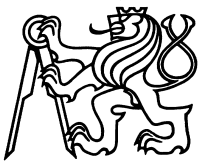
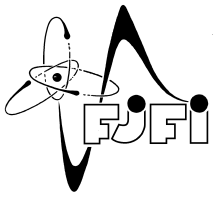


CZECH TECHNICAL UNIVERSITY
IN PRAGUE

Faculty of Nuclear Sciences and Physical
Engineering



Stochastic methods combining neural networks with random forests and their applications to medical data analysis

Stochastické metody propojující neuronové sítě s náhodnými lesy a jejich využití v analýze medicínských dat

Masters's thesis

Author: **Bc. Martin Oharek**

Supervisor: **Ing. Marek Bukáček**

Academic year: 2019/2020

Katedra: matematiky Akademický rok: 2019/2020

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student:	Bc. Martin Oharek
Studijní program:	Aplikace přírodních věd
Studijní obor:	Matematické inženýrství
Název práce (česky):	Stochastické metody propojující neuronové sítě s náhodnými lesy a jejich využití v analýze medicínských dat
Název práce (anglicky):	Stochastic methods combining neural networks with random forests and their applications to medical data analysis

Pokyny pro vypracování:

1) Sumarizujte základní klasifikační algoritmy s důrazem na rozhodovací stromy a neuronové sítě.
2) Proveďte rešerši možností propojení výše zmíněných metod.
3) Proveďte analýzu medicínských dat z projektu Lucas.
4) Navrhněte vhodný algoritmus kombinující rozhodovací stromy a neuronové sítě a aplikujte ho na medicínská data.
5) Analyzujte výstupy a vyhodnoťte výsledky ve srovnání s alternativními metodami.

Doporučená literatura:

1) G. Biau, E. Scornet, J. Welbl, Neural Random Forests. Sankhya A, 2018. \url{https://doi.org/10.1007/s13171-018-0133-y}
2) L. Breiman, Random Forests. Machine Learning, Kluwer Academic Publishers, 2001, 5-32.
3) A. Criminisi, J. Shotton, and E. Konukoglu, Decision Forests: A Unified Framework for Classification, Regression, Density Estimation, Manifold Learning and Semi-Supervised Learning. Foundations and Trends in Computer Graphics and Computer Vision, Now Publishers Inc., 2012, 81-227.
4) M. A. Nielsen, Neural Networks and Deep Learning. Determination Press, 2015.

Jméno a pracoviště vedoucího diplomové práce:

Ing. Marek Bukáček
Katedra matematiky \\Fakulta jaderná a fyzikálně inženýrská \\České vysoké učení technické v Praze \\Trojanova 13 \\120 00 Praha 2

Jméno a pracoviště konzultanta:

Datum zadání diplomové práce: 31.10.2019

Datum odevzdání diplomové práce: 4.5.2020

Doba platnosti zadání je dva roky od data zadání.

Acknowledgement:
I would like to thank my supervisor, Ing. Marek Bukáček, for his guidance, willingness and valuable advisory on this thesis. I would also like to thank my family and my girlfriend Adéla for their patience and support. My gratitude also belongs to Mgr. Věra Viertlová and Mgr. Žaneta Voldánová for their help with the language corrections.

Declaration:
I declare that this master's thesis is entirely my own work and I have listed all the used sources in the bibliography.

Prague, 29 June 2020 Martin Oharek

Název práce:
Stochastické metody propojující neuronové sítě s náhodnými lesy a jejich využití v analýze medicínských dat

Autor: Bc. Martin Oharek

Studijní obor: Matematické inženýrství

Druh práce: Diplomová práce

Školitel: Ing. Marek Bukáček

Abstrakt: Tato práce se zabývá možnou kombinací náhodných rozhodovacích lesů a neuronových sítí, která by dala vzniknout hybridním klasifikátorům s velkým klasifikačním potenciálem, využívajících předností obou těchto základních modelů. V práci jsou navrženy čtyři hybridní klasifikátory, obecně nazývány jako náhodné neuronové lesy, které jsou vytvořeny na základě transformace rozhodovacího stromu na ekvivalentní dopřednou neuronovou síť. Všechny teoretické náležitosti jsou v práci důkladně popsány a schopnost klasifikace náhodných neuronových lesů je testována na velkém množství klasifikačních úloh. Důležitou součástí je také analýza parametrů náhodných neuronových lesů a zkoumání jejich vlivu na kvalitu klasifikace. Závěry jsou podpořeny experimentálně jak na veřejných datových sadách, tak na uměle vytvořených testovacích sadách. Náhodné neuronové lesy jsou mimo jiné testovány na reálné datové sadě zahrnující medicínské záznamy získávané projektem LUCAS. Výstupy experimentů naznačily lepší klasifikační schopnost náhodných neuronových lesů než u jiných testovaných modelů.

Klíčová slova: klasifikace, náhodný neuronový les, náhodný rozhodovací les, neuronová síť

Title:
Stochastic methods combining neural networks with random forests and their applications to medical data analysis

Author: Bc. Martin Oharek

Abstract: This thesis is focused on the potential combination of random forest classifiers and feedforward neural networks into hybrid classifiers, which could benefit from the valuable properties of these two popular machine learning algorithms and could thus possess a strong classification ability. We propose four different hybrid classifiers, generally called neural random forests, which are created on the basis of transformation of a single decision tree into an equivalent feedforward neural network. We provide substantial theoretical background of related domains and verify the classification power of neural random forests on numerous classification tasks. Moreover, we analyze the effect of neural random forest parameters on the overall performance and support the conclusions by evaluating and comparing the performance on public and toy datasets. Additionally, the performance is tested on a real-world dataset, consisting of medical data collected by the LUCAS project. The vast majority of experiments suggested the superiority of the proposed neural random forest models among all tested models.

Key words: classification, neural network, neural random forest, random forest

Contents

Chapter 1

Introduction

Random forests and neural networks have belonged to the most popular machine learning algorithms for several decades already. They both have a broadband application to numerous classification and regression tasks, producing a state-of-the-art performance. Neural networks are able to successfully approximate even very complex, non-linear functions. The main disadvantage of neural networks is especially the large number of parameters to set and tune. On the contrary, random forest classifiers have much fewer parameters and are still able to perform at least competitively.

In the last couple of years, attempts to combine these two models have been carried out in order to create hybrid classifiers which could exploit the advantages of both algorithms (such as to preserve the perfect learning ability) and compensate for or reduce the effect of disadvantages (such as to decrease the number of parameters). We can mention paper [1], which describes the way of using the structure of the decision tree and initiates the architecture of the neural network alike. Afterwards, the backpropagation algorithm and gradient-based optimization methods are used to adjust the parameters, which is considered as a better alternative to simple decision tree splits, eventually producing more complex decision boundaries. Another approach is presented in paper [2], which redefines the regular split functions of the decision tree and tune their parameters via a deep neural network.

Our approach is motivated by the theory presented in [3]. It defines the transformation of a single regression tree into an equally performing neural network. Unfortunately, it is not possible to equivalently convert this approach to the classification case. Therefore, we propose an alternative for a classification case and design four different architectures that can be applied to transform a decision tree classifier into a neural network. The greatest emphasis is put especially on transforming a random forest classifier into an ensemble of neural networks. This model is called neural random forest.

Neural random forests combine several beneficial properties of random forests and neural networks, which allows them to produce competitive results and often surpass the performance of their predecessing models. We provide a detailed theoretical derivation of individual models, a verification of functionality and a thorough analysis and comparison of the performance. Their classification ability is tested in many experimental tasks, including the application to a real dataset and a comparison with other competitive models.

Main contributions of this thesis are as follows:

- We design four different types of neural random forest, including detailed theoretical descriptions and illustrations.
- We compare the performance of neural random forest models to each other and also to other competitive models, such as random forest, neural network and logistic regression. The performance is evaluated on both public and real datasets with respect to several evaluation metrics.
- We investigate and describe the effect of neural random forest parameters on the overall performance, which is tested on different datasets. In addition, the decision boundaries are depicted on toy datasets, which have been artificially created to simulate various deployments of data points in $\mathbb{R}^2$. The shape of the decision boundaries reveals some of the valuable properties of different neural random forest models.
- Since we are not aware of any public programming library that deals with neural random forests of the same kind as we proposed, we develop our own Python library that provides comfortable tools to train and test the proposed neural random forest models. At the moment, we have a suboptimal solution, thus we expect a future work in this area.

The thesis is structured into several chapters and two appendixes. Chapter 2 comprises a brief introductory into basic terminology and classification and also detailed descriptions of different evaluation metrics.

Chapter 3 and 4 focus mainly on the theory behind random forests and neural networks. It provides definitions, illustrations and descriptions of related terminology and also addresses theoretical details and derivations, which are often neglected or omitted in related literature.

Chapter 5 comprises the main theoretical part of the thesis. It describes the transformation of a single decision tree into a neural network, which is used in all subsequently proposed models. This chapter contains theoretical details and thorough illustrations, which depict the connection between a decision tree and the architecture of a neural network and simplify the understanding.

Chapter 6 is closely related to chapter 5. It presents all proposed transformations of a single decision tree into a neural network, which are later applied to form different neural random forest models. In this chapter, the transformation of random forest to an ensemble of neural networks is also explained.

Chapter 7 comprises the experimental part of the thesis. This chapter is divided into several sections based on the researched topic. It includes experiments on public datasets, experiments on toy datasets and experiments on real dataset consisting of data from the LUCAS project. An important part of this chapter is the analysis of neural random forest parameters and the examination of their influence on the performance. Graphical visualizations related to the experimental part are shown in a separate appendix and referenced at the corresponding paragraphs in chapter 7.

The conclusion of the thesis is presented in chapter 8.

Chapter 2

Classification

2.1 Terminology

In this thesis we focus mainly on the task of classification, which embodies the procedure of assigning one or more labels to the observations of studied area. Individual labels represent different *classes*, which group observations with common features. We distinguish two main categories of classification - *binary* and *multiclass* classification. In the binary case we discriminate 2 classes, which are generally denoted as 1 and 0 for work purposes. They could represent either a positive or a negative case. Specifically, we could distinguish for example *rain* vs *no rain*, *ill* vs *healthy*, *declined* vs *granted* and so on. In the multiclass case we discriminate several different classes (binary classification is a special case of multiclass classification). In this thesis we focus exclusively on the classification of instances to only one class (only one label is predicted). There is also a multi-label classification, which can assign multiple labels to an instance, but we do not use this type of classification.

Classification can be included among *supervised learning*, which belongs to the main machine learning paradigms:

1. **Supervised learning**: This paradigm comprises machine learning task of learning a function that maps input value to output value based on delivered input-output evidence. The input observations are usually vectors lying within input space $\mathbb{X} \subset \mathbb{R}^d$, where $d \in \mathbb{N}$ denotes dimension. $\mathbb{X}$ is often referred to as *feature* space and d corresponds to a number of features. Vectors belonging to feature space will be referred to as *instances* (instance of feature space). Each feature usually represents a certain property of the instance. They can be either *numerical*, such as weight or age, or *categorical*, such as gender or marital status. Sometimes we also work with features that do not provide such straightforward interpretation as previously mentioned cases. These features are usually produced artificially by special preprocessing techniques, e.g. dimensionality reduction [4], data compression, etc.

   In the case of classification, we must deliver set of labelled instances to the machine learning model, which then learns a function that maps instances to labels (finite set of discrete values). This model could be used afterwards on the previously unseen instances from the same feature space. If the labels are continuous, we talk about *regression* instead of classification.

2. **Unsupervised learning**: Unlike the supervised learning, this paradigm does not work with previously labelled evidence. Algorithms belonging to this group look for patterns and structures among a set of unlabelled instances from feature space. Typical representative of this group are *clustering* algorithms [5], which create labelled groups among previously unlabelled data.

3. **Reinforcement learning**: This paradigm comprises a group of machine learning algorithms, whose learning is based on feedback. In other words, the algorithm monitors surroundings and performs actions in order to maximize predefined reward. It does not require labelled inputs as in the case of supervised learning methods.

Before we train the classification model (to learn the mapping function), we usually divide the available set of instances into the training subset and the testing subset (occasionally an extra validation subset). Then, the training subset is used to train the classification model and the testing subset is used to evaluate its performance. Validation subsets are often used to tune the parameters of the model (if necessary). When we do not explicitly supply the training and testing subsets, the standard way is to randomly split the set of instances multiple times and each time choose one subset as a training set and the other one as a testing set. The example of this procedure is *K-fold cross validation*. This procedure divides the dataset into $K \in \mathbb{N}$ disjunct parts (usually of the same length) and then the K-1 parts are used as the training set and the remaining part is used for testing. In the next iteration, another K-1 parts are picked for training and the rest for testing. This procedure is repeated for K times, so in each iteration a different combination of training and testing set is applied. Then, the evaluation results are usually averaged across all iterations of the procedure.

Speaking of the classification and machine learning in general, we must mention also two related problems associated with the training of classification models. The first one is called *underfitting*. This problem often arises, when the complexity (usually measured by a number of parameters of the model) of our model is too low. Simply put, the model is too easy. In this case the model is unable to fit sufficiently to the training data, which causes a significant testing error on the testing dataset. We say that the model has high bias and low variance. On the contrary, the *overfitting* problem arises, when the complexity of our model is too big and the model is too complicated. In this case, the performance on the training dataset is perfect, but the performance on the testing dataset is poor. Overfitted models tend to approximate the training data too accurately, which substantially degradates their generalization ability. We say that the model has low bias and big variance. Both problems are undesirable and degradate the performance of our model.

The evaluation of model performance is a crucial part of classification and will be discussed in the following section.

2.2 Evaluation

This section is focused on the summary and description of evaluation metrics used in the experimental part, alongside the properties of each metric and suitable application. This section includes only metrics that are applied in the testing phase. The correct choice of evaluation is the main and crucial task in the case of searching for an optimal classification model. In practice, there are numerous evaluation methods applied frequently to many classification tasks. Before we choose a set of evaluation metrics, we must be aware of the processed data and classification model to set our

evaluation correctly. Also the purpose and future application scope of our model is significant. If this foremost analysis of the data and model is neglected or done wrongly, we could end up choosing bad-shaped evaluation techniques and therefore misinterpret the performance of our model.

In the literature we can find a significant amount of studies addressing the choice of evaluation metrics in the case of either binary classification [6, 7] or multi-class classification [8, 9].

The classification models in this thesis are tested on multiple datasets, both binary and multi-class classification oriented. Consequently, there is not any particular property that could discriminate correct (the most correct) setting of evaluation methods in general, unless we study each dataset separately, which is not the purpose of this thesis at all. We only desire to compare the classification performance of neural random forests mainly to the performance of regular random forests and provide evidence of superiority of neural random forests. Therefore we select subset of worldwide-accepted, well-functional evaluation techniques, summarize them and describe them separately. A combination of these metrics provides sufficient evaluation to form a conclusion.

Firstly, we define an important notation that is referenced in the upcoming descriptions. This applies generally to the multi-class classification with arbitrary number of classes. The set of all classes is denoted as $\mathbb{C}$.

Definition 2.2.1. *Positive* class $c \in \mathbb{C}$ is such a label that is in the current scope of interest. Other classes $d \neq c$, $d \in \mathbb{C}$, are called *negative* classes.

To clarify 2.2.1, if we test the classification performance of our model with respect to class $c \in \mathbb{C}$, then c is called a positive class and other labels $d \neq c$, $d \in \mathbb{C}$, are called negative classes. It is important to realize that there could be many positive classes. It depends mainly on the current scope of interest. For instance, computing classification evaluation of class $c_1 \in \mathbb{C}$ means that c_1 is (currently) a positive class and other classes are negative. In the following, current positive class c will be put into parentheses (such as $\mathrm{TP}(c), \mathrm{FN}(c)$ etc.) Instances belonging to the positive, resp. negative class could also be referred to as positive, resp. negative instances.

$\mathrm{TP}(c)$	number of positive instances (class $c \in \mathbb{C}$) correctly predicted
$\mathrm{FP}(c)$	number of negative instances predicted incorrectly as positive class c
$\mathrm{TN}(c)$	number of negative instances (with respect to the positive class c) correctly predicted
$\mathrm{FN}(c)$	number of positive instances (class $c \in \mathbb{C}$) incorrectly predicted as negative class (with respect to the positive class c)

Table 2.1: Definitions of TP,FP,TN and FN.

Accuracy

Accuracy is probably the most frequently applied evaluation metric in the machine learning society, which provides quick and simple evaluation of the classification performance. The outcome of this metric is the ratio between the number of correctly classified instances and the number of all instances. It is usually a single value per classifier (not specific to individual classes) and it is defined as

$$accuracy = \frac{\text{number of correctly classified instances}}{\text{number of all instances}} \quad . \tag{2.1}$$

Altough accuracy can simply provide an evaluation of the quality of classification, it possesses some significant drawbacks [8] and therefore relying only on this metric in final evaluation is dangerous. It does not take into account whether the instances belong into a positive or a negative class. A serious problem arises especially when processing imbalanced data [10], which is the case of many real-world applications.

For instance, imagine data distributed into two classes, class 0 with 98% occurrence and class 1 with only 2% occurrence. This dataset is clearly significantly imbalanced. Moreover, we give higher priority to correctly classifying minority class 1 (e.g. consider class 1 as malware type communication and class 0 as arbitrary safe network communication). We also define a naive classifier, which constantly classifies any instance as class 0. In this case, accuracy of this model is $Acc = 98\%$, which is very high. Based only on this value, the classification performance of our naive model is almost perfect. But claiming that this model is good is wrong, since it completely fails in classifying minority class, which has the highest priority. So even though accuracy is very high, the model itself is absolutely useless.

Precision

Another evaluation metric is called precision and is class-sensitive. In contrast to accuracy, it is computed for any particular class $c \in \mathbb{C}$ as

$$precision(c) = \frac{\mathrm{TP}(c)}{\mathrm{TP}(c) + \mathrm{FP}(c)} \quad . \tag{2.2}$$

It expresses the ratio between correctly predicted positive instances and all instances predicted as positive. In other words, it shows how precise our classification model is in predicting a positive class.

This metric is also dependent on the rate of imbalance among current data. Let's suppose that negative instances are added to the dataset. Then

$$\mathrm{FP}^{\mathrm{before}}(c) \leq \mathrm{FP}^{\mathrm{after}}(c) \implies precision^{\mathrm{before}}(c) \geq precision^{\mathrm{after}}(c)$$

So even if our classifier predicts all positive instances correctly, the precision tends to decrease with the increasing number of negative instances.

Recall

This metric is computed for class $c \in \mathbb{C}$ as

$$recall(c) = \frac{\mathrm{TP}(c)}{\mathrm{TP}(c) + \mathrm{FN}(c)} \quad . \tag{2.3}$$

Recall expresses the ratio between the number of correctly classified positive instances and the number of all positive instances. Clearly, this metric does not depend on the rate of imbalance in our data and therefore is a suitable choice for the evaluation on imbalance datasets.

In many applications of classification on imbalance datasets, there is a strong emphasis on a model predicting minority classes as precisely as possible (capture all positive instances, but also do

not cause a lot of mistakes on negative instances), because misdetections could lead to a significant damage (e.g. malware detection). Some studies indicate that a combination of precision and recall is the appropriate choice to satisfy these requirements and provides reliable evaluation [11].

F-measure

F-measure metric combines precision and recall into a single value as

$$F_\beta(c) = \frac{(1+\beta^2)\cdot precision(c)\cdot recall(c)}{\beta^2\cdot precision(c) + recall(c)} \ , \tag{2.4}$$

where $c \in \mathbb{C}$ and β is a coefficient which serves to adjust relative importance between precision and recall (often $\beta = 1$). For $\beta = 1$ it expresses harmonic mean of precision and recall. As $\beta \to 0$ the formula considers only precision and as $\beta \to \infty$ the formula considers only recall. Generally, $\beta < 1$ favors precision and $\beta > 1$ favors recall. It depends on each individual application to choose β appropriately. For $\beta = 1$ we refer to the corresponding F-measure as F1-score.

G-measure

G-measure is computed as

$$G(c) = \sqrt{\frac{\mathrm{TP}(c)}{\mathrm{TP}(c)+\mathrm{FN}(c)}\cdot\frac{\mathrm{TN}(c)}{\mathrm{TN}(c)+\mathrm{FP}(c)}} \ , \tag{2.5}$$

where $c \in \mathbb{C}$. The left factor in multiplication under the square root in (2.5) is called *sensitivity* or *true positive rate* (the same as recall) and the right factor is called *specificity* or *true negative rate*.

Optimization of this metric secures good balance between the classification performance on both minority and majority classes. In the case of imbalance, even if the classification performance on negative instances is perfect, G-measure would end significantly low if classification of positive instances is poor [12]. This is a valuable property which makes this metric well-applicable to evaluation of the classification model on imbalanced data.

Note: Since we compare different classification models and track their overall performance on various datasets in the experimental part, the main purpose is not to compare metric values for each individual class, but rather to compare summaries of these evaluation outcomes. For this reason, we adopt **macro/weighted average** to represent individual metrics for different classes as a single value.

For arbitrary (class-sensitive) evaluation metric f, macro/weighted average is defined as

$$macroAvg(f) = \frac{\sum_{l=1}^{M} f(c_l)}{M} \tag{2.6}$$

$$weightedAvg(f) = \frac{\sum_{l=1}^{M} n_l \cdot f(c_l)}{\sum_{l=1}^{M} n_l} \ , \tag{2.7}$$

where $c_l, l \in \{1, 2, ..., M\}$ is l-th class, n_l is the number of (testing) instances belonging to c_l and M is the total number of classes.

It is also worth mentioning **micro average** of precision and recall, which is defined as

$$microAvg(precision) = \frac{\sum_{l=1}^{M} \mathrm{TP}(c_l)}{\sum_{l=1}^{M} (\mathrm{TP}(c_l) + \mathrm{FP}(c_l))} \tag{2.8}$$

$$microAvg(recall) = \frac{\sum_{l=1}^{M} \mathrm{TP}(c_l)}{\sum_{l=1}^{M} (\mathrm{TP}(c_l) + \mathrm{FN}(c_l))} \quad . \tag{2.9}$$

$microAvg(precision)$ computes basically fraction of the number of all correctly predicted instances and the number of all correctly and incorrectly predicted instances, which is the number of all instances. Therefore, it expresses the same value as accuracy of the model. Furthermore, since $\sum_{l=1}^{M} \mathrm{FN}(c_l) = \sum_{l=1}^{M} \mathrm{FP}(c_l)$, both values (2.8) and (2.9) are equal. Note that this may not apply in the case of multi-label classification.

ROC curve

ROC (receiver operating characteristics) curve [13] is a popular graphical technique used for the comparison of performance of classifiers mainly in the case of binary classification tasks. It is a two dimensional visualization showing the true positive rate (TPR) on the vertical axis and the false positive rate (FPR) on the horizontal axis for all meaningful decision thresholds. In the binary settings (negative class 0 and positive class 1), TPR and FPR are defined as

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{2.10}$$

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}} \quad , \tag{2.11}$$

where we omit parentheses (as defined in Table 2.2.1) since there is only one positive class and one negative class.

Each threshold represents the decision boundary between a negative and a positive class (in binary classification) and therefore defines a specific classifier. If the prediction score (e.g. probability) of the classifier exceeds the given threshold, the prediction is set to a positive class. Otherwise, the prediction is a negative class. So each threshold defines FPR and TPR pairs which are then drawn in the 2D visualization.

Meaningful thresholds are meant to be values that define different TP, FP, TN and FN combinations, which determine FPR and TPR pairs. Imagine a model classifying 5 instances belonging to 2 different classes (class 0 and class 1) labelled as 0,0,1,1,0. The model output probabilities of positive class for these instances as 0.1, 0.6, 0.8, 0.4, 0.1. Then if we choose the decision threshold as 0.2, we obtain TP=2, FP=1, TN=2, FN=0. But the same values are also obtained if the decision threshold is set to 0.3. Therefore, both choices of the decision threshold result in the same FPR and TPR pair. Thus, only one threshold is used and the other is redundant. One possible procedure of picking the thresholds would be to select the unique prediction scores (probabilities) first, then sort them by size, add the limit values below respectively above the lowest respectively the highest value and, finally, select one value in each outlined interval.

Example of ROC curves is depicted in Figure 2.1. Imagine a random classifier assigning positive class probability randomly (drawn from uniform distribution between 0 and 1) to each instance. Then its ROC curve corresponds to the line connecting points [0;0] and [1;1] (red ROC curve in Figure 2.1) [14]. This can be correctly proven by means of probability, but we can intuitively imagine

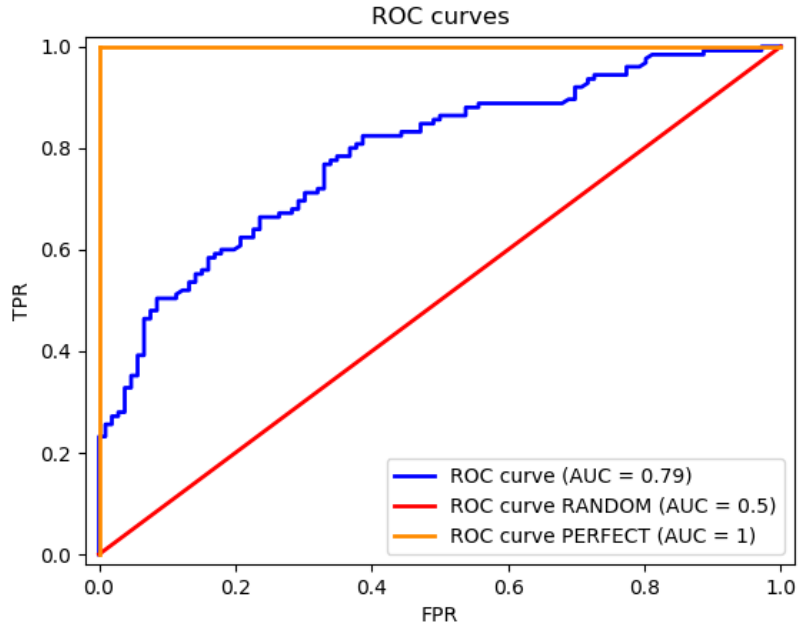


Figure 2.1: This graph shows examples of ROC curves. The blue curve represents ROC curve of random forest classifier. The red curve represents ROC curve of random classifier and the orange curve represents ROC curve of perfect classifier.

that each instance is predicted with the positive class probability of 0.5 and therefore there are only two possible thresholds (below and above 0.5), which corresponds to the [0;0] and [1;1] points in ROC space, because once everything is classified as negative and once as positive. Classifiers with ROC curve shape similar to this straight diagonal line perform similarly as random guessing, which is not very good. On the other hand, a perfect classifier assigning always 1 to positive instances and 0 to negative instances has the same shape as the orange ROC curve in Figure 2.1. This represents the best possible performance that may be reached in the ROC space. It is generally considered that the closer the classifier is to the ideal point [0;1] in the ROC space, the better performance it has.

Apparently, it could be difficult to compare some curves only visually. Another popular technique to evaluate performance of classifiers is to aggregate ROC curve into a single value by computing the area under the ROC curve (AUC) [13]. Since the ROC curves are closed in the square with the unit side, the maximum value of AUC is 1 and the minimum value is 0. AUC of the random classifier is 0.5 and AUC of the perfect classifier is 1. Higher AUC values usually characterize better performance. AUC value could also be statistically interpreted as the probability that the classifier will give higher score to the randomly chosen positive instance than to the randomly chosen negative instance [13]. The disadvantage of this method is that even very different ROC curves could reach the same AUC values.

Precision-recall curve

The precision-recall (PR) curve also graphically visualizes the performance of classification models. It depicts recall of the positive class on the horizontal axis and precision of the positive class on the vertical axis for varying thresholds. It is usually more informative in the case of strongly imbalanced datasets than ROC, where ROC curves could significantly overestimate the performance.

We can demonstrate this issue on example: consider a dataset of 10 positive instances and 1000 negative instances and a classifier which predicts everything correctly except for 100 negative instances, which are predicted as positive. Then TP = 10, FP = 100, TN = 900, FN = 0. Therefore, TPR = 1 and FPR ≈ 0.1, which reaches almost the ideal point [0;1] in ROC space. But precision of the positive class is only approximately 0.01, which is very poor. So even if the classifier reached almost the ideal point in ROC space, it could be very far from it in the PR space (the ideal point in PR space is [1;1]).

There are some technical properties of PR curves, especially regarding the estimate of the first and the last point of the graph. The first point corresponds to the case when everything is predicted as negative. Therefore, there are no positive predictions and precision is undefined (denominator is zero). This issue is usually solved by fixing the first point or estimating it from the second point. It is done in the way that the curve starts at the vertical axis. On the contrary, the last point could be determined accurately, because it corresponds to the case of predicting everything as positive. Therefore, recall is 1 and precision equals $\mathrm{P}/(\mathrm{P}+\mathrm{N})$, where P denotes the number of all positive instances and N denotes the number of all negative instances.

The PR curves in our experiments start at the [0;1] point on the vertical axis. As well as that, only the first point reaching recall = 1 (point with the highest precision and recall = 1) is visualized and the other points with the same recall are omitted, since they do not add any useful information to the graph. Example of the PR curve is shown in Figure 2.2.

The PR curve of the random classifier corresponds to the straight line connecting points [0;1] and $[1;\mathrm{P}/(\mathrm{P}+\mathrm{N})]$, whereas the PR curve of the perfect classifier corresponds to the straight line connecting points [0;1] and [1;1] (the line connecting points [1;1] and $[1;\mathrm{P}/(\mathrm{P}+\mathrm{N})]$ is omitted).

We also compare individual PR curves by a single value (similarly to the case of ROC curves), but instead of AUC we adopt *average precision score* (AP) that is computed as the weighted sum of all precision values obtained for each threshold. It is defined as

$$\mathrm{AP} = \sum_k (\mathrm{recall}^{(k)} - \mathrm{recall}^{(k-1)}) \cdot \mathrm{precision}^{(k)} \quad , \tag{2.12}$$

where $\mathrm{recall}^{(k)}$ denotes recall value at k-th threshold and analogously for precision. The AP value will be used to compare individual PR curves in the experimental part. It is a preferred choice over AUC since [15] suggests that AUC may produce over-optimistic evaluation of performance. Generally, higher AP values mean better performance.

In [15] is also proven that the ROC and PR curves are closely related. To be more precise, a curve dominates in the ROC space if and only if it dominates in the PR space (there is one-to-one relationship).

We also define a special PR curve, which summarizes an overall performance of the classifier among several independent simulations. It is created simply by concatenation of individual labels and corresponding prediction scores from all simulations and then visualizing the PR curve of this entire union. We refer to this curve as the *aggregation PR curve* and use it in the experimental part.

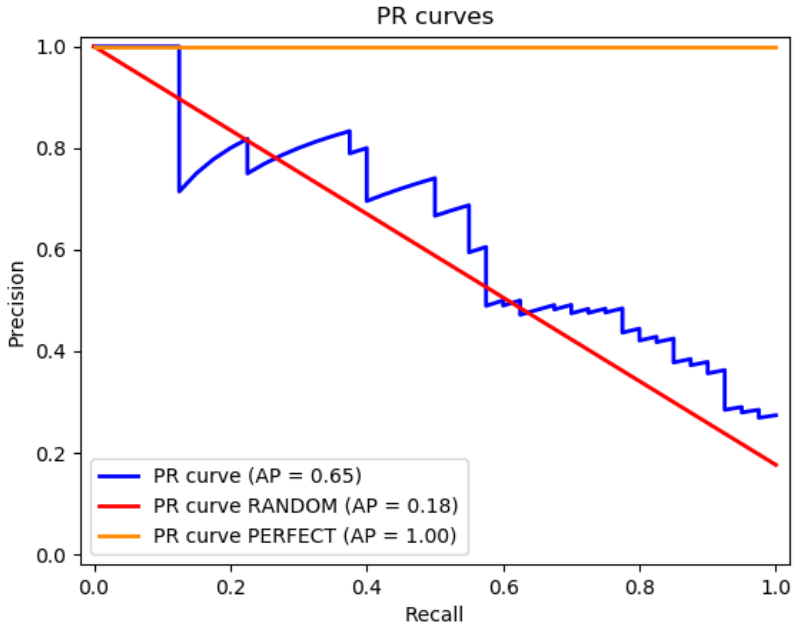


Figure 2.2: This graph shows examples of PR curves. The blue curve represents PR curve of random forest classifier. The red curve represents PR curve of random classifier and the orange curve represents PR curve of perfect classifier.

Both the ROC and PR curves can be adapted to the multiclass environment. It is usually done by applying the one-vs-all scheme (setting one class as positive and other classes as negative) and then visualizing individual curves. We applied them only on the binary classification in the experimental part, hence we will not describe the multiclass case in this thesis.

Chapter 3

Random forest

Random forest algorithm is a popular machine learning method used primarily on classification and regression tasks. Its functionality is based on collecting predictions from several independent classifiers (ensemble of decision trees) and merging their outcomes into the final prediction. It belongs to the group of supervised learning algorithms. In this thesis the usage of random forests is restricted only to classification tasks, so the theoretical part of this section will be limited only to random forests used for classification accordingly.

Random forest nowadays is a widespread method exploited frequently in various domains, such as text classification [16], bioinformatics [17], network threat detection [18] etc. Individual classifiers - decision trees - could also be used as a clustering algorithm [19]. The popularity of random forests is caused by good generalization ability and application to wide-ranged spectrum of classification tasks with registering competitive classification quality in comparison with other state-of-the-art methods, such as logistic regression [20], support vector machine [21], neural networks etc. Moreover, the method itself is in its basics quite simple and therefore it is easy to implement and could be analyzed and interpreted once it has been trained.

Unlike the neural networks, it has fewer parameters to tune and it is better-applicable to small-sample sized datasets. With the usage of randomization, bagging, bootstrapping and other techniques based on randomness, random forests overcome easily problems of overfitting as well [22].

3.1 Decision tree

As mentioned previously, a random forest consists of independent classifiers called decision trees (Figure 3.1).

In this thesis, we focus only on binary trees, which means that every node has always 2 child nodes (following nodes in the hierarchical structure of the tree). The process of applying decision tree to obtain final prediction is basically described in Figure 3.1. Beginning from the root node, the input instance $\boldsymbol{x} \in \mathbb{R}^d$ from d-dimensional input space is iteratively tested by the split functions associated with every single node in the hierarchical order and based on the result, it is further sent either to the left or the right child (the split functions usually define hyperplanes in the input space). This process is repeated until the instance reaches the final node (leaf node). In the leaf node, there are classification predictions, which are extracted at the end of the procedure.

The work with decision tree could be divided into two major parts:

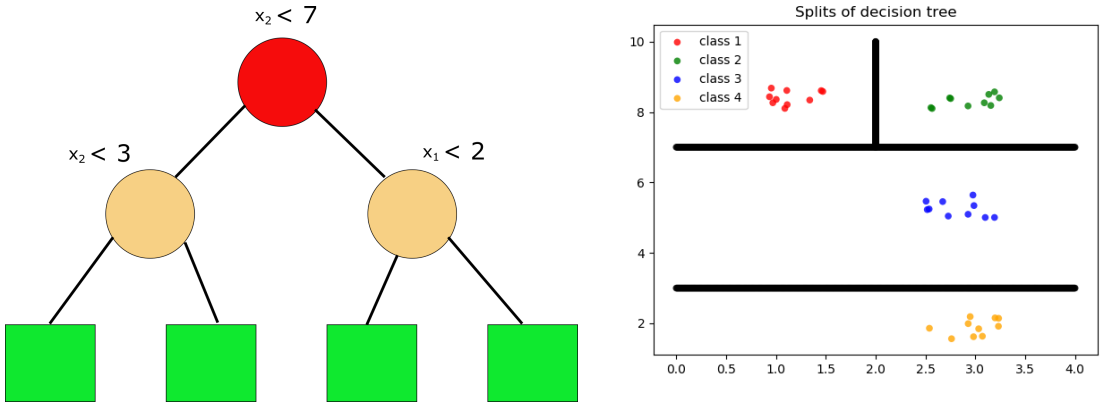


Figure 3.1: The left part of the figure illustrates a decision tree with depth = 2. The first circle in red is called the root node. The two beige circles in the intermediate layer are inner nodes and the green rectangles are leaf nodes, where probabilities of individual classes are stored . Each input instance $\boldsymbol{x} = (x_1, x_2)^T \in \mathbb{R}^2$ starts from the root node and traverses down the tree by applying subsequent split criteria in nodes until it reaches the leaf node. The right-hand part of the figure shows segregation of input space by the hyperplanes (defined in root node and inner nodes of the decision tree) parallel to the axes (black lines).

