Training von gemischten Regel- und neuronalen Netzwerk-Klassifizierern

Training of mixed deep classifiers of symbolical rules and neural networks

Master-Thesis von Andreas Straninger aus Frankfurt am Main

Tag der Einreichung:

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Dr. Eneldo Loza Mencía

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Knowledge Engineering Group

Training von gemischten Regel- und neuronalen Netzwerk-Klassifizierern
Training of mixed deep classifiers of symbolical rules and neural networks

Vorgelegte Master-Thesis von Andreas Straninger aus Frankfurt am Main

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Dr. Eneldo Loza Mencía

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den December 13, 2016

(Andreas Straninger)

Abstract

This thesis deals with combining a decision list classifiers with neural networks in order to gain a mixed classification model, which can be further transformed into a pure multi-layered decision list classifier. The resulting classifier has the potential of increase the accuracy especially for deep network topologies. In contrast to neural networks, the rules in decision lists are better understandable by humans.

This thesis shows the relations of the mixed network to ensemble trainers (especially gradient boosting), fuzzy neural networks and other descriptive rule learning algorithms.

The contribution of this thesis is a method for performing an online training of neural networks and decision lists. For this purpose, a way of training decision lists with small portions of data (mini-batches) instead of complete data sets is introduced. Further contributions include the way of combining rule learning algorithms with neural networks similar to gradient boosting and a binary backpropagation scheme for symbolical methods.

During evaluation with the MNIST dataset, it is shown that existing classifiers like C4.5 and RIPPER benefit from the results of the decision list layers and can increase their performance. In comparison to existing approaches, a model with a reduced complexity can be obtained.

Zusammenfassung

Diese Abschlussarbeit befasst sich mit der Kombination von Decision-List-Klassifizierern und neuronalen Netzwerken mit dem Ziel einen gemischten Klassifizierer zu erstellen, welcher danach in einen reinen mehrschichtigen Decision-List-Klassifizierer überführt werden kann. Der daraus entstehende Klassifizierer hat das Potential, die Genauigkeit inbesondere in tiefen Netzwerktopologien zu erhöhen. Im Gegensatz zu neuronalen Netzwerken sind Regeln einer Decision List für Menschen besser verständlich.

Diese Abschlussarbeit zeigt die Ähnlichkeiten des gemischten Netzwerkes zu Ensembletrainern (besonders Gradient Boosting), fuzzy-neuronalen Netzwerken und weiteren deskriptive Regellern-Algorithmen auf.

Diese Abschlussarbeit stellt eine neue Methode vor, die ein Onlinetraining von neuronalen Netzwerken und Decision Lists ermöglicht. Zu diesem Zweck wird eine Trainingsmethode mit kleinen Datenstücken (mini-batches) vorgestellt. Weitere Beiträge sind die Kombinierung von Regellernalgorithment mit neuronalen Netzwerken ähnlich zu Gradient Boosting und eine binäre Backpropagation-Methode.

In der Evaluation mit dem MNIST-Datensatz wird gezeigt, dass Klassifizierer wie C4.5 und RIPPER von den Ergebnissen der Decision-List-Schichten profitieren und dadurch die Genauigkeit erhöhen können. Im Vergleich zu vorherigen Ansätzen kann ein Modell mit geringerer Komplexität erzielt werden.

Contents

1 Introduction

The field of machine learning has evolved recently. Whereas in the early days, mainly separated domains were analyzed, the amount of available data today has grown rapidly. Today, a huge amount of data is collected by sensors in small devices such as smartphones, digital cameras or fitness trackers. These devices collect data such as image data, geo data, temperature, or data on health. Furthermore, there is a tendency that previously unconnected control devices become more frequently connected to the internet, such as devices in smart homes (thermostats, lighting controls etc.), vehicles (autonomous driving).

Besides the increasing connectivity of existing everyday objects, there are many applications for cheap new sensors to perform many kinds of monitoring tasks. Together with the growing availability of sensor devices, there is a financial interest in collecting and analyzing these large amounts of data. In light of the ubiquity of cloud services, sensor data are stored for a large quantity of users. On the one hand, this enables the users to easily control their data from the internet more comfortable. On the other hand, it empowers the cloud services to aggregate and analyze large amounts of data. Hence, there is more and more data available which is automatically collected by devices rather than by domain experts.

With the increasing amount of data, there is a need for scalable algorithms. One example domain is image processing. In an image collection, a data sample may be an image of about 1 mio pixels. An algorithm then uses each pixel as an individual attribute. Due to the large amount of attributes, the algorithms that will be used for the example domain need a low computational complexity concerning the attribute space. Furthermore, it is beneficial if an algorithm can process large amounts of images.

To process large amounts of data, deep neural networks have gained focus. These types of neural networks are constructed layerwise, and can be largely scaled. In deep neural networks, the costs of training a network are expensive and much computational resources are needed. However using these networks to make predictions and to perform their tasks without much computational resources. Therefore, neural networks are used increasingly in embedded devices. The applications include speech recognition in smartphones, lane-keeping assistants in vehicles and collision detection in drones. For these applications, often objectives like costs, power consumption, weight and size have to be optimized. Usually control circuits in these devices include a CPU with a RISC chip design. RISC chips allow for fast control flow and logical operations. As floating point operations are primarily used in neural networks, special processors are required which take into account.

Besides, the architectural limitations in terms of the command sets, there is a need for quickly accessing the memory for neural networks. For this purpose, in embedded devices, there are specialized neural network chips, or chips that have integrated GPU functionality.

However, it would be favorable to also use RISC Processors for that specific task. Therefore, it is necessary to use a different model representation. One such representation are symbolical rules. Currently, symbolical rules cannot be trained as efficiently as neural network. In this thesis, it is examined if symbolical rules can be trained efficiently by combining them with a neural network. For this purpose, a general scheme for training rule models is adapted specifically to the needs of large datasets.

In this thesis, a method will be further developed, that creates layered architectures of symbolical rules. This thesis is structured as follows. In chapter 2, the foundations of machine learning are introduced. In chapter 3, the related works is introduced. Chapter 4 deals a method for training symbolical rules will be introduced. In chapter 5, the new method will be evaluated. Finally, chapter 6 concludes the thesis and gives a perspective on future works.

Outlook	Temperature	Humidity	Windy	Class
sunny	hot	high	FALSE	not play
sunny	hot	high	TRUE	not play
overcast	hot	high	FALSE	play
rain	mild	high	FALSE	play
rain	cool	normal	FALSE	play
rain	cool	normal	TRUE	not play
overcast	cool	normal	TRUE	play
sunny	mild	high	FALSE	not play
sunny	cool	normal	FALSE	play
rain	mild	normal	FALSE	play
sunny	mild	normal	TRUE	play
overcast	mild	high	TRUE	play
overcast	hot	normal	FALSE	play
rain	mild	high	TRUE	not play

Table 1: Dataset example (taken from [Quinlan, 1986])

2 Machine Learning

In this chapter, machine learning is introduced. It can be considered as part of artificial intelligence. Machine learning algorithms are used to detect regularities in a dataset. They are used in the following way: first, a dataset is analyzed and a model is created during a training phase. Depending on the machine learning algorithm, the trained model allows to distinguish important from unimportant information within the data, make inference on the dimensionality of the underlying problem, give an insight for humans on the underlying problem and make predictions on new data of the given domain.

The data that will be analyzed can have various forms. In the most basic case, a dataset consists of sets of attributes. Each attribute is either binary, nominal or numeric. A binary attribute is either true or false, a nominal attribute can have a value from a predefined set of possible values and a numeric attribute can be a natural or a real valued variable. Besides the simple case in which a dataset consists of attributes only, the data can be composed of attributes and target attributes. Attributes can be measured or otherwise collected or constructed for a given problem, whereas the target attributes usually are collected with much effort and shall be predicted by the machine learning algorithms. Target attributes can be natural or real numbers, or also binary or nominal values. They are called class attributes if they are binary or nominal. If a prediction is to be made for a single class attribute the prediction is also called classification. A prediction for one or more continuous target attributes is also called regression. The smallest unit of a dataset is called sample. A sample is an observed state which contains information on all attributes.

In table 1, an example of a dataset taken from [Quinlan, 1986] is given. It contains information about the weather condition and an observation of a golf player that either plays or does not play. The dataset can be used to predict if the player does play golf depending on the weather situation. The dataset has four attributes (*outlook*, *temperature*, *humidity* and *windy*) and one target attribute (*play/don't play*). The nominal attribute *outlook* can have the value *sunny*, *overcast* or *rain*. The attributes *temperature* (*hot*, *mild* or *cool*) and *humidity* (*high*, *normal*) are also nominal attributes. The binary attribute *windy* states whether a certain wind speed threshold is exceeded. The present example does not contain any numerical attributes. The binary target attribute states whether the player does or does not play. Each line of the table represents a sample. The observation is made at a certain point in time.

It may be necessary to preprocess the given data, to meet the requirements of a specific machine learning algorithm. Preprocessing steps include discretization, string preprocessing, attribute selection and many more. Discretization transforms a numerical value to a binary or nominal value. Likewise, string preprocessing usually transforms a string into binary attributes or a nominal attribute. Attribute selection is a step to reduce the number of attributes that is available for the training algorithm. It is possible to discretize a target attribute in order to perform a classification.

For machine learning, it can be distinguished between three types of learning: supervised, unsupervised and semi-supervised learning. In the case of supervised learning, the training data used consists of attributes and pre-labeled target attributes. Supervised training is usually used to train a model for classifications or predictions of a numerical target attribute. In unsupervised learning, an algorithm is only trained with attribute values and without any hand-labeled data. Unsupervised training can be used to create a simplified dataset with reduced complexity, e.g. with autoencoders (see section 2.4.1). Other unsupervised learning methods make predictions about arbitrary attributes instead of predicting a target attribute. If rules are used, the data structure is called association rules. Training methods for association rules are introduced in section 3.1. Another unsupervised training task is clustering. Here, similar samples

Real	Predicted: A	Predicted: B	Predicted: C
A	0	1	1
B	1	0	1
C	1	1	0

(a) Misclassification error: correctly classified instances have a cost of 0, incorrectly classified instances have a cost of 1.

Real	Predicted: Critical = FALSE	Predicted: Critical = TRUE
Critical = FALSE	-1 (TP)	1 (FP)
Critical = TRUE	10 (FN)	-1 (TN)

(b) Custom cost matrix: classifying a critical state as normal in this example has 10 times higher costs than classifying a normal state as critical.

Table 2: Cost matrix example for misclassification error and a custom cost matrix.

are grouped into a cluster. This method will not be covered in this thesis and can be seen in e.g. [Xu and Wunsch, 2005] or [Baraldi and Blonda, 1999b].

Finally, it can also be beneficial to reduce the amount of attributes to speed up further processing. This step is called feature selection. Here, attributes are removed based on similarity measures like covariance. Semi-supervised learning is a mixture of supervised and unsupervised training. In semi-supervised learning tasks, only some of the available instances are pre-labeled. A model can be trained here by applying an unsupervised algorithm in a pre-processing step for the unlabeled data and a supervised algorithm in a second step for the labeled data.

According to [Baraldi and Blonda, 1999a], training methods can be distinguished between offline and online learning methods. Offline trainers operate on a training set which is fully known in advance and process data. In contrast, online training methods sequentially process data. While offline trainers process data as a block (also denoted as batch), online trainers can process samples as they arrive and therefore require less memory for storage. Online methods either operate on individual samples or on so-called mini-batches. In mini-batches, the complete dataset (batch) is divided into blocks of few samples.

Most classification algorithms return the confidence for each possible target value. The confidence is the estimated probability that the predicted target will actually be present in reality. In order to make a decision on how to improve a model, as well as on how to compare two models, it is of importance to define a cost function (also denoted as loss). For each sample, a cost function measures how well the algorithm maps the target function. During training, the costs of the model for the provided training data should be minimized. In case of a binary or nominal target variable, a simple definition is the misclassification error where a correct classification results in a cost of 0 whereas an incorrect classification results in a cost of 1.

While the misclassification error defines a simple error measure, there might be training situations which require a more elaborate balance of costs. Therefore, a cost matrix can be defined which defines a cost measure for each combination of predicted label and target label based on prior knowledge. In table 2, two example cost metrices are defined: table 2a defines a cost matrix for the misclassification error, table 2b is an example of a fictive industrial facility. In the latter example, some attributes may include measured sensor data, and a prediction target could be if a hazardous event may occur in the facility. The desired behavior would be that a critical state is reported with high priority. Therefore, reporting a critical state as non-critical, also denoted as *false negative* (short: *FN*) should be absolutely avoided and has to be associated with a high cost. In contrast, reporting a non-critical state as critical, also denoted as *true negative* (short: *TN*) would require a manual check for a problem, and therefore be associated with a low cost. In the other cases, where a critical state is correctly predicted (also: *true positive*, shortly: *TP*) and a non-critical state which is correctly predicted (also *true negative*, shortly: *TN*) are here associated with negative cost.

Besides defining a cost function for each sample, the function can also be defined for a set of samples. Usually this is done by averaging the individual costs of the samples. In case of a binary or nominal target variable, where no cost matrix is defined, this leads to the amount of wrong predictions, denoted as misclassification error. The opposite of the misclassification error, which is the amount of correct predictions is denoted as accuracy. Besides averaging the individual costs to gain a cost function for a set of samples, there are ways to include the confidence of the prediction. One such cost function is the entropy (see [Shannon, 2001]), which will be explained in chapter 2.2.

To estimate the result of a machine learning algorithm, a trained model can be validated against data. One way to achieve this is to use the available data once for training and then make a prediction for all the data available. The original data can be then compared to the predicted data, and an estimation about the quality of the model can be made. Models will achieve a good estimation if they can be used to reconstruct the target values accurately. However, the training examples may include noise, e.g. in the form of falsely labeled instances. The reconstruction of noisy samples is called *overfitting*. To detect if a model overfits the training data, samples have to be used for either training or validating. Therefore, the labeled data is split into a training set and a test set. Then a model is trained with the training data. In the second step, the algorithm makes a prediction about the test data by using only the attribute values. The resulting prediction of the target value is then compared to the provided target value of the test set. Validation is used to estimate

whether an algorithm fits well for a given problem or for the provided data. It can be used to compare machine learning algorithms.

In training scenarios where sufficient training data is available, the data can be split into a training set and a validation set. This is best done in a way that preserves the distributions among all attributes which can be achieved by using stratification (see e.g. [Esfahani and Dougherty, 2013]). Here, homogeneous subgroups are created, which are then equally distributed among the splits of the dataset. There are also cases, where the data is divided due to prior knowledge. This was done in [LeCun et al., 1998], where hand-written digits are assigned to a training and a test set, so that a writer either contributes written samples to the training set or to the test set, but not to both sets.

The split of available data into a training and a validation set in order to estimate the accuracy of the given dataset has to drawbacks. On the one hand, there is only a reduced amount of data available for training. Especially for small data sets, the reduced amount of training data can result in a model which is inferior to the model that is trained on the complete training data. On the other hand, only a subset of samples is validated. It may be the case that these samples are biased, and the estimation of the accuracy is prone to the split choice. To overcome these shortcomings, cross-validation can be applied. Here, the available data is split into n equal-sized folds. Then n training sets and n corresponding validation sets are created, and n models are trained and validated. Each training set contains $n-1$ folds, the corresponding validation set contains the fold that is left out for training. The n training and validation sets are created so that each fold is used once for validation. At cost of training multiple models, cross-validation creates models based on the majority of samples and at the same time uses all samples once for validating these models.

So far, the use of training data in machine learning algorithms has been dealt with. In the remainder of this chapter, machine learning algorithms will be introduced. This includes instance based, symbolical and neural network methods. For each method, first a data model that generalizes the training data is presented. Besides, the training methods for creating a model are introduced.

2.1 Instance based Algorithms

Instance based machine learning algorithms are the most basic way to perform a prediction. Here, no explicit model is created for the training data. Instead, the training instances themselves are used to make a prediction for unlabeled instances. The most prominent algorithm which uses instance based methods is the K-Nearest-Neighbors algorithm, which was first published in [Cover and Hart, 1967] (also see [Peterson, 2009]).

The algorithm maintains a set of labeled samples. For each prediction, the $k \in \mathbb{N}$ most similar samples are searched in the set of labeled samples. The prediction is made based on a voting of the k most similar samples. For a given dataset, a suitable similarity measure and a voting scheme has to be defined. A frequently used similarity measure is the eucledean distance:

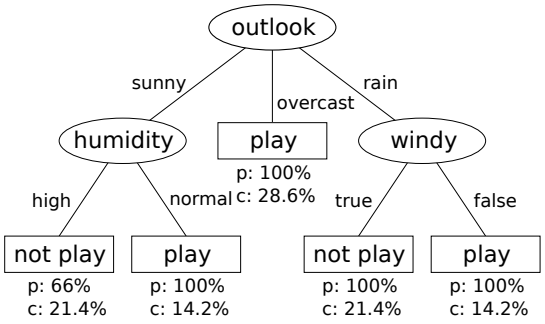
$$d(x_i, x_j) = \sqrt{(x_{i0} - x_{j0})^2 + (x_{i1} - x_{j1})^2 + ... + (x_{ip} - x_{jp})^2}$$

Here, x_i and x_j are two samples with p attributes and $d(x_i, x_j)$ is the eucledean distance. The eucledean distance can be applied well to numerical attributes. Other distance metrics include the hamming distance which counts the number of unequal binary attributes, and the cosine similarity which uses the angle between two attribute vectors as a measure for inequality. In contrast to the hamming distance, the cosine similarity has the property that two random vectors in high dimensional spaces are likely to be orthogonal to each other (and thus have a maximum distance to each other).

As a voting scheme for classification, the majority voting can be chosen. Here, the class with the highest number of labels among the nearest neighbors is chosen. By using $k > 1$, the prediction is less sensitive to noise. Voting schemes can also take into account the distance of a sample to its nearest neighbors, so that samples with a larger distance have less weight when voting.

In the general k-nearest-neighbor algorithm, all training samples are iterated. However, a prediction of one sample has a high complexity of O(|training samples|*|features|) Therefore, there are algorithms that address this problem by creating data structures where nearest neighbors can be found more quickly. These algorithms differ in the following aspects:

- Complexity of creating the data structure
- Complexity of finding the nearest neighbors
- Applicable distance metrics
- Dimensionality of the feature vector
- Approximation or exact nearest neighbors



(a) Example decision tree.

if outlook = sunny **and** humidity = high **then** not play
else if outlook = sunny **and** humidity = normal **then** play
else if outlook = overcast **then** play
else if outlook = rain **and** windy = true **then** not play
else if outlook = raun **and** windy = false **then** play

(b) Derived decision list. For each leaf node, a rule has been created by following the path from the root to the leaf. The traversed conditions of the decision tree are used as the rule body.

if outlook = sunny **and** humidity = high **then** not play
else if outlook = rain **and** windy = true **then** not play
else play

(c) Derived decision list. An optimal decision list for the example given. Here, the label that is present in the majority of the leaf nodes is taken as a default class.

