University of Basel

Tree Regularization of Deep Networks

Master's Thesis

Faculty of Science of the University of Basel
Department of Mathematics and Computer Science
Biomedical Data Analysis Group

Examiner: Prof. Dr. Volker Roth
Supervisor: Fabricio Arend Torres, MSc.

Gowthaman Gobalasingam
gowthaman.gobalasingam@stud.unibas.ch
2016-050-619

October 25, 2021

Acknowledgments

I am grateful to Prof. Dr. Volker Roth for allowing me to work on this thesis. I appreciate his generous supervision and him sharing his insights and feedbacks in our regular meetings throughout the work. Furthermore, I thank my co-supervisor Fabricio Arend Torres for his guidance. His continuous inputs have steered me in the right direction.

Calculations were performed at sciCORE scientific computing core facility at the University of Basel. At this point, a special thanks goes to the sciCORE support center for helping out to fix several technical problems to properly run the scripts.

Lastly, I thank my friends for their comments and proofreading.

Abstract

In recent years Deep Neural Networks have become ubiquitous in a wide range of Machine Learning applications. While they are capable of modeling highly complex patterns, they often appear as black box models resistant to deeper insight and interpretation. Therefore, many practitioners from critical domains like medicine hesitate to adopt these models. In this thesis, we explore a recent series of publications (Doshi-Velez and Kim [1] and Wu et al. [2]) that focus on constraining highly parametrized deep models by using a novel technique called "Tree Regularization". It optimizes deep models in a way, that enables approximations by decision trees. The resulting trees provide the user a great facility to easily simulate the outcomes of the deep model, and at the same time to understand the model's decision-making. The aim of this work is to study and evaluate, whether the tree regularization is suited as an interpretable model, and assess its performance with feed-forward deep architectures on fairly large synthetic data sets. We also showcase optimization difficulties of the training procedure, which may even lead to limited applicability of tree regularization in practice.

Table of Contents

1 Introduction

> If you can't explain it simply, you don't understand it well enough.
>
> – *Albert Einstein*

Deep Learning lives quite everywhere. For the past few years, it has become the underlying method for many state-of-the-art applications like image classification, speech recognition, machine translation, autonomous driving to the point of life sciences. Using Deep Models became attractive in the current era of Big Data, and they frequently outperform classical Machine Learning techniques to complex problems. However, they fall back in terms of interpretability, and hence it is hard to fully understand the model's decision-making, meaning, how they arrive at their predictions. In some areas, there is hesitation in using such models. Thus, it is required that the model's decisions are justifiable so that a practitioner can simulate the outcomes by himself. In the following section, we describe in detail the notion of *interpretable Machine Learning*.

1.1 Interpretable Machine Learning

The *interpretability* of Machine Learning algorithms has been greatly neglected over the years. Making *black box models* comprehensible to humans has become crucial in critical domains such as medicine or criminal justice. In a nutshell, we require from a model not only to answer what the predictions are, but also *why*, or rather *how* the predictions are inferred [3].

Doshi-Velez and Kim [1] define interpretability as "the ability to explain or to present in understandable terms to a human". There does not exist any mathematical specification of the term interpretable machine learning. Depending on the model's methodology, the interpretability can be perceived differently. Incomplete problem setting can cause poor interpretability, and hence only getting accurate predictions is in some fields not sufficient. Explaining the way of how the results are inferred is crucial to fully gain the user's trust, and to completely understand the problem. The interpretability of a model can be achieved by fulfilling the following auxiliary criteria [1, 4]:

- Fairness/Unbiasedness: Potential groups in the input should not discriminate each other due to some biases

- Privacy: Preserve data protection, e.g., protect sensitive data from patients
- Reliability/Robustness: Reach better performance due to variations of the input and parameters
- Causality: Causal relationship of the outputs and the model, changes in the model causes changes in the results
- Trust: Interpreting black box models attains user's trust

Generally, interpretable models can be categorized in two ways [3], namely as *intrinsic/model-specific* or as *post-hoc/model-agnostic* interpretation. Sparse models, that fall into the first category, result in a constrained structure, e.g., Linear Regression, Logistic Regression or Decision Trees. Model-agnostic interpretation applies methods to feature inputs and outputs for better analysis, and usually we cannot see inside the model, e.g., highlighting feature importance that has contributed to the prediction. This type of interpretability is more flexible. Any method that builds based on interpretation can be used subsequently after the training, e.g., the LIME algorithm [5] (more details in Section 3.3).
Sometimes in the literature, a clear distinction is made between interpretability and explainability. The latter appear under the designation *eXplainable Artificial Intelligence* (XAI), and is closely tied with model-agnostic interpretation. Arrieta et al. [6] provide in their publication basic approaches by means of post-hoc explainability tools, like an interface between black box models and the human, for example visual explanation techniques [6] or feature relevance as stated before.
Lastly, the evaluation takes place by assessing the interpretability of the model according to the following levels suggested by Molnar [3] and Doshi-Velez and Kim [1]. They do not have to be strictly followed, but are potential baselines to get feedbacks:

- Application-grounded Evaluation: Reasoning will be primarily done by domain experts. The model will be provided with explanations. With Hands-on experiments, the user can test and audit the quality of the model.
- Human-grounded Metrics: Reasoning will be done by laymen, in contrast to application-grounded method. Simplified experiments will be provided, e.g., comparing different model explanations.
- Functionally-grounded Evaluation: Formal assessment with a proxy, e.g., (optimized) Decision Trees with smaller depth are seemingly easier to interpret

In the next section, we take a glance at the urgency of interpretable models in healthcare and medicine.

1.2 Interpretability in the Fields of Healthcare and Medicine

The huge set of electronic health records (EHR) allows the usage of Deep Learning models for the patient's diagnosis, also called phenotype discovery. With interpretable models, clinicians would be able to give proper personalized healthcare [7].

Of course, such models are only used as an auxiliary tool to guide the diagnostic processes. Nevertheless, clinical decisions remain critical, and thus one must work with these models with caution. In addition, medical regulations and ethical requirements must be strictly followed [8].
Making decisions in medicine is crucial in order to assign the appropriate therapies to the diagnosis, that were made beforehand. Therefore, to support the decision-making, decision trees are appropriate models to characterize the patient's data [9]. They are intuitive and effective to use as a general model, and are simple to visualize and thus to interpret. Still, they come with some drawbacks. Usually, real-world data are often incomplete (missing features) which causes poor tree building. Another downside is that only one tree can be built for one problem setting. In general, we prefer to have several competing models, and to choose the best one. Therefore, Podgorelec et al. [9] have suggested in their review a hybrid model built of neural network and decision trees, so that we can benefit both from their advantages.

1.3 Contributions

The aim of this thesis is to study, implement and test the so-called *Tree Regularization* proposed by Wu et al. [10]. It was introduced as a novel type of regularization method, with the focus on creating interpretable models from deep networks. With this, high-parametrized complex deep models will be tackled and constrained to a certain degree, such that the predictions can be simulated with decision trees of small depth. It can be categorized as model-specific interpretation, because the model itself will be altered. We first study the concept of tree regularization in detail, and then propose our model along with its practical training approach. Then we evaluate the model's performance, and its outcomes on synthetic data sets, and audit, if the resulting artifact can serve as an interpretable model. At last, we discuss at the end of the work, in the future work part, the challenges, and limitations, that have arisen during the work.

1.4 Outline

The thesis is arranged as follows. In Chapter 2 we summarize all important fundamentals, that are essential to understand Tree Regularization. Chapter 3 contains detailed explanations of the model's structure and its training procedure. Furthermore, we present some related work based on this topic and in general to interpretable models. To test the model, we consider a binary classification problem with synthetic data sets, and evaluate the model's behavior, which are shown in Chapter 4. In the last chapter, we briefly summarize this work, comment on challenges we have faced during the implementation, and provide suggestions for the future work.

2
Background

In this chapter, we describe the on important theoretical principles of our model. We first give a general introduction to Machine Learning, and then we elaborate on the architecture and the optimization methods of Deep Neural Networks, especially feed-forward networks. After that, we describe Decision Trees and their algorithms to build them.

2.1 Machine Learning: General Setting

Machine Learning incorporates a subset of methods from Artificial Intelligence. It is based on the idea of how humans learn from examples and experience. Technically, we build and train specific *machines* or *models*, that can detect underlying patterns of some given data, and apply them to unseen data to make predictions. For example, we feed labeled images of dogs and cats to a machine. After learning, the machine is then able to recognize the correct animal on an image that has not been seen before. Overall, we will be dealing here with a learning task. The following formal setting and explanations to train a machine is, if not otherwise specified, based on Murphy [11] and Vapnik [12].
Generally, we can consider a Machine Learning model as a *black box* learning model. Figure 2.1 shows an overview of it.

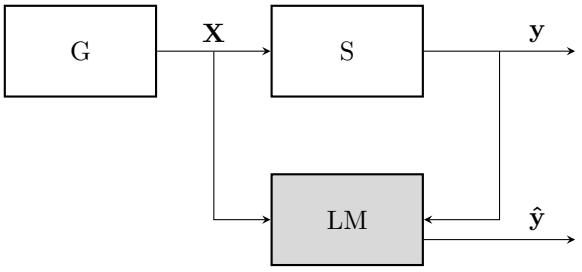


Figure 2.1: Overview of a *black box* model of learning (based on Vapnik [12])

Let $\mathcal{X}$ and $\mathcal{Y}$ be finite or countable infinite sets. The generator (G) draws from $\mathcal{X}$ i.i.d. samples, arranged in a matrix $\mathbf{X} \in \mathbb{R}^{N \times D}$, from an unknown, underlying marginal distribution $p(\mathbf{X})$. The supervisor (S) delivers the associated label vector $\mathbf{y} \in \mathcal{Y}$, and assigns for each data instance $\mathbf{x}_i \in \mathbb{R}^D$ from the row $\mathbf{X}_{i,:}$ the true target label $y_i \in \mathbf{y}$, that are drawn from an unknown conditional distribution $p(\mathbf{y}|\mathbf{X})$. We assume, that the observable pairs

$(\mathbf{x}_i, y_i) \in (\mathbf{X}, \mathbf{y})$ come from a joint distribution on $\mathcal{X} \times \mathcal{Y}$

$$(\mathbf{X}, \mathbf{y}) \sim p(\mathbf{X}, \mathbf{y}) = p(\mathbf{X})p(\mathbf{y}|\mathbf{X}). \tag{2.1}$$

In the *learning process*, we strive to find an appropriate learning machine (LM) $f : \mathcal{X} \to \mathcal{Y}$ in order to minimize the expected risk R, which measures the expected loss $\mathcal{L}$ over the joint distributions of $(\mathbf{X}, \mathbf{y})$

$$R[f] = E_{(\mathbf{X},\mathbf{y})\sim p}\{\mathcal{L}(\mathbf{y}, f(\mathbf{X}))\} = \int_{\mathcal{X}\times\mathcal{Y}} \mathcal{L}(\mathbf{y}, f(\mathbf{X}))p(\mathbf{X}, \mathbf{y})d\mathbf{X}d\mathbf{y}. \tag{2.2}$$

However, the distributions are usually unknown, and hence the minimizer in 2.2 is analytically not tractable. In practice, we extract a finite training set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{M}$ of size $M \leq N$ from the given samples. Each data instance $\mathbf{x}_i$ is a D-dimensional feature vector that describes observable, characteristic properties of the data. The associated label y_i allocates a class or a real number to the data instance $\mathbf{x}_i$. From a frequentist point of view, we try to find a machine $f_{\mathcal{D}}$ among other competing machines from the restricted hypothesis space $\mathcal{H}$ (solution space containing potential learning machine candidates), that best approximates the probability distribution from 2.1

$$\hat{y}_i = f_{\mathcal{D}}(\mathbf{x}_i; \boldsymbol{\theta}). \tag{2.3}$$

This machine maps each data instance $\mathbf{x}_i$ to a prediction $\hat{y}_i$. Naturally, it is equipped with a set of model parameters $\boldsymbol{\theta}$, and will be adjusted during the learning process to minimize the empirical risk (approximates the expected risk), in order to come closer to the superior's true labels y_i

$$R_{emp}[f_{\mathcal{D}}] = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(y_i, f_{\mathcal{D}}(\mathbf{x}_i)). \tag{2.4}$$

The most common types of learning are:

- Supervised/Predictive Learning: Exactly the form we already described before, where the learning process is provided with a fully labeled data set. When the labels are of type classes $\mathbf{y} \in \{c_1, \ldots, c_k\}^N$, then it is a multiclass classification task. Otherwise, if $\mathbf{y} \in \mathbb{R}^N$, then it is a regression task.
- Unsupervised/Descriptive Learning: There is no supervisor (S), we have to discover the structure in the data analytically. Several algorithms exist, for example Cluster Recognition or Principal Component Analysis (PCA)
- Reinforcement Learning: We call a utility-based *agent* acting in an environment, which follows certain rules to make specific movements. The next move is based on a utility function, that computes the optimal policy, conditioned on the observed reward, to select the next best action [13].

2.2 Deep Neural Networks

Deep Learning is a subfield of Machine Learning, where we exclusively work with the engineering of neural networks with deep architectures. Neural networks were introduced in

1943, where McCulloch and Pitts [14] have proposed in their publication the first "neuron" model, which is a computational replication of the processes in biological neurons with propositional logic. This idea was further investigated, and algorithms for forward- and backward processing were developed in the 1980s. Due to limited computing power at that time, they had been not realizable for general use, and hence were mostly put on ice until 2007. Meanwhile, the Canadian Institute of Advanced Research (CIFAR) has continued research into neural networks, and has intended new architectures and learning approaches. In 2006, neural networks found the breakthrough with the conviction to build "deep" architectures, meaning, models with multiple stacked layers including many computing units called *neurons*, in order to extract high-level features from low-level features efficiently from large data sets. With the availability of faster processors and graphical processing units (GPUs), deep networks have become the de-facto standard for many domains [15].
We first start by explaining the Perceptron model, then go over to general "shallow" artificial neural networks. Finally, we end up with feed-forward networks, and show the significance of training deep networks.

2.2.1 Perceptron

A Perceptron [16], or a neuron model as shown in Figure 2.2, is a computing unit proposed by Rosenblatt [17], and is the simplest form of neural networks.

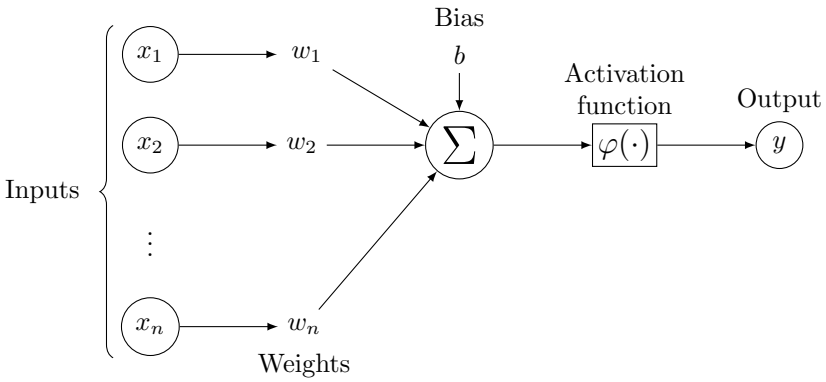


Figure 2.2: Mathematical model of the perceptron (based on from Haykin [16])

The input nodes receive input signals $\mathbf{x} = [x_1, \ldots, x_n]^\top$, and forwards them to a processing unit called "adder". This unit induces an affine transformation by linearly combining the signals from the input nodes, which are weighted by the sensitive terms $\mathbf{w} = [w_1, \ldots, w_n]^\top$ and adding a bias term b. The intermediate result yields a linear hyperplane in a n-dimensional space, and divides it in half. The resulting sum will be passed to a non-linear activation function $\varphi(\cdot)$

$$\varphi\left(\sum_{i=1}^{n} w_i x_i + b\right). \tag{2.5}$$

The activation function for perceptrons is a hard threshold function and derives an output

y, where

$$y = \begin{cases} +1 & \text{if } \varphi(\cdot) > 0 \\ -1 & \text{otherwise.} \end{cases} \tag{2.6}$$

Accordingly, the goal of this perceptron model is to classify the input signals, either class c_1 if $y = +1$ or c_2 if $y = -1$. To minimize the classification error, we have to search for the appropriate hyperplane in the hypothesis space. This can be done by adjusting the weights and bias term (summarized into one vector $\boldsymbol{\theta}$) using an error-correction learning algorithm, or simply Perceptron algorithm [16, 18], in order to optimize the following cost function $\mathcal{J}(\boldsymbol{\theta})$ over a set of misclassified data points $[x_1, \ldots, x_m]^\top$

$$\mathcal{J}(\boldsymbol{\theta}) = \sum_{i=1}^{m} \delta_{x_i} \theta_i x_i, \tag{2.7}$$

where δ_{x_i} signifies a falsely assigned class. The optimization algorithm is an iterative and gradient based approach, or better known as the *gradient-descent* method

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}}, \quad \text{where } \eta > 0. \tag{2.8}$$

The cost function is a continuous, non-convex function, and measures the magnitude of the classification error. In each iteration $t+1$, the parameter vector $\boldsymbol{\theta}^{(t)}$ from the previous state t gets updated by adding the negative gradient of the cost function. After a certain amount of iterations, the algorithm converges to a local minimum, where it tries to classify as many data points as possible. The parameter η is the so-called *learning rate*, which is crucial to control the tempo of the convergence, and should be adapted during the training to ensure, that the algorithm lands at a minimum. In general, one can differentiate between local and global minima, where the latter is the absolute low point, and is therefore preferred. However, this cannot be guaranteed, and we have to assume, that the algorithm converges towards a local minimum. The Perceptron Convergence Theorem, see [16] and [18], expounds that the minimum can be achieved in a finite number of iteration steps. Many sophisticated alternatives based on gradient-descent method exist, which are rather applied in larger architectures like the Multi-layer Perceptron (MLP) and Deep Neural Networks.

2.2.2 Feed-forward Neural Networks

A more general model is the Multi-layer Perceptron, or fully-connected neural network, which consists of an input layer, at least one intermediate, so-called hidden layer, with arbitrary number of neurons, and an output layer. This kind of architecture allows us to solve complex, non-linear separable problems. In this section, we go through the concept of learning fully-connected networks, that includes a *forward* and *backward* propagation. In the figure below, we can see an example of a deep network.

