DD2437 Lab Assignment 1
Learning and generalization in feed-forward networks from perceptron learning to backprop

Thomas Gaddy, Daniel Persson Pross and Matthaios Stylianidis

I. INTRODUCTION

The first part of this laboratory assignment involved implementing single- and multi-layer perceptrons with focus on the associated learning algorithms. Specifically, we trained our networks using the *Delta rule* and *Perceptron learning rule* for a single-layer perceptron and the *generalized Delta rule*—also known as the error backpropagation algorithm—for a two-layer perceptron. These implementations were done from scratch using Python and the numpy library for efficient matrix operations. In the second part of assignment we used *sklearn's* multi-layer perceptron for chaotic time-series prediction.

II. CLASSIFICATION WITH A SINGLE-LAYER PERCEPTRON

A. Generation of linearly-separable data

The first task involved generating data for binary-classification. The two classes were drawn from multivariate normal distributions with different means and the same covariance. The generated data is presented in figure 1.

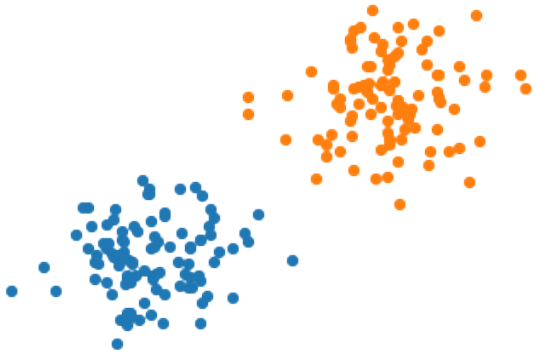


Fig. 1: Linearly Separable Classes

B. Classification with a single-layer perceptron

Here we compare both perceptron learning and the delta rule on the generated dataset. We also compared both sequential and batch learning approaches. In sequential learning the data is fed sample-by-sample to the learning algorithm while for batch learning the data is fed all at once. Although perceptron learning is often done sequentially and the delta rule is performed with a batch approach, we implemented all combinations.

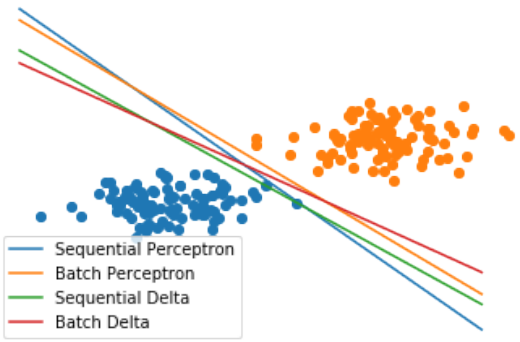


Fig. 2: Decision Boundaries

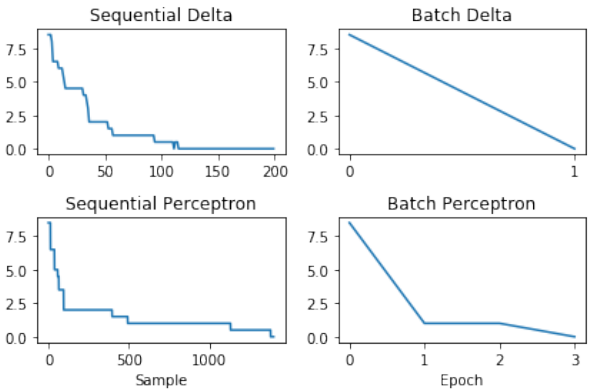


Fig. 3: Learning Rates by misclassification percentage

As we can see in figure 2, all learning approaches were successful in that the decision boundaries correctly separate the two classes. That being said, there is very poor generalization from this approach. Intuitively, we would want the decision boundary to lie "between" the two classes and instead we see that the decision boundaries sometimes run very close to the data points.

Figure 3 shows the learning rates for all four approaches. The weights were initialized the same way for each approach for a more fair comparison. As we can see, the perceptron rule seems to be the slowest, with the sequential approach being slower than the batch approach. Overall, the delta rule with a batch approach seems to learn the fastest.

