Novel Constructive Neural Network Algorithms for Supervised Learning

Theodor Hartleitner

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Mobile Computing

in Hagenberg

im Juni 2023

Advisor:

Ing. Dr. Rainhard Dieter Findling BSc MSc

Declaration

Hagenberg, June 19, 2023

Theodor Hartleitner

Contents

Abstract

Artificial Neural Networks (ANNs) are valuable tools in artificial intelligence and machine learning. Advances in recent years allowed these methods to contribute greatly to advancements in fields such as computer vision, natural language processing, or speech recognition. The performance of ANNs heavily depends on their architecture, making its optimal selection a crucial yet complex task. Constructive Neural Networks (CoNNs) address this challenge by learning not only the network parameters but also finding fitting network topologies throughout learning. This work presents three novel CoNN algorithms: Layerwise, CasCor Ultra, and Uncertainty Splitting. We conduct multiple evaluations on a number of datasets for each algorithm including various hyperparameter configurations to assess overall performance, generated network sizes, and construction efficiency. To measure the computational cost of construction in a robust way, we additionally developed the novel construction effort (CE) metric. Additionally, three state-of-the-art CoNN algorithms are evaluated on the same datasets to enable meaningful comparisons with our proposed algorithms. We furthermore add a simple magnitude pruning step to our algorithms and evaluate variants with and without pruning to give additional insights into the effectiveness of each approach. In order to simplify large-scale evaluations using multiple algorithms, hyperparameter sets, benchmark datasets, and random reruns, we developed an open-source software system capable of automatically performing such evaluations. The results of the evaluations reveal notable potential in our proposed algorithms. CasCor Ultra outperformed the reference algorithms in constructing especially compact networks with low parameter counts. Uncertainty Splitting exhibited superior network construction efficiency due to its large and targeted construction steps. Finally, the Layerwise algorithm manages to show the best balance of performance, final network size, and construction effort in comparison to the other evaluated algorithms. Incorporating the simple magnitude pruning step into the evaluation demonstrated a notable decrease in the final parameters in the network, particularly with the CasCor Ultra, which halved the number of parameters across multiple datasets without sacrificing performance accuracy. Overall, we find that the three proposed algorithms each outperform the state-of-the-art in one or multiple areas and thus show potential for advancing the field of CoNN algorithms.

Kurzfassung

Künstliche neuronale Netze (ANNs) sind wertvolle Werkzeuge im Bereich künstliche Intelligenz und dem maschinellen Lernen. Fortschritte in den letzten Jahren haben es diesen Methoden ermöglicht, maßgeblich zu Entwicklungen in Bereichen wie Maschinelles Sehen, Sprachverarbeitung, Spracherkennung usw. beizutragen. Die Leistung von ANNs hängt jedoch stark von ihrer Architektur ab, womit die Auswahl dieser eine entscheidende, aber komplexe Aufgabe darstellt. Konstruktive neuronale Netze (CoNNs) bewältigen diese Herausforderung, indem sie nicht nur die Netzwerkparameter lernen, sondern auch passende Netzwerkstrukturen während des Lernens finden. Diese Arbeit stellt drei neue CoNN-Algorithmen vor: Layerwise, CasCor Ultra und Uncertainty Splitting. Wir haben mehrere Auswertungen mit einer Reihe von Datensätzen für jeden Algorithmus, einschließlich verschiedener Hyperparameter-Konfigurationen durchgeführt. Dadurch versuchen wir die Leistung, die Netzwerkgrößen und Konstruktionseffizienz bewerten zu können. Um die Berechnungskosten der Konstruktion auf robuste Weise zu messen, haben wir zusätzlich eine neue Konstruktionsaufwand Metrik entwickelt. Zusätzlich wurden drei CoNN-Algorithmen, welche dem derzeitigen Stand der Technik entsprechen, auf den gleichen Datensätzen ausgewertet. Das erlaubt uns aussagekräftige Vergleiche mit unseren Algorithmen zu machen. Es wurde weiters ein einfacher Schritt zur Verringerung von Netzwerkparametern hinzugefügt und Varianten des Algorithmus mit und ohne dieser Modifikation wurden ausgewertet, um weitere Einblicke in die Wirksamkeit jedes Ansatzes zu erhalten. Um aufwändige Auswertungen mit mehreren Algorithmen, Hyperparameter Konfigurationen, Datensätzen und zufälligen Wiederholungen zu vereinfachen, haben wir ein quelloffenes Programm entwickelt, das solche Auswertungen automatisch durchführt. Die Ergebnisse der Auswertungen zeigen beachtliches Potenzial in unseren vorgeschlagenen Algorithmen. CasCor Ultra übertraf alle Referenzalgorithmen bei der Konstruktion besonders kompakter Netzwerke mit niedrigen Parameterzahlen. Uncertainty Splitting zeigte aufgrund seiner großen und gezielten Konstruktionsschritte eine überlegene Netzwerkkonstruktionseffizienz. Der Layerwise Algorithmus schaffte es, das beste Gleichgewicht zwischen Leistung, endgültiger Netzwerkgröße und Konstruktionsaufwand im Vergleich zu den anderen ausgewerteten Algorithmen zu zeigen. Die Einbeziehung der Parameter-Eliminierung in die Auswertung zeigte eine bemerkenswerte Verringerung der Parameter im Netzwerk, insbesondere beim CasCor Ultra, der die Anzahl der Parameter über mehrere Datensätze hinweg halbierte, ohne die Genauigkeit zu beeinträchtigen. Insgesamt stellen wir fest, dass die drei vorgeschlagenen Algorithmen den derzeitigen Stand der Technik in einem oder mehreren Bereichen übertreffen und somit das Potenzial haben, den Bereich der CoNN-Algorithmen voranzubringen.

Chapter 1

Introduction

In this chapter, we introduce the main topics discussed in this work and provide important information for the rest of this thesis. Our motivations for choosing this area of research are outlined in section 1.1. In section 1.2 we present the goals we aim to achieve with this work as well as the corresponding research questions. The chapter concludes with an overview of the organization of the thesis in section 1.3.

1.1 Motivation

Artificial neural networks (ANNs) are useful tools in artificial intelligence and machine learning. Their capabilities as universal function approximators [41], as well as their abilities to discern patterns and extract features, enable them to have applications in various areas of machine learning. ANNs are also highly versatile as they can handle both regression and classification tasks in supervised learning. These attributes enabled neural networks to contribute greatly to the advancements in fields such as computer vision, natural language processing, speech recognition, automotive computing, etc. [1].

The performance of ANNs is highly dependent on its network architecture [51], that is, the number of layers and nodes in the network and how they are connected. An architecture that is too small will not be able to capture the full problem complexity, while an architecture with too many parameters will slow down training and could lead to overfitting if suitable regularization is not applied [67]. The architectures of conventional ANNs are determined in advance and do not change during training, which means that selecting a fitting architecture is very important. In practice, it tends to be selected through trial-and-error as well as using expertise and experience [100]. Given these limitations, it is worth considering if more dynamic approaches with adaptable architectures might improve the application and performance of ANNs. Biological brains, which are continuously evolving neural networks, also hint at the potential for a more dynamic approach to network architecture [85]. Biological neural networks grow and adapt, contributing to a continually changing brain structure, which is crucial for learning and information storage [15, 68].

Constructive Neural Networks (CoNNs) [27, 48, 53] offer a solution to the limitations mentioned above. CoNN approaches not only learn the network parameters but also the network topology. They initially start with a minimal network architecture, such as fully connected input and output layers, and incrementally construct a fitting network topology throughout the training process. The construction process is performed by iteratively adding additional neurons and connections to the network. This allows the network to gradually improve its performance and become more accurate in its predictions. CoNN approaches provide some major advantages to conventional ANNs: First, there is no need to manually find well-performing network architectures. Next, the constructed architectures usually have fewer parameters than a network created by trial-and-error would have [67]. This is because small solutions are explored first. Smaller architectures also make CoNNs less memory intensive and generally faster in the inference stage. Finally, their dynamic nature makes it possible to apply to problems with changing complexity and evolving computation requirements. By integrating pruning methods into the construction process, this adaptability of CoNNs can be further improved leading to even better-performing solutions [35, 98, 103].

Overall, constructive neural network algorithms have the potential to improve various areas of machine learning and alleviate some of the complexities associated with ANNs. They provide an intuitive approach that aligns more closely with our understanding of biological learning mechanisms. By introducing novel ideas and refining existing approaches in this space, we aim to contribute to the improvements of constructive neural networks. We believe that advancements in this area will lead to the creation of more efficient and adaptable machine-learning approaches.

1.2 Goals and Research Questions

The main objective of this thesis is to advance the field of constructive neural networks by introducing new ideas and methods in this space. This objective is divided into three sub-goals that we aim to achieve with this thesis. First and most important, we propose and evaluate three novel CoNN or GPNN approaches. These algorithms are developed with the aim to offer competitive performance against the current state-of-the-art and demonstrate notable improvements on various metrics such as training efficiency, size of the constructed network, and learning accuracy. The second goal is the creation of an open-source modular software system that allows the automatic evaluation of multiple constructive neural network algorithms. This evaluation system should include implementations of multiple state-of-the-art CoNN algorithms and a wide range of benchmark datasets. By allowing other researchers to use and extend this software we hope to facilitate easy testing and comparisons of new constructive algorithms in the future. As the third and final objective, we hope to present a comprehensive review of the current state of research in the field of CoNN and GPNN algorithms. This involves the analysis of the most important algorithms as well as the identification of the challenges and opportunities for further investigation. We selected this objective as the current literature seems to lack a research review of such depth.

In order to achieve these goals, we pose the following research questions:

1. **RQ 1**: How well do our proposed algorithms perform in terms of training efficiency, network architecture, and final performance compared to state-of-the-art CoNN algorithms, and under what conditions does improvement occur?
2. **RQ 2**: How does the introduction of a magnitude pruning step influence the train-

ing process, network architecture, and performance of our proposed algorithms? What trade-offs exist between the purely constructive and grow-and-prune variants of those algorithms?

Comprehensive evaluations of our proposed algorithms enable us to address these research questions and contribute towards our ultimate goal of advancing the field of constructive neural networks.

1.3 Thesis Outline

This thesis is divided into seven chapters ranging from analyzing the current state of the literature to interpreting the evaluation results for the algorithms. First, in chapter 2, we provide relevant theoretical foundations that serve as the basis for the ideas and methods used throughout the thesis. The explored concepts include artificial neurons, neural networks, optimization algorithms, network architecture types, and network construction and pruning methods. Chapter 3 presents a comprehensive review of prior work in the field of constructive neural networks (CoNNs) and grow-and-prune neural networks (GPNNs). We investigate the most important algorithms in each area, provide a discussion of the current state of research, and list possible opportunities for improvement. The general structure and the various algorithmic ideas for each of the proposed algorithms are presented in chapter 4. Our three algorithms are: the Layerwise algorithm, the CasCor Ultra algorithm, and the Uncertainty Splitting algorithm. Next, chapter 5 provides an in-depth analysis of the procedure used to evaluate the proposed algorithms as well as several state-of-the-art algorithms. The evaluation involves multiple datasets and uses various metrics to determine algorithm performance, efficiency, and speed. Additionally, we present the software system that we developed to facilitate automatic evaluations of multiple algorithms using various datasets and hyperparameter configurations. The corresponding results for the evaluations are discussed in chapter 6. The results of each of the proposed algorithms are first analyzed individually with different sets of hyperparameters. Then, we compare the best configurations of our proposed algorithms with several state-of-the-art algorithms to be able to ascertain their relative effectiveness and determine possible improvements. Finally, chapter 7 gives a summary of the content of the thesis, provides answers to the research questions posed in this chapter, and presents an outlook on possible future research.

Chapter 2

Background

This thesis proposes novel algorithmic approaches to the area of constructive neural networks (CoNNs). In this chapter, we introduce the theoretical foundations that CoNNs are based on and explore related concepts important for this thesis. Section 2.1 incrementally builds an understanding of neural networks by starting with the smallest building block, which is the artificial neuron [2]. Next, we explore how this building block can be combined into larger networks and how these networks learn by using optimization algorithms. Finally, sections 2.2 and 2.3 provide a brief overview of the two main concepts relevant to the thesis: constructive neural networks and neural network pruning.

