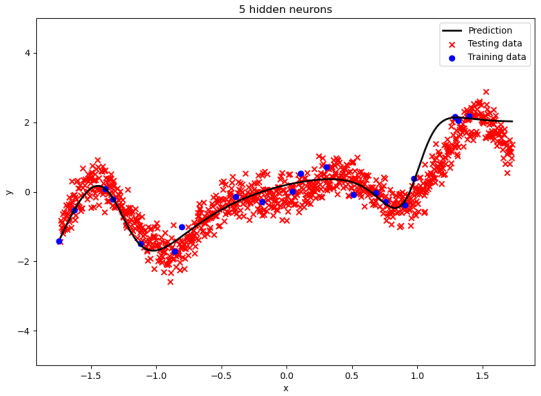

Figure 4: good fitted function for 5 neurons

Out of the given number of neurons this is by far the best value where neither over- nor under fitting occurs.

The following figure shows a typical bad and good fit for $n_h = 50$.

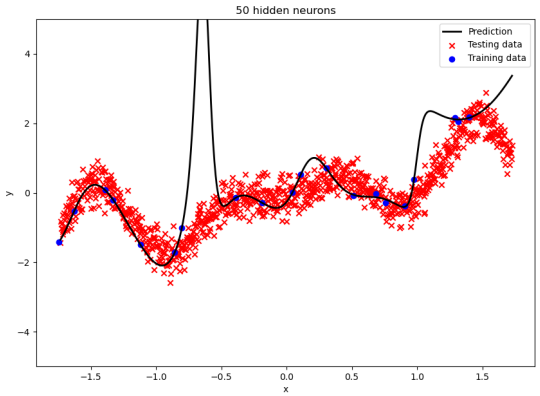

Figure 5: poorly fitted function for 50 neurons

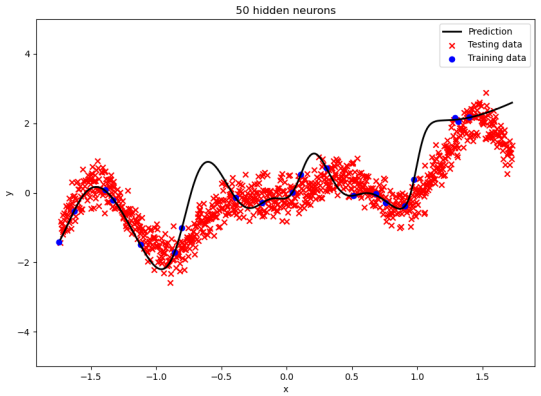

Figure 6: good fitted function for 50 neurons

As one can see this is a perfect example of over fitting, as the function that the network is trying to create does not represent the actual data. It tries to fit a curve through all training points (blue dots) to decrease the MSE_{train}.

Interpret your results in the context of under/over fitting.

As we expected we have underfitting for a too small number of neurons as seen for 2 hidden neurons, a good fit for 5 hidden neurons and overfitting for 50 hidden neurons.

1.1.2 Variability of the performance of deep neural networks

In this example we have to write a function that calculates the MSE as well as test it using 10 random seeds and looking at the the values.
The formula to calculate the MSE is as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2 \tag{1}$$

What is the minimum, maximum, mean and standard deviation of the mean squared error obtained on the training set?

Seed Nr.	min	max	mean	standard deviation
0-9	0.0325	0.262	0.152	0.0676

Is the minimum MSE obtained for the same seed on the training and on the testing set?
No, the minimum MSEs are not on the same seed. The Minimum values for each set are displayed in table 1, marked in **bold**. The training set is not directly linked or bound to the test set so it wouldn't make sence why the minimum should be at the same seed.

Seed	MSE_{train}	MSE_{test}
0	0.262	0.560
1	0.151	0.497
2	0.151	0.491
3	**0.032**	0.249
4	0.200	0.537
5	0.194	0.529
6	0.151	0.487
7	0.035	**0.240**
8	0.151	0.488
9	0.188	0.350

Table 1: Caption

Explain why you would need a validation set to choose the best seed.
A validation set is needed to choose the best seed since you would have to test the seeds on a set which is not the testset otherwise you would indirectly optimise for the testset and not generally.

Unlike linear-regression and logistic regression, even if the algorithm converged, the variability of the MSE across seeds is expected. Why?
It all depends on how the model is chosen. We basically get the values by using a gradient descend function which gives us the minimum, however it is only a local minimum which doesnt guarantee the best model. It all depends on the initial value of the weights or in this case the chosen seed.