C. Classification of non-linearly separable classes

Here the class means of the generated data are much closer together, leading to overlapping classes. Although normally we would have a suitable stopping condition based on if the error stopped decreasing or if the weights stopped changing, here we let the learning algorithms run for 30 epochs in order to visualize non-convergent learning in both cases. We see in figure 4 that both learning algorithms can eventually converge to optimal or near optimal solutions. When we visualize the learning rates, however, we see highly erratic behavior for the perceptron learning rule. We see in figure 5 that all approaches reach minimal error after only a few iterations. While the delta rule maintains a relatively consistent error, the perceptron rule results in unpredictable behavior if we let the algorithm continue to run.

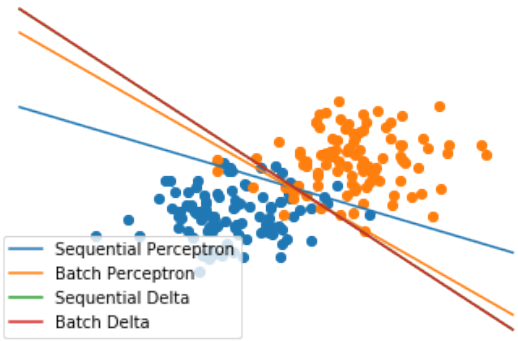


Fig. 4: Decision Boundaries for Non-Linearly Separable Classes

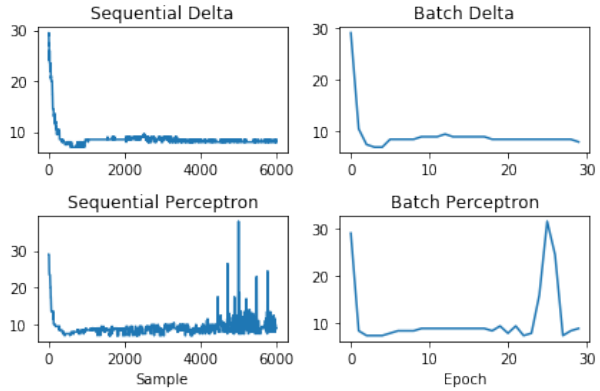


Fig. 5: Learning rates for non-linearly separable classes

III. Classification and Regression with a Two-Layer Perceptron

A. Classification of linearly non-separable data

This task involves using a 2-layered perceptron in order to classify non-separable data. To study the performance of the perceptron we used different numbers of neurons for the hidden layer. To perfectly separate the available data we need at least as many hidden neurons as the hyperplanes that are needed to separate that data.

Figures 6 and 7 show how our neural network classified the training and the test data for one of the non-linearly separable datasets we generated. The decision boundaries in the plots are derived by the weights of the hidden layer where we have used only 2 neurons for this dataset.

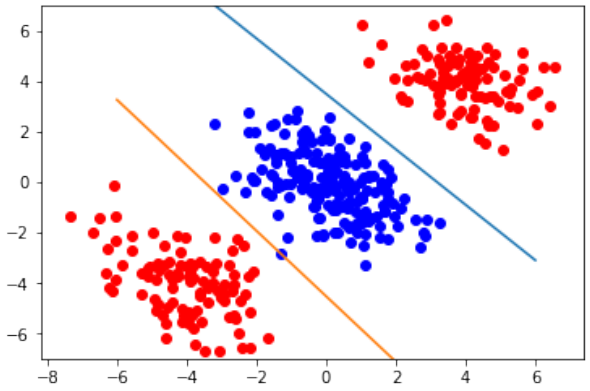


Fig. 6: Two-layered perceptron classification of non-separable data (training data).

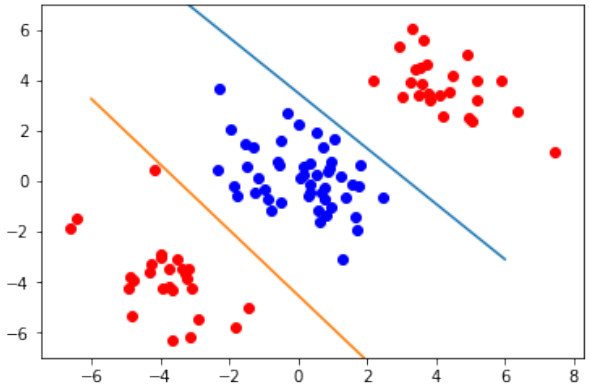


Fig. 7: Two-layered perceptron classification of non-separable data (test data).