We call again a training set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{M}$. In the beginning of the forward propagation, each data instance $\mathbf{x}_i \in \mathbb{R}^D$ are fed to the input layer containing D neurons. Similar to the Perceptron Algorithm, all D features are individually weighted and passed to the first hidden layer $\mathbf{h}^{(1)} = \left[h_1^{(1)}, \ldots, h_m^{(1)}\right]^\top$. Each neuron in the hidden layer linearly combines the

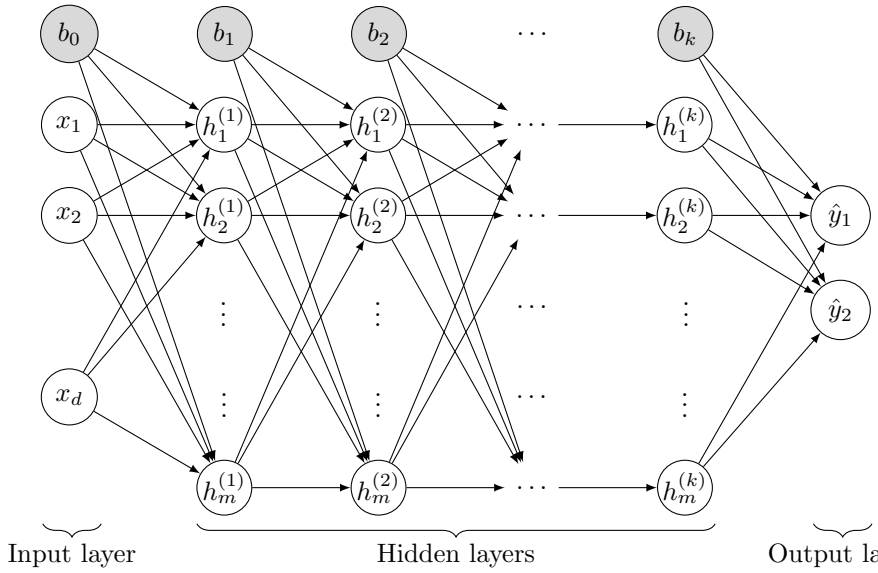


Figure 2.3: Neural network architecture with k stacked hidden layers of width m.

weighted input signals, adds the bias b_0 from the previous layer, and passes the intermediate result to a non-linear activation function. This process will be continued until the signals reaches the last hidden layer $\mathbf{h}^{(k)} = \left[h_1^{(k)}, \ldots, h_m^{(k)}\right]^\top$, and finally the output layer. The whole architecture is portrayed as a directed computational graph. Depending on the classification or regression task, the output layer might have different amount of neurons. For example in a multi-class classification task with three classes, we would have three output neurons, respectively. An important point to mention is, that each hidden layer can have an arbitrary number of neurons, independently of other layers, and that the network can have an arbitrary number of hidden layers. This facility is quintessential for building deep networks.

A neural network can be treated as a complex learning machine, but the objective remains the same. The training data comes with associated true labels, and the goal is to minimize the empirical risk between the network's output $\hat{y}_i$ and the true label y_i. The deviation between the true and estimated label can be measured with a suitable loss function $\mathcal{L}(y_i, \hat{y}_i)$. The process to minimize the empirical risk is called the *Backpropagation*, which we see in the next section. The following content is based on Goodfellow et al. [15] and Hart et al. [19].

2.2.2.1 Backpropagation

In the Backpropagation process, we follow the gradient-descent optimization technique (see Section 2.2.1). The following cost function $\mathcal{J}(\boldsymbol{\theta})$ evaluates the network's fitting capability. The main goal of this entire learning process is to find the best fitting machine $f_{\mathcal{D}} \approx f^*$ by minimizing the cost function

$$\mathcal{J}(\boldsymbol{\theta}) = \min_{\boldsymbol{\theta}} \sum_{i=1}^{N} \mathcal{L}(y_i, \hat{y}_i). \tag{2.9}$$

The network parameters $\boldsymbol{\theta}$ comprise the set of weight matrices $[\mathbf{W}_1, \ldots, \mathbf{W}_k]$ and the set of biases $[b_0, \ldots, b_k]$. Each weight matrix contains all sensitive weights between two layers.
In the beginning, the parameters are randomly initialized. Usually, weights have small random values initially, and biases have positive random values close to zero. The parameters are updated according to the Formula 2.8. The error-correction δ propagates back from the output layer to the first hidden layer. Since the values of the output layer results from a composition of multiple activations, we can use the *chain rule* to compute the partial derivative over $\mathcal{J}(\boldsymbol{\theta})$, with regard to the j-th parameter $\theta_{ij}^{(k)}$, with $z_i^{(k)} = \varphi(\cdot)$ denoting the resulting activation of the i-th neuron from the hidden layer k

$$\frac{\partial \mathcal{J}}{\partial \theta_{ij}^{(k)}} = \frac{\partial \mathcal{J}}{\partial z_i^{(k)}} \frac{\partial z_i^{(k)}}{\partial \theta_{ij}^{(k)}}. \tag{2.10}$$

Each hidden unit has a connection to every neuron in the next layer. Therefore, the error δ_i for the i-th hidden unit in the layer k can be computed by summing up of all the unit errors of the layer $k+1$, again by using the chain rule

$$\delta_i = \frac{\partial \mathcal{J}}{\partial z_i^{(k)}} = \sum_l \frac{\partial \mathcal{J}}{\partial z_l^{(k+1)}} \frac{\partial z_l^{(k+1)}}{\partial z_i^{(k)}}. \tag{2.11}$$

This rule is applied recursively over all neurons starting from the output layer. The forward and backward process is repeated until a desired minimum of the cost function is reached.
There exist several ways in passing the data through the network. The simplest method is to feed the whole data set at once in every epoch, also called the batch version. However, this slows down the convergence since the parameters are not updated very often. In contrary, the stochastic gradient descent method chooses randomly one data instance in every iteration. Significantly, it has a higher update frequency, and yields a higher chance to end up at a lower local minimum, or even by chance at the global minimum. A much more intuitive version is to divide the data set into small chunks, called mini-batches, where the algorithm randomly chooses a mini-batch in every iteration. This variant uses a decent amount of memory, and speeds up the convergence.
The gradient-descent method we explained so far, is the basic concept of optimization, and is mostly used for Perceptrons. In Deep Learning, we rather use one of the sophisticated optimization algorithms, see Goodfellow et al. [15] and Géron [20], that have been proposed over the years. Ideally, they often lead to faster optimization for large and densely connected networks. Popular optimizers are for example Stochastic Gradient Descent (SGD), AdaGrad, RMSProp, and Adam [15, 20].

2.2.2.2 Activation Functions

In each hidden neuron, the signals are passed through a non-linear activation function. This is essential, because it allows us to find, through the composition of all hidden neurons, the appropriate, non-linear machine $f_{\mathcal{D}}$. At the same time, are differentiable and hence suitable for Backpropagation. In the following illustration, we can see some widely used activation functions and their derivatives.

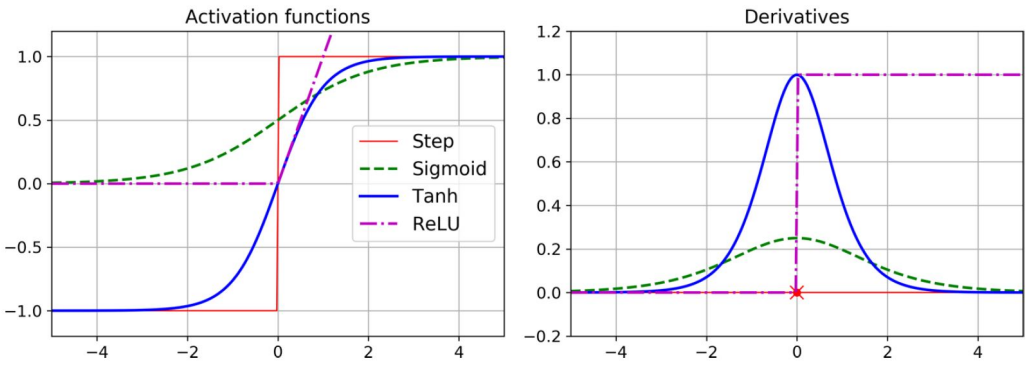


Figure 2.4: Common activation functions (left) and the corresponding derivatives (right) (from Géron [20])

The Step function is not differentiable, and thus not desired for Feed-forward Networks. Instead, we can choose one of the following activation functions [21].

Sigmoid The sigmoid is a continuous, S-shaped function, and has a probabilistic interpretation, because it maps the input z into the range of $[0, 1]$

$$\varphi(z) = \frac{1}{1 + e^{-z}}. \tag{2.12}$$

Hypobolic Tangent Similar to the sigmoid, it is also an S-Shaped function with the range $[-1, 1]$

$$\varphi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}. \tag{2.13}$$

ReLU The Rectified Linear Unit (ReLU) is the mostly used function in Deep Learning, which emulates sort of the way how biological neurons fire chemical signals

$$\varphi(z) = \max(0, z). \tag{2.14}$$

The derivative is either 0 or 1, and thus avoids the vanishing and exploding effects, which used to be the major issues with the sigmoid or tanh function. An extension to the ReLU function is the leaky ReLU, which has a small slope for negative input.

2.2.2.3 Loss Functions

To quantify the discrepancy or the empirical loss, respectively, between the network's estimation $\hat{y}_i$ and the desired true label y_i, we use one of the following loss functions [21].

Mean Squared Error The popular Mean Squared Error loss (MSE) is mainly used for regression tasks

$$\mathcal{L}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2. \tag{2.15}$$

Binary Cross Entropy The Binary Cross Entropy Loss (BCE), actively used in logistic regression models, outputs a probability for binary classification tasks. To do so, the value of the output neuron must be passed through a sigmoid function $\varphi(\cdot)$

$$\mathcal{L}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} y_i \log(\varphi(\hat{y}_i)) + (1 - y_i) \log(1 - \varphi(\hat{y}_i)). \tag{2.16}$$

For this task, the network's output layer must have only one neuron. If the sigmoid's output is above 0.5, then the prediction would be 1, otherwise 0.

2.2.2.4 Regularization

Indispensable phenomena in large networks are the so-called underfitting and overfitting [15]. Figure 2.5 shows a model fitting example of a regression task.

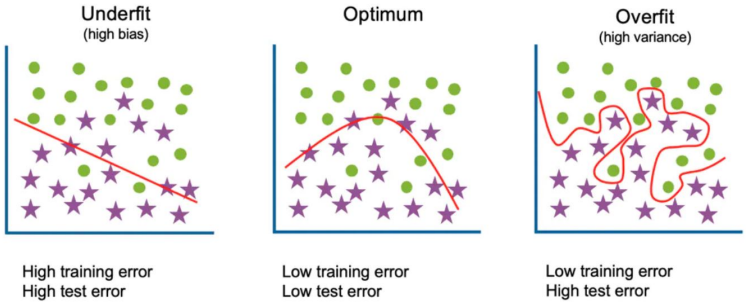


Figure 2.5: Illustration of simple classification task with underfitting (left) to overfitting (right). The figure in the middle has the ideal fitting. (from IBM [22])

In the left figure, the model underfits the training data, which is always the case in early learning stages. The right figure the model overfits the data, and poorly performs on test data. This happens, when the model has completely memorized the training data, and shrunken its ability to perform on unseen data. The model in the middle is a proper balance, that has a good accuracy on training data, and generalizes well on unseen data. The plot in Figure 2.6 shows a potential overfitting, which can be detected by the growing *generalization gap* between the trainings and the generalization error.

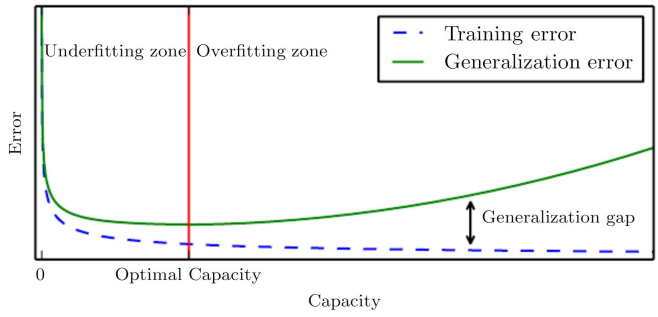


Figure 2.6: Model capacity curve (from Goodfellow et al. [15])

In order to do that, a separate validation set should be used to audit the model's generalization. The capacity of the model is considered as an informal heuristic to determine

the coverage of possible machines in the hypothesis space to a certain extent. The desired optimal capacity is the function $f_{\mathcal{D}} \approx f^*$ which we are primarily looking for.
These are two central challenges in Deep Learning, that can be prevented with *regularization* techniques during the learning process, which helps to shrink the hypothesis space. We can use one of the popular regularizations to discourage too complex and highly parametrized machines, e.g. weight regularization, dropout, early stopping or data augmentation. We take a closer look at weight regularization, from which our tree regularization is inspired.

Weight Regularization In this type of regularization, we add to the cost function an additional penalty term $\Psi(\boldsymbol{\theta})$ associated with its regularization parameter $\lambda \in [0, \infty)$, which controls the strength of the penalty

$$\hat{\mathcal{J}}(\boldsymbol{\theta}) = \min_{\boldsymbol{\theta}} \sum_{i=1}^{M} \mathcal{L}(y_i, \hat{y}_i) + \lambda \Psi(\boldsymbol{\theta}). \tag{2.17}$$

Ideally, the optimizer means to minimize both the training loss and the complexity of the model by reducing the magnitude of the parameters $\boldsymbol{\theta}$. Common baselines for the penalty term are the l_1- or l_2-norm. Before we continue, let us take a brief detour to the general theorem of vector p-norms [23].
A *norm* of a vector $\mathbf{x} \in V$ can be understood as the distance from the origin in a vector space V [24]. The l_p-norm is the generalization of the norm functions, and defines the distance to the defined contour line of degree p, as shown in Figure 2.7. For example, every l_1-norm (*Manhattan norm*) touches the purple diamond contour line and the l_2-norm (*Euclidean norm*) the blue circle contour line.

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}}, \quad 1 \leq p \leq \infty \tag{2.18}$$

$$\|\mathbf{x}\|_\infty = \max(|x_1|, \ldots, |x_n|) \tag{2.19}$$

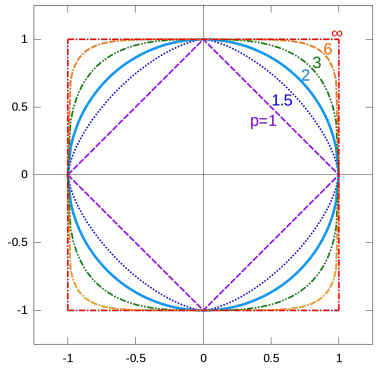


Figure 2.7: Illustration of unit level sets of different p-norms in the $\mathbb{R}^2$-space. [23]

Both l_1- and l_2-regularization reduce the values of dominating model parameters. The regularization term λ controls the regularization strength, and must be defined before training. The efficiency of regularization with norm penalties can be explained with sparse models.

For instance, in a linear regression task, having a small N and large D problem, meaning, the given a data set has more dimensions than the amount of data points. In this case, we can reduce the amount of features, and use only the relevant ones, but still maintain a high predictive power. A classical regression problem, that models the Gaussian distribution $\mathcal{N}(y|\boldsymbol{w}^\top\varphi(x), \sigma^2)$, can be solved by minimizing the squared difference between the target value y_i and the estimate $\hat{y}_i$, also called *residual sum of squares* (RSS), in order to find the maximum likelihood under the Gaussian error model (least squares fit). Adding a l_1-penalty to the RSS results in a *least absolute shrinkage and selection operator* regression (LASSO), and with l_2-penalty in a *Ridge regression* [11, 25]. In Figure 2.8, we can see the regions of the l_1- and l_2-norm. The blue contour lines indicate the least squares of the regression problem.

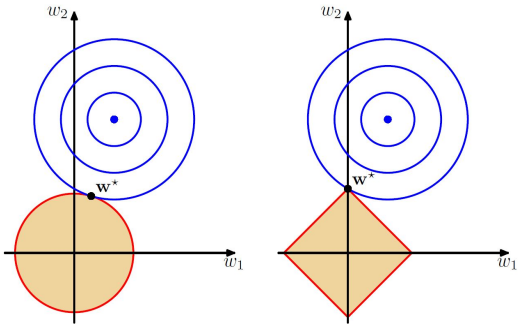


Figure 2.8: Illustration of $l_2$norm (left) and l_1-norm (right) regions in an two-dimensional space. The axis denotes the weights $\mathbf{w} = \{w_1, w_2\}$. Intersection with the least squares (blue lines) denotes the optimal fit. The higher λ is, the larger are the regions and would touch the least squares much earlier. (from Bishop [25])

The intersection between the norm and the least squares yields optimal weights $\mathbf{w}^*$. With the l_1-norm, we have the additional property due to its diamond shape, that both functions can intersect on an axis. This means, that certain weights can be 0, for example in the figure the weight $w_1 \leftarrow 0$, which leads to a strict selection of features. In the case of l_2, on the other hand, regularized weights become sparse.

2.2.2.5 Expressive Power of Deep Networks

The trend of building deep networks arose due to the theory of the *Universal Approximation* theorem. We have seen so far, that using non-linear approximation functions leads to fitting a non-linear learning machine in a high-dimensional space. Neural networks have the flexibility, to add an arbitrary number of neurons.. For this sake, a simple network with one shallow hidden layer can implement any continuous function (see Kolmogorov [26] and Arnol'd [27]). However, this theorem guarantees only the existence of such functions. A more essential alternative is to build a network with multiple layers rather than just one. The following theorem from the publication of Montúfar et al. [28] confirms the claim.

Theorem 1 A rectifier neural network with d input units and L hidden layers of width $m \geq d$ can compute functions that have $\Omega\left(\left(\frac{m}{d}\right)^{(L-1)d} m^d\right)$ linear regions.

A linear region is a subregion of a d-dimensional input space, in which a part of the learning machine $f_{\mathcal{D}}$ is linear. Therefore, it is also called as a piecewise linear function. According to the Theorem 1, the amount of linear regions grows exponentially in the number of hidden layers L and polynomially in the number of neurons m. Even if we would have the same amount mL of neurons in just one layer, we obtain a much larger number of regions with L stacked layers. This allows us, despite the overfitting problem, to build complex machines to better capture the data distribution.

2.3 Decision Trees

Decision Trees are another, much simpler type of building non-linear learning machines strictly based on a tree structure. Instead of using patterns, it takes a set of attributes or features as input, and returns a single output as a decision to categorize the data.