Figure 1: Symbolical methods: decision tree and decision list

One optimization approach is to reduce the number of training samples that are used during the search of the nearest neighbors. In the ibl algorithms (ib1, ib2, ib3) which is introduced in [Aha et al., 1991], samples are iteratively added to a reduced training set. The basic concept of the algorithm is that all samples are discarded which can already be correctly classified with the reduced training set. As a consequence, samples are discarded in those areas of the input space where only one label is present. The remaining samples reside near the borders of similarly labeled areas. [Aha et al., 1991] also introduced a variant which is more resilient to noise.

Whereas one way to improve the lookup speed is to reduce the number of samples that are relevant for the nearest-neighbor search, other optimization techniques address the efficiency of finding samples by creating database-like structures. For low-dimensional input spaces, the complexity can be reduced by using multi-dimensional trees (see [Dasgupta and Freund, 2008]). Instead of traversing the complete training set, only one or a few branches of the tree have to be traversed. In high-dimensional input spaces, locality-sensitive hash functions can be used. Here, the dimensionality of the input space is reduced, by e.g. applying random projections from the high dimensional input space to an input space with reduced dimensions. Usually, these methods do not realize an exact lookup so that there are cases in which one nearest neighbors cannot be found with a small probability. Further information can be found in [Pan and Manocha, 2011] and [Dasgupta et al., 2011].

2.2 Symbolical methods

So far, the algorithms that have been used operate on samples and do not create a model (see e.g. [Kubat, 2015]). In this section, symbolical methods will be introduced which check the presence of attributes and make predictions based on logical operations. First, some definitions are introduced, that are related to symbolical methods. A boolean expression, that will check if a boolean or nominal attribute has a certain value, or if a numerical attribute value is in a certain range will be denoted as feature. The amount of samples, that can be applied for parts of the model will be denoted as coverage. A sample is said to cover a part of a model, if, according to the features of the model, a prediction can be made. If a sample is not covered by one part of the model, another part will be checked for coverage.

In this section, two symbolical methods, decision trees and decision list will be introduced. In a first part, it will be explained how the model is constructed. Then a general method for training models and two specific methods for training will be introduced.

2.2.1 Decision trees

One symbolic model for making predictions is the decision tree. A decision tree is a directed tree, that is labeled with features inside the inner nodes and possible feature values for its outgoing connections. The leaf nodes contain a class prediction. An example is given in 1a. A decision tree is evaluated by following a path from the root node to one leaf node. At each inner node, the feature value of the inner node is checked. Then the outgoing edge with that feature value is followed. When arriving at a leaf node, the label of the leaf node is taken as the prediction for the given sample.

2.2.2 Decision lists

Another symbolical structure besides decision trees are if-then rules. An example for an if-then rule is the following:

(1) **if** temperature = hot **and** windy = true **then** class := not play golf

(2) **if** outlook = overcast **then** class := play golf

(3) **if** a > 5 **then** class = 1
else if c < 3 **then** class = 2
otherwise class = 3

Here, rule (1), which is along the dataset shown in table 1, states a causality, that if the outlook is sunny and the humidity is high, the player does not play golf. Here, if the outlook is not sunny or the humidity is not high, there is no statement as to whether the player does play golf. An if-then rule is composed of two parts, a rule body and a rule head. The rule body (e.g. temperature = hot **and** windy = true) is a conditional expression that can be evaluated for a given sample. The rule body is composed of conditions (e.g. temperature = hot, a > 5) which are checks on whether a feature has a certain value. The rule head (e.g. class := not play golf) is a statement which is predicted if the rule body of a given sample is correct.

In the given example, rule (1) and rule (2) can be extracted from the dataset given in 1. However, the rules given do not account for all possible combinations. Therefore, there might be feature combinations where no rule applies. If e.g. the outlook is sunny and the temperature is mild, no prediction can be made when applying rule (1) and (2). There are also cases where predictions are contradicting, e.g. if the temperature is hot, it is windy, and the outlook is overcast. In such a case, there should be a defined order of evaluation for the rules.

A data structure that defines an evaluation order for if-then rules is a decision list. An example of a decision list is given in (3). Here, the first rule body is evaluated first. If the first rule fires, the first rule head is returned. Otherwise, the remaining rules are evaluated. The final rule of a decision list is a default rule, that always fires if its body is evaluated. This rule does not contain any condition. The structure of the decision list ensures that exactly one head is returned for all possible feature combinations.

The concept of decision lists is related to decision trees. A decision tree can be converted to a decision list by traversing all paths from the root to the leaves and by creating a rule for each traversed path. The rule body is composed of the conditions that are traversed by the path. The rule head is composed of the label of the decision tree. An example for the conversion is given in figure 1.

2.3 Training with optimization methods

A very common approach to training a model in machine learning by solving optimization problem (see [Bianchi et al., 2009]). Optimization methods are widely used, when the number of possible solutions is large, and cannot be calculated by brute-force. To solve an optimization problem with a local search, a model is initialized with a starting solution and refined stepwise. The starting solution usually is a random state (e.g. a random matrix), or a trivial state (e.g. an empty set).

First, an objective function will be defined. The objective function is used as an estimate for the quality of a solution. During optimization, the objective function will be maximized or minimized, depending on the optimization task. For symbolical methods, common objective functions are the accuracy (which has to be maximized) and the entropy (which has to be minimized). The entropy according to [Shannon, 2001] is defined as follows:

$$H = -\sum p_i * log_2(p_i)$$

where p_i are the probabilities, in with which an event i occurs. In case of a decision tree, the entropy is defined for an inner node and its outgoing edges. For each edge $e_i, i = 1, ..., n$ of a node n, the probabilities p_i are the probabilities of traversing an edge e_i after node n has been traversed. For example, in figure 3d, the node with the feature *windy* takes the edge which is labeled with *true* in 6 of 14 cases and the edge which is labeled with *false* in 8 of 14 cases. Therefore, $p_{true} = 6/14$ and $p_{false} = 8/14$. The entropy here is calculated as

$$H = -p_{true} * log_2(p_{true} - p_{false} * log_2(p_{false})$$

$$= -\frac{6}{14} * log_2(\frac{6}{14}) - \frac{8}{14} * log_2(\frac{8}{14}) = 0.89$$

In figure 3c, the node with the feature *windy* is not used in the root node. It is traversed in 5 of 14 cases, and $p_{true} = 2/14$ and $p_{false} = 3/14$. In this case the entropy is calculated as follows:

$$H = p_{windy} * (-p_{true} * log_2(p_{true} - p_{false} * log_2(p_{false}))$$

$$= -\frac{5}{14} * (\frac{2}{5} * log_2(\frac{2}{5}) + \frac{3}{5} * log_2(\frac{3}{5}) = 0.35$$

Entropy is related to the minimum number of bits which are necessary to encode a message. A low entropy of the decision tree means, that a low amount of information is necessary to correct the predictions of the decision tree.

The number of possible solutions may be arbitrary large and a brute-force approach, where all solutions are constructed may exceed the computational capacities. However, solutions that are similar except some small detail will probably have a similar result concerning the objective functions. Here, solutions that are similar to a good solution that has been evaluated earlier might be a good candidate for further improvements. In contrast, solutions that are similar to a bad solution will not further have to be evaluated. This observation leads to the definition of a so-called neighborhood. A neighborhood of a solution is defined as a set of solutions, which can be constructed by changing an atomic detail of the solution. For each domain, neighborhoods can be defined in many ways, and there is not one single definition.

In case of decision trees, a neighborhood definition can include splitting a leaf node by adding two child nodes (also denoted as *grow* operation) and removing all children from an inner node (also denoted as *prune* operation). In case of rules, the corresponding operations would be adding conditions (*grow*) and removing conditions (*prune*). Usually, grow and prune operations are performed as separate optimization steps. Therefore, there may be one neighborhood definition which only includes *grow* operations and a separate neighborhood definition which only includes *prune* operations.

An example for the neighborhood of decision trees for the dataset introduced earlier is given in figure 2 - 5. In figure 2, the empty decision tree as the trivial starting solution is shown. Figure 3 shows the neighborhood for the empty decision tree. The lowest entropy has the decision tree in figure 3a. The neighborhood of this decision tree is shown in 4. After three steps, the decision tree in figure 5 can be constructed.

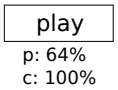


Figure 2: Empty decision tree with a single root node. The value *play* is used as the default class. Entropy: 0.94, Misclassification Error: 35.7%

Besides defining a small set of feasible refinement steps, compared to the set of possible solutions, a neighborhood should be constructed in a way, that allows to transform one solution into any feasible solution.

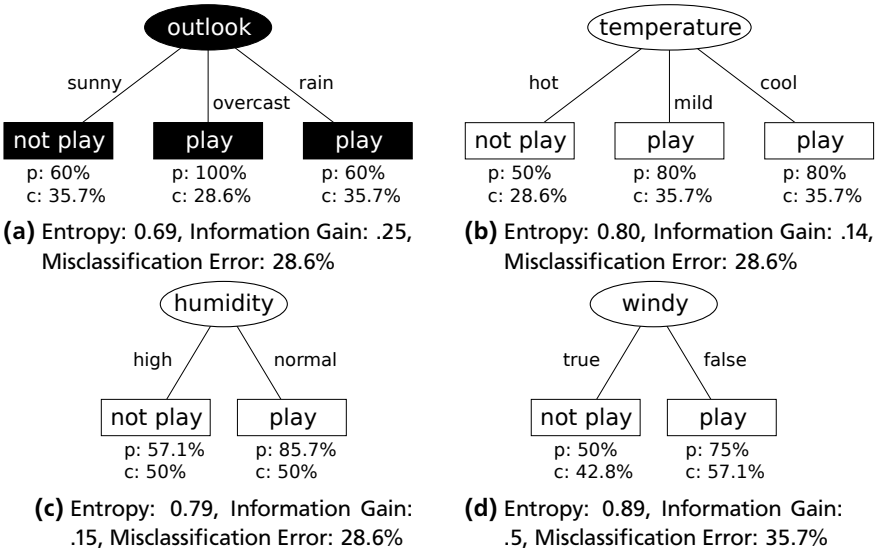
After having determined a suitable objective function and having defined a neighborhood as the search space, a strategy for efficiently exploring the search space is needed. This strategy is also referred to as the search heuristic. A quite simple approach is to perform a Breadth-First Search (BFS) or a Depth-First Search (DFS) on the search space, which traverses all possible solutions, and would probably exceed the computational capacities. For this exhaustive search, a list of all neighbors, which have not yet been visited, would need to be stored. However, many of those neighboring solutions could be neglected due to the bad performance in terms of the objective function.

The most basic approach in searching the search space is the greedy best-first search strategy. Among all possible neighbors, only the best neighbor according to the objective function is chosen. The greedy best-first search is used for training of decision trees and decision lists. While the greedy variant of the best-first search only follows one path in the search space, the beam search maintains a list of the best *n* solutions. At each step of the beam search, the neighborhood of the maintained list is explored. Among the neighborhood, existing solutions are replaced by the best solutions in the neighborhood.

A problem of the greedy search is that instead of terminating with a globally optimal solution, the algorithm might terminate with a solution, that is only best among its neighborhood. Such a case is denoted as a locally optimum solution. While the beam search follows more than one path in the search space, it might find superior solutions. Another way to explore multiple paths in the search space is to start with more than one random starting solution. This is done in an optimization phase of the RIPPER algorithm (see section 2.3.2).

2.3.1 C4.5

A common training algorithm is the C4.5 algorithm (see [Quinlan, 2014]). It performs a best-first search and uses information gain as the split criterion. To handle numeric attributes, the attribute values that are present in a leaf node during training are sorted. Then, the mean value of two consecutive attribute values x is checked as a split criterion.



(a) Entropy: 0.69, Information Gain: .25, Misclassification Error: 28.6%

(b) Entropy: 0.80, Information Gain: .14, Misclassification Error: 28.6%

(c) Entropy: 0.79, Information Gain: .15, Misclassification Error: 28.6%

(d) Entropy: 0.89, Information Gain: .5, Misclassification Error: 35.7%

Figure 3: Example neighborhood of the empty decision tree: There are four attribute choices for splitting the root node. In terms of misclassification error, the decision trees in figures 3a - c are the best choice. They have the lowest misclassification error. When it comes to entropy, the best choice is figure 3a. It has the lowest entropy (and the highest information gain respectively).

The split point x separates all attributes samples with an attribute value $a_i < x$ from attribute values $a_i >= x$. For each possible split point, the information gain criterion is checked.

In the beginning of this chapter, overfitting has been introduced as a lack of generalization abilities of the trained model. During validation the separation of the training data into a training and a validation set has been identified as a countermeasure. To detect and revise overfitting during training of a decision list, a similar approach is followed. Here, a split is made to separate a grow set from a prune set. The decision tree is trained by applying grow operations based on samples of the grow set. The prune set is then used to detect overfitting. There are two types of pruning: pre-pruning and post-pruning. During pre-pruning, the decision list is grown until overfitting is detected when splitting a leaf node. The leaf node is then not further refined. Another pruning method is post-pruning. Here, a decision tree is first grown with the grow set until no improvement can be achieved. The resulting decision tree is then pruned by removing edges and leaf nodes as long as an improvement can be achieved concerning the prune set. While grow operation use the information gain criterion, the prune operation uses the misclassification error as the optimization metric.

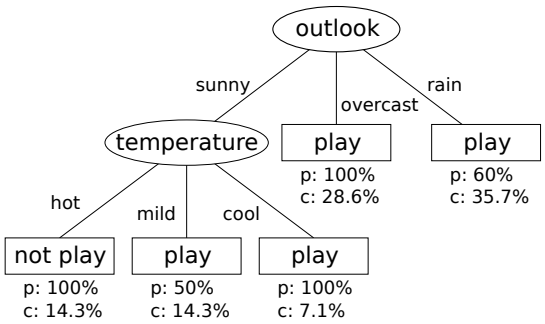
2.3.2 RIPPER

While decision trees are trained by iteratively splitting nodes, there are two general ways to create sets of rules: general-to-specific and specific-to-general. When creating a set of rules in a general-to-specific way, the set of rules is initialized with an empty rule. This rule is then refined by adding conditions. Whenever there are uncovered samples, new rules are created. When training rules in a specific-to-general way, a rule is created for each sample. Each rule contains a condition for each attribute. Then, conditions are removed and rules with the same conditions are combined. While these general schemes can also be applied for decision lists, the most popular way is to train the decision lists rule by rule. After having trained a rule, all samples covered by this rule are removed from the training set and further rules are trained with the remaining rules.

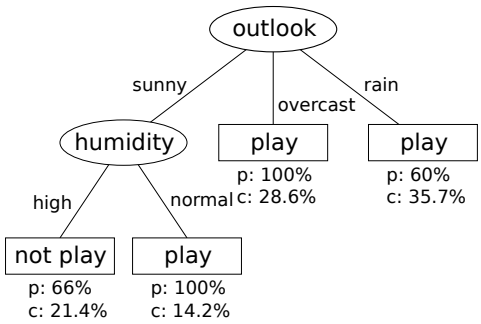
A popular training algorithm for decision lists is the RIPPER algorithm (Repeated Incremental Pruning to Produce Error Reduction). The training in RIPPER is performed in two phases: an initial phase and an optimization phase. In the initial phase, rules are created by growing and pruning. A rule corresponds to a decision tree in which only one path from the root to a leaf is refined. The rule head contains the features of the path while the body bears the label of the leaf. During the optimization phase, rules are replaced and removed based on their minimum description length. Due to the rule changes, some previously covered samples are not covered by the optimized rules. Therefore, new rules are trained based on these remaining samples.

In RIPPER, all rules are grown by using the information gain as the split criterion. In the initial phase, rules are pruned with reduced error pruning. Here, the metric is

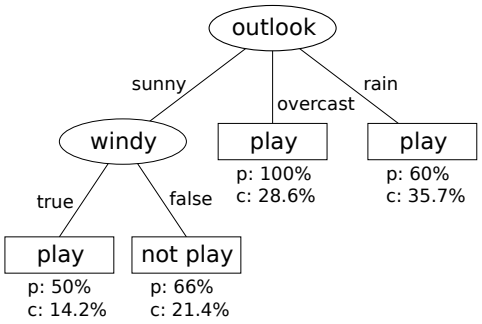
$$\frac{p-n}{p+n}$$



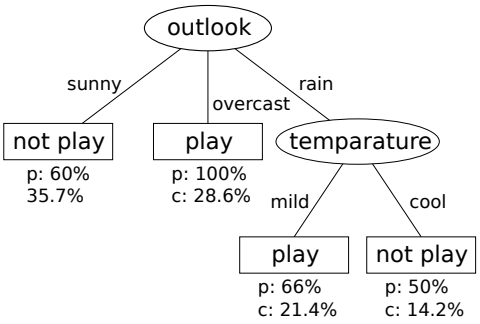
(a) Entropy: 0.49, Information Gain 0.20, Misclassification Error: 21.4%



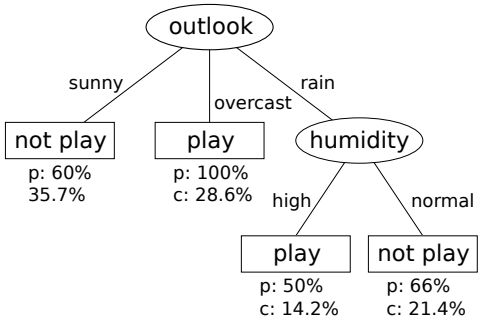
(b) Entropy: 0.54, Information Gain: 0.15 Misclassification Error: 21.4%



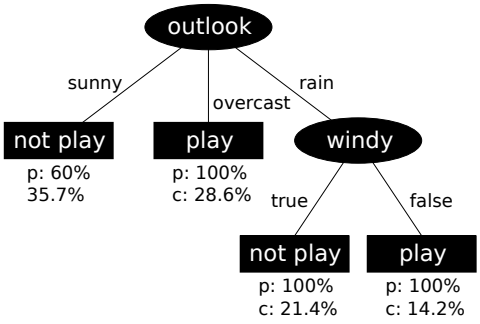
(c) Entropy: 0.69, Information Gain: 0.00, Misclassification Error: 28.6%



(d) Entropy: 0.68, Information Gain: 0.01, Misclassification Error: 28.6%



(e) Entropy: 0.68, Information Gain: 0.01, Misclassification Error: 28.6%



(f) Entropy: 0.34, Information Gain: 0.35, Misclassification Error: 14.3%

Figure 4: Example neighborhood for the decision tree in figure 3a: In figures 4a - c, the first leaf node is used for the next split, in figures 4d - f, the third leaf node is used. The second leaf node already has a precision of 100% and does not need to be further refined. While in figure 3 there have been four attributes available as split candidates, the attribute overcast will not be used for a second time here. The best candidate with the lowest misclassification error and the lowest entropy is shown in figure 4f.