Training our classifier with less hidden layer neurons (i.e. 1 neuron) resulted in underfitting as our model would never be able to separate the two classes with a single hyperplane. Moreover, we observed that by adding more hidden layer neurons we needed more epochs for the neural network to converge and achieve the same training error as the classifier with 2 hidden layer neurons. For example, the classifier that used only two neurons needed about 200 epochs to converge while one with 50 neurons needed about 280 epochs using 400 training samples.

We also observed that the training and test learning curves were similar. This is due to the fact that the training data involve no noise and we used enough training data to capture the difference between the classes. Thereby, by decreasing the training error we also decrease the test error at the same

time. However, in the case were fewer data points were involved the error rate of the test dataset was higher as it is shown in Figure 8.

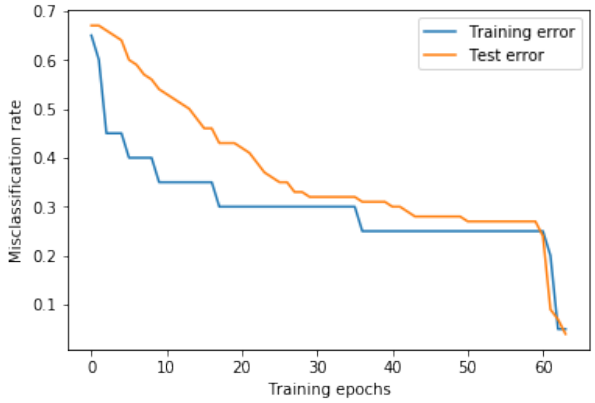


Fig. 8: Training and test learning curves for 20 training data points.

Finally, we implemented the sequential version of back propagation and we observed that the batch learning converges faster and that training the network multiple times using sequential learning leads to quite different results. This can be explained by the fact that sequential learning might avoid certain minima which can cause the training to sometimes find a better or worse solution.

B. The Encoder Problem

In this section we will describe how our two-layered perceptron performed in the task of encoding patterns. Based on the assignment description for this task we have 8 distinct patterns. Thereby, the optimal number of hidden neurons is 3 since each neuron output be discritized into two values based on its sign, making the number of possible combinations of the output values equal to 8.

Our implementation of the neural network always converged, being able to reduce the training error to zero. That can be explained by the fact that we had enough hidden layer neurons to encode the pattern and also because we had all the possible instances of our dataset with no noise.

In the case of 8 distinct patterns, choosing less than 3 hidden layer neurons would make it impossible to encode the patterns while using more would still encode the patterns but at the cost of an unnecessarily more complex neural network which results in a worse compression rate.

Finally, Table I depicts the encoded patterns with each hidden layer output rounded to 0 or 1. We observe that the first layer of weights practically encodes the patterns into a binary representation.

Input pattern	Encoded representation
1	1 0 0
2	1 1 1
3	0 0 1
4	1 1 0
5	0 0 0
6	0 1 0
7	0 1 1
8	1 0 1

TABLE I: Encoded patterns using an auto-associative MLP

C. Function Approximation

To study the regression performance of a two-layer perceptron we used the bell shaped Gauss function, pictured in figure 9. To test the generalization performance, the neural network was trained on a small subset of the data used to plot figure 9 for 200 epochs and then used to predict the function values for x and y in the span $[-5, 5]$. We looked at the performance of the network configured with 5, 10 and 25 hidden nodes when trained on the same 10, 25 and 100 data points.

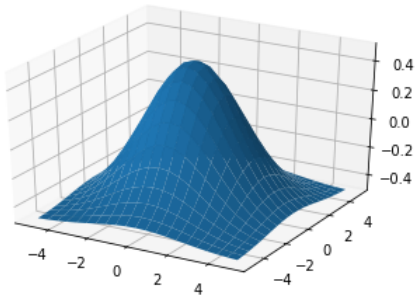
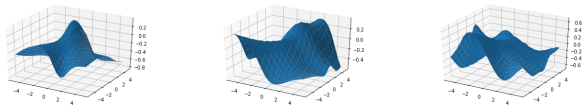


Fig. 9: Bell shaped Gauss function.

As can be seen in figures 10 and 11, the generalization using 10 data points is quite poor. None of the configurations of the network generalize well, which would suggest that we are using too few data points to train the network. The configuration with 5 hidden nodes is the one with the smallest MSE on the validation set. This could mean that the other configurations overfit the given datapoints.



