Deep Learning FSS22
Assignment 1: Neural Networks

Timur Michael Carstensen - 1722194

27.03.2022

Contents

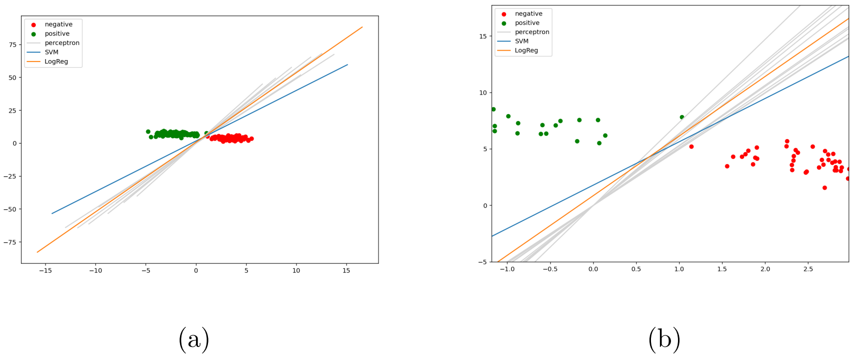
1 Perceptron Learning

1.1 a)

Cf. code for the implementation.
When the perceptron learning algorithm completes an entire training epoch (i.e. one pass through the entire training set) without any weight updates, it is safe to stop perceptron learning earlier. This is because all training samples are now being correctly classified by the perceptron. This will only work in the separable case. In the non-separable case, there will always be at least one sample that cannot be correctly classified and thus weight updates will continue until the maximum number of epochs is reached. Early stopping cannot be implemented for the pocket algorithm as we perform random sampling with replacement and hence cannot be sure if we actually went through the entire training set without misclassifying a sample or not.

1.2 b)

As would be expected, the perceptron learning algorithm perfectly classifies all samples in $\mathcal{D}_1$ and misclassifies one sample in $\mathcal{D}_2$ (cf. Figure 1, Figure 2).

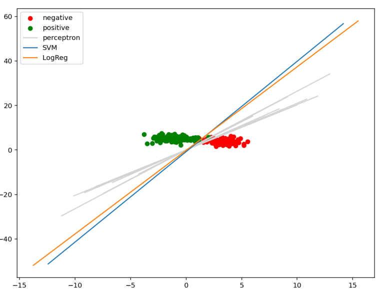


(a) (b)

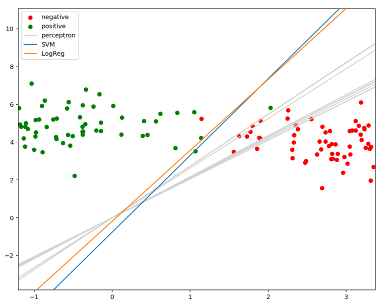
Figure 1: perceptron on $\mathcal{D}_1$ (a) without cropping in (b) with cropping in

As we learned in the Deep Learning lecture, in the separable case, the linear support vector machine (LSVM) yields us the perceptron of optimal stability. That is, the one that maximises the margin. This is coherent with Figure 1 where we can see that the LSVM decision boundary is the most "centered".

In the non-separable case (cf. Figure 2), there is a bit more heterogeneity in the results. First, one can see that the decision boundaries found by the ten (10) runs of the perceptron learning algorithm are not very similar to those found by LSVM and LR. The latter two find very similar ones. The perceptron does not converge and just keeps "jumping around", if you will. Therefore, the decision boundaries found are rather different from another.



(a)



(b)

Figure 2: perceptron on $\mathcal{D}_2$ (a) without cropping in (b) with cropping in

The perceptron performs better on our training data than LR and LSVM. The former consistently misclassifies only one (1) sample whereas LSVM and LR misclassify three (3). This can be explained by the fact that the latter two optimise for a different objective function than the perceptron. The perceptron minimises the misclassification rate, whereas LR minimises the negative log-likelihood (NLL) and LSVM minimises the SVM objective with soft margin constraints.