What is the source of randomness introduced by Stochastic Gradient Descent (SGD)? What source of randomness will persist if SGD is replaced by standard Gradient Descent?
By using SGD you calculate the gradient for 1 sample at a time, the sample changes in each iteration. This means the Gradient is not optimal, however it is much faster since it only calculates with 1 sample and not the entire test set. The sample is chosen randomly which adds another randomness component, however the randomness of in which local minimum(as described above) the function

ends up will not change, it only changes the way he descends downwards to the valley.

Analogy:
SGD is a drunk man going down a hill with fast steps. He will strife to the left and right a little, but will end up in the valley eventually.

GD is a man who precisely calculates each step he takes downwards to the valley. He needs a lot of time to calculate each step but takes the steepest path downwards each step. q

1.1.3 Varying the number of hidden neurons

Include plots of how the MSE varies with the number of hidden neurons.

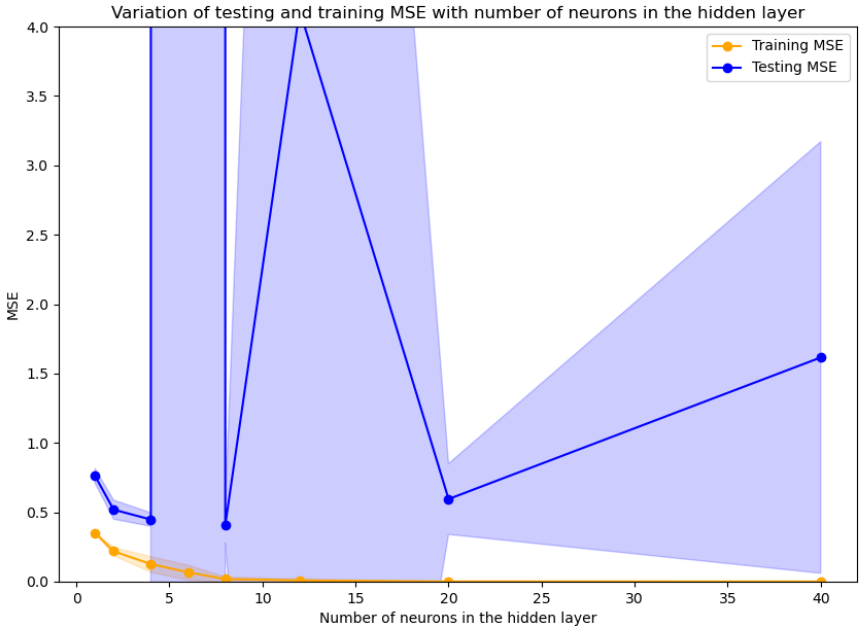

Figure 7: MSE for different Nr. of neurons

We expected a concave function like mentioned in the lecture. With too few neurons there would be underfitting, thus a high MSE. Increasing the number of neurons (i.e. model complexity) the MSE lowers until its minimum. Increasing the number of neurons even further we expect overfitting and thus a high MSE.

Since the result in figure 7 does not display this, we looked at each individual plot for all of the 10 random seeds by using the plot_learned_function(). There we noticed that most plots fitted very well but in one or two cases there was a small part where fitted curve went off the charts. This causes a high MSE and can be seen in figure 8.
These were rare but massive statistical outliers which completely ruined the value of the mean error and the variance. We tried replacing the mean function with a median function which displayed the MSE in a better statistical manner. But we assumed we shouldn't modify the mean function and reverted everything back.

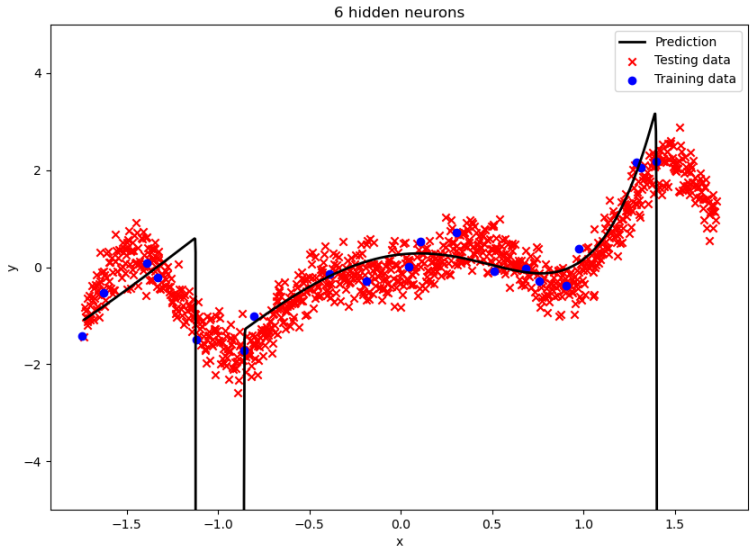

Figure 8: Off the chars fitted curve which increases the MSE substantial

What is the best value of n_h independent of the choice of the random seed? Use errors that are averaged over runs with different random seeds.