(a) 5 hidden (b) 10 hidden (c) 25 hidden

Fig. 10: Two layer network trained with 10 data points.

The results using 25 training data points are shown in figures 12 and 13. Here, the configurations with 10 and 25 hidden nodes have similar performance on the validation set, while with 5 nodes it performs worse. This is evident in figure 12, where the plot for 5 hidden nodes shows a simpler curve than the other two. This would suggest that the configuration with 5 hidden nodes slightly underfits the data. When comparing 10 and 25 nodes, they perform similarly well on the validation set. The configuration using 25 hidden nodes takes more than 100 epochs to converge,

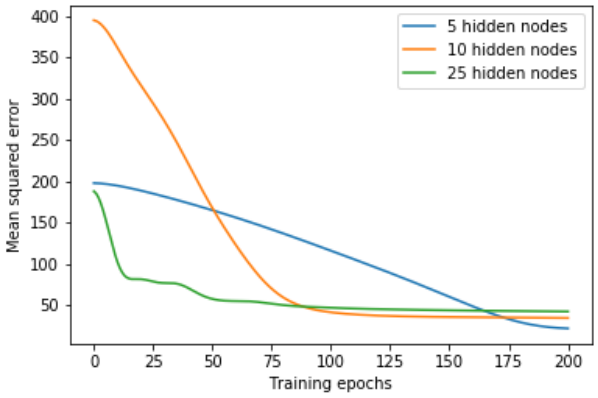
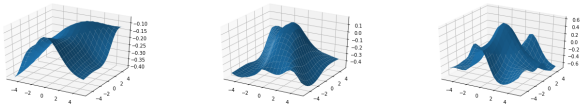


Fig. 11: Validation MSE for 10 training points.

which is twice as many as for 10 nodes. Some parts of the approximations for 10 and 25 nodes, especially parts close to the fringe of the approximation span, do deviate from the true function. This is likely due to there still being too few training data points to correctly approximate the entire chosen span of the function.



(a) 5 hidden (b) 10 hidden (c) 25 hidden

Fig. 12: Two layer network trained with 25 data points.

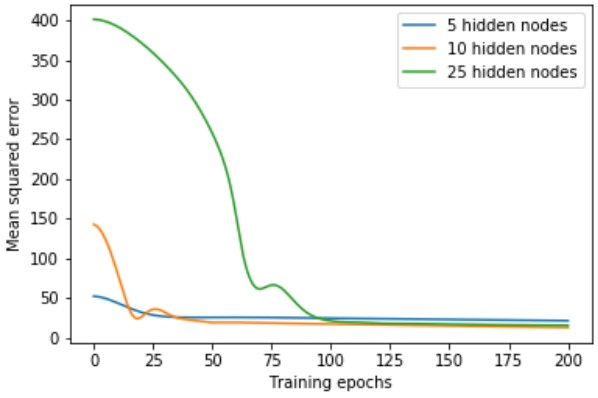
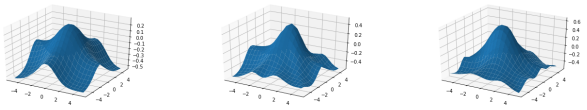


Fig. 13: Validation MSE for 25 training points.

The results using 100 training data points are shown in figures 14 and 15. For this training data set, all chosen configurations of the two-layer network have good generalization results and are very close in terms of MSE on the validation set. The configuration with 5 hidden nodes is similar in shape to the configurations using 10 and 25 hidden nodes with 25 training data points. The configurations using 10 and 25 hidden nodes do a very good approximation of the true function, with very low error rate on the validation set. The configuration using 25 hidden nodes takes the longest to converge, but is the one with the best generalization.



(a) 5 hidden (b) 10 hidden (c) 25 hidden

Fig. 14: Two layer network trained with 100 data points.

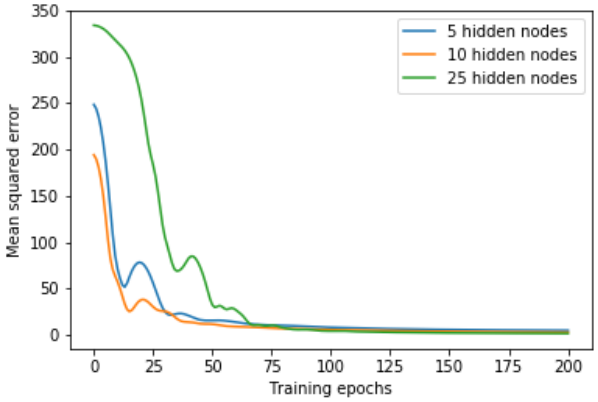


Fig. 15: Validation MSE for 100 training points.