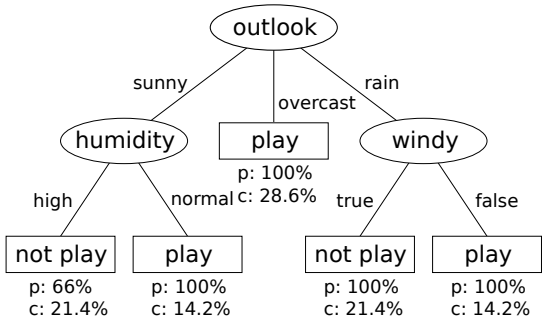


Figure 5: Decision Tree with three leaf nodes, Entropy: 0.20, Information Gain: 0.14, Misclassification Error: 7.1%

where p is the amount of positive samples and n is the amount of negative samples covered. Since the decision list is an ordered list which is constructed by growing and pruning the last rule in the initial phase, there is no other rule succeeding the currently pruned rule. Contrary to the initial phase, in the optimization phase an intermediate rule is adopted. Therefore, the pruning in the optimization phase takes into account the prediction of the succeeding rules. The prune metric maximizes the true positives TP and the true negatives TN of the complete pruning set, so that the metric

$$TP + TN$$

is used. If both the label of the pruned rule and the prediction of the succeeding rules have the correct target label of a sample a prune operation would not affect the accuracy of the sample. The same applies in cases where both have the wrong target label, These cases are therefore being ignored.

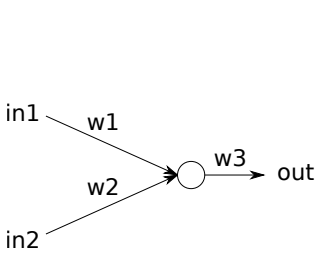
At the begin of the optimization phase, the training set is shuffled and a new grow and prune set is created. For each existing rule, two alternative rules are generated: one rule by growing and pruning the existing rule, the other one by creating a new rule from scratch. The original rule is replaced by an alternative rule if it has a smaller minimum description length, whereby the alternative rule with the smallest minimum description length is chosen. Besides searching for alternative rules during the optimization phase, existing rules are removed if they increase the overall minimum description length.

To sum up the results of the optimization, the following effects might play a role. Firstly, the result of the pruning takes into account the prediction of other rules and therefore leads to an increased accuracy compared to the pruning during the initial phase. As the preceding rules in the decision list might change, the succeeding rules need to be adapted. Additionally, training a rule with a different split of the grow and prune set leads to a lower influence of the random assignment for the split.

2.4 Neural networks

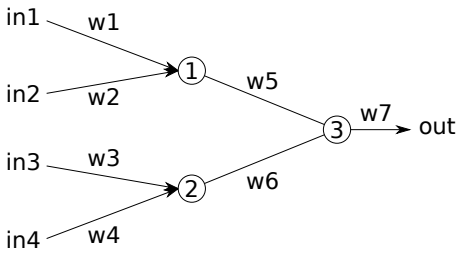
Neural networks (also referred as artificial neural networks) are machine learning algorithms, which are inspired by biological processes, that take place in the brain of humans and animals. Neural networks are widely used for tasks like image processing, speech analysis and text analysis. Common image processing tasks for neural networks are optical character recognition and image classification. Speech analysis tasks like transcription and text analysis tasks like synonym learnings can also be done with these algorithms. Neural networks are described e.g. in [Haykin and Network, 2004]

An artificial neural network can be represented as a directed graph. It contains neurons as nodes and weighted connections as edges. A neuron is a unit that has several real valued inputs and one output, and has an activation function, that restricts the range of the output. The calculation of an output value based on the input values is also denoted as activation. During activation, the input values are multiplied by the incoming weights of a neuron and summed up. The sum of the weighted inputs is then transformed by an activation function $f : \mathbb{R} \to [0,1]$ or $f : \mathbb{R} \to [0,\infty)$ or $f : \mathbb{R} \to (-\infty,\infty)$. An illustration for a neural network with a single output is shown in figure 6a.



$out = f(w_1 in_1 + w_2 in_2) w_3$

(a) Neural network with one neuron and corresponding network function.



$out = f_3(f_1(w_1 in_1 + w_2 in_2) w_5 + f_2(w_3 in_3 + w_4 in_4) w_6) w7$

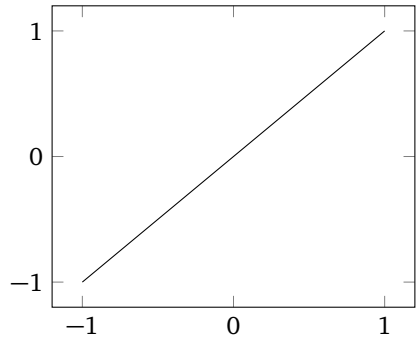
(b) Neural network with three neurons and corresponding network function.

Figure 6

The output of a neuron can then be used as an input for other neurons. The connections connect an output of a neuron with the input of another neuron. Each time a neuron has been activated, the output of a neuron is passed to the next inputs. This is illustrated in figure 6b. If an output of a neuron is used as an input for other neurons, such a neuron is called a hidden neuron. Besides, a neuron that is used to predict a target value is called an output neuron. Finally, it

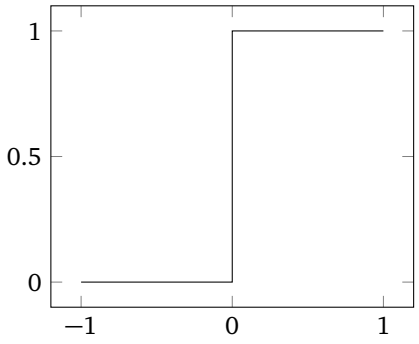
is a convention to connect each input with a single neuron without applying an activation function. Such a neuron is denoted as input neuron.

Common activation functions are shown in figure 7. They include the linear function (see figure 7a), the step function (see figure 7b), the sigmoid function (see figure 7c), the relu function (see figure 7d) and the softmax function.



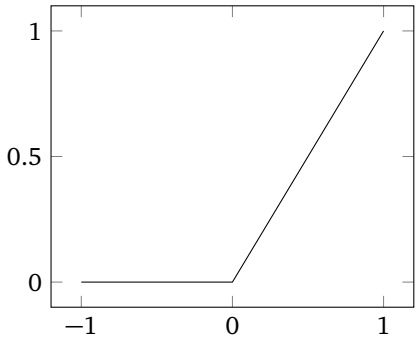
$$f(x) = x$$

(a) Linear function



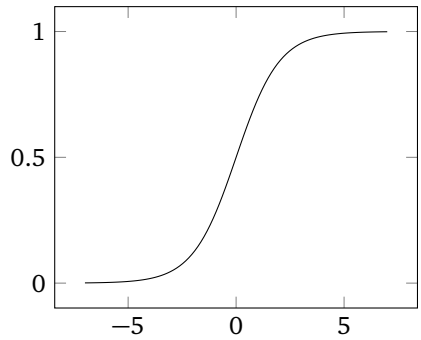
$$f(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

(b) Binary step function



$$f(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases}$$

(c) ReLU function



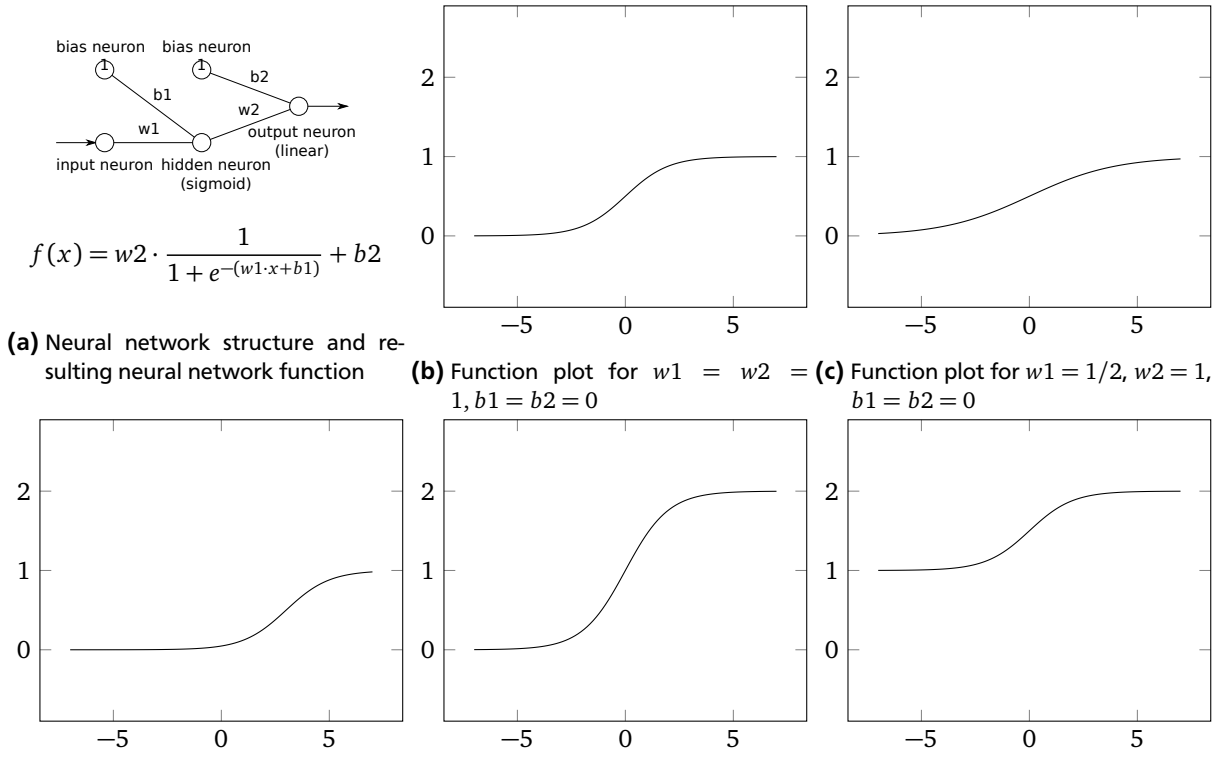
$$f(x) = \frac{1}{1 + e^x}$$

(d) Sigmoid function

Figure 7: Activation Functions

The linear activation function (see figure 7a) applies no transformation and can be used if the underlying problem depends on a linear combination of the inputs. As an example, the summation function can be implemented with a linear activation function with the neural network described in figure 6a when using $w_1 = w_2 = w_3 = 1$. While the outputs value of neurons with the linear activation function is in the range of $(-\infty, \infty)$, the result of the stepwise activation function is bounded to the range of $[0, 1]$. Moreover, the stepwise activation function has discrete outputs and therefore is well suited to implement logical functions like the logical AND, OR and XOR functions.

While in the previous examples, target functions are composed of the functions that are available in neural networks, other target functions can be approximated by summing up the result of overlapping transpositions of sigmoid (see figure 7d) function. Before approximating arbitrary functions, it will be further looked at how sigmoid functions can be moved and scaled along the x and y axes. This is illustrated in figures 8a - f. The neural network which is used for this purpose has one input neuron, one hidden neuron with a sigmoid activation function and one output neuron with a linear activation function (see figure 8a). The sigmoid function is returned if all weights are 1 and the biases are 0 (see figure 8b). Both, the hidden neuron and the output neuron have a bias unit as an additional input. Hence, there are four weights, that can be adapted. By changing the weight between the hidden neuron and the output neuron, the sigmoid function can be scaled along the y-axis (see figure 8c). Changing the weight between the bias unit and the output neuron can move the sigmoid function along the y-axis (see figure 8d). Furthermore, the function can be scaled among the x-axis by adjusting the weight between the input neuron and the hidden neuron (see figure 8e), and can be moved along the x-axis by adjusting the bias of the hidden neuron (see figure 8f).



(a) Neural network structure and resulting neural network function

(b) Function plot for $w1 = w2 = 1, b1 = b2 = 0$

(c) Function plot for $w1 = 1/2$, $w2 = 1$, $b1 = b2 = 0$

(d) Function plot for $b1 = -3$, $b2 = 0$, $w1 = w2 = 1$

(e) Function plot for $w2 = 2$ and $w1 = 1$, $b1 = b2 = 0$

(f) Function plot for $b2 = 1$, $b1 = 0$, $w1 = w2 = 1$

Figure 8: Transposed sigmoid function

After having illustrated how a single sigmoid or ReLU function can be transposed, examples for the approximation of the sine function on a limited interval based on the ReLU and the sigmoid activation function are given. Figures 9a - d show how the sine function can be approximated by the sigmoid function, figures 9e shows the plot of the sine function for comparison. The neural network has been trained with the training methods that will be introduced later in this chapter and are here presented for illustration purposes.

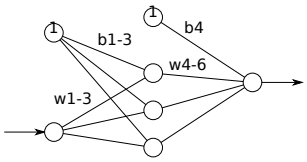
Finally, the limit of the sigmoid function is closer looked at. In the interval of $[-x, x]$, the sigmoid function is linear for $x \to 0$, whereas in the interval of $[x-1, x+1]$ for $x \to \infty$ and $x \to -\infty$. As a consequence, both the linear behavior of the linear activation function and the discrete behavior of the binary step function can be imitated by adapting the weights appropriately.

2.4.1 Topology

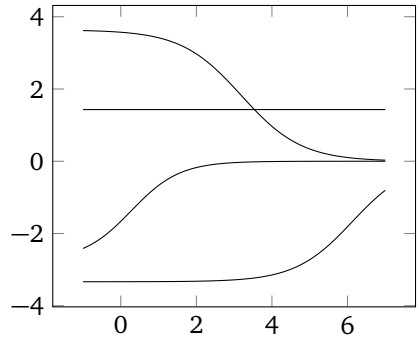
Previously, it has been explained how neural networks are constructed with neurons and connections. Neurons have been further divided into input neurons, hidden neurons and output neurons. The neural networks have been visualized as a directed graph. So far, a neural network configuration has been described as individual neurons with individual connections to each other. However, as neural network grow large, there is a need for finding a more compact description. Therefore, neurons are grouped into layers. All nodes of a layer share the same activation function. Besides, all neurons of a layer are of the same type: input layers contain only input neurons, hidden layers contain only hidden neurons and output layers contain only output neurons. Layers are then arranged hierarchically, so that in input layer is followed by hidden layers, which are the followed by the output layer. Hidden layers are then enumerated due to their order.

Besides arranging neurons in layers, there are conventions for constructing the connections between neurons. A layer i and layer $i+1$ are called fully connected, if there is a connection between every neuron of layer i and every neuron of layer $i+1$. The edge weights between two fully connected layers can be stored in a matrix which is denoted as the weight matrix. During a prediction, the activations of a layer can be described as a vector, and the propagation of one layer to the next layer can be described as a matrix operation. Besides describing the connections between layers as a matrix, the connections to a bias unit can be described as a vector. An example for a layered neural network is visualized in figure 12. The resulting activation can then be formulated as

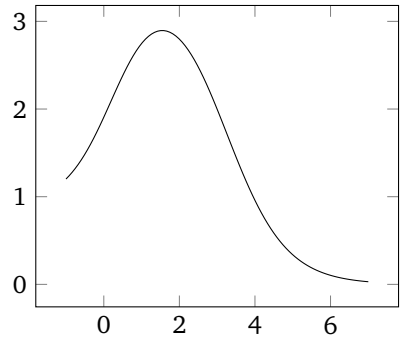
$$f_i(x) = a(xw + b)$$



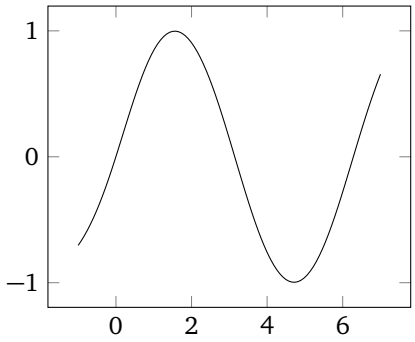
(a) Neural network structure for approximating the sine function



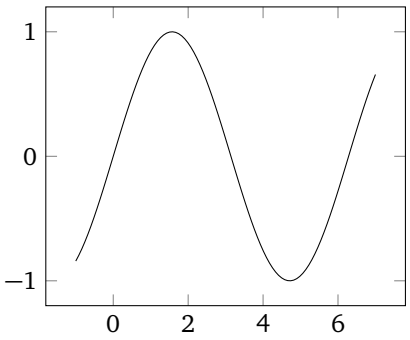
(b) Plot of the functions, that are summed up at the output neuron



(c) Sum of two selected functions from (b). Here, the shape of the sine function can be seen.



(d) Plot of the approximated function. It can be well approximated in the interval $[0, 2\pi]$



(e) $sin(x)$

Figure 9

where f_i is the activation function of layer i, x is the input vector of layer i, w is the weight matrix between layer $i-1$ and layer i, and b is the bias vector of layer i.

Besides the layered construction of a neural network, there are more characteristics that help distinguishing certain types of neural networks. In the following, three types of neural networks are introduced: feed-forward neural networks (also denoted as multilayer perceptron), recurrent neural networks and autoencoders.

Feed-forward neural networks are stateless neural networks, where the prediction only depends on the input vector, that is applied. The graph that is constructed is a directed acyclic graph. Connections only exist from layer i to layer $i+1$. No connection from layer i to a layer $j < i$ is permitted. So far, all examples of neural networks have used the feed-forward structure. In a feed-forward network, the layers are activated layerwise from the first to the last layer. Due to the stateless property, the network can be activated with samples in an arbitrary order. Furthermore, the network can be trained on individual samples instead of a series of samples.