Due to these different objective functions, it can happen that LR finds a hyperplane that minimises NLL while also misclassifying more than the perceptron. This also applies to LSVM where the objective is not to minimise absolute misclassifications but to the find the hyperplane with the maximum margin, which in theory should generalise better than a hyperplane that has the least misclassifications.

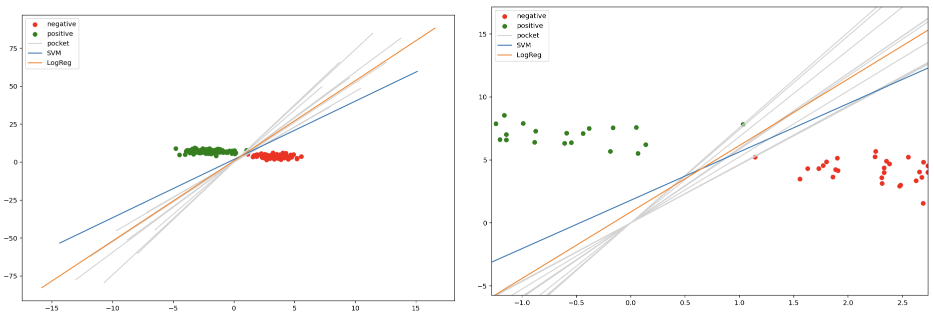
1.3 c)

Cf. code.

1.4 d)

As compared with the perceptron learning algorithm, the pocket algorithm converges. However, as the pocket algorithm is unconstrained in the sense that there are many different separating hyperplanes, the found solutions heavily depend on the random initialisations of the initial weight vector. Thus, the solutions, just like with the perceptron learning algorithm, differ quite vastly from another. Though, all of them are correct in the sense that they are separating hyperplanes for our two classes.

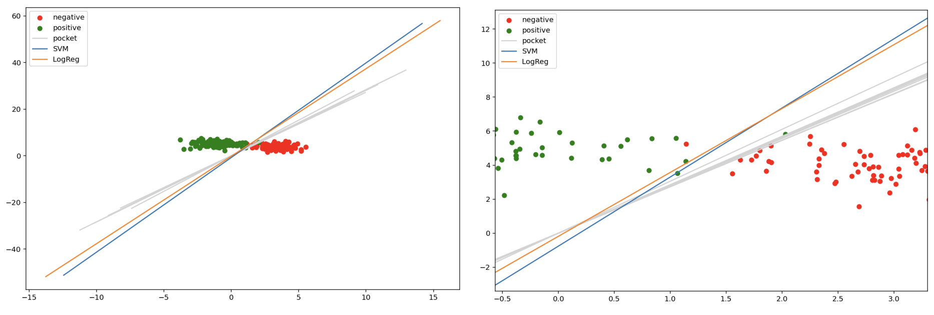
As would be expected, LR and LSVM also find a separating hyperplane (cf. Figure 3) in the separable case. Contrary to the perceptron learning algorithm, the runtime of the pocket learning algorithm is higher as one cannot implement early stopping and now must go through all epochs as alluded to in subsection 1.1.



(a) (b)

Figure 3: pocket on $\mathcal{D}_1$ (a) without cropping in (b) with cropping in

In the non separable case, the pocket learning algorithm is as good as the perceptron learning algorithm as measured by misclassified samples. That is, the pocket algorithm misclassifies one (1) sample, while LSVM and LR both misclassify three (3). However, compared with the perceptron learning algorithm, the pocket algorithm seems to find hyperplanes that are much closer to the ones found by LSVM and LR (cf. Figure 4). As the pocket algorithm only updates the final weight vector if it actually misclassifies fewer samples than the current best estimate. Also, now the algorithm is constrained in the sense that the surface in our dataspace in which it can manage to misclassify only one sample is rather small. Hence, once the algorithm finds a hyperplane in that space, it will converge there, and unlike the perceptron elarning algorithm, stop "jumping around"



(a) (b)

Figure 4: pocket on $\mathcal{D}_2$ (a) without cropping in (b) with cropping in

2 Multi-Layer Feed-Forward Neural Networks

2.1 a)

With zero (0) hidden neurons, the neural network (NN) will have a single input which directly goes to the output and hence the calculated function will look as follows:

$$\hat{y} = wx + b \tag{1}$$

That is, the input will be scaled by a single weight (w) and shifted by a bias term (b). During training, to reduce the loss, the fitted line should take the form of a constant with about 50% of the data points on each side of it.