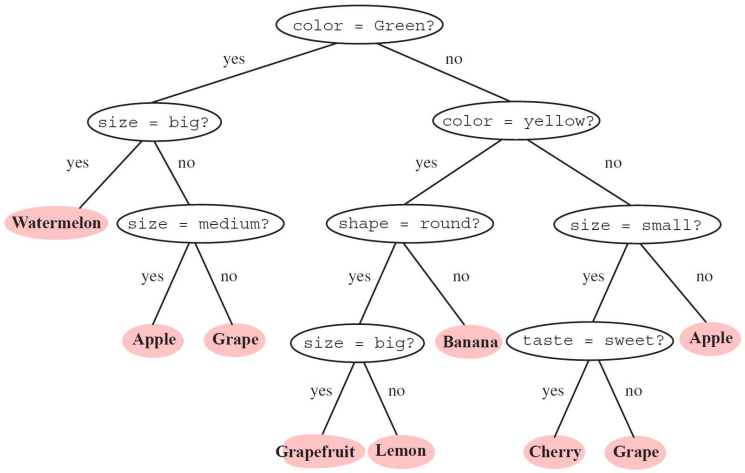


Figure 2.9: Decision tree of a fruit classification task (from Hart et al. [19])

In Figure 2.9, we can see an illustration of such a decision tree. The tree is usually directed and is composed of a root node on the top, several internal nodes organized on increasing levels, and leaf nodes positioned at the bottom. All nodes are connected via branches, where a sequence of tests happens to incrementally filter the category of an unknown sample. Looking at the example in the figure, we have the set of attributes [*color, size, shape, taste*]. In every decision node (root and internal nodes), we test whether the sample matches a certain attribute. If "yes", we follow the left branch, otherwise the right one, and continue with the next test. In the beginning, we start from the root node and follow the decision path, until we land in a leaf node. The corresponding label of this leaf is finally assigned as the category of the random sample. Visually, each decision can be seen as a decision line in the direction of the attributes. Overall, the decision boundary is axis-aligned, and splits the space into regions, see the example in Figure 2.10.
Decision trees became popular with their simple training, high predictive quality, and their interpretability. In the case of a binary decision tree, like in the example before, the decision

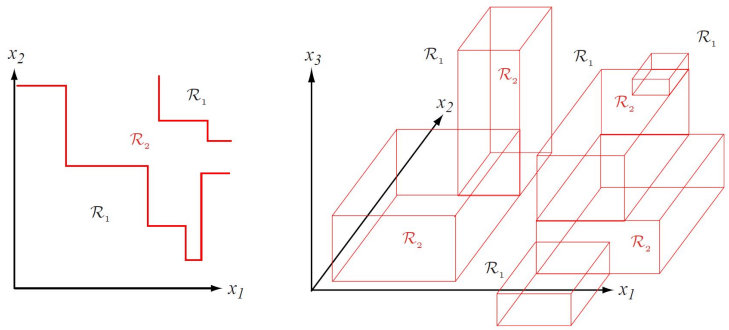


Figure 2.10: Decision regions of a binary classification task, (left) with two-dimensional example with attributes $\{x_1, x_2\}$, and (right) tree-dimensional example with attributes $\{x_1, x_2, x_3\}$. (from Hart et al. [19])

path is mutually distinct and exhaustive, meaning, only one conjunction of decisions can be achieved to find the appropriate category, which is completely sufficient according to the universal expressive power of binary trees [19].

2.3.1 CART

To build and train a decision tree, we first acquire a set of training data $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{M}$, and process them through the CART (Classification and Regression Trees) framework. For each node, we have to choose the best attribute to purely split the data. That means, the further subsets in lower levels should be more and more class-homogenous [18]. The algorithm to build the tree is a recursive greedy approach, where we principally use a splitting rule in each node for the purpose of choosing the best attribute a^*. The *impurity* or *entropy* (from Shannon's Information Theory) of a node t is a convenient measure for that [19]. Let $P(c_i|\mathcal{D}_t)$ be the portion of class c_i observations, $i = 1, \ldots, M$, in node t with the corresponding subset $\mathcal{D}_t$ (split along a chosen attribute from the previous decision in node $t-1$)

$$P(c_i|\mathcal{D}_t) = \frac{1}{|\mathcal{D}_t|} \sum_{d \in \mathcal{D}_t} I(d = c_i). \tag{2.20}$$

Otherwise, we take the respective percentages if the distribution is unknown. The impurity measure H of the node t is used as a splitting rule and is defined as

$$H(\mathcal{D}_t) = -\sum_{i=1}^{M} P(c_i|\mathcal{D}_t) \log_2(P(c_i|\mathcal{D}_t)). \tag{2.21}$$

We split the data according to the attribute with the lowest impurity. Minimizing the impurity is equivalent to maximizing the Information Gain IG [11]

$$IG(a) = H(\mathcal{D}) - \sum_{t} \frac{\mathcal{D}_t}{\mathcal{D}} H(\mathcal{D}_t). \tag{2.22}$$

A different and alternative approach of choosing the best attribute is using the *Gini index*, which computes the expected error rate in node t [19]

$$Gini(\mathcal{D}_t) = 1 - \sum_{t} P(\mathcal{D}_t)^2. \tag{2.23}$$

The splitting quality of both Entropy and Gini index does not differ much in practice. The one main difference is, that Entropy is computationally expensive because of the logarithm.

2.3.2 Minimal Cost-Complexity Pruning

For better interpretability, we prefer small decision trees, those that are consistent with the input data, and are less likely to fall into overfitting. *Pruning* is essential to exclude branches with less predictive power. Even though it loses precision on training data, it generalizes well on unseen data. Among the many available pre- and post-pruning methods, we only consider the Minimal cost-complexity pruning method provided by the sci-kit learn library[1], which we used in our implementation.
Given a decision tree T, the cost-complexity is measured by adding the total classification error $R(T)$ (impurity) with the number of terminal nodes $|\widetilde{T}|$ scaled by a complexity parameter α

$$R_\alpha(T) = R(T) + \alpha|\widetilde{T}|, \quad \text{where } \alpha \geq 0. \tag{2.24}$$

The goal is to find a subtree of T in order to minimize the cost-complexity $R_\alpha(T)$. The impurity of a single node t is generally greater than its branch T_t. If we can find the effective parameter α_{eff} of a node, where both its impurity and the impurity of its branch are equal, then the following relation can be satisfied

$$\alpha_{\mathit{eff}}(t) = \frac{R(t) - R(T_t)}{|T| - 1}. \tag{2.25}$$

In the implementation, we can set an upper bound value for the parameter `ccp_alpha`. If the algorithm reaches a node, whose α_{eff} is greater than `ccp_alpha`, then this branch will be pruned. This process will be repeated recursively over all nodes.

[1] https://scikit-learn.org/stable/modules/tree.html#minimal-cost-complexity-pruning

3
Model

So far we have the theoretical bases together, we now come to the actual part of this work. We first explain the innovation of Tree Regularization, and show conceptually its training procedure. In the end, we take a glance at some proposals in interpretable machine learning, and compare to what extent our method can be viewed as an interpretable model.

3.1 Tree Regularization

In Chaper 2, we got to know the architecture of feed-forward networks in detail. In terms of interpretability, it would be hard to understand exactly which patterns are transmitted from input data through the individual neurons. We follow the vision of Doshi-Velez and Kim [1], and specifically address the problem of how deep models can be well-approximated by compact models, that are comprehensible to humans, for example with decision trees.
The tree regularization method is a new optimization concept for deep networks, in which the model parameters $\boldsymbol{\theta}$ are constrained so far, that the resulting decision boundary resembles that of a decision tree, see Figure 2.10 in Section 2.3. We consider a supervised learning task with labeled training set $\mathcal{D} = \{\mathbf{x}_i, y_i\}_{n=1}^{M}$. The objective would be to train and optimize the model in order to minimize the cost function (see Formel 2.9). We introduce a novel tree regularization function $\Omega(\boldsymbol{\theta})$ for the model parameters $\boldsymbol{\theta}$ of a deep network. Instead of using a l_p-norm penalty, as we saw in Section 2.2.2.4, we create a function that reflects the complexity of a decision tree, and plug it as the regularization term, together with the regularization parameter λ, to the cost function $\mathcal{J}(\boldsymbol{\theta})$

$$\mathcal{J}(\boldsymbol{\theta}) = \min_{\boldsymbol{\theta}} \sum_{i=1}^{N} \mathcal{L}(y_i, \hat{y}_i) + \lambda\Omega(\boldsymbol{\theta}). \tag{3.1}$$

This function neither measures the time nor the space complexity. It roughly quantifies the size of decision trees, which is comparable to the distance measure of norm penalties. With this form of regularization, the model can be better generalized, and its predictions would be easily *simulatable*. Doshi-Velez and Kim [1] constitutes that decision trees are the right choice for measuring the simulatability. The methodology to approach this regularization involves two steps, that take place in every training iteration. In the first step, we compute the predictions from the current model state given the input data. Then, we feed the input data together with the predictions into a binary decision tree, and compute $\Omega(\boldsymbol{\theta})$.

The *average decision path length* is an ideal cost function for assessing the complexity and is practically traceable. It indicates how many decision nodes one has to traverse on average for one data instance. The corresponding algorithm is originally depicted from the publication of Doshi-Velez and Kim [1].

Algorithm 1 Average-Path-Length Cost Function

Require:
$f(\cdot;\boldsymbol{\theta})$: prediction function with parameters $\boldsymbol{\theta}$
$\mathcal{D}_{\text{ref}} = \{\mathbf{x}_i, y_i\}_{n=1}^{N}$: reference data set with N examples
function $\Omega(\boldsymbol{\theta})$
tree $\leftarrow$ TrainTree($\{\mathbf{x}_i, f(\mathbf{x}_i;\boldsymbol{\theta})\}$)
return $\frac{1}{N}\sum_i$ PathLength(tree, $\mathbf{x}_i$)
end function

The TrainTree subroutine is the CART algorithm (see Section 2.3.1), which implements the binary decision tree data structure. The PathLength function computes the length of the decision paths for each data point, and will be normalized in the end. For both subroutines, a separate designated data set $\mathcal{D}_{ref}$ is used, to make sure that the tree model is not consistent with the training set $\mathcal{D}$. We perform several CART runs over several random seeds, and take the mean of the resulting average path lengths, to avoid any bias. The `DecisionTree` module from the sci-kit learn library[2] provides the necessary functionalities for the implementation.
We go through the training procedure more in detail, and roughly relate to the implementation paradigms.

3.2 Training

The desired deep model is trained with the gradient-descent method, as explained in Chapter 2. The focus lies mainly on creating the tree regularization cost function, and the appropriate combination of the hyper-parameters. In Figure 3.1, we show the pipeline of the training process.

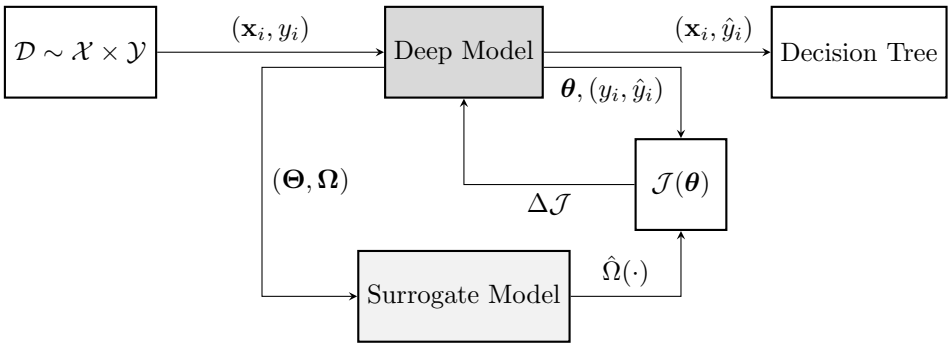


Figure 3.1: Overview of the neural network training pipeline with Tree Regularization

In the beginning of the forward propagation, we gather a training set $\mathcal{D}$, and feed a data

[2] https://scikit-learn.org/stable/modules/tree.html

instance $\mathbf{x}_i \in \mathbb{R}^D$ to the deep model. The model infers an estimate $\hat{y}$, and measures the error to the true target y_i via the cost function $\mathcal{J}(\boldsymbol{\theta})$. In the back propagation, we update the model parameters $\boldsymbol{\theta}$ with a optimization algorithm (see Section 2.2.2.1). For regularization, we need to compute the average decision path length (APL) of the associating binary decision tree given the training examples $\{\mathbf{x}_i\}_{i=1}^M$ and the model estimates $\{\hat{y}_i\}_{i=1}^M$. However, it is not differentiable with respect to the $\boldsymbol{\theta}$, and hence cannot be embedded into the cost function $\mathcal{J}(\boldsymbol{\theta})$. For this reason, we need a machine $supp(\boldsymbol{\theta}) \rightarrow \mathbb{R}_+$ to assign each model parameters from previous training iterations to APL estimates. Therefore, we build a *surrogate model* $\hat{\Omega}(\cdot)$ in addition to the deep model. For example a simple Multi-layer Perceptron with one layer would be sufficient to mimic the APL of the tree. This model is differentiable, and can be incorporated into the chain rule for the gradient-descent method

$$\boldsymbol{\theta}^* = \min_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) \tag{3.2}$$

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta \frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}} \tag{3.3}$$

$$= \boldsymbol{\theta}^{(t)} - \eta \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} + \lambda \frac{\partial \Omega}{\partial \boldsymbol{\theta}} \right). \tag{3.4}$$

To train the surrogate model, we have to gather data $(\Theta, \boldsymbol{\Omega})$ during the training of the target deep model. The input features $\Theta = \{\boldsymbol{\theta}_j\}_{j=1}^J$ contain the model parameters from the previous J training iterations right from the beginning of the training, and the labels $\boldsymbol{\Omega} = \{\Omega(\boldsymbol{\theta}_j)\}_{j=1}^J$ are computed by the Algorithm 1. After collecting the data, the surrogate model is trained with forward and backward processing techniques by optimizing the following cost function $\widetilde{\mathcal{J}}(\boldsymbol{\xi})$ with respect to the parameters $\boldsymbol{\xi}$ of the surrogate model

$$\widetilde{\mathcal{J}}(\boldsymbol{\xi}) = \min_{\boldsymbol{\xi}} \sum_{j=1}^{J} \mathcal{L}(\Omega(\boldsymbol{\theta}_j), \hat{\Omega}(\boldsymbol{\theta}_j; \boldsymbol{\xi})) + \gamma \Psi(\boldsymbol{\xi}). \tag{3.5}$$

Since we aggregate the data set after every training iteration with new data, the size J grows as well. We collect data throughout the whole training from hundreds of iterations, hence the model might be prone to overfitting. Thus, a small weight regularization $\Psi(\boldsymbol{\xi})$ with the regularization parameter $\gamma \in [0, \infty)$ can help, for example with the l_2-norm penalty.

The more often we train the surrogate model, the better it can track the APL of the underlying tree. Therefore, we find that it should be trained after every training epoch of the deep model. During our experiments, we found it reasonable to apply the regularization only after the deep model has fitted well on the training data. Hence, we separate the entire training process of the deep model into a *warm-up* phase, in which no regularization takes place, then the *regularization* phase until the end of the training, yet the surrogate model will be trained right from the beginning.

In early iterations, the amount of data $(\Theta, \boldsymbol{\Omega})$ are not sufficient for the surrogate model. For that reason, we strongly recommend enlarging the data set. We propose, to fill the data set from a few random restarts of the deep model at the very beginning. Then, during the training, provide additional synthetic data $(\widetilde{\Theta}, \widetilde{\boldsymbol{\Omega}})$, which can be obtained via data augmentation, and feed them together with $(\Theta, \boldsymbol{\Omega})$ to the surrogate model. The augmentation is one of the optimization keys, which was proposed by Wu et al. [29]. This publication is an extension of the tree regularization method, more details in future work.

3.2.1 Data Augmentation

In this work, we used the Dirichlet distribution for augmentation, which is a multivariate continuous distribution of the order $K >= 2$, and is the generalization of the Beta distribution [11]. It is known to be the conjugate prior (same functional form) of the Multinoulli likelihood. We are given a data set $\mathbf{x}$, that comes from a probability simplex S_k

$$S_K = \{\mathbf{x} : 0 \leq x_k \leq 1, \sum_{k=1}^{K} x_k = 1\}. \tag{3.6}$$

The density of a $K-$dimensional Dirichlet distribution, parameterized by $\boldsymbol{\alpha} \in \mathbb{R}_+^K$, is given as

$$\mathrm{Dir}(\mathbf{x}|\boldsymbol{\alpha}) \triangleq \frac{1}{\mathrm{B}(\boldsymbol{\alpha})} \prod_{k=1}^{K} x_k^{\alpha_k - 1}, \tag{3.7}$$

where $\mathrm{B}(\boldsymbol{\alpha})$ is the Beta function, and $\Gamma(\cdot)$ is the Gamma function (see Murphy [11])

$$\mathrm{B}(\boldsymbol{\alpha}) \triangleq \frac{\prod_{k=1}^{L} \Gamma(\alpha_k)}{\Gamma(\alpha_0)}, \quad \alpha_0 \triangleq \sum_{k=1}^{K} \alpha_k. \tag{3.8}$$

Figure 3.2 shows two simplex examples of a Dirichlet distribution of $K = 3$, which can be illustrated as triangular surfaces.

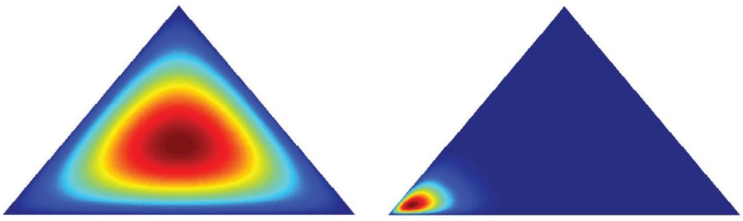


Figure 3.2: The density surface (left) with $\boldsymbol{\alpha} = [2, 2, 2]$ has a peak in the center. With parameters $\boldsymbol{\alpha} = [20, 2, 2]$, the peak is on the left corner (right) (from Murphy [11]).

For the augmentation, we have the following setting. Given the model parameters $\boldsymbol{\Omega} \in \mathbb{R}^{J \times M}$, we draw L new samples $\mathbf{D} \in \mathbb{R}^{L \times J}$ from the Dirichlet distribution $\mathrm{Dir}(\boldsymbol{\Omega}|\boldsymbol{\alpha})$ with $\boldsymbol{\alpha} \in \mathbb{R}^J$, e.g. using the `dirichlet` function from the numpy.random module[3]. To generate new synthetic parameters, we use the samples $\mathbf{D}$ as weights, and multiply them with the model parameters $\hat{\boldsymbol{\Omega}} = \mathbf{D} \cdot \boldsymbol{\Omega}$, which induce a weighted convex combination.