2.1 Neural Networks

The artificial neural network (ANN) is a computational model imitating the organization and function of biological neural networks found in living organisms such as the human brain [2]. The main building blocks of ANNs are artificial neurons and they can be connected to create various network structures and architectures. Since their introduction, ANNs have become valuable tools in machine learning. Their capabilities as universal function approximators [41] allow them to effectively model the underlying relationships between inputs and outputs in various problem domains. Thus, these models are successfully applied to a wide variety of machine learning tasks and domains. Especially approaches in the area of deep learning, which involves the use of ANNs with multiple hidden layers, have led to significant breakthroughs in the fields of computer vision, natural language processing, speech recognition, automotive computing, etc. [1]. These advancements are the reason ANNs continue to be an active area of research, with efforts on continual fine-tuning and enhancing their performance and efficiency.

This section offers an overview of the most crucial components and concepts that form the backbone of neural networks. These include artificial neurons and their various types, the possible network architectures that can be constructed, and the optimization algorithms that are used to train the networks to achieve the desired performance level. Additionally, the concept of the stochastic neural network is introduced as it is an important part of the Uncertainty Splitting algorithm proposed in the thesis (see chapter 4).

2.1.1 Artificial Neurons

Artificial neurons, which are also known as nodes or units, are elementary building blocks of ANNs. They are loosely based on the structure and functionality of biological neurons found in the human brain. To allow the creation of networks of neurons they are connected to each other using weighted links, which represent the synapses in biological neural networks. Using these weighted connections enables each artificial neuron to propagate input signals and produce an output signal. The output signal is generally calculated using a combination of the input signals and the neuron's activation function.

Biological Background

Biological neurons found in living organisms represent the inspiration for their artificial counterparts. These biological neurons function by processing information in the form of electrical and chemical signals using different types of neural systems [9]. A single neuron generally consists of a cell body (soma), an axon, dendrites, and synapses. The dendrites communicate with neighboring neurons to receive input signals, which are then combined into a single signal in the cell body. A generated action potential is sent if the processed inputs exceed a certain activation threshold. Using the axon and the synapses, this action potential signal is sent to the next neurons.

The simplified computational artificial model of biological neurons uses this general principle by combining and transforming input signals to an output signal [4]. These models attempt to replicate the core functions of their biological counterparts, without capturing the full complexity of the natural neural systems. As our understanding of biological neural networks deepens, the principles of artificial neurons may continue to evolve, leading to improved designs and capabilities [4].

Artificial Neuron Structure

Artificial neurons follow a uniform process that describes how inputs are converted into a corresponding output. Figure 2.1 shows the general structure of an artificial neuron [46]. It displays the process of how the three input signals x_1, x_2, and x_3 flow through the neuron and are transformed into the output signal designated as y. The signal processing can be read from left to right in the figure. Each of the input signals is multiplied with their associated weight value w_1, w_2, and w_3 to produce the weighted input signals. Each weight value thus determines how much the specific input value influences the output. All weighted inputs are subsequently summed together with an additional bias value. The addition of the bias value is done to offset the resulting output signal in the same way a constant term is added to a simple linear function to shift the output on the y-axis away from the origin. The sum of all signals is then fed into the transfer function Φ. This function is also known as the activation function and determines the final output that the artificial neuron will produce. The transfer function can in principle be any mathematical function, with differentiability and non-linearity being among its desirable properties [90]. Some of the most useful and widespread transfer functions are discussed in subsection 2.1.1.

We see that an artificial neuron is nothing but a combination of mathematical functions. The complete process can thus be written in mathematical notation as seen in

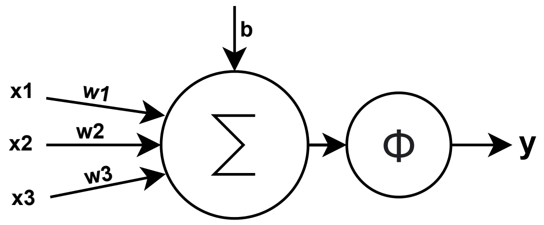


Figure 2.1: Mathematical representation of an artificial neuron [46].

equation 2.1. This equation is evaluated for each new input coming to the neuron. The parameters w_1, w_2, w_3, and b within the figure are those that can be trained and optimized to make the artificial neuron perform as needed for a specific problem. Typically an optimization algorithm is used to slowly learn weight and bias values for the neurons in a way that the inputs are transformed into the desired output.

$$y = \Phi(w_1 \times x_1 + w_2 \times x_2 + w_3 \times x_3 + b) \tag{2.1}$$

Transfer Functions

The transfer, or activation function of an artificial neuron plays a pivotal role in shaping the overall performance of a neural network. Many different transfer functions exist, however, ones that produce high-performing and easily trainable neural networks generally have the following desirable properties [90]: (1) Non-linearity, which enables neural networks to model complex, non-linear relationships between inputs and outputs. (2) Differentiability, which is essential for gradient-based optimization algorithms to allow weights and biases to be smoothly improved. (3) Computational efficiency, critical as the function will be heavily used throughout the training and inference process. The type of transfer function used can also vary depending on the network layer or problem domain [20]. For instance, some functions may be better suited for classification tasks, while others may be more appropriate for regression problems. Some common transfer functions include [80, 90]:

- **Sigmoid function:** Compresses its inputs into an output range between 0 and 1, which means it provides a smooth transition between values. This attribute makes it especially useful for modeling probabilities.
- **Rectified Linear Unit (ReLU):** Lets positive input values pass through unchanged while converting negative values to zero. This promotes sparsity and improves the vanishing gradient problem.
- **Tanh:** Similar to the Sigmoid function, the hyperbolic tangent function maps input values to an output space between -1 and 1, achieving a steeper gradient and often faster convergence times.
- **Softmax:** Normalizes the output values of a set of neurons, creating a probability distribution over a predefined set of classes. As a result of these attributes, the Softmax function is often used in the output layer of neural networks for multi-class classification problems.

2.1.2 Neural Network Architecture

Single artificial neurons can be connected to create a neural network. The structure and design of a network are often referred to as network topology or architecture [100]. These typologies affect the performance and behavior of the network and mainly vary in two ways: The number of neurons that are present in the network and the number and direction of connections between the different neurons. Architectures are generally divided into layers [2], which are collections of neurons that receive and send signals to other collections of neurons but do not influence each other. A single layer includes the incoming connections (inputs and weights) as well as the nodes and the corresponding bias value for each node. Such a layered network can be seen in figure 2.2 where the network consists of one input, one hidden, and one output layer. The number of layers in a network is considered its depth, where networks with more than one hidden layer are often referred to as deep neural networks [79]. These deeper networks have shown to be better suited for highly complex tasks where the input data can be transformed step by step in a hierarchical fashion [11]. This means that a network starts learning abstract features in the first few layers and then learns increasingly higher-level features in deeper layers. The number of input and output neurons is defined by the data and the problem that is solved [89]. The input neurons depend on the dimensionality of the input data, which means if my input is a point in three dimensions the network will have three input neurons. The number of output neurons is given by the type of problem that is being solved, for example, a three-class classification problem will likely have three output nodes with each of them giving the probability for its class as output. The number of nodes for the hidden layers can however differ greatly and will have a major impact on the performance of a neural network [106].

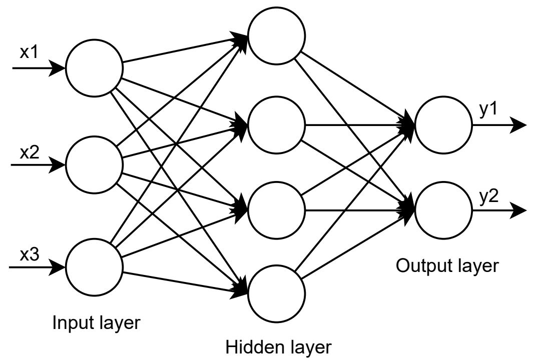


Figure 2.2: Simple feed-forward neural network with a single hidden layer.

Usually, networks tend to be fully connected (dense), which means the neurons in the layer are connected to each neuron in the previous as well as the following layer. Layers that are not fully connected are referred to as sparse layers. In a sparsely connected setup, each neuron is only connected to a subset of neurons in the previous and subsequent layers. Sparse layers shrink the size of the network in comparison to dense layers and thus reduce memory requirements and computational overhead [43]. However, they are harder to set up, as the training process has to include adding or removing the right connections for the performance to be high. Artificial neural networks can further be classified by how information travels through them [2], with the most basic type being the feed-forward network. Here, the signals travel from the inputs to the outputs with no feedback connections. Recurrent neural networks (RNNs) on the other hand have feedback, or circular, connections. This means that information can also flow backward and thus create feedback loops. This enables networks to build internal memory, which can be useful to interpret time-varying data like audio signals [13]. Using these architecture categorizations, it can be said that the example network in figure 2.2 is a dense, feed-forward neural network.

In addition to the types of network topologies discussed above, there are other variations that do not fit in the traditional neural network architectures. Among these are convolutional neural networks (CNNs) [79] and transformer networks [97]. CNNs are specialized for processing 2-dimensional data, such as images, by using convolutional layers that can learn local features and maintain spatial information. Transformer networks, on the other hand, are specifically designed to work with sequential data by utilizing a mechanism called self-attention. This allows the model to weigh the importance of different elements in a sequence and capture long-range dependencies effectively.

2.1.3 Network Training and Optimization

ANNs need a suitable architecture as well as fitting values for the weights and biases to be able to convert inputs into desired outputs [4]. Traditional neural networks select a fixed architecture before starting the training process and learn only the values for the network parameters through training. To achieve this, the training process involves continually adjusting the weights and biases in an error-correcting way until the desired performance is achieved. The specifics of the training process vary depending on the used learning paradigm.

Learning Paradigms

As with biological brains, there needs to be a type of feedback for the network, so it can recognize the quality of its behavior and adapt accordingly. The way this feedback signal is calculated and used in the network depends on the learning paradigm. The three major learning paradigms are supervised, unsupervised, and reinforcement learning, whereas supervised learning is the most prominent paradigm and also the most useful for solving real-world problems [78]. In supervised learning, the network is trained on labeled data, which means each input has a known corresponding output. These labels are used to calculate errors in the prediction of the network and train the network to minimize errors [17]. Unsupervised learning [89] deals with unlabeled data, where the network attempts to discover patterns within the input data without any guidance. Common use cases for this paradigm are clustering, dimensionality reduction, and anomaly detection. Finally, reinforcement learning involves an agent network that learns by interacting with its environment and receiving rewards or penalties for its actions [88]. Tasks that reinforcement learning is well-suited for include problems where there is an actor in an environment trying to achieve a certain task (e.g. robotics, game playing, driving autonomous vehicles, etc.). As the thesis title suggests, this work only focuses on the supervised learning paradigm, thus the following sections in this chapter also primarily explore concepts from the viewpoint of supervised learning.

Training Process

In supervised learning neural networks are trained using sets of inputs with the corresponding outputs [17]. By sending these inputs into the network and comparing the predictions of the network with the known outputs we can obtain an error signal. This error is calculated using a loss function, the exact type of which varies based on the problem domain and network architecture. The cross-entropy loss function for example is often used for classification problems, while the mean squared error loss is a popular choice to solve regression datasets [99]. The calculated error signal is then used to modify the weight and bias values in the network in a performance-improving way. This weight and bias update is performed using a learning rule, or optimization algorithm. These algorithms aim to minimize the error of the network by calculating fitting adjustments for the values of the learnable parameters. Most optimization algorithms for neural networks use the gradient of the error with respect to the weights and biases to get information about the direction and the amount of change needed in the learnable parameters [78]. Specific types of optimization algorithms are covered in the following section. As more adjustments are made, the predictions of the network gradually converges to the target output. This learning process is typically continued until either the desired performance is reached or performance no longer improves with more weight updates [17]. In summary, the training process adjusts weight and bias values by minimizing the error between the predictions of the network and the known target outputs.

Optimization Algorithms

The optimization algorithm represents the backbone in the training phase of neural networks. They are responsible for iteratively adjusting the weights and biases of the network to minimize the error between predicted outputs and actual outputs. As the error is calculated by a loss function the goal is to minimize the output of that function and thus reduce mistakes. This is why gradient-based optimization methods are frequently used, as they help discover function minima with the help of its slope [16]. One of the most commonly employed techniques is gradient descent [75], which is an iterative method that adjusts the weights by moving in the direction of the steepest decrease in the slope of the loss function. Typically, a learning rate is specified that defines the size of the steps that are taken toward the direction of the steepest decrease. Gradient descent can be enhanced by applying momentum (simulates a physical system with inertia) or adaptive rate (allows varied learning rates for each trainable parameter) methods [16]. Examples of these more advanced optimization algorithms are AdaGrad, RMSprop, and Adam. An extension to the basic gradient descent method, known as backpropagation, efficiently calculates gradients for all weights and biases in multi-layer neural networks by leveraging the chain rule for derivatives [4]. Furthermore, the Quickprop variant aims to improve the convergence speed of gradient descent by estimating the second-order derivatives of the error function [23].

