Hot Topics in Machine Learning (HWS17) Assignment 4: Neural Networks

Steffen Schmitz
University of Mannheim
stefschm@mail.uni-mannheim.de

1. PERCEPTRON LEARNING

Task. We look at binary classification using a perceptron. We use two 2D datasets: a separable dataset $\mathcal{D}_1(X1$ and $y1)$ and a non-separable dataset $\mathcal{D}_2(X2$ and $y2)$. Both datasets include an additional bias feature.

1.a Rosenblatt Learning Algorithm

Task. Complete function pt_train to train a perceptron using Rosenblatt's perceptron learning algorithm (without an explicit bias term).

The Rosenblatt algorithm assumes uses a binary label vector with -1 and 1 as possible classes. We reformat our input labels to $\hat{\mathbf{y}}$ with

$$\hat{y}_i = \begin{cases} -1 & \text{if } y_i = 0 \\ 1 & \text{if } y_1 = 1 \end{cases}$$

We use the signum function[1] to bring the product of our weights with the inputs into the same shape as the modified label vector $\hat{\mathbf{y}}$. The perceptron with the current weight vector $\mathbf{w}$ makes an error, whenever $\text{sgn}(\langle \mathbf{w}, \mathbf{x}_i \rangle) \neq \hat{y}_i$.

Rosenblatt's algorithm starts with a zero or random weight vector and updates it as long as it makes errors on the dataset. If the dataset is not separable by a hyperplane the algorithm runs infinitely. Due to the infinite runtime on inseparable data we add a number of maxepochs to our function, but we may end our computation early, once the error is zero.

For every point, where the prediction is wrong it updates the weight vector by adding or subtracting the current point from the weight vector, i.e. we add the current data point $\mathbf{x}_i$ to $\mathbf{w}$, if we falsely predicted a label of -1 and subtract the data point from $\mathbf{w}$ if we falsely predicted 1. The implementation is shown in Figure 1.

```
def pt_train(X, y, maxepochs=100, w0=None):
    N, D = X.shape
    w = w0 if w0 is None else np.zeros(D)

    yhat = list(map(
        lambda i: i if i == 1 else -1, y))
    ypred = np.sign(X @ w)

    if np.array_equal(ypred, yhat) or \
        maxepochs == 0:
        return w

    for i in range(N):
        if yhat[i] != ypred[i]:
            w += yhat[i] * X[i]

    return pt_train(X, y, maxepochs - 1, w)
```

Figure 1: Rosenblatt Perceptron Learning.

1.b Rosenblatt - Experimentation

Task. Test your function on $\mathcal{D}_1$ and $\mathcal{D}_2$. You can plot the decision boundaries and print the misclassification rates of multiple runs of your training algorithm as well as a linear SVM and a logistic regression classifier. Discuss and explain the results.

First, we run Rosenblatt's perceptron learning algorithm on the linearly separable dataset $\mathcal{D}_1$. We expect to get a line that perfectly separates the two clusters in the dataset. The decision boundary is plotted in Figure 2. Due to the in-

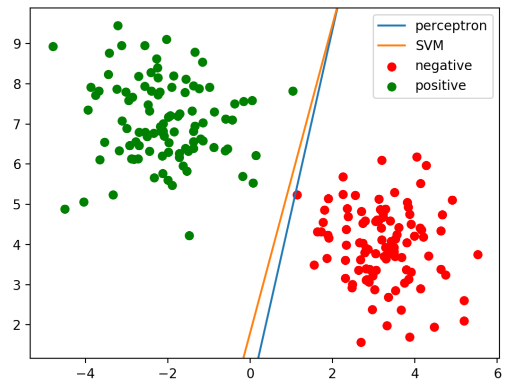

Figure 2: Rosenblatt learning on $\mathcal{D}_1$.

cremental updates in the order of data points in $\mathbf{X}$ in the pt_train method this algorithm is deterministic and we will receive the exact same result for multiple, consecutive runs.

If we repeat this process on the non-separable dataset $\mathcal{D}_2$ we would expect the training function to run until maxepochs is reached, because it will never find a perfect fit that would

[1] https://en.wikipedia.org/wiki/Sign_function

allow the function to abort early. The result will still be deterministic and should fit the dataset well. It is shown in Figure 3.