3.2.2 Regularization Strength

The regularization parameter λ controls the influence of the regularization term. We find taking a constant parameter value yields an abrasive effect towards the model's fitting. Hence, an increasing parameter value over the regularization phase would be more eligible. We take the inspiration from Simulated Annealing by using an Exponential Additive Cooling function [30], and calculate the parameter λ in every epoch. This function is commonly used

[3] https://numpy.org/doc/stable/reference/random/generated/numpy.random.dirichlet.html

in Physics to simulate a finite sequence of decreasing temperature values, but for our purpose it is convenient. In the beginning, we define an initial lambda value λ_{init}, a target lambda λ_{target}, and the amount of n epochs as cycles for the regularization phase. In each cycle k, the λ value will be computed as follows

$$\lambda \leftarrow \lambda_{\text{target}} + (\lambda_{\text{init}} - \lambda_{\text{target}}) \left(\frac{1}{1 + \exp\left(\frac{\alpha \ln |\lambda_{\text{init}} - \lambda_{\text{target}}|}{n} \left(k - \frac{n}{2} \right) \right)} \right) \tag{3.9}$$

$$\alpha \in \begin{cases} (0, \infty) & \text{if } \lambda_{\text{target}} > 1 \\ (-\infty, 0) & \text{otherwise .} \end{cases} \tag{3.10}$$

The parameter α can be manually adjusted. The higher the α value is, the more the function looks like a S-shape. The smaller the range of the λ values is, the larger the α should be chosen. Our proposal is to use such a S-shaped function, such that the regularization can be started smoothly with a low strength, and then should be cooled down in the end. In Figure 3.3, we can see some examples with $\lambda_{\text{init}} = 0.1$, $\lambda_{\text{target}} = 2$, and the parameters $\alpha = [10, 15, 25, 35, 45]$ over 100 cycles. The blue or green curve are suitable candidates for the regularization.

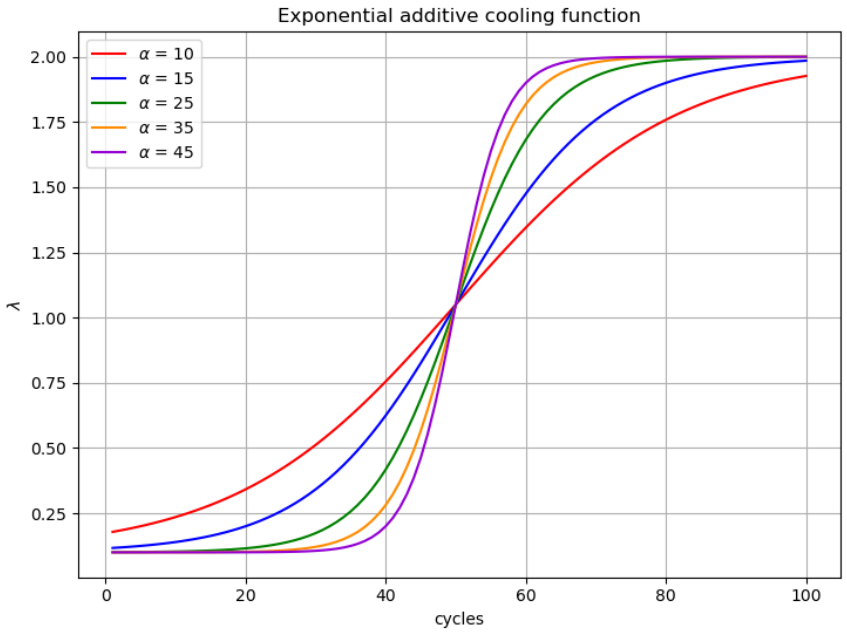


Figure 3.3: Examples of λ curves with $\alpha = [10, 15, 25, 35, 45]$

3.3 Related Work

We briefly describe here some existing proposal in interpretable machine learning. Our approach in this thesis is to express the deep model's decision-making via a decision tree, which comes under the category of intrinsic interpretation. One of the first attempts of extracting a tree-based representation from a black box model is the TrePan algorithm proposed by Craven and Shavlik [31]. One can extract a decision tree by answering queries during the training, for example asking for the class estimate, or to fulfill the required constraints on the features. However, even if this algorithm has achieved success, there is a lack of interplay between the black box model and the decision tree. On the other hand, the tree regularization method intends to constrain the deep model so far, that we would automatically get a smaller and more understandable tree. The following proposals are post-hoc interpretable tools. These are less related to tree regularization, but are still relevant to see the variety of interpretable models.

Selvaraju et al. [32] have shown in their publication a visual explanation concept based on a class-discriminative visualization technique called Grad-CAM for Convolutional Neural Networks (CNNs). It draws a heat map on the input image by processing the gradient of a target label through the last convolutional layer (see publication for more detail), which highlights important features in certain positions. This method is widely applicable to different visual tasks like image classification. Compared to our method, it is not intuitive for tabular data sets, and vice versa, image classification tasks are not suitable for building decision trees with tree regularization.

A different visualization approach by Yosinski et al. [33] attains to provide in-depth understanding by illustrating a grid of active neurons from a pre-trained network. Additionally, each neuron can be further investigated to get detailed information about the patterns in data through new regularization techniques. Such a tool allows us to understand, how the final answers are computed by a black-box model.

The LIME method, proposed by Ribeiro et al. [5], is a model-agnostic interpretation system. Given a complex, highly non-linear model, the LIME framework goes on a local level in the data landscape, where a single data point is chosen to be explained. It fits a linear sparse model at this point by sampling new data around the point, and computing the distances to them, which are then used as weights. This simple model finally explains the original model's prediction. In summary, the LIME framework shows that one can reasonably explain the black-box model with sparse linear models.

4

Demonstration and Results

In this chapter we show our experiments with the tree regularization method and discuss the results. We exclusively work with synthetic data sets, which we explain in the next section.

4.1 Data Sets

We present two labeled data sets defined in a two-dimensional space. This allows us to plot the decision boundaries of the deep model and of the decision trees. We separate the data set into training (70%), validation (15%), and test set (15%).

4.1.1 Parabola Data Set

This data set specification is adopted from the work of Doshi-Velez and Kim [1]. We uniformly draw samples $\{\mathbf{x}_i\}_{i=1}^{N}$ from a two-dimensional space $[0, 1.5] \times [0, 1.5]$. Using the following parabola decision function $g(\cdot)$, we separate the samples into two classes $\{c_1 = 0, c_2 = 1\}$. Each data point contains two features $\mathbf{x}_i = [x_1, x_2]$, where the first feature is fed to the function, and the second feature is used to evaluate, whether it is above or below the function

$$g(x_1) = 5 \cdot (x_1 - 0.75)^2 + 0.4 \tag{4.1}$$

$$y_i = \begin{cases} 1 & \text{if } x_2 \geq g(x_1) \\ 0 & \text{otherwise .} \end{cases} \tag{4.2}$$

All samples that are above the decision line are assigned to class c_2, otherwise to class c_1. To reduce overfitting, we add some noise to the data. We do this by shifting the decision function up and down by 0.2 units and randomly assign to all samples from that margin the class labels using the Bernoulli distribution [11] with probability $p = 0.5$.

$$\text{Ber}(y_i|p) = p^{I(y_i=1)}(1-p)^{I(y_i=0)}, \tag{4.3}$$

which is the same as tossing a fair coin.

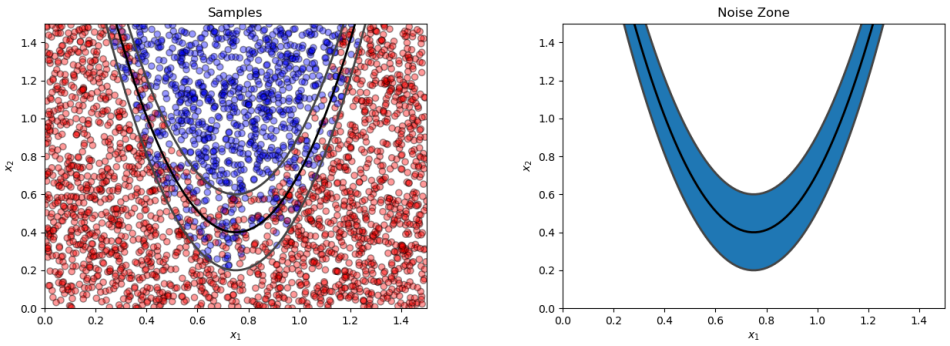


Figure 4.1: Illustration of random samples (left) with the black parabola decision line. Blue data points belong to class c_2 and red data points to class c_1. The blue margin (right), specified by the gray lines, is the zone where the noise is added.

4.1.2 Cosine Data Set

The cosine data set is defined in the space $[-6, 6] \times [-2, 2]$, and classifies the samples with the cosine decision function

$$g(x_1) = \cos(x_1). \tag{4.4}$$

Similar to the parabola data set, we shift the decision line up and down, here by 0.4 units, and add the noise in that blue margin area.

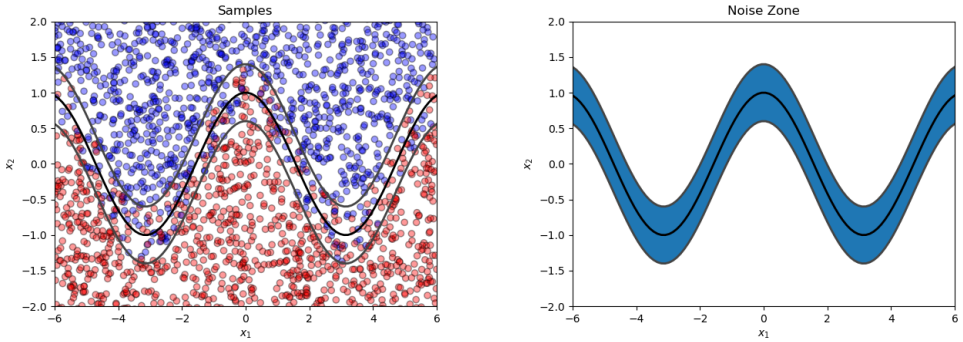


Figure 4.2: Illustration of random samples (left) with the black cosine decision line and the noise margin (right).

4.2 Pruning

To draw the decision trees given the sample set and the model's estimates $\{\mathbf{x_i}, \hat{y}_i\}_{i=1}^{N}$, we apply the minimal cost-complexity pruning (see Section 2.3.2) to reduce the tree size. In Figures 4.3 and 4.4, we can see an example of a non-pruned and pruned tree, that are fed by a parabola data set with 3000 samples.

To find the appropriate complexity parameter α^*, we use the k-fold cross validation method with 5 folds with the `cross_val_score` function provided by the sklearn.model_selection module. We determine a split of 70% training data and 30% test data. The cross-validation

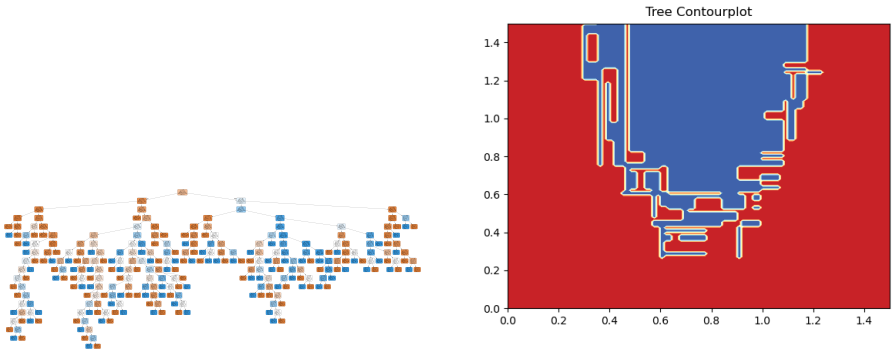


Figure 4.3: Decision tree (left) without pruning and the corresponding contour plot (right) in a two-dimensional space.

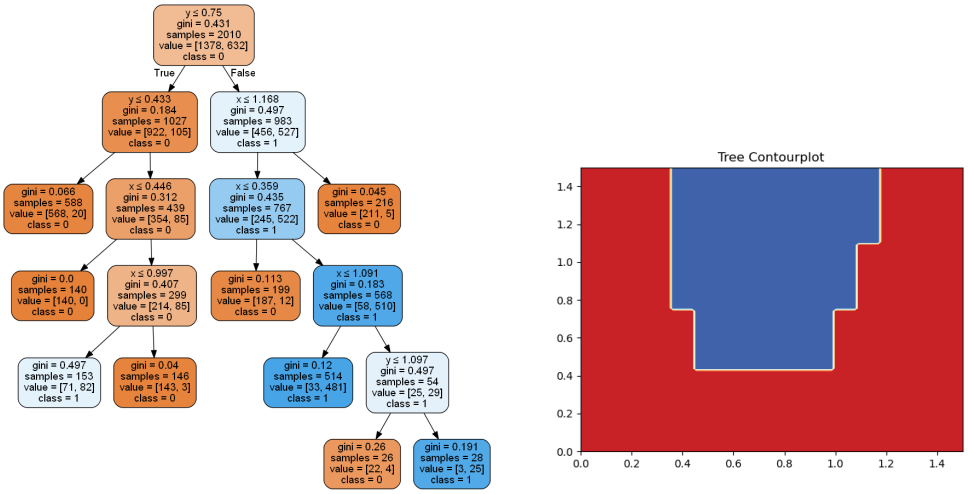


Figure 4.4: Decision tree (left) with pruning and the corresponding contour plot (right).

runs over 5 iterations over the whole data set, and evaluates in each iteration a different test set. This form of searching for the best tree in the hypothesis space is more confident and stable. In the next figure, we show the rotation scheme over k iterations.

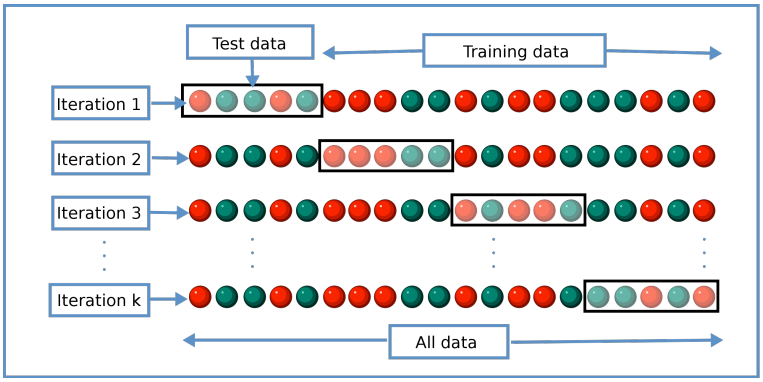


Figure 4.5: Overview of a general k-fold cross validation rotation scheme (from [34]).

We give to the input parameter `scoring` the "neg_mean_squared_error" value, which denotes the Mean Squared Error Loss for each fold. In the end, we save the final score by taking the average over all scores.
To compute the possible pruning paths of a tree, we use from the `DecisionTreeClassifier` function the subfunction `cost_complexity_pruning_path`, which provides an array of effective complexity parameters $\boldsymbol{\alpha}^*$. We iterate through all $\boldsymbol{\alpha}^*$, execute for each α_i^* a cross validation and store its score. Finally, we apply the "one-standard-error" rule, and choose the most parsimonious tree model with the highest α^* within the standard deviation of the best model with the minimal score [35].

4.3 Training Setting

The tree regularization method is implemented in Python. We used PyTorch[4] for building neural networks, and scikit-learn [36] for decision trees and to implement the training pipeline. The models were trained at the sciCORE scientific computing core facility at University of Basel. Calculations were performed mainly on NVidia A100 tensor-core and NVidia Titan X GPUs. In the following, we give an overview of the training specifications, which look for both data sets quite similar. The code base can be found in our Github repository[5].

Deep Model: Training Settings

	Parabola Problem	Cosine Problem
Data Set Size	20'000	35'000
Loss Function	Binary Cross Entropy	Binary Cross Entropy
Optimizer	Adam	Adam
Learningrate	1e−4	5e−4
Batch Size	1024	1024
Epochs warm-up	300	300
Epochs Regularization	700	700

Surrogate Model: Training Settings

	Parabola Problem	Cosine Problem
Loss Function	Mean Squared Error	Mean Squared Error
Optimizer	Adam	Adam
Learningrate	1e−3	1e−3
Regularization	l_2	l_2
Regularization Strength	1e−5	1e−5
Batch Size	256	256
Epochs	5	5

[4] https://pytorch.org/
[5] https://github.com/GowthamanG/tree-regularisation

The deep model is built of 3 hidden layers. The first two hidden layers each have 100 neurons, and the third layer has 10 neurons. For the activation, we used the tanh function. The surrogate model is a simple network with one hidden layer that contains 25 neurons, and uses the ReLU function as activation. To be on the safe side, we found it intuitive to define both the deep and surrogate model as one complete model in Python. We first defined both models as individual classes, and instantiate the surrogate model in the deep model class. This type of implementation ensures that the surrogate model is included as a regularization in the gradient-descent process.
We defined 1000 epochs in total, such that the model has enough time to fit and to regularize properly. The surrogate model is trained for 5 epochs after every training epoch of the deep model from the beginning on. In the regularization phase, we start with a small $\lambda_{\text{init}} = 1e{-}3$ and try possible targets $\lambda_{\text{target}} \in \{0.5, 1, 2, 3, 4, 5\}$. The data set for the surrogate model $(\Theta, \mathbf{\Omega})$ is aggregated after every optimization step, and fed to the surrogate model when it gets trained. When calculating the APLs, we deliberately do not use post-pruning method, because we find it too intense and the depth of the underlying tree gets reduced. Hence, the APLs would not be expressive enough for regularization. A potential pre-pruning method, for example the *minimal samples leaf* parameter, can be instead used as pre-pruning to get the APLs from not far too complex decision trees. In our experiments, we set the parameter `min_samples_leaf` to 5. To enlarge the data set $(\Theta, \mathbf{\Omega})$, we first do 25 random restarts, and add new synthetic data sets $(\widetilde{\Theta}, \widetilde{\mathbf{\Omega}})$ of size 500 after every epoch. We compute the synthetic APLs $\widetilde{\mathbf{\Omega}}$ with an auxiliary surrogate model to prevent the current model from being overwritten.
The Binary Cross Entropy Loss function evaluates the deep model's predictive error. The `BCEWithLogitsLoss` function from the torch.nn module already includes the Sigmoid layer, thus we do not have to add it manually in our network definition. For the surrogate model, we use the Mean Squared Error Loss function. We additionally modify the loss output, which allows us to see the fraction of the data variance that the surrogate model has learned (known as *explained variation* from statistics). The loss value $\mathcal{L}(y_i, \hat{y}_i)$ is normalized by the sample variance s^2, which is the averaged squared deviation from the mean of the targets, plus a small bias of 0.01. The sample variance is an unbiased estimator of the variance $\sigma^2 = \text{Var}[\mathbf{y}]$. This modification is not intended to be used in optimization steps.

$$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (y_i - \bar{\mathbf{y}})^2 \tag{4.5}$$

$$\widetilde{\mathcal{L}}(y_i, \hat{y}_i) = \frac{\mathcal{L}(y_i, \hat{y}_i)}{s^2 + 0.01} \tag{4.6}$$

4.4 Results

We define the benchmark for the evaluation as follows. The accuracy measures the predictive power of the deep network. We compute the accuracy score in every epoch using the validation set, which measures the percentage of how far the model estimate $\hat{y}_i$ matches the true target label y_i. We plot the training loss of both deep and surrogate model. For

the deep model, we chart both the training and validation error to track the generalization gap. Lastly, the most important assessment is the surrogate model's predictive power, which estimates the APLs $\hat{\Omega}(\cdot)$ of the underlying trees, and plot them together with the true APL $\Omega(\cdot)$ course.