This would only be possible if we average over a nearly infinite amount of random seeds. We compared two different seed ranges(from 0 to 9 and from 10 to 20), which resulted in two different $n_{h,best}$. For the range from 0 to 9 this resulted in $n_h = 8$ to be the best neuron amount.

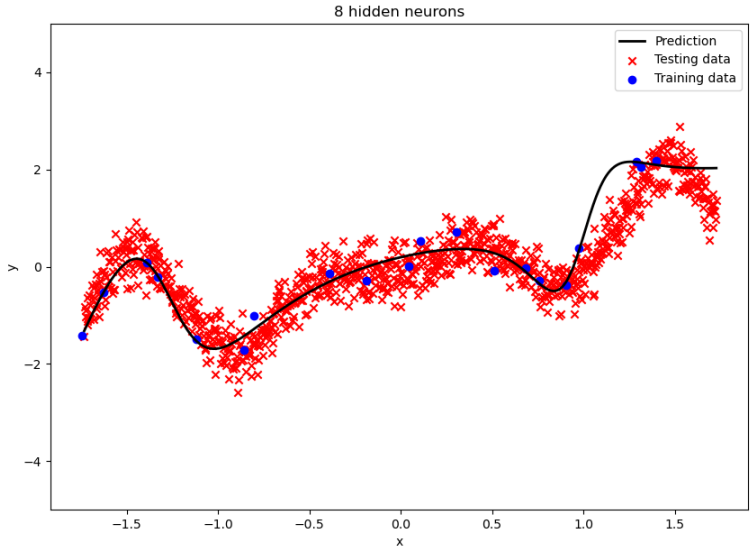

Figure 9: Function for the best model $n_h = 8$

1.1.4 Change of MSE during the course of training

Include the plot of how the MSE varies during the course of training with the three different number of hidden neurons.

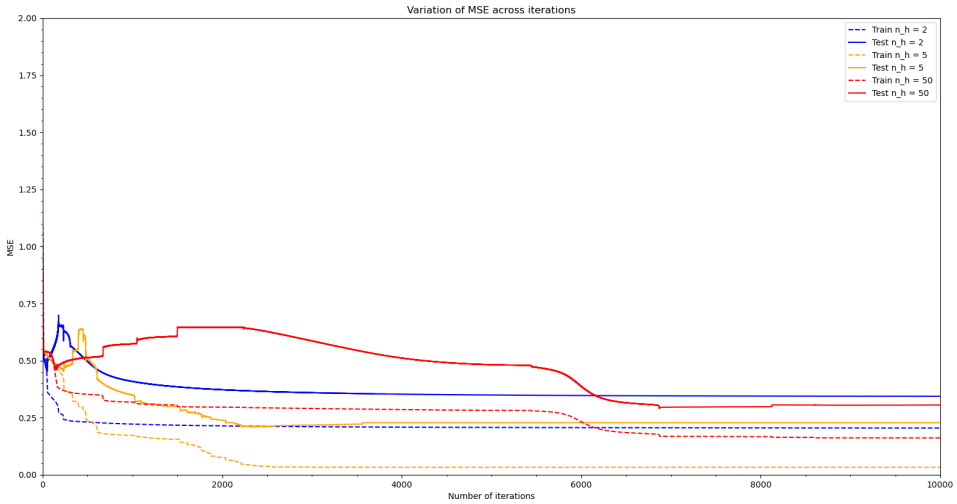

Figure 10: MSE depending on Nr. of Iterations, Solver=lbfgs

Figure 10 shows the change of the MSE over the number of iterations. As expected the overall trend for all three neuron amounts decreases. One strange thing is the high MSE on the training set for $n\_hidden = 50$. Since the model order is the highest we expected to fit the training set the best and have the lowest MSE on the training set. But this is not the case and we cant explain why this happens. This also contradicts the graphic seen in 1.1.3 where the MSE of the training set decreases for higher number of neurons. We also created an MLPRegressor which allowed for 50 iterations each call, where the MSE for the 50 hidden neurons behaved as expected.
Since the task description asked how SGD helps with overfitting, we also plotted the MSE vs iterations for the SGD-solver (figure 11) and described in the last question of this task.