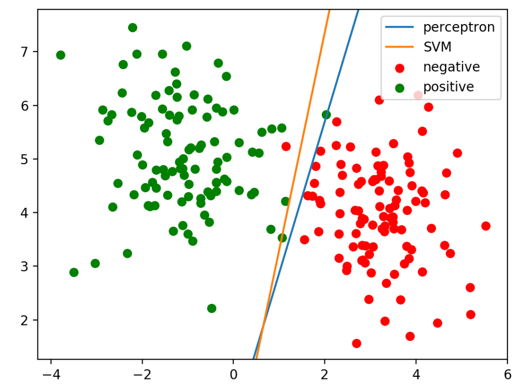

Figure 3: Rosenblatt learning on $\mathcal{D}_2$ after 100 epochs.

To compare the result with the "perfect" linear fit we added the decision boundary of a Support Vector Machine to the graphs that maximizes the distance between data points of the opposing clusters.

1.c Pocket Learning Algorithm

Task. Extend your function such that it can also train a perceptron using the pocket algorithm. In each epoch, your function should process N random examples sampled with replacement.

One of the major issues with Rosenblatt's perceptron learning algorithm that we presented in Section 1.a is the infinite runtime for non-separable datasets and the sudden stop, once we reached the maxepochs threshold. The pocket algorithm tries to improve this short comings by using random examples and keeping the best weight vector $\mathbf{w}$ that it found so far in its "pocket" [3, p.91f.].

In each epoch we pick n random samples and update the weight vector, when a sample was misclassified, and track the number of subsequent samples that are correctly classified. If this number is bigger than the one for our weight vector in the pocket, the current vector replaces it. In contrast to Rosenblatt's learning algorithm the resulting hyperplane is not the same for subsequent runs. It is possible to show that the pocket algorithms converges to the optimal solution after a finite number of runs [2].

1.d Pocket - Experimentation

Task. Test your implementation of the pocket algorithm on $\mathcal{D}_1$ and $\mathcal{D}_2$. Are the results different than before? Discuss and explain.

On the separable dataset $\mathcal{D}_1$ we would expect that the pocket algorithm finds an optimal solution and aborts the execution early. Due to the randomness in the algorithm there is a small probability that it does not find a perfect solution. If we run the pocket training algorithm multiple times on the first dataset we see that decision boundary changes a bit on subsequent runs, but always provides a perfect classification.

Using the non-separable dataset $\mathcal{D}_2$ we can see that the pocket algorithm uses as many epochs as specified by the maxepochs parameter. It is impossible to abort the run early, because there is no perfect classification. Nevertheless, the pocket algorithm usually finds a very good approximation and has a small misclassification rate.

Now, we will compare multiple runs of the pocket algorithm with the results of Logistic Regression and Support Vector Machines. Surprisingly, the best perceptron result misclassifies only one sample in the training set, while Logistic Regression and Support Vector Machines misclassify 3 samples each. The resulting decision boundaries are shown in Figure 4. This is a typical property of an overfitted model. In this

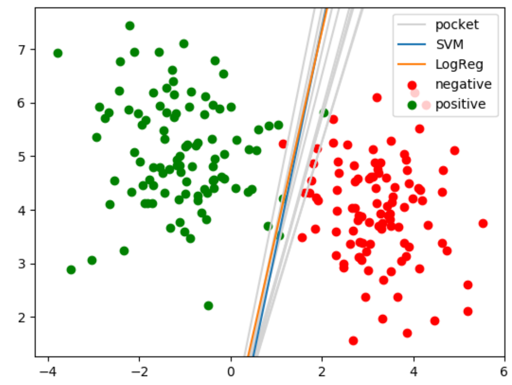

Figure 4: Comparing different classification algorithms on $\mathcal{D}_2$.

case we would expect that the perceptron fitted model does not generalize well on unseen data and performs worse on the test dataset, compared to the Logistic Regression and Support Vector Machines models.