4.4.1 Parabola Problem

During the training, after every 10^{th} epoch, we output a snapshot of the deep models decision boundary as contour plots, the intermediate network's approximation as decision trees with post-pruning, and its contour plots. In the following, we picked some snapshots. These plots are from the experiment with regularization parameters $\lambda_{\text{init}} = 1\text{e}{-3}$ and $\lambda_{\text{target}} = 1$.

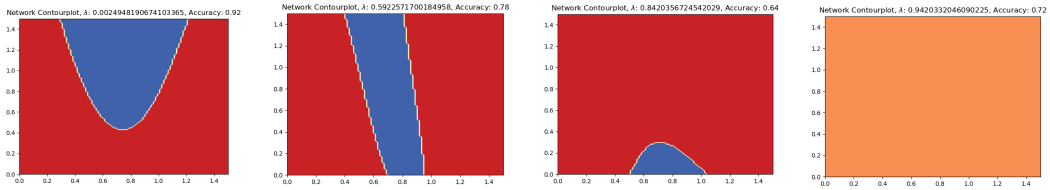


Figure 4.6: Deep model contour plots from epochs 300, 670, 740 and 800. After fitting, the network breaks down in epoch 740 with higher regularization.

In epoch 300, after the warm-up phase, the model has fits the training data as it should be. When regularization begins and the strength increases, the decision boundary starts changing and gets constrained. In epoch 670, we have a situation, where the boundary is defined as two straight, slightly sloping lines. It is hard to deduce, whether the boundary goes in the direction of axis-aligned decision boundaries, as we would expect from decision trees, because we do not have similar indications in no other snapshots (see Appendix A). The corresponding decision trees of these snapshots are shown below.

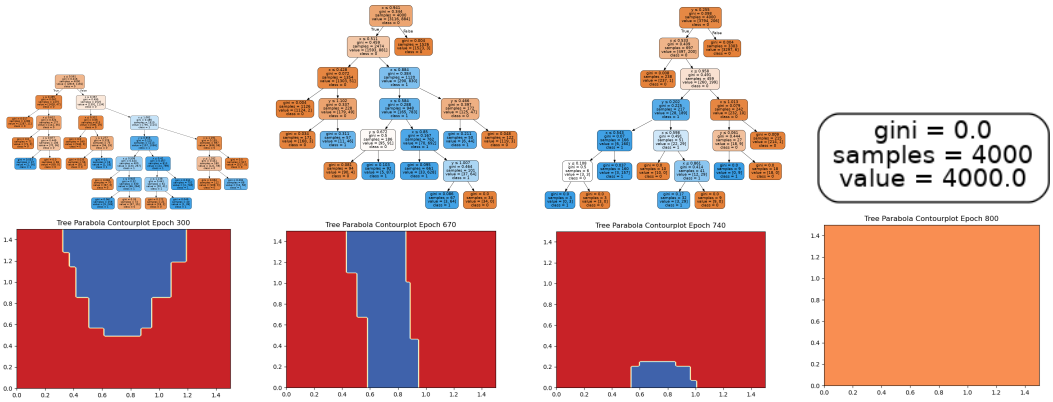


Figure 4.7: Intermediate decision trees (above) and the associated contour plots (below) from epochs 300, 670, 740 and 800.

One issue we have in this experiment is that the decision boundary breaks down and loses its fitting, when the regularization gets large. In epoch 740, we can clearly see, that the fitting gets smaller. If we look in the following plot, the APL curve decreases accordingly. In the warm-up phase (left of the red line), the surrogate model slowly learns the APLs and

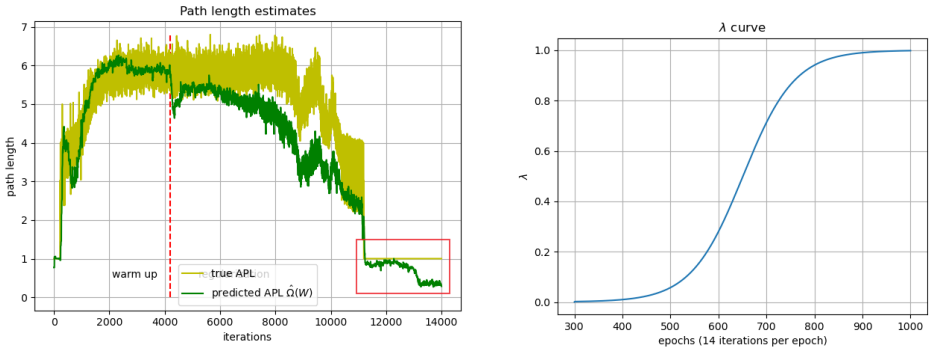


Figure 4.8: True APL and APL estimates (left) and the λ curve (right) of the experiment with $\lambda = 5$.

remains quite constant around the 2000^{th} iteration, or gradient step, respectively, where the deep model gets fitted. In the regularization phase (on right hand side from the red line), the APLs get reduced when λ increases. In the end, when λ is high, the APL has the value 1, meaning the tree has only one leaf. This is reflected in the plots (Figures 4.11 and 4.7), when the boundary gets broken. The training course can also be seen in the loss plot.

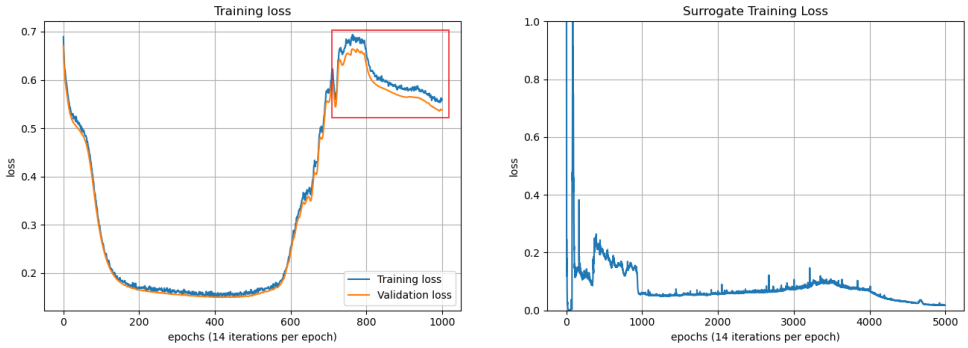


Figure 4.9: Training loss of the deep model (left) and training loss of the surrogate model (right). The loss of the surrogate model decreases quickly and stays low, which is a good sign. The training loss and validation loss of the deep model has looks similar, which shows that the model generalizes well.

After epoch 600, the loss starts growing due to the increasing regularization strength. The point, where the APL becomes 1 and no reasonable trees are produced, the loss reaches the highest loss value around 0.7 in epoch 800. We have marked the last few hundred epochs with a red square in the plots (Figures 4.8, 4.13 and 4.10), where the network loses its predictive power.

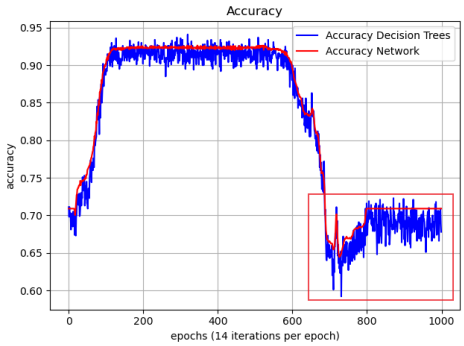


Figure 4.10: Prediction accuracy of the deep network and the underlying decision tree. The decision tree has mostly a clear accuracy measure due to the post-pruning. The network has much more variations. The accuracy decreases in the epoch 600, when the λ gets a steeper slope.

The accuracies of both deep network and decision tree are similar. After epoch 600, the accuracy decreases, when the loss increases. When the deep model does not infer anything reasonable, the accuracy is around 0.7.

We now show briefly a second experiment with $\lambda_{\text{target}} = 5$, that yield unfavorable results.

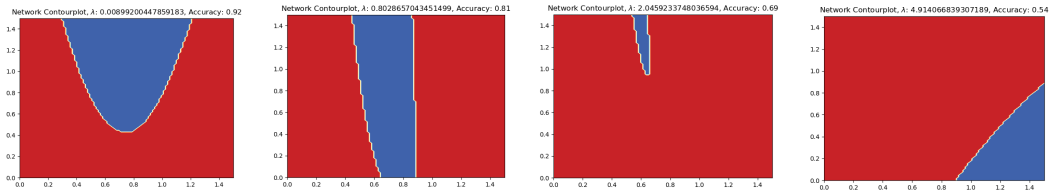


Figure 4.11: Deep model contour plots from epochs 300, 560, 630 and 870. With $\lambda_{\text{init}} = 1\text{e}{-3}$ and $\lambda_{\text{target}} = 5$, the regularization gets even stronger, and the model gets quickly constrained drastically.

In the regularization phase, the model has again the decision boundary as two lines (left line is more straight), but this time earlier in epoch 560. However, the regularization seems to be drastic here and the deep model is unable to create any meaningful decision boundaries.

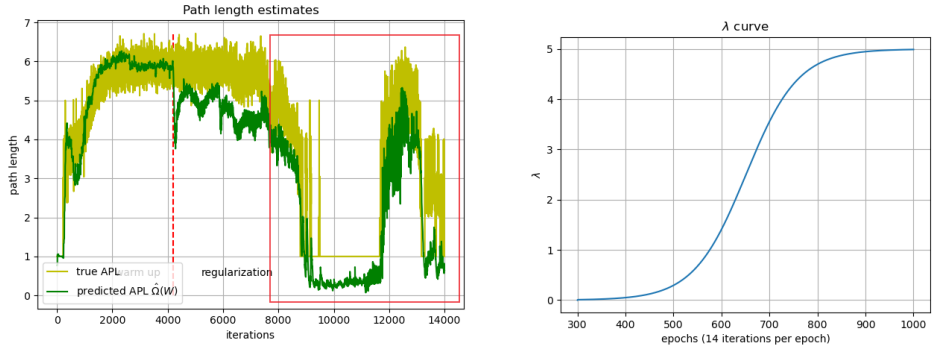


Figure 4.12: True APL and APL estimates (left) and the λ curve (right).

The regularization occurs earlier than in the previous experiment, and constraints the model much stronger. Towards the end, the optimizer tries to make the network functional again around the 12'000$^{\text{th}}$ iteration, but without success.

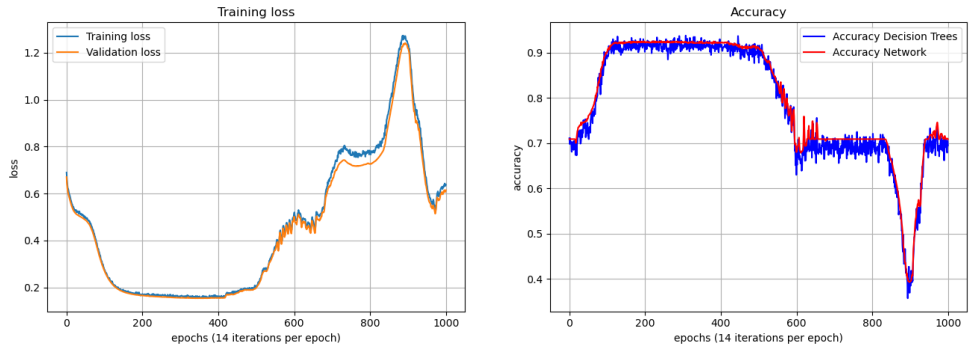


Figure 4.13: Training loss of the deep model (left) and its accuracy (right). The model clearly gets over constrained.

The loss curve, compared to the loss from the first experiment, has a much faster increase due to the larger lambda, and reaches above 1.2 in epoch 900, where the network collapses. The accuracy behaves in the opposite way. It decreases and has poor accuracy with higher lambdas. Overall, this experiment shows how the model gets regularized with large regularization strengths. We assume that λ_{target} should not be chosen too big.

4.4.2 Cosine Problem

For the cosine problem we conduct an experiment with the regularization parameters $\lambda_{\text{init}} = 1\text{e}{-3}$ and $\lambda_{\text{target}} = 1$.

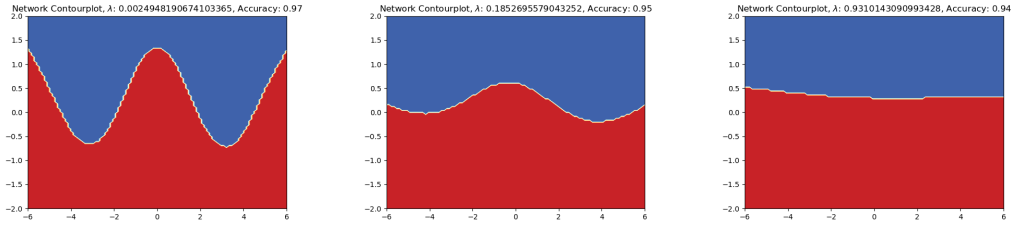


Figure 4.14: Snapshot deep model contour plot from epoch 300 (left), epoch 570 (middle) and epoch 790 (right)

The cosine problem has a larger space and the decision boundary is much more curvy. The decision boundary starts changing in epoch 570, and becomes almost a straight line parallel to the axis in epoch 790. We intentionally created this data set to see, how tree regularization affects a decision function with multiple curves.

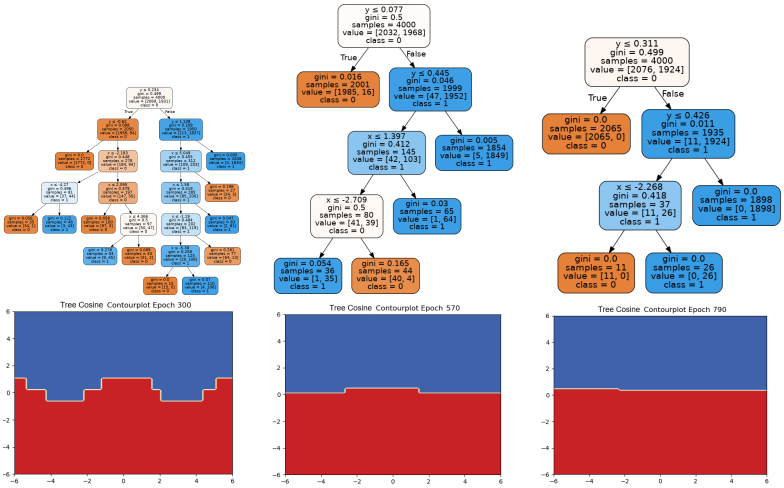


Figure 4.15: Intermediate decision trees (above) and the associated contour plots (below) from epochs 300, 570 and 790.

The resulting intermediate decision tree boundaries have great similarities with the boundaries of the deep network.

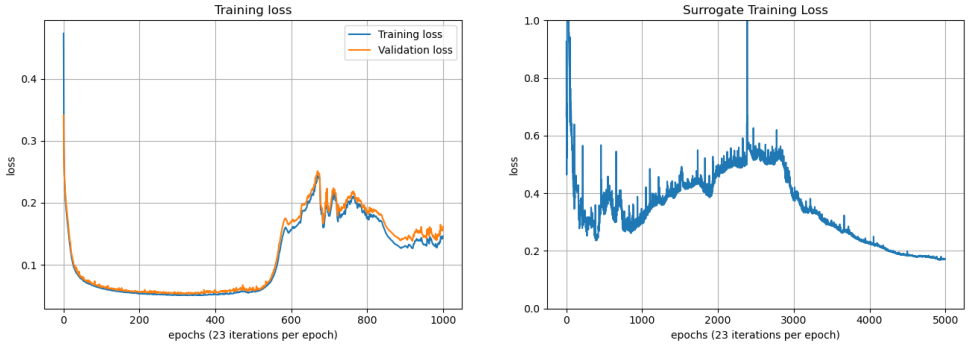


Figure 4.16: Training loss of the deep model (left) and training loss of the surrogate model (right). The loss of the surrogate model is a bit higher than the one from the parabola problem.

The loss of the deep model behaves similar, when regularization takes place. Compared to the parabola problem, the loss does not increase as much. Surprisingly, after epoch 800, the loss decreases again after the decision boundary has constrained to an axis-aligned boundary.
The regularization in the cosine problem is much neater (see Figure 4.17). The APL value moves from 5 to 3 and remains constant until the end. We also can see this effect in training loss at the end. Looking at the accuracy course, we notice that it decreases from 0.97 to 0.94 and remains still high, despite the regularization. This example gives us insight, how the APL course cohere with the model's accuracy.