In contrast to the above-mentioned methods, there are also gradient-free optimization methods that comprise of various techniques to reduce the loss function of the network without using derivatives. Such methods often draw inspiration from natural processes or heuristics and can be particularly useful for problems where gradient information is not conveniently accessible [77]. Two examples of gradient-free methods are evolutionary algorithms and simulated annealing. Evolutionary optimization uses mechanisms such as mutation, crossover, and selection to search for optimal solutions in a population of candidates [7]. Simulated annealing on the other hand generates random neighboring solutions and gradually reduces the randomness in in the process until a near-optimal solution is found [10].

Additional Learning Techniques

Alongside optimization techniques, various additional strategies can be employed to enhance the performance of neural networks and prevent overfitting. Overfitting occurs if the network learns the training data too well by also capturing noise, which negatively impacts its performance on new, unseen data [102]. Regularization can help reduce overfitting by adding a penalty term to the loss function, which discourages excessively large weights and promotes simpler models [65]. Excessive weight values can lead to overly complex models that overfit the training data, capturing not just the underlying patterns, but also the noise in the training data. Early stopping is another approach that monitors the performance of the network on a validation dataset and stops training when the performance no longer improves or begins to degrade [69]. These complementary techniques work together with optimization algorithms to fine-tune the overall performance of neural networks.

2.1.4 Stochastic Neural Networks

Stochastic neural networks are a variation of regular ANNs which introduce an element of randomness to the neural network model [95]. This integration of randomness can happen either directly in the network architecture (neurons, weights, activation functions, etc.) or somewhere throughout the training and learning process. The goal of incorporating randomness is to provide better generalization, robustness, faster convergence, handle uncertainty, and prevent overfitting. An example of the usage of randomness during the training process are dropout neural networks [102]. They randomly deactivate a fraction of neurons or connections within the network throughout the training. The goal of this method is to reduce overfitting and improve generalization by distributing the learning process over all neurons in the network.

Bayesian neural networks are arguably the most prominent example of stochastic neural networks. They incorporate uncertainty by swapping scalar values of the weights and biases for probability distributions [34]. This means that each learnable parameter can describe its uncertainty through the width of its distribution, which is often chosen to be a Gaussian distribution. A forward pass through the network is mostly the same as in traditional neural networks, except that the weight and bias values that are used for the calculations are random samples from their respective probability distributions. The final output of the network is a single value, although this value will vary with each new forward pass due to the randomness introduced in the weight and bias sampling. In contrast to traditional neural networks, the optimization algorithm for Bayesian neural networks has to find fitting values for a mean and standard deviation for each weight and bias. After meaningful distributions have been found for each weight and bias, multiple forward passes can be used to create probability distribution for each network output to reflect the uncertainty in the results. By quantifying the uncertainty associated with each prediction, Bayesian neural networks can provide more robust and reliable results compared to their deterministic counterparts [44]. Additionally, overfitting to the noise in the data is less likely to occur, because noise would instead widen the distribution for certain parameters in the network, which in turn can be observed in the output distributions. However, Bayesian neural networks typically have more trainable parameters since each weight requires two values to model a distribution, implying greater computational resources for training and inference. In addition, the training process generally requires more data to learn the correct probability distributions for the weights and biases.

2.2 Constructive Neural Networks

Constructive neural networks (CoNNs) are a class of neural network that can dynamically grow their architecture during the training process [48]. This is unlike conventional neural networks, which require a fixed, specified architecture before training can begin. CoNN algorithms generally start with a minimal network and incrementally add neurons or connections to it. This adaptable step-by-step growth process aims to create networks with the optimal size for the complexity of a specific problem. Having an appropriate number of parameters reduces the risk of overfitting compared to networks that are too large for a given problem [81]. Another significant advantage is that no architecture has to be selected beforehand, which can be a challenging process depending on the problem [42]. However, there are also some disadvantages of CoNN algorithms, such as higher computational complexity, convergence issues, and less interpretability of resulting network architectures.

CoNNs have multiple aspects throughout their growing process that can be modified to change characteristics like construction speed, final network size, performance, and efficiency. This section provides an overview of the main steps that CoNN algorithms take in the process of constructing and training networks. These steps include architecture growth strategy, network training specifics, network freezing, and the selection of a stopping criterion. The taxonomy created in [48] is used as guidance for the overview. Concrete examples of CoNN algorithms are discussed in this section only to explore the integration of certain algorithmic aspects and growing concepts. Chapter 3 goes into detail about specific CoNN approaches in the literature.

2.2.1 Growth Strategy

The selected growth strategy is the most crucial part of a constructive neural network algorithm as it determines the effectiveness of the approach. The main aspects that need to be considered for effective network construction steps are:

- **Where to extend the architecture?** Most CoNNs extend the network architecture between the last hidden layer and the output layer. This is because extending the network closer to the output layer allows the network to fine-tune the decision boundary or the output function with minimal disruption to the previously learned features. In addition, previous hidden layers are often frozen (weights and biases are not trained anymore), which means that their weights need the expected inputs to achieve their learned functions. Modifying earlier layers could lead to a loss of previously learned information and might require substantial retraining.

- **How many neurons are added?** Early approaches [22, 30] in the area of CoNN algorithms typically add only one node per growing step. However, more recent variants [61] tend to add multiple nodes at a time to improve construction speed and efficiency, as well as to capture more complex features within the added layer.
- **How are the neurons connected to the existing network?** The process of connecting new nodes can be approached in various ways. Some algorithms use the outputs from every hidden node and every input and feed them to the newly added node [22], while others more carefully construct a layered architecture and only attach the outputs from the previous layer as inputs for the added neurons [60]. The latter approach can help in maintaining a more structured and interpretable network architecture.

2.2.2 Network Training

The training process of constructive algorithms is similar to that of conventional neural networks. The weights and biases are slowly adjusted in repeated training epochs using a known set of input and output pairs. The error signal is often not calculated using one of the traditional loss functions discussed previously, instead, the covariance measure is used to train for maximum correlation of a newly added node and the output error [22, 103]. The goal of this is that the added node can then create a simple connection to the output nodes to correct this error. The specific optimization algorithms also vary for each of the different CoNN algorithms with some using modifications of the original perceptron algorithm [32] and others using types of the backpropagation algorithm [70]. Some CoNN algorithms even include multiple different training steps, where candidate neurons are trained on their own and are then integrated into the current network [22]. Another training process is then performed on the complete network to fine-tune the weights with the newly added architecture. The final important aspect of weight and bias training is the idea of weight freezing [22, 30], where parts of the network architecture stop updating their parameters. This is often done by freezing the weights of older layers or neurons, which means only the newly added parameters have to be trained. Weight freezing approaches are vastly superior in terms of training speed and efficiency in contrast to simple CoNNs which choose to retrain the complete network from scratch after each construction step [5]. However, it increases the risk of landing in local minima, because only a small subset of parameters in the network can be improved.

2.2.3 Stopping Criterion

Similar to traditional neural networks, a metric is needed to specify when training is finished. For CoNN algorithms, this stopping criterion is generally one of the following conditions [48, 67]: (1) A maximum size of the network architecture is specified and the construction process continues as long as that size is not reached. This is a simple criterion but it may be useful for training CoNNs using memory or resource-constrained devices. (2) A set time frame for the duration training process has been exceeded. This is also a simple metric but in contrast to the architecture criterion it is impacted by the underlying hardware, which is why it is rarely used. (3) The network performance reached the desired value. For this criterion, it needs to be considered from which set of data the performance metric is calculated, as using the training data could lead to overfitting. The implementations of [22] split an evaluation set from the training set which is only used to measure the stopping criterion, which is the accuracy in their case. Other possible performance estimation methods for CoNNs include cross-validation or bootstrapping [48]. (4) The performance of the network has converged (lack of improvement over a period of time). This is a good metric if the network should perform as well as possible because training only stops when there is no more improvement. However, the problem of overfitting the training set needs to be considered. Thus, a robust performance estimation is necessary. Also, a threshold and a timespan have to be defined that are used to check the amount of improvement over that timespan and stop training if necessary.

2.3 Neural Network Pruning

Neural network pruning is a method to shrink the architecture of fully trained neural networks by removing less important neurons or connections [6, 12]. The main objective of such methods is to obtain a smaller and more computationally efficient network while preserving or minimally affecting the performance of the network. This aim makes it a useful approach for reducing memory requirements, providing faster inference times, and potentially improving the generalization capabilities of the network by mitigating overfitting. Most existing pruning techniques follow a similar fundamental algorithmic process. However, a wide array of variations and adaptations of this process exist, targeting different network architectures, performance objectives, and computational constraints [96]. Additionally, pruning techniques can be paired with constructive neural networks to create grow-and-prune algorithms, which try to gain the benefits of both approaches [18, 35, 98].

2.3.1 General Pruning Algorithm

The fundamental algorithm for pruning can be summarized as follows:

Algorithm 2.1: Step in a high-level pruning algorithm [62]

```
1: assess the significance of each network parameter (weights and biases)
2: order the parameters based on their importance
3: remove one or more of the least significant parameters from the network
4: optionally fine-tune the network after architecture was reduced
5: check a defined termination criterion to evaluate if pruning should continue
```

This algorithm outlines the most crucial steps for a single iteration in a typical pruning process. The basic algorithm provides a foundation for more advanced pruning strategies, which may incorporate alternative methods for assessing neuron importance or different criteria for neuron removal.

2.3.2 Pruning Strategies

Pruning strategies can be segregated into two primary categories: structured and unstructured methods. Unstructured pruning eliminates individual parameters, thereby crafting a sparse neural network, while structured pruning considers groups of parameters like neurons, filters, or channels, and removes entire groups at once [12]. Multiple different approaches exist within these subcategories, differing in the kind and amount of architecture they discard, how they evaluate the significance of network parameters, and how they fine-tune the pruned network.

To determine which network parameters should be pruned, importance metrics assess their significance based on various criteria, such as absolute values, contributions to network activations, or gradients [6]. The amount of parameters removed at each step is also variable, with some pruning the desired weights all at once in a single step, while others prune iteratively over several steps [12]. The process of fine-tuning after the removal of parameters typically involves training the network starting with the existing weights. However, alternative approaches exist which include rewinding the network to an earlier state or reinitializing the network entirely [12]. These variations provide a wide range of options, allowing researchers and practitioners to select the best method based on their specific requirements and constraints.

2.3.3 Grow-And-Prune Algorithms

Grow-and-prune algorithms try to merge the advantages of both constructive and pruning approaches, with the aim of creating optimally sized neural network architectures and reducing redundancy [18, 19, 37, 98]. These techniques work by alternatively applying construction and pruning steps to the network throughout the training process with the goal of producing architectures that have only as many parameters as needed for a specific problem [35, 98]. Research suggests that this dynamic adaptation of the network during training yields superior performance and generalization compared to exclusively employing constructive or pruning methods [37, 40].

Despite their notable benefits, grow-and-prune algorithms also introduce added complexity to the network training process. Further, the final performance of the neural network can be affected by the choice and integration of growth and pruning algorithms. Nevertheless, these algorithms have proven successful in various applications, such as generating efficient, compact, and accurate deep neural network architectures [18, 19]. By striking a balance between expansion and reduction, grow-and-prune algorithms can create compact and efficient neural network architectures. They address the issues of over-parameterization and model redundancy associated with purely constructive methods, resulting in networks that require less memory and have faster inference times. Chapter 3 dives deeper into some state-of-the-art grow-and-prune algorithms.

Chapter 3

Related Work

This chapter presents a review of prior work in the field of constructive neural networks (CoNNs) and grow-and-prune neural networks (GPNNs). The goal is to explore the current state of research in these areas and outline gaps and limitations. This also provides a solid foundation for the algorithmic properties of the proposed algorithms in the following chapters. Section 3.1 discusses the most influential CoNN algorithms for solving supervised learning problems using feed-forward neural networks. The subsequent section 3.2 reviews a few of the most well-known GPNN algorithms and discusses the differences to purely constructive approaches. We only delve into a few well-known GPNN algorithms, because the novelty of this thesis falls primarily in the category of CoNN algorithms. The final section 3.3 provides a discussion of the explored algorithms and summarizes the current state of the research. It also points out current challenges as well as areas where unsolved problems exist and establishes how the proposed algorithms aim to solve these problems.