All in all, we can see that the perceptron learning algorithms work well on separable and non-separable data, but are computationally expensive and have to be aborted, if the dataset is non-separable. For separable data they provide (almost always) a perfect classification, but return a random decision boundary that classifies the datasets correctly, while Support Vector Machines maximize the distance between the points of two datasets and, therefore, work better on previously unseen samples.

2. MULTI-LAYER FEED-FORWARD NEURAL NETWORKS

Task. In this task, we look at regression using multi-layer feedforward networks (FNN) where we vary the number of units in the hidden layer. The shape is displayed in Figure 5. We use a 1-dimensional dataset $\mathcal{D}_3$.

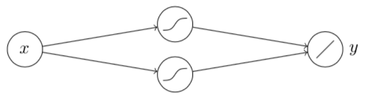

Figure 5: Neural Network Architecture.

2.a Conjecture how a fit will look like

Task. Look at the training data and conjecture how a fit for an FNN with zero, one, two, and three hidden neurons would look like.

The Neural Network that we use in this section has a single numeric input, one hidden layer with n hidden logistic neurons and a single linear output neuron. In this section we try to predict the shape of the resulting fit for $n = 0$, $n = 1$, $n = 2$ and $n = 3$ hidden neurons.

If $n = 0$, we directly connect our input to the linear output neuron and obtain a linear fit. We would expect a constant prediction of $y \approx 0$ for every possible input x, because the input dataset looks like a sine function and with equal positive and negative amplitudes the mean is close to zero.

With one hidden unit, we transform the input into a real valued number in the range $h_1 \in [0, 1]$. If we use the output of the hidden layer as the input for our output layer we expect a S-shaped curve that we associate with logistic units, scaled by the linear neuron from the output. Obviously, it is also possible to have a reversed S-shape by switching the sign. We expect the S-shape to follow the shape of one of the pikes in the sine-like function and take a continuous value afterwards.

For $n = 2$, we have two logistic hidden units and a single linear output neuron. This architecture is represented in Figure 5. In this case we get two real-valued outputs in the range from 0 to 1 from the hidden layer. Those two functions can be combined in multiple ways to fit the dataset $\mathcal{D}_3$ more or less well. It should be possible to follow the shape of the function well. We assume that it resembles the two spikes in the middle and diverges close to the edges. We will look at this architecture more closely in Section 2.b.

Adding another hidden unit, we would expect that the fit is even better than with two possible units. In the given range of data points it should fit the dataset well and closely resemble it. This follows from the approximation theorem by Cybenko [1]. Although it imposes the condition that this only holds in the unit-cube, we can rescale our input dataset to use x-values in the range from $[0, 1]$. This means that we expect an even better fit for an increasing number of neurons. We will explore this behaviour in Section 2.d.

To conclude, we would expect the error of our fit to decrease with an increasing number of hidden neurons, as the Neural Network approximates the true fit of our dataset better.

2.b Train with 2 hidden units

Task. Train an FNN with two hidden neurons, determine the mean squared error (MSE) on the training and the test data, and plot. Is the result as you expected? Now repeat training multiple times. What happens? Explain.

Training and plotting the Neural Network multiple times results in three major fits that appear regularly. The Neural Network either fits the first positive or the first negative spike in the dataset $\mathcal{D}_3$ well or it fits the both spikes in the middle well and only approximates the edges. This is shown in Figure 6, 7 and 8, respectively.

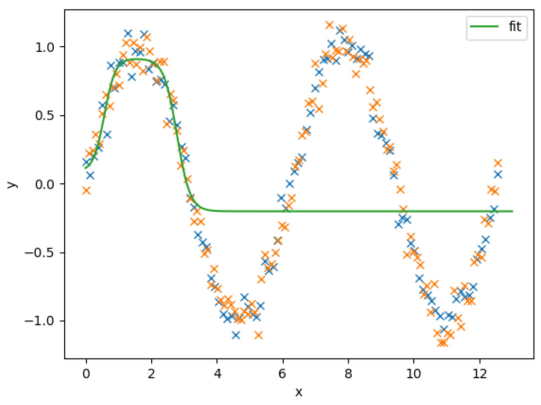

Figure 6: Fit for first spike.