1. **Training stage**: In this stage the entire decision tree is built from the scratch on the basis of predefined criteria and properties. To perform this phase, the **training dataset** has to be adopted and used. It is usually selected as a subset of the complete data that are available. The magnitude of the training dataset depends on the complexity and application of our model (e.g. if we are about to compare different models, it should be sufficient to use fewer data, or if we prepare the model for the production application, it is desirable to use as much data as possible) and on the complexity of the problem. There is not any general consensus that dictates how large the training dataset should be. It often depends on the available amount of data, personal preferences and experience and, of course, on the quality of evaluation results.

2. **Testing/application stage**: After training, the the decision tree is directly applicable and therefore could be either tested with various evaluation methods or applied in practice. The usage of the decision tree itself is very straightforward and has already been described earlier. Therefore, we will focus mainly on the description of merging the individual results from the ensemble of decision trees and obtaining final predictions. This procedure will be described later.

Mathematical details of both stages shall be provided below.

3.1.1 Training stage

Consider dataset $\mathbb{X} = \{\boldsymbol{x}_1, ...\boldsymbol{x}_N\}$, where $\boldsymbol{x}_k \in \mathbb{R}^d$ and $k \in \{1, ..., N\}$, where N is the total number of instances in the dataset and d is the feature space dimension. The goal of the training stage is to establish a proper decision tree structure and find suitable split functions in every node that

provides the best class segregation. Splitting continues until some of the predefined stopping criteria are met.

The training begins always from the root node. Starting with the training set $\mathbb{M}$ chosen for the current decision tree (could be entire dataset or its subset), it interatively splits in each node to the two disjunct subsets $\mathbb{M}_1$ and $\mathbb{M}_2$, where $\mathbb{M}_1, \mathbb{M}_2 \subset \mathbb{M}, \mathbb{M}_1 \cup \mathbb{M}_2 = \mathbb{M}$. $\mathbb{M}_1$ corresponds to the subset of $\mathbb{M}$ sent to the left child and $\mathbb{M}_2$ to the right child. Splitting in each node is done via split functions. The shape of the split functions associated with the nodes is defined in advance. Generally, it could be expressed as the binary function

$$s(\boldsymbol{x}, \boldsymbol{\theta}) = \begin{cases} 0, & \boldsymbol{x} \text{ is sent to the left child} \\ 1, & \boldsymbol{x} \text{ is sent to the right child} \end{cases} . \tag{3.1}$$

The parameters of the split functions are the input instance $\boldsymbol{x}$ and the parameter vector $\boldsymbol{\theta}$, which defines geometric separation of the data (e.g. hyperplanes). This comprises thresholds, a group of data features which will be considered in the split function and other necessary values needed for complete determination of the split function.

In practice, a widely chosen split function is [23]

$$\psi(\boldsymbol{x}, i, \tau) = \begin{cases} 0, & x_i < \tau \\ 1, & \text{otherwise} \end{cases} . \tag{3.2}$$

The parameter vector in this case is $\boldsymbol{\theta} = (i, \tau)^T$, where $i \in \{1, ..., d\}$ represents one feature from the d-dimensional feature space and $\tau \in \mathbb{R}$ is a threshold. Together, equation $x_i = \tau$ embodies hyperplane parallel with one axis. The split function divides the instances into two groups based on their relative position with respect to this hyperplane. With regard to the fact that our proposed transformation of the decision tree into the corresponding neural network is built on the usage of this special split function (3.2), we are not going to focus on the other types of split functions in this thesis.

In each iteration, the algorithm seeks for the best split, ie. for the parameters of the split function that provides this split. It is achieved by optimizing a pre-specified split criterion, which expresses measure of (non)impurity in the corresponding node. Impurity means that if the node contains instances belonging only to one class, the impurity is 0, whereas it increases if instances of multiple classes are present in the node.

Among widely applied impurity functions belong **Entropy**, **Gini index**, which are adopted in the heuristic algorithms as C4.5 [24], CART [25] and ID3 [26]. Another possibility is a **Classification error**. They are defined as (in the same order as mentioned)

$$H_E(\mathbb{M}) = -\sum_{c \in \mathbb{C}} p(c) \log p(c) \tag{3.3}$$

$$H_G(\mathbb{M}) = \sum_{c \in \mathbb{C}} p(c)(1 - p(c)) \tag{3.4}$$

$$H_{CE}(\mathbb{M}) = 1 - \max_{c \in \mathbb{C}} [p(c)] \ , \tag{3.5}$$

where the sum iterates over all classes $c \in \mathbb{C}$, $\mathbb{C}$ is a set of all classes, and

$$p(c) = \frac{\sum_{\boldsymbol{x} \in \mathbb{M}} 1|_{\boldsymbol{x} \in c}}{|\mathbb{M}|} \ , \tag{3.6}$$

which is the probability of an instance in $\mathbb{M}$ belonging to class c. $|\mathbb{M}|$ represents the number of instances in set $\mathbb{M}$ and

$$1|_{\boldsymbol{x}\in c} = \begin{cases} 1, & \boldsymbol{x} \in c \\ 0, & \text{otherwise} \end{cases} \quad . \tag{3.7}$$

These impurity functions are in general exploited in the split criteria, which are then optimized in order to obtain the best possible split. By optimizing these criteria should decrease the impurity function value after each split. There are plenty of possibilities to choose a split criterion that suits our problem, but the most popular choice is often considered to be the **Information gain** criterion, which is defined as [23]

$$I(i, \tau, \mathbb{M}) = H(\mathbb{M}) - \sum_{i \in \{1,2\}} \frac{|\mathbb{M}_j(i,\tau)|}{|\mathbb{M}|} H(\mathbb{M}_j(i,\tau)) \quad . \tag{3.8}$$

$H(\mathbb{M})$ is the arbitrary impurity function (e.g. entropy, gini index etc.) of set $\mathbb{M}$. $\mathbb{M}_j(i,\tau), j \in \{1,2\}$, represent the subsets of $\mathbb{M}$ proceeding to the left or right child. It is given in the parentheses, that these subsets strictly depend on parameters i, τ of the split function. By maximizing information gain in each node final parameters i^*, τ^* are obtained. These are stored and used in the testing stage. Thus, parameters of the split function of the j-th node are acquired as

$$\boldsymbol{\theta}_j^* = (i_j^*, \tau_j^*)^T = \arg\max_{(i,\tau)} I(i, \tau, \mathbb{M}_j) \quad , \tag{3.9}$$

where $\mathbb{M}_j \subset \mathbb{M}$ is the subset of $\mathbb{M}$ in the j-th node.

Figure 3.2 shows a basic simulation of finding the best split.

	Horizontal split	**Vertical split**
IG (Entropy)	0.34	**0.69**
IG (Gini)	0.135	**0.25**
IG (Classification error)	0.20	**0.25**

Table 3.1: Values of IG for splits in Figure 3.2.

We can see that the change of the horizontal split into the vertical split resulted in the increase in values of all studied criteria (see Table 3.1). This suggests that the vertical split divides data better than the horizontal split, and hence should be preferred.

Apart the split function and the split criterion, another necessary feature must be added to the decision tree model in advance. We need to define the stopping criterion, which interrupts the growth of the tree in the current node when the specified condition is met.

One natural choice of the stopping criterion is to stop splitting when the corresponding node contains instances belonging to the same class. Therefore, it does not make sense to continue in splitting and the growth can be ended. It is not possible to continue either if only one instance ends in the node. In practice, multiple different criteria are often combined, which helps to avoid overgrown trees prone to overfitting. If one of them is fulfilled, the growth in the current node is interrupted. As an example could be mentioned [27]:

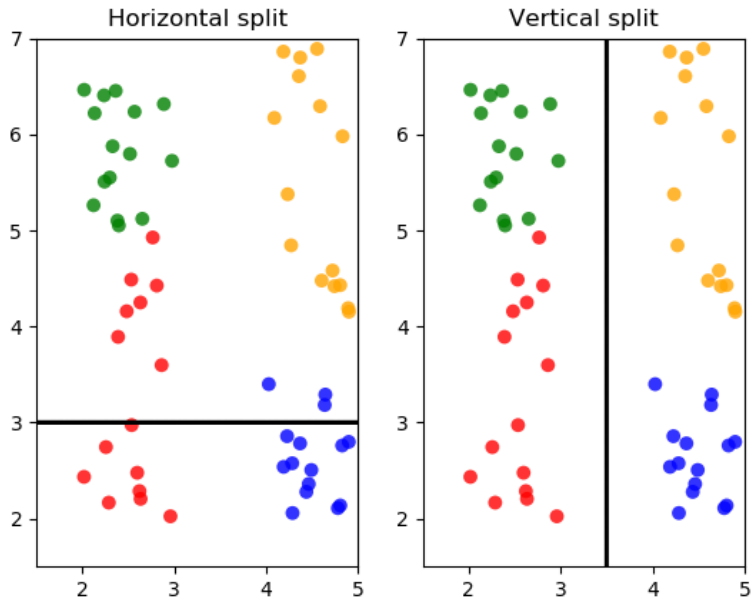


Figure 3.2: The figure depicts two cases of different splits.

- *Low information gain*: Lower threshold for the information gain is set. When a subsequent split produces insufficient information gain below specified threshold, the growth is terminated and the node becomes a leaf node.
- *Maximum depth*: When the number of splits in the current tree branch reaches the maximum depth, the current node becomes a leaf node.
- *Maximum instances in the leaf*: This is a threshold for the maximum number of instances in the leaf nodes. When the number of instances in the current node is equal or below this threshold, the node becomes a leaf node.

All of these stopping criteria defined in advance belong to the *prepruning* methods [27]. The fact that these methods are defined in advance embodies certain drawback. For instance, we do not know in advance, if the predefined maximum depth would produce sufficient requirements. Maybe it is too small and the decision tree will tend to underfit or is too large and the decision tree will tend to overfit.

For this reason, *postpruning methods* can be adopted for the large-depth trees which encounter this issue and solves efficiently the problem of overfitting. They have not been applied in our experiments, we will provide only their brief overview [28].

- *Reduced Error Pruning (REP)*: The model is pruned by usage of the independent validation (pruning) dataset different from the training dataset. The subtree starting from the chosen inner node is replaced by the leaf node and the classification error on the validation dataset is

measured. If the error is lower than the error in case of the previous non-pruned tree, the tree is pruned. This procedure runs in the bottom-up manner and is ended when no improvement in the classification error is observed.

- *Pessimistic Error Pruning (PEP)*: Unlike the REP method, PEP uses entire training dataset for the pruning procedure. It is based on the continuity correction of the error rate, since the error rate on the training dataset is significantly biased. Details could be found in [28] alongside the descriptions of other applied methods.

When the growth of the tree is successfully finished, the class distributions of the instances that ended in each leaf node form a probability distribution of classes, which is stored in the leaf node. For probability $p^j(c)$ of class $c \in \mathbb{C}$ in the leaf node $j \in \{1, ..., L\}$, where L is a total number of leaf nodes applies

$$p^j(c) = \frac{n_c^j}{N^j} \quad , \tag{3.10}$$

where n_c^j is the total number of instances of class c in the leaf node j and N^j is the total number of instances in the entire leaf node j.

3.1.2 Testing/application stage

After the training stage, the decision tree is obtained and could be applied in practice. Starting from the root node, the testing instance $x \in \mathbb{X} \subset \mathbb{R}^d$ is traversed down the tree by iterative application of the split functions associated with individual nodes. When the instance reaches the leaf node, the probability distribution of classes in the leaf node is extracted and the values are observed. The class which has the maximum value is output as the final prediction. Mathematically, the output of the single decision tree is

$$c^* = \arg\max_{c \in \mathbb{C}} p^j(c) \quad , \tag{3.11}$$

where j is the leaf node where x ended.

3.2 Ensemble of decision trees

An ensemble of decision trees forms a strong classifier with higher predictive ability than individual decision trees. The idea behind this is to combine several weak classifiers to create one strong classifier. Ensembles generally boost performance in comparison with the single classifier (decision tree). For instance, single classifiers might often get stuck in the local optima when optimizing different criterions (e.g. minimizing error rate). With more and more data, it could be computationally unreachable for many classifiers to find the best optimal value. In such cases, application of ensemble methods is valuable, because training of independent classifiers and merging their output could approximate the real decision function far better [29].

Ensemble methods also significantly benefit from averaging results from individual classifiers, especially in the case of insufficient number of training data points. In such a case, the learning algorithm could find many applicable parameters (e.g. hyperplanes of the decision tree) that will result in the same accuracy on the training dataset. Combining those models could suppress the possibility of choosing a wrong classifier [29].

Among popular techniques to create an ensemble of decision trees belong:

- **Bagging**: This technique divides the training datasets into several subsets with replacement (bootstrapping) and each subset is used to train the decision tree. Then the ensemble of the decision trees is obtained and the result from individual decision trees is combined. This technique helps to reduce the variance of a single decision tree [30] (the variance means, that if we compare the models trained on slightly different datasets, they could vary a lot (unlike the linear regression, for example, which has generally low variance)).
- **Boosting** This method is based on the idea of iterative boosting of weak classifiers into a strong one. In each iteration, the algorithm aims to boost the previous-staged classifier by checking the misclassified instances and attempting to fix these errors. An example of a popular boosting algorithm is AdaBoost algorithm [31].

In the experimental part of this thesis an entire training dataset is used for tuning individual decision trees, thus it is a variation of the bagging technique with bootstrapping of N instances, where N is the total number of instances in the training dataset. To obtain a random forest from the ensemble obtained by bagging algorithm, *randomization* of single trees needs to be added into the process. This technique helps to avoid correlation between individual decision trees and secures better indepedence of classifiers. The idea is to suppress the influence of strongly distinguishing features (dimensions) that are natural choice to split by. Occurrence of these features could result in the correlation of the decision trees, because the algorithm would prefer those features especially in the early stages of the growing phase and therefore indicates detectable similarity of the trees. Randomization attempts to obviate this problem and selects random subset of features in each splitting iteration and only those selected features are used for the current split.

3.2.1 Combining predictions of the decision trees

As mentioned previously, either probability distribution of classes from the leaf node or directly predicted class label could be extracted from every decision tree. Naturally, there are numerous possibilities of merging these results. Unless there is a special requirement, two ways are used in practice. If the outputs are discrete class labels, than the voting scheme is applied and the class with maximum votes is taken as the final prediction. If the outputs are probability distributions, those values are averaged and the class with maximum probability is taken. To be more precise, the final prediction in this case is

$$c^* = \arg\max_{cin\mathbb{C}} \frac{1}{|\mathbb{T}|} \sum_{t \in \mathbb{T}} p_t(c) \ , \tag{3.12}$$

where $\mathbb{T}$ is a set of all trees in the random forest and $p_t(c)$ is a resulting probability distribution from the tree $t \in \mathbb{T}$. In (3.12) a weighted average could be used instead, which is applied e.g. if predicting a certain class is more preferred over predicting others.

Chapter 4

Neural network

4.1 Introduction to neural networks (NN)

For thousands of years, people have been pursuing the dream to perfectly simulate human or animal brains and adopt their ability to solve problems on the daily basis. For this purpose, the (artificial) neural network was developed and nowadays it is frequently used for solving difficult data modelling tasks, statistical analysis and many others. Its functionality is based on simplified version of real brain information processing and reasoning, where single processing units of brain - neurons - transfer information to other neurons in a predefined structure and get activated based on the activity status of other adjacent neurons. In the true brain, the signal from the neuron is transferred to the connected neuron by the synapse. When the neuron accumulates all coming signals from synapses, then it evaluates this coupled signal and if it exceeds the threshold, the neuron sends signal through the synapse (connection) to another neuron. This principle is basically what drives artificial neural networks today.

The beginnings of the artificial neural network date back to the early 1950s, almost simultaneously with the development and application of programmable electronic units. In 1943, Warren McCulloch and Walter Pitts introduced models inspired by real brain neurons and adopted threshold switches [32]. They provided the evidence that even simple model (network) motivated by this approach could compute almost any logic or arithmetic function. In 1957-1958, Frank Rosenblatt and Charles Wightman et al introduced the first successful neurocomputer, Mark I Perceptron, which could identify 20x20 pixel images of simple patterns. In 1959, Frank Rosenblatt covered different version of the perceptron and formulated and proved the convergence theorem of the perceptron (the term of perceptron will be described later). Then Marvin Minsky and Seymour Papert published a mathematic paper that studied perceptron in 1969. The conclusion of this paper pointed out the insufficiency of the single perceptron, which was not able to represent classical boolean XOR or sets that are not linearly separable [33]. This led to an approximately 15-year black-out in the field of the neural network research. In 1974 learning algorithm called backpropagation of error was introduced by Paul Werbos [34]. Suprisingly, this algorithm was fully acknowledged only approximately ten years later. The backpropagation of error as the learning procedure was further developed and expanded in 1986 and widely published by the Parallel Distributed Processing Group [35]. That time, the non-separability could be effectively solved by multilayer perceptrons and previous negative conclusions about perceptrons were disproved right away [36]. Since then,

the development in the field of neural networks and machine learning in general proceeded intensively and it actively continues. It is not even possible to mention all the milestones in this thesis. Many different neural networks architectures and training algorithms have been tested, resulting in complex deep learning structures able to handle a massive amount of unlabelled data. They are sucessfully applied in many challenging tasks as bioinformatics, speech recognition, natural language processing etc [37]. Special NN called convolutional neural networks (CNN) are frequently used especially in the field of image processing.

4.2 Building blocks of ANN

In this section we are going to describe the main components of an artificial neural network. As we mentioned in the introductory section, architecture of neural network consists of processing units - neurons, and connections between neurons. A general neuron is associated with so called *propagation function*, *activation function* and *output function*. These functions process internally the input from other neurons and build the output of a neuron. The propagation function firstly considers all inputs to the neuron and the output of the propagation function serves as an input for the activation function. The activation function transforms this information into an activation value of current neuron and this activation value is then transformed by the output function and, subsequently, the output is sent by the connection to the following neuron. The entire process is illustrated in Figure 4.1.

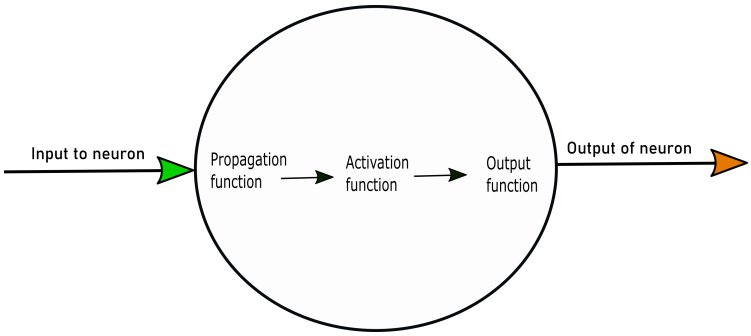


Figure 4.1: Processing of the input and creating the output of neuron could be formulated as a composition of three functions - the propagation function, the activation function and the output function.

As the propagation function is almost solely used a simple weighted sum of all inputs generally with an addition of a constant (called bias). Consider the input as a vector (if a single neuron is connected by the connections from multiple neurons) $\boldsymbol{x} = (x_1, x_2, ..., x_k)^T$, where $k \in \mathbb{N}$ is the number of inputs to neuron and $b \in \mathbb{R}$ is a bias and $\boldsymbol{w} = (w_1, ..., w_k)^T \in \mathbb{R}^k$ are connection weights used for the weighted sum. Then the output $z \in \mathbb{R}$ of the propagation function $f : \mathbb{R}^k \to \mathbb{R}$ is

$$z = f(\boldsymbol{x}) = \sum_{i=1}^{k} (w_i \cdot x_i) + b \ . \tag{4.1}$$

The term *weights* is crucial for the neural network terminology. Weights are usually associated with the connections and bias term is associated with the neuron. It depends mainly on personal preference how to imagine and comprehend the neural network settings.

As the output function of the neuron a simple identity function is used in most cases. This thesis is not going to focus on details of other possible choices of propagation and output functions, since a weighted sum for the propagation function and the identity function for the output function are a common choice in practice and other options are very rarely used.

With these presumptions on the propagation and output function, the output of a neuron is fully controlled by the activation function, which takes as input the weighted sum of input values with an added bias. The procedure for the arbitrary activation function ϕ is visualized in Figure 4.2.

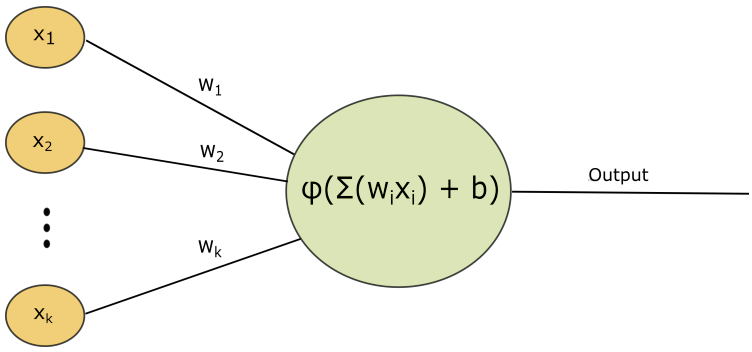


Figure 4.2: The sum of input values $x_1, ..., x_k$, where $k \in \mathbb{N}$ weighted by the weights $w_1, ..., w_k$ of connections with an added bias term b is taken as argument of the arbitrary activation function ϕ. The resulting value of the activation function embodies the amount of activity produced by the neuron. The activity value of the neuron is taken as the single output of the neuron and could be transferred to other connected neurons.

The bias term might be thought as the negative threshold of the neuron ($b = -\text{threshold}$). When the weighted sum exceeds the absolute value of the threshold, it gets activated. This discrete behaviour is perfectly simulated by a special neuron - perceptron. Its activation function is so called threshold function $\tau : \mathbb{R} \to \mathbb{R}$ with a binary output

$$\tau(z) = \begin{cases} 1 & z > 0 \\ 0 & \text{otherwise} \end{cases} . \tag{4.2}$$

If the argument of τ is positive (the threshold value for this neuron is 0), the output is 1, otherwise it is 0. Even though this simple model is applicable to many different tasks, it reliably solves only problems with linearly-separable data. This is due to the determinative equation $\sum_{i=1}^{k}(w_i \cdot x_i) + b = 0$, which defines a hyperplane in $\mathbb{R}^k$ and divides the space in two half-spaces. This drawback was surpassed by the application of multilayer perceptrons, which is a term for interconnected layers of multiple perceptrons (neural network with perceptrons).

4.3 Architecture of neural network

Since we have defined a general shape of one processing unit of the neural network - neuron, we could define more complex architectures of neurons that help to solve more complex, non-linear problems. In this thesis we mainly focus on so called **feedforward** neural networks [38], which almost exclusively refer to the topology of multiple layers of neurons and weighted connections only between neurons of two consecutive layers (no skipping allowed). The first layer is referred to as the *input layer*, followed by an arbitrary number of *hidden layers* and the last layer is called the *output layer*. General architecture of the feedforward neural network is shown in Figure 4.3.

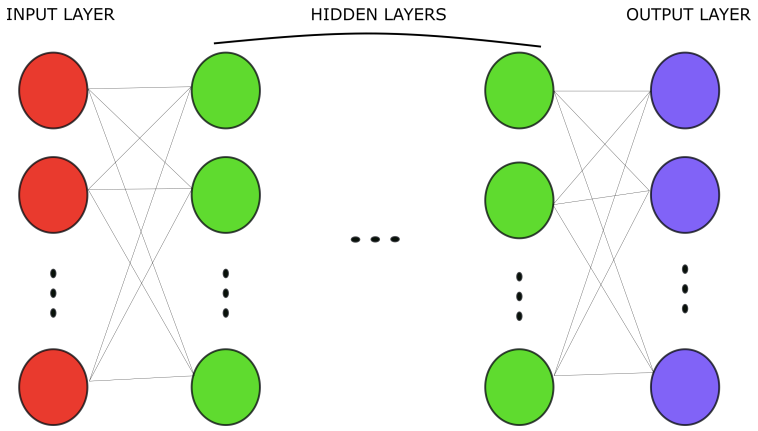


Figure 4.3: The figure is an illustration of a general architecture of feedforward neural network. Connections are only between neurons of two consecutive layers. The input (each neuron usually receives the value of one feature - neuron $j \in \mathbb{N}$ of the input layer accepts value of j-th feature x_j of input instance $\boldsymbol{x}$) of the neural network is inserted into the input layer and the output is consequently extracted from the output layer.

The universal approximation theorem proved by G. Cybenko [39] states that under a certain presumption on the activation functions of the neuron (e.g. with the usage of the sigmoid function, to be described later herein), this neural network structure could approximate any continuous function on compact subsets of $\mathbb{R}^n$. Later studies also approves that multilayered (even with only one hidden layer) feedforward neural networks could be considered to be universal approximators.

When speaking about feedforward neural networks, we usually mean full-connected system, which means, that every single neuron in arbitrary layer l is connected to all neurons in layer $l+1$. If it is necessary to turn down the neurons output to 0 (delete the connection), we force the connection weight to 0 and therefore suppress the output of the neuron.

There are other possible architectures of neural networks, which are not examined in this thesis, but which are certainly worth mentioning.

- **Shortcut connections architecture**: This feedforward setting allows connections to skip one or more levels. These skipped connections have to be directed towards the output layer (no backwards skipping). The application of this architecture was successfully exploited for example in residual networks (ResNet) [40].

- **Direct recurrence networks**: Some neurons in networks could influence themselves. The simple realization is to connect a single neuron with itself, which may therefore adaptively strenghten or weaken its activation value.
- **Indirect recurrence networks** If connections are permitted backwards (to the foregoing layers), we talk about indirect recurrence [36].

Insufficiency of perceptron

When working with neural networks, the usual procedure is to adapt all parameters of neurons (weights and biases) to force the neural network to behave in the way we require. This procedure is done by employing a predefined learning strategy (this will be described later in the thesis), which defines the manner in which neural networks learn on the data and then act independently.

Consider for a moment a feedforward neural network consisting of perceptrons with τ as the activation function and consider an arbitrary classification problem (for example classification of digits 0-9). If the neural network encounters a training instance, it adapts the weights and biases in way that the instance is then correctly classified. But even in the case of a small change in the weights and biases, the output from τ function could flip vigorously to the opposite value $(1 - \text{previous value})$, so to a completely different type of activity of the neuron. This jump may cause the previous instances to be totally misclassified. So the discontinuity property of threshold function τ makes it very difficult for neural network to learn correct parameters, especially when solving more complicated, multiclass problems.

To enable better learning ability of neural networks, it is desirable to ensure that small change in the weights or biases causes only a small change in the output of the neuron [38]. This is the property that is not met by τ function. It is the key to employ other types of activation functions that meet this property in order to establish effective learning procedures.

4.4 Activation functions

This section provides an overview and descriptions of other applied activation functions.

Sigmoid function

One way of solving a discontinuity problem of perceptron is to replace τ with its smooth approximation - the sigmoid function. It is also called the logistic function. The shape of this function can be seen in Figure 4.4.

For real input $z \in \mathbb{R}$ the sigmoid function σ is defined as

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad . \tag{4.3}$$

This function is bounded, nonlinear, differentiable and has positive derivatives. It is successfully applied especially in the output layers of deep neural networks (output lies within (0,1) range, so in the case of classification it could be thought as the probability of the instance acquiring a particular label) or in the shallow neural networks (with 1 or max. 2 hidden layers) [41]. Despite some of its favourable properties, it suffers from significant drawbacks related to popular training methods. These drawbacks will be mentioned later once the gradient descent and backpropagation algorithm for training have been described.

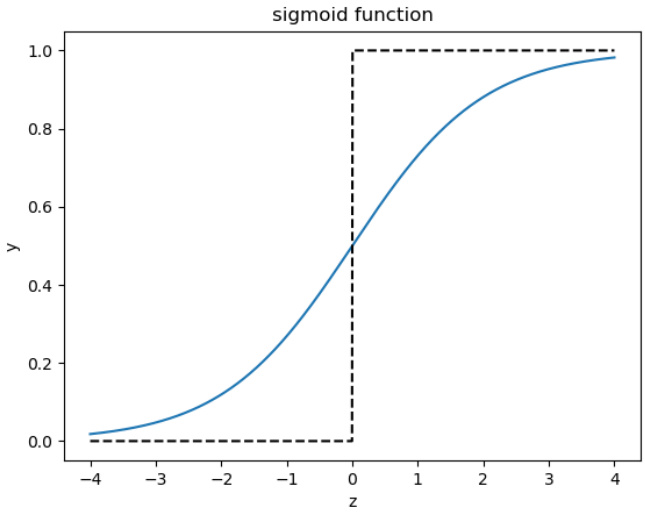


Figure 4.4: Sigmoid function approximates threshold function τ.

Hyperbolic tangent

Hyperbolic tangent (tanh) is another alternative used as the activation function. The shape of the tanh is depicted in Figure 4.5.

For real input $z \in \mathbb{R}$ is defined as

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad . \tag{4.4}$$

Hyperbolic tangent has very similar properties as the sigmoid function, but inherents some more beneficial properties, e.g. its range lies within (-1,1) and therefore is zero-centered, which produces advantegous behaviour in backpropagation training. Hyperbolic tangent has also become a preferred choice over the sigmoid function in the case of multilayer neural networks due to the fact that it shows usually a better training performance [42].

On the other hand, it shares a few drawbacks common with the sigmoid function, e.g. the vanishing gradient problem, which will be mentioned and described later in the thesis.

Softmax

The last of the exponential-based activation functions mentioned in this section is called the softmax activation function $\boldsymbol{\theta} : \mathbb{R}^k \to \mathbb{R}^k$. It is defined for the input vector $\boldsymbol{a} \in \mathbb{R}^k$ and $j \in \{1, ..., k\}$, where $k \in \mathbb{N}$ as

$$[\boldsymbol{\theta}(\boldsymbol{a})]_j = \frac{e^{a_j}}{\sum_{i=1}^{k} e^{a_i}} \quad . \tag{4.5}$$

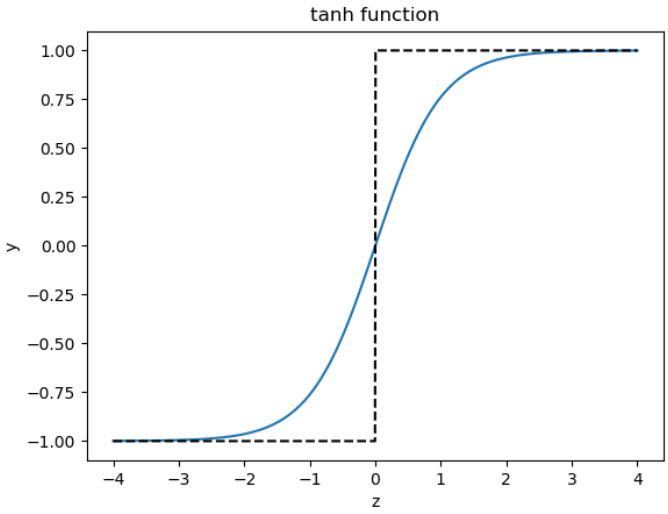


Figure 4.5: Hyperbolic tangent function. It has similar shape to the sigmoid function, but its range lies within (-1,1).

The softmax activation function is frequently used especially in the output layers of neural networks, because its definition ensures that the output vector of softmax is interpretable as probability distribution (output components can be added to 1 and the range of softmax is (0,1)).

Rectified Linear Unit (ReLU)

The ReLU activation function was introduced in 2010 by Nair and Hinto [43] and since then it has belonged to the most popular activation functions worldwide with state-of-the-art results. It is the most widely used activation function in the field of deep learning. The ReLU function is visualized in Figure 4.6

It is defined for real input $z \in \mathbb{R}$ as

$$\mathrm{ReLU}(z) = \max\{0, z\} \quad . \tag{4.6}$$

For positive values of z it behaves as a simple linear function and for negative values the output value remains 0. Its similarity to simple linear functions makes it easy to optimize learning algorithms and therefore the training of ReLU neurons is more effective and quicker (it does not need to count exponentials and divisions). The generalization ability, speed of convergence and often better performance in comparison with the sigmoid and tanh functions cause that ReLU functions are much more often used mainly in the hidden layers than competitive exponentially-based activation functions [41]. In addition, the ReLU function lacks the vanishing gradient drawback, which will be discussed later.

The ReLU function also suffers from drawbacks, such as easy inclination to overfitting in comparison with the sigmoid function and production of the so called "dead" neurons. It is inflicted when the argument of ReLU is negative and ReLU outputs 0. With regard to the fact that the

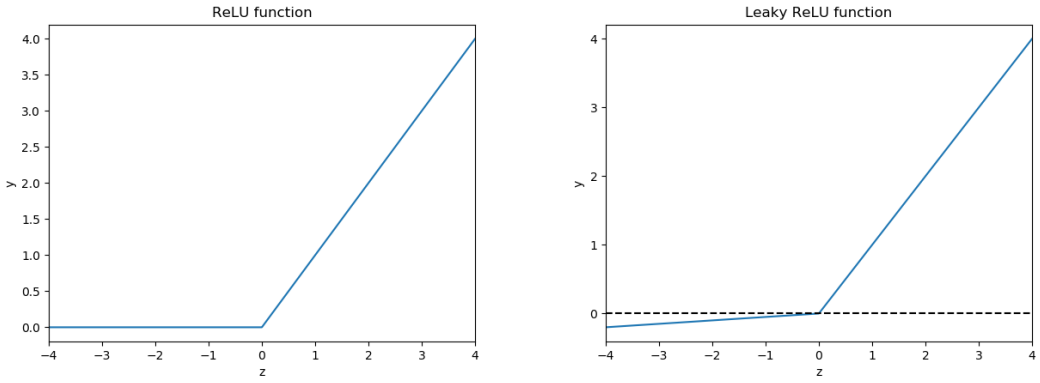


Figure 4.6: The left part of the Figure shows the graph of the ReLU function. The right-hand part of the figure shows the graph of the Leaky ReLU for $\alpha = 0.05$.

slope of ReLU in the negative part is also 0, it is probable that the output of this neuron will remain on 0 and will not return back. This neuron has no further effect in determining the output and is useless. This issue will be covered after the description of gradient-based learning methods.

The problem of dead neurons could be solved by the adoption of the Leaky ReLU activation function [44], which softens the strict condition of forcing the output to 0 when the input is negative and replaces constant 0 value in the negative part of the horizontal axis with the linear function of small slope parameter $\alpha \in \mathbb{R}$. Therefore, the gradient will never be 0. It is defined for real input $z \in \mathbb{R}$ as

$$\text{LeakyReLU}(z) = \begin{cases} z & z > 0 \\ \alpha z & \text{otherwise} \end{cases} . \tag{4.7}$$

The values of parameter α should be small, a popular choice is $\alpha = 0.01$. The shape of the Leaky ReLU can be seen in the right-hand part of Figure 4.6.

4.5 Training methods

We have already sufficiently defined the neural network architecture and the way it works by transferring activation values between neurons. What remains is to describe the algorithms which enable our neural network to learn and adapt to new problems.

Let us define some notation in order to describe this issue more clearly. The notation is inspired by [38]. We denote the correct (desired) output of neural network as $\boldsymbol{y}(\boldsymbol{x})$ for input instance $\boldsymbol{x}$ and real output (estimation) of the neural network as $\hat{\boldsymbol{y}}(\boldsymbol{x})$. The quality of performance is evaluated by the means of minimizing the so called *loss function* (in literature it can also be found as the objective function or the cost function). We denote the general loss function as Λ. One typical example of the loss function is *Mean Squared Error* (MSE), also known as the quadratic cost function, which is defined as

$$\Lambda(\mathbb{W}, \mathbb{B}, \mathbb{X}) = \frac{1}{2n} \sum_{i=1}^{n} ||\boldsymbol{y}(\boldsymbol{x}_i) - \hat{\boldsymbol{y}}(\boldsymbol{x}_i)||^2 \ , \tag{4.8}$$

where $\mathbb{W}$ is a set of all weights, $\mathbb{B}$ is a set of all biases and $\mathbb{X} = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_n\}$ is a set of all training instances, where $n \in \mathbb{N}$ is a number of all training instances. Norm $|| \cdot || = || \cdot ||_2$ is Euclidean norm. It is evident that estimation $\hat{\boldsymbol{y}}(\boldsymbol{x})$ depends on $\boldsymbol{x}$ as well as on all weights and biases in the neural network. Later we will mention other frequently used loss functions with better functionality, especially in the task of classification.