So far, neural networks have been introduced with the intention of approximating a function. This has been achieved by performing a regression with a single output neuron. In general, multivariate functions or more than one function at once can be trained with a neural network, that has an output for each dimension of the multivariate function result. Besides training a neural network for approximating functions, neural networks can also be trained to perform a classification. Here, for each possible value of the nominal class value, an output neuron is created. The output neuron then predicts the confidence for each class value of the given sample. While in case of multivariate regressions, an output might be independent of other outputs, in classification problems the confidence sums up to 1. This can be achieved with the softmax activation function, which is applied to the output layer for classification problems:

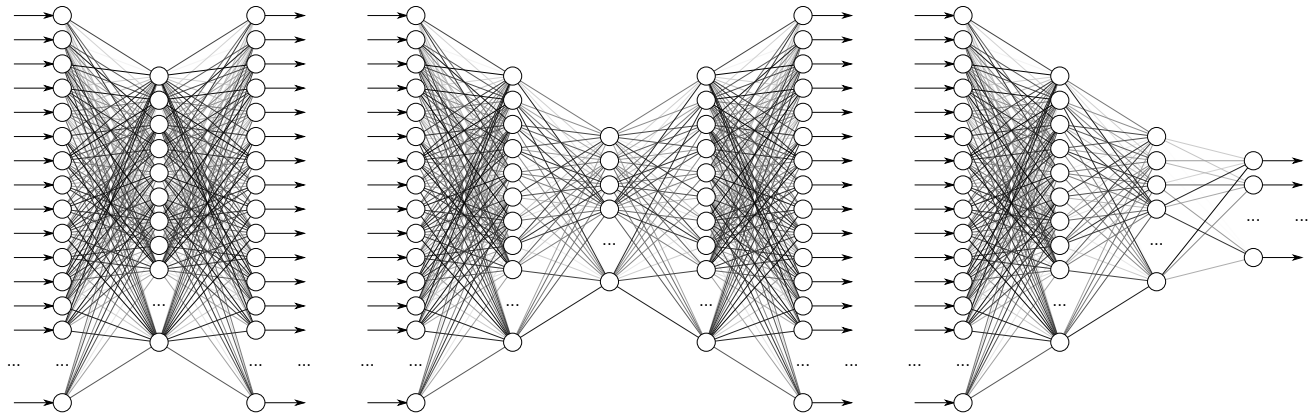
$$f_i(x) = \frac{e^{x_i}}{\sum_{k=0}^{n} e^{x_k}}$$

Here, i denotes the ith output of the softmax layer with n neurons. While feed-forward networks are used to predict individual samples, recurrent networks can be used to predict time series of samples. Recurrent networks are stateful, so

that a prediction alse depends on the result of previous predictions. This is achieved by using the previous result of the last hidden layer as additional inputs for the first hidden layer for the next prediction.

Another neural network topology which is frequently used if many hidden layers are used are autoencoders (see [Goodfellow et al., 2016]). Instead of training a network in a supervised way, where labeled classes or target variables are available, an autoencoder is trained in an unsupervised way. An autoencoder can be trained with unlabeled samples which may be available in addition to labeled instances. Therefore, by training an autoencoder in a preprocessing step, the final classification result can be improved. The goal of the autoencoder is to find a representation of the attribute space with a reduced dimensionality. The autoencoder mainly is a network where the attributes are used as both, inputs and as outputs. It has two parts which are arranged symmetrically. The first part reduces the dimensionality while the second part reconstructs the original attribute values. In the first part, the number of output units decreases with each hidden layer while in the second part, the number of input units increases with each hidden layer. An example of an autoencoder with one hidden layer is given in figure 10a.

To achieve a fast convergence, autoencoders are best trained layerwise. This means, that the training starts with an autoencoder that has one hidden layer. After having trained the autoencoder with a single hidden layer, the hidden layer is duplicated, and a new hidden layer is positioned between the duplicated hidden layer. The resulting new autoencoder can be seen in figure 10b. It consists of three hidden layers. After training this new network, additional hidden layers may be added by again duplicating the hidden layer in the middle and adding a new hidden layer in between. Due to the symmetric structure of the autoencoder, the weight matrixes can be used twice, once for constructing the space of reduced dimensionality, and as a transposed matrix to reconstruct the original attribute space. This is denoted as a tied weight matrix.



(a) First training step of the autoencoder (one hidden layer).

(b) Second training step of the autoencoder (three hidden layers).

(c) Usage of the autoencoder in classification. The first two layers are taken from the autoencoder, the third layer is added.

Figure 10: Autoencoder topology: the intesity of the node-connecting lines symbolizes the weights of connections.

2.4.2 Training of neural networks

At this point, it has been shown how arbitrary functions can be approximated by a neural network and a selection of network topologies has been introduced. In the following, it will be described, how neural networks can be trained (see [Haykin and Network, 2004]). The training method introduced here can be applied to feed-forward neural networks and autoencoders. The training of recurrent neural networks will not be dealt with here.

The main idea of the neural network training is to minimize a loss function. The loss function is a measure how well a target attribute can be predicted by the neural network. Common loss functions are the mean square loss, which is frequently used for regression tasks and is defined as:

For classification tasks when using the softmax activation function, a frequently used loss function is the cross-entropy:

For a given input vector, and a given target value, the loss can be determined by activating the neural network with the input vector and comparing the result with the given target value.

The general training of the neural network is done as follows. In a first step, the neural network is initialized with random weights. Given the random weights, the prediction initially produces some random output. If the weights of the

neural network are changed, the prediction for a given input may change. The idea is to change the weights with an amount which is proportional to the reduction of the loss function. Weight changes which lead to a high reduction of the loss are done with a high amount while weights which have no effect on the loss function remain unchanged and can be adapted when optimizing the network for other samples. A mathematical definition of the loss for the given weight is:

$$\frac{\partial W}{\partial loss}$$

where W is the weight matrix of the neural network and $loss$ is the loss for a given sample. The loss of the bias b is:

$$\frac{\partial b}{\partial loss}$$

. To achieve a convergence and avoid divergence, the weights are adapted with a small learning rate α so that

$$W_{t+1} = W_t + \alpha \sum_{s \in Samples} \frac{\partial W_t}{\partial loss(s)}$$

. and

$$b_{t+1} = b_t + \alpha \sum_{s \in Samples} \frac{\partial b_t}{\partial loss(s)}$$

. In practice, learning rates like $\alpha = 0.01$ are used. The network is iteratively refined. This can be done in an online way, either by applying the weight updates for complete epochs in a batch way or by training in mini-batches for faster convergence.

3 Related Works

This section deals with topics which are of interest for this thesis. Of interest are works which either combine the training of symbolical models with connectionist methods like neural networks as well as works in which sets of rules are created in a competitve way. More specifically, three methods are further described in this section: descriptive symbolical learners, ensemble training and fuzzy neural networks. The descriptive symbolical methods that are introduced here are association rule learners, subgroup discovery, contrast set mining and emerging pattern mining. These methods create a set of descriptive symbolical models instead of a single classification model. They are of interest, because a set of symbolical rules is trained simultaneously. Ensemble learners are methods for training multiple instances of a training algorithm and creating an improved solution by combining the results of the individual classifiers. Finally, fuzzy neural networks are special instances of neural networks with a special topology. They exploit the training capabilities of neural networks and can be transformed into fuzzy rules.

3.1 Descriptive symbolical methods

In this part, descriptive symbolical methods will be introduced. These methods can be used to generate a set of models that is human-understandable. One descriptive method that creates rules in an unsupervised way are association rules (see e.g. [Fürnkranz and Kliegr, 2015]. Association rules are if-then rules which explain relations in a set of attributes. Whenever possible, a prediction for each attribute is made by an if-then rule, which uses other attributes as features.

Association rules can be trained with the apriori algorithms. This algorithm performs a breadth-first search in order to get sets of so-called frequent items. Frequent items are conditions that have a high coverage on the training data. Each set of frequent items is then transformed into association rules by checking all combinations of rules, that can be created with the features of a frequent itemset.

While association rules generate a set of rules that is of interest for explaining the attributes, the aim of subgroup discovery (see [Herrera et al., 2011]) is finding a set of symbolical models which are of interest for a given target variable. In contrast to classifying a target variable with a single complex model and optimizing the precision concerning the target, multiple models are created that optimize other measures like simplicity and generality. Training algorithms for Subgroup Discovery can be deduced from classification algorithms. Similar to Association Rules, the space of possible subgroups of a given dataset can be explored with a beam search algorithm.

While subgroup discovery aims to find rules that bear resemblance to a classification rule, contrast set mining is an unsupervised approach, where objective is to predict groups that bear no resemblance to each other (see e.g. [Novak et al., 2009]). Here, the sets that are created are tested to be stochastically independent concerning the used dataset.

All three methods have in common, that a set of symbolical models is trained. The results of these methods could be used as attributes for further classifiers and create a hierarchical classifier.

3.2 Ensemble Learners

Ensemble learning is a method of combining the result of multiple classifiers. Here, stacking, bagging and boosting are introduced. In stacking (see [Wolpert, 1992]), a dataset is trained with different classification algorithms. The results of the individual classifiers is then classified by a meta classifier. The meta classifier decides, which combination of predictions leads to the best result.

Another ensemble training approach is Bootstrap Aggregating (short: Bagging) [Breiman, 1996]. Here, random subsets of training data are created. These subsets may be smaller than the original training data and can lead to less complex classifier models. A majority voting is then performed to get the final result.

The final ensemble training method that is introduced is boosting [Schapire, 2003]. Whereas Bagging uses a fixed set of classifiers, boosting iteratively creates new classifiers. Here, the randomized training data includes misclassified instances with a higher propability. The final classification result is then determined by performing a majority vote, where the class is returned which is predicted by the majority of the classifiers.

3.2.1 Gradient Boosting

While boosting in uses a majority voting scheme most promising class is returned, the precision of the classifier can be also taken into accout by applying a weighted voting. Gradient boosting as introduced in [Friedman, 2001] creates a decision tree and stores the precision for each class in the leaf nodes. For a given set of attributes, a weight matrix can be created. This matrix can be exploited to perform a gradient decent and determine nodes that can be split to refine the decision trees as well as updating the precisions at the leafs for each class. An efficient implentation is introduced in [Chen and Guestrin, 2016].

3.2.2 Ensemble Pruning

Ensemble pruning is a method, that can be applied for trained ensembles of base classifiers which are combined by a weighted voting. It is a pruning method, where the size of the trained ensemble is reduced by removing base classifiers. This is done in order to decrease the model size and reduce the overhead that is produced by large models (see [Tsoumakas et al., 2009]).

As stated in [Chen et al., 2009] simultaneously generating new instances of base classifiers and pruning existing classifiers while updating the underlying meta classifiers outperforms separate ensemble training and ensemble pruning. The authors of that paper used a genetic algorithm to maintain a set of base classifiers, and evaluated their method against ten UCI datasets. Model refinements of the base classifiers were done by randomly mutating existing model instances.

3.3 Deriving descriptive rules from trained neural networks

Whereas ensemble learning methods can be applied to any learning algorithm, there are related topics that focus more on the combination of the symbolical rules and neural networks. One such topic is the derivation of symbolical rules from trained neural networks (see e.g. [Zöller, 2014]). Whereas neural networks gain high accuracy in classification tasks, the trained model is difficult to understand by humans. In contrast, symbolical rules often provide a better insight. The objective of deriving rules from trained neural networks is to find a model, that has a better descriptiveness. On the other hand, a loss of accuracy is condoned.

These methods can be of use in real world applications like credit scoring or in critical applications like medicine or power plants. In credit scoring (see [Baesens et al., 2003]), transparency may be required by law or the usage of some features may be prohibited. In power plants or other critical applications, decisions made by neural networks may lead to dangerous situations or harm humans. Therefore, a review of the decisions of neural networks might be necessary.

These methods however have some drawbacks. Firstly, the trained model of neural networks and symbolical rules differs largely. For instance, there is no way to precisely describe a small neural network with few rules. Secondly, the descriptiveness of symbolical rules is not good for all data sets. Here labeled features like e.g. temperature, humidity can be well understood in contrast to features with unknown meaning. Furthermore, if there are many features with a similar meaning, like pixels in an image, rules do not provide an adequate insight. This problem is increasingly present for large attribute spaces, and if the extracted rules get more complex.

3.3.1 Pedagogical Methods

The first method of deriving symbolical rules from trained neural networks uses the neural network as a black box (or 'Oracle') and trains a symbolical rule. Therefore, instead of using the original training samples, a classification is performed with the neural network and the results are used as new training instances for the symbolical rules (see e.g. [Martens et al., 2008]). The main target of this method is to find suitable training samples for the symbolical rules. One way is to use the attribute of the original training samples, and replace the class assignment by the classification result of the neural network. However, samples with no class assignment can be used if present in a dataset. Moreover, derived training samples may be generated from the original samples.

Further pedagogical methods are introduced in [Craven and Shavlik, 1996]. Here, symbolical rules are trained with the available training samples. If additional samples are needed, they are sampled with respect to the marginal distribution of the original samples. Moreover, [Craven and Shavlik, 1996] trains N-of-M rules. An N-of-M rule is evaluated by suming up M selected attributes. If the sum exceeds the threshold of N, the result is set to true (otherwise false). These rules are trained based on the information gain criterion.

3.3.2 Classifying Hidden Units

The second method of deriving symbolical rules from trained neural networks is to treat the neural network as a white box, and reclassify layer-wise all hidden and output units of the neural network. In [Zilke, 2015] this approach has been identified as the best way to derive symbolical rules from deep neural networks. In the suggested method, common decision tree classifiers were used. However, it is expected that due to the large attribute space, these classifiers will not scale well for large deep neural networks.

3.4 Fuzzy Neural Networks

As described in [Baraldi and Blonda, 1999a] fuzzy rules are if-then rules which operate on a transformed attribute space (also denote as *generic state*). The assignment of the generic state is covered in fuzzy set theory. Fuzzy neural networks are neural networks which model fuzzy rules as multi-layer neural networks. These neural networks can be trained and transformed into fuzzy rules.

Fuzzy neural networks can be roughly seperated into three parts: the first part maps the attribute space to the generic state; the second part models the symbolical rules as a neural network function; the third part maps the generic state

to the target attribute space. According to [Leng et al., 2005], two kinds of learning can is of learning can be applied in fuzzy neural networks, structure learning and parameter learning: “The parameter learning makes the network converge quickly [...]. The structure learning attempts to achieve an economical network size with a [...] self-organising approach.” [Leng et al., 2005, p. 218]. One fuzzy neural network topology which is called ANFIS is introduced in [Jang, 1993]. It has a five-layer topology. The first layer is a gaussian bell-shaped function layer which models the membership function. The second layer is a multiplication layer and corresponds to the logical AND function. Layer three is a normalization layer, layer four is a consequent layer corresponds to the then part of a fuzzy rule. Layer 5 is a summation and normalization layer which is similar to the softmax function.

Another fuzzy neural network is the SOFNN (see [Leng et al., 2005]). It uses a special neuron denoted as EBF neuron, which models the a fuzzy rule. An EBF neuron itself consist of two layers, one layer which has one or more nodes, and a second layer which consists of a single node. The nodes of the first layer have a gaussian bell-shaped activation function which is applied to the inputs of the EBF neuron. The neuron of the second layer also applies a gaussian function. Nodes of the second layer are dynamically created and correspond to the selection of features in an if-then rule.

3.5 Further methods

Other methods that combine connectionist and symbolical methods are used in [rey Mahoney and Mooney, 1993]. Here, a method has been created that combines rules from a knowledge base in a connectionist way. Then, backpropagation is used to refine existing rules and add new connections by expanding decision trees. Even though the adaption phase of the rules is of interest, the method requires prior knowledge and does not arrange rules in a layered way.

4 Multi-Layer Rule networks

In chapter 2, Neural Networks and Rule Learning Algorithms have been introduced as two seperate concepts, that can be used in classification problems. It has been explained that Feed-Forward Neural Networks are usually constructed hierarchically by creating one input layer, one or more hidden layers, and an output layer. Additionally, to the final concept at the neurons of the output layer, intermediate concepts are trained at the input layers. For datasets with large numbers of attributes and samples, the amount of hidden layers and hence the amount of intermediate models can be increased. While Neural-Networks provide ways to adapt the model size to the complexity of the training data, there is no way in common symbolical methods to construct hierarchical topologies with symbolical rules.

Besides weakness of not being able to train hierarchical models, according to [Cohen, 1995], decision trees and decision list are prone to noisy data. This weakness can be well illustrated by examining decision trees. Here, only a small number of attributes are checked for a prediction, compared to a neural network, where all available attributes are checked. The problem for decision trees is, that the model size (i.e. the number of nodes of the decision tree) grows exponentially with the average number of attributes that are checked at each prediction. As an example, a balanced decision tree with 10 conditions needs space for $2^{10} = 1024$ Leaf Nodes and $2^{10} - 1 = 1023$ inner Nodes. A balanced decision tree with 20 conditions already needs $2^{20} = 1,048,576$ leaf nodes and $2^{20} - 1 = 1,048,575$ inner nodes. In conjunction with the large model, the coverage of the training data for each leaf would be evanescently small. In practice, the training algorithm would stop at a certain point, so that the number of conditions remains small.

In most problems that can be solved with decision trees, an acceptable result will be achieved by only using a few, but most promising conditions. However, there exist problems, where a decision tree model can only make accurate predictions, if all attributes are checked. An examples is the parity function:

$$y = (\sum_{i=0}^{n} x_i) mod 2, (x_i \in B)$$

For $n = 2$, the parity function is equal to the XOR function. Because all attributes have to be checked for a correct prediction of the parity function, 2^n leaf nodes and 2^{n-1} inner nodes need to be created for complete model. However, for $n > 2$, the parity can be calculated in a hierarchical way by combining XOR functions. The result of two XOR functions is used as attributes for another XOR function. An example for the parity function with $n = 16$ is given in figure 11. Here, a decision tree would need $2^16 = 65536$ leaf nodes. However, the XOR-Function can expressed with 4 leaf nodes. In the example, 15 XORs were used, so that in total only $15 * 4 = 60$ leaf nodes are needed in the hierarchical approach. More general, the number XORs and likewise the number of nodes that are needed in the hierarchical model only grows linearly for the parity function.

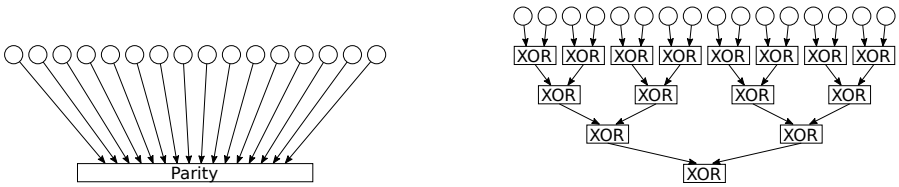


Figure 11: Parity function with 16 inputs, Parity function modeled hierarchically with XORs

The example of the parity function is a motivation to develop methods, that can create hierarchical topologies of symbolical models. In this thesis, such a method will be introduced. Here, a mixed model that consists of hierarchical symbolical and neural layers will be created. This chapter is structured as follows ...