With one (1) hidden neuron, the NN will have one sigmoid unit at its "disposal". Now, the function that the NN calculates will take the following form:

$$\hat{y} = w_{2,1} * \sigma(w_{1,1}, x + b_{1,1}) + b_{2,1} \tag{2}$$

During training the weight $w_{2,1}$ and bias $b_{2,1}$ will be used to scale and shift the sigmoid function of the hidden unit to better fit the training set while $w_{1,1}$ and $b_{1,1}$ will be used to control the "steepness" of the sigmoid (i.e. how quickly it saturates) and for which range of values of x it will saturate. The fit will most likely look like a scaled, shifted and/or mirrored sigmoid function to fit along one of the slopes of our target function (i.e. the datapoints in our training set). With only one sigmoid unit we should not be able to fit a turning point, yet.

With two (2) hidden sigmoid units the function calculated by the NN will look as follows:

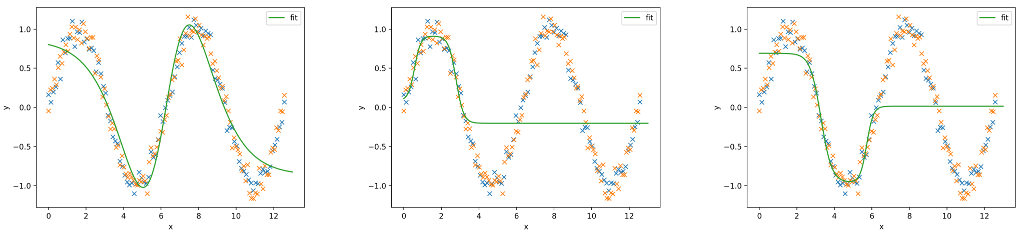
$$\hat{y} = [w_{2,1} * \sigma(w_{1,1}, x + b_{1,1})] + [w_{2,2} * \sigma(w_{1,2}, x + b_{1,2})] + b_{2,1} \tag{3}$$

Now, the NN would be able to learn some of the curvature in our dataset during training because one can now combine the two sigmoidal units. To model the turns in our dataset, the signs of the weights of the sigmoid units or the subsequent weights (the ones going to the output) must be opposite such that there are regions in our input space where their outputs cancel each other out. With two sigmoid units one would only be able to model two (2) turning points.

With three sigmoid units in our hidden layer, the function that the NN calculates would be Equation 3 with another term added for the third sigmoid. Similar to the case with two sigmoid units, we would now be able combine the outputs of three sigmoid units to model more than just two of the turns in our dataset.

2.2 b)

The results, generally, line up with my expectations. For the case with two hidden units, in the best case, as can be seen in Figure 5 (a), the FNN learns to fit two (2) of the turns in the dataset and performs quite well on both the training and test set as measured by MSE (cf. Table 1).



(a) (b) (c)

Figure 5: The FNN with two hidden (sigmoid) units. (a) shows a good fit; (b) and (c) show bad fits.

However, as can be seen in Figure 5 and Table 1 (b), (c), for multiple training runs, we achieve differing results. Here, the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm, which is the optimizer used here, seems to get stuck in local minima and converges at relatively "bad" values of our cost function as compared with (a). This is because of the random weight initialisations used when setting up the FNN. In particular, we are using Glorot initialisation (cf. [GB10]), which draws the initial weights from a normal distribution whose variance parameter depends on the number of in -and outgoing hidden units for a particular unit.