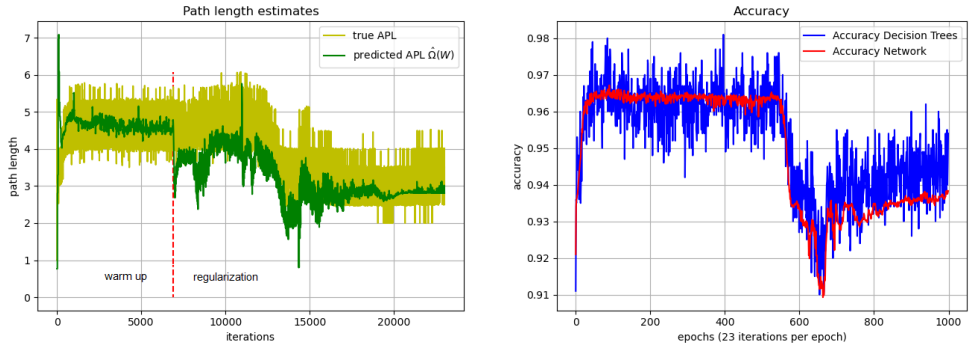


Figure 4.17: APL estimates (left) and the prediction accuracies (right) of the deep network and the underlying decision tree

4.5 Discussion

In the parabola problem, we can see that the APL moves constantly towards the end of the warm-up phase and slowly goes down as soon as the λ increases. At a certain point in the regularization phase, depending on how strong we choose the λ_{target}, the APL decreases to the value 1, and the deep model cannot longer learn any trees. This is also observable in training loss and the accuracy plot. As soon as the regularization goes stronger, the model parameters become sparse, the training loss increases and the accuracy score decreases. We do not have in the parabola experiment any strong indication, that the tree regularization favors axis-aligned decision boundaries. We run some other variations with different regularization strengths, and other optimizers like the SGD, but we could not see any improvements. It is difficult to give a clear explanation about the cause of these issues, whether if the data sets are too good-natured, or if tree regularization concept needs some improvements. We give some suggestions for it in Section 5.1.

On the other hand, we got some good results in the cosine problem. After epoch 570, the model's decision boundary becomes a tree-like boundary, and keeps it until the end. The accuracy behaves also in a convenient way. This experiment is significant for us, which shows that tree regularization could yield high fidelity. With larger regularization λ_{target}, the model does not behave well anymore, similar to the parabola problem (not shown in this work). Even if the results of the cosine problem are auspicious, we cannot fully guarantee their applicability on real data sets, because they would be larger in size and high dimensional. That is why we limited our experiments to synthetic data sets.

We conclude upon these results, that tree regularization tries to make the model behave like a decision tree, but still struggles to perform a smooth optimization. If we briefly compare it to a standard regularization with l_2-norm, the model is constrained (no overfitting), but the APLs remain the same throughout the whole training, see the figures below.

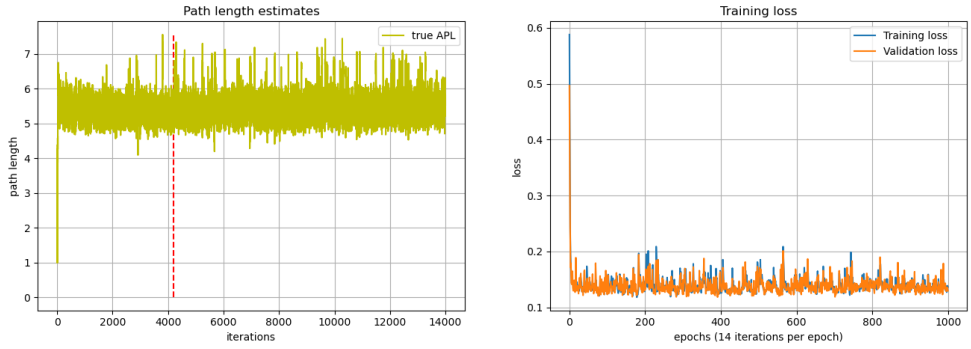


Figure 4.18: APLs of the underlying decision trees (left) and training loss over 100 epochs (right) of the deep model with l_2-regularization, and with learning rate 2e−2, and regularization strengths $\lambda_{\text{init}} = 1\text{e}{-3}$, and $\lambda_{\text{target}} = 1$. First fit the model for 300 epochs, and regularize for 700 epochs with the l_2-norm.

5 Conclusions

In this work, we got inspired by the meaning of interpretable models and studied specifically a novel optimization concept, the tree regularization, which belongs to the category of model-specific methods. The conviction was to optimally constrain deep networks into a far simpler, yet powerful model. By means of decision trees, we were able to regulate the model parameters of the deep network with the average decision path length function, which measures the complexity, so that the decision-making resembles that of a decision tree. This makes it easy to feed the input data and the network's predictions into a decision tree, and use this model for human simulation.
Using the example of synthetically generated data sets, we could test the tree regularization, and have met the issues that exist with this type of regularization and observed its limitations. These findings still do not ensure to use tree-regularized, interpretable models for larger data sets from critical domains.

5.1 Future Work

There is still a great need for further explorations. It is yet not entirely clear, when and how strongly one should use regularization. We have to find out, how the regularization strength can be applied more intuitively. Our approach to use a cooling function primarily allows us not to fixate on one strength value.
Further research needs to be done on the tree regularization concept, and extensive experiments with synthetic, maybe with higher dimensional data sets, should be conducted. This does not only give confidence in this method, but we can also find other limitations. One extension of tree regularization was proposed by Wu et al. [29]. We briefly explain its technical approach in Section 5.1.1.
The main disadvantage of our training pipeline is the training of the surrogate model after each epoch. It takes quite a long time, when the data set gets large. The complex network also makes augmentation more difficult, because thousands of new parameters have to be synthesized and written to an auxiliary model to infer the APLs. The total training time for the parabola problem was about 24 hours, and for the cosine problem about 72 hours. Therefore, the implementation needs to be further optimized.
Our work is only based on feed-forward networks. After the tree regularization method would have been thoroughly studied, different architectures types like the Recurrent Neural

Networks (RNN), auto-encoders, or even hybrid networks like the GRU-HMM proposed by Doshi-Velez and Kim [1] can be considered. Initially, the intention of Doshi-Velez and Kim [1] for tree regularization was the application of sequential data sets and to find out, whether there exist a high-accuracy-at-low-complexity sweet spot.

5.1.1 Regional Tree Regularization

The extended version of tree regularization is the *regional* tree regularization. Compared to the classical or global tree regularization, respectively, it favors constrained models, where the resulting trees are faithful to pre-defined, human-intuitive regions $\{X_1, \ldots, X_R\}$ of the input data. This is not a biased approach, but allows to better capture variations in the input data set, and to find learning machines, that are much closer to the optimal machines in the hypothesis space.

The regional tree regularization principle orientates on a l_0-norm based regularization (see Section 2.2.2.4), that penalizes the APL of the most complex region X_r

$$\Omega^{\text{regional-}l_0} \triangleq \max_{r \in \{1,\ldots,R\}} \Omega(X_r, f(\cdot; \boldsymbol{\theta})). \tag{5.1}$$

Bibliography

[1] F. Doshi-Velez and B. Kim, "Towards a rigorous science of interpretable machine learning," *arXiv preprint arXiv:1702.08608*, 2017.

[2] M. Wu, S. Parbhoo, M. C. Hughes, V. Roth, and F. Doshi-Velez, "Optimizing for Interpretability in Deep Neural Networks with Tree Regularization," *arXiv e-prints*, p. arXiv:1908.05254, Aug. 2019.

[3] C. Molnar, *Interpretable machine learning*. Lulu. com, 2020.

[4] R. Marcinkevičs and J. E. Vogt, "Interpretability and explainability: A machine learning zoo mini-tour," *arXiv preprint arXiv:2012.01805*, 2020.

[5] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should i trust you?" explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 1135–1144.

[6] A. B. Arrieta, N. Díaz-Rodríguez, J. Del Ser, A. Bennetot, S. Tabik, A. Barbado, S. García, S. Gil-López, D. Molina, R. Benjamins *et al.*, "Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai," *Information Fusion*, vol. 58, pp. 82–115, 2020.

[7] Z. Che, S. Purushotham, R. Khemani, and Y. Liu, "Distilling knowledge from deep networks with applications to healthcare domain," *arXiv preprint arXiv:1512.03542*, 2015.

[8] M. A. Ahmad, C. Eckert, and A. Teredesai, "Interpretable machine learning in healthcare," in *Proceedings of the 2018 ACM international conference on bioinformatics, computational biology, and health informatics*, 2018, pp. 559–560.

[9] V. Podgorelec, P. Kokol, B. Stiglic, and I. Rozman, "Decision trees: an overview and their use in medicine," *Journal of medical systems*, vol. 26, no. 5, pp. 445–463, 2002.

[10] M. Wu, M. C. Hughes, S. Parbhoo, M. Zazzi, V. Roth, and F. Doshi-Velez, "Beyond sparsity: Tree regularization of deep models for interpretability," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[11] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[12] V. Vapnik, *The nature of statistical learning theory*. Springer science & business media, 2013.

[13] S. J. Russell and P. Norvig, *Artificial Intelligence: a modern approach*, 3rd ed. Pearson, 2009.

[14] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[15] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[16] S. Haykin, *Neural networks and learning machines, 3/E*. Pearson Education India, 2010.

[17] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[18] S. Theodoridis and K. Koutroumbas, *Pattern Recognition*, 4th ed. Elsevier, 2009.

[19] P. E. Hart, D. G. Stork, and R. O. Duda, *Pattern Classification*. Wiley Hoboken, 2000.

[20] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2019.

[21] "MIT Introduction to Deep Learning 6.S191," note = (accessed: 29.09.2021). [Online]. Available: http://introtodeeplearning.com/

[22] "Overfitting," IBM Cloud Education, note = (accessed: 17.10.2021). [Online]. Available: https://www.ibm.com/cloud/learn/overfitting

[23] "L^p space," Wikipedia, note = (accessed: 20.07.2021). [Online]. Available: https://en.wikipedia.org/wiki/Lp_space

[24] M. P. Deisenroth, A. A. Faisal, and C. S. Ong, *Mathematics for Machine Learning*. Cambridge University Press, 2020.

[25] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[26] A. Kolmogorov, *On the representation of continuous functions of several variables by superpositions of continuous functions of a smaller number of variables*.

[27] V. I. Arnol'd, "On functions of three variables," in *Doklady Akademii Nauk*, vol. 114, no. 4. Russian Academy of Sciences, 1957, pp. 679–681.

[28] G. Montúfar, R. Pascanu, K. Cho, and Y. Bengio, "On the number of linear regions of deep neural networks," *arXiv preprint arXiv:1402.1869*, 2014.

[29] M. Wu, S. Parbhoo, M. Hughes, R. Kindle, L. Celi, M. Zazzi, V. Roth, and F. Doshi-Velez, "Regional Tree Regularization for Interpretability in Black Box Models," *arXiv e-prints*, p. arXiv:1908.04494, Aug. 2019.

[30] A comparison of cooling schedules for simulated annealing (artificial intelligence). (accessed: 28.07.2021). [Online]. Available: http://what-when-how.com/artificial-intelligence/a-comparison-of-cooling-schedules-for-simulated-annealing-artificial-intelligence/

[31] M. Craven and J. Shavlik, "Extracting tree-structured representations of trained networks," *Advances in neural information processing systems*, vol. 8, pp. 24–30, 1995.

[32] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 618–626.

[33] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson, "Understanding neural networks through deep visualization," *arXiv preprint arXiv:1506.06579*, 2015.

[34] "Cross-validation (statistics)," (accessed: 20.10.2021). [Online]. Available: https://en.wikipedia.org/wiki/Cross-validation_(statistics)

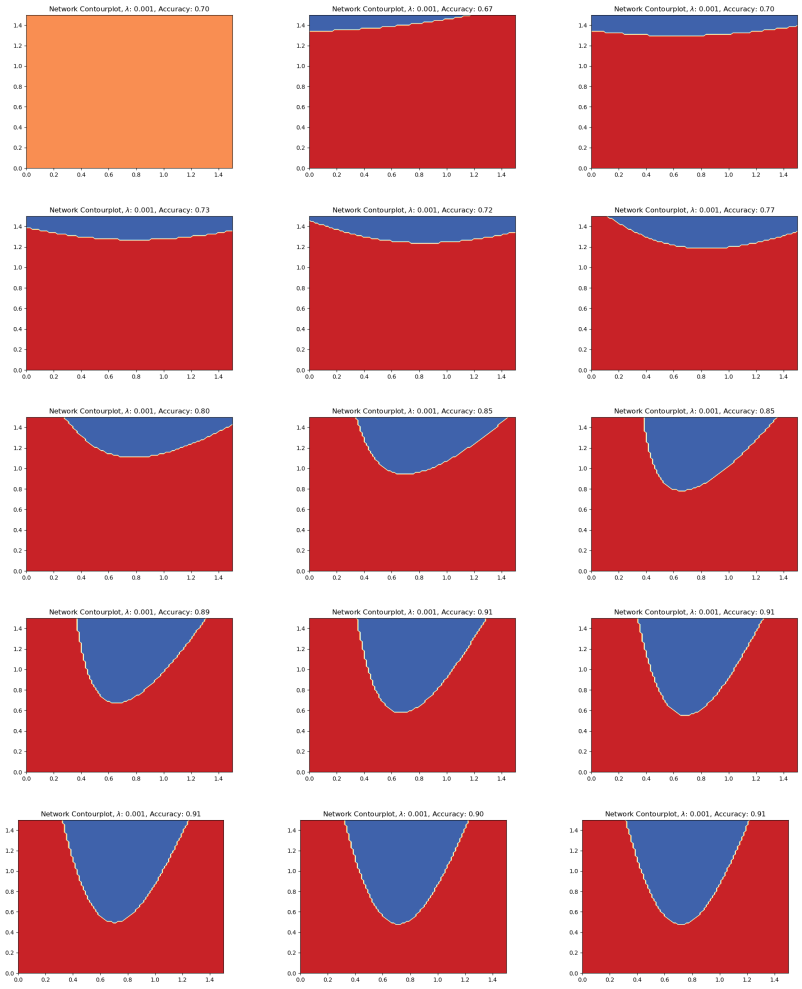
[35] J. Friedman, R. Tibshirani, and T. Hastie, *The elements of statistical learning: Data mining, inference, and prediction*. Springer, 2008.

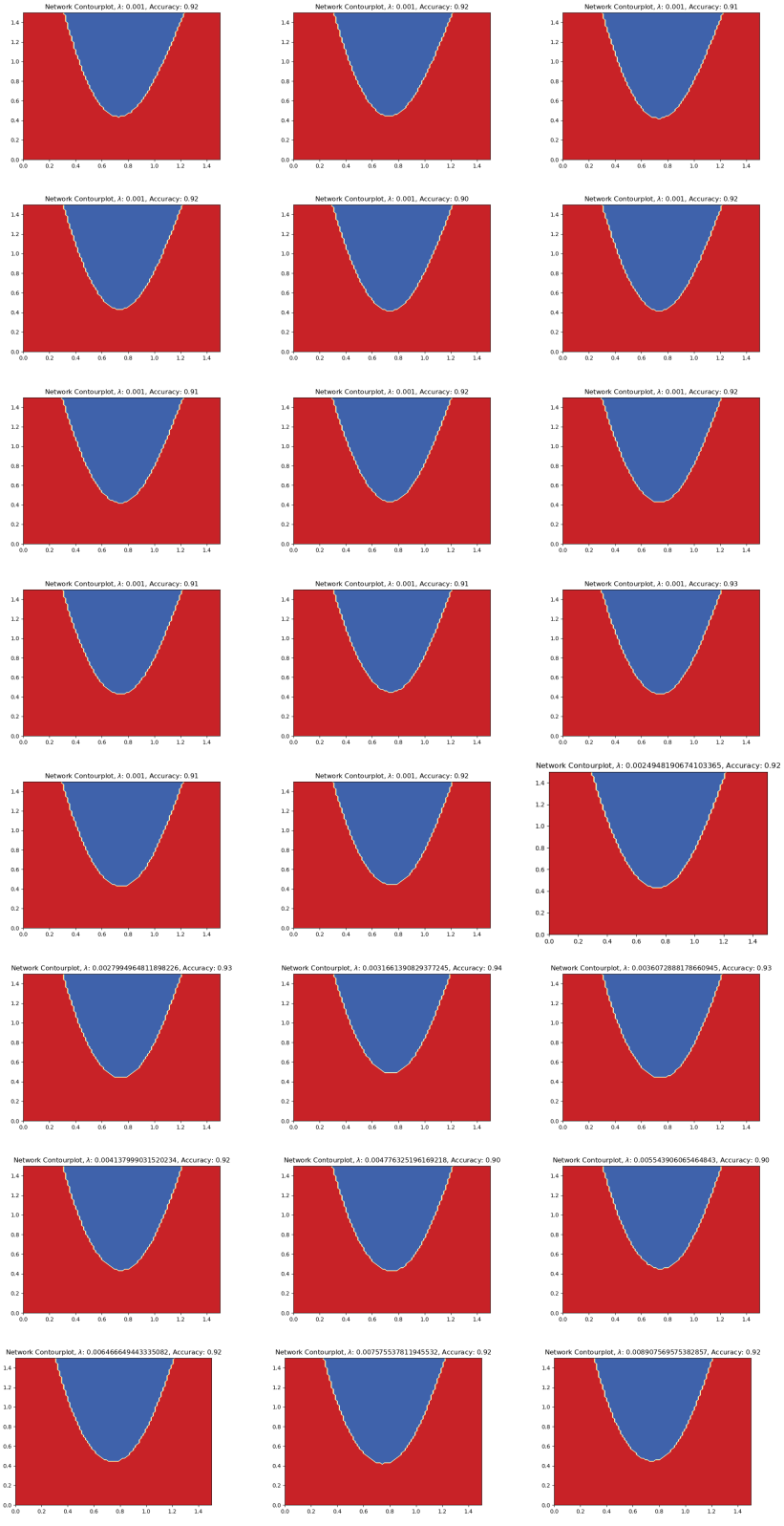
[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and Édouard Duchesnay, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. 85, pp. 2825–2830, 2011. [Online]. Available: http://jmlr.org/papers/v12/pedregosa11a.html