4.1 Relation to earlier works

The idea of combining symbolical methods and connectionist methods like neural networks hierarchically has been addressed earlier. The method that is developed in this thesis can be seen as a further development of three methods, that have been introduced in chapter 3, Fuzzy Neural Networks (see 3.4), Gradient Boosting (see 3.2.1 and the classification of hidden units.

Even though, both symbolical and connectionist methods are present in all three methods, the relation between the connectionist part and the symbolical part varies. In Fuzzy Neural Networks, the structure of Neural Networks is exploited to train fuzzy rules. As a simplification, the fuzzy rules can be expressed as a 3-layer feed-forward structure, where the first and the third layer are a connectionist layer that model the fuzzy membership function, and the second layer is a symbolical layer that models the symbolical if-then-rules. It should be noted, that a set of symbolical rules can be achieved by training a fuzzy neural network, where only the second layer and the third layer are present. While fuzzy neural networks use special layers like RBF-Layers, that are later transferred to symbolical rules, Gradient Boosting

explicitly trains decision trees, with the appropriate training method. In Gradient Boosting, the connectionist part resides in the leaves of the decision trees. The boosted trees can be transformed into a two-layer feed-forward structure, where the first layer contains the decision trees and the second layer contains all weights as the connectionist part. Due to the usage of a backpropagation method, the method can be applied to more than one connectionist layer.

While in Fuzzy Neural Networks and Gradient Boosting the symbolical part and the connectionist part will be trained simultaneously, during classification of the hidden units, the connectionist and a symbolical model a constructed separately. Here, a Feed-Forward Neural Network is created in a first step, and symbolical methods like decision trees or decision lists are used to predict the result of the hidden units. However, there are two drawbacks with this method. Firstly, final model contains errors of two models, the neural network model and the symbolical model. Secondly, a symbolical unit cannot compensate errors, that are generated by another symbolical unit.

In this thesis, the ideas of the three methods picked up. The resulting topology is the same as in the method used by [Zilke, 2015]. While the model is created in two separate training steps, a mixed network of symbolical layers and neural network layers will be created in a single training step here. The mixed network is used as an intermediate result and successively transferred to a hierarchical symbolical topology. The model is trained in mini-batches, and symbolical layers and the neural network layers are refined simultaneously. Like in Gradient Boosting, the gradient descent method will be used to determine an error for each of the symbolical nodes. However, instead of defining the weights to the outputs in the leaves of the decision trees, a binary regression will be performed for each decision tree, and the weights will be defined as a separate layer. By doing this, more than one connectionist layer can be added. The topology bears resemblance to Fuzzy Neural Network topology concerning this aspect. A difference to the Gradient Boosting implementation XGBoost will be, that the number of symbolical units of the symbolical layer will be statically defined in advance, while in XGBoost, new decision trees are added when necessary.

The resulting topology can be summed up as follows. From the neural network point of view, the symbolical rule sets are considered to be a black box pre-classifier. The outputs of the symbolical rule sets are used as inputs for the neural network. From the point of view of the symbolical rule part, the neural network is considered to be a support for the information, that is not yet covered by symbolical rules. On the one hand, the neural network covers all layers, that will be later replaced by symbolical rule layers. Furthermore, the neural network provides a distributed error measure for the symbolical rules as a basis for rule refinement. The Idea of this thesis is to train multiple small decision lists instead of one large decision list. These small decision list are trained in parallel and hierarchically combined to a single decision list. Here, at the upper layers, regressions are performed. The result of the regressions is used as a new input vector for the next layer. A classification is only performed at the final layer. The topology and the training process will be further explained in the following.

4.2 Structure of the mixed network

The mixed neural network that is constructed in this thesis has a feed forward topology. There are two types of layers that are used for the mixed network, Neural Network Layers and decision list layers. NNet-Layer consists of incoming connections to a set of NNet-Nodes. The nodes are transformed by an activation function. At this thesis we assume, that the outputs of the previous layer is fully connected to the nodes of a current layer. During training, the connection weight is updated once each minibatch. Figure 12 illustrates the structure of a neural network layer.

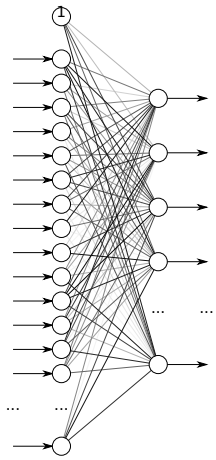
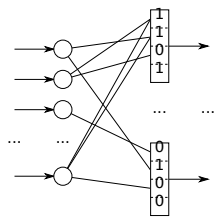


Figure 12: Neural Network Layer

In the decision list layers, each node is a decision list, that contains an ordered list of if-then-rules. In this thesis, a binary class label is chosen for the decision list layer. The results are labeled with ’0’ and ’1’. It should be noted, that

each decision list can have more than one rule with a label '0' and more than one rule with a label '1' in the rule body. Figure 13 illustrates, how a decision list layer is constructed and gives an example for the propagation of inputs of the rule layer to its outputs. Similar to neural networks, the results of rule layers are passed to the following layers of the mixed network.



Decision List 1
(1a) if $in_1 = 1$ & $in_n = 1$ then $out_0 := 1$
(1b) else if $in_0 = 1$ & $in_n = 1$ then $out_0 := 1$
(1c) else if $in_1 = 1$ then $out_0 := 0$
(1d) otherwise $out_0 := 1$

Decision List 2
(2a) if $in_2 = 1$ then $out_n = 0$
(2b) else if $in_0 = 1$ then $out_n = 1$
(2c) else if $in_n = 1$ then $out_n = 0$
(2d) otherwise $out_n := 0$

Example Evaluation
Input vector $in = (1, 1, 1, ..., 1)$ would match Rules 1a, 1b, 1c, 1d and 2a, 2b, 2c, 2d. Due to first order evaluation, labels from 1a and 2a are taken. The output vector $out = (1, ..., 0)$ is passed as an input to the next layer.

Figure 13: a) Structure of a decision list, b) Example decision list

In this thesis, it has been chosen to use decision lists in favor of decision trees. Even though decision trees are a popular choice for symbolical methods and can be trained with a less complex algorithm, they can be adapted less flexible. This is the case when pruning a decision tree, because conditions that reside in inner nodes cannot be removed without also removing complete branches (or reordering the decision tree). Furthermore, if the training data changes, the conditions that were chosen for inner nodes based on previous training data may deteriorate the performance of a decision tree and may need to be pruned. It is assumed, that changes of the training data of the symbolical model will occur quite frequently.

4.3 Topology and training steps

So far, the two building blocks of the mixed network, neural layers and rule layers, have been introduced. While neural layers provide a way to propagate an error from the outputs to the inputs, this is not done with common symbolical algorithms. In section 4.7, a scheme will be developed for propagating an error through the binary decision lists. However, at this point of the thesis, the topology will be constructed without using backpropagation through symbolical rule layers. Besides this limitation, only the feed-forward topology will be examined. A consequence of the lack of backpropagation is, that only one rule layer can be trained at once. For two consecutive rule layers, only the last of the two layers has an error measure, and only this layer can be trained. Besides, it is necessary that the first of two consecutive layers is already trained, so that the inputs of the second consecutive layer already have the information of the intermediate concepts of the first layer.

This observation leads to the following step-wise scheme, were only one rule layer is trained at once. The training process starts with a mixed neural network that has only a single rule layer, which is the first hidden layer (see figure 14). The remaining layers are neural layers. After having trained the network (including the rule layer and all neural layers), the second hidden layer is replaced with an empty rule layer. The network is then trained again, including all layers except the first hidden layers. The procedure is repeated, until the network only consists of rule layers. All rule layers that have been trained in previous steps are kept static. In contrast, all neural network layers are trained again, so that they adapt the rules, that are added iteratively. An example of the training order is given in figure 14.

At this point, the role of the final layer should be further examined. In neural networks, a node is created for each possible class. A common activation function of the final layer is the softmax function, for which the output then is the confidence for the returned class. Decision lists (and decision trees) in contrast can have multiple labels without having to create multiple models for each class. Here, if there are two or more rules with different labels that can fire, the order of the decision list determines which rule is evaluated first and therefore which label is returned. In this thesis, the final layer therefore has been classified with a common algorithm like RIPPER and C4.5.

Up to this point, the structure of the mixed network has been explained. In the following, the implications of the topology for the training process will be further dealt with.

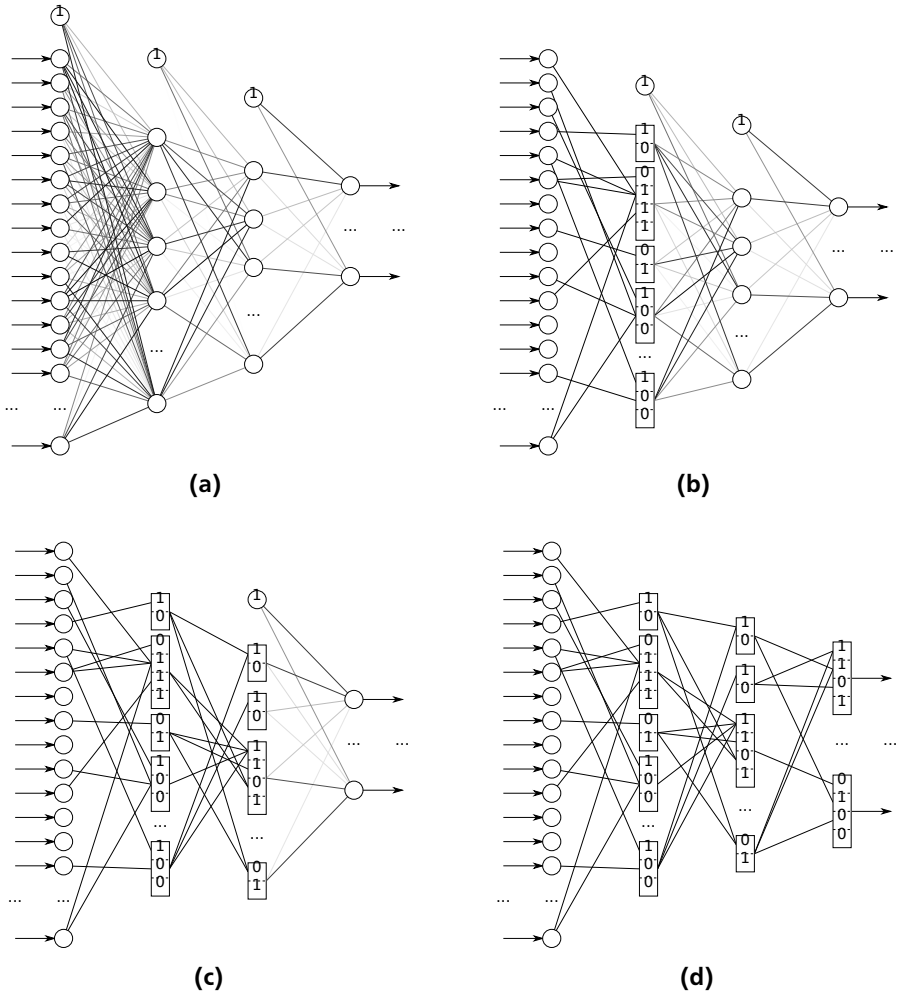


Figure 14: Training steps of the Mixed Network

4.4 Refinement of rule layers

Before examining the training process of the complete mixed network, the training process of the layers is further looked at. As in pure neural networks, the neural layers are trained by adapting the weights based on the error gradients. The training process has been described in chapter 2 and is applied for all neural network layers. While the training process of the neural layers has not been changed, the training process of the rule layers differs from commonly used rule learning algorithms.

In this thesis, a mini-batch scheme for training the neural layers should also be applied for the rule layers. At each time, the decision list should cover the complete input space. As a consequence, rule layers will be initialized with a starting solution and refined with each mini-batch. It was decided to choose, a general-to-specific scheme, so that the initial decision list consists of a default rule with no condition. Rules are then refined by adding conditions and new rules. This will be explained in detail below.

Here, it has been chosen not to follow the training process of RIPPER. In the initial phase of RIPPER, decision lists are constructed by successively adding rules. The input space is not completely covered until the initial phase is completed. A low coverage of the input space is a problem for training the neural layers, because no weight adaption can be applied for missing inputs. Another problem of rule-wise training is, that especially in the beginning of the training, the neural layers will be heavily changed. The training of rules with many conditions will probably be very ineffective at this point, so that rules will have to be pruned and regrown at a later time. The heavy changes of rules also adds a dynamic, which should be avoided in RIPPER.

The usage of a different algorithm than RIPPER also comes with a possible drawback concerning the accuracy of the generated rules. However, the neural network will probably balance the errors that are made at the rule layer, so that some errors that cannot be compensated by one rule could be reduced by another rule. Unfortunately, the errors of the final layer cannot be balanced by a neural layer. Here, the used algorithm will probably have its drawbacks.

In the following, it will be explained how the decision lists are created in this thesis. The decision lists are trained here iteratively by applying the refinement operations *grow*, *prune* and *cut*. The *grow* and *prune* operations have been previously defined in chapter 2. In this thesis, an operation that removes a complete rule from the decision list is denoted as a *cut* operation. While existing training algorithms are usually divided into a *grow* (and a *prune*) phase, where only *grow* operations (or *prune* operations respectively) are applied, the operations here can be applied in no predefined order. This is done, so that rules can faster adapt to changes in the neural layers. As an optimization criterion, the misclassification error has been chosen. Based on this decision, the improvement that can be achieved is only determined by the samples, that are classified with a different label after a refinement operation.

The chosen method of refining rules has a few implications on the operations that are applicable and on additional changes that have to be applied. It is intended to avoid constellations, that could block the rule generation process. After a *grow* operation of the default rule , a new empty rule has to be appended to the decision list. The new rule then matches all samples that are not covered by any previous rule. Due to the setup, where only '0' and '1' labels are used, the label of the new default rule is set to the opposite label of the old default rule. In the implementation of the proposed algorithm, it is made sure that two default rules with opposite labels exist. This way, the improvement of a *grow* operation for the first default rule can be determined more easily.

While the *grow* operation adds complexity to the model, a *prune* operation can be used to revoke early decisions, that do not contribute to the accuracy of the model at a later time. One constellation that should be avoided during a *prune* operation, is an intermediate rule that has zero conditions. The consequence of such a rule is, that this rule will always fire, and later rules will never be evaluated. Further rule refinements only affect earlier rules as long as the intermediate rule contains no conditions. If this is the case for a longer period of time, the rules that are followed by the zero condition rule may drift apart from the rest of the decision list. Growing the rule then will become less likely. A prune operation which results in an empty rule can be therefore only applied, if the rule is immediately followed by a default rule, that has a different label.

Similar to the restriction of *prune* operations, a cut operation is not permitted, if the resulting decision tree would contain a rule, that is followed by a default rule with the same label.

4.5 Training of rule layers

After having defined the structure of the rule layers and the possible refinement steps, the basis on which grow and prune operations are applied will be further focused. The main objective of choosing a refinement operation is the expected improvement of the operation. In section 4.6, it will be explained how the expected improvement for a sample will be calculated, based on the backpropagated error and the change of a label due to a refinement operation. The improvement of a refinement operation is then determined by summing up all expected improvements among the samples.

A crucial difference between the training of rule layers with this algorithm and the training of symbolical models with other algorithms will be, that neural layers and rule layers will be trained simultaneously. The model will be trained in mini-batches. At each mini-batch, the weights of neural layers are adapted and decision lists are refined. Among all possible refinements of all decision lists, only the refinements with the highest improvements will be applied. Moreover, only few decision lists will be affected by the refinement operations at one mini-batch. In contrast to neural networks, where all weights are changed by a small amount each mini-batch, a change by a refinement results in a maximum change of the activation value for the samples affected. After doing refinements, it may it will be necessary to adapt the neural layers to compensate the error, that may occur for some samples after rule training and ensure, that the distributed error is calculated correctly.

While symbolical methods use the complete training data to estimate the improvement of refinement operations, estimating the improvement of refinement operations based on mini-batches would be too inexact. This is because mini-batches only contain a small portion of the available training data. As a consequence, grow operations would add many conditions, that would have to be pruned at a later time. Furthermore, prune operations could prematurely remove important conditions. To overcome this, the improvements of grow, prune and cut will be collected over multiple mini-batches.

For prune and cut operations, an exponential moving average has been chosen

$$\overline{imp_t} = \alpha \cdot imp_t + (1-\alpha) \cdot \overline{imp_{t-1}}$$

where $\overline{imp_t}$ is the average improvement of a refinement operation of the mini-batch at time t, and imp_t is the improvement at time t. The parameter $\alpha < 1$ determines the size of the moving average. The expected improvement of prune and cut operations can be efficiently calculated in $\mathcal{O}(|conditions|)$ and $\mathcal{O}(|rules|)$ respectively for each mini-batch. In the following, a condition, that can be pruned, and that has an associated estimation for the improvement of a prune operation, will be denoted as a prune operation. Likewise, a rule with such an estimation will be denoted as cut candidate and a condition that can be grown will be denoted as grow candidate.

Whereas a list of prune candidates and cut candidates can be easily maintained, keeping a list of all possible grow operations requires a large amount of memory and computational time. The complexity for this operation is $\mathcal{O}(|rules| *$

$|attributes|$) and exceeds the complexity of $\mathcal{O}(|attributes|^2)$ which can be achieved with neural networks. To reduce the complexity, only the most promising grow refinements will be monitored, while the rest is being discarded. More precisely, at each mini-batch, a condition for a possible grow operation is added to a list of grow candidates with a probability that is proportional to the improvement of a single sample. The list of grow candidates is then updated for a defined number of batches. If a low number of batches is chosen, the estimated improvement may become imprecise. A high number of batches may result in outdated estimations, if there are large changes in other decision lists and in the neural layers.