3.1 Constructive Neural Network Algorithms

This section provides an overview of the most established CoNN algorithms. A total of six algorithms, which we consider to be the most important ones, are reviewed in detail. A list of lesser-known approaches is briefly covered in subsection 3.1.7. Among the discussed algorithms are the Cascade Correlation (CasCor) algorithm [22], the Dynamic Node Creation (DNC) algorithm [5], CCG-DLNN [60], and ConstDeepNet [105]. The review of each algorithm is structured as follows: First, we explore the main idea of the algorithm, its historical context, and the problem domain in which the approach was developed. In addition, a step-by-step description of the algorithmic process is provided for each algorithm, and, if applicable, illustrations of the evolution in network topology throughout the growing process are presented. Any known strengths and weaknesses as well as available results on benchmark problems for the presented algorithms are discussed briefly. Finally, possible algorithm extensions and further developments are presented.

The field of constructive artificial neural networks (CoNNs) has existed for more than 30 years with the first algorithms being proposed around 1986 [12]. Since then, many papers have been published that present new algorithms or variations of existing algorithms for constructing neural networks during training. Most of the algorithms were proposed between 1985 and 2000 and interest in the field seems to have declined since then. Some of the approaches are specifically designed to solve only discrete problems. That means the output of each node in the network is either 0 or 1. This comes from early models of the perceptron [74] having a step function, or threshold function, as its activation function. Therefore, one can categorize constructive algorithms into discrete and continuous [52]. Of the six algorithms presented in detail, only two are defined as a discrete approach (see subsections 3.1.3 and 3.1.4), while the remaining methods are continuous. This imbalance is because having continuous outputs allows the algorithms to be applied in far more problem domains and thus they tend to be used more often.

3.1.1 Cascade Correlation

The cascade correlation algorithm was introduced by Fahlman and Lebiere in 1989 [22] and is the most well-known constructive network approach [52]. The main idea behind the Cascade Correlation algorithm is to start with a fully connected input and output layer and add new hidden nodes one by one. Each new hidden node will be connected to all previous nodes in the network except the output nodes and additionally receive a trainable bias parameter. The weights of the hidden nodes are optimized to maximize the covariance between its output and the residual error of the network. After a new hidden node is added to the network, its weights and biases are frozen. By freezing the weights of the added hidden nodes, the algorithm only ever needs to train the final layer, reducing the moving target problem and removing the vanishing gradient problem [22]. The subsequently trained connections between the hidden node and the output nodes are then able to easily cancel out this correlated error with one connection. The step-by-step procedure for the original CasCor algorithm is shown in algorithm 3.1.

Algorithm 3.1: Cascade correlation algorithm [22]

```
1: start with inputs and outputs fully connected
2: train connection weights until convergence using Quickprop
3: while !$isGoodEnough$ and !$hasConverged$ do
4:     generate hidden node candidates
5:     candidate nodes get inputs from all input and hidden nodes
6:     while !$candidateNodesConverged$ do
7:         train candidates to maximize covariance with residual error using Quickprop
8:     add the best candidate to the network
9:     freeze candidate input weights
10:    train non-frozen weights until convergence
```

The optimization algorithm used for updating the weights and biases was also developed by Scott Fahlman and is known as Quickprop [58]. It is a modification of the backpropagation algorithm [76] and uses a quasi-Newton method to speed up the learning process. The loss function used for adding new hidden nodes is the covariance loss and it is calculated using the following equation: $S = \sum_o |\sum_p (o_{h,p} - \bar{o}_h)(E_{p,o} - \overline{E}_o)|$ where S_j is the covariance, o are outputs, E is the output error, p is an input pattern, and h denotes the index of the hidden node.

CasCor generates a network architecture with single-node hidden layers where each successive node increases in its number of input connections. Figure 3.1 shows the construction process of the algorithm starting with only the input and output layer, then with a single hidden layer, and finally with two hidden layers. The dashed lines indicate that these connections are frozen and no longer undergo optimization.

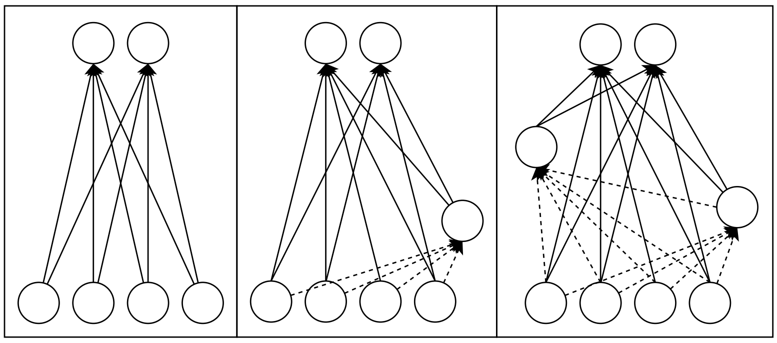


Figure 3.1: Network construction of the CasCor algorithm [22].

The CasCor algorithm has been tested on various benchmark problems including the 2-spirals problem [22] where it has shown good performance. Its training process tends to be faster than other constructive approaches and the moving target problem and vanishing gradient problem are eliminated due to freezing of earlier layers [22]. However, the performance can be sensitive to the selected activation function and the emerging non-linearities often are too strong for the problem at hand [70].

Since its first introduction, many variations of the CasCor algorithm have been proposed that slightly manipulate or modify specific aspects of the algorithm to improve convergence speed, network size, final performance, etc. [21, 70, 84, 94]. An extension devised by the original author is the recurrent cascade correlation algorithm [21], which creates a recurrent neural network and can be used for problems with sequential data such as text interpretation. Lutz Prechelt investigated a total of six cascade correlation alternatives aiming to address the problems mentioned above [70]. In this investigation, the cascade 2 algorithm was devised to fix the problem of the covariance error function [70], which lets hidden nodes minimize the residual error directly, instead of maximizing the covariance. Another variant was proposed in [84], which grows the network horizontally instead of vertically. This means that each new node is added to a single hidden layer, instead of creating a new hidden layer. Treadgold and Gedeon [94] developed the Casper algorithm with the aim of improving the regression performance of the cascade correlation algorithm. Instead of weight freezing and using the covariance error function, Casper employs variations of resilient backpropagation (RPROP) [73] to train the whole network.

3.1.2 Dynamic Node Creation

Dynamic node creation (DNC) [5] is a constructive neural network algorithm developed by Ash Timur in 1989. The main idea behind the DNC is to add nodes one by one to a single hidden layer feed-forward network and optimize them using standard backpropagation. The hidden nodes generally have sigmoid activation functions. A new node is added to the hidden layer if the current network is not able to achieve the mapping between the input and output space correctly. This means that training continues if the error is too high or it is not decreasing for a certain amount of time. The algorithmic process of the DNC algorithm is shown in algorithm 3.2 and an example of the resulting network topology can be seen in figure 3.2. The illustration shows how a new node is added to the single hidden layer network.

Algorithm 3.2: Dynamic node creation algorithm [5]

```
1: start with inputs, one hidden node, and outputs
2: each layer is fully connected (inputs → hidden layer → outputs)
3: train all weights until convergence with backpropagation
4: while $!isGoodEnough$ and $!hasConverged$ do
5:     retrain all weights until convergence with backpropagation
6:     add new hidden node and fully connect it to input and output layer
```

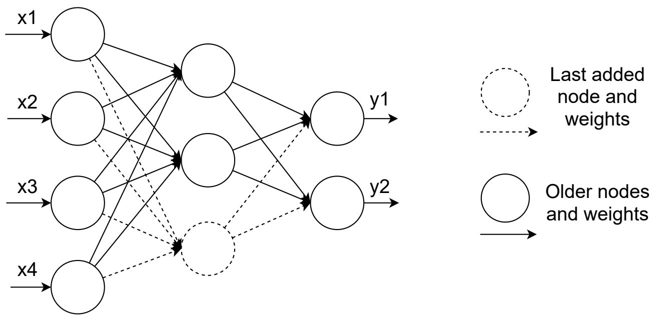


Figure 3.2: Network structure of the DNC algorithm [5].

The DNC algorithm was evaluated on various simple benchmark problems, such as binary addition and the encoder problem, where the algorithm was able to solve the problem while finding minimal network topologies [5]. However, we are not aware of any evaluations that have been done on more complex problems as well as any comparisons to other CoNN algorithms. The primary strength of the DNC algorithm is its simplicity and ease of implementation. This straightforward approach is also its main disadvantage. Instead of freezing weights or memorizing certain parameters, DNC retrains the whole network after a new node is added. This means that, if the algorithm is applied to problems that need large network architectures, a high amount of computational resources are needed. Additionally, having only one hidden layer might lead to worse generalization and more overfitting for certain problems [93].

3.1.3 Tower Algorithm and Pyramid Algorithm

The tower algorithm and the pyramid algorithm were proposed by Gallant in 1986 [32], making them the earliest in the field of CoNN algorithms. Both approaches belong to the discrete set of constructive algorithms and create networks by continuously adding single-node hidden layers to the network. Due to their discrete nature, these two algorithms are only applicable to solve classification problems. The two approaches only differ slightly, which is why they are covered as a collective in this subsection. The specific construction process can be seen in algorithm 3.3. Note that the only difference between the tower algorithm and the pyramid algorithm stems from how the newly added nodes are connected to the other nodes in the network. Both algorithms use a variant of the simple Perceptron learning algorithm [31] for the weight optimization process. This variant is called the pocket with ratchet modification (PRM) algorithm [33] and it works by caching the best set of weights during training to not allow weight updates unless a better set is found.

Algorithm 3.3: Tower/pyramid algorithm [32]

```
1: start with only inputs and output node connected
2: train current network weights until convergence using PRM
3: while !$isGoodEnough$ and !$exceedsMaxIterations$ do
4:     freeze all active weights
5:     create a new output node
6:     Tower: connect it to the inputs and the previous output node
7:     Pyramid: connect it to the inputs and all hidden nodes
8:     train non-frozen network weights until convergence using PRM
```

In terms of performance, both the tower and pyramid algorithms were tested on various two-class classification problems including parity-5, rain prediction, and financial forecasting, where they produced near-optimal solutions and showed good generalization on the testing data [32]. Figure 3.3 show the emerging network typologies for both algorithms. We can see that the evolution of the architecture dictates a single output node which limits the possible uses of a neural network generated using these algorithms. This is the reason why these algorithms were originally designed only for two-class classification problems. However, multi-class variants with real-valued input values have been proposed for both algorithms [66].

3.1.4 C-Mantec

The competitive majority network trained by error correction (C-Mantec) algorithm [86] was proposed by Subirats, Franco, and Jerez in 2008 to solve various classification problems. The novelty of this approach lies in the use of competition between the hidden neurons to allow them to learn throughout the complete construction process. The algorithm also tries to minimize overfitting of the training data by focusing on sets of inputs from the training set that have been misclassified a high number of times. The basic approach of C-Mantec is to create a single-hidden-layer network of binary neurons (i.e. neurons have the threshold function as activation function). The algorithm is versatile in its usage as it allows for one or multiple output nodes. The optimization process is performed by the thermal perceptron learning rule [29], which is an extension of the perceptron learning algorithm [31]. It distinguishes itself by adding a temperature parameter (inspired by simulated annealing [77]) which controls the amount of change to the weight values. The temperature is high initially to allow for the exploration of a wider

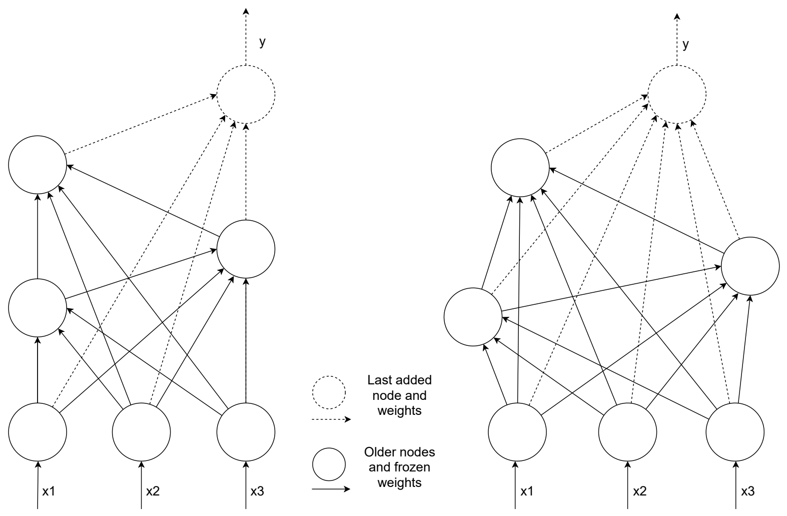


Figure 3.3: Emerging topology of tower vs pyramid algorithm [32].

solution landscape and cools down in later training stages to converge towards an error minimum. The basic algorithm shown in 3.4 relies on the aforementioned temperature factor as well as a growth factor, which specifies the threshold at which new neurons are created.