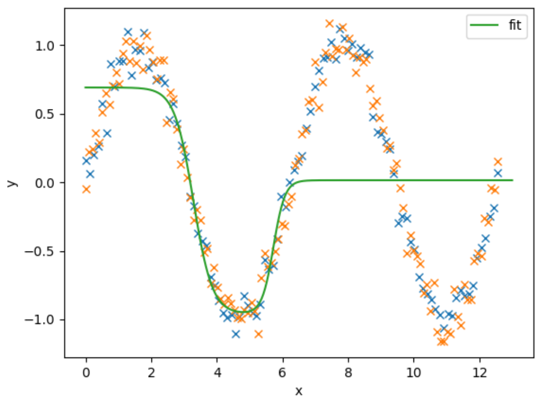

Figure 7: Fit for second spike.

Figure 8 corresponds to our assumption in Section 2.a and has the smallest error on the training and on the test set. It has an error of about 0.08 for both sets, while the other two models have an error of about 0.3.

The first and second results are not as expected, because it seems that they prefer to fit one spike really well, instead of minimizing the error on the whole dataset. They combine the two S-shaped functions they receive from the hidden

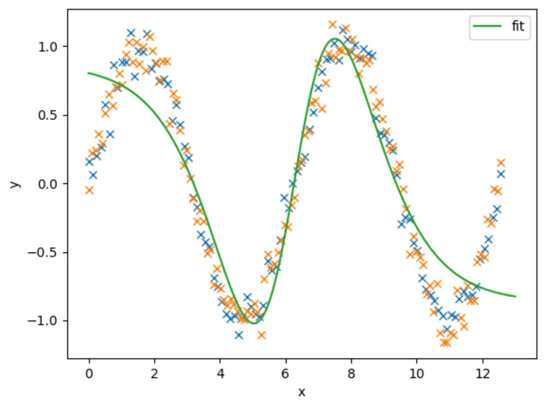

Figure 8: Fit for middle spikes.

layer without much modification through the linear neuron. The constant tail is similar to the single-neuron architecture.

This shape could appear if the optimizer function becomes stuck in a local optimum. We would assume that the error function for this Neural Network architecture is not convex. One solution to avoid this is to initialize the weight vectors randomly, train the network multiple times and use the result that minimizes our error function.

2.d Width Experimentation

Task. Train a FNN with 1, 2, 3, 10, 50, and 100 hidden neurons. In each case, determine the MSE on the training and the test dataset. Then plot the dataset as well as the predictions of each FNN on the test set into a single plot. What happens when the number of hidden neurons increases? Is this what you expected? Discuss!

Figure 9 shows the resulting fits for different widths of the hidden layer in our Neural Network. It is possible to clearly distinguish the plot for 1 and 2 hidden units, while the other fits blend into each other and model our input dataset nearly perfectly.

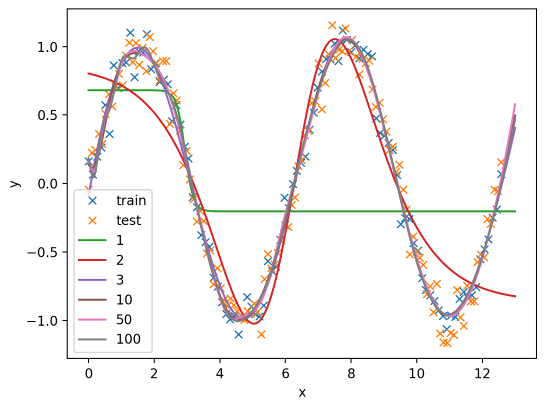

Figure 9: Fit for different widths of hidden layer.

The corresponding error for the numbers of hidden units is shown in Figure 10. We can see that the error on the training and on the test data is similar for all numbers of hidden units and that the error decreases rapidly for 1, 2 and 3 hidden units, while staying almost constant in the range of 3 up to 100 hidden units.