Algorithm: Rule Layer Refinement

Variables : S: Samples of the mini batch
RS: Set of Rulesets, which form the rule layer
n: Number of Layers
H_i: Hidden Layer $i, i \in \{0, ..., n-2\}$
O: Output Layer
$first\_rule$: First Rule of a Ruleset R, which matches s
$second\_rule$: Second Rule of a Ruleset R, that matches s

Functions: $improvement\_grow(r,s)$: Reduction of error, that is achieved by a grow operation of r -> r', if r matches s and r' does not match s;
$improvement\_prune(r,s)$: Reduction of error, that is achieved by a prune operation of r -> r', if r does not match s and r' matches s

; **forall** *Sample* $s \in S$ **do**
  Classify s ;
  Backpropagate the error from O through H_i to RS to obtain err_s ;
  $first\_rule := NULL, second\_rule := NULL$;
  forall *Rule* $r \in R$ **do**
    if *s matches r* **then**
      if $first\_rule = NULL$ **then**
        $first\_rule := r$
      else if $second\_rule == NULL$ **then**
        $setsecond\_rule := r$
      calculate $improvement\_grow(first\_rule, s)$, under consideration of the target values of $first\_rule$, $second\_rule$ and err_s ;
    end
  end
  forall *Rule* $r \in R$ **do**
    calculate $improvement\_prune(r,s)$, under consideration of the target values of r, $first\_rule$ and err_s ;
  end
end

Algorithm 1: Rule Layer Refinement

Algorithm: Grow Operation

$improvement := \{\}$ Stochastically get grow candidates $GC : \{(r, r') | r, r' \in R\}$;
forall $r, r' \in GC$ **do**
  set $improvement(r, r') := 0$;
  forall $s \in S$ **do**
    if *r matches s and r′ not matches s* **then**
      $improvement(r, r') := improvement(r, r') + improvement\_grow(r, s)$;
    end
  end
end
get r, r' with $MAX(improvement)$ $R := R$ r $R := R + \{r'\}$

Algorithm 2: Grow Operation

Algorithm: Prune Operation

forall *Rule* $r \in R$ **do**
  forall *Predicate* $p \in r$ **do**
    $improvement(r, r\ p) := 0$;
    forall $sinS$ **do**
      if r *not matches* s *and* r' *matches* s **then**
        $improvement(r, r\ p) := improvement(r, r\ p) + improvement\_prune(r, s)$;
      end
    end
  end
end
get r, r' with $MAX(improvement)$ $R := R\ r\ R := R + \{r'\}$

Algorithm 3: Prune Operation

4.6 Error estimation

In this part, it will be explained how an error is calculated for the outputs of the rule layer. It will be further explained, how an improvement for a sample is estimated, based on a change of the label due to a refinement operation. For the purpose of determining the error in rule layers, it will be shortly recapped, how error estimation for hidden units in neural networks is realized. As a first step, the output error is calculated. The output error is then propagated backwards through the neural network by applying the derivative functions in the inverse order of the forward pass. Finally, for each weight, a delta is calculated. The delta is multiplied by the learning rate (usually about 1%) and added to the existing weight.

While in neural networks, an error at the output neurons can be traced back to the weights of all hidden units, the concept cannot be easily adapted to propagate an error from the output of the decision lists to the rule conditions. Firstly, symbolical rules use predicates to make boolean decisions. If a predicate should be added or removed, this could only be realized in a stochastically way. Secondly, for decision lists, the rule order of a rule set is crucial to the result of a rule evaluation. Therefore it is also of importance, which rule is altered. To overcome these difficulties, a method is followed that bears much resemblance to Gradient Boosting. Here, the error is only backpropagated to the outputs of the decision list layer, and used as the criterion for refinement operations. In other words, refinement operations will be performed to compensate the backpropagated error. At this point it is expected, that reducing the largest sum of error yields the best improvement. Likewise, if a change is made that points to the opposite direction of the error, a decline is expected. This leads to the following metric:

$$imp_{ref,dl} = \sum_{s \in Samples} -err_{s,dl} * change_{s,ref}$$

where $imp_{ref,dl}$ is the improvement of a given refinement operation ref and a given decision list dl, $err_{s,dl}$ is the error of sample s, that has been backpropagated to decision list dl. $change_{s,ref}$ here is the label change, that will be induced by refinement operation ref and sample s.

The error backpropagation method has been successfully applied for training the decision list layers. A problem that occurred was, that after some time, there were dead decision lists, which still contained no condition. Further training then lead to no change for these decision lists. After some tests, a dynamic has been identified, that leads to dead decision lists. Due to the stepwise refinement, there were decision lists, where no grow operation has been applied after some time. These decision lists with no condition always return a constant value. Furthermore, the neural layers increased the weights to other decision lists, which had already some conditions. Due to the higher weight, the backpropagated error has been higher, if a decision list already had contained some conditions.

As a countermeasure, a normalization has been applied to the backpropagated error. For each decision list, the backpropagated error is divided by the sum of outgoing weights. [1] Here, it has also been observed, that errors of different layers vary. E.g. if a decision list at layer i is trained, the average error at layer i varies from the average error at $i+1$, which is observed when training the decision lists at layer $i+1$. Here, an additional normalization of the error is applied, so that the average absolute error is normalized to a value of 1.

[1] XBoost Paper also states that increasing small weights increases the probability for a refinement operation. However, weights are only increased initially to increase the probability for a split. For older rules, they state, that a rule is considered to be less important. Therefore it is not further refined. In this thesis, the goal is to gain a balanced size of the rules at each layer. Therefore, small decision lists are more interesting split candidates, in order to avoid imbalances.

Identity Function:

in_i	out_j
1	1
0	0

Inverse Function:

in_i	out_j
1	0
0	1

Constant ($out_j = 0$):

in_i	out_j
1	1
0	1

Constant ($out_j = 0$):

in_i	out_j
1	0
0	0

Function	Derivation
Identity Function	1
Inverse Function	-1
Constant Function	0

Table 3: Caption here

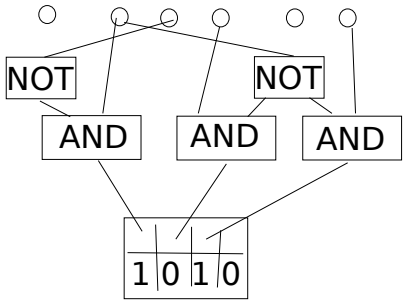


Figure 15: Decision List Scheme

4.7 Backpropagation through rule network

So far, the decision list network is constructed layerwise, so that all neural layers are being replaced iteratively by decision list layers. While during layerwise construction, only the neural layers are used for determining an error for the decision lists, a symbolical backpropagation scheme will be introduced here. The new backpropagation scheme will allow to get an error estimate at the inputs of a decision list layers. Moreover, the error can be propagated through all rule layers.

As a first step, the hidden units should be further focused. At each hidden unit, a regression is performed. While the regression in the neural layers operates on a continuous scale, the regression in the rule layers is applied on a binary scale. The symbolical derivation will be based on theoretical considerations. Here, it is assumed, that only one input is changed at a time. The other inputs are considered to be constant. This assumption makes it possible to decompose the rule function into binary logical functions with one input and one output (see table 15). Each of the functions used to model the decision lists can be reduced to one of the following three functions:

1. Identity function:
   A change at an input results in the same change at the output. The derivative of the identity function is 1. Therefore, the error of the output is passed to the input.

2. Inverse function:
   A change at the input results in the opposite change at the output. If the input is true, the output is false and vice versa. The derivate of the identity function is -1. Therefore, the negative error of the output is passed to the input.

3. Constant function.
   A change at the input results in no change at the output. The derivation of the constant function is 0. Thus, the error at the output results at a zero error at the input.

To achieve one of those three binary functions, the decision lists are first decomposed into three functions, a logical *NOT* function, a logical *AND* function, and a decision list function that models the first order nature of the decision lists.

In the method described, only boolean attributes are accepted. Boolean attributes can be used directly as features in rule heads or inverted before by a logical *NOT* operation. These features are combined as the head of the rule with a logical *AND* operation. If the result of the *AND* operation is true, the rule fires. Depending on which rules of a rule list fire and on the body of the rule lists, the result of the rule lists is determined. Though possible, this function is not further decomposed into logical components. Instead, it will be directly replaced by the identity, inverse or constant function for the case that all but one inputs are constant.

AND

If all inputs of the *AND* operation are true, the result is true. Otherwise, the result is false. Here, we will distinguish 3 cases:

(1) all inputs are true,

(2) all inputs except one are true,

(3) more than one inputs are not true.

In case (1), the result of the *AND* operation can be changed from true to false by changing one of its inputs from true to false. Hence, the change that is done at one input results in the same change at the output. Therefore, it is assumed that the error of the input is equal to the error of the output here. In case (2), the result of the AND operation can be changed from false to true by changing one input from false to true. Changing one of the inputs that are true to false has no effect on the result. Therefore, it is assumed that the error of the input that is set to false is equal to the error of the output. For all other outputs, an error of 0 is assumed. In case (3), the output of the *AND* operation cannot be changed by changing only one of its inputs. Therefore, the error of the inputs is set to 0 here.

NOT

The logical *NOT* operation has one input and inverts its value. For a true input, the output value is false and vice versa. For the error, it is assumed, that the input is inverted.

Decision list function

As already stated before, the decision list can be seen as a set of boolean function, when determining the gradients. The output of the *AND* operation of the rule heads are the inputs for these functions. It has to be checked if the output of the decision list changes, if one of the rules inverts its result, i.e. a rule that currently fires will not fire, and a rule that currently does not fire will fire. Because of the first order manner of the decision lists, the following procedure can be applied:

1) The rules of the decision list are evaluated in order until two rules are found, that fire

2a) Collect all rules that were evaluated before the first firing rule and that have a different rule body. If the body of the not firing rule is false and the body of the first firing rule is true, the error of the output is passed to the head of the rule. If the body of the not firing rule is true and the body of the first firing rule is false, the negative error is passed to the head of the rule

2b) Next, the first two firing rules are evaluated, if the body of the first firing rule and the body of the second firing rule differ. If the body of the first firing rule is true and the body of the second firing rule is false, the error is passed from the output to the head of the first rule. If the body of the first firing rule is false and the body of the second firing rule is true, the negative error is passed from the output of the first rule to its head.

4.8 Combined Training

Initially, the training of the decision list network was done layer-wise, so that only one rule layer was trained at a time. Now, by using the symbolical backpropagation method, a decision-list-only approach, where no neural layer is constructed could be used. However, all decision lists would be initialized with zero conditions. Therefore, all decision list functions would be constant in the beginning. Hence, all derivations would be zero, and no error would be propagated through the decision-list-only network.

Instead of training a decision-list-only network, it has been decided to train a network, that still uses neural layers to gain a distributed error for all decision lists. The error will now be calculated by voting the error of the neural layers and the error of the rule layers. The network for the combined training is created as follows. First, a topology is created, that only contains decision list layers. For each decision list layer except the first hidden decision list layer, an additional neural network layer is created.

1. Forward activate all decision list layers $DLL_0, DLL_1, ..., DLL_n$

2. Backpropagate last decision list layer DLL_n and get $err_{DL,n-1}$

3a. Activate last neural layer NL_n with output of DLL_{n-1}

3b. Backactivate error of last neural layer NL_n and get normalized error $err_{N,n-1}$

4. Generate voted error err_{n-1} by adding $err_{N,n-1}$ and $err_{DL,n-1}$
5. Proceed steps 2-6 with the previous layer. The following generalizations are made for layer i in steps 2. and 3.:

2'. $err_{N,i-1}$ is gained by backpropagating the voted error err_i

3'. The neural network that consists of layers NL_i, NL_{i+1}, ..., NL_n is used

5 Evaluation and Results

In this chapter, the method that has been introduced earlier will be evaluated and compared. First, some training parameter adjustments will be performed and discussed. The evaluation starts with a two-layer model and will be successively performed for deeper topologies. The results will be compared to Jrip (a RIPPER implementation) and J48 (a C4.5 implementation). As a second reference, the method of [Zilke, 2015] will be used. More detailed, a multi-layer neural network will be trained. The hidden unit activations, that result for the training set will be re-trained with a the regression tree algorithm REPTree. The final layer will be trained with Jrip and J48.

The method that has been introduced will be evaluated in a similar way. All rule layers except the final rule layer is trained with the mixed network. The final layers will then be trained with Jrip and J48.

5.1 MNIST Dataset

The method described in chapter 4 will be evaluated here with the dataset MNIST. MNIST consists of 70000 images of handwritten digits, which were centered and normalized to a resolution of 28x28 pixels. All images are further hand-labeled with the target numeric value. The images are separated into a training set of 60000 samples and a test set of 10000 samples. According to [LeCun et al., 1998], the samples were originated from two datasets, SD-1 and SD-3, where SD-1 was collected among high school students and SD-3 was collected amount Census Bureau employees. When separating the data, it was ensured, that a writer only contributed either to the training set or the test set. The 60000 training samples were further divided for training neural networks into a training set of 50000 samples and a test set of 10000 samples (see [mni,]). Here, the training set is used to adjust the network weights, and the validation set is used 'for selecting hyper-parameters like learning rate and size of the model' [mni,]. The validation set is also used for detecting, when the training algorithm converges to a local optimum.

The MNIST dataset suits well as a task that can be accomplished by the new method introduced. It has many training samples compared to other datasets. Besides, the number of attributes is large, so that existing decision tree and decision list classifiers which create a single layer model will perform worse compared to the multi-layered approach.

While the source images of the MNIST dataset are gray-scale images, where each pixel is mapped to a float value between 0 (black) and 1 (white), the mixed network is only capable of processing boolean values. Therefore, the continuous scale of $(0, 1)$ has been discretized, so that inputs in the range of $(0, 0.5)$ have been interpreted as 0 (black) and inputs in the range of $[0.5, 1)$ have been interpreted as 1 (white).

For evaluating the algorithms, the tensor framework Theano [Theano Development Team, 2016] is used. Theano is written in python and runs on a CPU or a GPU. While there is a good support for using the CPU, the GPU implementation needs additional setup and does not support all graphic cards. The framework provides methods for numerical operations like dot products of arrays and metrics or fast element-wise operations like the sigmoid function. In contrast to an imperative definition, where a method is created and evaluated for each sample, theano requires a functional syntax and provides methods for deriving and inverting the defined functions. This way, the framework can generate a gradient for the defined neural networks, that is used for the update function of the weights.

On top of Theano, the library theanets[2] is used. Theanets provides an abstraction for various neural network topologies like tied autoencoders or classifcation nnets. Besides, it has utility functions for using common evaluation datasets like MNIST.

Unfortunately, extending theano and theanets with custom element-wise operations is difficult to achieve. Therefore, the training of the rule has been separated from the training of the neural network. Whereas the gradients of weights are symbolically derived by theano, the backpropagation error for the outputs of the rule layer has been manually derived.

5.2 Mixed Network Setup

Training neural networks usually is performed in mini-batches. Each mini-batch, the network is trained with a small amount of samples (e.g. 32 samples). For the setup in this thesis, mini-batches of a size of 320 samples were used. The value was chosen larger in order to better compare samples of a single mini-batch. Both parts of the mixed network were trained alternatingly with their respective training algorithm. Mini-batches for the neural layers and the rule part were created independently, by selecting random samples from the training set.

The topologies that are used include two topologies with a single rule layer and one neural network layer. Here the first tests were done with 256 hidden layers. The comparison of the reference method was done with the topologies 784-256-10 (784 input units, 256 hidden units, 10 output units) and 784-64-10. For topologies with multiple layers, a fixed size of 64 hidden units has been chosen for the last hidden layer, in order to reduce the influence of the final classifier (J48 and Jrip).

For the neural network layers of the mixed network, mainly the default parameters of theanets are used. This includes an evaluation every 10 mini-batches, a minimum improvement of 1% for each evaluation and a patience of 10

[2] https://github.com/lmjohns3/theanets

evaluations, after which the training is interrupted. For training, the algorithm rmsprop is used. Standard parameters for theanets are a training rate of 0.1%, a halflife of 14 training batches. The parameter that was adapted is the size of the mini-batches, which has been increased from a default of 32 samples per mini-batch to a value of 320 samples per mini-batch. This was done, because the decision wether an attribute or its negation is taken as a feature was done based on the samples of a mini-batch.

5.3 Evaluation

In [Herrera et al., 2011], four groups of interest were introduced for subgroup discovery, which can also be applied for the training task in this thesis: complexity, generality, precision and interest. Complexity measures are measures that estimate the simplicity of the model and include the number of rules and conditions. Generality measures include coverage and support and determine to which amount a rule can be applied on the training data. Precision is a measure for correctness and includes the confidence. For subgroup discovery, interest is identified as a measure of ranking subgroups in relation to each other which includes novelty and significance.

During evaluation, two of the four interest groups were covered: complexity and precision. As a complexity measure, the number of conditions is counted among the classifiers. For the reference method, the number of conditions for the regression at the hidden units and the number of conditions for the classifier is summed up. For the method introduced in this thesis, the total number of conditions during regression and the number of conditions at the final classifier are summed up. The precision is measured as the accuracy of the neural network and as the accuracy of the final classifiers.

5.3.1 Regularization

The first test series deals with a regularization factor. In order to keep the amount of conditions small, conditions which do not contribute much to the result are removed. Besides, in first tests, the number of rules per decision list of a layer has been varying strongly. Here, some rules with many conditions and many rules with few conditions existed. However, it is preferred that decision lists have a similar number of rules so that the information gain is uniformly distributed among the decision lists of a layer.

As a countermeasure, grow and prune operations have been adapted. For grow operations, if the number of rules of a decision list is small, a condition is added even if the expected improvement is below average. Besides, more grow candidates were created for small decision lists. For prune operations, a penalty for is added for large rules, so that a prune operation becomes more probable. Here, the penalty is proportional to the number of rules per decision list. A condition is removed in the following case:

$$imp_{cond} > avg(imp) \cdot size_{dl(cond)}/rf$$

Here imp_{cond} is the improvement that is determined for a condition, $size_{dl(cond)}$ is the size of the decision list which includes the condition $cond$, and rf is the regularization factor. For most conditions, the expected improvement in case of a prune will be negative, and thus pruning will lead to a decline. Therefore, conditions are pruned only if a positive improvement will be expected. With regularization, a condition will also be pruned, if the size of the decision list that contains the condition is equal to the regularization factor and if the expected improvement is above average.

For the tests, a topology with one decision list layer and one neural layer have been used. There are 784 input units, 256 hidden decision list units and 10 neural network output units in the mixed network. The outputs of the 256 hidden units have been used as attributes of the final J48 and Jrip layer. For regularization, three measures have been used: one absolute measure for the regularization factor, and two relative measures.