Algorithm 3.4: C-Mantec algorithm overview [86]

```
1:  start with inputs, outputs, and one hidden node
2:  connect layers (inputs → hidden layer → outputs)
3:  while !$allPatternsClassifiedCorrectly$ do
4:      perform inference with randomly selected input pattern
5:      if $patternMisclassified$ then
6:          calculate neuron with highest $temperatureFactor$
7:          if $temperatureFactor > growthFactor$ then
8:              modify weights of this neuron with thermal perceptron rule
9:              lower temperature value of this neuron
10:         else
11:             add new neuron to the hidden layer
12:             reset temperature values of all neurons
13:             train the new neuron with the misclassified pattern
```

The authors evaluated C-Mantec on multiple classification problems (3-circles, ionosphere structure, image segmentation, etc.) where it showed generalization capabilities comparable to other state-of-the-art classification algorithms while creating especially compact network topologies [86]. The idea of using competition between neurons is to allow specific neurons to focus solely on capturing certain input patterns, instead of sharing the learning of features over multiple neurons. This approach will often lead to very minimal network architectures [86]. However, the design of the stopping criterion and the focus on misclassified patterns dictate the problems to be perfectly solvable, which is unlikely to be the case using real-world data. Nevertheless, the idea is still interesting, and with certain adjustments, the algorithm could be used for more complex problems. The original design of the algorithm focused on two-class classification only. However, in 2010, the same authors investigated three possible extensions, One-Against-All (OAA), One-Against-One (OAO), and P-against-Q (PAQ), to make multi-class classification possible [87].

3.1.5 CCG-DLNN Algorithm

The cascade-correlation growing deep learning neural network algorithm (CCG-DLNN) [60] is a recent variation of Fahlman's original CasCor algorithm [22]. The main novelty of CCG-DLNN is that it creates networks with multiple layers and multiple nodes per layer, instead of only having single-node hidden layers. Furthermore, the inputs of newly added hidden nodes come only from the last hidden layer as opposed to CasCor, where every node in the network is used as input for new nodes. An overview of the steps of CCG-DLNN can be seen in algorithm 3.5. Mohamed et al. use the weight-constrained neural network training algorithm (WC-NNTA) [55] to optimize the weights in the network. This is a variation on the limited-memory Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS) [54], which is a quasi-newton optimization method that tries to reduce memory requirements.

Algorithm 3.5: CCG-DLNN algorithm [60]

```
1:  start with only inputs and outputs fully connected
2:  train current network with backpropagation
3:  while !$isGoodEnough$ and !$maxIterationsReached$ do
4:      if $reachedMaxNodeAmountPerLayer$ then
5:          candidate ← new node in a new hidden layer
6:      else
7:          candidate ← new node in this hidden layer
8:      connect added candidate with all nodes from the previous layer
9:      train candidate by maximizing the correlation between its output and the resid-
        ual network error
10:     connect trained candidate to outputs and freeze its input weights
11:     train non-frozen weights with WC-NNTA [55]
```

By implementing the aforementioned modifications to the original CasCor algorithm, the authors aim to address the problem of the ever-increasing amount of input connections for newly added nodes [70]. CCG-DLNN limits the number of input connections for a new node to the maximum of the number of nodes in the previous hidden layer [60]. Additionally, these changes allow for more stable non-linearities in the network in contrast to the single-node hidden layers. In figure 3.4 we can see a snapshot of the CCG-DLNN construction process where the second hidden layer is currently being built. Each hidden layer develops to a width defined by the maximum-node-amount hyperparameter. The next node is shown using dashed lines and is trained using the CasCor correlation loss and thus not yet connected to the outputs. After the training it will be connected and the output weights are trained.

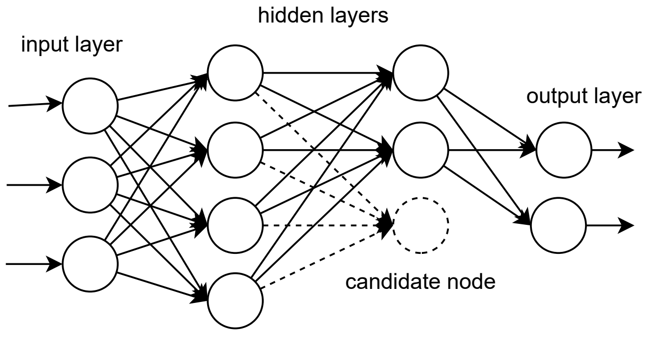


Figure 3.4: Construction process of the CCG-DLNN algorithm showing the addition of a new node to an existing hidden layer [60].

CCG-DLNN was evaluated on a total of 9 benchmark problems including the 2-spirals problem, Diabetes 130-US, Breast Cancer Wisconsin, and Heart Disease [60]. In comparison to the original CasCor algorithm, CCG-DLNN obtained higher accuracies for all evaluated data sets and additionally showed a faster decrease in output errors per construction step. The authors mention that reducing the runtime of the CCG-DLNN algorithm is a goal for future improvements and extensions. So far, however, there have been no further developments of the algorithm.

3.1.6 ConstDeepNet Algorithm

In 2018, Zemouri et al. [105] proposed the ConstDeepNet algorithm for breast cancer computer-aided diagnosis (BC-CAD). This algorithm was developed with the aim of predicting the recurrence score of the Oncotype DX (ODX) assay, which is a widely-used Gene Expression Profiling (GEP) test for breast cancer patients. The core principle of this algorithm is to build a multi-layer neural network one node at a time, in which each layer is fully connected with its predecessor and its successor layers. Even though this approach was designed to solve the problem of predicting a breast cancer identifier, it is general enough to be used in various other domains that require accurate and efficient classification models.

The step-by-step process of ConstDeepNet is specified in algorithm 3.6. The ConstDeepNet algorithm starts with a network with a single hidden layer with one node in it. During the training process, new neurons and hidden layers are added incrementally, guided by the user-defined parameters $MaxHL$ and $MaxN$. The weights of only the current hidden layer are updated during each training step, while the weights of the other layers remain unchanged. The algorithm finishes when a convergence condition is met or when the number of hidden layers exceeds the user-defined maximum limit. The specific optimization algorithm used for updating the weights and biases was not mentioned in paper [105].

The algorithm has many similarities with the CCG-DLNN algorithm (see subsection 3.5) as both algorithms build hidden layers node by node until a maximum size is reached. They only differ in that CCG-DLNN uses the covariance measure to pre-train a candidate node, while ConstDeepNet simply connects a random new node and trains

Algorithm 3.6: ConstDeepNet algorithm [105]

```
1:  start with a single hidden layer with one neuron fully connected
2:  train current network with backpropagation
3:  while !$hasConverged$ and !$maxNetworkSizeReached$ do
4:      if $reachedMaxNodesForLayer$ then
5:          freeze weights of the old layer
6:          add a new layer with one node and randomize its weights
7:      else
8:          add a new node to layer and randomize its weights
9:      train unfrozen weights to convergence
10: fine-tune the last layer weights for $N$ iterations
```

it jointly with the whole network. However, the emerging network topology, as shown in figure 3.4, will be similar.

A main advantage of ConstDeepNet [105] is its simple growing strategy and the use of only one optimization algorithm. In comparison to CCG-DLNN [60], no pre-training is needed for newly added nodes, improving algorithm clarity and removing training overhead. Additionally, freezing all but the final two layers will keep the construction effort minimal while still providing the hierarchical and feature-combining advantages of a multi-layer network. Similar to CCG-DLNN [60], the problem of building a layer one by one is that it is a slow process, and adding multiple nodes at a time might be beneficial. This is a trade-off between construction speed and final network size. Another aspect to keep in mind is that the performance will be highly dependent on the selected hyperparameter for the maximum number of nodes per layer, as it is the deciding factor for the final network architecture.

In their evaluations, Zemouri et al. [105] showed that the ConstDeepNet algorithm achieves good performance on the breast cancer diagnosis dataset. According to the authors, the deep constructive nature of the algorithm allows for the efficient learning of complex patterns within the breast cancer data. In comparison to standard approaches for breast cancer diagnosis, the ConstDeepNet algorithm shows improved performance, enhancing the prediction accuracy of recurrence scores [105]. Even though the algorithm outperformed the tested classification approaches, a comparison with other CoNN approaches was not performed and would shed more light on the relative performance of ConstDeepNet in the broader context of CoNN algorithms.

3.1.7 Additional CoNN Algorithms

In addition to the six well-known approaches discussed above, this subsection gives an overview of five further CoNN algorithms. Due to space restrictions, the following algorithms are presented in a less detailed fashion. Although less well known, each of these algorithms uses interesting ideas that contribute unique perspectives and innovative strategies to the field of CoNNs and therefore they are reviewed here.

Upstart Algorithm

The Upstart algorithm [30] was developed in 1990 by Marcus Frean and works by adding discrete neurons to the network in a binary tree fashion. The main idea is to add a new node to the network if some input pattern is wrongly classified. For example, if an output node is wrongly turned off, a new child node is added between it and the last hidden node. The algorithm starts at the output node and recursively adds nodes to correct these errors, resulting in an unbalanced binary tree of nodes. This concept is not adaptable to continuous neurons, as it relies on the On and Off concepts to regulate node growth. Each training step of the Upstart algorithm uses the Pocket algorithm [33] to optimize the weights in the network.

Tiling Algorithm

The Tiling algorithm [59] is a discrete classification algorithm proposed by Mezard and Nadal in 1989. This approach grows a networks layer by layer with each layer having one main unit and zero or more ancillary units. Each layer is fully connected to its previous and the following layer. The addition of ancillary units relies on the faithfulness criteria, which considers a class faithful if every input vector of the class has the same intermediate representation (i.e. the output of the hidden layer). If unfaithful classes exist at a layer, ancillary units are added to that layer one by one. If a maximum layer size is reached, a new layer with a new main node is added to the network. Variations of the Tiling algorithm have been proposed for multi-class classification [66].

Restricted Coulomb Energy Network

The Restricted Coulomb Energy Network (RCE) [72] is a classifier network using hyperspheres. It is a type of radial basis function (RBF) network [91], where hidden neurons have radial basis activation functions. Each hidden neuron consists of a hypersphere center of the size of the input space and a radius, covering a local area in the input space. Inputs are forwarded to the hidden nodes to calculate the Euclidean distance to each hypersphere center. A hidden neuron is activated if the input vector distance is smaller than the neuron's hypersphere radius. If no neurons are activated, a new hidden neuron is generated with the input vector as its hypersphere center and a fixed radius.

Meiosis Networks

The Meiosis Network proposed by Hansen [36] in 1989 is a node-splitting algorithm where a new hidden node gets split off existing nodes if certain criteria are met. Single-valued weights are exchanged with probability distributions, meaning each weight has a distribution that is sampled each time the weight is needed. The weight optimization process finds fitting mean and variance values for each weight. If a weight has too much uncertainty in its distribution, a new node is created to support it. This strategy helps the algorithm to specifically target and improve weaknesses in the network.

BINCOR Algorithm

The BINCOR algorithm [83] is similar to the Upstart algorithm, correcting wrongly on or off errors by adding units between inputs and outputs. Instead of growing in a binary tree fashion, BINCOR grows horizontally, where new units are added to the same layer to correct errors, or vertically, where unit pairs create a new hidden layer. This approach aims to combine the strengths of the tiling, tower, and upstart algorithms [83].

Further Algorithms

Several other algorithms exist that we are unable to cover in detail due to space constraints in this chapter. These lesser-known approaches include the Oil-spot algorithm [28], PTI [3], DistAI [101], the Perceptron Cascade algorithm [14], and the Sequential Algorithm [56].

3.1.8 Summary

This section provided a detailed overview of the most important CoNN algorithms found in the literature. The bulk of the research in the field of CoNN algorithms has been published from 2010–2020. The CasCor [22] algorithm, as well as other approaches such as the DNC [5] algorithm and the Tower and Pyramid algorithms [32], emerged at the start of this period and they are still well-known methods today. The proposal of new CoNN approaches has declined since then. The few novel algorithms of recent years such as CCG-DLNN [60] and ConstDeepNet [105] mainly make slight improvements to ideas of earlier algorithms. C-Mantec [86] being the exception as it combined many novel strategies into a successful CoNN algorithm.