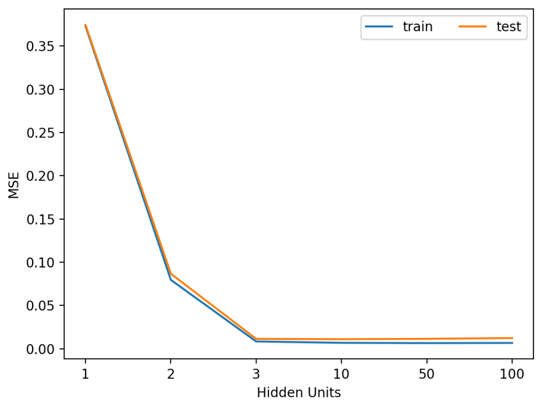

Figure 10: MSE for different widths of hidden layer.

This plot shows that there is little gain in training more than three hidden neurons. We call those Hockey Stick graphs and the corner in the stick usually marks a good choice for a parameter - the number of hidden units in our case.

This is similar to the expectation that an increasing number of hidden neurons fits the dataset even better, but that our approximation is good enough at some earlier point. We can also see that the error on our test set slightly increases for a bigger number of hidden neurons, which might be an indication that we overfit the dataset.

As a conclusion we can say that 3 hidden neurons fit our dataset well and are sufficient to make good predictions.

2.e Distributed Representations

Task. Train a FNN with 2 hidden neurons and visualize the output of the hidden neurons (= distributed representation). Then visualize the output of the hidden neurons scaled by the weight of their respective connection to the output. Now repeat with 3 hidden neurons, then 10 hidden neurons. Try to explain how the FNN obtains its flexibility. Is the distributed representation intuitive?

The distributed representation corresponds to the output of our hidden layer. It is composed of n S-shaped curves that get an associated weight to compose the final, linear representation. While the distributed representation is intuitive for 2 and 3 hidden neurons it is obfuscated for 10 neurons and, therefore, not intuitive or easy to grasp.

Figure 11 and Figure 12 show the distributed and the scaled distributed representation for two hidden neurons.

The flexibility in Neural Networks stems from the possibility to learn additional features and not only correct weights. If we interpret the hidden layer as a new set of features we can expand the realm of possible functions that we can model. With additional hidden layers we can add even more features and also features of features. One of the advantages of Neural Networks is the possibility to input raw data with little or no preprocessing and let the network do the feature engineering that may be necessary for classical machine learning methods.

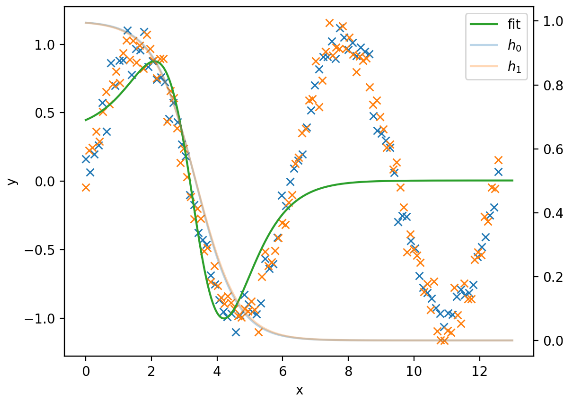

Figure 11: Distributed representation for 2 hidden neurons.

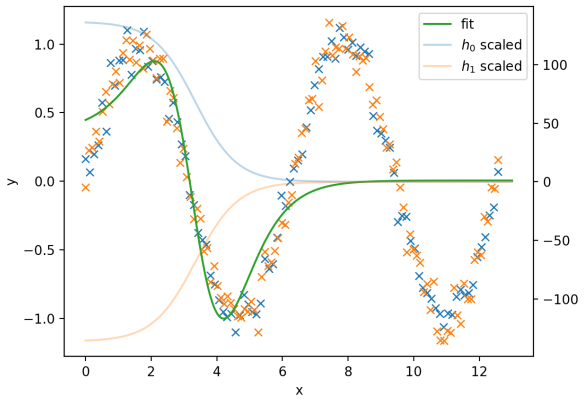

Figure 12: Scaled distributed representation for 2 hidden neurons.

3. REFERENCES

[1] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec 1989.

[2] S. I. Gallant. Perceptron-based learning algorithms. *IEEE Transactions on Neural Networks*, 1(2):179–191, Jun 1990.

[3] R. Rojas. *Neural Networks: A Systematic Introduction.* Springer-Verlag New York, Inc., New York, NY, USA, 1996.