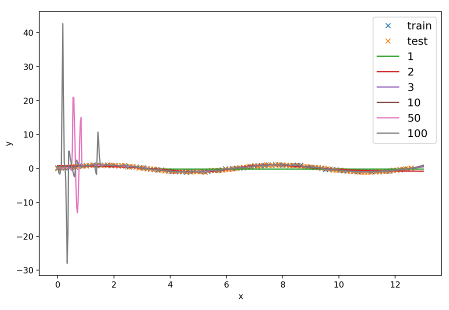
Run	**Train MSE**	**Test MSE**
(a)	0.0795	0.0867
(b)	0.3572	0.3593
(c)	0.2777	0.2863

Table 1: MSE on the training and test set for the runs depicted in Figure 5

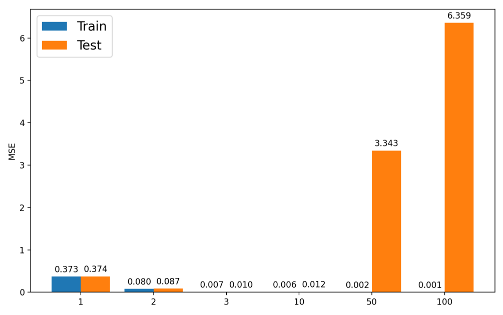
Essentially, this shows us how heavily dependent the training process is on our weight initialisation and that we must retrain/reinitialise multiple times such that we increase our chances of escaping local minima.

2.3 c)

As can be seen in Figure 6 and Figure 7, as we increase the number of hidden units (and thus our representational capacity), we first start decreasing the MSE on the training and test datasets simultaneously. A minimum for the MSE on the test set is reached at three (hidden) neurons after which it starts increasing with 10 and subsequently 50 and 100 neurons. All the while, the MSE on the training set continues to decrease. This is an indicator for overfitting. That is, if we only had the choice between three (3) and ten (10) hidden units, the former would be preferrable as it generalises better to unseen data.



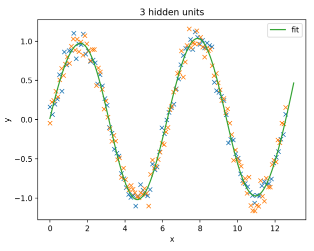
(a)



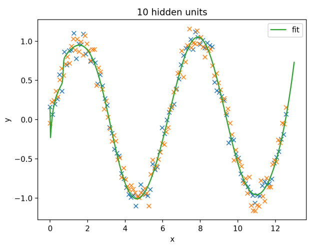
(b)

Figure 6: A FNN with 1, 2, 3, 10, 50 and 100 hidden units in its singular hidden layer. (a) shows the fit; (b) shows the MSE on the training and test sets.

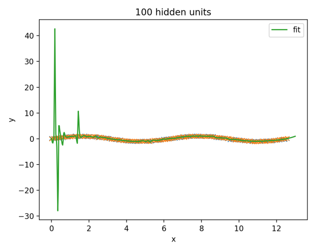
The above can also be see in Figure 7 where (a) seems to capture the underlying function that generated our data the best, while (b) already starts to fit some of the noise in our training set and (c) has so much representational capacity such that we fit every point in our training set perfectly.



(a)



(b)



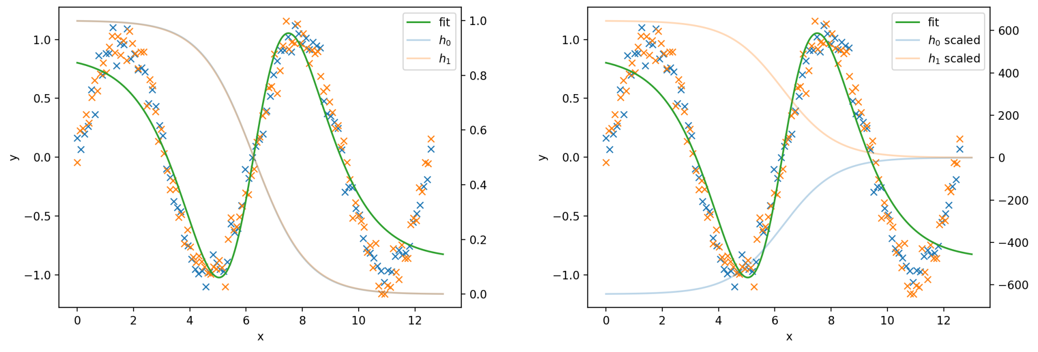
(c)

Figure 7: FNN fit with (a) 3, (b) 10 and (c) 100 hidden units in its singular hidden layer.