The reviewed methods employ various diverse strategies for the construction and training of neural networks. A common feature observed in many algorithms was the technique of weight freezing, where the optimization of specific network parameters is stopped to improve construction efficiency [22, 32, 60, 105]. Next, many approaches employ the correlation metric for training candidate nodes copying of the original CasCor algorithm [22, 60, 70]. This allows the weights of nodes to be trained without needing to consider the whole network. However, other research showed that the correlation measure often larger non-linearities in the network than is needed [70]. Other interesting ideas shown in the reviewed algorithms were the use of neuron competition to allow each neuron to focus on learning a specific input pattern [86] and the integration of probabilistic weights and using high uncertainty to find weaknesses in the network [36].

3.2 Grow-And-Prune Neural Network Algorithms

Grow-and-Prune Neural Network (GPNN) algorithms have emerged in recent years as a promising research field, combining both growth and pruning strategies to build neural networks. The general approach behind GPNN algorithms is to apply growing and pruning steps throughout training. A simple strategy might alternatively apply a construction step and then a pruning step until the network reaches the desired performance level. By incorporating pruning into CoNN algorithms, many approaches aim to address issues such as overfitting and redundant network architecture. These problems often arise due to the greedy construction process employed by most constructive algorithms [103]. Although they are not yet as extensively explored as CoNN algorithms, GPNNs have shown great potential by creating high-performing neural networks of small sizes [19, 92, 103] for various applications.

This thesis primarily focuses on constructive neural networks, however, the structure and algorithmic idea of GPNN algorithms are essential to how our proposed algorithms fit into the broader field of GPNNs. This section presents a brief overview of three notable GPNN algorithms, which are the CC-ODB algorithm [92], the GP-DLNN algorithm [103], and the NEST algorithm [19]. These have been selected because of their popularity and performance. Beyond them, other approaches exist and are continuously developed [18, 37, 38, 63]. Each review explores the key idea of the approach, the algorithmic procedure, the results of benchmark problems, and the strengths and limitations.

3.2.1 CC-ODB Algorithm

The CC-ODB algorithm was devised in 2003 by Rivest et al. [92] and combines existing construction and pruning approaches to create a GPNN algorithm. To our knowledge, this approach is the first time growing and pruning algorithms have been used in combination with each other to construct neural networks. The two existing methods used in the algorithm are the well-known CoNN algorithm Cascade Correlation (CC) algorithm [22] in the growing stage and the popular Optimal Brain Damage (ODB) [49] algorithm in the pruning stage. A detailed review of the Cascade Correlation algorithm can be found in subsection 3.1.1. The ODB pruning algorithm determines and removes the least important weights in the network based on their impact on the loss function. Calculating this impact value for each weight involves computing second-order derivatives of the loss function with respect to the weights, which can be used to estimate their effect on the output if they were to be removed. The pruning stage is split into two phases: Input pruning, which is used to prune weights of the added candidate nodes, and output pruning, which prunes weights of the output layer. As ODB uses the loss function to determine the least significant weights, the input pruning phase needs to use the covariance loss, and the output pruning uses the problem-specific error function (e.g. the cross-entropy loss). The full step-by-step process is shown in algorithm 3.7.

By building on top of two established algorithms (CC and ODB), the CC-ODB algorithm effectively combines their strengths and addresses some of their limitations [92]. Specifically, the overfitting problem of the CC algorithm is addressed by incorporating pruning, leading to more compact networks with more generalization capabilities. The CC-ODB algorithm was tested on two datasets of the PROBEN1 benchmarks (Glass and Diabetes) [71] using four different configurations, which range from using CC with no pruning to applying pruning for all input phases and the final output phase. The results of the evaluations revealed that network architecture sizes can be reduced by almost 50% by applying ODB pruning strategies to the cascade correlation algorithm. Additionally, the learning speed and the generalization capabilities were improved in comparison to the purely constructive algorithm [92].

Algorithm 3.7: CC-ODB algorithm [92]

```
1: start with inputs and outputs fully connected
2: train connection weights until convergence
3: while !$isGoodEnough$ and !$hasConverged$ do
4:     generate hidden node candidates
5:     candidate nodes get inputs from all input and hidden nodes
6:     while !$candidateNodesConverged$ do
7:         train candidates to maximize covariance with residual error
8:     add the best candidate to the network
9:     prune candidate weights using ODB
10:    freeze candidate input weights
11:    train non-frozen weights until convergence
12: prune output weights using ODB
```

3.2.2 GP-DLNN Algorithm

The GP-DLNN algorithm was proposed in 2019 by Zemouri et al. [103] as a tool for dynamically finding small and well-performing neural networks for solving specific problems. The main idea of the algorithm is to first generate an oversized neural network using an iterative construction approach and then trim this network using two different pruning methods. For the growing part of the algorithm, Zemouri et al. use the Deep Constructive Neural Network Algorithm with local search [104]. It works by adding new nodes to the current hidden layer as long as it results in a sufficient decrease in the network error. If adding nodes only provides little benefit, a new hidden layer with one node is added to the network. In the pruning process, the algorithm applies Iterative-Pruning (IP) as well as the Statistical Stepwise Method (SSM) consecutively [25, 26]. The IP algorithm eliminates unnecessary nodes from the network and tunes the weights so the impact on performance is minimized while SSM is based on statistical techniques and tries to determine and remove non-significant weights in the network. An overview of the procedure for the GP-DLNN algorithm is shown in algorithm 3.8. Initially, the network consists of only inputs and outputs with a single hidden layer containing one node. The growing step repeatedly adds new nodes one-by-one and retrains the last two layers until a target performance is reached. For the optimization of the weights, GP-DLNN uses the stochastic online backpropagation algorithm. After the intermediate network is constructed the IP and SSM are applied in succession with a training step in between to fine-tune the network weights.

Applying two pruning algorithms allows GP-DLNN to shrink the network architecture considerably in comparison the using purely constructive approaches [103]. However, using three additional algorithms repeatedly requires lots of computational resources which might be undesirable depending on the use case. GP-DLNN was evaluated on various toy benchmark datasets such as the 2-spirals problem, as well as multiple real-world medical datasets (e.g. Breast Cancer Wisconsin). The results on these problems were compared with evolutionary metaheuristics and traditional neural networks of fixed size. GP-DLNN outperformed all other tested algorithms and managed to consistently generate smaller final network sizes compared to the traditional neural network [103].

Algorithm 3.8: GP-DLNN algorithm [103]

```
1: start with inputs, outputs, and a single hidden layer with one neuron fully connected
2: while !$targetPerformanceReached$ do
3:     apply the Deep Constructive Algorithm growing step
4:     train non-frozen weights with backpropagation
5: while !$performanceDeteriorated$ do
6:     apply the Iterative-Pruning (IP) algorithm step
7: fine-tune new network with backpropagation
8: while !$performanceDeteriorated$ do
9:     apply the Statistical Stepwise Method (SSM)) pruning algorithm step
10: fine-tune new network with backpropagation
```

3.2.3 NEST Algorithm

NEST, or neural network synthesis tool, is a GPNN algorithm developed by Dai et al. in 2018 [19]. In contrast to many other methods in this field, it also allows for the growth of convolutional layers in the network. However, as this is not the focus of our thesis, we do not explore this variant further in this review. Also unconventional for such approaches, NEST does not start with a minimal network, but with a randomly initialized sparse seed architecture where some connections will be turned off. This starting network is then modified using gradient-based neuron and connection growth and magnitude-based pruning. The growth or activation of connections is done by finding the connection that, when activated, reduces the loss the most. This can be calculated using the derivative of the loss function with respect to the weights. Growing new neurons consists of selecting a layer l and finding two highly correlated nodes in layers $l-1$ and $l+1$. A new node is then generated in l and connected to the nodes in $l-1$ and $l+1$ with weights initialized such that the loss is minimized. The pruning approach used for the weight values is simple magnitude pruning, which considers small weights as insignificant to the output and removes them. The overview of the complete NEST procedure is illustrated in figure 3.5.

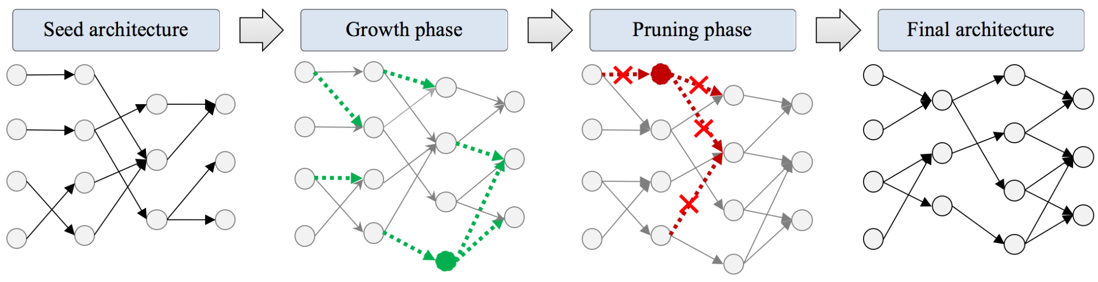


Figure 3.5: Overview of the NEST algorithm procedure [19].

A limitation of NEST is the need for a seed network, as this goes against one of the main principles of CoNN algorithms, which aims to solve the problem of finding a fitting network architecture upfront. This drawback is especially notable as the depth of the network can never be adjusted, making the selection of a good seed architecture very crucial [38]. NEST was tested on a variety of problems, including the MNIST[1] dataset. A wide range of seed architectures, such as LeNet-5 [50] or AlexNet [47], were used for evaluation, where network parameters were reduced by up to 99% for LeNet and up to 94% with no loss in accuracy. Although this algorithm is not able to produce a neural network from scratch, it is still considered to be a GPNN algorithm, and its integrated construction and pruning steps are useful guiding principles for other algorithms in this field.

3.2.4 Summary

Although the field of GPNN algorithms is not as extensively explored as purely constructive algorithms, the available methods [18, 19, 37, 38, 63, 92, 103] demonstrate the potential benefits in various applications. CC-ODB is a direct extension of the cascade correlation algorithm (see subsection 3.1) that integrates the optimal brain damage pruning algorithm in various stages. GP-DLNN creates oversized networks by adding nodes one by one and then uses Iterative-Pruning and the Statistical Stepwise Method successively to trim the network into a compact representation. The NEST algorithm modifies an initial seed network with gradient-based growth and magnitude-based pruning to generate small, well-performing network architectures.

Overall, GPNN solutions show high potential in creating compact networks with high performance and good generalization capabilities. Furthermore, they address some limitations of pure CoNN algorithms, such as overfitting due to the greedy addition of new nodes and the creation of redundant nodes and connections [103]. The presented algorithms were tested on various datasets and generally displayed higher final performance while creating smaller networks when compared to purely constructive approaches [92]. However, by adding additional pruning steps into the construction process, GPNNs generally require more computational resources during training which may be undesirable depending on the specific situation.

3.3 Discussion

Both CoNN and GPNN algorithms have proven their ability to be an effective solution to the problem of architecture selection. CoNNs build neural networks by alternating between the growth of new nodes or connections and the optimization of the weights and biases in the network. GPNNs extend this concept by integrating pruning methods to shrink architecture size and improve generalization capabilities. This means CoNN algorithms are a crucial building block for GPNNs as they provide many of the techniques for the growth steps in these approaches.

The algorithms employ different concepts and methods to achieve the construction of small and high-performing neural networks. The weight-freezing technique is crucial to avoid wasting resources in the training process, but it often leads to overfitting and non-optimal local minima. Thus, this area could be an opportunity for improvement. Most construction steps only grow the network by adding a single node at each step. This can lead to long training times and a waste of resources when applying the algorithms to harder problems. Applying larger and adaptable growth steps may lead to faster and

[1]http://yann.lecun.com/exdb/mnist/

more efficient network construction. Most algorithms add nodes to the network at a fixed location, targeting no specific weaknesses. Another aim should hence be to allow injecting of new architecture into the network where it is most useful. The integration of a pruning method is a good strategy to reduce network size and improve generalization. However, pruning approaches that are too complex will slow down training time. This means that using a lightweight pruning approach is often preferable. Combining these insights, we conclude that our goals for new algorithms in this space are:

1. Having a correction technique to avoid overfitting due to weight freezing.
2. Including an adaptive growing step allowing for the addition of multiple nodes and connections at a time.
3. Specifically targeting weak points in the network when adding new architecture.
4. Integrating a lightweight pruning step to capture GPNN advantages.