IV. Time Series Prediction

A. Data

In order to study the use of multi-layer perceptrons for chaotic time-series prediction we generated a Mackey-Glass time series. The generated data were split into training, validation, and test sets as is shown in figure 6. The first 800 points were used for training while the rest of the data was evenly split into validation and testing sets.

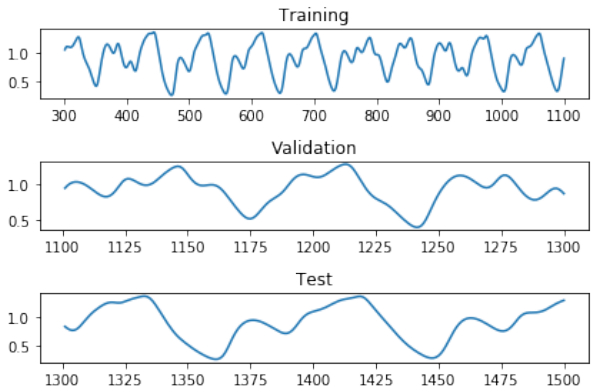


Fig. 16: Mackey-Glass Time Series Data

B. Two-layer Perceptron for Time Series Prediction

In this part of the assignment we developed a multi-layer perceptron using *sklearn*. We explored having up to 8 nodes in the hidden layer. Increasing the number of hidden nodes beyond 3 did not result in significantly better performance, as can be seen in table 1.

Hidden neurons	Training error	Validation error
1	0.0835	0.0527
2	0.0152	0.0038
3	0.0008	0.0006
4	0.0090	0.0078
5	0.0090	0.0077
6	0.0006	0.0006
7	0.0090	0.0077
8	0.0007	0.0006

TABLE II: Average error (MSE) over 20 training runs.

Sklearn also has a built-in parameter alpha which is a L2 penalty term. For different numbers of hidden nodes we explored a wide range of alphas over more than 10 orders of magnitude. Although the validation performance did not change dramatically, there was some improvement when using a small regularization parameter. It should be noted, however, that the data used for training had no noise. As a result, it is almost impossible to "overfit" the training data since we will be able to capture the underlying function. Nevertheless, we found that an alpha value of around 0.1 gave good results on the validation set. Using an alpha of 0.1 and 8 hidden nodes, we performed a final evaluation on the test set. As can be seen in the figure below, the generalization was very good, with the predicted time series following the true series almost exactly. Of course, this high level of performance could only be achieved since none of the data had added noise.

Fig. 17: Test Set Performance with 8 hidden neurons and regularization parameter of 0.1

C. Three Layer Perceptron for Noisy Time Series Prediction

In order to explore more realistic examples of generalization, we added zero-mean Gaussian noise to the data. We suspected that our models would now be more susceptible to overfitting since we could be following the noise in the data rather than the true underlying function. Indeed, though the exact same model as in the previous section could still follow the training data very well, the test performance was noticeably worse.

We explored different three-layer perceptron models. In general, the validation prediction performance increased when adding more than a couple hidden neurons to the second layer. This increase in performance did not continue when adding more neurons to the second layer; anywhere from 3-8 neurons in the second layer gave about the same performance. The increase in performance when going from 1 to 3 neurons in the hidden layer was also much more noticeable when there was less noise added to the data. This make sense since when we have less noise, the benefits of having a complex model are more pronounced. This is due to the fact that a more complex model can do a better job of capturing the nature of the underlying function.

Having explored different network configurations, we also studied the effect of regularization. With a small amount of noise in the data ($\sigma = 0.03$) the optimal value for the regularization parameter was again about 0.1 to 1. As we added more noise to the data we achieved better results with more aggressive regularization. For $\sigma = 0.18$ alpha values of about 10 gave us the best results. This makes intuitive sense as we would expect learning from noisy data to result in large weight values highly tuned to the noise. Aggressive regularization can therefore be helpful to prevent over-fitting in these cases.