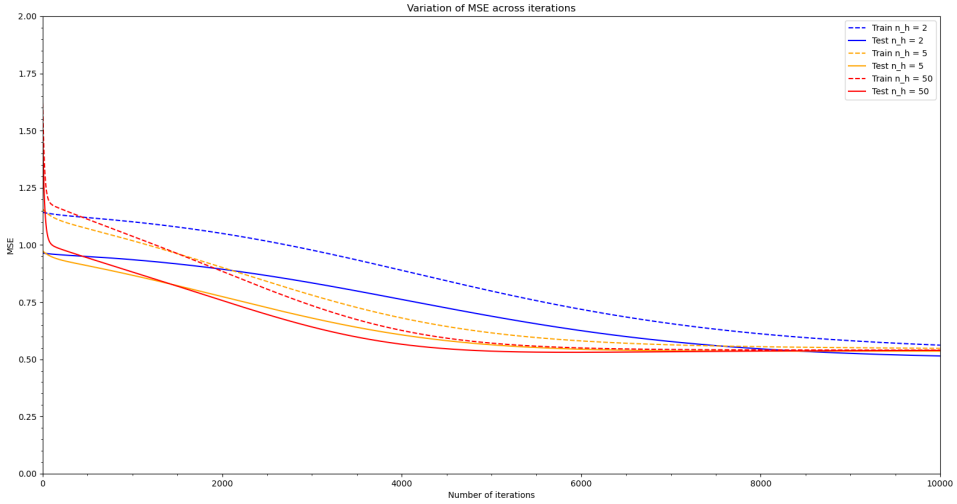

Figure 11: MSE depending on Nr. of Iterations, Solver=SGD

Does the risk of overfitting increase or decrease with the number of hidden neurons?
The risk function depends on two types of errors. The estimation error which is related to overfitting and the approximation which is related to underfitting. Increasing the model order allows to decrease the bias (approximation error) but it increases the estimation error (Variance).
Assuming we are overfitting in the first place, the risk of overfitting increases with the amount of hidden neurons.

What feature of stochastic gradient descent helps to overcome overfitting?
SGD only uses one data sample to calculate the gradient. Hence it is not able to find the true gradient which fits a curve to all given samples and therefore does not cause oscillations or extreme peaks (like usual overfitting). SGD only changes to curve in such a way that the new model fits better for the one sample it used to calculated the gradient. It does this every iteration and therefore generalizes the curve - preventing overfitting.
The plot in figure 11 shows this quite nicely. The 50 neurons line (which should overfit) for the test and training set are quite close to each other, unlike in figure 10 .

1.2 Regularized Neural Networks: Weight Decay

In this example we try to regulate the complexity of the model by adding a term to the calculation of MSE. Depending on the value of α the models are punished for having many and large values for their weights, which can help finding the optimal model. The formula to calculate the regulated MSE is as follows:

$$MSE_{reg} = \frac{1}{2n}\sum_{i=1}^{n}((\hat{y}^{(i)} - y^{(i)})^2 + \alpha\|\mathbf{W}\|_2^2) \tag{2}$$

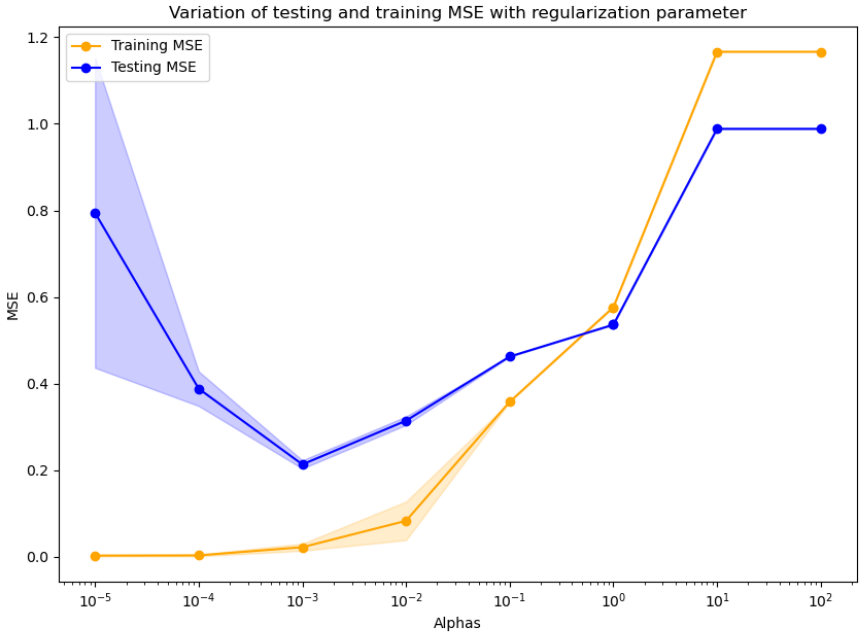

Figure 12: MSE in comparison to the values of α

What is/are the best value(s) of α?
According to figure 12 the best value for the testing set is at $\alpha = 10^{-3}$