Each of the existing algorithms utilizes a variety of strategies and innovative concepts to construct neural networks throughout the training process. These techniques, as well as their strength and weaknesses, have been considered in the approach for this thesis. While improvements to older methods continue to be proposed, entirely new strategies are less common. The research for GPNN algorithms has shown more progress in recent years, but knowledge and widespread adoption of these methods within the machine learning community remains relatively limited. This presents an opportunity for this work to introduce novel ideas and methodologies into the field. Promoting CoNN and GPNN algorithms as an alternative to traditional neural networks would provide machine learning approaches that are more accessible to inexperienced users as less trial-and-error is needed for the setup. The advancement of these research areas is one of the main goals of this thesis which we plan to achieve by developing three new CoNN approaches that also include GPNN elements.

Chapter 4

Approach

In this chapter, we propose three novel constructive neural network (CoNN) algorithms with the aim to improve current algorithms in terms of performance, efficiency, and network complexity. Each proposed algorithm incorporates a simple pruning technique making them a type Grow-and-Prune algorithm (GPNN). This chapter explains how the proposed algorithms work and discusses each of their distinctive features. Chapter 3 conducted an analysis of the current state-of-the-art CoNN and GPNN algorithms and identified their strength and weaknesses. Furthermore, potential opportunities for improvements and a list of possible features for new algorithms are devised. The ascertained features are included in one or more of the proposed algorithms featured in this chapter. In addition to explaining the structure of the algorithms, the integration into the larger evaluation system developed for this thesis is also analyzed.

The chapter is divided into two parts: Section 4.1 explores the common framework and principles between all algorithms. Then, the distinctive aspects and the structure of each of the proposed algorithms are discussed in section 4.2. Each algorithm is analyzed thoroughly including step-by-step algorithmic procedures and graphical representations. Finally, a short summary discusses comparisons between the algorithms, possible strengths, and challenges, and an outlook for the next chapters.

4.1 General Algorithm Structure

Most CoNN and GPNN algorithms are using a similar underlying structure and only vary in the way new architecture is added or how existing parameters are optimized. This unified process is a construction loop, where various steps are taken at each iteration to ensure the growth of a well-performing network. This section explores this common framework and provide an understanding of the key components that are integral to the functioning of the specific algorithms. The following subsections first give a step-by-step overview of this general process and then explore each of the steps in detail.

4.1.1 Common Construction Process

Each of our proposed algorithms follows a unified process, of which each step can have variations depending on the algorithm. Figure 4.1 gives an overview of this process and shows how CoNN algorithms construct a final network after starting with an initial network. The main part of the process is the construction loop, which continually applies incremental changes to the network architecture and then evaluates performance to check if the constructed network has achieved certain performance targets. This loop can optionally be extended by a pruning step where network parameters (i.e. nodes and connections) are removed to shrink network size and reduce computational costs. Although not shown in the figure, the constructed algorithm is evaluated one final time on the held-back test data after the loop is terminated to receive a final performance assessment. In addition to providing a clear overview, the unified framework for all algorithms also assists with algorithm implementation, because it allows common segments to be extracted from the specific algorithms and simplifies the evaluation of multiple algorithms. Each step of the construction process is discussed in detail in the next subsections.

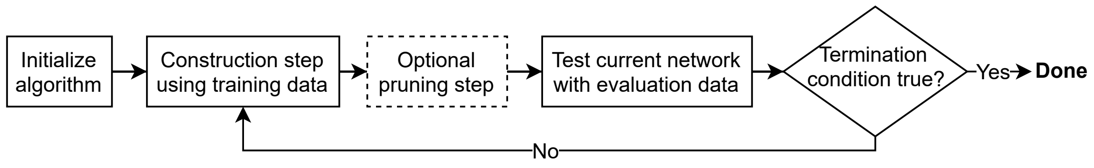

Initialize algorithm
Construction step using training data
Optional pruning step
Test current network with evaluation data
Termination condition true?
Yes
Done
No

Figure 4.1: The basic CoNN algorithm framework.

4.1.2 Algorithm Initialization

The initialization step includes selecting the used dataset, shuffling the data, and splitting the data into training, evaluation, and test partitions. Then, a minimal starting network (i.e. a network with the correct number of inputs and outputs) is set up. This initial network is generally fully connected with each input node connecting to each output node. The weight and bias values for the network are set randomly. However, the range of values to pick from, as well as the distribution used for sampling, can vary for each specific algorithm. Initialization weights can greatly influence the learning process and the overall performance [64] of the network and thus multiple evaluations with different random seeds are especially important. Most nodes also include a type of activation function, which is set depending on the algorithm. Finally, any algorithm-specific initializations are done, which might include the computation of any metrics or parameters needed in further construction steps.

4.1.3 Construction Step

After the data and network are initialized, the construction loop is started. The construction step is the most crucial part of the construction loop as it is responsible for simultaneously growing the network architecture and optimizing the weights and biases. This step utilizes the training data to optimize the network parameters. The specific way the construction happens is different for each algorithm and can vary in the following ways: The number of nodes and connections that are added each step, the locations in the network where the architecture is extended, the optimization algorithm used for modifying weight and bias values, and the freezing and cloning of parts of the network architecture. Some algorithms also employ multiple different training phases in one construction step. Such methods first train a pool of candidate nodes and after adding the best candidates to the network, they fine-tune the other parts of the network [22]. The specific construction step procedures for our proposed algorithms are explained in detail in section 4.2.

4.1.4 Optional Pruning Step

GPNNs incorporate an optional pruning step into CoNN algorithms to further optimize the network structure by removing unnecessary parts of the network architecture [18, 19, 92, 103]. The inclusion of pruning can lead to more efficient and compact networks, reducing computational complexity and memory requirements. Similar to the construction step, the specific pruning approach varies for each algorithm. The pruning technique used in all of our algorithms is a simple pruning method called magnitude pruning [6]. It works by removing weights and biases from the network that have a magnitude value below a certain specified threshold. This is done by computing the absolute values of all weights and biases in the network and setting the respective values to zero that fall below the threshold. Setting weights and biases to zero is mathematically equivalent to removing them from the network [49]. The level of the used magnitude threshold depends on the scale and distribution of the weight and bias values that are used in the algorithm. The specific algorithmic implementation of the pruning step for each of our proposed algorithms is discussed in more detail in section 4.2. We selected the magnitude pruning method as it is well-known, lightweight, and easy to integrate. Evaluating further pruning methods in our algorithms is an avenue for future work. Each algorithm uses the same type of pruning method because the novelty of this thesis lies in the construction methods of the algorithms. The pruning step is used as a tool to explore the effects of reducing the network architecture during construction and help to draw more sophisticated conclusions about the performance of our proposed algorithms.

4.1.5 Network Evaluation

Each architecture change is followed by a performance test, where the evaluation data is used to identify the performance of the network in its current construction state. The performance metrics used for the evaluation can be accuracy, F1-score, or other suitable measures depending on the problem domain [48, 67]. We use the evaluation data during this step to ensure the network's generalization capabilities are assessed by having it be tested on data that was not used directly in the training process. As the domain problems used for this evaluation are primarily classification-based, the accuracy metric is used to evaluate the algorithms. Note that the dataset must be balanced for accuracy to provide meaningful results. The resulting evaluation of the network can be used by the algorithm to determine termination or to guide the next construction steps in a performance-improving way.

Note that this evaluation is separate from the evaluation done in this thesis specified in chapter 5. This intrinsic evaluation is specific to each algorithm and is performed after a construction step (see figure 4.1), while the overall evaluation of this thesis assesses the performance of multiple algorithms on multiple benchmark datasets.

4.1.6 Termination Condition

The termination condition is a crucial aspect of CoNN algorithms, as it determines if the construction and training loop should end. It evaluates one or multiple criteria using data from other steps and ends the construction process if these criteria are fulfilled. Otherwise, the next construction loop iteration is performed. The criteria for termination can include reaching a desired performance threshold (e.g. reached desired accuracy), a maximum number of loop iterations, or a lack of significant improvement in network performance over a specified number of iterations (convergence) [27, 48, 67, 81]. For the evaluations in this thesis, a combination of target accuracy and convergence is used as the termination condition. Both of them are set specific to the domain problem and are kept the same for all algorithms to ensure fair evaluations (see chapter 5). By setting the termination conditions appropriately, overfitting can be prevented and it can be ensured that the resulting network is both of minimal size and effective in solving the given problem. This also aids in preserving a balance between computing resources and the quality of the final network structure.

4.2 Proposed Algorithms

This section goes into detail about the specific construction step for our three proposed CoNN algorithms. These are the Layerwise algorithm (4.2.1), the CasCor Ultra algorithm (4.2.2), and the Uncertainty Splitting algorithm (4.2.3). Each algorithm is designed with a specific construction strategy in mind which aims to address certain limitations or problems of existing methods (see chapter 3). The aim is to improve the final network performance and construction efficiency and to minimize the complexity of CoNN and GPNN algorithms. The following subsections describe the specific approach each algorithm takes for one construction step as well as the method used for the optional pruning step. For each suggested approach, we discuss the step-by-step algorithmic process, the implementation specifics, and any algorithm-specific initialization or termination configurations. We also explore the idea behind each approach and its expected behavior during the evaluations. The full implementation including the complete source code and documentation is available on GitHub[1] under the MIT License.

4.2.1 Layerwise Algorithm

This approach uses a straightforward strategy to build a fully connected network one layer at a time by alternating between horizontal and vertical growth. Algorithm 4.1 shows the procedure for a single construction step of this CoNN approach. The network initially starts off with only input and output nodes that are fully connected. Then, the construction algorithm adds nodes either to an existing hidden layer or creates a new layer. A hidden layer can be extended horizontally, by doubling the amount of nodes in the layer. When a layer reaches a fixed amount of nodes, the next layer is added and the weights of the previous layer are frozen (i.e. they will not be trained anymore). This layer freezing reduces computational overhead because only one layer has to be optimized at

[1] https://github.com/TheoHarti/CoNN_Evaluation

any time. The goal of this approach is to construct simple high-performing layered neural networks that maintain a balance between network depth and breadth. A visual example of the evolution of a network architecture using the Layerwise algorithm is displayed in Figure 4.2. It shows how the network alternates between horizontal and vertical growth. As can be seen in algorithm 4.1, this approach has two main hyperparameters which steer the growing process of the network. The *startingNodeAmount* defines the amount of nodes a newly added layer has and *maxLayerExpansions* specifies how often a layer can be expanded before it is frozen and a new layer is added. These values are varied in the evaluation to provide a broader scope of results for this CoNN algorithm and show how it behaves under different conditions and configurations.

Algorithm 4.1: Layerwise algorithm construction step

```
if $currentLayerExpansions == maxLayerExpansions$ then
    freeze weights and biases in the current layer
    add a new layer with the number of nodes equal to $startingNodeAmount$
    $currentLayerSize = startingNodeAmount$
    randomly initialize weights and biases of the new layer
    $currentLayerExpansions = 0$
else
    double the number of nodes in the current layer
    randomly initialize weights and biases of the current layer
    $currentLayerExpansions = currentLayerExpansions + 1$
setup new output layer weights with correct dimensions
train non-frozen weights until convergence
```

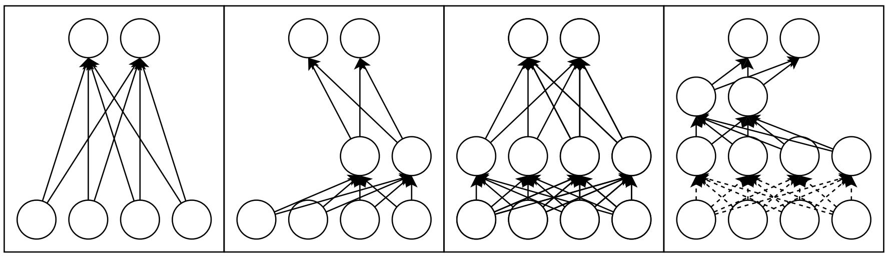


Figure 4.2: This figure shows an example of the evolution of a neural network architecture using the Layerwise algorithm. From left to right, we see how the algorithm first expands horizontally by doubling the layer size and then vertically by starting a new layer.

The weights and biases for each layer in the network are initialized using the Kaiming initialization [39]. Each node in the hidden layer has a ReLu activation function [80]. This setup was chosen because the Kaiming initialization specifically focuses on ReLu activation functions. The last step in the algorithm is the training of all non-frozen weights. This training is performed using the Adam optimization algorithm using a learning rate of 0.001 and no weight decay. Finally, the magnitude pruning step of this approach is implemented by setting all weights and biases that have an absolute value below a certain threshold to zero. The threshold value is set to 0.1 for all evaluations as this value has been shown to provide stable pruning with minimal performance reduction.