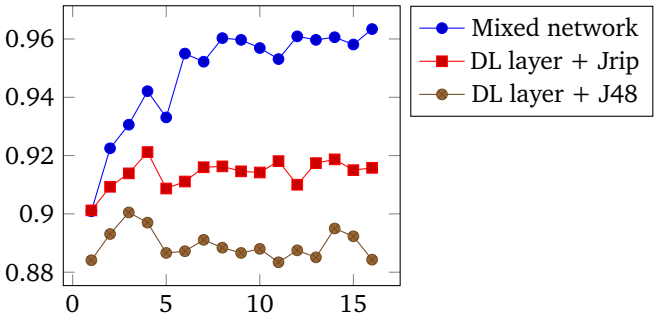
1. Constant $rf = x$ (see figure 16a, b and g)
2. Relative $rf = max(size_{dl}) \cdot x$, which is relative to the maximum decision list size (see figure 16c, d and h)
3. Relative $rf = avg(size_{dl}) \cdot x$, which is relative to the average decision list size (see figure 16e, f and i)

Observations:

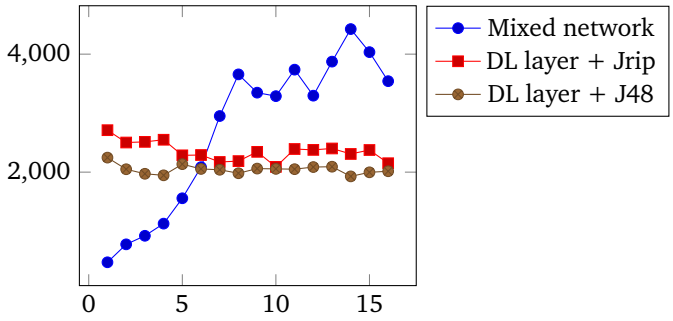
The following observations can be made if all tests are compared with each other:

a) The accuracy and model complexity increases for growing regularization factors. It can be seen, that smaller regularization factors decrease the decision list size.

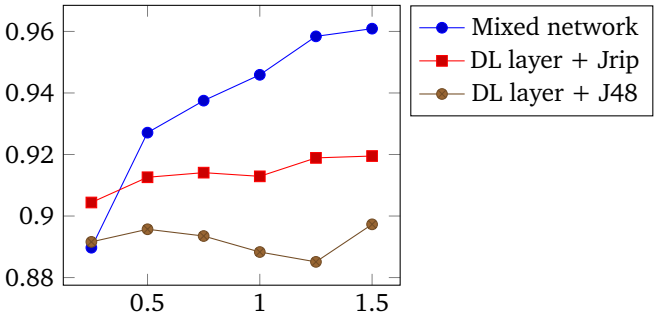
b) The accuracy of the neural network layer exceeds the accuracy of the symbolical classifiers. It can be seen, that the sybolical classifiers are inferior here due to the high number of attributes (256). The accuracy of J48 is worse than the accuracy of Jrip. This is probably because a decision tree has limited conditions that are checked at a path from the root to a leaf. A decision list in contrast can check more conditions.



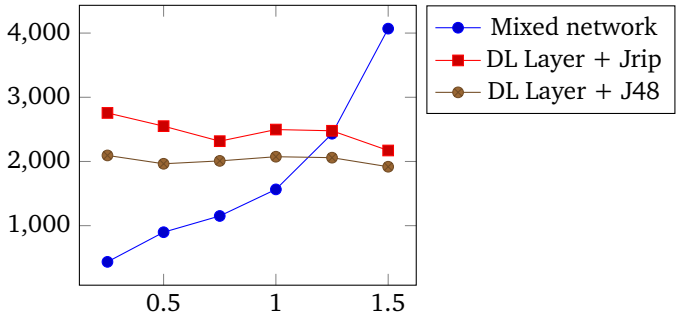
(a) Accuracy for a constant regularization factor.



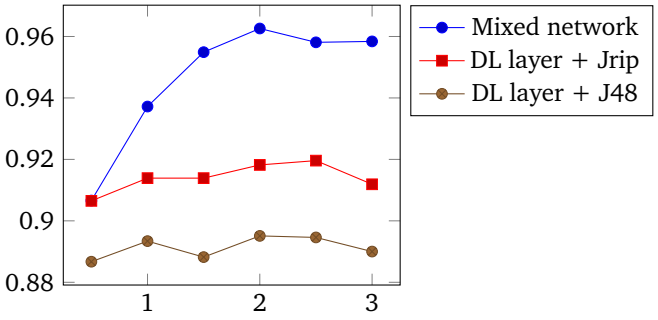
(b) Number of conditions for constant regularization factor.



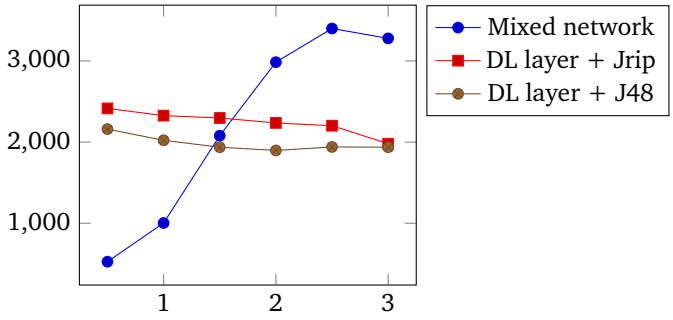
(c) Accuracy for a regularization factor relative to the maximum decision list size.



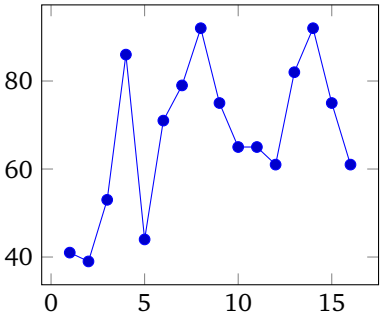
(d) Number of conditions for a regularization factor relative to the maximum decision list size.



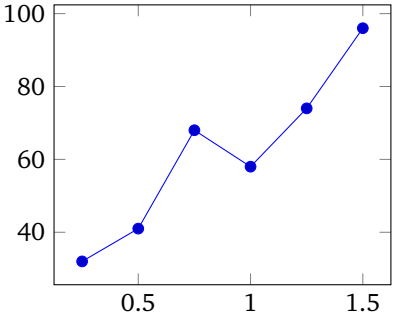
(e) Accuracy for a regularization factor relative to the average decision list size.



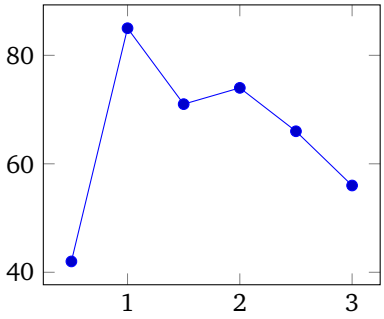
(f) Number of conditions for a regularization factor relative to the average decision list size.



(g) Number of training batches for the mixed network with a constant regularization factor.



(h) Number of training batches for the mixed network with regularization factor relative to the maximum decision list size.



(i) Number of training batches for the mixed network with regularization factor relative to the average decision list size.

Figure 16: Regularization results.

c) With a regularization factor of 7 in test 1), a factor of 1.5 in test 2) and a factor of 2.0 in test 3), the accuracy converges to the accuracy of a model without regularization.

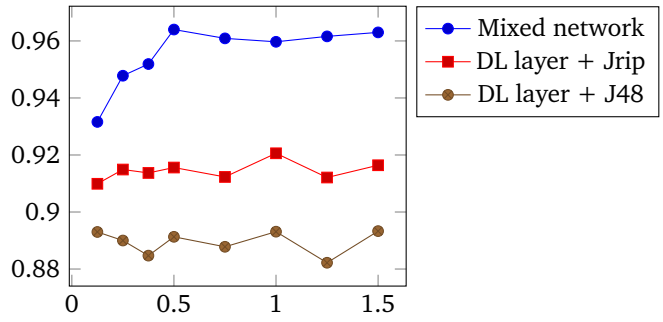
d) While the model complexity of the decision list layer of the mixed network increases, the model complexity of the final classifiers slightly decreases. Here, it is assumed that more information is captured by the decision list layer, so that slightly less information has to be captured by the final layers.

d) The number of training batches differs for the three tests. In the constant test and test which uses the regularization that is relative to the maximum size, the number of training batches increases for higher regularization factors. In case of the regularization that is relative to the average size, the number of training batches have a peak at 1, and decrease after. There might be two effects: on the one hand, with increasing regularization factor, the model might become more detailed, and the accuracy of the neural layer increases for a longer time during training; on the other hand, for the average size regularization, a factor of 1 removes too many conditions, that are recreated again. This might consume additional time. For larger regularization factors relative to the average size, bad conditions are probably removed at an early stage, and this might decrease the number of training batches.

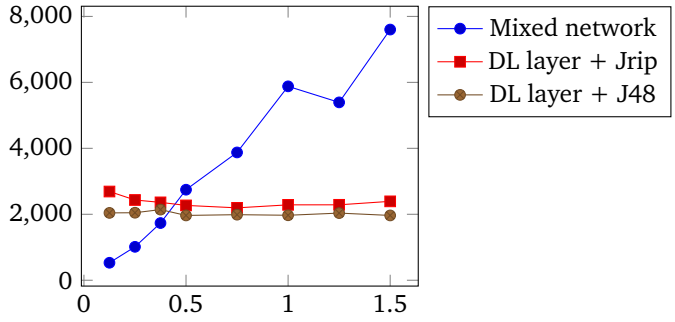
In this test, it has been shown that regularization during prune operations can decrease the model size while the accuracy of the model stays constant. It is concluded, that a regularization that is relative to the average decision list size yields the best results. Here a factor of 2 is a good choice. Other regularization factors also are of use. In the following tests, a regularization has been applied. In some tests, a marginal constant factor of 10 has been used, other tests have been applied with a relative factor.

5.3.2 Grow and prune rate

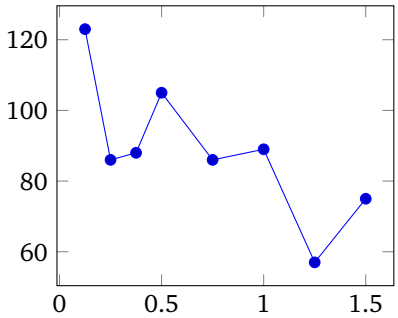
The next tests deal with the number of grow operations, that can be applied per mini-batch (see figure 19). In common symbolical methods, an offline training is performed. Therefore, it could be a problem if many changes are applied at the same time. In the training method that has been developed here, the number of grow and prune operations scales with the total number of rules. This way, the number of grow and prune operations for each rule constant as decision lists become larger. A factor of 1 means, that every rule is changed once per epoch on average. Both, the grow and the prune rate have been adapted, so that the maximum number of prune operations is equal to the number of grow operations for each mini-batch.



(a) Accuracy for different grow rates.



(b) Number of conditions for different grow rates.



(c) Number of mini-batches until convergence for different grow rates.

Figure 17: Impact of different grow/prune rates.

Observations

The following observations have been made based on the tests:

a) The accuracy concerning the neural layer increases until a grow and prune rate of 50% is reached and then stays constant. For prune and grow rates $< 50\%$, the mixed network probably grows too slow, so that the neural layer stops training too early.

b) The number of conditions for the mixed network increases for a growing grow and prune rate. Here, it is assumed that the heavy simultaneous adding of conditions leads to a high complexity of models, which is not reduced later.

c) The number of training batches until convergence decreases with a growing grow and prune rate. Here, it is assumed that adding conditions faster will lead to a faster convergence.

As a result of these tests, concerning the model complexity, low grow and prune rates are favored. Concerning the training speed, high grow and prune rates are favored. As the complexity of the model also has an impact on the training speed, because larger models have more refinement opportunities and take more time to be evaluated, the model complexity is considered to be the more important factor. Therefore, a grow and prune rate between 50% and 100% is considered to be best.

5.3.3 Grow and prune split

Until now, grow operations have been applied on 50% of the data (78 mini-batches) and prune operations have been applied on an exponential moving average. The window has been chosen with $\alpha = 1/78$. Here, the influence is after 78 mini-batches is about three times lower (36.8%), compared to the influence of the sample in the beginning.

Here, three test series are performed and compared (see figure 18): at the first series, the grow and prune size are scaled simultaneously. In the other tests, grow and prune rates are adjusted individually. Higher values will probably increase the accuracy. However, improvement statistics are collected over a longer period of time so that the training speed will decrease. Another problem could be, that the same samples are used for both, growing and pruning. Therefore, overfitting might occur.

In the two tests, where grow and prune sizes are adapted individually, the amount of grow and prune operations had to be adapted. For smaller grow windows, the amount of grow operations that can be applied during an epoch is higher. Here the amount of grow operations per epoch had to be reduced to match the amount of prune operations per epoch. On the contrary, for larger grow windows, a lower amount of grow operations can be applied during an epoch. Here, the prune amount needs to be reduced.

Observations:

The following observations could be made:

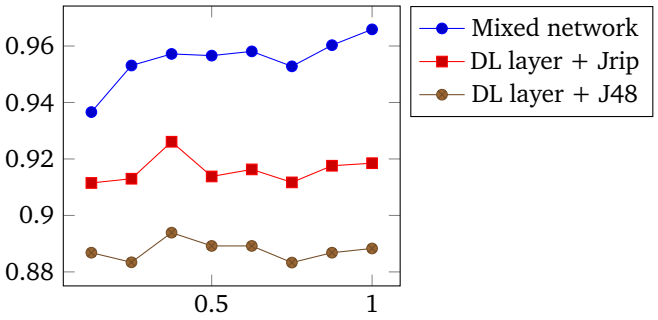
a) During adaptation of both windows, the accuracy has a peek at 35.7% for Jrip and J48 (see figure 18a). Only the accuracy of the neural layer increases afterwards, while the accuracy of Jrip and J48 slightly decline. Here, the behaviour of the neural network part could overestimate the accuracy, because the samples which were applied during prune and grow are overlapping for the window between 50% and 100%. The change of the accuracy of Jrip and J48 could be linked to the model complexity, which has its peak at 37.5%.

b) The number of conditions for the rule layer of the mixed network in figure 18b has the characteristic, that it increases until 37.5% and decreases afterwards. Here, two effects could have an impact. At the begining, the small windows could lead to an inexact model, which leads to an early stop of the training algorithm of the neural network (also see figure 18g). The decline of model complexity afterwards could occur due to the reduced number of grow and prune operations which need to be applied for large grow and prune windows.

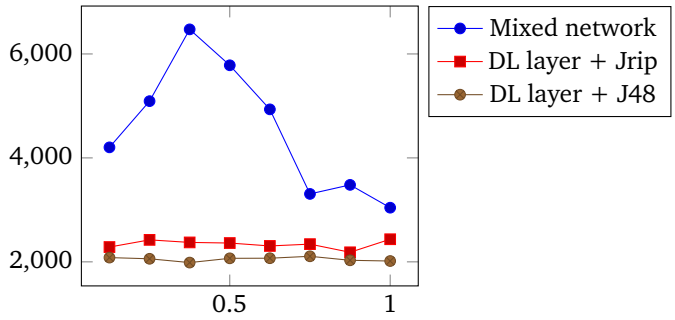
c) In figure 18c and d, there could be observed that with larger grow windows, the accuracy increases and the model size decreases.

5.3.4 Regularization with two layers

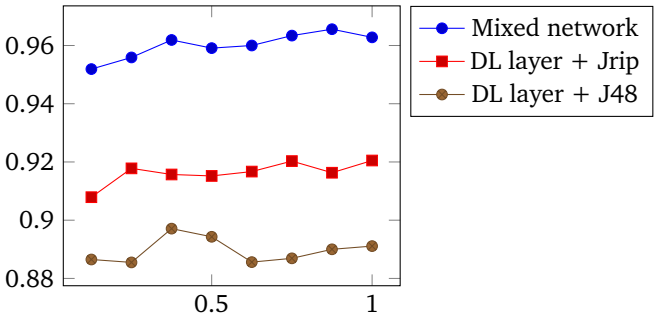
In the following test series the regularization factor with two rule layers is examined. While all decision lists of the last rule layer of the mixed network will probably contribute to the classification result, the other layers will probably contain rules that are not used as conditions by the following layer. Therefore, in the two-layer approach, a smaller regularization factor can be used for all decision lists of the first layer, that do not contribute to the classification result. Here, a constant regularization factor of 10 is used for all decision lists that are used as conditions.



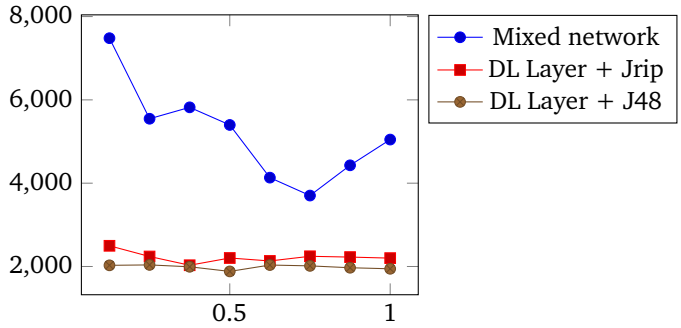
(a) Accuracy for the simultaneous adaptation of grow and prune windows.



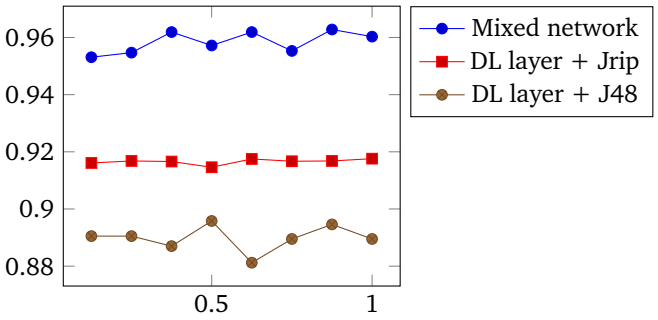
(b) Number of conditions for the simultaneous adaptation of grow and prune windows.



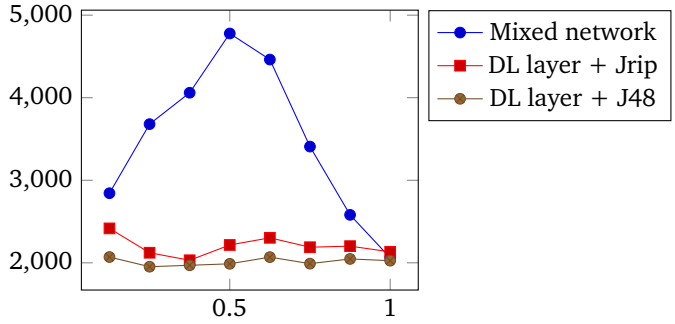
(c) Accuracy for the adaption of the grow rate.



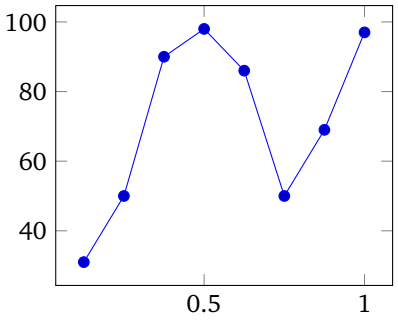
(d) Number of conditions for the adaption of the grow rate.