Is regularization used to overcome overfitting or underfitting? Why?
overfitting, because it punishes complex models and therefor forcing simpler models to be more successfull(since the weight matrix $\mathbf{W}$ contains more and higher values $\Rightarrow ||\mathbf{W}||_2^2$ increases $\Rightarrow MSE\_reg$ increases as seen in the formula above)

2 Classifcation with Neural Networks: Fashion MNIST

2.1 Fashion MNIST

The goal in this exercise was to create a program to train a feed-forward neural network with one hidden layer for the Fashion MNIST. For the multi-class classification the MLPClassifier uses soft-max across the output neurons and the loss function used is cross-entropy:

$$CE_{reg} = -\frac{1}{n}\sum_{i}^{n}\sum_{c}^{C} y_c^{(i)} \ln \hat{y}_c^{(i)} + \frac{\alpha}{2}\|W\|_2^2 \tag{3}$$

Include the confusion matrix you obtain and discuss. Are there any clothing items which can be better separated than others?

The confusion matrix displays how often object i has been classified as object j. The first row displays how often first object has been classified as j-th object, where j also denotes the column. For example: The first object has been classified as the second object 3 times and 16 times as the third object. The main diagonal shows often the objects were classified correctly (for our data it would be ideally 1000).

$$\text{confusion matrix} = \begin{bmatrix} 836 & 3 & 16 & 50 & 4 & 1 & 76 & 0 & 14 & 0 \\ 3 & 947 & 9 & 35 & 3 & 0 & 2 & 0 & 1 & 0 \\ 16 & 4 & 783 & 12 & 110 & 2 & 64 & 0 & 9 & 0 \\ 27 & 10 & 15 & 882 & 29 & 2 & 24 & 0 & 11 & 0 \\ 0 & 1 & 101 & 41 & 786 & 0 & 67 & 0 & 4 & 0 \\ 1 & 0 & 1 & 0 & 0 & 936 & 0 & 33 & 5 & 24 \\ 161 & 3 & 113 & 52 & 88 & 0 & 561 & 1 & 21 & 0 \\ 0 & 0 & 0 & 0 & 0 & 29 & 0 & 929 & 0 & 42 \\ 3 & 1 & 4 & 9 & 5 & 6 & 12 & 5 & 955 & 0 \\ 1 & 0 & 0 & 0 & 0 & 10 & 0 & 38 & 1 & 950 \end{bmatrix}$$

The second ("trousers/pants") and last two objects ("bag" and "ankle boot") can be detected very well, which is displayed by the large number in the main diagonal.
An example for badly recognised object would be the row number 7 ("shirt"), because its main diagonal number is quite low (only 561). It was often mistaken for object 1 ("T-shirt/top", 161 times), object 3 ("pullover shirt", 113 times) and object 5 ("coat", 88 times).

Can you find particular regions of the images which get more weights than others?

As one can see in Figure 13 at the ninth plot. There is a white spot in the top left corner of the image. This shows that the weight values there are in comparison to the other weights very high. The values in the lower right corner show the minimum and maximum displayed value. A completely dark spot has a weight value of -1.001 and a completely white spot 8.427.

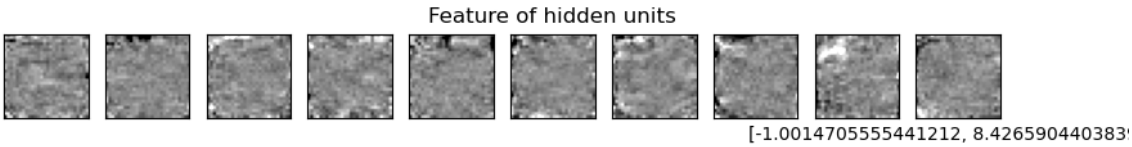

Figure 13: Weights of hidden units

Figure 14 shows the training accuracy in form of a boxplot. Since we only calculate 5 points of accuracy, each line of the boxplot SHOULD correspond to its respective calculated accuracy. But this is here not the case due to boxplot interpreting one value as a flier.