Now we are going to describe the most popular algorithm used for finding optimal weights and biases, which optimizes the general loss function Λ - *gradient descent*.

4.5.1 Gradient descent

Gradient descent algorithm is a popular nonlinear programming technique for a wide range of minimization tasks. It is considered a benchmark for the minimization of loss functions in the field of neural networks. Let us consider general loss function $\Lambda(\boldsymbol{v})$ depending on vector $\boldsymbol{v}$. Gradient descent constructs the sequence of $\boldsymbol{v}^{(k)}$, which for each $k \in \mathbb{N}$ satisfies the condition $\Lambda(\boldsymbol{v}^{(k+1)}) < \Lambda(\boldsymbol{v}^{(k)})$ (except when $\boldsymbol{v}^{(k)}$ is optimal). The sequence is constructed as

$$\boldsymbol{v}^{(k+1)} = \boldsymbol{v}^{(k)} + \eta \Delta \boldsymbol{d}^{(k)} \ , \tag{4.9}$$

where $\Delta \boldsymbol{d}^{(k)}$ is a real vector called the step or search direction, with the same dimension as $\boldsymbol{v}^{(k)}$ and $\eta > 0$ is called a learning rate [45]. The question is how to choose $\Delta \boldsymbol{d}^{(k)}$ in order to fulfill the condition $\Lambda(\boldsymbol{v}^{(k+1)}) < \Lambda(\boldsymbol{v}^{(k)})$.

We must choose $\Delta \boldsymbol{d}^{(k)}$ in such a way the Λ function decreases in that direction. We could apply the calculus and search the direction in which the following condition is met:

$$\nabla_{\Delta \boldsymbol{d}^{(k)}} \Lambda(\boldsymbol{v}^{(k)}) = \Delta \boldsymbol{d}^{(k)} \cdot \nabla \Lambda(\boldsymbol{v}^{(k)}) < 0 \ , \tag{4.10}$$

where $\nabla \Lambda(\boldsymbol{v}^{(k)})$ is the gradient of function Λ and is defined for vector $\boldsymbol{v} = (v_1, ..., v_m)^T$, $m \in \mathbb{N}$ as

$$\nabla \Lambda(\boldsymbol{v}) = (\frac{\partial \Lambda}{\partial v_1}(\boldsymbol{v}), ..., \frac{\partial \Lambda}{\partial v_m}(\boldsymbol{v}))^T \ . \tag{4.11}$$

Condition (4.10) expresses derivative of $\Lambda(\boldsymbol{v}^{(k)})$ in the direction of $\Delta \boldsymbol{d}^{(k)}$. If this derivative is negative, we can be sure that Λ decreases in that direction.

Suppose choosing $\Delta \boldsymbol{d}^{(k)} = -\nabla \Lambda(\boldsymbol{v}^{(k)})$. Then

$$\Delta \boldsymbol{d}^{(k)} \cdot \nabla \Lambda(\boldsymbol{v}^{(k)}) = -||\nabla \Lambda(\boldsymbol{v}^{(k)})||^2 \leq 0 \ . \tag{4.12}$$

Moreover, equality in (4.12) is held only if $\nabla \Lambda(\boldsymbol{v}^{(k)}) = \boldsymbol{0}$ and therefore no further update is performed, so the algorithm could be terminated. In practice, the euclidean norm of gradient is compared to the small positive threshold ϵ and the algorithm is terminated when $||\nabla \Lambda(\boldsymbol{v}^{(k)})|| \leq \epsilon$ [45].

The final equation for constructing the sequence is therefore

$$\boldsymbol{v}^{(k+1)} = \boldsymbol{v}^{(k)} - \eta \nabla \Lambda(\boldsymbol{v}^{(k)}) \ . \tag{4.13}$$

To provide a more intuitive perspective, the gradient descent algorithm iteratively searches for directions in which the value of the loss function is decreasing and stops when it is sufficiently close to the optimum (it can be local or global minimum). The initial (starting) point of the algorithm

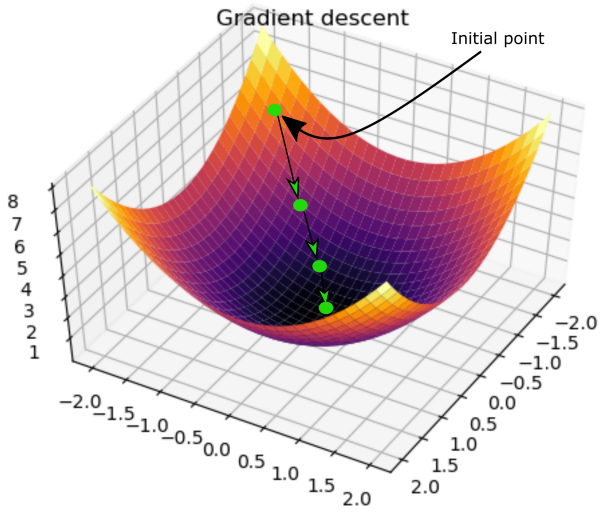


Figure 4.7: Gradient descent algorithm. In each iteration it finds the direction in which the loss function decreases. In this case, the green point is "rolling down" towards the global minimum.

needs to be selected in advance. It is usually chosen randomly. The procedure is illustrated in Figure 4.7.

Of course, the gradient descent is not a universal method which always finds the global minimum of the arbitrary function. There are some requirements on the loss function to ensure the functionality. Especially when the loss function is not "nice" like the one in Figure 4.7, it can get stuck in the local minimum instead or even do not converge, which also depends on the initial point of algorithm. The theory behind the convergence to the global optimum is properly described almost exclusively for (strictly) convex functions [45].

Another factor that significantly affects convergence is the learning rate η. If it is too small, the time until convergence could be very long and if it is too big, it could bounce back and forth around the optimal value and never reach it. Deriving the most suitable learning rate for the particular application usually requires a bit of experimenting around. There are methods to set the learning rate or adjust it during the execution. Some of them have been used in the experimental part of this thesis and will be mentioned later.

We should specify the update rule in the case of modifying weights and biases of the neural network by gradient descent. Update rules for arbitrary weight w, bias b and k-th iteration of gradient descent is

$$w^{(k+1)} = w^{(k)} - \eta \frac{\partial \Lambda}{\partial w}(\mathbb{W}^{(k)}, \mathbb{B}^{(k)}, \mathbb{X}) \tag{4.14}$$

$$b^{(k+1)} = b^{(k)} - \eta \frac{\partial \Lambda}{\partial b}(\mathbb{W}^{(k)}, \mathbb{B}^{(k)}, \mathbb{X}) \quad . \tag{4.15}$$

4.5.2 Stochastic gradient descent (SGD)

In order to accelerate learning of the neural network, the *stochastic gradient descent* algorithm was developed [38]. It is based on estimating the gradient of the loss function instead of computing it precisely. For instance, we show the algorithm in the case of the quadratic cost function in (4.8). It could be rewritten as

$$\Lambda(\mathbb{W}, \mathbb{B}, \mathbb{X}) = \frac{1}{n} \sum_{i=1}^{n} \Lambda_{\boldsymbol{x}_i} = \frac{1}{n} \sum_{i=1}^{n} \frac{||\boldsymbol{y}(\boldsymbol{x}_i) - \hat{\boldsymbol{y}}(\boldsymbol{x}_i)||^2}{2} \tag{4.16}$$

$$\Lambda_{\boldsymbol{x}_i} = \frac{||\boldsymbol{y}(\boldsymbol{x}_i) - \hat{\boldsymbol{y}}(\boldsymbol{x}_i)||^2}{2} \quad . \tag{4.17}$$

Therefore, the gradient of the loss function could be written as (we omit the parameters of Λ for simplification)

$$\nabla\Lambda = \frac{1}{n} \sum_{i=1}^{n} \nabla\Lambda_{\boldsymbol{x}_i} \quad . \tag{4.18}$$

Basically, this term is only the average of gradients of $\Lambda_{\boldsymbol{x}_i}$ across all training instances. The idea of the stochastic gradient descent is to replace averaging across all training instances with averaging only across a randomly sampled mini-batch $x_{i_1}, ..., x_{i_m}$ of size $m < n$, $m \in \mathbb{N}$. We obtain estimate

$$\nabla\Lambda = \frac{1}{n} \sum_{i=1}^{n} \nabla\Lambda_{\boldsymbol{x}_i} \approx \frac{1}{m} \sum_{l=1}^{m} \nabla\Lambda_{\boldsymbol{x}_{i_l}} \quad . \tag{4.19}$$

If the size of the mini-batch is large enough, we can expect that this approximation will be roughly equal to the real gradient. This algorithm works for loss functions that could be rewritten in the same manner as quadratic cost function from the example. Of course, there might be some statistical deviations and the estimation is not always precise, but what we need is to move in the direction where the loss function is expected to decrease, it does not need to be exactly the direction of the gradient, so working with an estimation is justified. We shall rewrite the update rule in the case of SGD as

$$w^{(k+1)} = w^{(k)} - \frac{\eta}{m} \sum_{l=1}^{m} \frac{\partial \Lambda_{\boldsymbol{x}_{i_l}}}{\partial w}(\mathbb{W}^{(k)}, \mathbb{B}^{(k)}, \mathbb{X}) \tag{4.20}$$

$$b^{(k+1)} = b^{(k)} - \frac{\eta}{m} \sum_{l=1}^{m} \frac{\partial \Lambda_{\boldsymbol{x}_{i_l}}}{\partial b}(\mathbb{W}^{(k)}, \mathbb{B}^{(k)}, \mathbb{X}) \quad . \tag{4.21}$$

The algorithm in practice usually runs in several epochs. Each epoch is characterized by random sampling of mini-batches of given size m and then by training with those mini-batches. When one epoch is finished, training instances are randomly mixed and the mini-batches are sampled again and the training continues in a new epoch with new mini-batches.

In (4.20) and (4.21) $\frac{1}{m}$ factor can be omitted, which corresponds only to the scaling of the learning rate. Factor $\frac{1}{n}$ is sometimes omitted also in the definition of MSE in (4.8) [38].

4.5.3 Other training methods

Altough the gradient descent or SGD are both popular optimization algorithms applied in many optimization tasks (especially in the training phase of neural networks), they are not entirely flawless. In some situations they could report drawbacks which can make the learning procedure much more difficult. Especially in the means of the speed of convergence and stucking in suboptimal (local) minima. Moreover, the overall sensitivity to the learning rate parameter is in the case of the gradient descent and SGD very high, which also affects the convergence and the poor choice of the learning rate can even lead to divergence.

To address these issues, several alternatives to the SGD have been proposed [46]. They target to accelerate the convergence and reduce oscillations. Some of them also adapt the learning rate parameter at the runtime and thus possess better ability of controlling the direction and amount of change.

SGD with momentum

This method helps to accelerate the convergence of classic SGD and reduce fluctuations around the optimal value, which is common for SGD for instance in the valley areas with only slight slope of decrease at the bottom and steep slopes on sides (basically different slopes in different dimensions). The example is shown in Figure 4.8.

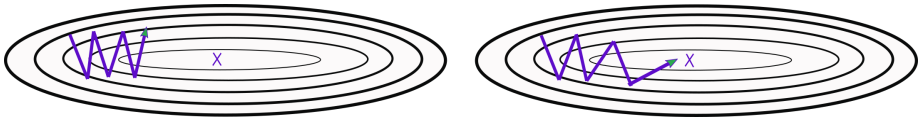


Figure 4.8: The left figure shows a situation where the classic SGD algorithm is applied. The right-hand figure represents the case of the SGD with momentum.

The SGD with momentum works in the way that it gradually accumulates fractions of previous update vectors and uses it for a new update. The update rule is defined with the use of momentum parameter $\gamma, 0 \leq \gamma \leq 1$, which represents the fraction of the previous update vector, and velocity vector at k-th iteration $\boldsymbol{s}^{(k)}$, which represents the update vector at k-th iteration ($\boldsymbol{s}^{(0)} = \mathbf{0}$). For arbitrary loss function $\Lambda = \Lambda(\boldsymbol{v})$ (depending on vector $\boldsymbol{v}$, which could represent for instance weights and biases) and learning rate $\eta > 0$, the SGD with momentum is defined as

$$\boldsymbol{s}^{(k+1)} = \gamma \boldsymbol{s}^{(k)} + \eta \nabla \Lambda(\boldsymbol{v}^{(k)}) \tag{4.22}$$

$$\boldsymbol{v}^{(k+1)} = \boldsymbol{v}^{(k)} - \boldsymbol{s}^{(k+1)} \quad . \tag{4.23}$$

This mechanism could also be interpreted in the physical point of view. Imagine ball rolling down the hill. It gradually accelerates and accumulates momentum, approaching the bottom faster and faster. It is afterwards more difficult to change suddenly its direction of movement by some lateral impulses and conversely the impulses in the direction of movement can increase the speed rapidly. Same reasoning could be applied to our case of optimization, where the effect of gradients that are deviated from the direction of movement is suppressed. This ensures faster convergence and smaller fluctuations.

The magnitude of momentum parameter γ controls the level of considering previous update vectors. The higher it is, the stronger influence of accumulated momentum is considered and the harder it is for new gradients to change its direction.

Even though this algorithm possesses these improvements over the SGD, it sometimes acts too thoughtlessly. It is intuitively apparent that in some situations it may be beneficial to slow down and change the direction significantly, not only follow the direction of accumulated gradients. This is the reason why the **Nesterov algorithm** has been developed. It is a smarter version of the SGD with momentum, which roughly anticipates the future position and exploits the gradient computed in the estimated future point instead of the gradient in the current point. We can roughly estimate future points as $\hat{\boldsymbol{v}}^{(\boldsymbol{k+1})} = \boldsymbol{v}^{(k)} - \gamma \boldsymbol{s}^{(k)}$, thus the Nesterov algorithm computes new update vector $\boldsymbol{s}^{(k+1)}$ with usage of the gradient evaluated in this estimated point. The equations of the SGD with momentum are modified to

$$\boldsymbol{s}^{(k+1)} = \gamma \boldsymbol{s}^{(k)} + \eta \nabla \Lambda(\boldsymbol{v}^{(k)} - \gamma \boldsymbol{s}^{(k)}) \tag{4.24}$$

$$\boldsymbol{v}^{(k+1)} = \boldsymbol{v}^{(k)} - \boldsymbol{s}^{(k+1)} \quad . \tag{4.25}$$

Adam

Adam (Adaptive Moment Estimation) belongs to the class of optimization algorithms that adapts the learning rate for each parameter during runtime. Currently Adam is considered a member of state-of-the-art optimization methods with a wide scope of application in the context of neural networks.

The theory behind Adam is slightly complicated and not intuitive as in the case of previous methods. We only provide a short introduction and a detailed description and analysis can be found in [47].

Adam at k-th iteration of algorithm updates exponential moving average of gradient $\boldsymbol{m}^{(k)}$, which estimates the first moment (the mean) of the gradient, and exponential moving average of squared gradient $\boldsymbol{r}^{(k)}$, which estimates the second raw moment (the uncentered variance) of the gradient. It turned out that these terms are biased towards zero, so afterwards they are bias-corrected and applied to update the parameters. The algorithm is parameterized by decay rates $0 \leq \gamma_1 < 1$ and $0 \leq \gamma_2 < 1$. The lower they are, the more the moving average is shifted towards the current values. There is also a special term $\epsilon > 0$, which helps to avoid division by zero. For simplification of the notation we denote the gradient of loss function $\Lambda = \Lambda(\boldsymbol{v})$ with respect to $\boldsymbol{v}$ at k-th iteration as $\boldsymbol{g}^{(k)}$ ($\boldsymbol{v}$ generally symbolizes any vector of parameters in the current scope of interest). Finally, for learning rate η, the complete algorithm could be written as (all vector operations are element-wise):

$$\boldsymbol{m}^{(k+1)} = \gamma_1 \boldsymbol{m}^{(k)} + (1-\gamma_1)\boldsymbol{g}^{(k)} \tag{4.26}$$

$$\boldsymbol{r}^{(k+1)} = \gamma_2 \boldsymbol{r}^{(k)} + (1-\gamma_2)[\boldsymbol{g}^{(k)}]^2 \tag{4.27}$$

$$\hat{\boldsymbol{m}}^{(k+1)} = \frac{\boldsymbol{m}^{(k+1)}}{1-\gamma_1^{k+1}} \tag{4.28}$$

$$\hat{\boldsymbol{r}}^{(k+1)} = \frac{\boldsymbol{r}^{(k+1)}}{1-\gamma_2^{k+1}} \tag{4.29}$$

$$\boldsymbol{v}^{(k+1)} = \boldsymbol{v}^{(k)} - \frac{\eta}{\sqrt{\hat{\boldsymbol{r}}^{(k+1)}} + \epsilon}\hat{\boldsymbol{m}}^{(k+1)} \quad . \tag{4.30}$$

In the (4.26) respectively (4.27), the exponential moving average of gradient respectively squared gradient is updated. Then both terms are bias-corrected in (4.28) and (4.29) and the final update of parameters is performed in (4.30). Usually both $\boldsymbol{m}^{(0)} = \boldsymbol{0}$ and $\boldsymbol{r}^{(0)} = 0$. A good default choice of parameters suggested by [47] is $\gamma_1 = 0.9$, $\gamma_2 = 0.999$ and $\epsilon = 10^{-8}$.

4.6 Backpropagation algorithm

We have already presented the idea of training the neural network via different optimization methods, but we have not provided the way of computing partial derivatives (gradients) of the loss function with respect to weights and biases, which are necessary to execute the training algorithm. This could of course be done by computing the derivatives analytically, which is very uncomfortable and almost insane idea, especially when adopting large neural networks. For this reason, the *backpropagation algorithm* was developed in order to retrieve partial derivatives with respect to all weights and biases involved in the neural network in a comfortable and quick way.

The derivation and description of the backpropagation algorithm is motivated by papers [48] and [38]. We define some helpful notation in advance in order to simplify the derivation. Let us denote the output from l-th layer of the neural network and j-th neuron of that layer as a_j^l, the entire output vector of l-th layer will be denoted as $\boldsymbol{a}^l$. Term w_{jk}^l denotes weight of the connection going from neuron k in the previous layer $(l-1)$ to neuron j in layer l (it is in reversed order on purpose, it will simplify further notation with respect to the matrix multiplication) and term $\mathbb{W}^l = (w_{jk}^l)_{j,k}$ denotes matrix of weights directed from layer $(l-1)$ to l. The bias of node j in layer l is denoted as b_j^l and biases folded into vector of biases of layer l is $\boldsymbol{b}^l$. By applying this notation, we could write comfortably a feedforward dependency for the output from layer l as

$$\boldsymbol{a}^l = \phi(\mathbb{W}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l) \quad , \tag{4.31}$$

where ϕ is the arbitrary activation function used in neurons of layer l (this definition apparently depends on the current layer l, but we omit notation of this dependence for simplification), which is applied in element-wise fashion. We also denote argument of ϕ in (4.31) as

$$\boldsymbol{z}^l = \mathbb{W}^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l \quad . \tag{4.32}$$

The aim of the backpropagation algorithm is to compute $\frac{\partial \Lambda}{\partial w}$ and $\frac{\partial \Lambda}{\partial b}$ (we will omit arguments of the functions for simplification, all of them are evaluated in the values of current weights, biases and

training instances) with respect to arbitrary weight w and bias b. These partial derivatives express, how much the loss function changes when w or b is changed. Of course, the main goal is to find optimal weights and biases, which corresponds to the local (global) minimum of the loss function. Intuitively, we could imagine that in each layer of the neural network a certain error arises (from imperfection of current weights and biases), which causes flaws in the output layer. Furthermore, we could say that even individual neurons produce these errors, which consequently leads to a corrupted output. But what we can measure is only the deviation of the current output (extracted from the output layer) of the neural network from the desired value, because we do not know in advance the "correct" output (activation values) of other layers. Basically, what backpropagation really does is that it *backpropagates* an error from the output layer to the previous layers and then from these error terms computes corresponding partial derivatives $\frac{\partial \Lambda}{\partial w}$ and $\frac{\partial \Lambda}{\partial b}$.

We will now define this error term heuristically. The output of j-th neuron of layer l could be written as $a_j^l = \phi(z_j^l)$. Let's imagine for a moment that layer l has a corrupted output (not corresponding to the optimal value) denoted as $\hat{a}_j^l$. This corrupted value could be written as $\hat{a}_j^l = \phi(\hat{z}_j^l) = \phi(z_j^l + \Delta z_j^l)$, where Δz_j^l expresses the deviation of argument z_j^l from the optimal value. In case that other weights and biases are already optimal, the difference between the optimal cost and the "corrupted" cost would be approximately $\frac{\partial \Lambda}{\partial z_j^l} \Delta z_j^l$ (from Taylor expansion, we suppose that Δz_j^l is "small enough"). The deviation is getting larger, when value of $\frac{\partial \Lambda}{\partial z_j^l}$ is getting larger. On the other hand, when the value of $\frac{\partial \Lambda}{\partial z_j^l}$ is close to 0, then the neuron is very close to the optimal state. Therefore, we could define the error term as

$$\delta_j^l = \frac{\partial \Lambda}{\partial z_j^l} \ . \tag{4.33}$$

This is a heuristic derivation and of course, it is not absolutely precise and certainly requires deeper math behind (e.g. we omitted trade-off between $\frac{\partial \Lambda}{\partial z_j^l}$ and Δz_j^l in estimating the deviation), but this derivation serves mainly for the description of intuitive behaviour of δ_j^l. What is more important is the fact that δ_j^l is only a provisional term, which will lead us to the derivation of values of real interest - $\frac{\partial \Lambda}{\partial w}$ and $\frac{\partial \Lambda}{\partial b}$.

The algorithm always backpropagates the error of single training instance $\boldsymbol{x}$ (we can use only one training instance at a time), so what it really computes is $\frac{\partial \Lambda_{\boldsymbol{x}}}{\partial w}$ and $\frac{\partial \Lambda_{\boldsymbol{x}}}{\partial b}$. Therefore, the loss function Λ needs to be rewritten in the same manner as in (4.18), which we assume. From now on, we will omit the subscript and denote $\Lambda_{\boldsymbol{x}}$ simply as Λ. Another assumption on the loss function is that it can be expressed as the function of output $\boldsymbol{a}^L$ from the output layer, where L denotes the output layer. We need $\Lambda = \Lambda(\boldsymbol{a}^L)$, but it may also depend on other variables. This assumption will be explained later. For instance, both assumptions are met by the quadratic cost function and also by other loss functions presented later in this thesis.

Since we have defined δ_j^l, we now need to find equations for values of real interest, ig. partial derivatives of the loss function with respect to weights and biases. The entire backpropagation algorithm stands on 4 equations, which allow us to compute values of real interest from errors in individual layers of the neural network. They can be simply derived by careful application of the chain rule from multivariate calculus. Note that all of these equations are specifically written for arbitrary activation functions $\phi : \mathbb{R}^1 \to \mathbb{R}^1$ (tanh, sigmoid, ReLU). This corresponds to our experiments since we used such functions in the hidden layers of our networks (it is also common in practice). The only exception was the case of the output layer, where we used both sigmoid

and softmax. But softmax is a vector function and therefore the general equation (4.34) should be slightly modified as a special case for softmax (or vector functions in general). Other equations remain the same. We will mention this special shape for softmax in the commentary to the corresponding equation.

1. **Equation to calculate error $\boldsymbol{\delta}^L$ of the output layer L**:

$$\delta_j^L = \frac{\partial \Lambda}{\partial a_j^L} \phi'(z_j^L) \ . \tag{4.34}$$

As mentioned before, the backpropagation algorithm runs backwards. First, we need to compute error in the output layer. It is calculated by multiplication of partial derivative of the loss function with respect to the output a_j^L of j-th neuron of layer L and derivative of the activation function ϕ in z_j^L. This equation is perfectly intuitively understandable, because the error term is basically expressed as the trade-off between "how much the loss function changes with respect to the output from the neuron (represented by $\frac{\partial \Lambda}{\partial a_j^L}$)" and "how much the output of the neuron changes with respect to the input value to the activation function (represented by $\phi'(z_j^L)$)". If any of these values are low, then the error of that neuron is also low. Furthermore, both values could easily be retrieved and the shape of derivatives is usually computed analytically (obviously depending on the shape of the loss function and the activation function). We may also notice that the partial derivative of Λ is computed with respect to a_j^L. This is the reason why we need the assumption of Λ depending on $\boldsymbol{a}^L$. Note that this shape corresponds to the sigmoid function σ used in the output layer (or any arbitrary scalar function with the scalar input). For softmax function $\boldsymbol{\theta}$ it would be

$$\delta_j^L = \sum_{k=1}^{|\mathbb{C}|} \frac{\partial \Lambda}{\partial a_k^L} \frac{\partial [\boldsymbol{\theta}(\boldsymbol{z}^L)]_k}{\partial z_j^L} \ , \tag{4.35}$$

since $\forall k \in \{1, ..., |\mathbb{C}|\}$, the function $[\boldsymbol{\theta}(\boldsymbol{z}^L)]_k$ depends on z_j^L where $|\mathbb{C}|$ denotes the number of neurons in the output layer, which is the same as the number of classes (C denotes a set of all classes).

2. **Equation to backpropagate the error from layer $(l+1)$ to previous layer l**

$$\delta_j^l = [(\mathbb{W}^{(l+1)})^T \boldsymbol{\delta}^{(l+1)}]_j \phi'(z_j^l) \ . \tag{4.36}$$

This equation gives us a way to backpropagate the error backwards. Together with (4.34), we can compute $\boldsymbol{\delta}^l$ (vector of errors δ_j^l of neurons in layer l) in the arbitrary layer l. It is obtained by multiplying the transpose of weight matrix $\mathbb{W}^{(l+1)}$ with vector of errors of $(l+1)$ layer $\boldsymbol{\delta}^{(l+1)}$. This could be thought as a "projection" of error to the previous layer. Then the result is multiplied by $\phi'(z_j^l)$, similarly to the case (4.34), because the error is also influenced by the speed of change of the activation function.

3. **Equation to calculate $\frac{\partial \Lambda}{\partial b_j^l}$**

$$\frac{\partial \Lambda}{\partial b_j^l} = \delta_j^l \ . \tag{4.37}$$

This first value of real interest could be determined easily, because equation (4.37) states that it is equal directly to the error term. And (4.34) and (4.36) already give us tools to compute the error in the arbitrary layer.

4. **Equation to calculate $\frac{\partial \Lambda}{\partial w_{jk}^l}$**

$$\frac{\partial \Lambda}{\partial w_{jk}^l} = a_k^{(l-1)} \delta_j^l \quad . \tag{4.38}$$

This equation computes the partial derivative of the loss function with respect to any weight. It depends on the activation (output) of k-th neuron of layer $(l-1)$ and error of j-th neuron of the consecutive layer l.

By applying these 4 equations, we can compute all values of interest easily. These equations could also be in some cases fully rewritten into the vector (matrix) shape and therefore solved in a fast and efficient way (some libraries often effectively compute and manipulate with matrix-organized data, such as Numpy[1]). More information on the derivation of these equations can be found in [38].

Problems related to activation functions

In this part we shortly summarize and describe some properties of different activation functions related to the backpropagation that influence the entire training procedure.

First we are going to focus on so called *vanishing gradient problem*. This problem occurs especially in the case of multi-layer neural networks. Consider equations (4.34) and (4.36) mentioned in the previous part. These equations include the derivatives of activation functions in the corresponding layers. If the error backpropagates recursively from the output layer to previous layers as in the (4.36), then more and more derivative terms are multiplied together. If some of them are close to zero, than the final term is supposed to be low as well. The gradients of the loss function are computed with the usage of the error term (as suggested in (4.37) and (4.38)) and therefore the update of parameters could be insignificant in early layers due to small gradients. This is often the case of sigmoid activation function. The derivative of sigmoid is $\sigma'(z) = \sigma(z)(1-\sigma(z))$ [49] and therefore it reaches the maximum value of 0.25. Multiplication of multiple sigmoid derivatives could result in a significant reduction in gradients in early layers of the neural network due to chain interaction of individual derivatives.

The situation is slightly better in the case of the tanh activation function, whose derivative lies within 0 and 1, but still the vanishing gradient problem occurs, because values are often less than 1. Especially when the sigmoid or tanh output values close to the asymptotic tails, the derivative is close to 0 and therefore there will probably be only a small update of weight and bias which hardly change the current behaviour of a neuron. This state is called *saturation*. Vanishing gradient and saturation make the training procedure more difficult and could negatively affect the quality of performance.

The problem of the vanishing gradient could be considerably suppressed by the adoption of the (Leaky) ReLU activation function. In the case of the ReLU, the derivative is always either 1 or 0. When the ReLU operates in the positive area, the derivatives are always 1 and therefore no vanishing gradient occurs. A problem arises when the ReLU moves to the negative area, where the derivative is 0. This produces so called *dead neurons*, because the derivative is 0 and the error is not backpropagated from this neuron. Aditionally, it is not likely to recover from this state as

[1] Available at https://numpy.org/

the training proceeds. This is the reason why the Leaky ReLU was introduced as it has not-null derivative in the negative area and therefore allows the neuron to recover. This prevents producing dead neurons.

4.7 Loss functions

In this section we summarize popular loss functions among neural network context and discuss their properties. We have already defined the quadratic cost function (MSE) in (4.8). It is widely-applied in the machine learning area, but it has some drawbacks related to the backpropagation algorithm. Specifically, we refer to the *learning slowdown* problem [38, 50], which is the issue especially in the case of classification.

We demonstrate this problem on the quadratic cost function and sigmoid in the output layer. Since the quadratic cost function is a sum of functions defined in (4.18), we are justified to work only with these single functions. We are going to slightly simplify the notation and use some notation defined in the section 4.6. Let us denote the MSE single function as

$$\Lambda = \frac{||\boldsymbol{y} - \boldsymbol{a}^L||^2}{2} \ , \tag{4.39}$$

where $\boldsymbol{y}$ denotes the correct output and $\boldsymbol{a}^L$ denotes the output of the neural network (which is the same as the output from the output layer $\boldsymbol{a}^L$). Now we will express the derivatives of loss function Λ with respect to weights and biases in the output layer L with usage of backpropagation equations. First we compute the error term for node j in the output layer as in (4.34). Since $\frac{\partial \Lambda}{\partial a_j^L} = y_j - a_j^L$, the error term is

$$\delta_j^L = (y_j - a_j^L)\phi'(z_j^L) \ . \tag{4.40}$$

Now we substitute this term in the equations (4.37) and (4.38).

$$\frac{\partial \Lambda}{\partial b_j^L} = (y_j - a_j^L)\phi'(z_j^L) \tag{4.41}$$

$$\frac{\partial \Lambda}{\partial w_{jk}^L} = a_k^{(L-1)}(y_j - a_j^L)\phi'(z_j^L) \ . \tag{4.42}$$

In these equations occur derivatives of the activation function (sigmoid in this example) in the output layer. Consequently, this causes the learning slowdown problem. If the values of sigmoid are close to 0 or 1 (asymptotic tails), the derivative is very close to 0 and therefore it produces very low updates, since the corresponding partial derivatives are low. Switching the values in the output layer, which are close to asymptotic tails of sigmoid (if this is incorrect state), is very slow, hence the learning slowdown.

It would be similar in the case of MSE combined with softmax. It is easy to derive that derivative of softmax fulfills the equation [49]

$$\frac{\partial [[\boldsymbol{\theta}(\boldsymbol{z}^L)]_k}{\partial z_j^L} = [\boldsymbol{\theta}(\boldsymbol{z}^L)]_k(\delta_{jk} - [\boldsymbol{\theta}(\boldsymbol{z}^L)]_j) \ , \tag{4.43}$$

where δ_{jk} is Kronecker delta. Since softmax produces probability distribution, so if the value for specific k is close to 1, then other values are close to 0 and therefore $\forall k$ the derivative in (4.43) is close to 0. Thus the error term in (4.35) would also be low.

To fix this drawback, the Cross Entropy loss function can be exploited, which results in the absence of the learning slowdown problem.

Cross Entropy loss function

Cross Entropy loss function is often a preferred choice among loss functions in the case of classification. It is frequently applied in combination with softmax or sigmoid in the output layer. It is defined as

$$\Lambda^{CE}(\mathbb{W}, \mathbb{B}, \mathbb{X}) = -\frac{1}{n} \sum_{\boldsymbol{x} \in \mathbb{X}} \sum_{i=1}^{|\mathbb{C}|} y_i(\boldsymbol{x}) \ln \hat{y}_i(\boldsymbol{x}) \ , \tag{4.44}$$

where $\mathbb{X}$ is a set of input instances, $n = |\mathbb{X}|$, $|\mathbb{C}|$ denotes number of neurons in the output layer (equal to the number of classes), $y_i(\boldsymbol{x})$ is i-th component of $\boldsymbol{y}(\boldsymbol{x})$, which denotes the correct output for single instance $\boldsymbol{x}$ and $\hat{y}_i(\boldsymbol{x})$ denotes the i-th component of the prediction of neural network $\hat{\boldsymbol{y}}(\boldsymbol{x})$ for single instance $\boldsymbol{x}$. We also explicitly mark the dependence on a set of weights $\mathbb{W}$ and a set of biases $\mathbb{B}$. The definition in (4.44) acts as an overall loss across multiple instances (averaging similarly as in the case of the quadratic cost function). Cross Entropy of single instance $\boldsymbol{x}$ is therefore

$$\Lambda^{CE}(\mathbb{W}, \mathbb{B}, \boldsymbol{x}) = -\sum_{i=1}^{|\mathbb{C}|} y_i(\boldsymbol{x}) \ln \hat{y}_i(\boldsymbol{x}) \ . \tag{4.45}$$

It measures the similarity between the correct label distribution and the predicted label distribution for $\boldsymbol{x}$. Since there is only one $i \in \{1, ..., |\mathbb{C}|\}$ (corresponding to the ground-truth label), where $y_i(\boldsymbol{x}) = 1$ (all other components are 0), then the sum in (4.45) reduces only to one summand. Therefore the Cross Entropy of single instance $\boldsymbol{x}$ decreases if the predicted value corresponding to the ground-truth label of $\boldsymbol{x}$ increases and vice versa, which is the favourable behaviour in the task of classification.

Now we will demonstrate that combination of the Cross Entropy loss function and softmax in the output layer prevents the learning slowdown problem. We compute the error term for softmax as in (4.35) and show that it does not depend on the derivatives of softmax. We will also use the notation from the section 4.6 and denote $a_i^L = \hat{y}_i$. The parentheses with dependencies of Λ^{CE} are omitted for simplification.

$$\begin{aligned}\delta_j^L = \frac{\partial \Lambda^{CE}}{\partial z_j^L} = \sum_{k=1}^{|\mathbb{C}|} \frac{\partial \Lambda^{CE}}{\partial a_k^L} \frac{\partial [\boldsymbol{\theta}(\boldsymbol{z}^L)]_k}{\partial z_j^L} = \sum_{k=1}^{|\mathbb{C}|} (-\frac{y_k}{[\boldsymbol{\theta}(\boldsymbol{z}^L)]_k}) [\boldsymbol{\theta}(\boldsymbol{z}^L)]_k (\delta_{kj} - [\boldsymbol{\theta}(\boldsymbol{z}^L)]_j) = \\ = \sum_{k=1}^{|\mathbb{C}|} (-y_k \delta_{kj}) + \sum_{k=1}^{|\mathbb{C}|} y_k [\boldsymbol{\theta}(\boldsymbol{z}^L)]_j = -y_j + [\boldsymbol{\theta}(\boldsymbol{z}^L)]_j = a_j^L - y_j = \hat{y}_j - y_j\end{aligned}$$

We can see that δ_j^L is fully controlled only by the difference between the predicted and the correct value and no derivative of activation function occurs in the final equation. Therefore, this approach is not affected by the learning slowdown problem as in the case of MSE.

There are also other loss functions applied in the neural network context, but none of them are more popular than MSE and Cross Entropy. Other loss functions can be found in [51].