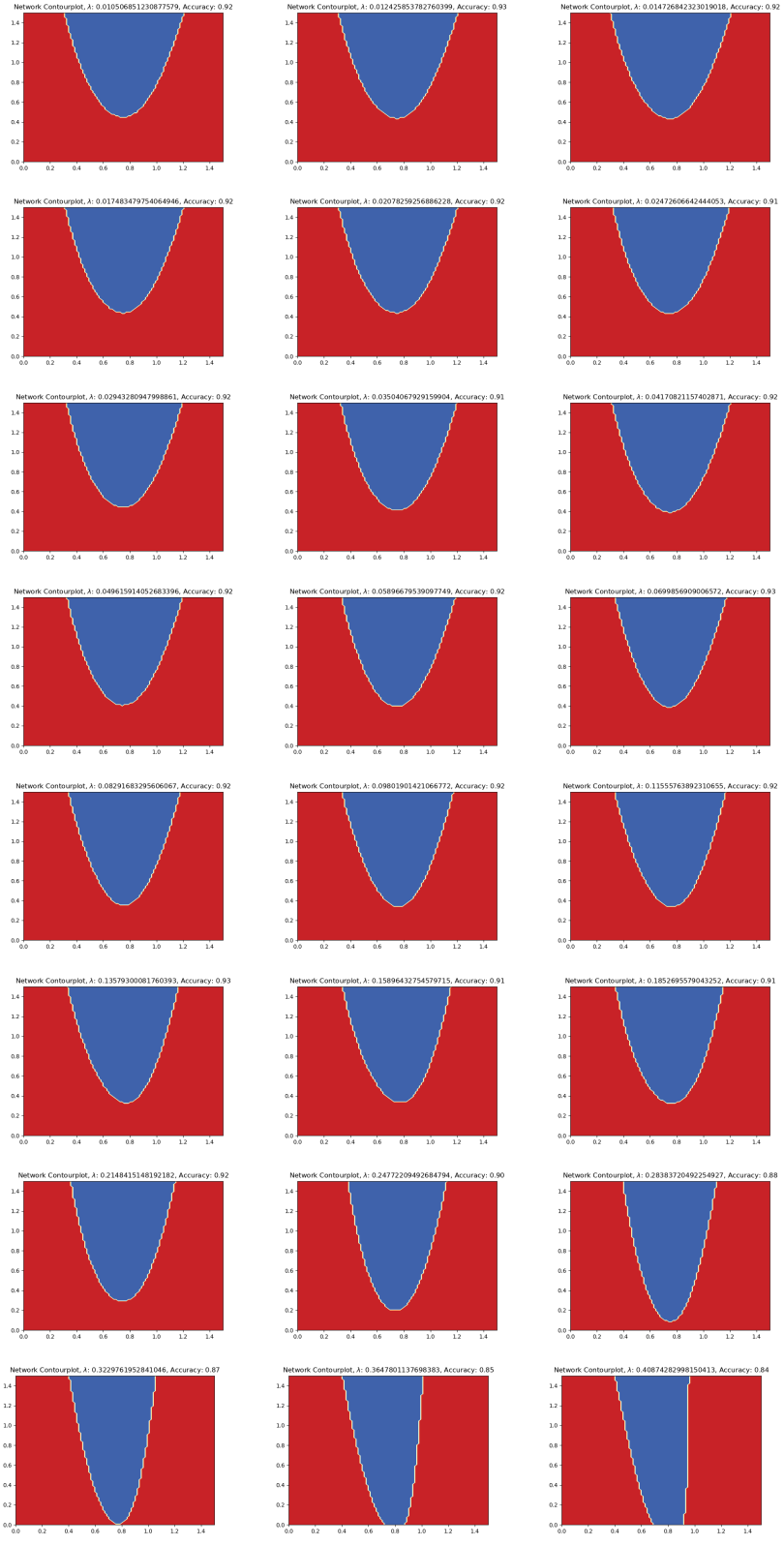
A
Appendix

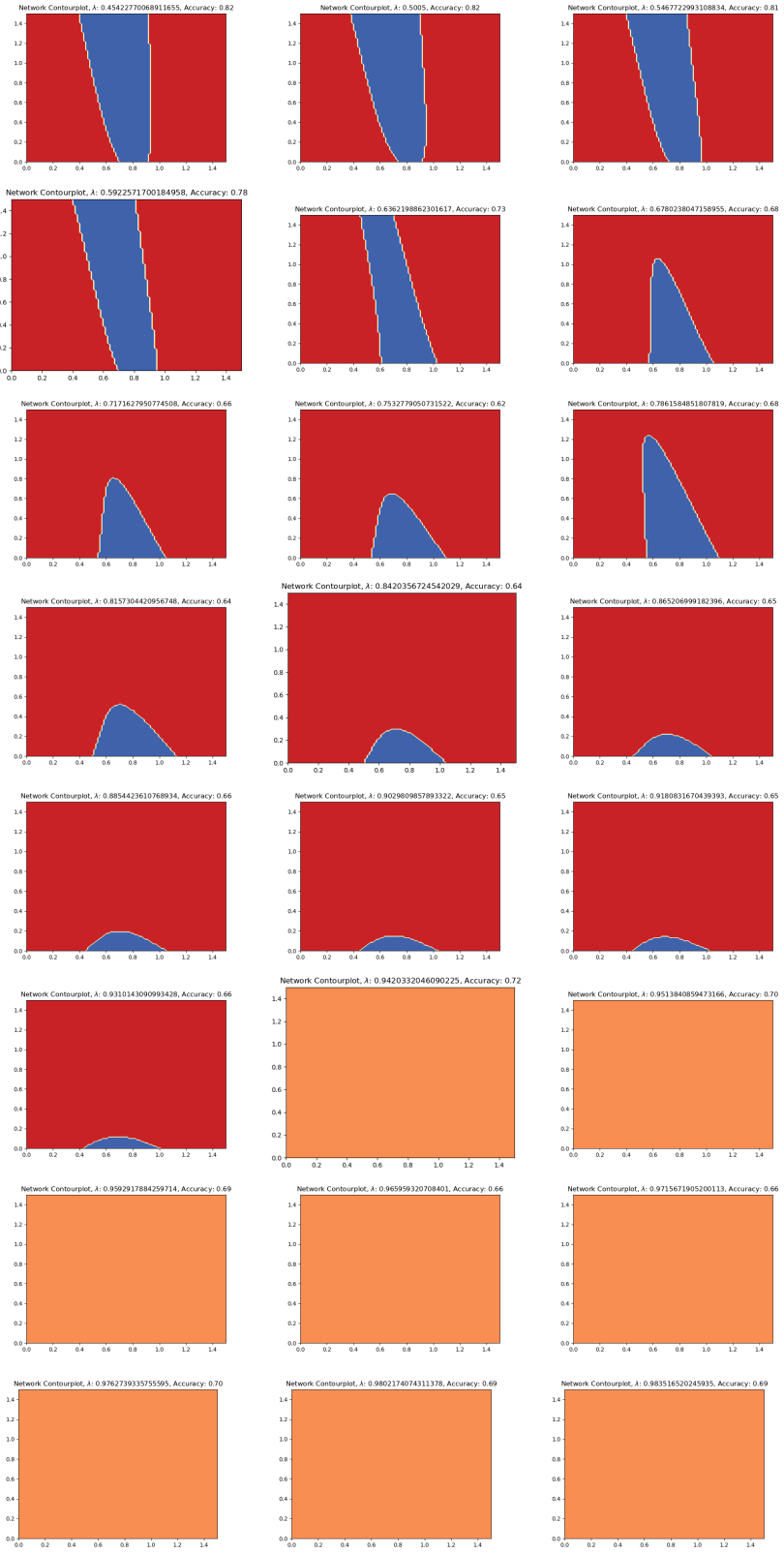
A.1 Parabola Problem

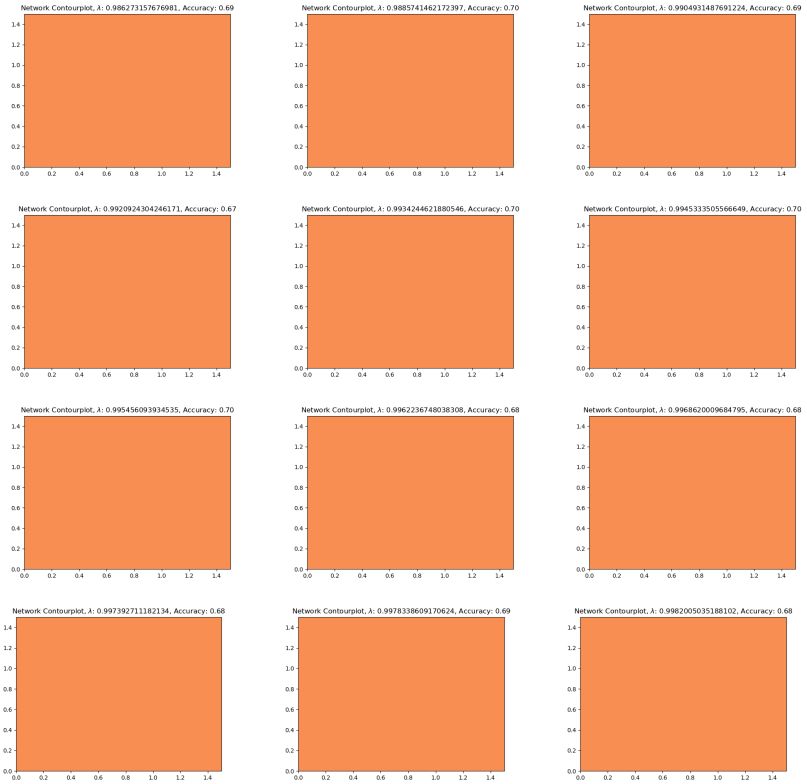
We show here all snapshots from every 10^{th} epoch of the first parabola experiment with $\lambda_{\text{init}} = 1\text{e}{-3}$ and $\lambda_{\text{target}} = 1$.




Network Contourplot, λ: 0.001, Accuracy: 0.92
Network Contourplot, λ: 0.001, Accuracy: 0.92
Network Contourplot, λ: 0.001, Accuracy: 0.91
Network Contourplot, λ: 0.001, Accuracy: 0.92
Network Contourplot, λ: 0.001, Accuracy: 0.90
Network Contourplot, λ: 0.001, Accuracy: 0.92
Network Contourplot, λ: 0.001, Accuracy: 0.91
Network Contourplot, λ: 0.001, Accuracy: 0.92
Network Contourplot, λ: 0.001, Accuracy: 0.92
Network Contourplot, λ: 0.001, Accuracy: 0.91
Network Contourplot, λ: 0.001, Accuracy: 0.91
Network Contourplot, λ: 0.001, Accuracy: 0.93
Network Contourplot, λ: 0.001, Accuracy: 0.91
Network Contourplot, λ: 0.001, Accuracy: 0.92
Network Contourplot, λ: 0.0024948190674103365, Accuracy: 0.92
Network Contourplot, λ: 0.0027994964811898226, Accuracy: 0.93
Network Contourplot, λ: 0.0031661390829377245, Accuracy: 0.94
Network Contourplot, λ: 0.0036072888178660945, Accuracy: 0.93
Network Contourplot, λ: 0.004137999031520234, Accuracy: 0.92
Network Contourplot, λ: 0.004776325196169218, Accuracy: 0.90
Network Contourplot, λ: 0.005543906065464843, Accuracy: 0.90
Network Contourplot, λ: 0.006466649443335082, Accuracy: 0.92
Network Contourplot, λ: 0.007575537811945532, Accuracy: 0.92
Network Contourplot, λ: 0.008907569575382857, Accuracy: 0.92

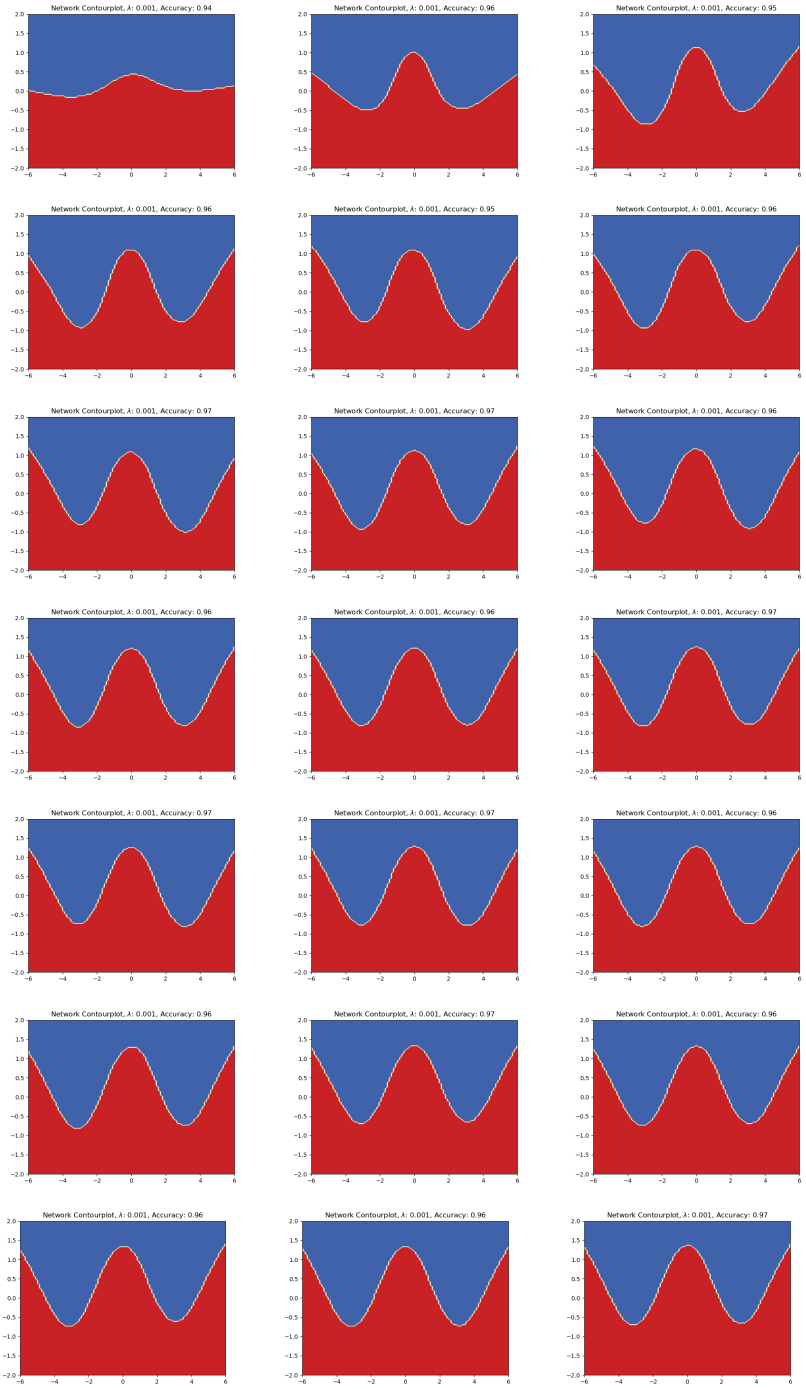

Network Contourplot, λ: 0.0105068512308775 79, Accuracy: 0.92
Network Contourplot, λ: 0.012425853782760399, Accuracy: 0.93
Network Contourplot, λ: 0.014726842323019018, Accuracy: 0.92
Network Contourplot, λ: 0.017483479754064946, Accuracy: 0.92
Network Contourplot, λ: 0.02078259256886228, Accuracy: 0.92
Network Contourplot, λ: 0.02472606642444053, Accuracy: 0.91
Network Contourplot, λ: 0.02943280947998861, Accuracy: 0.92
Network Contourplot, λ: 0.0350406792915990 4, Accuracy: 0.91
Network Contourplot, λ: 0.04170821157402871, Accuracy: 0.92
Network Contourplot, λ: 0.04961591405268339 6, Accuracy: 0.92
Network Contourplot, λ: 0.05896679539097749, Accuracy: 0.92
Network Contourplot, λ: 0.06998569090065 72, Accuracy: 0.93
Network Contourplot, λ: 0.08291683295606067, Accuracy: 0.92
Network Contourplot, λ: 0.09801901421066772, Accuracy: 0.92
Network Contourplot, λ: 0.11555763892310655, Accuracy: 0.92
Network Contourplot, λ: 0.13579300081760393, Accuracy: 0.93
Network Contourplot, λ: 0.15896432754579715, Accuracy: 0.91
Network Contourplot, λ: 0.18526955790432 52, Accuracy: 0.91
Network Contourplot, λ: 0.21848151481921 82, Accuracy: 0.92
Network Contourplot, λ: 0.24772209492684794, Accuracy: 0.90
Network Contourplot, λ: 0.28383720492254927, Accuracy: 0.88
Network Contourplot, λ: 0.32297619528410 46, Accuracy: 0.87
Network Contourplot, λ: 0.36478011376983 83, Accuracy: 0.85
Network Contourplot, λ: 0.40874282998150413, Accuracy: 0.84