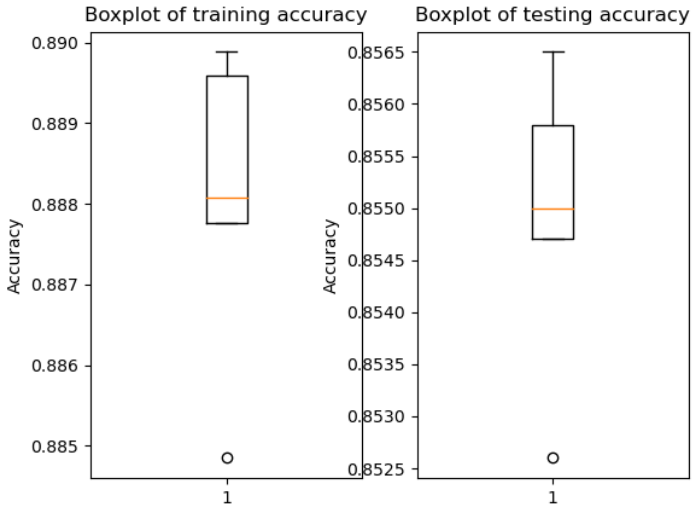

Figure 14: Boxplot

Flier points are those past the end of the whiskers and we dont know why it is displayed this way. We

checked the array we passed to the boxplot function several times, but couldnt figure it out, but since we still can see all the values displayed it doen't matter.

The following figure shows the first classification error. The picture was classified as dress, but should be a T-Shirt/Top.

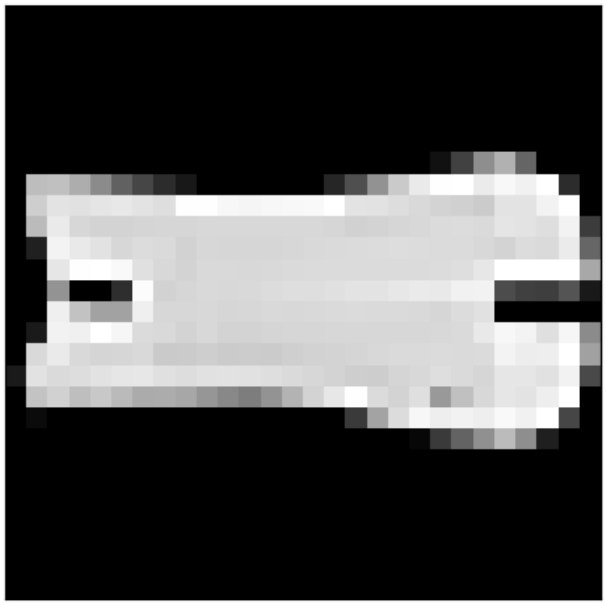

Figure 15: T-Shirt/Top falsely classified as dress

The following figures show some more classification error.

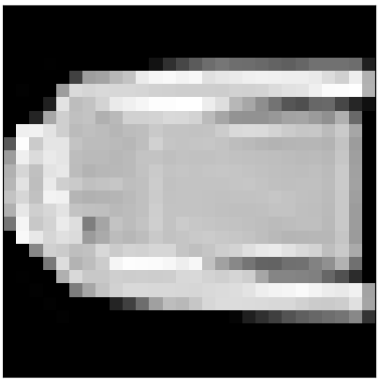

Figure 16: Coat falsely classified as pullover shirt

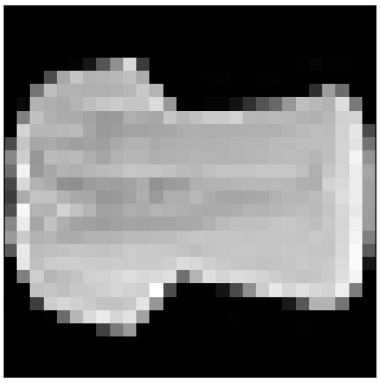

Figure 17: Shirt falsely classified as T-shirt/top

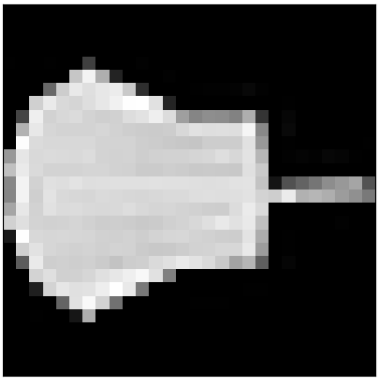

Figure 18: Shirt falsely classified as T-shirt/top

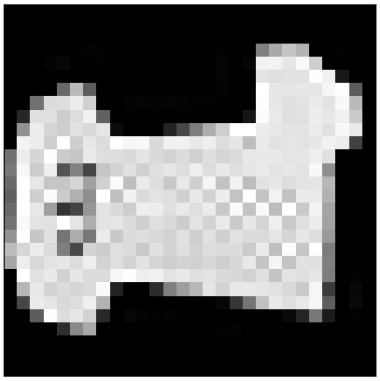

Figure 19: Dress falsely classified as T-shirt/top

3 Bonus: Implementation of a Perceptron

Include plots of the decision boundaries learned for the two datasets, two implementations, and different values of learning rates and number of iterations and briefly explain/discuss the graphs.