4.8 Regularization

Regularization is a crucial term, which is almost indispensable among machine learning in general nowadays. It addresses the problem of overfitting and introduces some solutions to attain better generalization ability of machine learning models. Neural networks are able to approximate even very complicated, non-linear functions and hence are prone to overfitting quite easily. This is especially true for large networks. We are going to focus on regularization specifically in terms of neural networks and describe some popular regularization techniques.

Modification of the loss function

This technique works in such a way that it adds a special regularization term to the loss function, which usually restricts the size of weights and penalizes the loss function for large weights. A heuristic idea behind this is that weight is allowed to be large only if it has a distinctive influence on decreasing the loss function by the improvement of the output of the neural network. Thus, it is more likely that the model increases only those weights that have high general effect on estimating the output and does not increase weights to gain only minor improvement of results by learning a noise in data [52].

We distinguish the representatives of this group based on different regularization terms that are added to the loss function. Among the most popular belong L2 and L1 regularization, which are defined as follows:

$$\begin{aligned}
\textbf{L2 regularization} \ : \ & \Lambda(\mathbb{W},\mathbb{B},\mathbb{X}) = \Lambda_0(\mathbb{W},\mathbb{B},\mathbb{X}) + \frac{\lambda}{2n}\sum_{w\in\mathbb{W}}|w|^2 \\
& \frac{\partial\Lambda}{\partial w} = \frac{\partial\Lambda_0}{\partial w} + \frac{\lambda}{n}w \\
\text{Update rule} \ : \ & w^{(k+1)} = (1-\eta\frac{\lambda}{n})w^{(k)} - \eta\frac{\partial\Lambda_0}{\partial w}(\mathbb{W}^{(k)},\mathbb{B}^{(k)},\mathbb{X})
\end{aligned} \tag{4.46}$$

$$\begin{aligned}
\textbf{L1 regularization} \ : \ & \Lambda(\mathbb{W},\mathbb{B},\mathbb{X}) = \Lambda_0(\mathbb{W},\mathbb{B},\mathbb{X}) + \frac{\lambda}{n}\sum_{w\in\mathbb{W}}|w| \\
& \frac{\partial\Lambda}{\partial w} = \frac{\partial\Lambda_0}{\partial w} + \frac{\lambda}{n}\mathrm{sgn}(w) \\
\text{Update rule} \ : \ & w^{(k+1)} = w^{(k)} - \eta\frac{\lambda}{n}\mathrm{sgn}(w^{(k)}) - \eta\frac{\partial\Lambda_0}{\partial w}(\mathbb{W}^{(k)},\mathbb{B}^{(k)},\mathbb{X}) \ ,
\end{aligned} \tag{4.47}$$

where $n = |\mathbb{X}|$, Λ_0 denotes the original loss function and Λ denotes the loss function after the addition of the regularization term. The effect of the regularization term is controlled by the magnitude of regularization parameter $\lambda \geq 0$. The larger λ is, the larger penalty is put on weights and vice versa. Both variants penalize large weights in a slightly different way. The difference is perfectly visible from the individual update rules of weights for gradient descent (or SGD) in (4.46) or (4.47). We can see that in the case of L2 regularization the weights are gradually decreased proportionally to the magnitude of the corresponding weight, whereas in the case of L1

regularization the weights are modified by the value of constant magnitude. This means that L2 regularization tends to shrink larger weights more than L1. L2 regularization is sometimes referred to as a *weight decay*.

In the experimental part we adopt mainly L2 regularization as a regularization technique. It could also be used in adaptive optimization algorithms, such as Adam. Generally accepted approach is to replace gradient $\nabla\Lambda_0$ with $\nabla\Lambda$ for initializing corresponding terms (e.g. equations (4.26) and (4.27)), as we did also in the case of SGD. There has been some criticism to this approach related to the adaptive algorithms [53], which led to some proposals and improvements, but they are beyond the scope of this thesis and we will not mention them. In our experiments we use replacement of gradients as suggested above.

Early stopping

Early stopping is possibly the most natural regularization technique. It basically terminates the training phase of the neural network if the model starts to deteriorate its generalization ability. It is usually demonstrated at some epoch of training by degradation (stagnation) of performance on the validation dataset, while the performance on the training dataset is still improving. The performance is usually measured by monitoring the accuracy or loss on the validation dataset. There is a patience constant, which determines the number of epochs the algorithm waits for the improvement. If no improvement occurs, the training is terminated.

Dropout

Dropout is a popular regularization technique among neural networks, which has shown state-of-the-art results in many applications. The algorithm basically works as follows: During the training of neural network is randomly dropped half of the neurons (temporarily) in each hidden layer for each mini-batch. Then the forward pass and backward pass is performed through the subnetwork and the weights and biases of the subnetwork are updated [54]. The dropped neurons are then recovered and the same procedure is repeated for the following mini-batches as the training continues. More generally, the neurons could be dropped with probability p, but at least one neuron must stay in each hidden layer in order to conduct meaningful updates.

For a testing phase the entire network with all neurons is applied, but (in the case of dropping half of the neurons in each iteration) we need to halve all weights of connections outgoing from the hidden neurons since we operate with twice as many hidden neurons as the subnetworks were trained with.

There are not many theoretical researches that would clarify the functionality of this method, but there are plenty of heuristic explanations. The first possible explanation of its functionality is that Dropout basically "average" several different neural networks, and therefore the full network is supposed to have a higher generalization ability than single networks. The second one is that this technique reduces the influence and dependencies between individual neurons and thus the neurons are forced to learn more robust features rather than relying on the outputs of particular neurons.

We have mentioned three different regularization techniques frequently used among neural networks and machine learning in general. Other regularization techniques are for instance *dataset augmentation*, *parameter sharing*, etc. The descriptions of other methods can be found in [52]. It is also common to combine individual regularization techniques to ensure even higher generalization

ability, but they must be applied carefully since the regularizers can be more decisive in training than the data themselves.

Chapter 5

Decision tree to neural network transformation

In this section we provide a detailed derivation of the neural network architecture, which is further employed in all subsequently proposed models. The main idea is built on the article [3] describing one-to-one transformation of the arbitrary regression tree to the specifically designed neural network.

Unfortunately, the approach presented in [3] is not uniformly convertible to the classification problem. Based on this issue, alternative architecture and initial settings are proposed in order to simulate the exact behaviour of corresponding decision tree classifier.

This reformulation provides a sensible opportunity to enhance the decision tree performance, because parameters of the newly constructed neural network could be better adapted with the usage of the backpropagation algorithm of neural networks and therefore achieve superior classification.

5.1 Architecture and initial settings

The initial weights and biases of neurons and architecture of the input layer and the first two hidden layers will remain same among all proposed models. A sample of such architecture (only the input layer and first two hidden layers) is illustrated in Figure 5.2 copying the decisions of the decision tree depicted in Figure 5.1, which splits the space by two hyperplanes as illustrated also in 5.1. Let's first consider neurons in the first and second hidden layers of the network as perceptrons. In our case, the activation function is specific threshold function

$$\tau(\boldsymbol{x}) = 2\mathbf{1}_{\boldsymbol{x}\geq 0} - 1 \ , \tag{5.1}$$

where all math operations on instance $\boldsymbol{x}$ are element-wise. Also

$$(\mathbf{1}_{\boldsymbol{x}\geq 0})_i = \begin{cases} 1, & \text{if } \ x_i \geq 0 \\ 0, & \text{otherwise} \end{cases} \ , \tag{5.2}$$

where $(\mathbf{1}_{\boldsymbol{x}\geq 0})_i$ is i-th element of $\mathbf{1}_{\boldsymbol{x}\geq 0}$.

So perceptrons from the first and second hidden layers output $+1$ or -1. Why it is chosen in such manner will be clear from further explanations.

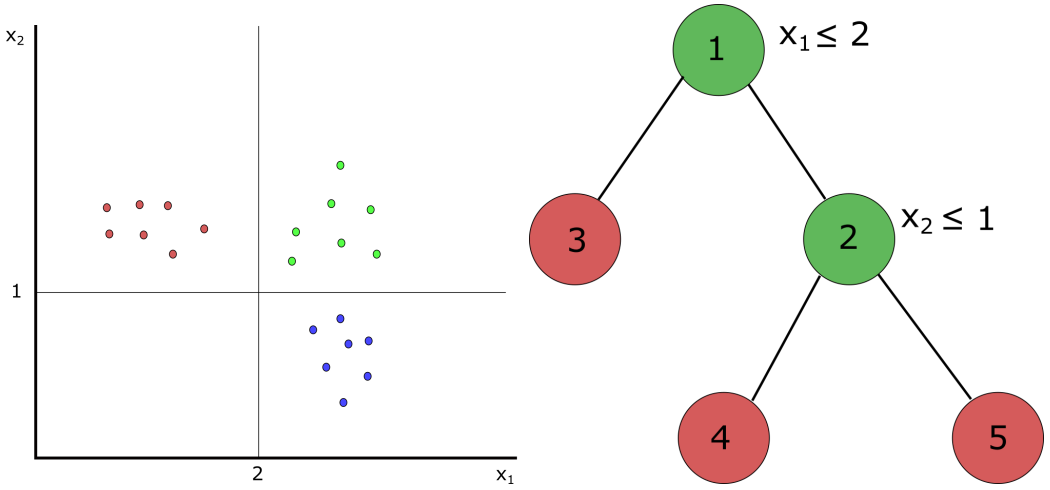


Figure 5.1: The left figure illustrates 3 color-coded classes divided by the decision tree in the right-hand figure, that are separated by two hyperplanes: $x_1 - 2 = 0$ and $x_2 - 1 = 0$. Inner nodes of the decision tree are green colored with corresponding split function next to them, whereas the leaves are red colored. All nodes are numbered.

5.1.1 First hidden layer

The first hidden layer should copy the decisions of all inner nodes present in the corresponding decision tree. The first hidden layer has the same number of neurons as the number of inner nodes in the decision tree. As explained in the chapter about decision trees and random forests, each inner node $k \in \{1, ..., L-1\}$ of the decision tree, where $L-1$ is the total number of inner nodes (in fact, if $L-1$ is the total number of inner nodes, then L is the total number of leaves present in the decision tree), possesses split function with parameters $j_k \in \{1, ..., n\}$, which is one dimension in a n-dimensional space that is used for split and also α_{j_k}, which is a threshold. Let us also define function s_k as

$$s_k(\boldsymbol{x}) = x_{j_k} - \alpha_{j_k} \quad . \tag{5.3}$$

It is apparent that equation $s_k(\boldsymbol{x}) = 0$ defines a hyperplane in $\mathbb{R}^n$, which splits the space in inner node k.

In order to obtain all decisions of inner nodes in the corresponding neurons of our neural network, we initialize weights of connections pointing from the input layer to the first hidden layer in neuron k as $(0, .., 0, 1, 0, .., 0)^T$ with single 1 in j_k-th position and 0 otherwise. Bias is set to $-\alpha_{j_k}$. Hence, the output of the first hidden layer is $(\tau(s_1(\boldsymbol{x})), \tau(s_2(\boldsymbol{x})), ..., \tau(s_{L-1}(\boldsymbol{x}))^T$ and it precisely copies the decisions of inner nodes, with +1 indicating that the input instance belongs to the right side of the hyperplane and -1 otherwise (and +1 if it belongs to the hyperplane). This is also done for the inner nodes outside the decision tree path of the input instance. The illustration can be seen in Figure 5.3.

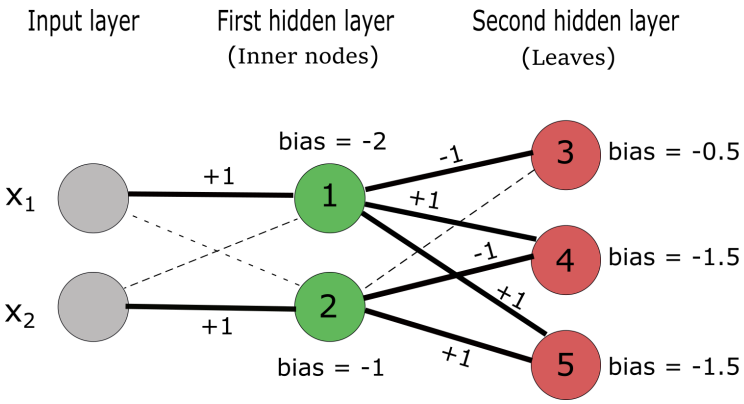


Figure 5.2: Transformation of the decision tree in Figure 5.1 to the neural network with two hidden layers. The first hidden layer detects the decisions of inner nodes with the same numbers as in 5.1. The second hidden layer retrieves leaf membership of the input instance similarly to the decision tree. Not null connections between neurons are in bold with corresponding weights. Biases are written next to the neurons. All dashed connections indicate null connections (weight equals 0).

5.1.2 Second hidden layer

From all inner node decisions obtained by the first hidden layer, it should be possible to reconstruct the exact leaf membership of input instance $\boldsymbol{x}$. This is the main task to accomplish by the second hidden layer. If there are $L-1$ neurons in the first hidden layer, then the second hidden layer consists of L neurons, each one corresponding to one individual leaf in the decision tree.

We connect neuron m in the first hidden layer to neuron m' in the second hidden layer with not null weight if and only if the inner node corresponding to neuron m belongs to the path from the root node to the leaf corresponding to neuron m'. The weight is initialized to +1 if the split by inner node m is to the right child and -1 otherwise. If neuron m is not part of the path from the root node to the leaf, the weight is always initialized to 0.

Based on this setting, it could be simply deduced that the number of not null weights of connections from the first hidden layer to arbitrary neuron m' in the second hidden layer is the same as the length of the path from the root node to leaf m'. This is illustrated in Figure 5.4.

If the output from the first hidden layer is $\boldsymbol{v} = (\pm 1, \pm 1, ..., \pm 1)^T$, which encodes all decisions of inner nodes of the decision tree, then the output of neuron m' in the second hidden layer is $\tau(\sum_{i=1}^{L-1}(w_i^{m'} v_i) + bias(m'))$, where
$\boldsymbol{w}^{m'} = (w_1^{m'}, w_2^{m'}, ..., w_{L-1}^{m'})^T$ is a vector of weights for connections to neuron m'. These weights are not null if the corresponding inner node is involved in the path from the root node to leaf m' and are +1 if it is sent to the right child and -1 otherwise. For all inner nodes that are not involved in the root-leaf path the weights are initialized to 0.

Desired behaviour of the neuron m' in the second hidden layer is to output +1 if the input instance ends in leaf m' and -1 otherwise. For this purpose, $bias(m')$ must be correctly set. To do so, it is sufficient to notice that the sum $\sum_{i=1}^{L-1}(w_i^{m'} v_i)$ as the first term in the argument of τ function equals to the length of the path from the root node to leaf m' if and only if the input ends in leaf m'. For convenience, let us denote this length as $l(m')$. It is a simple consequence of the fact that the number of not null weights $w_{i_k}^{m'}$ is the same as $l(m')$ and also they have the same

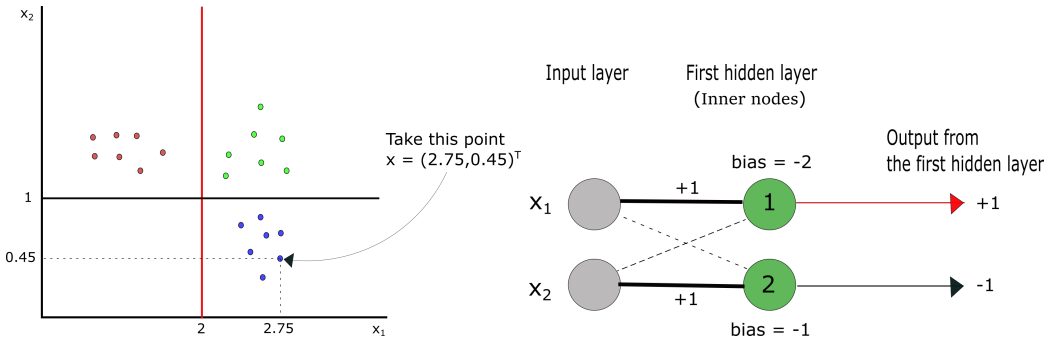


Figure 5.3: Space in the left figure is divided by red (in inner node number 1) and black (in inner node number 2) hyperplanes. If we consider point $\boldsymbol{x} = (x_1, x_2)^T = (2.75, 0.45)^T$ highlighted in the left figure, the output from the first hidden layer is +1 from neuron 1 (corresponding to the red hyperplane) and -1 from neuron 2 (corresponding to the black hyperplane).

magnitude ($|w_{i_k}^{m'}| = 1$) and same sign as v_{i_k}, where $i_k \in \{1, ..., L-1\}, w_{i_k}^{m'} \neq 0$. Therefore, each member of the sum $\sum_{i=1}^{L-1}(w_i^{m'} v_i) = \sum_{i_k=1, w_{i_k}^{m'} \neq 0}^{L-1}(w_{i_k}^{m'} v_{i_k})$ equals to +1 and all members sum up to $l(m')$.

Moreover, if the input does not end in leaf m', then in the sum $\sum_{i=1}^{L-1}(w_i^{m'} v_i)$ exists a not null member, in which 2 integers (ones) with different signs interfere, resulting in -1. Hence, it is clear that if the input does not end in leaf m', the sum holds inequality $\sum_{i=1}^{L-1}(w_i^{m'} v_i) \leq l(m') - 1 < l(m')$. For more precise intuition, the illustration is provided in Figure 5.5.

After these considerations, a reasonable choice of $bias(m')$ is

$$bias(m') = -l(m') + 0.5 \tag{5.4}$$

and then $\sum_{i=1}^{L-1}(w_i^{m'} v_i) + bias(m')$ has the following property:

$$\sum_{i=1}^{L-1}(w_i^{m'} v_i) + bias(m') \begin{cases} > 0, & \text{if input ends in leaf } m' \\ < 0, & \text{otherwise} \end{cases} . \tag{5.5}$$

With respect to this property, the second hidden layer outputs $(-1, ..., -1, +1, -1, ..., -1)^T$ after the application of τ, with a single positive +1 indicating the correct leaf membership of the input.

To retain (5.5), it is sufficient to choose any other arbitrary constant in (5.4) in range $(0, 1)$ instead of 0.5. But to stay consistent with [3], we also used the proposed value of 0.5 in conducted experiments.

At this stage, we have already defined the architecture and initial weights and biases settings of the first two hidden layers in order to transform the decision tree into the neural network with the same properties. All that remains to do is to gain classification predictions from the second hidden layer.

5.1.3 Output layer

In this section we propose the architecture and initial setting for the output layer, which will gain the same predictions as the decision tree. Unfortunately, the method proposed for regression trees

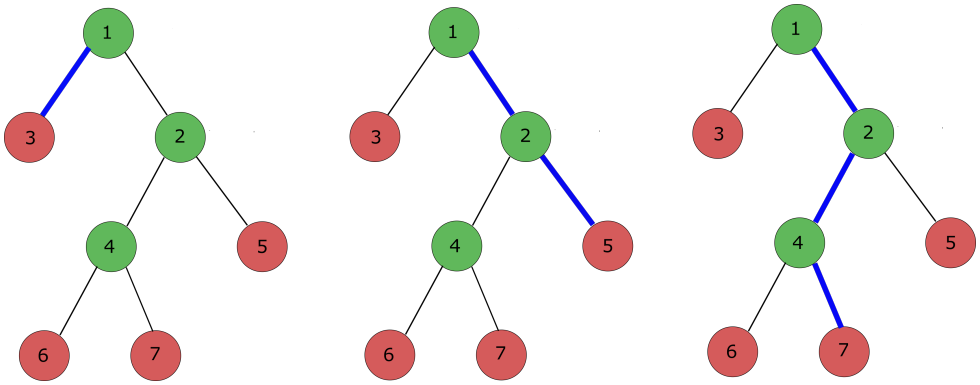


Figure 5.4: In the left figure a path from the root node 1 to leaf 3 is highlighted in blue. This has length 1 and there is only one initialized not null connection from the first hidden layer, because only the root node is part of the path. In the remaining figures the paths for other leaves (blue colored) from the root node are shown (except for leaf 6, which is in same depth as leaf 7). It is easy to see that there is always equality between the length of the path from the root node to the particular leaf and the number of not null connections from the first hidden layer to the corresponding neuron.

in [3] is not directly applicable in the classification case. Therefore, we propose an alternative for the classification case, that gives the same classification outcomes as the corresponding decision tree.

The output layer will be constructed as follows: The number of neurons in the output layer is equal to the number of classes we desire to classify. Each neuron corresponds to only one particular class (one label). A neuron with the highest activation represents the predicted class of neural network. In the experiments, we primarily applied softmax activation function in the output layer. To get the same performance as the decision tree, we must retrieve probability distributions stored in leaves from the leaf membership encoded in the second hidden layer. If the output layer outcomes the same probability values for classes as the decision tree, hence the neural network performs alike.

Let us denote the output of the second hidden layer as $\boldsymbol{r} = (-1, ..., -1, +1, -1, ..., -1)^T$, where the position of $+1$ indicates the leaf which the input falls in. For each leaf $l \in \{1, ..., L\}$ we denote a probability vector $\boldsymbol{p^l} = (p^l_1, p^l_2, ..., p^l_{|\mathbb{C}|})^T$ with probabilities of individual classes that are stored in leaf l, where $|\mathbb{C}|$ is the total number of classes. If $\boldsymbol{r}$ has $+1$ as the first element (corresponding to the first leaf), i.e. $r_1 = +1$, the output layer should outcome $\boldsymbol{p}^1$. If $r_2 = +1$, the outcome should be $\boldsymbol{p}^2$ etc.

If we initialize biases in the output layer to 0, then the appropriate initialization weights can be obtained by solving the system of linear equations with a regular matrix $\mathbb{A}$

$$\mathbb{A} = \begin{pmatrix} 1 & -1 & -1 & -1 & \dots & -1 \\ -1 & 1 & -1 & -1 & \dots & -1 \\ -1 & -1 & 1 & -1 & \dots & -1 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ -1 & -1 & \dots & -1 & 1 & -1 \\ -1 & -1 & \dots & -1 & -1 & 1 \end{pmatrix} . \tag{5.6}$$

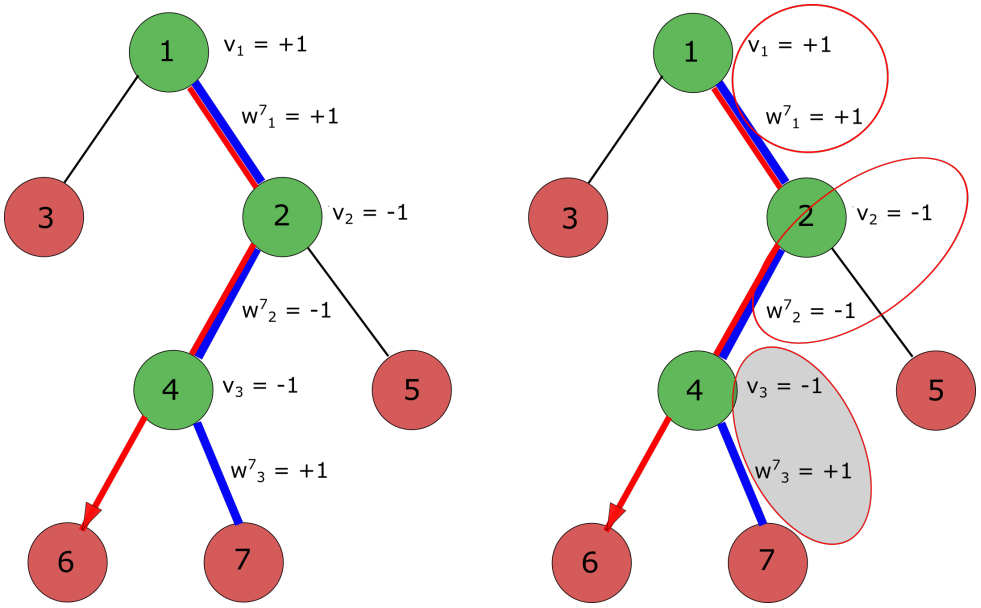


Figure 5.5: Demonstration of inequality $\sum_{i=1}^{L-1}(w_i^{m'} v_i) \leq l(m') - 1 < l(m')$ in case of leaf number 7 (corresponding to one neuron in the second hidden layer of our network), if the input does not end in leaf m'. The red path in the picture indicates the real path of the input in the decision tree. Our sample input ends in leaf number 6, as can be seen from the illustration. The output from the first hidden layer would therefore be $\boldsymbol{v} = (v_1, v_2, v_3)^T = (+1, -1, -1)^T$. But weights corresponding to neuron 7 (leaf 7) are $\boldsymbol{w}^7 = (w_1^7, w_2^7, w_3^7)^T = (+1, -1, +1)^T$. After multiplying the values in red circles and summing the results up, we obtain $\sum_{i=1}^{L-1}(w_i^7 v_i) = 2 < 3$, where 3 means the length of the root-leaf path. The decrease is caused by the interference of different signs in the grey circle.

Matrix $\mathbb{A} \in \mathbb{R}^{L \times L}$, where L is the total number of leaves in the decision tree. The determinant of matrix $\mathbb{A}$ is

$$\det\mathbb{A} = \begin{vmatrix} 1 & -1 & -1 & -1 & \dots & -1 \\ -1 & 1 & -1 & -1 & \dots & -1 \\ -1 & -1 & 1 & -1 & \dots & -1 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ -1 & -1 & \dots & -1 & 1 & -1 \\ -1 & -1 & \dots & -1 & -1 & 1 \end{vmatrix} = (-1)^{2L-3} \cdot 2^{L-1} \cdot (L-2) \ . \tag{5.7}$$

For the proof of (5.7) see Proof 1 in Appendix B. Matrix $\mathbb{A}$ is therefore always regular with the exception of $L = 2$. In this case, $\det\mathbb{A} = 0$ and matrix $\mathbb{A}$ is singular. But for $L = 2$, the decision tree has only the root node and 2 leaves, which is rarely a well-functional model in practical use. It could have a good performance almost only in case of data with significantly unambigous geometric deployment, where occurs only 2 classes and exists one hyperplane that sufficiently separates them. In practice, we will almost never encounter such an elementary model.

In case of invertible activation function $\boldsymbol{\phi}$ in the output layer, we obtain appropriate weights for neuron $c \in \{1, ..., |\mathbb{C}|\}$ in the output layer after solving the following system of linear equations:

$$\mathbb{A} \begin{pmatrix} w_1^c \\ w_2^c \\ \vdots \\ w_L^c \end{pmatrix} = \boldsymbol{\phi}^{-1}(\begin{pmatrix} p_c^1 \\ p_c^2 \\ \vdots \\ p_c^L \end{pmatrix}) \ , \tag{5.8}$$

where $\boldsymbol{w}^c = (w_1^c, ..., w_L^c)^T$ are weights of connections from the second hidden layer to neuron c in the output layer and $\boldsymbol{\phi}^{-1}$ is inverse function of $\boldsymbol{\phi}$. We used simplified notation and $\boldsymbol{\phi}$ denotes either vector functions (e.g. softmax) or scalar functions applied in element-wise fashion (sigmoid).

In other words, the appropriate weights for neuron c in the output layer are obtained as

$$\begin{pmatrix} w_1^c \\ w_2^c \\ \vdots \\ w_L^c \end{pmatrix} = \mathbb{A}^{-1}(\boldsymbol{\phi}^{-1}(\begin{pmatrix} p_c^1 \\ p_c^2 \\ \vdots \\ p_c^L \end{pmatrix})) \ . \tag{5.9}$$

This is perfectly feasible for sigmoid or the Leaky ReLU, which are both invertible. The situation is a little bit problematic in case of softmax. It could easily be derived that softmax inversion is not unique. More specifically, if we consider vector $\boldsymbol{w}$ and $\hat{\boldsymbol{w}}$, where $\forall i \in \{1, ..., L\}, \hat{w}_i = w_i + r$ for arbitrary real constant r, then the application of softmax on both variants would produce $\forall k \in \{1, ..., L\}$ the same result, which follows from:

$$[\boldsymbol{\theta}(\hat{\boldsymbol{w}})]_k = \frac{e^{w_k + r}}{\sum_i e^{w_i + r}} = \frac{e^r e^{w_k}}{e^r \sum_i e^{w_i}} = \frac{e^{w_k}}{\sum_i e^{w_i}} = [\boldsymbol{\theta}(\boldsymbol{w})]_k \ .$$

Fortunately, we are interested in the arbitrary solution, which would produce probability vector after the application of softmax. Thus, we define (one-sided) inversion of softmax $\forall k \in \{1, ..., L\}$, $\forall \boldsymbol{z}$, where $\forall k \in \{1, ..., L\}, 0 < z_k \leq 1, \sum_k z_k = 1$, as

$$[\boldsymbol{\theta}^{-1}(\boldsymbol{z})]_k = \ln(z_k) \ . \tag{5.10}$$

This definition fulfills condition $\boldsymbol{\theta}(\boldsymbol{\theta}^{-1}(\boldsymbol{z})) = \boldsymbol{z}$, because $\forall k \in \{1, ..., L\}$ applies

$$\frac{e^{\ln z_k}}{\sum_i e^{\ln z_i}} = \frac{z_k}{\sum_i z_i} = z_k \ ,$$

because $\sum_i z_i = 1$. Note that this only holds when the original softmax is applied on the inversion defined in (5.10). It does not work in reverse order. We can sometimes come across the situation when $\exists k \in \{1, ..., L\}, z_k = 0$. In such case we slightly modify the original value to 10^{-3}, which ensures only negligible deviation from the original probabilities and (in vast majority of cases) does not affect predictions of the model.

After initialization of biases in the output layer to 0 and solving the system of linear equations in (5.9) for all neurons in the output layer (or computing the inverse of matrix $\mathbb{A}$) and for appropriate activation function chosen in advance, we also gain all initialization weights. This initial setting causes the neural network to output the same predictions as from the corresponding decision tree and hence to get an equally performing classification model.

Chapter 6

Decision-tree-motivated models

6.1 Threshold function replacement

In order to apply a reasonable training procedure with the backpropagation algorithm to the proposed method, it is suitable to replace the perceptron activation function τ (threshold function) with its smooth approximation. For this purpose, the hyperbolic tangent function was adopted. This affects the original transformation, because due to this approximation the resulting neural network is no longer one-to-one transformation to the former decision tree. But still under certain conditions (especially the transition slope from the negative part of the tanh function to the positive one - the more upright, the better approximation we get) it could come fairly close to τ and therefore entire model performance remains unchanged with respect to the decision tree. Usage of tanh in the first and the second hidden layer is necessary with respect to the transformation procedure, even that tanh has some drawbacks as activation function (described in section 4.6). Using other activation functions in this case is not appropriate, because it will lack any decision tree relationship.

The closeness of tanh to the threshold function in our experiments is controlled with $\beta \geq 1$ parameter.

$$\tanh(\beta x) = \frac{e^{2\beta x - 1}}{e^{2\beta x} + 1} \tag{6.1}$$

The higher β is, the better approximation of perceptron activation function is observed. With $\beta \to \infty$, tanh converges to τ. In the experiments were exploited two parameters, β_1 in the first hidden layer and β_2 in the second hidden layer, each to control transitions of tanh to τ in the corresponding hidden layers. We will refer to them as *transition parameters*. Their effect on overall performance will be examined in further chapters. The influence of these parameters on the shape is depicted in Figure 6.1.

6.2 Competitive decision-tree-motivated models

In this section, an overview of all decision-tree-motivated models examined in the experimental part of the thesis is provided. Alongside the reference model proposed in chapter 5, we propose

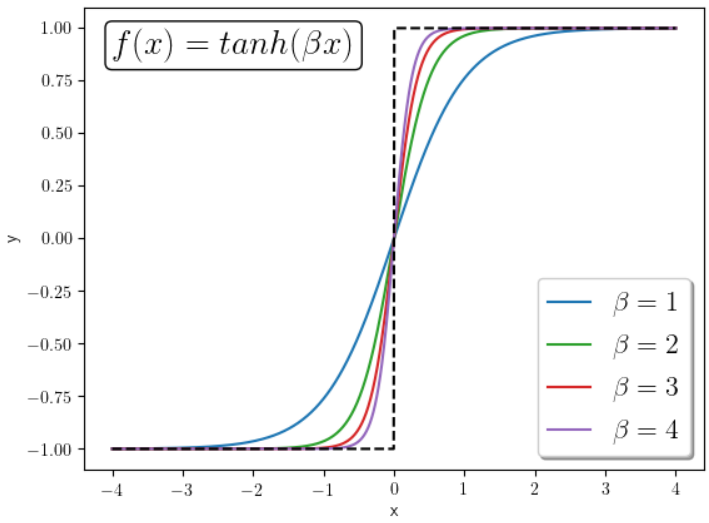


Figure 6.1: The shape of hyperbolic tangent for different β parameters.

other competitive models motivated by the original decision tree structure. These models exploit knowledge acquired by training the decision tree and use it for setting the proper weights and biases, but not all of them. They especially relax conditions on weights and biases corresponding to the output layer (the reference model has deterministic weights and biases gained from the decision tree in all layers) and initialize them randomly. This does not provide accurate decision tree transformation as our reference model but instead gives neural network more space to adapt in its own direction. Generally, in the experimental part were all random weights of connections pointing from arbitrary layer l to following layer $(l+1)$ initialized randomly from Gaussian distribution of mean equal to 0 and standard deviation equal to $\frac{1}{\sqrt{n_l}}$, where n_l denotes the number of neurons in layer l. All biases were initialized randomly from normalized Gaussian distribution of mean equal to 0 and standard deviation equal to 1. This random weight initialization was inspired by the recommendation from [38].

Reference model (NT)

Acquisition of this model is described in chapter 5. Its main purpose is to simulate corresponding decision tree behaviour as good as possible from the very beginning (the quality of simulation is controlled by β_1 and β_2 transition parameters of hyperbolic tangent in the first and second hidden layer). With both βs approaching infinity, the reference model gives the same predictions as the decision tree. Essentially, with respect to the predicting procedure of the decision tree based on picking the label with the highest probability, it could converge to predicting the same results with βs only "high enough". After transformation, the neural network is already a sufficient model capable of making relevant predictions, because it could produce similar predictions as the corresponding decision tree. This fact causes the neural network to share resembling features as the decision

tree, mainly regarding the adjustments of decisions, inclination to overfitting, etc. The closeness of this model to the corresponding decision tree provides a good opportunity to enhance performance of the decision tree by further backpropagation training, achieving improvement only after a few epochs of training. Analysis of these issues will be included in later chapters.

NT_basic

The first competitive model has weights and biases corresponding to the first and second hidden layer initialized similarly as the same layers in the reference model. The difference is in setting weights and biases corresponding to the output layer. In this case we initialize weights and biases randomly, as is depicted in Figure 6.2.

This modification should reveal some properties related to advantages or disadvantages of simulating the decision tree already before the beginning of backpropagation or leaving the output layer relaxed with space for the neural network to adapt it in its own way. Also, the sensitivity to different neural network or decision tree hyperparameters may vary. These questions will be examined and tested in the experimental part.

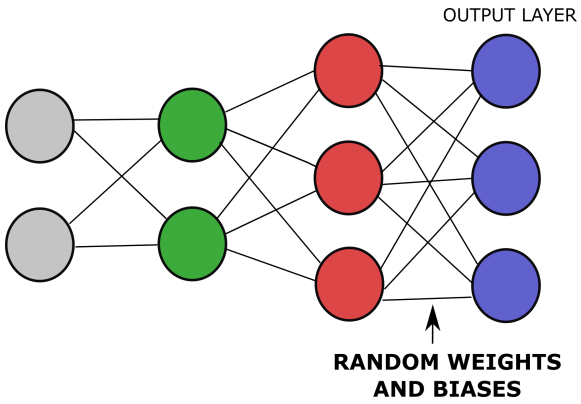


Figure 6.2: NT_basic. In contrast with the reference model, the weights and biases corresponding to the output layer are initialized randomly.

NT_EL

Next competitive model adds an extra layer of neurons between the second hidden layer and the output layer of the reference model. The weights and biases corresponding to all layers up to this new extra layer are initialized in the same manner as in the case of the NT (the extra layer now corresponds to the output layer of the NT). This makes the extra layer to output the same results as the output layer of NT. Finally, the weights and biases corresponding to the output layer of NT_EL are initialized randomly. In the extra layer, the Leaky ReLU is used as the activation function. This model is illustrated in Figure 6.3.

Adding an extra layer increases complexity, but could have a positive impact on the final performance and outperform previous models.