We compared the best three layer model for each amount of noise with the best two-layer model from the previous task. At low levels of noise, neither model clearly dominated over the other. At high noise, the three layer perceptron performed slightly better. Additionally, adding more noise worsened the generalization performance overall regardless of which model was used. This was somewhat noticeable with the three levels of noise we used but became especially noticeable when the noise standard deviation was 0.3 or greater.

σ	Two Layer	Three Layer
0.03	0.1244	0.1244
0.09	0.1333	0.1373
0.18	0.1550	0.1803

TABLE III: Average Error (MSE) for the best two and three-layer perceptrons. The number of hidden nodes for the three layer perceptrons were (8,6), (8,7), and (8,5) with alpha values of 1, 0.1, and 10, respectively

We also visualized the training and validation errors for the best models with $\sigma = 0.18$. There are some interesting things to not with this graph. First, the training error is much lower than the validation errors. We also note that the three-layer validation performance was much better than than the two layer performance over many iterations. This likely has a lot to do with the aggressive regularization in the three-layer model, allowing it to somewhat avoid over-fitting even

though it was a complex model. The two layer model, on the other hand, was still relatively complex (8 hidden nodes) and had very little regularization, making it prone to overfitting on the training set.

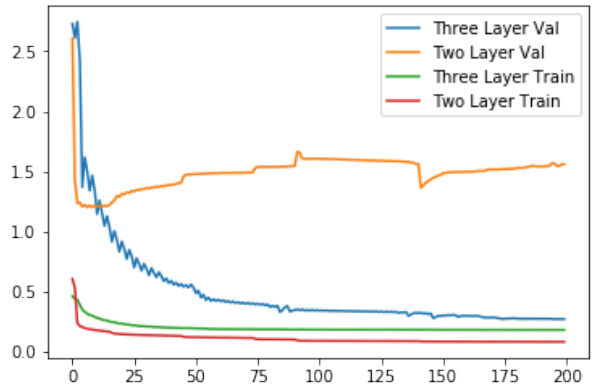


Fig. 18: Training and Validation Errors for Two and Three-layer Perceptron with $\sigma = 0.18$

In order to visualize the degradation in performance, we also plotted the test set performance. We see that test predictions are not very different for two or three-layer perceptrons, though both predictions are very far off from the true series.

Fig. 19: Test Set Performance for Two and Three-layer Perceptron with $\sigma = 0.18$

There is a trade-off between using a simpler model and a more complex model. When we have a lot of data and little noise, we can afford to use a more complex model whereas a simpler model is less susceptible to over-fitting when there are a small number of noisy data points.

The development of backpropagation was important primarily because of its efficiency compared to earlier, more naive methods. If W is the total amount of weights and biases in the network, then a single evaluation of the error function is $\mathbf{O}(W)$. This follows from the fact that much of the computational complexity comes from summing the inputs to a node, rather than calculating a node's activation. Directly evaluating the derivatives of the error function with respect to the weights would lead to a complexity that was $\mathbf{O}(W^2)$. Backpropagation allows these derivatives to be evaluated in $\mathbf{O}(W)$.

V. CONCLUSIONS

In this paper, we studied the performance of a single layer Perceptron using both the Perceptron and the Delta weight update rule. We also studied the performance of a multi-layer Perceptron neural network in classification and regression. We performed our tests using both batch and sequential learning.

Our results are in accordance with the Perceptron convergence theorem which states that a single layer Perceptron always converges when the data is linearly separable. However, if the data is non linearly separable an at least two-layer Perceptron is needed with at least as many hidden layer neurons as the hyperplanes needed to separate the classes. Furthermore, we found that using too complex a neural network (too many neurons) for a relatively small dataset leads to overfitting. We also found that for all cases the sequential type of learning convergences much slower than a batch learning approach.

Finally, we evaluated the performance a multilayer Perceptron on time series analysis using the Mackey-Glass time series. Our results indicate that early stopping and regularization do not improve the performance if enough training data is fed to the network and there is no noise in that data. Nevertheless, if noise is added to the data we show that regularization by weight decay or early stopping increases performance by reducing the variance of our model.

REFERENCES

[1] Bishop, C.M., 2016. Pattern Recognition and Machine Learning. Springer-Verlag New York.
[2] Marsland, S., 2015. Machine learning: an algorithmic perspective. CRC press.