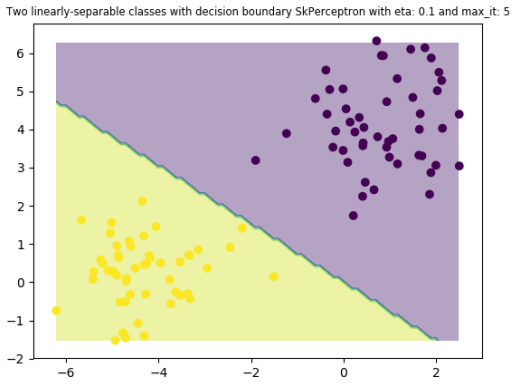

Figure 20: Perceptron SK $\eta = 0.1$ and iterations $max\_it = 5$

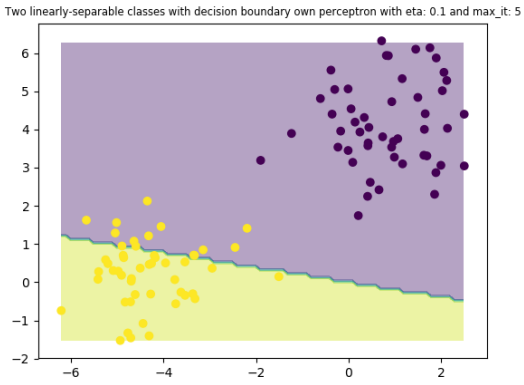

Figure 21: Perceptron own $\eta = 0.1$ and iterations $max\_it = 5$, started from zeros

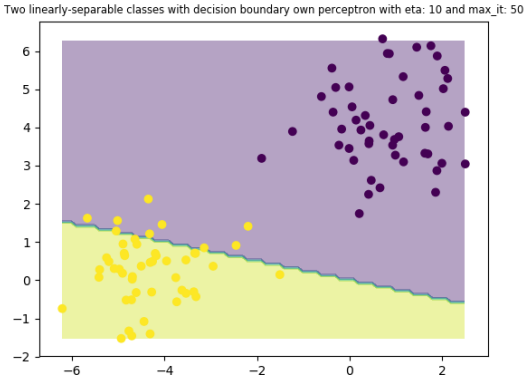

Figure 22: Perceptron own $\eta = 10$ and iterations $max\_it = 50$

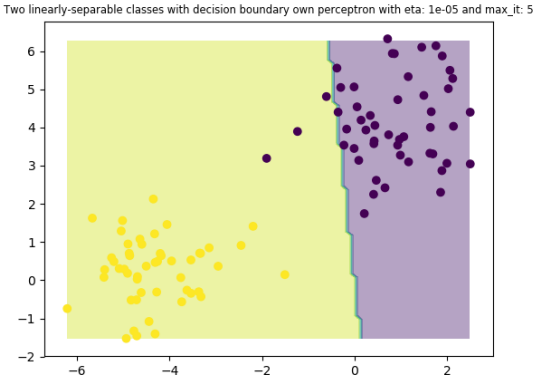

Figure 23: Perceptron own $\eta = 10^{-5}$ and iterations $max\_it = 5$

linear separable data:
The learning rate is an indicator of how much the weights change with each iteration. By having a too high value it is not possible to converge towards a valley. By having a too low value and random initialisation the weights will only change minimally after each iteration, so a much higher maximum number of iterations is needed or the convergence time is longer.