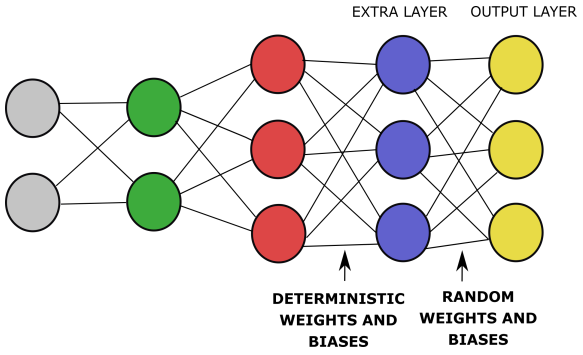


Figure 6.3: NT_EL. The weights and biases corresponding to the extra layer are initialized deterministically like in the case of the NT model. Other weights and biases are initialized randomly.

NT_EL_DW

This model is motivated by the NT_EL model. The weights and biases up to the extra layer are initialized in the same way as in the case of the NT_EL model. On the contrary, the weights and biases corresponding to the output layer are initialized deterministically. The goal is to adjust the weights in a way that neural network can output the same predictions as the decision tree already at the beginning of backpropagation (for sufficiently high transition parameters β_1 and β_2), similarly as the reference model. Since the extra layer can output the same probabilities as the decision tree, it would be sufficient to traverse the output of the extra layer to the output layer. We achieve this by initializing the weights of connections from the extra layer to the output layer by identity matrix. In other words, the weight of the connection from the j-th neuron of the extra layer to the j-th neuron of the output layer is initialized to 1 and other weights pointing from the j-th neuron to the output layer are initialized to 0. This procedure is applied to each neuron of the extra layer. The biases corresponding to the output layer are initialized to 0. If we then use softmax in the output layer, the probabilities would be modified, but the order of these values would remain the same. Since the final prediction corresponds to the class with the highest probability, we end up with the same predictions as the decision tree even though the individual values are not exactly the same as in the case of the decision tree.

6.3 Neural random forest

We have already defined several transformations of a single decision tree into the neural network. Our main interest is primarily to transform an ensemble of decision trees to an ensemble of neural networks. More specifically, we focus on transforming a random forest classifier into an ensemble of neural networks, which we refer to as *neural random forest* (NF).

Neural random forest is obtained by transforming each decision tree into a neural network by predefined procedure (we described 4 different ways in the previous section). All neural networks are then trained by backpropagation and gradient-based methods.

To extract the final prediction, we combine decisions from individual neural networks. We adopted two different ways to extract predictions.

1. **Voting**: This procedure extracts predictions from individual neural networks and then outputs the label of the most frequently represented class. If two or more equally represented classes occur among predictions, then the *averaging* procedure should be applied.

2. **Averaging**: This works similarly as it does in the case of random forest. We average probability predictions of individual neural networks and then the neural random forest outputs the label corresponding to the highest averaged probability.

Commentary on the complexity

Neural random forests have to be initialized by a trained random forest in order to establish the architecture of individual neural networks. This fact naturally increases the time and space complexity in comparison with the initialization random forest. Also the number of hyperparameters of neural random forest is larger than in the case of the random forest. On the other hand, since the architecture is defined beforehand, we significantly reduce the number of hyperparameters in comparison with regular feedforward neural network.

The substantial inconvenience of neural random forest is caused by the subsequent training of every single neural network in an ensemble after the initialization. This operation is computationally very expensive. Fortunately, we can parallelize the training procedure, since the training of individual neural networks is completely independent. Moreover, we can accelerate the matrix-based operations (matrix multiplications, etc.) in the backpropagation algorithm by usage of graphics processing units (GPU), which is already a common practice among neural networks. These optimizations significantly accelerate the training time and therefore reduce the training time requirements of the neural random forests. On a sufficiently powerful machine, we can roughly match a training time of neural random forest (after initialization) to a training time of a single feedforward neural network of similar architecture as the individual estimators in an ensemble. This is already a fine compensation, which makes the neural random forests well-applicable to a wide range of applications.

Chapter 7

Experiments

This chapter focuses on the experimental part of the thesis. Proposed models were applied to several experimental datasets under various parametric settings. The performances were analysed among all models to provide a quantitative comparison of quality and stability of the outcomes. The main emphasis is put especially on the comparison of the neural random forest (NF) models with the corresponding random forest (RF) models. Moreover, the results of the NF and RF models are also compared to the performance of other popular classification algorithms, such as the logistic regression (LR) and the feedforward neural network (NN). The experiments are divided into three separate parts.

The first part is dedicated to the evaluation on the pre-selected public datasets. All proposed NF models are evaluated on these datasets and compared with RF and other models. Additionally, the effect of the NF and RF hyperparameters on the classification performance is examined and visualized on the supporting graphs.

The second part comprises various studies of the NF parameters that could be highly influential on the overall performance of the models. The experiments are conducted on the so called "toy" datasets - artificially created datasets that simulate different benefitial or disadvantegous features, which are more or less favourable to the RF or NF. These experiments reveal some patterns and features of the NF models that could be exploited in the practical application.

In the last part, the best NF models are applied to the classification task on the real dataset consisting of medical data from the LUCAS project. The LUCAS project collaborates with medical institutions and hospitals in order to collect data of patients diagnosed with lung cancer. These data combine personal information about the patients together with their diagnostic and treatment records. The classification task is to predict the first treatment based on the patient's input diagnostic data. Generally, the early correct treatments play a crucial role in the overall survival of the patient with such diagnosis. This classification task is characterized by high-dimensionality, sparsity and the lack of significant records, which makes it challenging for a lot of classification algorithms.

7.1 Public dataset experiments

This section provides a comparison of the NF and competitive models evaluated on the set of public datasets selected in advance. Also the effect of the choice of the NF parameters on the

overall performance is examined and visualized.

MODEL	DESCRIPTION	HYPERPARAMETERS
LR	Logistic regression model	-
NN	Neural network with 2 hidden layers	Number of epochs = 30, Mini-batch size = 10 Learning rate $\eta = 0.01$ Number of neurons in 1^{st} hidd. layer = 60 Number of neurons in 2^{nd} hidd. layer = 60
RF20	Random forest with 20 estimators	Number of estimators = 20 Maximum depth = 6 Num. of features used for split = $\lfloor\sqrt{d}\rfloor$ ($d \in \mathbb{N}$ denotes total number of features)
RF30	Random forest with 30 estimators	Number of estimators = 30 Maximum depth = 6 Num. of features used for split = $\lfloor\sqrt{d}\rfloor$ ($d \in \mathbb{N}$ denotes total number of features)
RF50	Random forest with 50 estimators	Number of estimators = 50 Maximum depth = 6 Num. of features used for split = $\lfloor\sqrt{d}\rfloor$ ($d \in \mathbb{N}$ denotes total number of features)
NRF	Neural random forest consisting of *NT_basic* defined in chapter 6	Number of epochs = 30 Mini-batch size = 10 Learning rate $\eta = 0.002$ Initial random forest = RF20 Transition parameters $\beta_1 = 1$, $\beta_2 = 1$
NRF_DW	Neural random forest consisting of *NT* defined in chapter 6	Number of epochs = 30 Mini-batch size = 10 Learning rate $\eta = 0.0035$ Initial random forest = RF20 Transition parameters $\beta_1 = 1$, $\beta_2 = 1$
NRF_EL	Neural random forest consisting of *NT_EL* defined in chapter 6	Number of epochs = 30 Mini-batch size = 10 Learning rate $\eta = 0.005$ Initial random forest = RF20 Transition parameters $\beta_1 = 1$, $\beta_2 = 1$
NRF_EL_DW	Neural random forest consisting of *NT_EL_DW* defined in chapter 6	Number of epochs = 30 Mini-batch size = 10 Learning rate $\eta = 0.0045$ Initial random forest = RF20 Transition parameters $\beta_1 = 1$, $\beta_2 = 1$

Table 7.1: Shortcut notation of the individual models and basic descriptions with the choice of hyperparameters.

7.1.1 Overall results

The models are compared with respect to the accuracy and macro-average of F1-score on all datasets. Both values are presented in terms of mean value and standard deviation after two

subsequent 5-fold cross validations (10 iterations in total) in order to limit statistical fluctuations and gain higher robustness of the outcome.

All applied models are summarized in Table 7.1 with their shortcut notations (column MODEL).

Models in Table 7.1 are color-coded into three different areas. The yellow area includes competitive models to the RF and NF models. We adopted logistic regression with L2 regularization and neural network with ReLU activation function in the hidden layers and softmax in the output layer. We apply Cross Entropy as loss function and optimization is done via Adam optimization algorithm (Adam had superior results in comparison with the stochastic gradient descent). The hyperparameters of Adam was chosen as follows: $\gamma_1 = 0.9$, $\gamma_2 = 0.99$, $\epsilon = 10^{-8}$. We use same notation as defined in subsection 4.5.3. The implementation of the LR and NN was taken from Python modules - Scikit learn [55] and Keras [56].

The green area comprises the RF models. We adopted three variations with different numbers of estimators. All decision trees were trained with the entire training dataset and we use entropy as the impurity function in the information gain criterion. We use the implementation from Scikit learn.

The orange area includes all NF models. We tested many different combinations of hyperparameters and choices of different loss functions and activation functions (in layers where it is possible) and we present the best obtained solution. All NF models were built from the RF20 model and use Cross Entropy as loss function and softmax in the output layer of all individual neural networks. Transition parameters β_1 and β_2 in the first and second hidden layer were set to 1. Learning rates were chosen as a compromise through a grid search so it is as suitable as possible for all our datasets. We use Adam as optimization algorithm instead of SGD since the results were more favourable. The parameters of Adam were selected equally as in the case of NN: $\gamma_1 = 0.9$, $\gamma_2 = 0.99$, $\epsilon = 10^{-8}$. Single neural network estimators in the NF models were trained for 30 epochs each. We added L2 regularization term to the loss function with regularization constant $\lambda = 0.01$. Finally, the voting scheme is applied as an ensemble method to obtain final predictions from the NF (differences between results obtained by voting or averaging were negligible).

We use our own implementation of the NF models in Python. Datasets used in the experiments are summarized in Table 7.2. All categorical features were one-hot encoded, ie. transformed by function $\zeta : \{k_1, ..., k_m\} \to \mathbb{R}^m$, where $\{k_1, ..., k_m\}$ is a set of categories and $m \in \mathbb{N}$ is a number of categories. It is defined $\forall j, l \in \{1, ..., m\}$ as

$$[\zeta(k_j)]_l = \begin{cases} 1, & j = l \\ 0, & \text{otherwise} \end{cases} . \tag{7.1}$$

Eventually, all dataset features (including the newly created ones by one-hot encoding) were standardized (normalized to mean equals 0 and standard deviation equals 1) to ensure the better numerical stability. It is done by formula $\hat{x}_j = \frac{x_j - \mu_j}{\sigma_j}$, where x_j is a single realization of the j-th feature, $\hat{x}_j$ is a new value of this realization, μ is the mean value of realizations of j-th feature and σ_j is the standard deviation of realizations of the j-th feature. This normalization also had a positive effect on performance since all evaluation metrics were more favourable in the case of normalization than without normalization.

The final results of macro-average of F1-score resp. accuracy are presented in Table 7.3 resp. Table 7.4 (the results in both tables correspond to the original values multiplied by 100).

ID	DATASET	#instances	#features	#classes	feature type
1	Bank[1]	4521	16	2	categorical and numerical
2	Cars[2]	1728	6	4	categorical
3	Diabetes[3]	768	8	2	numerical
4	Messidor[4]	1151	19	2	numerical
5	USPS[5]	9298	256	10	numerical
6	Vehicle[6]	846	18	4	numerical
7	Wine[7]	178	13	3	numerical
8	OBS[8]	1075	21	4	categorical and numerical

Table 7.2: Public datasets.

DATASET	NN	LR	RF20	RF30	RF50	NRF_DW	NRF	NRF_EL	NRF_EL_DW
Bank	69.3 ± 1.8	68.6 ± 2.5	55.5 ± 2.0	55.7 ± 2.2	54.6 ± 2.4	$\mathbf{71.9} \pm 3.2$	71.6 ± 1.5	71.7 ± 2.6	$\mathbf{71.9} \pm 2.7$
Cars	99.8 ± 0.4	66.6 ± 5.4	61.5 ± 6.4	59.5 ± 4.9	60.1 ± 6.3	$\mathbf{100} \pm 0.0$	$\mathbf{100} \pm 0.0$	$\mathbf{100} \pm 0.0$	$\mathbf{100} \pm 0.0$
Diabetes	68.9 ± 3.0	$\mathbf{73.0} \pm 4.8$	72.7 ± 4.7	72.3 ± 2.8	72.7 ± 2.9	71.6 ± 3.9	72.7 ± 2.4	72.3 ± 3.1	71.6 ± 5.2
Messidor	71.3 ± 3.0	72.0 ± 1.5	67.4 ± 3.0	70.0 ± 3.7	65.7 ± 2.1	$\mathbf{73.8} \pm 2.6$	73.5 ± 1.0	73.7 ± 2.3	73.6 ± 3.7
USPS	94.0 ± 1.9	92.6 ± 1.2	90.2 ± 1.8	90.9 ± 1.7	90.9 ± 1.3	$\mathbf{97.3} \pm 1.4$	97.1 ± 1.5	97.0 ± 1.4	97.0 ± 1.3
Vehicle	81.3 ± 3.3	77.3 ± 2.5	71.8 ± 3.5	72.1 ± 1.9	71.4 ± 3.0	$\mathbf{84.2} \pm 3.5$	$\mathbf{84.2} \pm 2.3$	83.2 ± 3.1	83.9 ± 3.0
Wine	97.7 ± 1.9	98.2 ± 2.1	97.9 ± 2.0	97.5 ± 3.0	97.9 ± 2.0	97.9 ± 2.3	97.9 ± 2.4	97.5 ± 2.0	$\mathbf{98.5} \pm 2.1$
OBS	99.4 ± 0.4	99.2 ± 1.0	90.2 ± 2.0	91.9 ± 1.9	92.4 ± 1.8	99.8 ± 0.4	$\mathbf{99.9} \pm 0.3$	99.6 ± 1.0	$\mathbf{99.9} \pm 0.2$

Table 7.3: Macro average of F1-score.

DATASET	NN	LR	RF20	RF30	RF50	NRF_DW	NRF	NRF_EL	NRF_EL_DW
Bank	88.1 ± 1.0	$\mathbf{90.0} \pm 1.4$	89.1 ± 0.8	89.1 ± 0.8	89.1 ± 1.0	89.6 ± 0.6	89.4 ± 0.7	89.8 ± 0.8	89.5 ± 1.0
Cars	99.9 ± 0.2	88.5 ± 1.8	89.6 ± 1.3	89.6 ± 1.5	90.2 ± 2.1	$\mathbf{100} \pm 0.0$	$\mathbf{100} \pm 0.0$	$\mathbf{100} \pm 0.0$	$\mathbf{100} \pm 0.0$
Diabetes	71.9 ± 3.1	$\mathbf{76.9} \pm 4.4$	76.4 ± 4.1	76.2 ± 2.0	76.7 ± 2.2	74.6 ± 3.7	75.8 ± 2.6	75.0 ± 2.8	74.3 ± 5.1
Messidor	71.3 ± 2.9	72.1 ± 1.5	67.5 ± 2.9	67.1 ± 3.7	65.9 ± 1.9	$\mathbf{73.8} \pm 2.7$	73.5 ± 1.0	$\mathbf{73.8} \pm 2.3$	73.7 ± 3.6
USPS	94.6 ± 1.9	93.4 ± 1.2	91.3 ± 1.8	91.9 ± 1.7	91.9 ± 1.4	$\mathbf{97.6} \pm 1.4$	97.4 ± 1.4	97.3 ± 1.3	97.3 ± 1.2
Vehicle	81.4 ± 3.1	77.9 ± 2.6	73.2 ± 4.1	73.5 ± 1.7	72.8 ± 3.1	84.1 ± 4.1	$\mathbf{84.3} \pm 2.3$	83.3 ± 2.4	83.9 ± 3.0
Wine	97.7 ± 1.8	98.3 ± 2.0	97.8 ± 3.2	97.8 ± 2.9	98.0 ± 1.9	97.8 ± 2.2	98.0 ± 2.3	97.5 ± 2.1	$\mathbf{98.6} \pm 2.0$
OBS	99.3 ± 0.5	99.1 ± 0.9	87.7 ± 2.5	89.5 ± 2.6	90.2 ± 2.5	99.7 ± 0.6	$\mathbf{99.9} \pm 0.4$	99.6 ± 0.9	$\mathbf{99.9} \pm 0.3$

Table 7.4: Accuracy.

In the first observation we may notice that all models performed very similarly on the majority of datasets with respect to both researched evaluation metrics. Essentially, the overall performance of the NF models in comparison with other competitive models is in fact equal or slightly better

[1] Available at: mldata.io/dataset-details/bank_marketing/
[2] Available at: mldata.io/dataset-details/cars/
[3] Available at: data.world/data-society/pima-indians-diabetes-database
[4] Available at: archive.ics.uci.edu/ml/datasets/Diabetic+Retinopathy+Debrecen+Data+Set
[5] Available at: kaggle.com/bistaumanga/usps-dataset
[6] Available at: mldata.io/dataset-details/vehicle_silhouette/
[7] Available at: scikit-learn.org/stable/modules/generated/sklearn.datasets.load_wine.html
[8] Available at: archive.ics.uci.edu/ml/datasets/Burst+Header+Packet+%28BHP%29+flooding+attack+on+Optical+Burst+Switching+%28OBS%29+Network

than in case of concurrent models. There is no significant deterioration of any of the NF models compared to other models, especially compared to the RF models. On the contrary, we can see enhancement of values of the NF models in most cases.

The most significant differences between the NF models and the RF models could be seen in the cases of the Cars, Messidor, USPS, Vehicle and OBS datasets, where the NF models have a superior performance in comparison with the RF models. On the Bank dataset, the performance of the NF models is basically equal or only slightly better. The only exception is visible in the case of the Diabetes dataset, where the NF models do not produce any improvement and perform similarly or sligthly worse than the RF models. It could be caused by many influencing factors, especially the inappropriate choice of hyperparameters of the NF models. The effect of hyperparameters on final performance will be researched later.

The NF models among themselves produce very similar results. NRF_DW and NRF_EL_DW have almost the same performance on all datasets with very small differences, which is a good sign, since they both aim to simulate random forest already before training individual estimators with backpropagation and therefore have comparable initial conditions. Moreover, NRF_DW has one hidden layer less, so stacking more layers does not necessarily mean better performance. NRF and NRF_EL have relaxing layers with random weights, so there is more space left for individual neural networks to modify decision tree initial knowledge and adapt in its own way. This could be beneficial especially in those cases where the RF models produce a superior performance in comparison with other models, and therefore, there is not much space for further significant improvement if the RF model is imitated. This is nicely demonstrated in the case of the Diabetes dataset, where the RF models produce fine results in the beginning and then NRF_DW and NRF_EL_DW are not successful in further improvements. It even reached slightly "worse" results (note the quotation marks since there is a relatively high standard deviation, which makes it hard to distinguish the better model from the worse one). On the other hand, NRF and NRF_EL managed to come fairly close to the RF model performance or even equalize it and have a slightly better and more stable (lower standard deviations) performance than NRF_DW and NRF_EL_DW in this case. Of course, a better choice of hyperparameters of NRF_DW and NRF_EL_DW (especially number of training epochs and transition parameters β_1 and β_2 which control the quality of imitation of a single decision tree) could lead eventually to surpassing even the well-performing RF. This issue will be studied later in the thesis.

7.1.2 The effect of hyperparameters

In this section, various combinations of hyperparameters of the NF and RF models are studied and different visualizations are provided to compare the results. Prior knowledge of the RF model features and the NF model could define the suitable range of hyperparameter values, which could result in correct choice and superior performance of the model.

We use the Vehicle and Diabetes datasets to analyze performance based on hyperparameter choice. The RF model performed faintly on the Vehicle dataset and finely on the Diabetes dataset in comparison with other models, thus the NF models initialized with the RF models trained on those two datasets cover situations that could arise in practical use: boosting the well-performing RF or the poor-performing RF by the NF. If we choose bad hyperparameters in the case of the poor RF, the resulting NF could be poor as well, and if we choose bad hyperparameters in the case of the fine RF, the resulting NF could perform worse than the corresponding RF, which is undesirable.

All graphical visualizations are presented in Appendix A and referenced in this section. Unless stated otherwise, all experiments were performed as 2 subsequent independent 5-fold cross validations (10 iterations in total) and the outcomes are shown in the form "mean $\pm$ std" in the corresponding figures. Also, if the results of the RF models are presented together with the results of the NF models in one figure, then these RF models were used to initialize all coresponding NF models in the same figure. In all following experiments, the hyperparameters and settings of models which are not examined at the time remain same as defined in Table 7.1 and subsection 7.1.1.

Learning rate

Learning rate stands for one of the most crucial hyperparameters that need to be set conscientiously in order to achieve quality performance. This choice usually depends on the model type and the dataset itself. It is done almost exclusively by exhaustive grid search on a specific validation dataset before picking a suitable value for the training phase. It is not possible to state a versatile learning rate optimal for all applications.

We study different settings for learning rates in our model and research the sensitivity of the NF models to the learning rate choice. We compare this sensitivity with the same feedforward NN as applied on public dataset experiments (defined in Table 7.1).

Other hyperparameters of the NF and RF models except for learning rate remain the same as defined in Table 7.1. NF and RF have 20 estimators.

Graphs for varying learning rates are presented in Figure A1 for the case of the Vehicle dataset. Graphs for the Diabetes dataset are in Figure A2. We present only accuracy results, since both visualizations of accuracy and F1-score have very similar trend and shape. Moreover, both datasets are balanced or have very low degree of imbalance between classes, so referring to accuracy is safe in this case.

Based on Figure A1, we can see that in all cases, the optimal value for learning rates could be found close to zero, approximately in the range $(0.001, 0.02)$. With an increasing value of the learning rate, the performance of models is declining. The steepest downfall is registered in the case of the NN model, where for the learning rate equals 0.1 drops accuracy to approximately 50% of maximum reached accuracy. On the other hand, the NF models did not suffer from such eminent decrease in the given range. This experiment suggests that the NF models could be slightly less sensitive to the learning rate than the single NN, but it really depends on the specific application. A resembling trend could be seen in Figure A2 on the Diabetes dataset, altough not that vigorous. In this case the accuracy of the NF models fluctuates a bit around an almost constant value (a mild decrease at the end of the range) in the given range of learning rates, whereas in the case of the NN, we can notice an early decrease of accuracy in the range $(0.001, 0.12)$ and then it remains basically constant. Consequently, we could choose suitable learning rates from a wider range in the case of the NF models than in the case of the single NN.

Number of epochs and number of estimators

Another important hyperparameter of NF is the number of training epochs. Especially in the case of the NF models which simulate RF behaviour from the beginning, they could improve the performance of the initial RF after a few epochs already, which could be time-efficient. We will investigate this issue alongside the effect of the number of estimators in NF (RF), because the high number of estimators could compensate for the low number of training epochs and vice versa. The

hyperparameters (except for the number of epochs and number of estimators) of NF and RF remain the same as defined in Table 7.1.

First we will focus on the experiments on the Vehicle dataset showed in Figure A3. This is the case when the RF model does not perform well in comparison with other models, as was already seen in Tables 7.3 and 7.4. One may notice an apparent pattern that the increase of number of training epochs causes enhancement of accuracy. This happens among all NF models and also in the case of the single NN. NRF_DW and NRF_EL_DW models could outperform the initial RF already after 5 epochs of training and reach slightly higher values of accuracy than the NF models with one layer of random weights until 30 epochs of training. For the higher number of epochs, the results are very similar among all NF models except for NRF_EL, which has a slightly lower accuracy. Also, NRF_DW and NRF_EL_DW converge faster to the maximum values of accuracy, acquiring it at approximately 30 epochs of training, whereas other NF models still register an increase of accuracy with the higher number of training epochs. It could be due to the nature of individual models, where NRF_DW and NRF_EL_DW imitate RF more precisely than other NF models and therefore could be closer to some optimal value of loss function than NRF and NRF_EL from the beginning. Therefore they are more likely to reach their optimal performance earlier.

We can see only small differences in accuracy values with the varying number of estimators. With the increasing number of estimators the accuracy values usually also increase, but the number of epochs is a more distinguishing factor.

A different situation occurs in the case of the Diabetes dataset, as is illustrated in Figure A4. This is the case of the RF model performing well in comparison with other models.

In this case, quite an opposite situation than in the previous case on the Vehicle dataset arises. With an increasing number of training epochs, accuracy of the NF models decreases. All NF models except for NRF_EL outperform RF with any number of estimators in the given range after 5 epochs of training already. They also reach their maximum accuracy in the given range of epochs. With a higher number of epochs, the accuracy decreases, eventually ending up with a worse model than the initial RF. It could be caused by the problem of overfitting, which will be examined later. Especially in the case of NRF_DW and NRF_EL_DW, the accuracy decreases quickly. In the case of NRF_EL, the accuracy acquires its maximum value at approximately 20 epochs and then slowly decreases. It would be beneficial to use early stopping during training the NF models to avoid this problem. Moreover, the single NN model follows the same pattern of a decreasing accuracy with an increasing number of epochs as the NF models, which suggests that neural networks in this case converge quickly to an optimal solution and with an increasing number of epochs overfit to the problem. One noticeable fact is that the performance of the NF models degradates more slowly than the single NN model, which may be caused by the benefit of ensemble voting (or averaging) of NF.

Maximum depth of trees in RF

Another examined hyperparameter of NF is the number of neurons in the first and second hidden layer of the individual neural networks. This number is fully controlled by the depth of the corresponding tree. We will examine the influence of maximum depth of trees in RF (which is set before the training of RF) on performance of NF. Especially for overgrown trees, there is a significant risk of overfitting either in the case of the decision tree or subsequently in the case of transformed neural network, which will be tested. We also add the number of training epochs of NF as another

variable. RF and subsequent NF have 20 estimators. Other hyperparameters of all models remain the same as defined in Table 7.1.

We only present graphs created on the Diabetes dataset for NRF_DW and NRF, because for the NF models with an extra added layer, the trend has turned out to be same. We also added values of Cross Entropy loss function on testing datasets in order to demonstrate overfitting. Accuracy and values of loss function depending on the maximum depth and number of epochs are depicted in Figure A5.

It can be seen in the graphs that the NRF_DW model is much more prone to overfitting than NRF model, which is logical, since NRF_DW model simulates RF behaviour better already before training with backpropagation. This applies especially to larger maximum depths (starting from maximum depth equal to 6), where we can notice a fast decrease in accuracy and increase in loss function with the increasing number of training epochs. In the case of NRF, this decrease is not that vigorous and straightforward, but for larger depths, it is still visible.

In Figure A6, the same models with a regularization constant increased to $\lambda = 10$, which intensifies the influence of regularization term in the loss function, are illustrated. We can see that this modification prevents the models from significant decrease of accuracy especially for the larger numbers of epochs, but in the majority of depth values, the maximum reached accuracy among those two variants remains favourable in the first case, as is clear from Figure A5. Still, the differences are very small.

Transition parameters β_1 and β_2

Transition parameters β_1 and β_2 (defined in section 6.2) control the degree of transition of RF to NF. The higher both values are, the better approximation of RF is obtained by NF. Under certain circumstances, it could be advantegous to increase those parameters and approximate RF accurately, especially when RF produces competitive results. Both RF and NF have 20 estimators and other hyperparameters except for transition parameters remain the same as defined in Table 7.1. Accuracy depending on both transition parameters is depicted for the Vehicle dataset resp. the Diabetes dataset in Figure A7 resp. Figure A8.

In the case of the Vehicle dataset, we can observe that it is not beneficial to increase β_1 and β_2 since the highest reached accuracy belongs to the point $\beta_1 = 1$ and $\beta_2 = 1$ among all NF models. Since the RF model performs badly in comparison with other models, this suggests that copying this RF model makes the subsequent NF model performs worse than it could when β_1 and β_2 are kept low and the NF model is more relaxed. Increasing both values simultaneously causes deterioration of the results almost monotonically.

The situation is different in the case of the Diabetes dataset in Figure A8. In contrast with the Vehicle dataset, we may observe some pairs of β_1 and β_2 other than [1;1] that are more beneficial in terms of higher accuracy. Even points [12;12] perform more favourably in the most cases of NF models than point [1;1]. This suggests that taking advantage of simulating the well-performing RF model more accurately is the step towards a better performance of NF.

Be aware of the problem of choosing transition parameters too high since it could negatively affect optimization process of neural networks. For high transition parameters the tanh functions approximate threshold functions too precisely and therefore the neurons in the first and second hidden layer could suffer from early saturation. Consequently, this could make the learning process of neural networks more difficult.

Number of features considered in the split

We also examined the number of features considered in the split during the RF training and its influence on the performance of NF. Standardly, we leave this hyperparameter on the value $\lfloor\sqrt{d}\rfloor$, where $d \in \mathbb{N}$ represents the number of features in the dataset. We tested the NF performance for values of max. considered features set to 0.2, 0.4, 0.6, 0.8 and 1, where each number symbolizes the fraction of the considered features with respect to d. The number of estimators was set to 20 and we add maximum depth as another variable. Other hyperparameters remain same as defined in Table 7.1.

So far we can claim from the performed experiments on the Vehicle dataset and the Diabetes dataset, the effect of this hyperparameter is not very significant and the differences between various settings of this hyperparameter are negligible. We present the visualizations for the Vehicle dataset in Figure A9.

Summary of the previous observations

In this part, we shortly sum up the provisional experiments with NF and RF hyperparameters performed in the previous paragraphs of this section.

We saw that the sensitivity of the learning rate has been slightly lower in the case of the NF models than in the case of the single NN. This could be due to the ensemble nature of NF with variety of single specific NN estimators, each differently sensitive to the current learning rate value. An overall sensitivity of the NF model to the learning rate is then "averaged" over single NN estimators and therefore could be reduced in comparison with only single NN. Note that this comparison has been done for other parameters fixed and the sensitivity to the learning rate could correlate with other hyperparameters too, mainly with the transition parameters β_1 and β_2. So the most suitable learning rate should be picked after some grid search on the validation dataset with other hyperparameters set beforehand.

Following the observations based on a varying number of training epochs and other hyperparameters, such as maximum depth or number of estimators, offered another interesting perspective. Especially the NF models initialized with RF consisting of trees with large depth are more prone to overfitting than in the case of lower depths, so it should be beneficial to prevent overfitting by employing techniques such as early stopping or adding a regularization term to the loss function (L2 regularization for instance). The differences between the results of the NF models with varying number of estimators were small, slightly favourable in the case of the larger number of estimators, as corresponds to the logical intuition.

As another hyperparameters, the transition parameters β_1 and β_2 were investigated. It turned out that in the case of a poor-performing RF, it is beneficial to leave both values low (for instance $\beta_1 = 1$ and $\beta_2 = 1$) and start with relaxed NF that do not simulate that RF accurately. On the other hand, when operating with a well-performing RF with competitive results, it could reach a more favourable performance when simulating RF more accurately by increasing both transition parameters. This issue will also be studied in the next chapter.

The last studied hyperparameter was the number of maximum considered features in the split of RF. From the performed experiments, we did not detect a significant dependence influencing the performance of subsequent NF.

7.2 Toy dataset experiments

This section comprises experiments on artificially created data in 2D, where we draw and compare decision boundaries learned by individual methods and search for properties and inclinations of the NF models based on varying hyperparameters and different properties of datasets.

We conducted the experiments by generating various datasets, each one of a different shape and trend. Moreover, we investigate datasets of different random noise and test the ability of classifiers to find the correct decision boundary and not to overfit on noise data, as tend to do especially overgrown RF models.

Each dataset was split to the training and testing set with the ratio of 6:4 and the decision boundary was drawn on the input space. It is difficult and not straightforward to aggregate decision boundaries of several NFs and visualize it while preserving correct interpretability, especially when classifying into more than two labels. We demonstrate the example addressing this issue. Suppose that we trained 4 independent NF classifiers on a 3-class dataset (classes are denoted simply as 0,1,2) and decisions of NFs are subsequently {0,0,1,2} in that point. Also suppose that each class occupies a different color (defined as a vector (R,G,B,α), where R,G,B denote the typical red,green,blue triplet and α denotes transparency level). If we pick the most frequent decision (or average the decision probabilities) and draw the point with the color of the most favourable class, then the result corresponds to the decision of an ensemble method consisting of several NFs, and that is not what we desire to illustrate. Also, we cannot depict all decisions in the picture, since the combination of 3 colors could result in an uninformative color that does not capture the trend.

For a 2-class case, it is possible to illustrate the frequency of decisions by adopting the color gradient between two colors, but since we work also with 3-class datasets, we will not present the results separately in a different way for 2-class datasets.

Instead, the decision boundaries and their tendencies are for individual NFs demonstrated on multiple datasets, each of a varying size, and for varying hyperparameters of NFs. In each studied case, the single NF is trained on one training set, which remains the same for all presented models. As the hyperparameters change, we can see the evolution of decision boundaries of different NFs and their inclinations. We also present a single value of accuracy on testing set for all models only for a quick comparison, but since this corresponds to only one realization, it should not be decisive in the final comparison of models. The comparison is more stable and reliable when visually comparing the decision boundaries (it is possible in 2D) in general and comparing their generalization ability.

7.2.1 Number of epochs

In this part, we investigate the effect of the varying number of training epochs on the evolution of individual decision boundaries. Decision boundaries are depicted on 3 types of artificial datasets, each type of 50 and 200 instances.

The choices of hyperparameters are described in Table 7.5. We use Adam with parameters $\gamma_1 = 0.9$, $\gamma_2 = 0.99$, $\epsilon = 10^{-8}$ as optimization algorithm among NN and all NF models. Regularization parameter λ is set to 0. All other settings are the same as defined in subsection 7.1.1.

All illustrations are depicted in Figure A10, Figure A11 and Figure A12. We can see that in the cases of 50 input instances the NRF_DW and NRF_EL_DW models manage to tune finely already after 5 epochs of training, whereas NRF and NRF_EL do not adapt very reliably and their decision boundary is very far from ideal separation (for 5 to 30 training epochs). For the larger number of training epochs, NRF_EL established decision boundaries competitively with other models, but

MODEL	HYPERPARAMETERS
RF	Number of estimators = 10 Maximum depth = 6 Num. of features used for split = 1
NN	Mini-batch size = 10 Learning rate $\eta = 0.01$ Number of neurons in 1^{st} hidd. layer = 60 Number of neurons in 2^{nd} hidd. layer = 60
NRF	Mini-batch size = 10 Learning rate $\eta = 0.002$ Initial random forest = RF Transition parameters $\beta_1 = 1$, $\beta_2 = 1$
NRF_DW	Mini-batch size = 10 Learning rate $\eta = 0.0035$ Initial random forest = RF Transition parameters $\beta_1 = 1$, $\beta_2 = 1$
NRF_EL	Mini-batch size = 10 Learning rate $\eta = 0.005$ Initial random forest = RF Transition parameters $\beta_1 = 1$, $\beta_2 = 1$
NRF_EL_DW	Mini-batch size = 10 Learning rate $\eta = 0.0045$ Initial random forest = RF Transition parameters $\beta_1 = 1$, $\beta_2 = 1$

Table 7.5: Choice of hyperparameters.

the NRF model did not. The NRF model even struggles to learn competitive separation in the case of 200 input instances and can catch up more slowly than other models, approximately around 30 training epochs. Eventually, for the higher number of input instances, the NRF model could perform competitively with other models.

Also, we can highlight the striking resemblance of decision boundaries of NRF_DW and NRF_EL_DW in all cases and even for every single number of training epochs. This proves the concept since both models are deterministic and they target to simulate RF, therefore they should share some similarities.

This experiment suggests that models simulating the RF (NRF_DW and NRF_EL_DW) share some similarities and could learn faster than other NF models. Moreover, they can adapt better to the problems with fewer training inputs.

7.2.2 Transition parameters β_1 and β_2

This part comprises experiments based on varying transition parameters β_1 and β_2, which control the accuracy of transition of RF to the corresponding NF. We will keep these parameters simultaneously on the same values and increase them at the same time, since increasing both values results in a more accurate simulation of RF. We use β to denote both values β_1 and β_2 simultaneously, thus $\beta = \beta_1 = \beta_2$.