The above is somewhat to be expected. As per the universal approximation theorem (cf. [Cyb89]), an artificial neural network (ANN) can learn any continuous function (in the D-dimensional unicube), given enough hidden neurons. This is essentially what happens in our setup. As we increase the number of hidden units, our representational capacity increases. Thus, with increasing the number of hidden units, each sigmoidal unit now "learns" a smaller and smaller subset of our training data and fits to it. In the end, we have so many units that we learn all the datapoints in our training set.

2.4 d)

As can be seen in Figure 8 and as previously described in subsection 2.1, in the case with two hidden (sigmoid) units, the model obtains its flexibility by learning two sigmoid units and combining them. The weights and biases of the units themselves control "steepness" and left- and righward shift. The weights connecting their ouputs to the output of the FNN, scale the activations.



(a) (b)

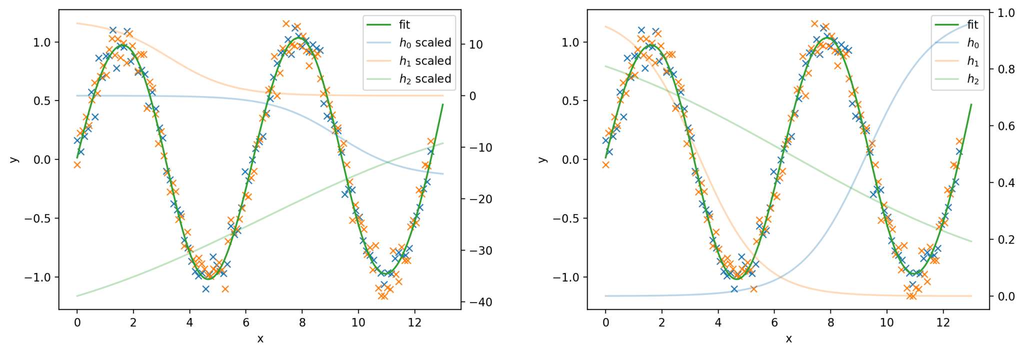
Figure 8: FNN fit with two hidden (sigmoid) units; (a) shows the hidden units activation and (b) shows the activations scaled by the output weights.

Table 2 exemplarily displays the weights of the model in Figure 8. Coherent with the figure, the model learns two nearly identitcal sigmoids and then scales them by the same magnitude, just with opposing signs. The combination of these result in the fit displayed above. One could just plug the values from Table 2 into Equation 3 and receive the same fit as in Figure 8 (ignoring the output bias).

Unit	**Unit weight**	**Unit bias**	**Output weight**
h_0	-1.0684	6.7106	-644.4341
h_1	-1.0569	6.6383	646.1637

Table 2: Weights of the FNN depicted in Figure 8 (output bias omitted).

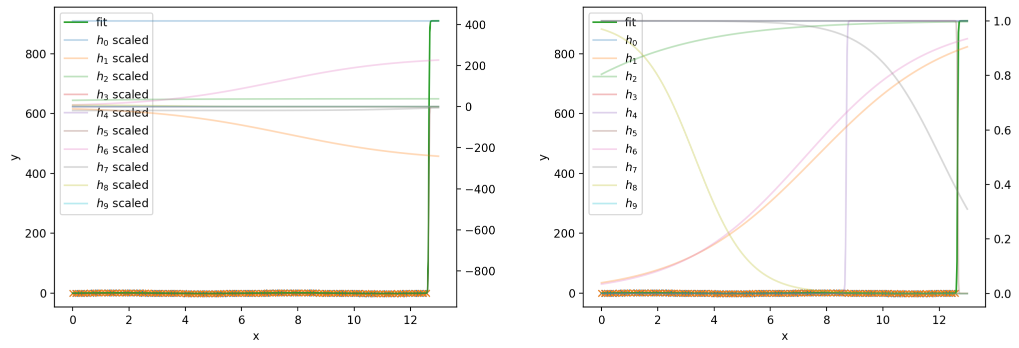
For the two (2) hidden unit case, this distributed representation is rather intuitive as one can easily think about how one unit starts saturating while the other stops doing so and they cancel out at some point to model a turning point in the underlying function, et cetera. However, this is not a given as we increase our representational capacity (read: number of hidden units). For the three (3) hidden unit case (cf. Figure 9), one can still somewhat make sense of the activations of the individual units and how they correspond to the fit of the FNN.