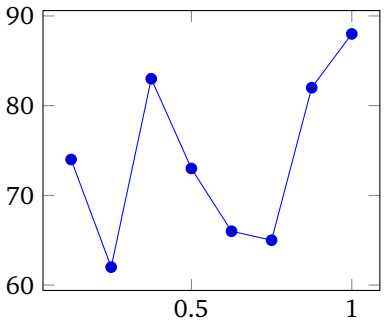
(e) Accuracy for the adaption of the prune rate.



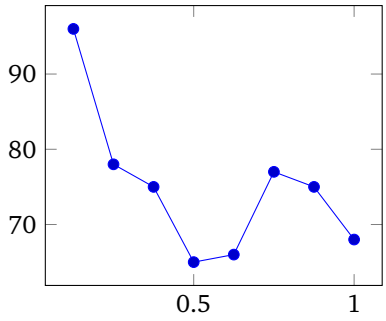
(f) Number of conditions for the adaption of the prune rate.



(g) Number of training batches for the mixed network during simultaneous adaptation of grow and prune windows.

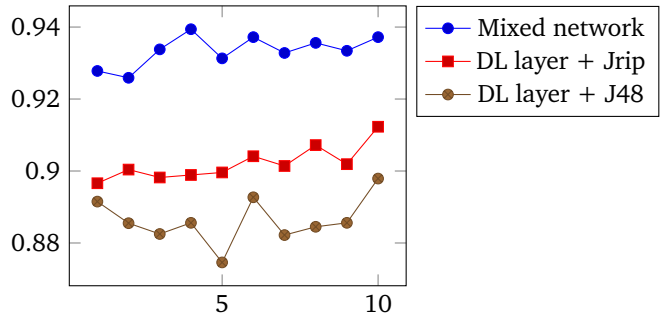


(h) Number of training batches for the mixed network during adaptation of the grow window.

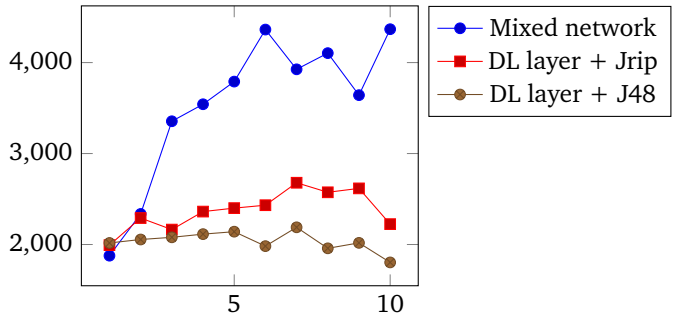


(i) Number of training batches for the mixed network during adaptation of the prune window.

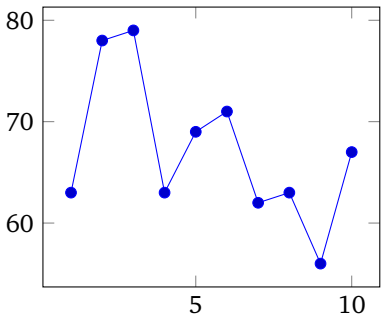
Figure 18: Results for adaption of grow and prune windows.



(a) Accuracy for different grow rates.



(b) Number of conditions for different grow rates.



(c) Number of mini-batches until convergence for different grow rates.

Figure 19: Impact of different grow/prune rates.

Observations:

In the plots, it can be observed that the number of conditions can be reduced whereas the accuracy does not drop much for small regularization factors. However, it was expected, that the unconnected decision list do not have an affect on the accuracy of the mixed network and could therefore be pruned more strongly.

5.3.5 Checking topologies

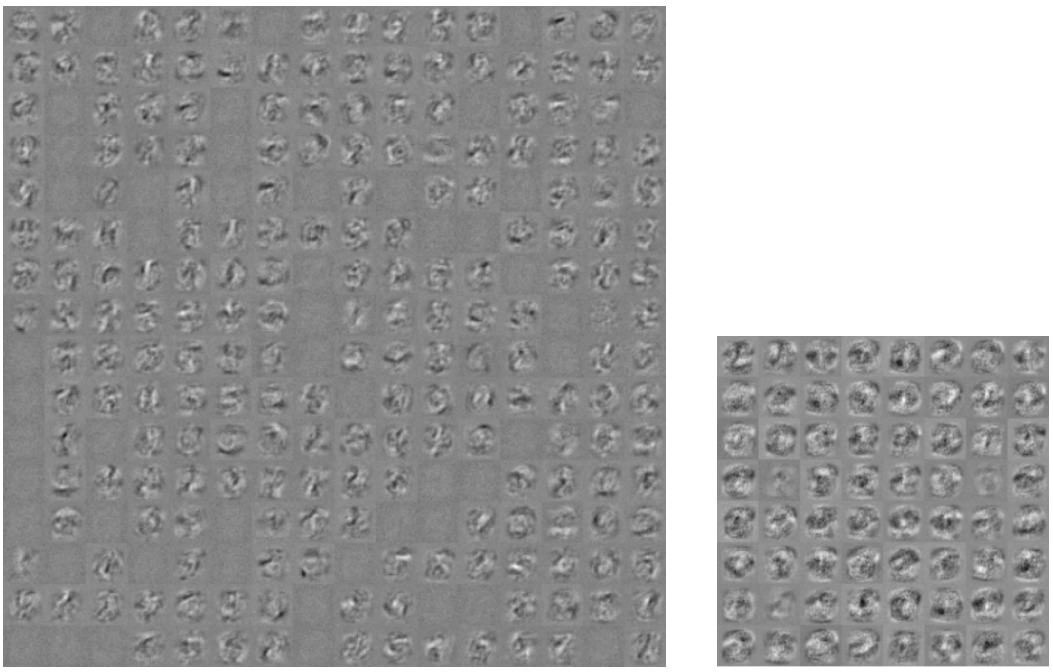
In Table 4, the different classifiers are compared with each other. Here it can be seen, that Jrip and J48 can benefit from the mixed network. In comparison to the pedagogical approach, it can be seen that much less conditions are need for the mixed network. However, the accuracy is worse than the accuracy that can be achieved with the pedagogical approach.

5.4 Visualization of hidden units

In neural networks, the matrixes of neural networks can be plotted, so that a representation of the training result can be visualized. This is done by reverting the trained network and activating single neurons. An example representation for the trained neural network for MNIST is given in figure 20. In contrast to the neural layers, the decision layers cannot be visualized like that. However, the activations of the hidden layers can be checked, and an overlay image of all digits that yield an activation can be created. During the image creation, all images that yield an activation of 1 are taken as they are and all images that yield an activation of 0 are negated. Figure 21 shows the visualization for the mixed network after layerwise training

Algorithm	Topology	Number of Conditions	Number of Conditions (last layer)	Number of mini-batches	Accuracy
Jrip		3451			89.65%
J48		2953			87.12%
Neural Network	784-64-10			104	96.84%
	784-256-10			60	97.51%
	784-256-64-10			71	97.82%
	784-256-128-64-10			45	97.3%
Pedagogical derivation (Jrip)	784-64-10	128000	1661		91.35%
	784-256-10	512000	1530		92.28%
	784-256-64-10	640000	798		93.29%
	784-256-128-64-10	1120000	562		93.52%
Pedagogical derivation (J48)	784-64-10	128000	1673		88.89%
	784-256-10	512000	1574		89.35%
	784-256-64-10	640000	802		91.34%
	784-256-128-64-10	1120000	637		92.55%
Mixed Network + Jrip	784-64-10	1038	2674	86	90.13%
	784-256-10	3978	2092		91.81%
	784-256-64-10	3856-1673-895	2363	57	90.44%
	784-256-128-64-10	3722-1433-769	1987	51	89.12%
Mixed Network + J48	784-64-10	1038	2060	86	88.93%
	784-256-10	3978	1921		88.22%
	784-256-64-10	3856-1673-895	2045	57	88.37%
	784-256-128-64-10	3722-1433-769	1963	51	87.97%

Table 4: Evaluation results. Comparison with other algorithms.



(a) 256 Hidden units of the first hidden layer are plottet in 16 rows and 16 columns

(b) 64 Hidden units of the second hidden layer are plottet in 8 rows and 8 columns

Figure 20: MNIST neural network weight visualization

6 Conclusion and Future Works

This thesis has dealt with training multi-layer feed-forward symbolical rules for classification by exploiting neural network structures. A mixed network has been created as an intermediate structure, that consists of decision list layers and symbolical layers. These two types of layer have been trained simultaneously. After training the mixed network, a final rule layer has been added, which is trained by the C4.5 or the RIPPER algorithm. This way, a multi-layer rule network could be trained.

Another contribution of this thesis is the usage of a symbolical backproagation scheme, which allowed to use back-propagation through decision list layers. A voting of the backpropagation errors from the decision list layers and addition neural network layers allowed a simultaneous training of the rule layers. However, when increasing the number of layers, the simultaneous training of all rule layers did not

The new method has been compared with a method that was introduced earlier, where the decision lists have been trained based on the hidden activations of a pre-trained neural network. The new method has been shown to be superior concerning model size and training speed when only a single layer is used. The training speed could be mainly increased by applying stochastic search methods for new conditions. The model size could be significantly reduced by training decision lists. Concerning the accuracy, the new method has shown to be inferior compared to the reference method. Especially the final layer could not yet be trained with the method.

In future works, the weak decision list training method could be exchanged. Current problems might be caused by the usage of the misclassification error instead of information gain. The decision in this thesis against rule-wise training, which is used in the RIPPER algorithm, might play a role. Besides, the RIPPER algorithm has an optimization phase which is not present in the method of this thesis. Therefore, in future works, an optimization phase similar to the second phase of the RIPPER algorithm could be introduced. During this phase, previously trained rules could be replaced by rules which are trained with the information gain as a split criterion.

The training of the hidden layers could also be improved by performing a regression on a linear scale instead of a binary scale. To realize this, the backpropagation scheme would need to be extended.

Another open issue that could be further looked at is the voting scheme, which is used for determining the error for the combined training. Here, a simple voting scheme has been applied, where the error of the two models has been added up. However, the error that is gained by the neural network layers is normalized to a mean absolute value of 1.0, whereas the error that is gained by backpropagation through the rule layers is not normalized. Besides, some rules will produce a backpropagated error of zero whereas a few rules could produce a quite large error. Here, a further investigation on the characteristics of the two types of error, as well as new schemes for another voting or a normalization of the errors could be the target of future research.

References

[mni,] `http://deeplearning.net/tutorial/gettingstarted.html#mnist-dataset`.

[Aha et al., 1991] Aha, D. W., Kibler, D., and Albert, M. K. (1991). Instance-based learning algorithms. *Machine learning*, 6(1):37–66.

[Baesens et al., 2003] Baesens, B., Setiono, R., Mues, C., and Vanthienen, J. (2003). Using neural network rule extraction and decision tables for credit-risk evaluation. *Management science*, 49(3):312–329.

[Baraldi and Blonda, 1999a] Baraldi, A. and Blonda, P. (1999a). A survey of fuzzy clustering algorithms for pattern recognition. i. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 29(6):778–785.

[Baraldi and Blonda, 1999b] Baraldi, A. and Blonda, P. (1999b). A survey of fuzzy clustering algorithms for pattern recognition. ii. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 29(6):786–801.

[Bianchi et al., 2009] Bianchi, L., Dorigo, M., Gambardella, L. M., and Gutjahr, W. J. (2009). A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing: an international journal*, 8(2):239–287.

[Breiman, 1996] Breiman, L. (1996). Bagging predictors. *Machine learning*, 24(2):123–140.

[Chen et al., 2009] Chen, H., Tiho, P., and Yao, X. (2009). Predictive ensemble pruning by expectation propagation. *Knowledge and Data Engineering, IEEE Transactions on*, 21(7):999–1013.

[Chen and Guestrin, 2016] Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. *arXiv preprint arXiv:1603.02754*.

[Cohen, 1995] Cohen, W. W. (1995). Fast effective rule induction. In *Proceedings of the twelfth international conference on machine learning*, pages 115–123.

[Cover and Hart, 1967] Cover, T. and Hart, P. (1967). Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27.

[Craven and Shavlik, 1996] Craven, M. W. and Shavlik, J. W. (1996). Extracting tree-structured representations of trained networks. *Advances in neural information processing systems*, pages 24–30.

[Dasgupta et al., 2011] Dasgupta, A., Kumar, R., and Sarlós, T. (2011). Fast locality-sensitive hashing. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1073–1081. ACM.

[Dasgupta and Freund, 2008] Dasgupta, S. and Freund, Y. (2008). Random projection trees and low dimensional manifolds. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 537–546. ACM.

[Esfahani and Dougherty, 2013] Esfahani, M. S. and Dougherty, E. R. (2013). Effect of separate sampling on classification accuracy. *Bioinformatics*, page btt662.

[Friedman, 2001] Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232.

[Fürnkranz and Kliegr, 2015] Fürnkranz, J. and Kliegr, T. (2015). A brief overview of rule learning. In *Rule Technologies: Foundations, Tools, and Applications*, pages 54–69. Springer.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). Deep learning. Book in preparation for MIT Press.

[Haykin and Network, 2004] Haykin, S. and Network, N. (2004). A comprehensive foundation. *Neural Networks*, 2(2004).

[Herrera et al., 2011] Herrera, F., Carmona, C. J., González, P., and Del Jesus, M. J. (2011). An overview on subgroup discovery: foundations and applications. *Knowledge and information systems*, 29(3):495–525.

[Jang, 1993] Jang, J.-S. (1993). Anfis: adaptive-network-based fuzzy inference system. *IEEE transactions on systems, man, and cybernetics*, 23(3):665–685.

[Kubat, 2015] Kubat, M. (2015). *An Introduction to Machine Learning*. Springer.

[LeCun et al., 1998] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[Leng et al., 2005] Leng, G., McGinnity, T. M., and Prasad, G. (2005). An approach for on-line extraction of fuzzy rules using a self-organising fuzzy neural network. *Fuzzy sets and systems*, 150(2):211–243.

[Martens et al., 2008] Martens, D., Huysmans, J., Setiono, R., Vanthienen, J., and Baesens, B. (2008). Rule extraction from support vector machines: an overview of issues and application in credit scoring. In *Rule extraction from support vector machines*, pages 33–63. Springer.

[Novak et al., 2009] Novak, P. K., Lavrač, N., and Webb, G. I. (2009). Supervised descriptive rule discovery: A unifying survey of contrast set, emerging pattern and subgroup mining. *Journal of Machine Learning Research*, 10(Feb):377–403.

[Pan and Manocha, 2011] Pan, J. and Manocha, D. (2011). Fast gpu-based locality sensitive hashing for k-nearest neighbor computation. In *Proceedings of the 19th ACM SIGSPATIAL international conference on advances in geographic information systems*, pages 211–220. ACM.

[Peterson, 2009] Peterson, L. E. (2009). K-nearest neighbor. *Scholarpedia*, 4(2):1883.

[Quinlan, 1986] Quinlan, J. R. (1986). Induction of decision trees. *Machine learning*, 1(1):81–106.

[Quinlan, 2014] Quinlan, J. R. (2014). *C4. 5: programs for machine learning*. Elsevier.

[rey Mahoney and Mooney, 1993] rey Mahoney, J. J. and Mooney, R. J. (1993). Combining connectionist and symbolic learning to refine certainty-factor rule bases.

[Schapire, 2003] Schapire, R. E. (2003). The boosting approach to machine learning: An overview. In *Nonlinear estimation and classification*, pages 149–171. Springer.

[Shannon, 2001] Shannon, C. E. (2001). A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55.

[Theano Development Team, 2016] Theano Development Team (2016). Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688.

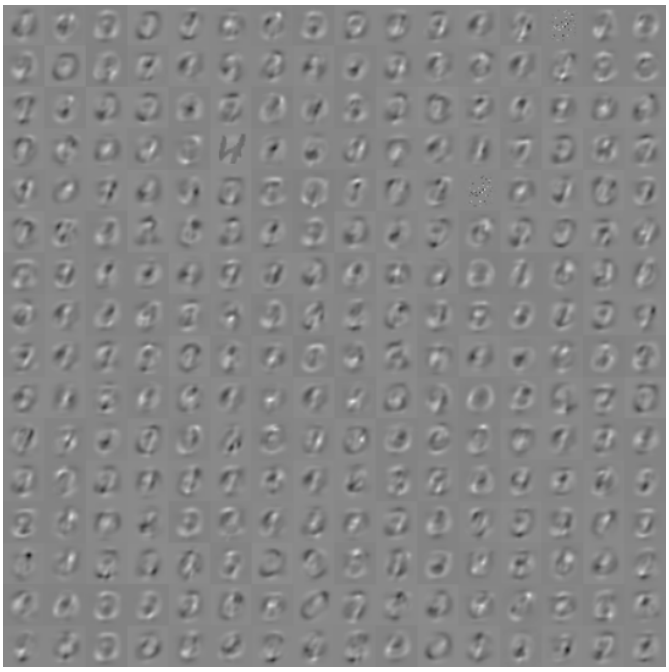
[Tsoumakas et al., 2009] Tsoumakas, G., Partalas, I., and Vlahavas, I. (2009). An ensemble pruning primer. In *Applications of supervised and unsupervised ensemble methods*, pages 1–13. Springer.

[Wolpert, 1992] Wolpert, D. H. (1992). Stacked generalization. *Neural networks*, 5(2):241–259.

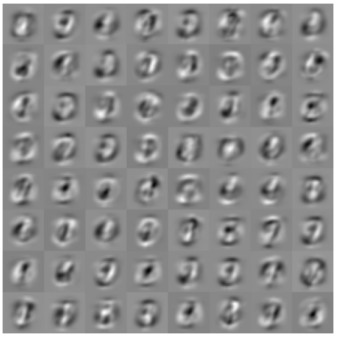
[Xu and Wunsch, 2005] Xu, R. and Wunsch, D. (2005). Survey of clustering algorithms. *IEEE Transactions on neural networks*, 16(3):645–678.

[Zilke, 2015] Zilke, J. (2015). Extracting rules from deep neural networks. unpublished thesis.

[Zöller, 2014] Zöller, B. (2014). Extraktion von regeln aus neuronalen netzen. unpublished thesis.



(a) 256 Hidden units of the first hidden layer are plottet in 16 rows and 16 columns



(b) 64 Hidden units of the second hidden layer are plottet in 8 rows and 8 columns

Figure 21: MNIST neural network weight visualization