MODEL	HYPERPARAMETERS
RF	Number of estimators = 10 Maximum depth = 6 Num. of features used for split = 1
NRF	Number of epochs = 25 Mini-batch size = 10 Learning rate $\eta = 0.002$ Initial random forest = RF
NRF_DW	Number of epochs = 25 Mini-batch size = 10 Learning rate $\eta = 0.0035$ Initial random forest = RF
NRF_EL	Number of epochs = 25 Mini-batch size = 10 Learning rate $\eta = 0.005$ Initial random forest = RF
NRF_EL_DW	Number of epochs = 25 Mini-batch size = 10 Learning rate $\eta = 0.0045$ Initial random forest = RF

Table 7.6: Choice of hyperparameters.

Choices of hyperparameters are presented in Table 7.6. Regularization parameter λ is set to 0. All other settings are the same as defined in subsection 7.1.1. We show decision boundaries on 3 different types of datasets, each consisting of 50 and 200 instances.

The illustrations of decision boundaries depending on β are depicted in Figure A13, Figure A14 and Figure A15. We can see that in the case of 50 data inputs, the NRF model could not tune its decision boundary sufficiently even after 25 training epochs (for all βs), which further confirms that this model is not suitable when working with a low number of training inputs. For 200 instances, the NRF model is able to catch up and produce competitive decision boundaries with other models.

For $\beta = 1$, all NF models produce various nonlinear decision boundaries, which are more adaptable then RF decision boundaries parallel with axes. They are able to create more complex (e.g. round or ellipsoidal related) shapes. This means that for low βs, the NF models could benefit more from their neural network properties and therefore could tune better to the problems that cannot be solved properly by a finite number of splits by RF, e.g. 3 classes ordered in concentric nested spheres made of data generated by standard normal distribution, as could be seen in Figure A15.

When β is increasing, we can see that the decision boundaries of NF models gradually converge to the decision boundary of the corresponding RF model. Especially in the case of NRF_DW and NRF_EL_DW, the resemblance is obvious and the decision boundaries for $\beta = 50$ are almost identical. Also, with increasing β, the decision boundaries of NF models evolve more in the RF manner, preferring more perpendicular and parallel shapes, as does RF itself. It is nicely demonstrated in Figure A15 in the case of 200 data inputs, where for $\beta = 1$, the decision boundaries of NF models are in great contrast with the RF boundary. They form concentric spheres which separate the classes, whereas the boundary of RF is created as concentric rectangles. But already for $\beta = 5$, the decision boundary of individual NF models becomes rectangular as well.

We see that all NF models with higher values of β_1 and β_2 simulate the initial RF quite precisely.

For lower values of β, the resemblance is less visible and the decision boundaries of NF models look more similar to the NN decision boundary, as could be seen for instance in Figure A12, where both transition parameters are set to 1.

7.2.3 Noise in data

In this part, we investigate the influence of noise in data on NF decisions. The choices of hyper-

MODEL	HYPERPARAMETERS
RF	Number of estimators = 10 Maximum depth = 6 Num. of features used for split = 1
NN	Number of epochs = 25 Mini-batch size = 10 Learning rate $\eta = 0.01$ Number of neurons in 1^{st} hidd. layer = 60 Number of neurons in 2^{nd} hidd. layer = 60
NRF	Number of epochs = 25 Mini-batch size = 10 Learning rate $\eta = 0.002$ Initial random forest = RF 1) Transition parameter $\beta_1 = 1$ and $\beta_2 = 1$ 2) Transition parameter $\beta_1 = 50$ and $\beta_2 = 50$
NRF_DW	Number of epochs = 25 Mini-batch size = 10 Learning rate $\eta = 0.0035$ Initial random forest = RF 1) Transition parameter $\beta_1 = 1$ and $\beta_2 = 1$ 2) Transition parameter $\beta_1 = 50$ and $\beta_2 = 50$
NRF_EL	Number of epochs = 25 Mini-batch size = 10 Learning rate $\eta = 0.005$ Initial random forest = RF 1) Transition parameter $\beta_1 = 1$ and $\beta_2 = 1$ 2) Transition parameter $\beta_1 = 50$ and $\beta_2 = 50$
NRF_EL_DW	Number of epochs = 25 Mini-batch size = 10 Learning rate $\eta = 0.0045$ Initial random forest = RF 1) Transition parameter $\beta_1 = 1$ and $\beta_2 = 1$ 2) Transition parameter $\beta_1 = 50$ and $\beta_2 = 50$

Table 7.7: Choice of hyperparameters.

parameters are presented in Table 7.7. We conduct these experiments in two variants, the first for $\beta = \beta_1 = \beta_2 = 1$ and the second for $\beta = \beta_1 = \beta_2 = 50$. We set the size of dataset to 100. Regularization parameter λ is set to 0. All other settings of the NF, RF and NN remain the same as defined in subsection 7.1.1. The noise is randomly generated from the normal distribution and takes on values ranging from 0.1 to 0.5. Each value denotes standard deviation of normal distribution that generates noise.

If noise occurs among the data, the RF model in general has the tendency towards overfitting on this noise, especially when the individual trees are overgrown (large depth) and unpruned. We depict the dependence of decision boundaries on the noise in Figure A16.

We can see that in the case of $\beta = 1$, all NF models establish a fine separation of data despite the increasing amount of noise, whereas the RF model produces signs of overfitting (demonstrated by small areas cut out in the territory of different class). In this case, the overfitting of the RF model was not simulated by the NF models and all NF models and the NN model produced sufficiently generalized boundaries.

A different situation arises when we increase both transition parameters to $\beta = \beta_1 = \beta_2 = 50$. Then we could see significant signs of overfitting among all NF models, especially in the case of NRF_DW and NRF_EL_DW, which simulate the RF accurately and thus share similar properties as the RF. This problem should be resolved by decreasing the transition parameters or adopting some regularization techniques, as will be briefly investigated in the next part.

7.2.4 Overfitting

When initializing NF models with the RF consisting of a number of overgrown trees, there exists a significant risk that the RF overfits the data and therefore even a subsequent NF model could be prone to overfitting as well, especially if the transition parameters are larger than 1. One way to avoid this problem is to decrease the transition parameters or add a regularization term to the loss function, where the overfitting should be controlled by modifying the regularization parameter λ.

We demonstrate the benefit of using L2 regularization term in the case of NRF_DW and NRF_EL_DW models initialized with the RF suffering from overfitting.

MODEL	**HYPERPARAMETERS**
RF	Number of estimators = 2 Maximum depth = 20 Num. of features used for split = 2
NRF_DW	Number of epochs = 30 Mini-batch size = 10 Learning rate $\eta = 0.0035$ Initial random forest = RF Transition parameter $\beta_1 = 10$ and $\beta_2 = 10$
NRF_EL_DW	Number of epochs = 30 Mini-batch size = 10 Learning rate $\eta = 0.0045$ Initial random forest = RF Transition parameter $\beta_1 = 10$ and $\beta_2 = 10$

Table 7.8: Choice of hyperparameters.

It is done on artificially created dataset consisting of 100 instances and divided into 2 classes as depicted in Figure A17. Choices of hyperparameters of researched models are presented in Table 7.8. All other settings remain the same as defined in subsection 7.1.1. The results were obtained after 2 subsequent 5-fold cross validations (10 iterations in total) and are presented in Figure A17. Apparently, adding the regularization term is helpful in this case, resulting in an increase of accuracy among both NRF_DW and NRF_EL_DW models. Unfortunately, the most suitable value of the

regularization constant cannot be stated generally and depends on the specific data and model, so it should be picked after a grid search.

7.3 LUCAS dataset experiments

In this part, we focus on applying the proposed NF models on the real-world dataset, which was obtained from the data collected by the LUCAS project[9]. This project focuses on patients diagnosed with bronchogenic carcinoma (lung cancer) and their treatment and examination. Unfortunately, the diagnosis of lung cancer belongs to the most severe and complicated cancer diseases. In the Czech Republic alone, approximately 5400 patients a year die of this diagnosis, which ensures its sad primacy among all other tumor diseases.

The LUCAS project was developed to thoroughly describe and analyze the complex path of such patients through the treatment process already from the diagnosis and therefore to provide subsequent analytical studies evaluating the scope and structure of the overall care, pharmacoeconomic background, overall survival and the sequence and efficiency of the treatments. All of these outcomes could help to enhance the diagnostic and treatment system and therefore could ensure superior care about patients diagnosed with lung cancer.

This project was launched on 1.6.2018 and already includes data of 2285 patients (in the period from 1.6.2018 to 14.5.2020). The data contain various information about the individual patients, including diagnostic records, personal characteristics, treatment records and information about patient's progress. The data are collected from seven medical centers in the Czech Republic - FN Olomouc, FN Hradec Králové, FN Brno, FN Motol, FN Plzeň, Nemocnice Na Bulovce Praha and Thomayerova nemocnice Praha.

The treatment options have widely expanded in the last couple of years and right now they include chemotherapy, radiotherapy, immunotherapy, targeted therapy and operational interventions. Since there are only few tumors that could be removed by operation intervention, the other treatments are gaining great importance, especially targeted therapy and immunotherapy, which may prolong the patient's life by several years. But still, chemotherapy is a more common choice and is often accepted as the preferred choice in the first line treatment (the first treatment after diagnosis).

7.3.1 Classification task and data

What is, in addition to an early diagnosis, very important for patient's overall survival, is the very first treatment after diagnosis. Based on the available data of the LUCAS project, this first treatment (among pharmacotherapy) is usually chemotherapy, but with regard to the previously mentioned, it could sometimes be beneficial to replace chemotherapy (if the patient and illness have the required properties) with other kinds of treatment, especially with targeted treatment and immunotherapy. It could also be used in combination with the chemotherapy.

We test the ability of our NF models to predict the first line treatment based on the patient's diagnostic and personal records. We define binary classification task with a negative class (class 0) symbolizing chemotherapy and a positive class (class 1) symbolizing all other treatments belonging to the pharmacotherapy (especially targeted therapy and immunotherapy). The solution of this

[9] More information available at http://www.lucascz.cz/

classification task could help doctors decide on the choice of the first line treatment and potentially highlight the possibility of employing other treatments besides chemotherapy.

Overall, we have 1133 records with any form of pharmacotherapy in the first line treatment. There are 937 cases of chemotherapy (class 0) and 196 cases of other treatment (class 1). We could notice that this dataset is significantly imbalanced with the negative instances to positive instances ratio equal approximately to 7:1. This makes this classification task challenging for many classifiers.

Unfortunately, according to the non-disclosure agreement, we cannot describe the dataset in full detail, so we at least provide a shallow description. The dataset consists of 21 features, which were picked from the diagnostic and personal data of individual patients. There are 6 numerical features including for instance the patient's height and weight, the number of years of smoking, etc. Other 15 features are categorical and comprise for instance symptoms, details of diagnosis, tumor type, tumor stage, etc. All categorical features were one-hot encoded, resulting, together with numerical features, in 91 final features. Finally, all features were standardized (normalized to mean value equal to 0 and standard deviation equal to 1).

7.3.2 Models and evaluation

MODEL	HYPERPARAMETERS
LR	-
RF	Number of estimators = 30 Maximum depth = 6 Num. of features used for split = 9
NN	Mini-batch size = 10 Learning rate $\eta = 0.01$ Number of neurons in 1^{st} hidd. layer = 150 Number of neurons in 2^{nd} hidd. layer = 150
NRF_DW	Mini-batch size = 10 Learning rate $\eta = 0.002$ Initial random forest = RF Transition parameter $\beta_1 = 10$ and $\beta_2 = 10$
NRF_EL_DW	Mini-batch size = 10 Learning rate $\eta = 0.0025$ Initial random forest = RF Transition parameter $\beta_1 = 10$ and $\beta_2 = 10$

Table 7.9: Choice of hyperparameters.

Evaluation

Since our classification task corresponds to the binary classification on imbalanced dataset, we evaluate the performance mainly with respect to the precision and recall of the positive class. Moreover, we present the comparison of individual models with respect to the precision-recall (PR) curves and also according to the average precision (AP) score as defined in section 2.2. Eventually, we present the macro-average of F1-score and accuracy as well. We conducted 2 subsequent 5-fold cross validations (10 iterations in total) and we present the final results in the form "mean±std", similarly as in the public experiments. To all individual iterations of single training - testing pairs

correspond one PR curve and we visualize all iterations in one graph for each classification model. Additionally, we draw aggregation PR curves as defined in section 2.2. We do not measure the AP of the aggregation PR curve and instead we present mean and standard deviation of individual AP values.

Models

We employ two NF models - NRF_DW and NRF_EL_DW - and compare their performance with logistic regression (LR) classifier, the single NN and the RF used for initialization of NF models. We use the same logistic regression model with L2 regularization as in subsection 7.1.1. In the case of NF models and the single NN, we employ early stopping technique to limit overfitting. It tracks the loss function for 10 subsequent epochs starting from each new epoch, and if there is no occurrence of decrease, the training process is terminated. All other settings of all models remain the same as defined in subsection 7.1.1. The testing fold in each iteration of cross validation was randomly divided into two halves, one used as validation dataset for early stopping and the rest for testing. Regularization parameter λ is set to 0.

7.3.3 Results

The results of individual models are presented in Table 7.10 (original values multiplied by 100).

MODEL	Class 0			Class 1			Macro-avg F1	Accuracy
	Precision	Recall	F1-score	Precision	Recall	F1-score		
LR	89.6 ± 1.9	96.3 ± 1.8	92.8 ± 1.3	72.1 ± 13.7	46.6 ± 12.1	56.0 ± 12.0	74.4 ± 6.6	87.6 ± 6.6
NN	88.6 ± 2.1	93.4 ± 2.7	90.9 ± 2.0	58.3 ± 15.3	43.0 ± 12.4	49.0 ± 12.6	69.9 ± 7.2	84.6 ± 3.4
RF	88.9 ± 1.7	98.7 ± 1.0	93.5 ± 0.9	87.0 ± 10.5	41.0 ± 11.4	54.8 ± 12.1	74.2 ± 6.4	88.7 ± 1.7
NRF_DW	89.6 ± 1.7	98.3 ± 1.0	93.7 ± 1.0	84.7 ± 7.8	45.7 ± 10.4	58.7 ± 10.5	76.2 ± 5.7	89.1 ± 1.8
NRF_EL_DW	89.8 ± 1.8	98.4 ± 0.9	93.9 ± 1.1	85.5 ± 7.4	46.6 ± 11.4	59.6 ± 11.3	76.8 ± 6.1	89.4 ± 2.0

Table 7.10: LUCAS dataset results.

We can see that except for the NN model, all other classification models performed quite similarly on this dataset with respect to all researched evaluation metrics. Considering the overall performance, NRF_DW and NRF_EL_DW reached slightly better results than the other models (mainly higher F1-score on the positive class), especially surpassing and enhancing the initialization RF model. It turned out that simulating the RF model by adjusting transition parameters to high values was a suitable choice, since the results with lower transition parameters ($\beta_1 = 1$ and $\beta_2 = 1$) were not that favourable. For $\beta_1 = 1$ and $\beta_2 = 1$, the macro-average of F1-score for NRF_DW was 74.1 ± 6.2 and for NRF_EL_DW, it was 74.0 ± 5.7. This mild decrease suggests that this classification task suits better the RF model than the NN, which is further supported by the unsatisfying results of the single NN model in comparison with other models. Therefore, it was beneficial to simulate the behaviour of the RF model by increasing transition parameters.

We may notice that in the case of the positive class, the standard deviations of all results are high. This variance could be caused mainly by the low number of testing instances of the positive class (approximately only 16 positive testing instances per one training - testing run). Therefore, we compare the models also on the level of individual training - testing runs and not only by mean values, since this could lead to a biased interpretation. We present PR curves with the

corresponding average precision scores (AP) for all training - testing runs (10 iterations in total) and also one aggregation PR curve in Figure A18.

If we compare the individual runs, we can see that in most cases, NRF_DW and NRF_EL_DW models reached higher AP values in comparison with other models. We can also notice the resemblance of the PR curves corresponding to the NF models and the PR curves corresponding to the initialization RF model, which is also the result of high transition parameters. In the summary graph in Figure 7.1, it is visible that the aggregation PR curves of NRF_DW, NRF_EL_DW and RF are similar in shape as well.

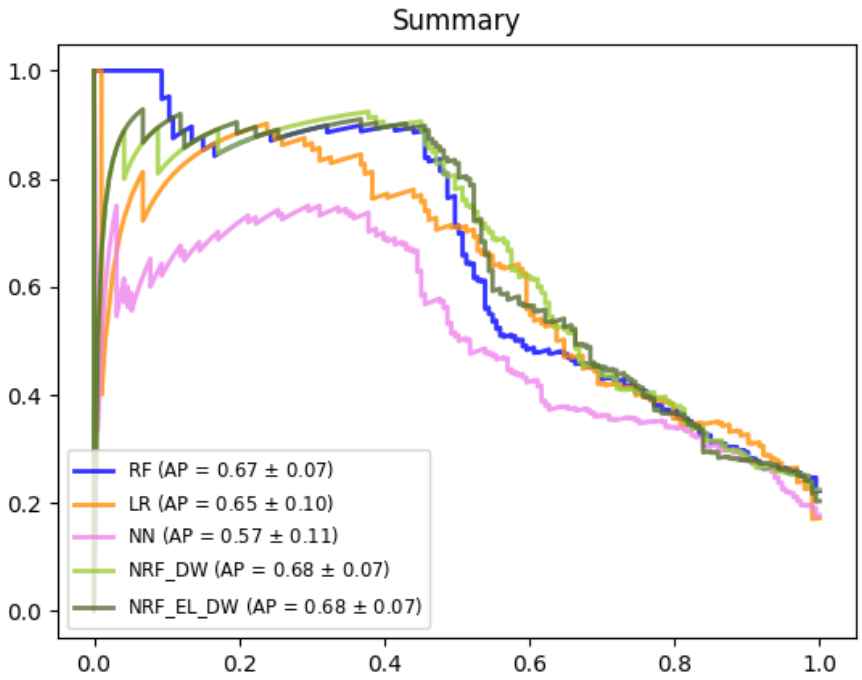


Figure 7.1: Visualizations of individual aggregation PR curves. The AP values correspond to the "mean±std" of all iterations.

In the conducted experiments of binary classification on the LUCAS dataset were only slight differences between the individual models, but the NF models showed the most favourable performance. On the contrary, the NN model could not achieve a competitive performance with other models, which could stand as a warning and the reason to increase the transition parameters in our NF models.

Chapter 8

Conclusion

This thesis focused mainly on the research of the relationship between random forest and feedforward neural network classifiers in terms of transformation of a single decision tree into a neural network. We provided detailed theoretical descriptions of related domains and proposed four different hybrid classifiers belonging to the group of neural random forests.

We compared the performance of all proposed models to each other and also to the performance of an initialization random forest, a single neural network and logistic regression. Firstly, the performance was tested on several public datasets. All neural random forest models reported a favourable performance in comparison with other competitive models with respect to various evaluation methods. The most favourable performance among all models was reported by neural random forests NRF_DW and NRF_EL_DW. We also investigated the effect of neural random forest parameters on the performance and the inclination to overfitting. Interestingly, we found a correlation between the initialization random forest and the neural random forest, when the transition parameters β_1 and β_2 are sufficiently high.

Certain similarities between random forests and neural random forests were detected especially in the case of NRF_DW and NRF_EL_DW models. This is further supported by illustrations of decision boundaries on 2D toy datasets, where we can see a strong resemblance between the decision boundary of the random forest and the neural random forest, when the transition parameters are increased. Also, NRF_DW and NRF_EL_DW models needed a lower number of training epochs to converge to a reasonable solution than concurrent models NRF and NRF_EL, especially in the case of datasets with fewer data instances.

Finally, we tested the performance of NRF_DW and NRF_EL_DW models on the real dataset, consisting of medical data obtained by the LUCAS project. This dataset has 91 features and 1133 instances divided into 2 classes, which represent different kinds of treatment. There is a significant imbalance between the number of positive instances and the number of negative instances. Even in this case, the performance of neural random forest models was more favourable than the performance of other competitive models. This case also higlighted the benefit of increasing the transition parameters, since the random forest model produced better results than an examined single neural network.

Neural random forests showed a good classification ability with an ambition to overcome other classic models. Even though the time and space complexity are higher than in the case of an initialization random forest or a single neural network of a similar architecture as individual estimators,

the training time can be significantly reduced by using parallelization and possibly acceleration on graphical processing units (GPU). If the machine possesses a sufficient number of computational units, the training time of an entire ensemble equals roughly the training time of a single decision tree and a single feedforward neural network, which makes neural random forests well-applicable even in the case of large datasets.

Bibliography

[1] WANG, Suhang, AGGARWAL, Charu and LIU, Huan, 2018. Random-Forest-Inspired Neural Networks. ACM Transactions on Intelligent Systems and Technology [online]. 15 November 2018. Vol. 9, no. 6p. 1-25. DOI 10.1145/3232230.
Available at: https://dl.acm.org/doi/10.1145/3232230

[2] KONTSCHIEDER, Peter, FITERAU, Madalina, CRIMINISI, Antonio and BULO, Samuel Rota, 2015. Deep Neural Decision Forests. In: 2015 IEEE International Conference on Computer Vision (ICCV) [online]. IEEE. 2015. p. 1467-1475. ISBN 978-1-4673-8391-2.
Available at: http://ieeexplore.ieee.org/document/7410529/

[3] BIAU, Gérard, WELBL, Johannes and SCORNET, Erwan, 2016. Neural Random Forest. [online]. 2016. Available at: arxiv.org/abs/1604.07143

[4] SAMMUT, Claude and WEBB, Geoffrey I. (eds.), 2010. Encyclopedia of Machine Learning. Boston, MA: Springer US. ISBN 978-0-387-30768-8.

[5] ROKACH, Lior and MAIMON, Oded. Data Mining and Knowledge Discovery Handbook. ISBN 978-0-387-24435-8.

[6] CANBEK, Gurol, SAGIROGLU, Seref, TEMIZEL, Tugba Taskaya and BAYKAL, Nazife, 2017. Binary classification performance measures/metrics: A comprehensive visualized roadmap to gain new insights. In: 2017 International Conference on Computer Science and Engineering (UBMK) [online]. IEEE. 2017. p. 821-826. ISBN 978-1-5386-0930-9.
Available at: http://ieeexplore.ieee.org/document/8093539/

[7] KOYEJO, Oluwasanmi, NATARAJAN, Nagarajan, RAVIKUMAR, Pradeep and DHILLON, Inderjit S., 2014. Consistent Binary Classification with Generalized Performance Metrics. Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2. 2014. P. 2744–2752.

[8] HOSSIN, M. and SULAIMAN, M.N, 2015. A Review on Evaluation Metrics for Data Classification Evaluations. International Journal of Data Mining & Knowledge Management Process [online]. 31 March 2015. Vol. 5, no. 2p. 01-11. DOI 10.5121/ijdkp.2015.5201.
Available at: http://www.aircconline.com/ijdkp/V5N2/5215ijdkp01.pdf

[9] BRANCO, Paula, TORGO, Luís and RIBEIRO, Rita P., 2017. Relevance-Based Evaluation Metrics for Multi-class Imbalanced Domains. In: Advances in Knowledge Discovery and Data Mining [online]. Cham: Springer International Publishing. p. 698-710. Lecture Notes in Computer

Science. ISBN 978-3-319-57453-0.
Available at: http://link.springer.com/10.1007/978-3-319-57454-7_54

[10] CHAWLA, Nitesh V., 2005. Data Mining for Imbalanced Datasets: An Overview. In: Data Mining and Knowledge Discovery Handbook [online]. New York: Springer-Verlag. p. 853-867. ISBN 0-387-24435-2.
Available at: http://link.springer.com/10.1007/0-387-25465-X_40

[11] HAIBO, HE and GARCIA, E.A., 2009. Learning from Imbalanced Data.
IEEE Transactions on Knowledge and Data Engineering [online]. 2009. Vol. 21, no. 9p. 1263-1284. DOI 10.1109/TKDE.2008.239.
Available at: http://ieeexplore.ieee.org/document/5128907/

[12] AKOSA, Josephine Sarpong. Predictive Accuracy : A Misleading Performance Measure for Highly Imbalanced Data. 2017. Available at:
https://support.sas.com/resources/papers/proceedings17/0942-2017.pdf

[13] FAWCETT, Tom, 2006. An introduction to ROC analysis. Pattern Recognition Letters [online]. 2006. Vol. 27, no. 8p. 861-874. DOI 10.1016/j.patrec.2005.10.010.
Available at: https://linkinghub.elsevier.com/retrieve/pii/S016786550500303X

[14] MAJNIK, Matjaž and BOSNIĆ, Zoran, 2013. ROC analysis of classifiers in machine learning: A survey. Intelligent Data Analysis [online]. 16 May 2013. Vol. 17, no. 3p. 531-558. DOI 10.3233/IDA-130592. Available at:
https://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/IDA-130592

[15] DAVIS, Jesse and GOADRICH, Mark, 2006. The relationship between Precision-Recall and ROC curves. In: Proceedings of the 23rd international conference on Machine learning - ICML '06 [online]. New York, New York, USA: ACM Press. 2006. p. 233-240. ISBN 1595933832.
Available at: http://portal.acm.org/citation.cfm?doid=1143844.1143874

[16] BERNARD, S., ADAM, S. and HEUTTE, L., 2007. Using Random Forests for Handwritten Digit Recognition. In: Ninth International Conference on Document Analysis and Recognition (ICDAR 2007) Vol 2 [online]. IEEE. 2007. p. 1043-1047. ISBN 0-7695-2822-8.
Available at: http://ieeexplore.ieee.org/document/4377074/

[17] DÍAZ-URIARTE, Ramón and ALVAREZ DE ANDRÉS, Sara. Gene selection and classification of microarray data using random forest. BMC Bioinformatics [online]. Vol. 7, no. 1. DOI 10.1186/1471-2105-7-3.
Available at: http://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-7-3

[18] FARNAAZ, Nabila and JABBAR, M.A., 2016. Random Forest Modeling for Network Intrusion Detection System. Procedia Computer Science [online]. 2016. Vol. 89, p. 213-217. DOI 10.1016/j.procs.2016.06.047.
Available at: https://linkinghub.elsevier.com/retrieve/pii/S1877050916311127

[19] OHAREK, Martin and BUKÁČEK, Marek. Analysis of the Ancient Egyptian Society with Utilization of Decision Tree. In: SPMS 2019 – Stochastic and Physical Monitoring Systems, Proceedings of the international conference. ISBN 978-80-01-06659-1.

[20] MAALOUF, Maher, 2011. Logistic regression in data analysis: an overview. International Journal of Data Analysis Techniques and Strategies [online]. 2011. Vol. 3, no. 3. DOI 10.1504/IJDATS.2011.041335.
Available at: http://www.inderscience.com/link.php?id=41335

[21] AWAD, Mariette and KHANNA, Rahul, 2015. Support Vector Machines for Classification. In: AWAD, Mariette and KHANNA, Rahul, Efficient Learning Machines [online]. Berkeley, CA: Apress. p. 39-66. ISBN 978-1-4302-5989-3.
Available at: http://link.springer.com/10.1007/978-1-4302-5990-9_3

[22] JEHAD, Ali, REHANULLAH, Khan, NASIR, Ahmad and IMRAN, Maqsood. Random Forests and Decision Trees. IJCSI International Journal of Computer Science Issues [online].
Available at: http://ijcsi.org/papers/IJCSI-9-5-3-272-278.pdf

[23] CRIMINISI, Antonio, 2011. Decision Forests: A Unified Framework for Classification, Regression, Density Estimation, Manifold Learning and Semi-Supervised Learning. Foundations and Trends® in Computer Graphics and Vision [online]. 2011. Vol. 7, no. 2-3p. 81-227. DOI 10.1561/0600000035.
Available at: http://www.nowpublishers.com/article/Details/CGV-035

[24] KORTING, Thales Sehn. C4.5 algorithm and Multivariate Decision Trees [online].
Available at: https://www.academia.edu/1983952/C4._5_algorithm_and_Multivariate_Decision_Trees

[25] BREIMAN, Leo, FRIEDMAN, J. H., OLSHEN, R. A. and STONE, C. J., 1984. Classification and Regression Trees. Monterey, CA: Wadsworth and Brooks. ISBN 0-534-98053-8

[26] XIAOHU, Wang, LELE, Wang and NIANFENG, Li, 2012. An Application of Decision Tree Based on ID3. Physics Procedia [online]. 2012. Vol. 25, p. 1017-1021. DOI 10.1016/j.phpro.2012.03.193.
Available at: https://linkinghub.elsevier.com/retrieve/pii/S1875389212006098

[27] ZHOU, Xinlei and YAN, Dasen, 2019. Model tree pruning. International Journal of Machine Learning and Cybernetics [online]. 2019. Vol. 10, no. 12p. 3431-3444. DOI 10.1007/s13042-019-00930-9.
Available at: http://link.springer.com/10.1007/s13042-019-00930-9

[28] ESPOSITO, F., MALERBA, D., SEMERARO, G. and KAY, J. A comparative analysis of methods for pruning decision trees. IEEE Transactions on Pattern Analysis and Machine Intelligence [online]. Vol. 19, no. 5p. 476-493. DOI 10.1109/34.589207.
Available at: http://ieeexplore.ieee.org/document/589207/

[29] DIETTERICH, Thomas G., 2000. Ensemble Methods in Machine Learning. In: Multiple Classifier Systems [online]. Berlin, Heidelberg: Springer Berlin Heidelberg. p. 1-15. Lecture Notes in Computer Science. ISBN 978-3-540-67704-8.
Available at: http://link.springer.com/10.1007/3-540-45014-9_1

[30] VAISHNAVI, S, 2017. Ensemble Methods and Random Forests. [online]. 2017. Available at: https://courses.engr.illinois.edu/ece543/sp2017/projects/Vaishnavi%20Subramanian.pdf

[31] CHENGSHENG, Tu, HUACHENG, Liu, BING, Xu, XU, Bing and CHEN, Yinong, 2017. AdaBoost typical Algorithm and its application research. MATEC Web of Conferences [online]. 2017. Vol. 139. DOI 10.1051/matecconf/201713900222.
Available at: http://www.matec-conferences.org/10.1051/matecconf/201713900222

[32] MCCULLOCH, Warren S. and PITTS, Walter, 1943. A logical calculus of the ideas immanent in nervous activity. The Bulletin of Mathematical Biophysics [online]. 1943. Vol. 5, no. 4p. 115-133. DOI 10.1007/BF02478259.
Available at: http://link.springer.com/10.1007/BF02478259

[33] MINSKY, Marvin and PAPERT, Seymour, [2017]. Perceptrons: an introduction to computational geometry. [2017 edition]. Cambridge, MA: The MIT Press. ISBN 978-0262534772.

[34] WERBOS, Paul John, 1974. Beyond regression : new tools for prediction and analysis in the behavioral sciences. PhD thesis.

[35] RUMELHART, David E., HINTON, Geoffrey E. and WILLIAMS, Ronald J., 1986. Learning representations by back-propagating errors. Nature [online]. 1986. Vol. 323, no. 6088p. 533-536. DOI 10.1038/323533a0.
Available at: http://www.nature.com/articles/323533a0

[36] KRIESEL, David. A Brief Introduction to Neural Networks [online]. Available at: http://www.dkriesel.com//_media/science/neuronalenetze-en-zeta2-2col-dkrieselcom.pdf

[37] HORDRI, Hordri Nur, YUHANIZ, Siti and SHAMSUDDIN, Siti Mariyam, 2016. Deep Learning and Its Applications: A Review.

[38] NIELSEN, Michael. Neural Networks and Deep Learning [online].
Available at: http://static.latexstudio.net/article/2018/0912/neuralnetworksanddeeplearning.pdf

[39] CYBENKO, G., 1989. Approximation by superpositions of a sigmoidal function. Mathematics of Control, Signals, and Systems [online]. 1989. Vol. 2, no. 4p. 303-314. DOI 10.1007/BF02551274.
Available at: http://link.springer.com/10.1007/BF02551274

[40] LIU, Tianyi, CHEN, Minshuo, ZHOU, Mo, DU, Simon S., ZHOU, Enlu and ZHAO, Tuo. Towards Understanding the Importance of Shortcut Connections in Residual Networks. [online].
Available at: https://arxiv.org/pdf/1909.04653.pdf

[41] NWANKPA, Chigozie Enyinna, IJOMAH, Winifred, GACHAGAN, Anthony and MARSHALL, Stephen. Activation Functions: Comparison of Trends in Practice and Research for Deep Learning. [online].
Available at: https://arxiv.org/pdf/1811.03378.pdf

[42] KARLIK, Bekir and OLGAC, A Vehbi. Performance Analysis of Various Activation Functions in Generalized MLP Architectures of Neural Networks. International Journal of Artificial Intelligence And Expert Systems (IJAE) [online]. Available at:
https://www.cscjournals.org/manuscript/Journals/IJAE/Volume1/Issue4/IJAE-26.pdf

[43] NAIR, Vinod and HINTON, Geoffrey E. Rectified Linear Units Improve Restricted Boltzmann Machines. [online].
Available at: https://www.cs.toronto.edu/ fritz/absps/reluICML.pdf

[44] PEDAMONTI, Dabal. Comparison of non-linear activation functions for deep neural networks on MNIST classification task. [online].
Available at: https://arxiv.org/pdf/1804.02763.pdf

[45] BOYD, Stephen P. and VANDENBERGHE, Lieven, 2004. Convex optimization. Cambridge: Cambridge University Press. ISBN 0521833787.

[46] RUDER, Sebastian. An overview of gradient descent optimization algorithms. [online].
Available at: https://arxiv.org/abs/1609.04747v2

[47] BA, Jimmy and KINGMA, Diederik P. Adam: A Method for Stochastic Optimization. [online].
Available at: https://arxiv.org/abs/1412.6980

[48] MAKIN, J.G. Backpropagation. [online].
Available at: http://www.cs.cornell.edu/courses/cs5740/2016sp/resources/backprop.pdf

[49] SADOWSKI, Petr. Notes on Backpropagation. [online].
Available at: https://www.ics.uci.edu/ pjsadows/notes.pdf

[50] GOLIK, P., DOETSCH, P. and HEY, N. Cross-Entropy vs. Squared Error Training: a Theoretical and Experimental Comparison. Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH 2013. 1756-1760.

[51] JANOCHA, Katarzyna and CZARNECKI, Wojciech Marian, 2017. On Loss Functions for Deep Neural Networks in Classification. Schedae Informaticae [online]. 2017. Vol. 1/2016. DOI 10.4467/20838476SI.16.004.6185.
Available at: http://www.ejournals.eu/Schedae-Informaticae/2016/Volume-25/art/9009/

[52] GRAESSER, L. Regularization for Neural Networks. [online].
Available at: https://pdfs.semanticscholar.org/0e9f/88e8f47b7a1b73d9090f0da3f0e5c2bec9aa.pdf

[53] Anonymous authors. FIXING WEIGHT DECAY REGULARIZATION IN ADAM. [online].
Available at: https://openreview.net/pdf?id=rk6qdGgCZ

[54] BALDI, Pierre and SADOWSKI, Peter, 2013. Understanding Dropout. In: Advances in Neural Information Processing Systems 26. Curran Associates. 2013.