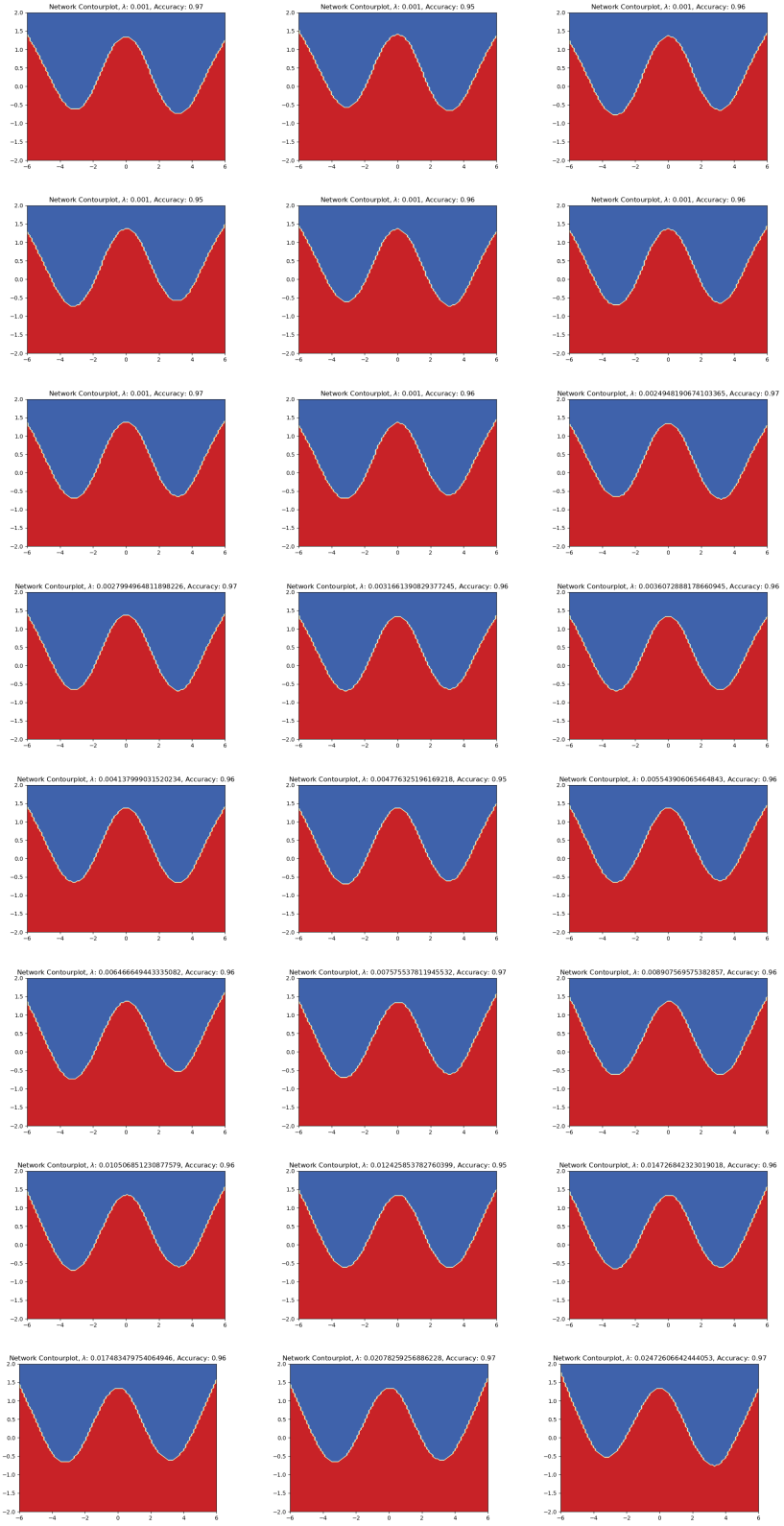

Network Contourplot, λ: 0.45422770068911655, Accuracy: 0.82
Network Contourplot, λ: 0.5005, Accuracy: 0.82
Network Contourplot, λ: 0.546772993108834, Accuracy: 0.81
Network Contourplot, λ: 0.5922571700184958, Accuracy: 0.78
Network Contourplot, λ: 0.6362198862301617, Accuracy: 0.73
Network Contourplot, λ: 0.6780238047158955, Accuracy: 0.68
Network Contourplot, λ: 0.7171627950774508, Accuracy: 0.66
Network Contourplot, λ: 0.7532779050731522, Accuracy: 0.62
Network Contourplot, λ: 0.7861584851807819, Accuracy: 0.68
Network Contourplot, λ: 0.8420356724542029, Accuracy: 0.64
Network Contourplot, λ: 0.8157304420956748, Accuracy: 0.64
Network Contourplot, λ: 0.8652069991823 96, Accuracy: 0.65
Network Contourplot, λ: 0.8854423610768934, Accuracy: 0.66
Network Contourplot, λ: 0.9029809857893322, Accuracy: 0.65
Network Contourplot, λ: 0.9180831670439393, Accuracy: 0.65
Network Contourplot, λ: 0.9420332046090225, Accuracy: 0.72
Network Contourplot, λ: 0.931014309099342 8, Accuracy: 0.66
Network Contourplot, λ: 0.9513840859473166, Accuracy: 0.70
Network Contourplot, λ: 0.9592917884259714, Accuracy: 0.69
Network Contourplot, λ: 0.965959320708401, Accuracy: 0.66
Network Contourplot, λ: 0.9715671905200113, Accuracy: 0.66
Network Contourplot, λ: 0.9762739335755595, Accuracy: 0.70
Network Contourplot, λ: 0.9802174074311378, Accuracy: 0.69
Network Contourplot, λ: 0.9835165202459 35, Accuracy: 0.69

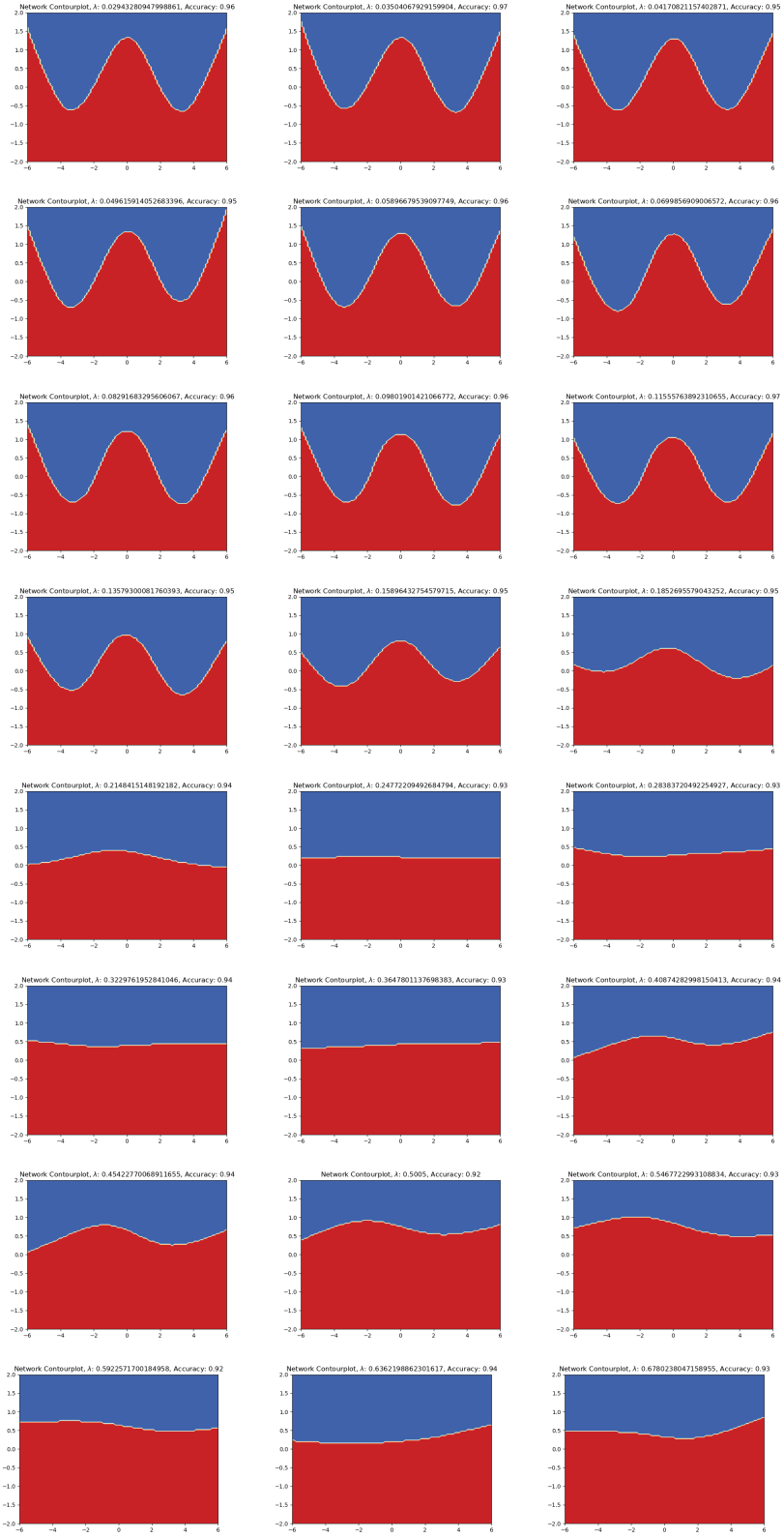

Network Contourplot, λ: 0.986273157676981, Accuracy: 0.69
Network Contourplot, λ: 0.9885741462172397, Accuracy: 0.70
Network Contourplot, λ: 0.990493148769122 4, Accuracy: 0.69
Network Contourplot, λ: 0.9920924304246171, Accuracy: 0.67
Network Contourplot, λ: 0.9934244621880546, Accuracy: 0.70
Network Contourplot, λ: 0.9945333505566649, Accuracy: 0.70
Network Contourplot, λ: 0.995456093934535, Accuracy: 0.70
Network Contourplot, λ: 0.9962236748038308, Accuracy: 0.68
Network Contourplot, λ: 0.9968620009684795, Accuracy: 0.68
Network Contourplot, λ: 0.997392711182134, Accuracy: 0.68
Network Contourplot, λ: 0.9978338609170624, Accuracy: 0.69
Network Contourplot, λ: 0.9982005035188102, Accuracy: 0.68

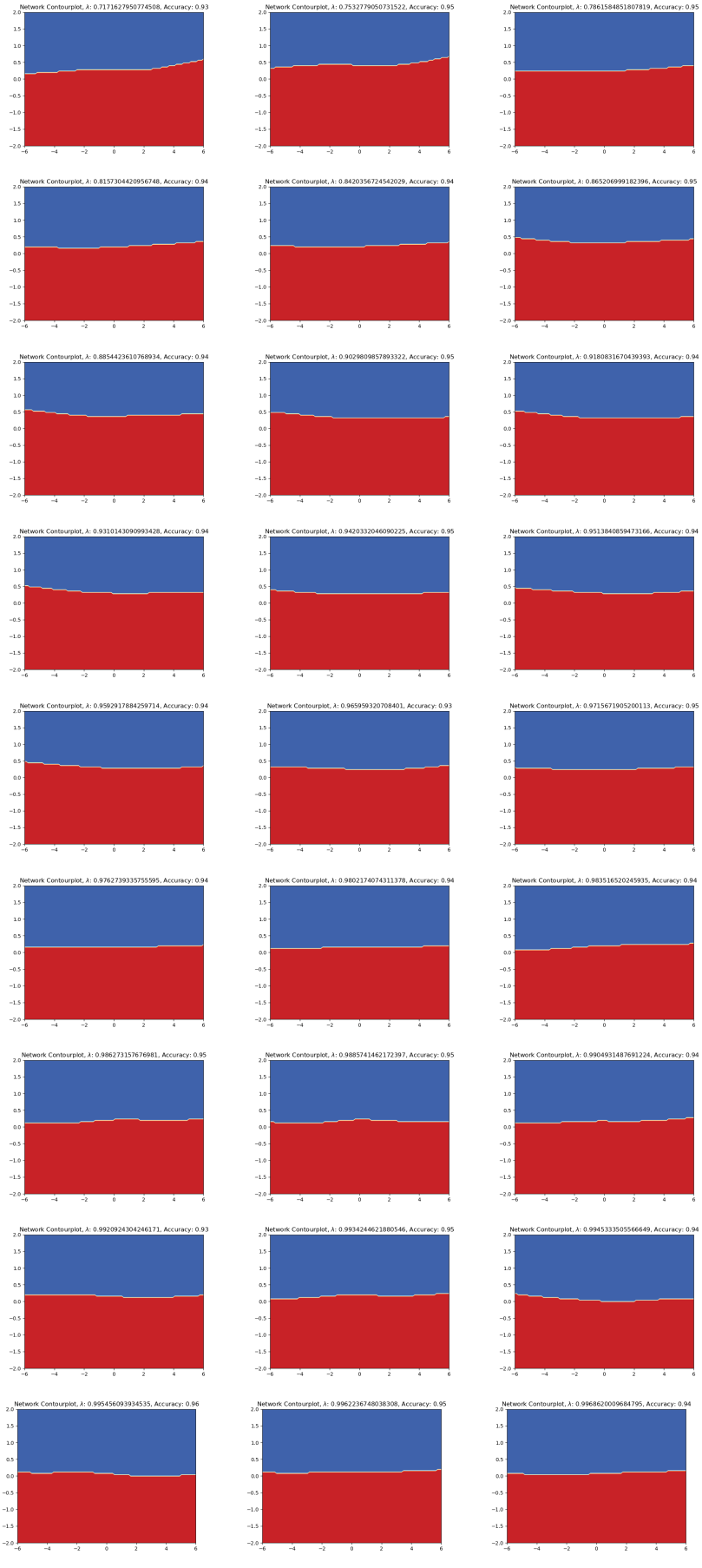
A.2 Cosine Problem

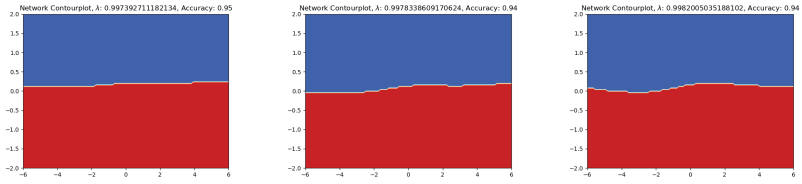
We show here all snapshots from every 10^{th} epoch of the cosine experiment with $\lambda_{\text{init}} = 1\text{e}{-3}$ and $\lambda_{\text{target}} = 1$.




Network Contourplot, λ: 0.001, Accuracy: 0.97
Network Contourplot, λ: 0.001, Accuracy: 0.95
Network Contourplot, λ: 0.001, Accuracy: 0.96
Network Contourplot, λ: 0.001, Accuracy: 0.95
Network Contourplot, λ: 0.001, Accuracy: 0.96
Network Contourplot, λ: 0.001, Accuracy: 0.96
Network Contourplot, λ: 0.001, Accuracy: 0.97
Network Contourplot, λ: 0.001, Accuracy: 0.96
Network Contourplot, λ: 0.0024948190674103365, Accuracy: 0.97
Network Contourplot, λ: 0.0027994964811898226, Accuracy: 0.97
Network Contourplot, λ: 0.0031661390829377245, Accuracy: 0.96
Network Contourplot, λ: 0.0036072888178660945, Accuracy: 0.96
Network Contourplot, λ: 0.004137999031520234, Accuracy: 0.96
Network Contourplot, λ: 0.0047763251961692 18, Accuracy: 0.95
Network Contourplot, λ: 0.005543906065464843, Accuracy: 0.96
Network Contourplot, λ: 0.006466649443335082, Accuracy: 0.96
Network Contourplot, λ: 0.007575537811945532, Accuracy: 0.97
Network Contourplot, λ: 0.008907569575382857, Accuracy: 0.96
Network Contourplot, λ: 0.010506851230877579, Accuracy: 0.96
Network Contourplot, λ: 0.012425853782760399, Accuracy: 0.95
Network Contourplot, λ: 0.014726842323019018, Accuracy: 0.96
Network Contourplot, λ: 0.017483479754064946, Accuracy: 0.96
Network Contourplot, λ: 0.020782592568862 28, Accuracy: 0.97
Network Contourplot, λ: 0.024726066424440 53, Accuracy: 0.97


Network Contourplot, λ: 0.02943280947998861, Accuracy: 0.96
Network Contourplot, λ: 0.03504067929159904, Accuracy: 0.97
Network Contourplot, λ: 0.04170821157402871, Accuracy: 0.95
Network Contourplot, λ: 0.04961591405268339 6, Accuracy: 0.95
Network Contourplot, λ: 0.05896679539097749, Accuracy: 0.96
Network Contourplot, λ: 0.06998569090065 72, Accuracy: 0.96
Network Contourplot, λ: 0.08291683295606067, Accuracy: 0.96
Network Contourplot, λ: 0.09801901421066772, Accuracy: 0.96
Network Contourplot, λ: 0.11555763892310655, Accuracy: 0.97
Network Contourplot, λ: 0.13579300081760393, Accuracy: 0.95
Network Contourplot, λ: 0.15896432754579715, Accuracy: 0.95
Network Contourplot, λ: 0.18526955790432 52, Accuracy: 0.95
Network Contourplot, λ: 0.2184151481921 82, Accuracy: 0.94
Network Contourplot, λ: 0.24772209492684794, Accuracy: 0.93
Network Contourplot, λ: 0.28383720492254927, Accuracy: 0.93
Network Contourplot, λ: 0.3229761952841046, Accuracy: 0.94
Network Contourplot, λ: 0.3647801137698383, Accuracy: 0.93
Network Contourplot, λ: 0.40874282998150413, Accuracy: 0.94
Network Contourplot, λ: 0.45422770068911655, Accuracy: 0.94
Network Contourplot, λ: 0.5005, Accuracy: 0.92
Network Contourplot, λ: 0.5467722993108834, Accuracy: 0.93
Network Contourplot, λ: 0.5922571700184958, Accuracy: 0.92
Network Contourplot, λ: 0.6362198862301617, Accuracy: 0.94
Network Contourplot, λ: 0.6780238047158955, Accuracy: 0.93


Network Contourplot, λ: 0.7171627950774508, Accuracy: 0.93
Network Contourplot, λ: 0.7532779050731522, Accuracy: 0.95
Network Contourplot, λ: 0.7861584851807819, Accuracy: 0.95
Network Contourplot, λ: 0.815730442095674 8, Accuracy: 0.94
Network Contourplot, λ: 0.8420356724542029, Accuracy: 0.94
Network Contourplot, λ: 0.865206999182396, Accuracy: 0.95
Network Contourplot, λ: 0.8854423610768934, Accuracy: 0.94
Network Contourplot, λ: 0.9029809857893322, Accuracy: 0.95
Network Contourplot, λ: 0.9180831670439393, Accuracy: 0.94
Network Contourplot, λ: 0.9310143090993428, Accuracy: 0.94
Network Contourplot, λ: 0.9420332046090225, Accuracy: 0.95
Network Contourplot, λ: 0.9513840859473166, Accuracy: 0.94
Network Contourplot, λ: 0.9592917884259714, Accuracy: 0.94
Network Contourplot, λ: 0.965959320708401, Accuracy: 0.93
Network Contourplot, λ: 0.9715671905200113, Accuracy: 0.95
Network Contourplot, λ: 0.9762739335755595, Accuracy: 0.94
Network Contourplot, λ: 0.9802174074311378, Accuracy: 0.94
Network Contourplot, λ: 0.983516520245935, Accuracy: 0.94
Network Contourplot, λ: 0.986273157676981, Accuracy: 0.95
Network Contourplot, λ: 0.9885741462172397, Accuracy: 0.95
Network Contourplot, λ: 0.9904931487691224, Accuracy: 0.94
Network Contourplot, λ: 0.9920924304246171, Accuracy: 0.93
Network Contourplot, λ: 0.9934244621880546, Accuracy: 0.95
Network Contourplot, λ: 0.9945333505566649, Accuracy: 0.94
Network Contourplot, λ: 0.995456093934535, Accuracy: 0.96
Network Contourplot, λ: 0.9962236748038308, Accuracy: 0.95
Network Contourplot, λ: 0.9968620009684795, Accuracy: 0.94


Network Contourplot, λ: 0.997392711182134, Accuracy: 0.95
Network Contourplot, λ: 0.9978338609170624, Accuracy: 0.94
Network Contourplot, λ: 0.9982005035188102, Accuracy: 0.94

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

includes Declaration on Plagiarism and Fraud
beinhaltet Erklärung zu Plagiat und Betrug

Author — Autor
Gowthaman Gobalasingam

Matriculation number — Matrikelnummer
2016-050-619

Title of work — Titel der Arbeit
Tree Regularization of Deep Networks

Type of work — Typ der Arbeit
Master's Thesis

Declaration — Erklärung
I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Basel, October 25, 2021

Signature — Unterschrift