(a) (b)

Figure 9: FNN fit with three hidden (sigmoid) units; (a) shows the hidden units activation and (b) shows the activations scaled by the output weights.

This is very much not the case anymore with ten (10) hidden units as depicted in Figure 10. Thus, we can say that in less complex cases, the distributed representation, where each unit learns a part (or feature, if you will) of our target function, is intuitive, though generally it is not the case.



(a) (b)

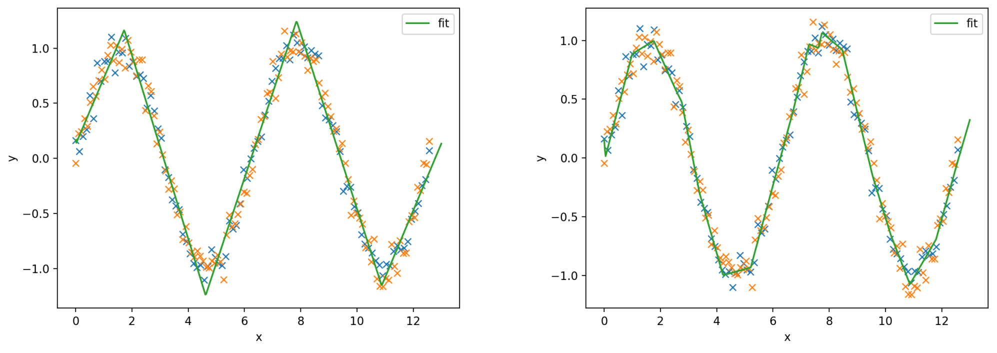
Figure 10: FNN fit with ten hidden (sigmoid) units; (a) shows the hidden units activation and (b) shows the activations scaled by the output weights.

2.5 e)

For the optimiser part, I tried Adam, LBFGS, RMSprop, Adagrad, Adadelta and plain SGD. To get a good comparison with the FNN trained with BFGS, I used only one hidden layer with three (3) sigmoid units. As seen in subsection 2.1, a FNN with three hidden sigmoid units should be able to fit our training set quite well. As expected, Adam and LBFGS were able to find similarily good fits those shown in Figure 7 (a). From the other optimisers that I tried, only RMSprop found a similarily good fit (note: the first and last turning points were not captured as well as before). I also noticed that RMSprop took much longer to initialise at first.

The other optimisers got stuck in local minima and converged at very high values of our cost function. These results are quite in line with what I would have expected, especially with regard to Adam, as I have seen it being used in real world applications predominantely.

In regards to the hidden units, I stuck with rectified linear units (ReLU). Here, I tried used the same setup as before and varied the number of hidden units from 1 to 100 to get a good comparison to the sigmoid units. It seems that we must use far more ReLUs to get a similar fit as the one achieved with just three (3) sigmoid units. In fact, only the fit with ten (10) ReLUs is somewhat comparable. Fifty (50) units already start to overfit our training set (cf. Figure 11).



(a) (b)

Figure 11: FNN fit with rectified linear units; (a) 10 units, (b) 50 units.

Another noteworthy observation is that the FNNs with ReLUs trained much faster than their counterparts with sigmoid units. This is because the logistic sigmoid function includes an exponent in its denominator which is computationally expensive. Thus, we can say that the trade-off between sigmoid units and ReLUs is between goodness/smoothness of fit and training time/speed.

Finally, varying the depth of the FNN shows that we can achieve the same fit (i.e. have the same representational capacity) by making our model deep instead of wide. This, is in line with the findings by [HS17].

References

[Cyb89] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[HS17] Boris Hanin and Mark Sellke. Approximating continuous functions by relu nets of minimal width. *arXiv preprint arXiv:1710.11278*, 2017.