[55] PEDREGOSA, Fabian et al. Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research [online]. P. 2825-2830.
Available at: http://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf

[56] CHOLLET, Francois. Keras. [online].
Available at: https://keras.io/

Appendices

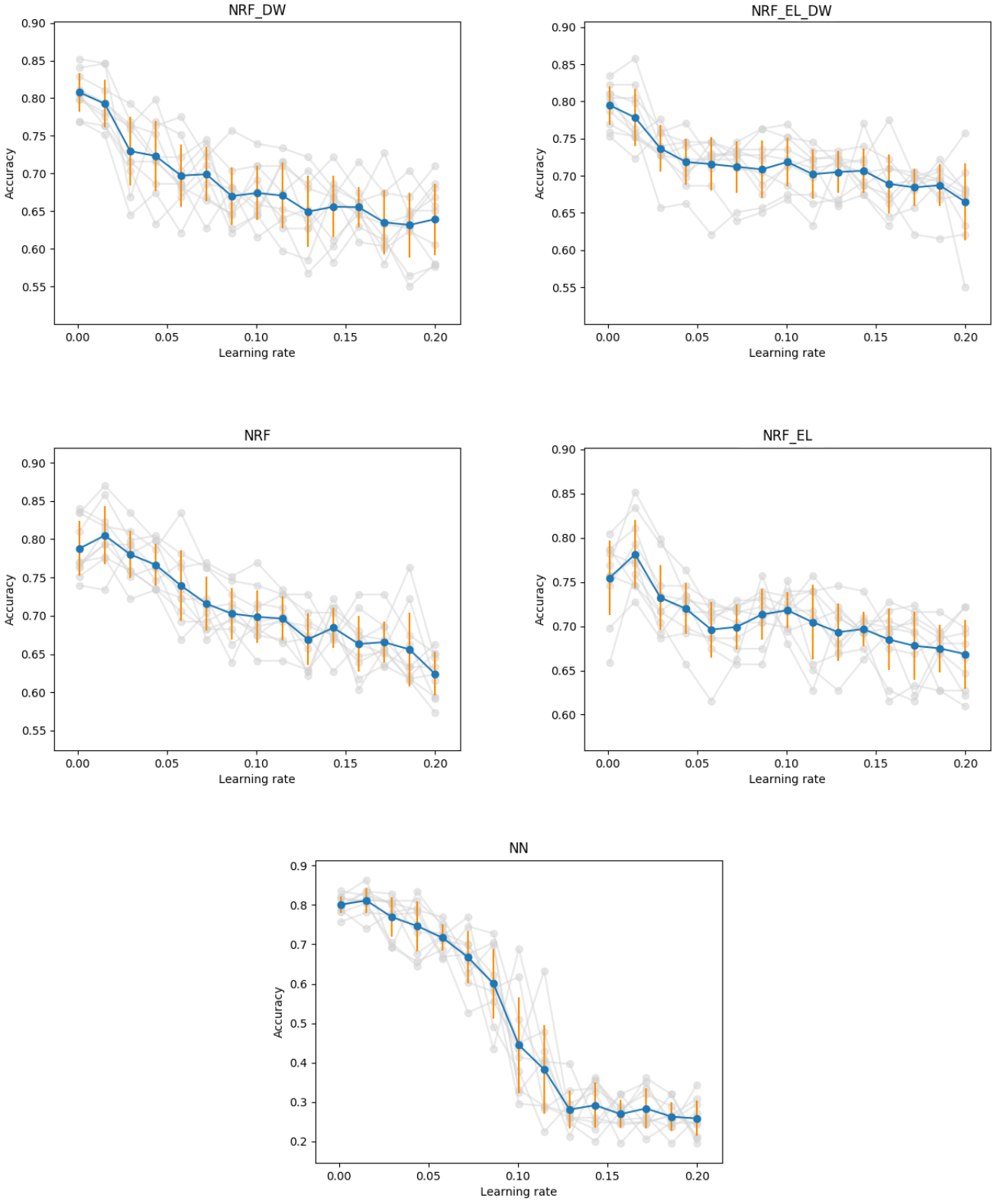


Figure A1: The effect of a varying learning rate on the Vehicle dataset. The blue line illustrates mean value and the orange lines symbolize standard deviation. The dimmed gray lines are single realizations. The experiment is described on page 69.

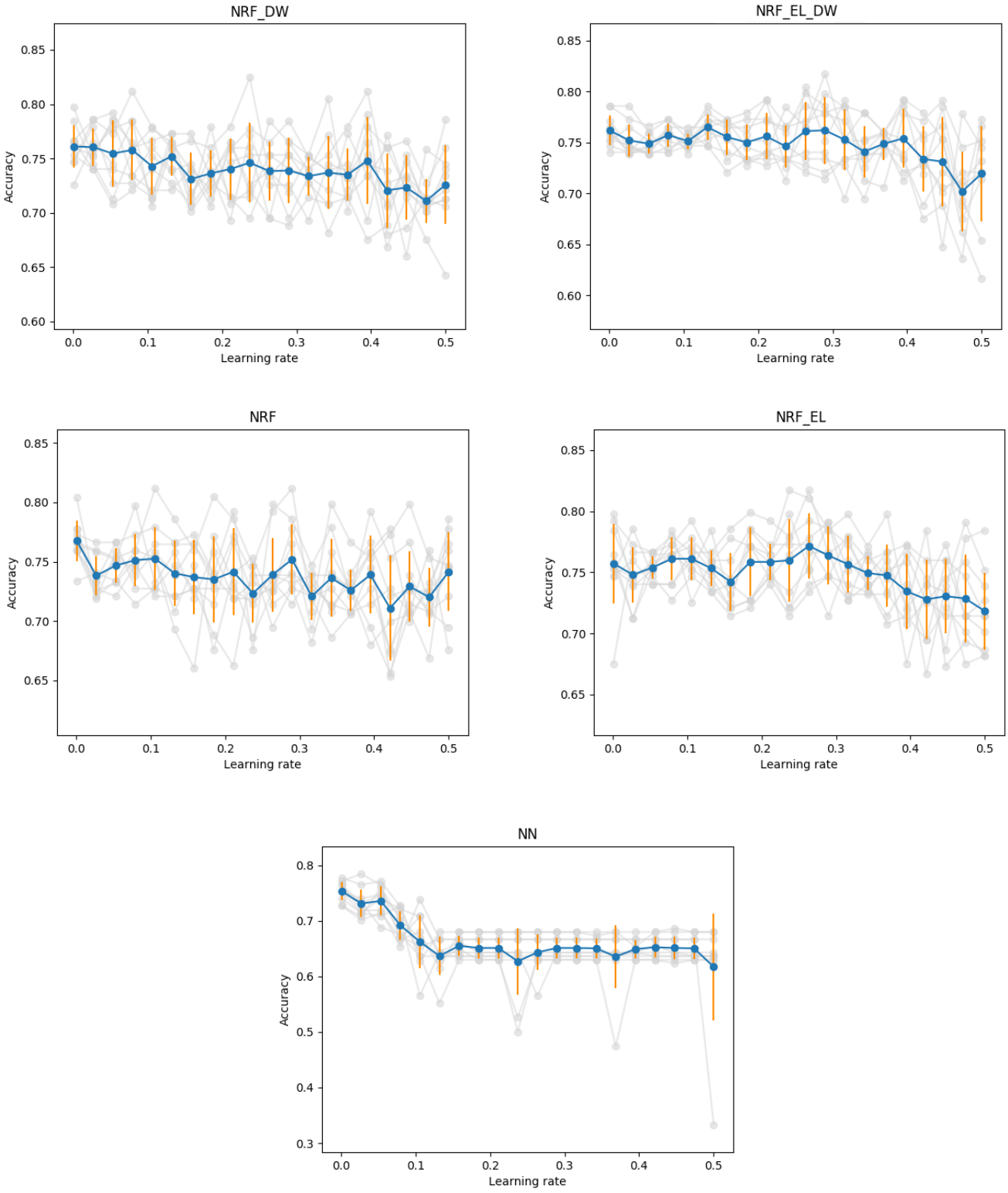


Figure A2: The effect of a varying learning rate on the Diabetes dataset. The blue line illustrates mean value and the orange lines symbolize standard deviation. The dimmed gray lines are single realizations. The experiment is described on page 69.

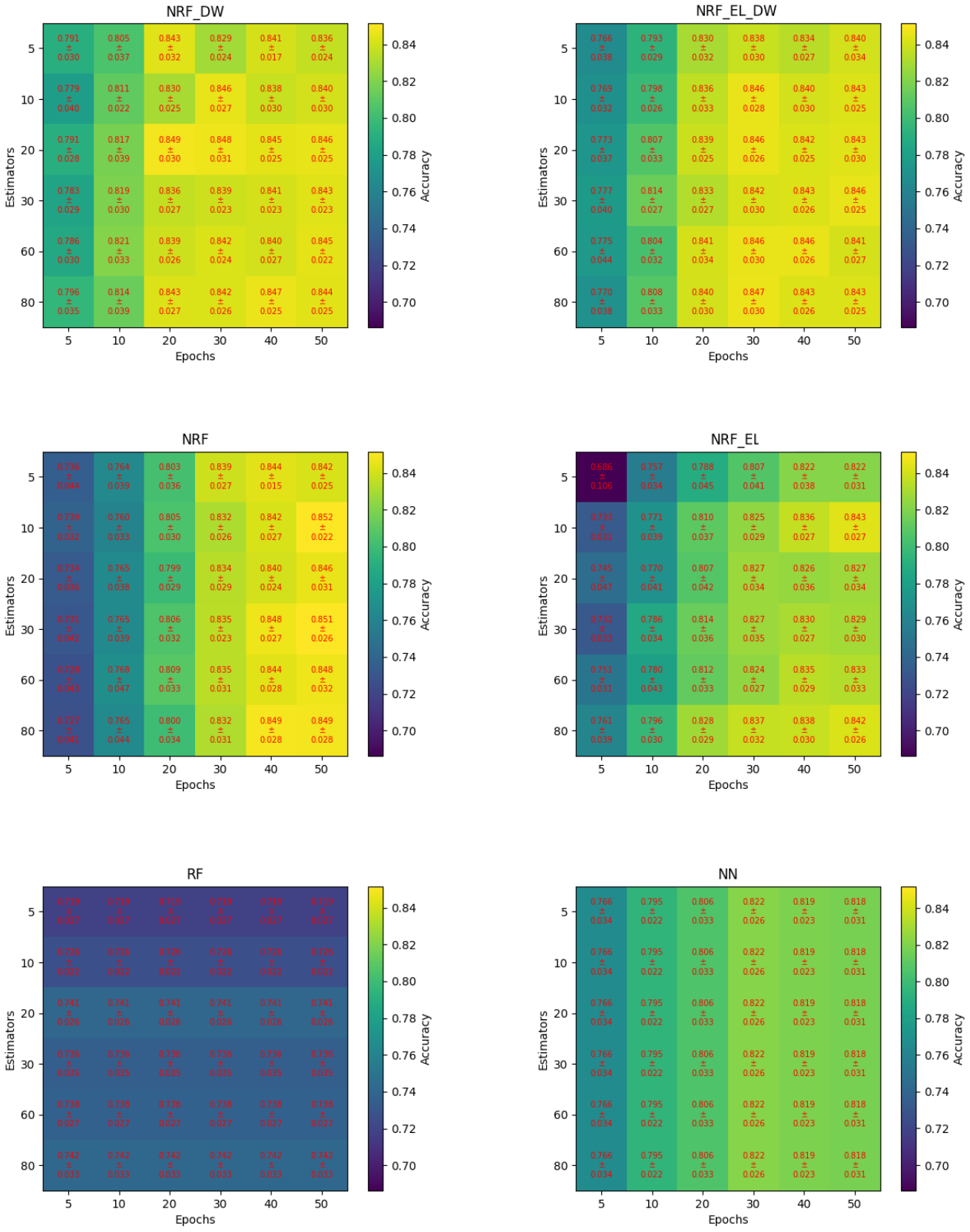


Figure A3: The dependence of accuracy on the number of estimators and the number of training epochs on the Vehicle dataset. The experiment is described on page 69.

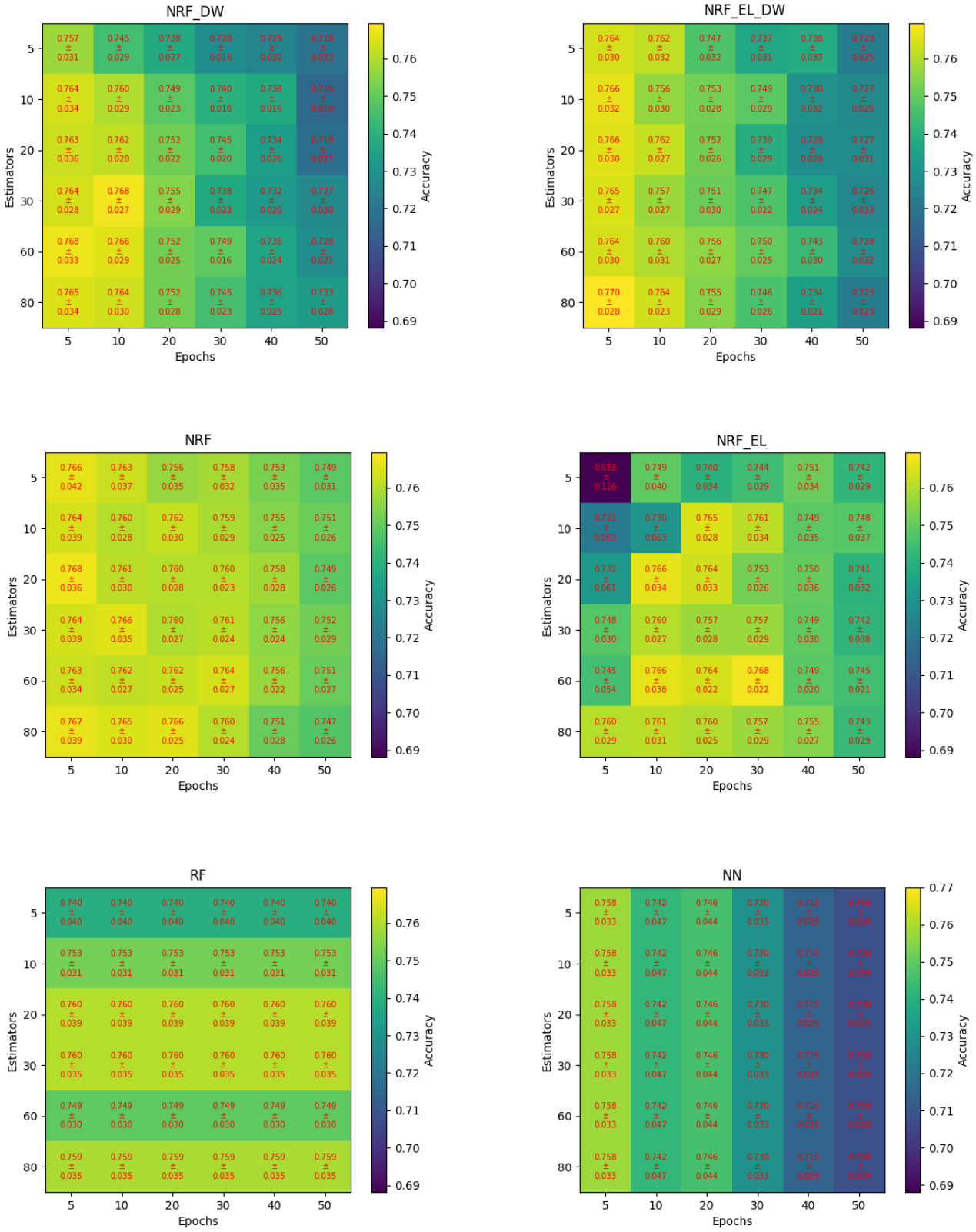


Figure A4: The dependence of accuracy on the number of estimators and the number of training epochs on the Diabetes dataset. The experiment is described on page 69.

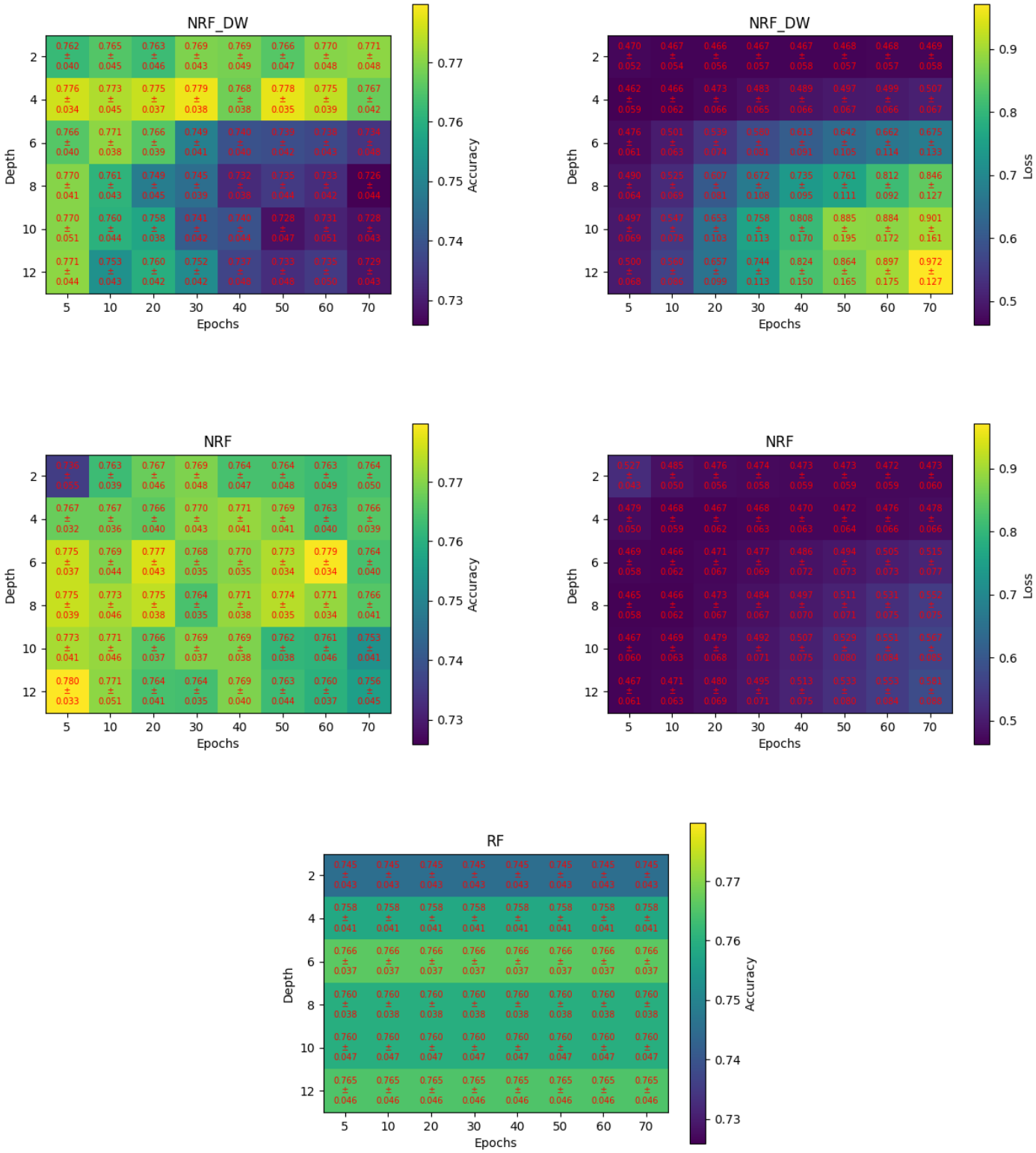


Figure A5: The dependence of accuracy and loss function on the maximum depth of the decision trees in the RF and the number of training epochs. This experiment is performed on the Diabetes dataset. The experiment is described on page 70.

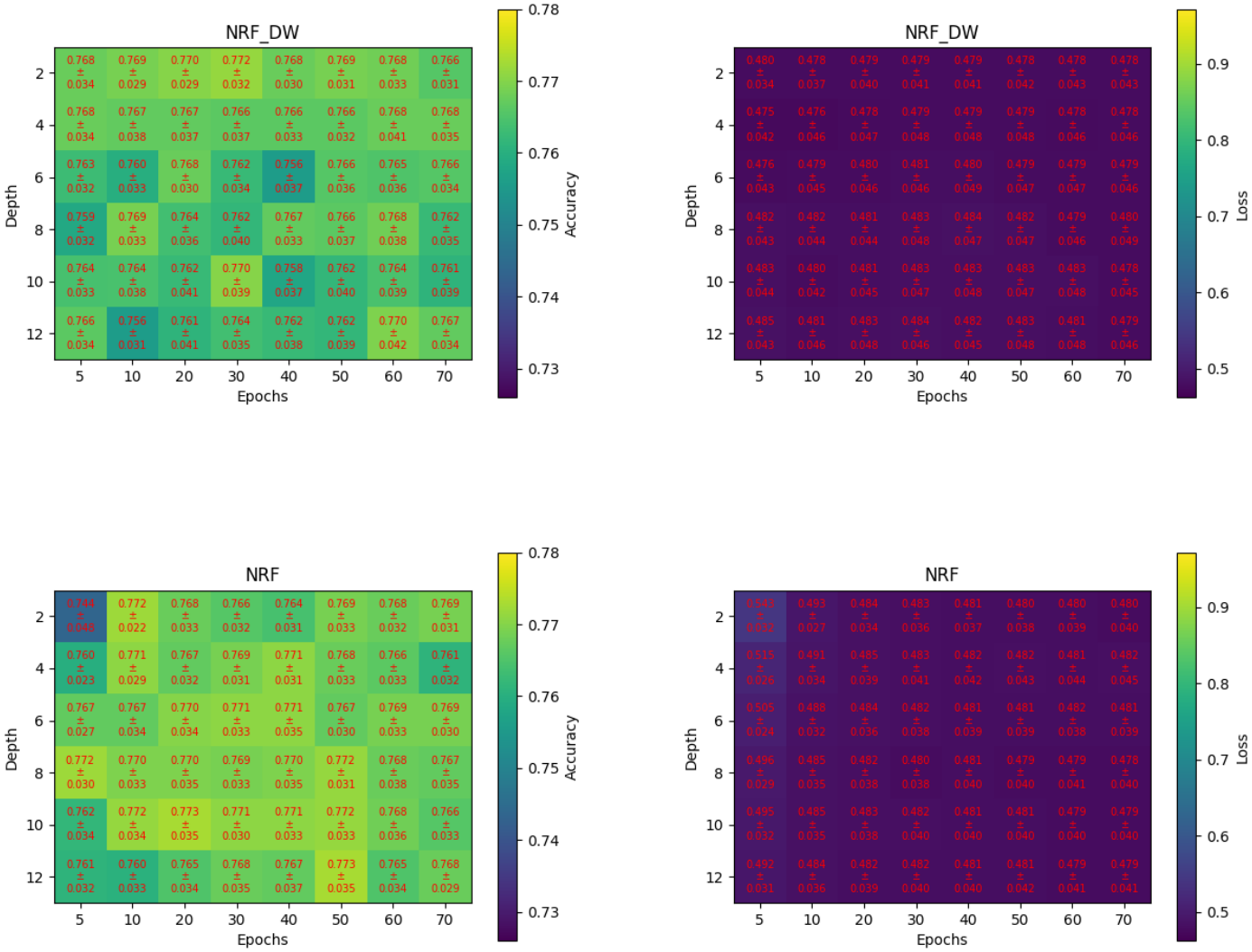


Figure A6: The same models as in Figure A5 with the regularization constant set to $\lambda = 10$. The experiment is described on page 70.

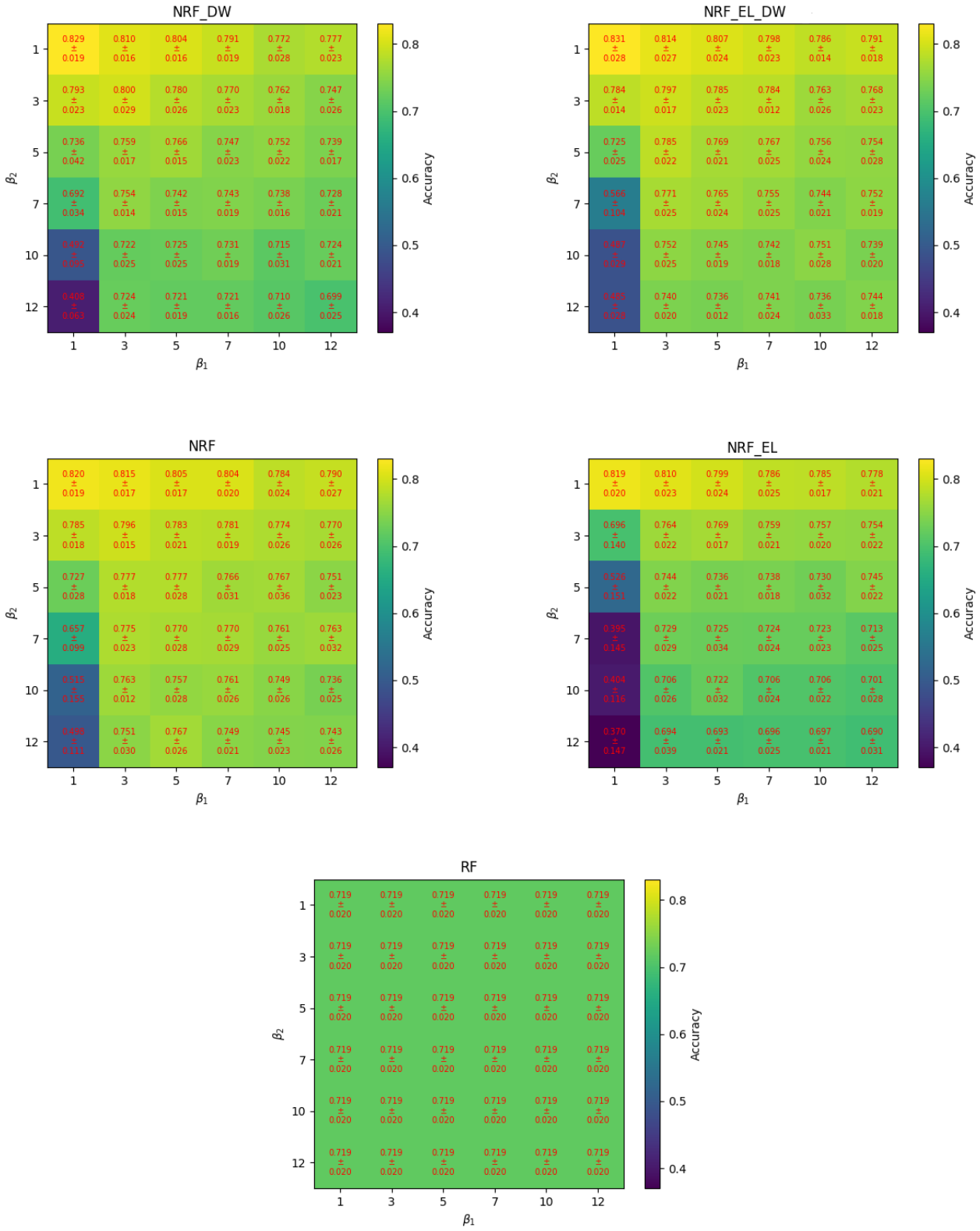


Figure A7: The dependence of accuracy on varying transition parameters β_1 and β_2 on the Vehicle dataset. The experiment is described on page 71.

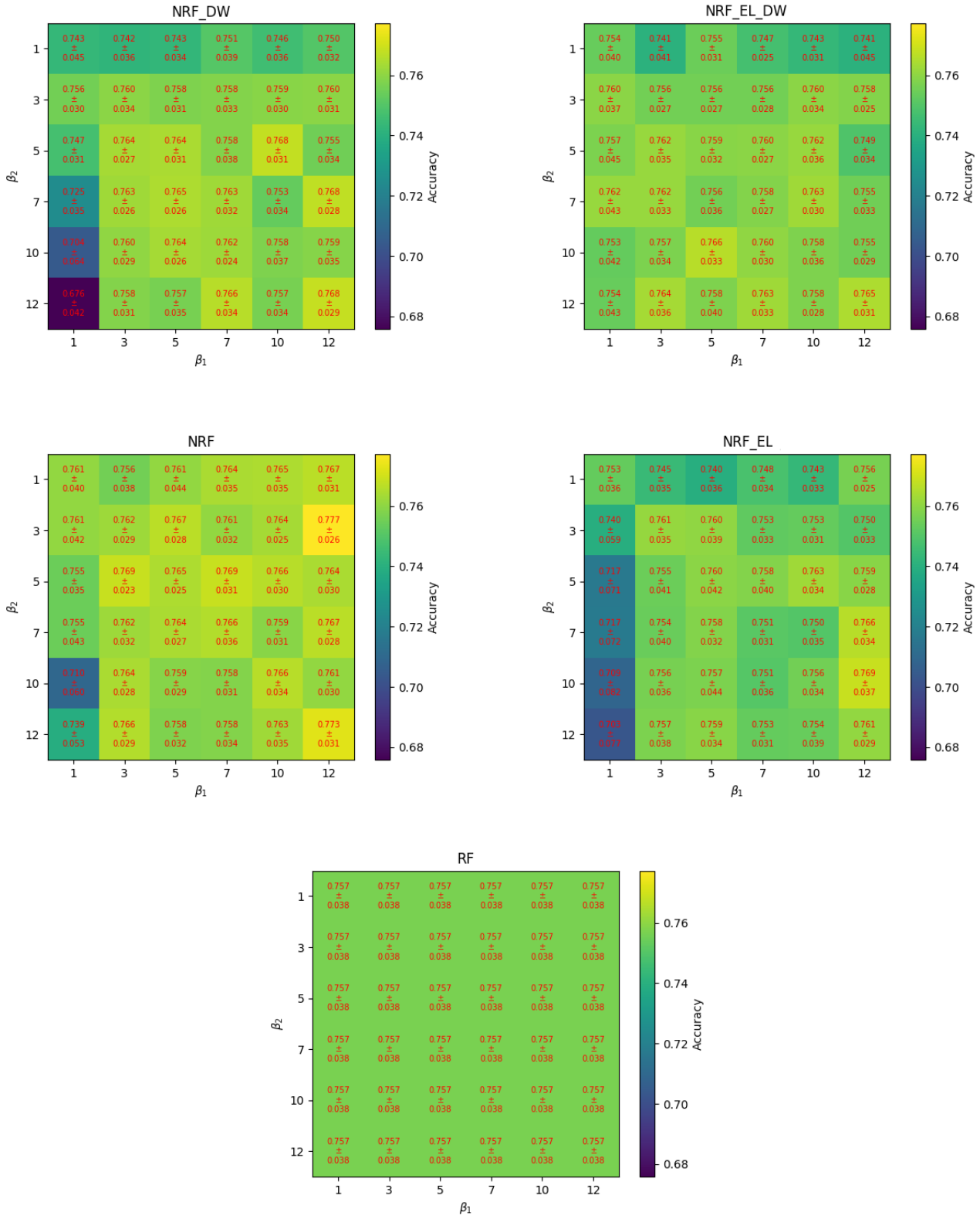


Figure A8: The dependence of accuracy on varying transition parameters β_1 and β_2 on the Diabetes dataset. The experiment is described on page 71.

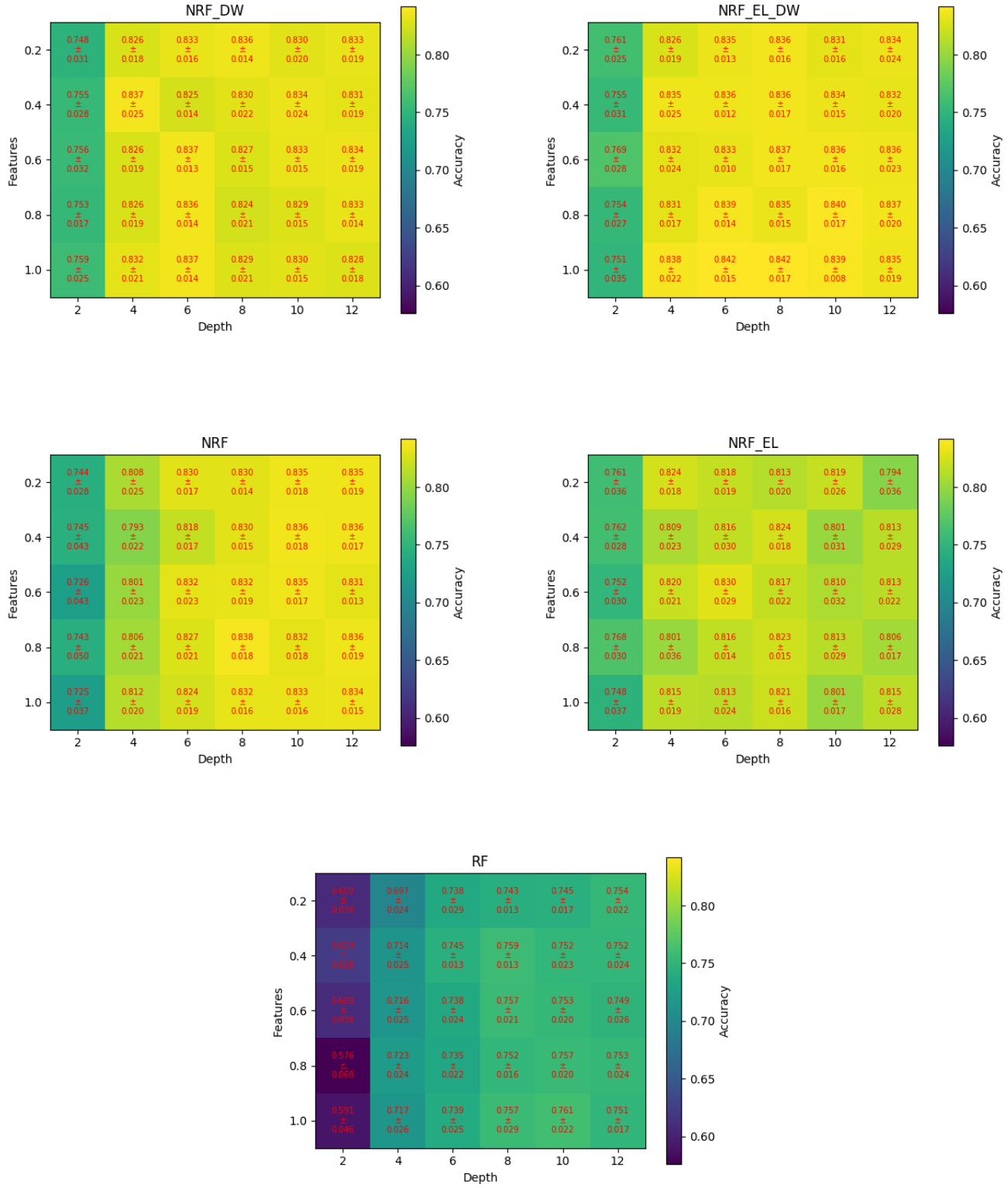


Figure A9: The ependence of accuracy on a varying number of features used in split on the Vehicle dataset. The experiment is described on page 72.

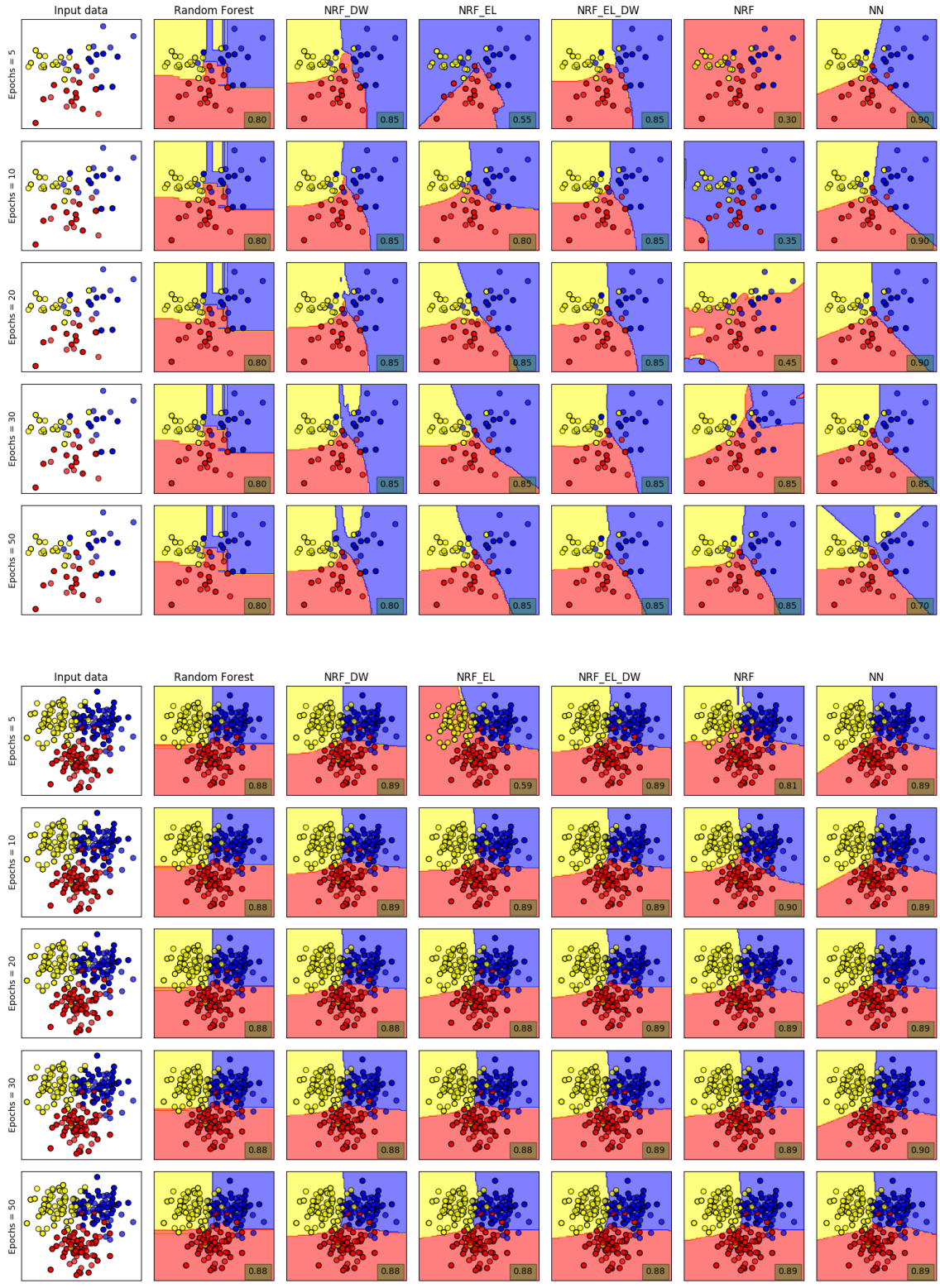


Figure A10: Type 1. Decision boundaries based on a varying number of training epochs. The sizes of the datasets are 50 and 200 in the top-bottom manner. The accuracy on the testing subset is in the right-bottom corner. The experiment is described on page 73.