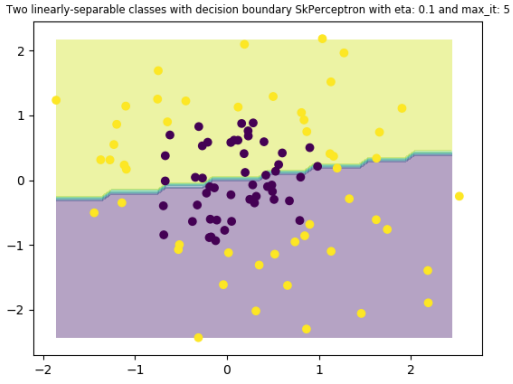

Figure 24: Perceptron SK $\eta = 0.1$ and iterations $max\_it = 5$

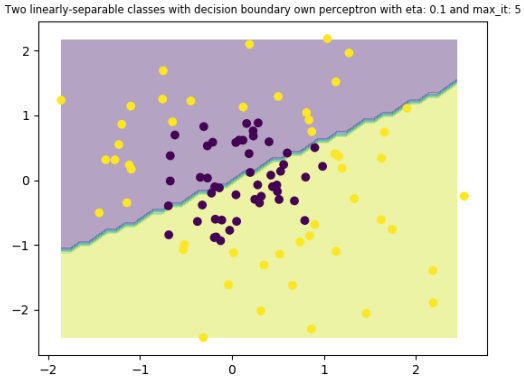

Figure 25: Perceptron own $\eta = 0.1$ and iterations $max\_it = 5$

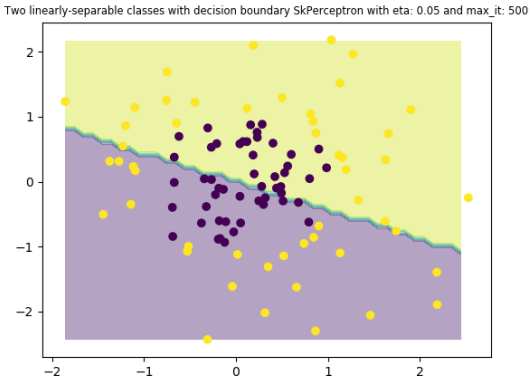

Figure 26: Perceptron SK $\eta = 0.05$ and iterations $max\_it = 500$

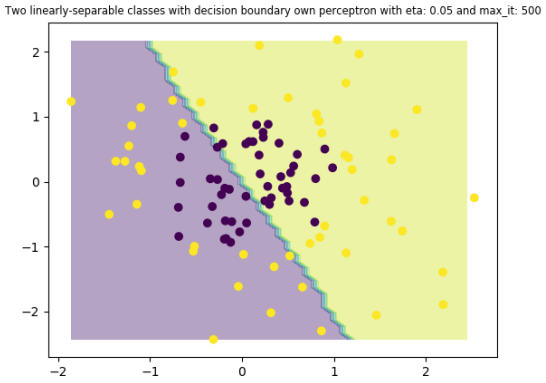

Figure 27: Perceptron own $\eta = 0.05$ and iterations $max\_it = 500$

non-linear separable data:
Neither the SciKit nor our perceptron is able to fit a good decision boundary. The single perceptron with the linear input features is not able to solve this problem. More perceptrons or non-linear input features could solve this task.

How do the results of scikit-learn's implementation of Perceptron differ from your implementation?
We tried different values for learning rate η and max_iteration for the scikit implementation and there was no visible change in its plot. No matter the values for learning rate η and max_iteration, the SciKit implementation always classified the linear-separable data correctly. Our Perceptron varied its decision boundary depending on step size and learning rate η, if the weights were randomly initialised.

NOTE: When the initial weights are equal to 0, the learning rate does not matter, since the only thing that matters is the sign of the weights due to the heaviside function as activation function and there relative size to one another. So any learning rate $\neq 0$ allows to change the sign of weights correctly. For a high learning rate, both weights get quite high, but are still proportional to each other.

For example: The resulting weights for two different learning rates are

$$w^{(1)} = \begin{bmatrix} -13.414 \\ 9.765 \end{bmatrix} \qquad \text{and} \qquad w^{(2)} = \begin{bmatrix} -1.3414 \\ 0.9765 \end{bmatrix}$$

The learning rate for $w^{(1)}$ is 10 times higher than for $w^{(2)}$.
We also tried to add a bias to the perceptron, but an assertion error generated in the file suggested, it was not supposed to be implemented.

How many training iterations does it take for the perceptron to learn to classify all training samples perfectly?
It depends on the settings of η as well as if we use an initial weight of zero or a random value. If the initial weight is zero, η only scales the weights and doesn't influence the number of iteration needed. For initial weights equal to zero it takes 6 iterations every time.

What is the misclassification rate (the percentage of incorrectly classified examples) for both datasets?
For the SKPerceptron it doesnt misclassify any example as seen in Figure 20. The misclassification rate for our own perceptron is 0.1 for the training set and 0.25 for the testing set with $\eta = 0.1$ and $max\_it = 5$ and the start weights set to zero. If we randomly set the start weights, the MSE changes between 0 and about 0.5, depending on the start weights (with $\eta = 0.1$ and $max\_it = 5$).

How would you learn a non-linear decision boundary using a single perceptron?
As one can see for the non-linear separable figures it doesn't matter what η or the number of max iterations is, it will always fail.
To solve this problem we would have to expand the input features by adding quadratic or cubic terms of our input x_0 and x_1.