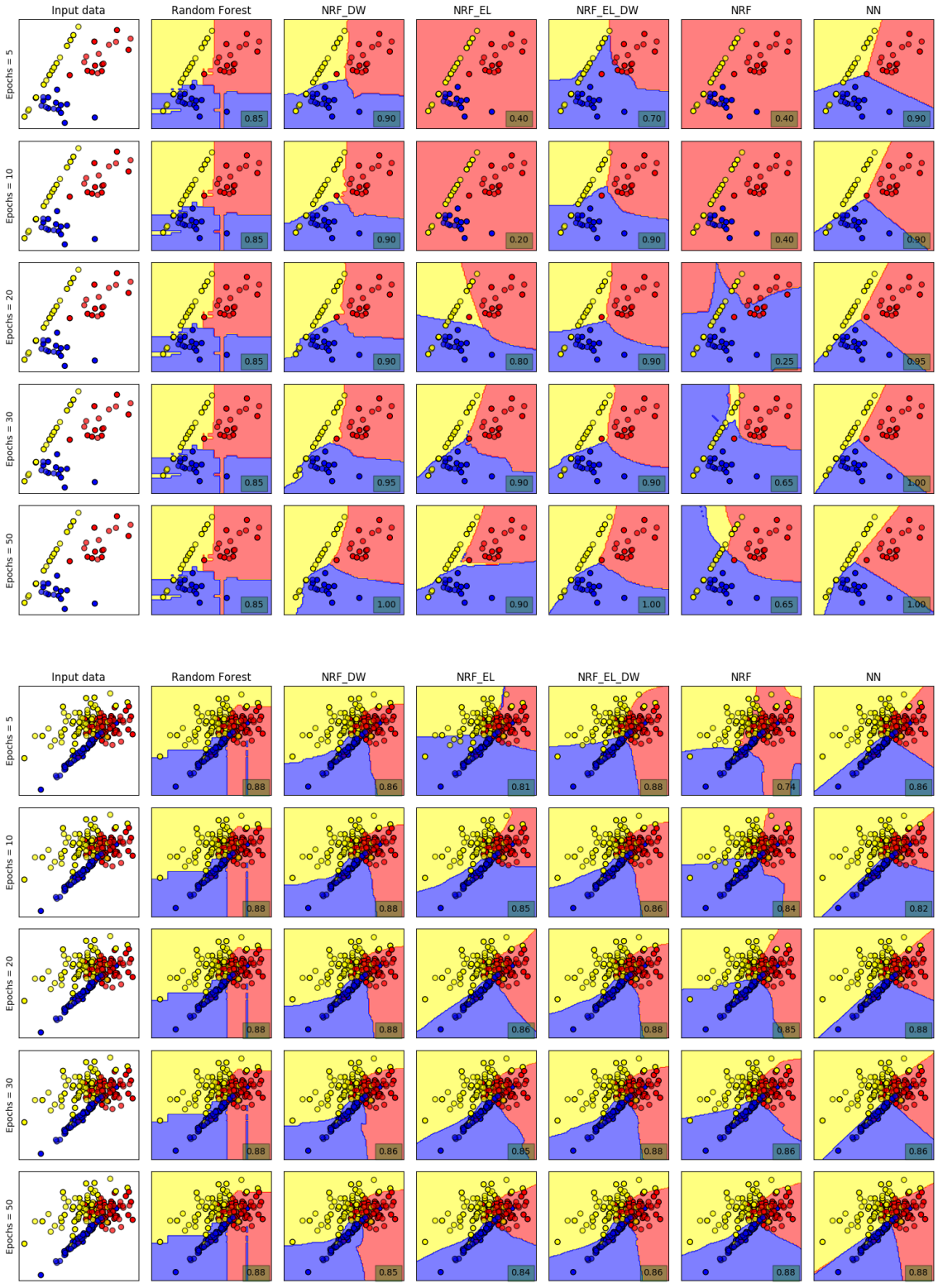


Figure A11: Type 2. Decision boundaries based on a varying number of training epochs. The sizes of the datasets are 50 and 200 in the top-bottom manner. The accuracy on the testing subset is in the right-bottom corner. The experiment is described on page 73.

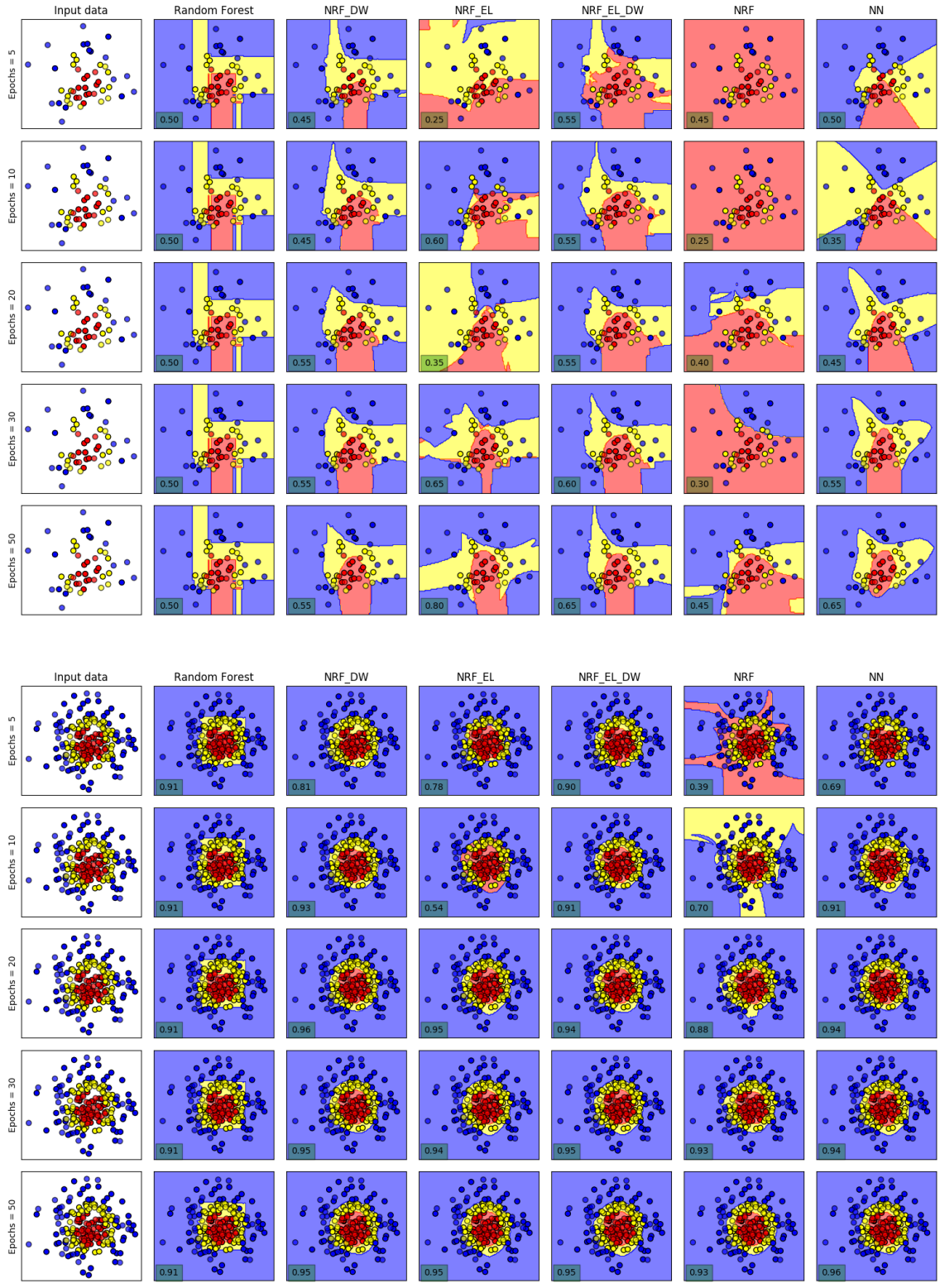


Figure A12: Type 3. Decision boundaries based on a varying number of training epochs. The sizes of the datasets are 50 and 200 in the top-bottom manner. The accuracy on the testing subset is in the left-bottom corner. The experiment is described on page 73.

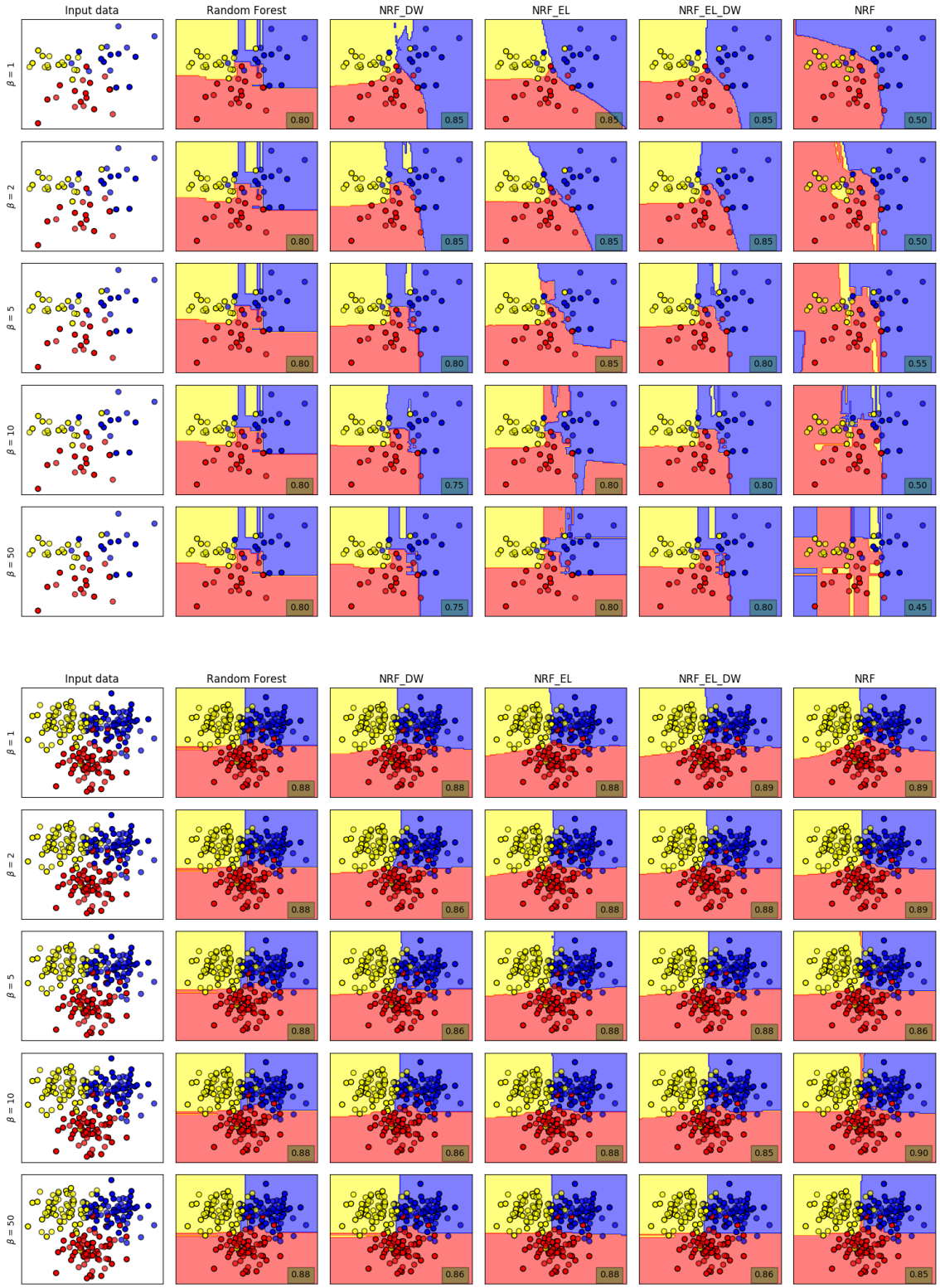


Figure A13: Type 1. Decision boundaries based on varying transition parameters, where $\beta = \beta_1 = \beta_2$. The sizes of the datasets are 50 and 200 in the top-bottom manner. The accuracy on the testing subset is in the right-bottom corner. The experiment is described on page 74.

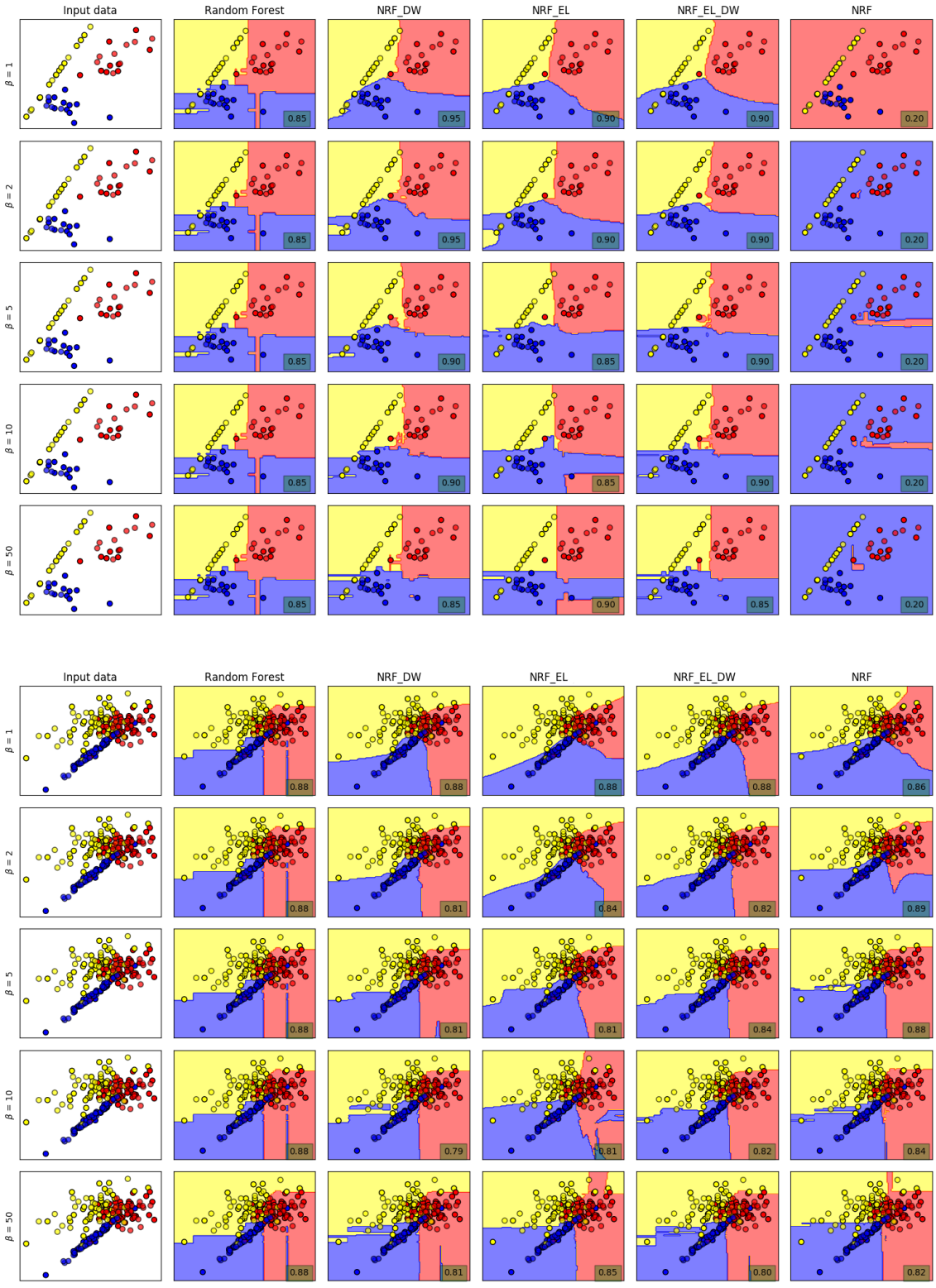


Figure A14: Type 2. Decision boundaries based on varying transition parameters, where $\beta = \beta_1 = \beta_2$. The sizes of the datasets are 50 and 200 in the top-bottom manner. The accuracy on the testing subset is in the right-bottom corner. The experiment is described on page 74.

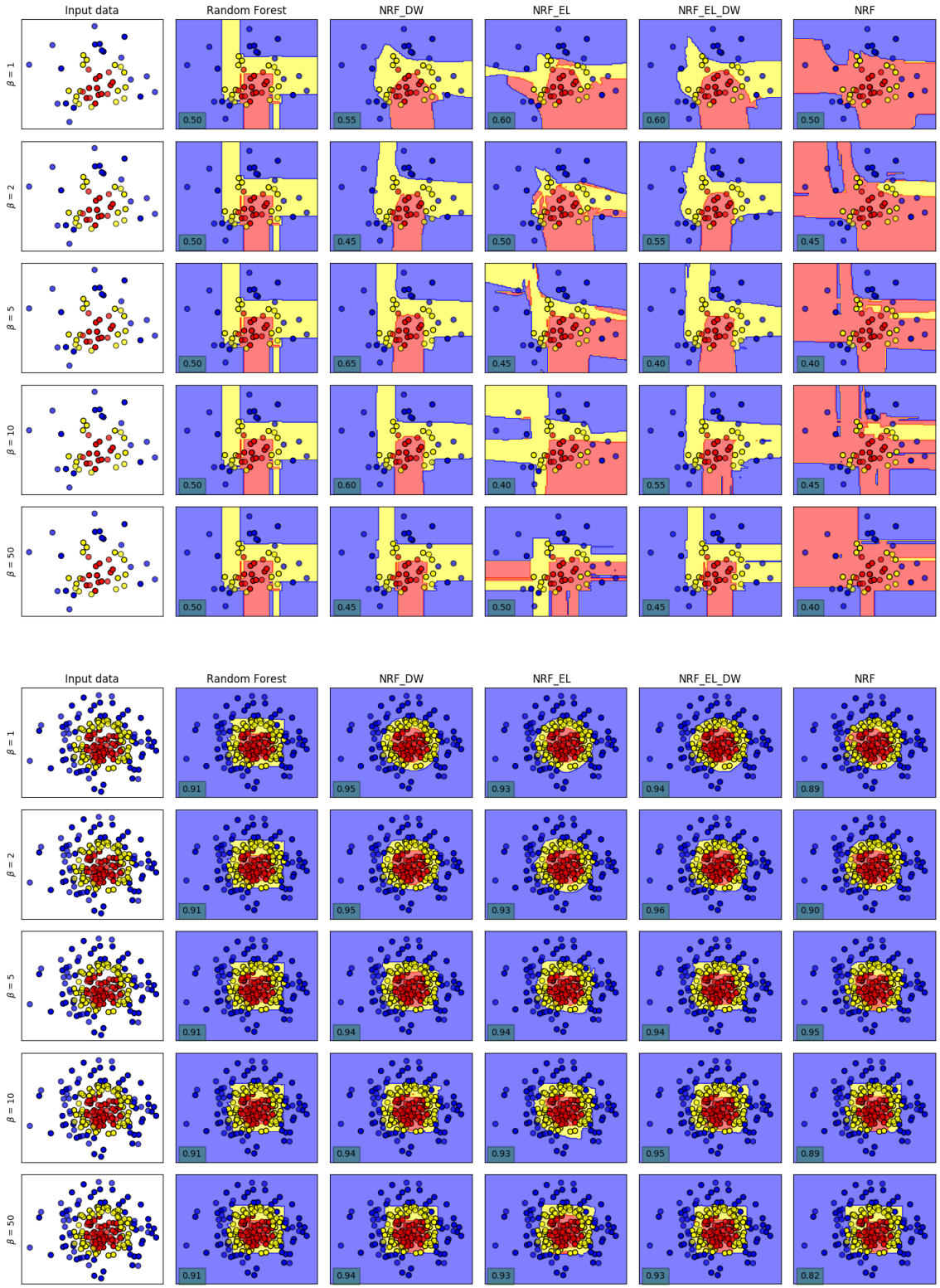


Figure A15: Type 3. Decision boundaries based on varying transition parameters, where $\beta = \beta_1 = \beta_2$. The sizes of the datasets are 50 and 200 in the top-bottom manner. The accuracy on the testing subset is in the left-bottom corner. The experiment is described on page 74.

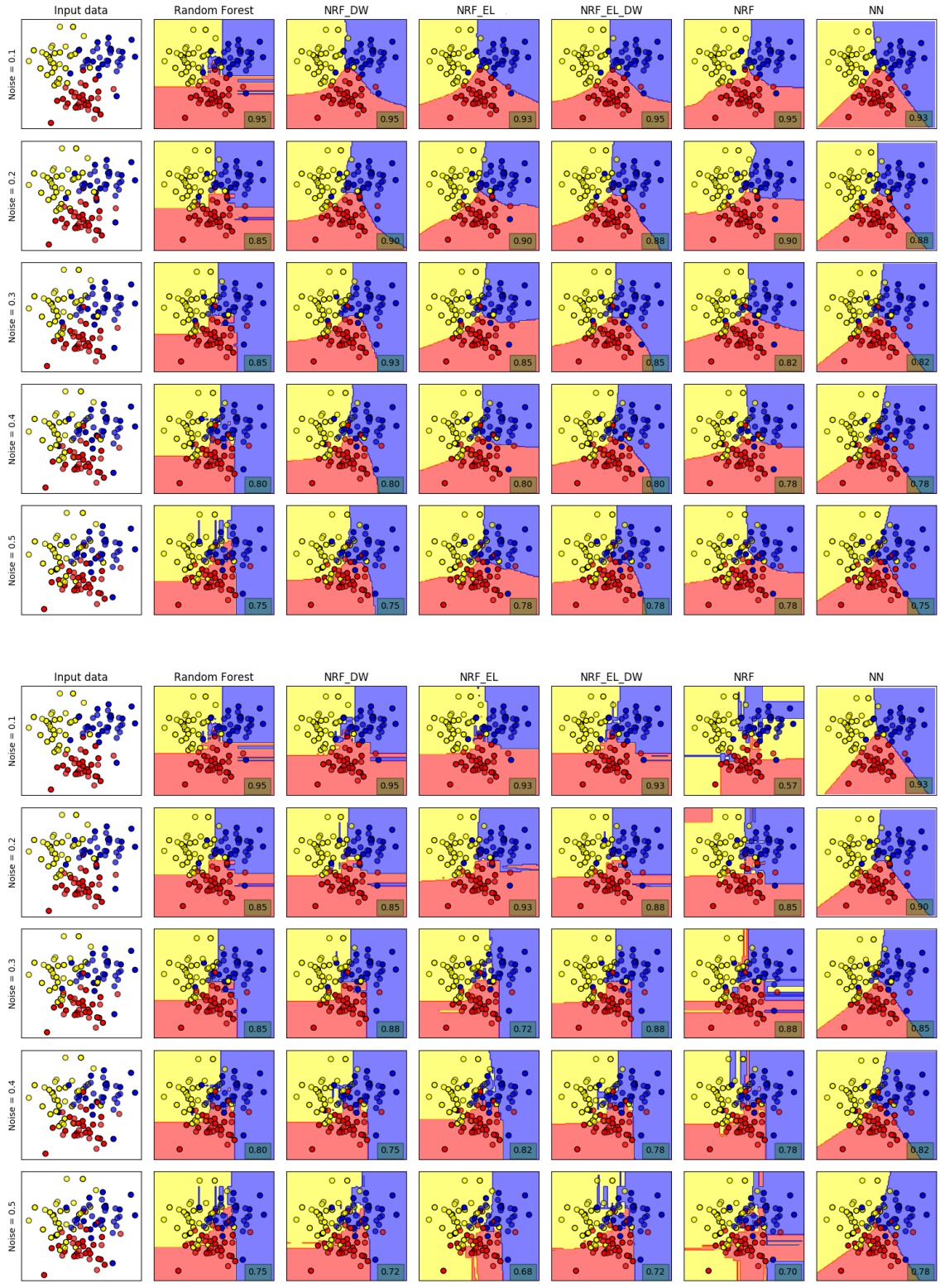


Figure A16: Decision boundaries based on the occurrence of noise in the data. The effect of noise is illustrated on a dataset with 100 instances. In the upper figure, there is $\beta_1 = \beta_2 = 1$ among all NF models. In the bottom figure, there is $\beta_1 = \beta_2 = 50$. The accuracy on the testing subset is in the right-bottom corner. The experiment is described on page 76.

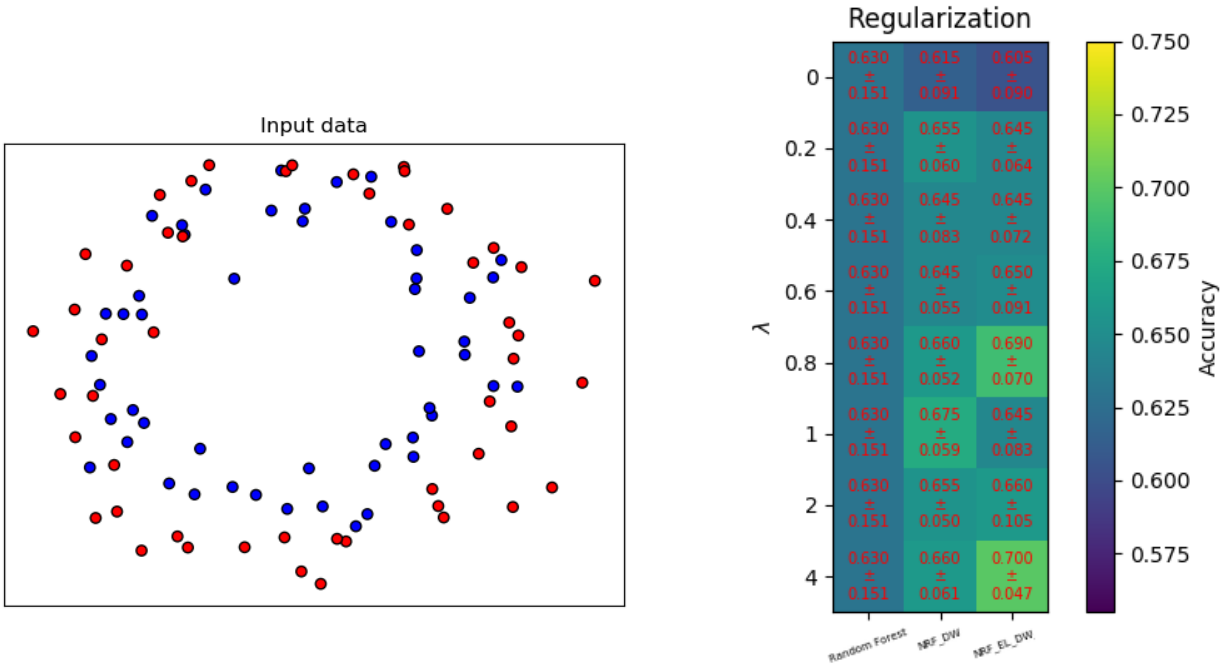


Figure A17: Accuracy depending on the regularization constant λ. The left figure shows the input data (100 instances) as two circles distorted by noise. The results are presented for the RF, NRF_DW and NRF_EL_DW in the form "mean ± std" after 2 subsequent 5-fold cross validations. The transition parameters β_1 and β_2 are set to 10. The experiment is described on page 77.

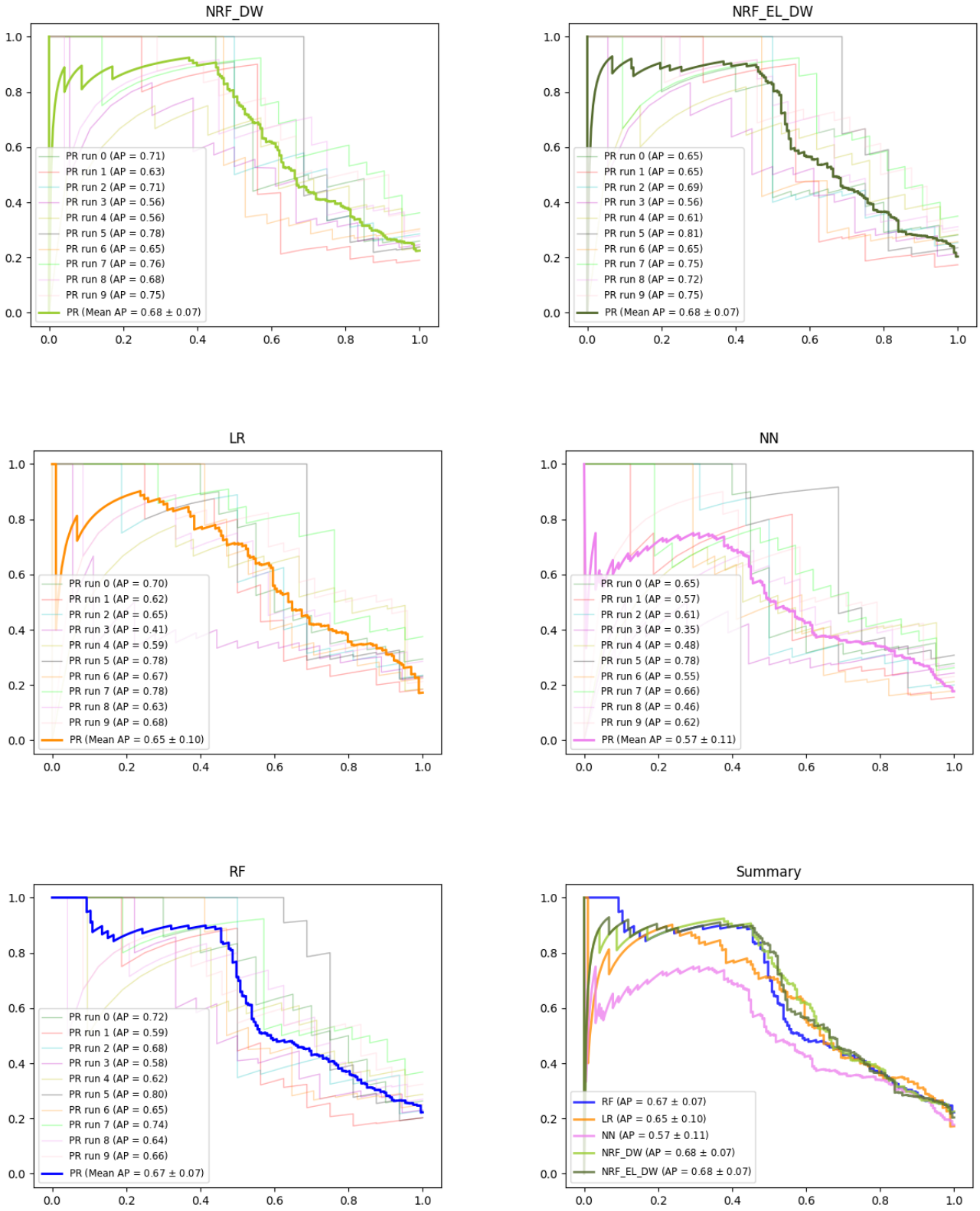


Figure A18: PR curves on LUCAS dataset. Each graph contains PR curves of individual training - testing runs with AP values and one highlighted aggregation PR curve with mean AP value and standard deviation of all iterations. In the summary, aggregation PR curves are visualized together. The experiment is described on page 80.

Proof 1:

$$\mathbb{A} \overset{\text{def}}{=} \begin{pmatrix} 1 & -1 & -1 & -1 & \dots & -1 \\ -1 & 1 & -1 & -1 & \dots & -1 \\ -1 & -1 & 1 & -1 & \dots & -1 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ -1 & -1 & \dots & -1 & 1 & -1 \\ -1 & -1 & \dots & -1 & -1 & 1 \end{pmatrix} \in \mathbb{R}^{L \times L}$$

$$\det\mathbb{A} = \begin{vmatrix} 1 & -1 & -1 & -1 & \dots & -1 \\ -1 & 1 & -1 & -1 & \dots & -1 \\ -1 & -1 & 1 & -1 & \dots & -1 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ -1 & -1 & \dots & -1 & 1 & -1 \\ -1 & -1 & \dots & -1 & -1 & 1 \end{vmatrix} \overset{(1)}{=} \begin{vmatrix} 1 & -1 & -1 & -1 & \dots & -1 \\ 0 & 0 & -2 & -2 & \dots & -2 \\ 0 & -2 & 0 & -2 & \dots & -2 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & -2 & \dots & -2 & 0 & -2 \\ 0 & -2 & \dots & -2 & -2 & 0 \end{vmatrix}$$

$$\overset{(2)}{=} (-1)^{1+1} \cdot 1 \cdot \begin{vmatrix} 0 & -2 & -2 & \dots & -2 \\ -2 & 0 & -2 & \dots & -2 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ -2 & \dots & -2 & 0 & -2 \\ -2 & \dots & -2 & -2 & 0 \end{vmatrix} \overset{(3)}{=} (-2)^{L-1} \cdot \begin{vmatrix} 0 & 1 & 1 & \dots & 1 \\ 1 & 0 & 1 & \dots & 1 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 1 & \dots & 1 & 0 & 1 \\ 1 & \dots & 1 & 1 & 0 \end{vmatrix}$$

$$\overset{(4)}{=} (-2)^{L-1} \cdot \begin{vmatrix} -1 & 0 & 0 & \dots & 0 & 1 \\ 0 & -1 & 0 & \dots & 0 & 1 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \\ 0 & \dots & 0 & -1 & 0 & 1 \\ 0 & \dots & 0 & 0 & -1 & 1 \\ 1 & \dots & \dots & 1 & 1 & 0 \end{vmatrix}$$

$$\overset{(5)}{=} (-2)^{L-1} \cdot \begin{vmatrix} -1 & 0 & 0 & \dots & 0 & 1 \\ 0 & -1 & 0 & \dots & 0 & 1 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \\ 0 & \dots & 0 & -1 & 0 & 1 \\ 0 & \dots & 0 & 0 & -1 & 1 \\ 0 & \dots & \dots & \dots & 0 & L-2 \end{vmatrix} \overset{(6)}{=} (-2)^{L-1} \cdot (-1)^{L-2} \cdot (L-2)$$

$$= (-1)^{2L-3} \cdot 2^{L-1} \cdot (L-2)$$

(1) Add the first row of the matrix to the others.
(2) Use Laplace expansion along the first column.
(3) Multiply all $L-1$ rows of minor matrix by $(-\frac{1}{2})$.
(4) Multiply the last row by (-1) and add it to all other rows.
(5) Add each row except the last one to the last row.
(6) The matrix is upper triangular, so the result is obtained by the multiplication of diagonal entries.