4.2.2 CasCor Ultra Algorithm

This approach is a modified version of the Cascade-Correlation (CasCor) algorithm by Fahlman and Lebiere [22]. The adjustments that were made to the original algorithm are as follows: First, the addition of multiple candidate nodes at each construction step is allowed. This modification helps the algorithm explore a larger solution space because it is now able to add layers with more than one hidden node, aiding the network's generalization capabilities. Further, it reduces the number of connections in the network because the number of layers is reduced. Second, instead of using the Quickprop [24] algorithm for optimization, we apply the commonly used Adam [45] optimization algorithm. The use of Adam optimization means there is no need to compute second-order derivatives in the optimization process which can be problematic depending on the domain problem [70]. As the Adam optimizer is a newer and more sophisticated optimization method we also expect faster convergence and better performance compared to the original Quickprop optimization method [16]. Finally, a corrective step is used, similar to [8], where frozen network weights are allowed to change for a few epochs each construction step. By including this corrective step, the CasCor Ultra method aims to address the problem of bad local minima by allowing the algorithm to potentially escape sub-optimal configurations. These local minima might arise in this algorithm because the best error-correcting nodes are chosen greedily and then frozen afterward, which limits the possible network configuration to a certain extent. We aim to fix this problem by having these frozen weights and biases be able to be fine-tuned for short periods of time during the construction of further layers. The complete step-by-step procedure of CasCor Ultra is shown in algorithm 4.2). We see that the algorithm relies on a total of three hyperparameters, which are $nCandidates$, $maxNodesPerLayer$, and $nCorrectiveSteps$. A visual example of the evolution of the constructed network architecture using the CasCor Ultra algorithm is shown in figure 4.3. We see that each construction step of a new layer in the network contains multiple nodes with the frozen connections being marked in dashed lines.

Algorithm 4.2: CasCor-Ultra construction step

```
1: generate hidden node candidate pool with size = $nCandidates$
2: each candidate node gets inputs from all input and hidden nodes
3: train candidates to maximize covariance with residual error
4: remove the highly correlated nodes from the candidate pool
5: select at most $maxNodesPerLayer$ nodes from remaining candidates
6: add selected candidates to the network and freeze their input weights
7: perform a corrective step by training all weights for $nCorrectiveSteps$ epochs
8: train non-frozen weights until convergence
```

The used activation functions are the Sigmoid function for hidden nodes and the Softmax function for the output nodes. Sigmoid was chosen because it is used in the

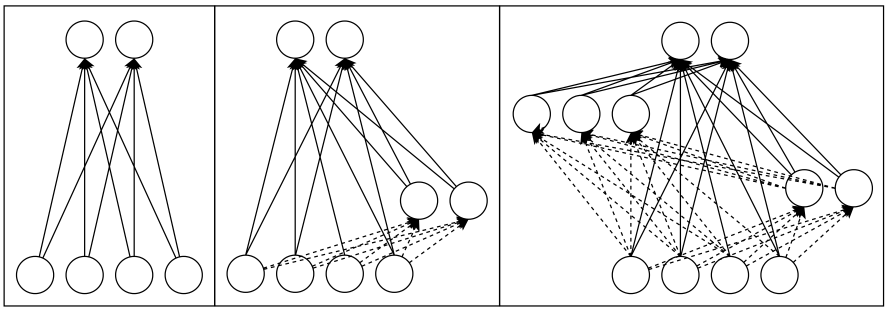


Figure 4.3: Evolution of neural network architecture using the CasCor Ultra algorithm.

original CasCor algorithm and works well while Softmax is used to normalize the output of a network to a probability distribution over the possible outcomes [22, 82]. All weights and biases are initialized using a scaled normal distribution. On the basis of the selected activation functions, the weights of the hidden nodes are scaled by 10 and the output nodes by 0.01. The Adam Optimizer for weight and bias modifications is initialized with the following parameters: A learning rate equal to 0.1, decay equal to 0.0001, and momentum equal to 0.9. The magnitude pruning step sets all weights and biases that have an absolute value below a certain threshold to zero. The pruning targets include candidate nodes and final network nodes. The threshold value is set to 0.1 for all evaluations as this value has been shown to provide stable pruning with minimal performance reduction.

4.2.3 Uncertainty Splitting Algorithm

The main idea of this approach is based on the usage of probabilistic neurons, which are implemented in Bayesian neural networks [44]. They differ from conventional artificial neurons in that each weight and bias is modeled as a probability distribution. A forward pass through the network then samples a value from this distribution and the backward pass uses the loss gradients to modify the distribution to decrease the loss. This uncertainty is used to determine which neurons to replace and which to freeze. Neurons with a high standard deviation can be interpreted as having high uncertainty and are thus replaced with multiple new nodes. By focusing on neurons with high uncertainty, the algorithm can adapt the network structure more effectively and concentrate on the areas of the problem space that are more challenging to learn. This leads to more efficient network construction. However, it comes at the cost of needing twice the amount of parameters (mean and standard deviation) for the network when compared to conventional scalar weights and biases. To mitigate this problem, the option exists to convert the final constructed Bayesian neural network into a conventional one by setting the mean value of each weight distribution as the fixed weight. This would result in faster inference times for the final network. The full algorithmic procedure is shown in 4.3 and a visual example of the construction process for the Uncertainty Splitting algorithm can be seen in 4.4. The figure shows how a high uncertainty node is selected in step three and then swapped with multiple new nodes.

The weights and biases are initialized using a standard uniform distribution. The

Algorithm 4.3: Uncertainty Splitting construction step

```
1: if $maxLayerSizeReached$ then
2:     freeze weights and biases in the current layer
3:     add a new layer with the number of nodes equal to $startingNodeAmount$
4: else
5:     find $nUncertainNodes$ nodes with the highest uncertainty in the current layer
6:     replace all high-uncertainty nodes with $nReplacementNodes$ new nodes
7: setup new output layer weights with correct dimensions
8: train non-frozen weights until convergence
```

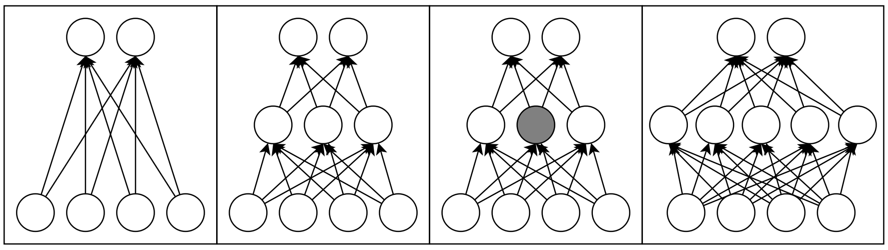


Figure 4.4: Evolution of neural network architecture using the Uncertainty Splitting algorithm.

selected starting mean and standard deviation values are set to 0 and 0.01 respectively for both the hidden and output layers. The activation functions in all hidden layer nodes is the ReLu activation and the loss function is again the Cross Entropy Loss. The Adam optimizer is initialized with a learning rate of 0.01 and no weight decay. For the magnitude pruning step, we focus on the magnitude of the mean value of the distribution and if it is below the threshold, both the mean and the standard deviation of the weights and biases are pruned. The threshold value for the mean is set to 0.1 for all evaluations of this algorithm.

4.2.4 Summary

The Layerwise, CasCor Ultra, and Uncertainty Splitting algorithms each offer novel construction strategies. By targeting different aspects of the construction process, such as layer growth, candidate node selection, weight freezing, uncertainty, and different optimization paradigms, these algorithms aim to improve the efficiency, performance, and complexity of the constructed neural networks. The proposed algorithms are evaluated in chapter 6 to determine their effectiveness on various benchmark problems using various hyperparameter configurations.

Chapter 5

Evaluation

This chapter discusses the methodology employed for the systematic evaluation of the proposed CoNN and GPNN algorithms. We provide an assessment of their performance using three well-established CoNN algorithms for comparison. These reference algorithms include multiple different approaches which enables us to show insights into the relative strengths and weaknesses of the proposed methods more easily. The complete evaluation framework encompasses six algorithms, a selection of four benchmark datasets, various defined hyperparameter sets, and a suite of result metrics. The datasets were chosen to include varying levels of complexity and data types to ensure the algorithms are evaluated on a wide range of problems. We selected a set of common result metrics as well as developed new metrics to provide multiple views on the performance of the algorithm such as accuracy, efficiency, and parameters count. To combine all the above-mentioned factors into a systematic and reproducible evaluation, we developed an open-source evaluation software system. This application is designed to automate the process of assessing multiple algorithms on various datasets using different sets of hyperparameters. In addition, random repetitions are performed to mitigate bias in the results.

The chapter is structured as follows: We first present the evaluation system in section 5.1, where we focus on software architecture, available user interfaces, and the logging of the result metric data. Next, section 5.2 presents each of the six evaluated algorithms and mentions any notable implementation details and fixed as well as varied hyperparameters. The four chosen benchmark datasets and the numerous selected result metrics are presented in sections 5.3 and 5.4 respectively. Finally, section 5.5 provides the complete evaluation procedure to ensure transparency and reproducibility of this evaluation.

5.1 Evaluation System

We created a software system to facilitate and simplify the automatic evaluation of CoNN algorithms on multiple datasets using different hyperparameter settings and random seeds. The evaluation software is written in Python using the machine learning framework PyTorch[1]. The extensive possibilities for data handling and analytics that

[1]https://pytorch.org

Python offers provide a versatile platform for implementing, training, and evaluating constructive neural networks. We chose PyTorch over other machine learning frameworks such as TensorFlow because it is intuitive to use, offers numerous tools and libraries, and is well integrated with the Python ecosystem. However, the most important reason is the use of dynamic computational graphs. This feature enables direct modification of neural network models during training, which is crucial for implementing CoNN algorithms.

The complete source code of the evaluation software, along with its documentation, is made available on GitHub[2] under the MIT License. This decision was made to enable other researchers to reproduce our findings and encourage further research in the field of constructive neural networks. We hope that other researchers will be able to integrate their algorithms directly into the evaluation system. This would simplify the evaluation procedure and allow for easier direct comparisons between algorithms in this field.

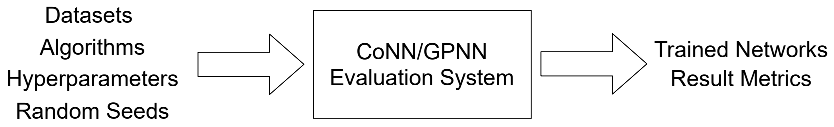


Figure 5.1: Overview of the inputs and outputs of the evaluation system.

Figure 5.1 shows a black box Input-Output Diagram of the evaluation system. The system receives a number of inputs such as the chosen algorithms for evaluation and their respective hyperparameters, the selected benchmark datasets, and a list of seed numbers to use for random repetitions of the evaluations. These input parameters can be supplied through different input modalities such as graphical interfaces or configuration files and the command line. Using the specified parameters, the system automatically configures a list of evaluations to run. A single evaluation incorporates one algorithm with one hyperparameter set and trains it on one dataset using a specific random seed. After all evaluations are finished, the system outputs a comprehensive set of result metrics, which provide detailed insights into the algorithm performance and the complete training process. In addition, users have the option to export the constructed neural networks after the evaluation has finished.

5.1.1 Software Architecture

The software architecture of the evaluation system was designed with a focus on modularity, reusability, and extensibility. Employing these principles makes it easier to understand, utilize, and adapt to new requirements. Additionally, other researchers can use existing interfaces to incorporate new constructive algorithms, benchmark datasets, evaluation metrics, data logging methods, and other enhancements into the software. By providing this modular system and allowing other researchers to integrate new functionality we aim to improve and simplify the evaluation process in the field of CoNN and GPNN algorithms.

[2]https://github.com/TheoHarti/CoNN_Evaluation

The class diagram in figure 5.2 illustrates the most important elements of the software architecture and shows their dependencies. Dashed lines indicate dependencies while the normal lines show inheritance relationships. The central component is the Evaluator class which is responsible for orchestrating the entire evaluation process. It obtains the required inputs, evaluates the algorithms with the specified hyperparameters on the given datasets, and organizes the logging of the results. The classes handling user input are the UIEvaluator as well as the ConsoleEvaluator. They collect and preprocess the input parameters and forward them to the Evaluator. The abstract ConstructiveAlgorithm class encapsulates the functionality that a CoNN algorithm should possess, whereas most specific implementations are located in the child classes. The abstract Logger component handles the recording and calculation of result metrics. Multiple variants such as the CSVLogger or the TensorBoardLogger are available as output modalities. The Dataset component contains the currently used data for evaluation and performs data shuffling, data splits, and data visualization tasks. Multiple enumerations such as DatasetType, LoggerType, and AlgorithmType are employed to specify and configure the subclasses for the abstract components.

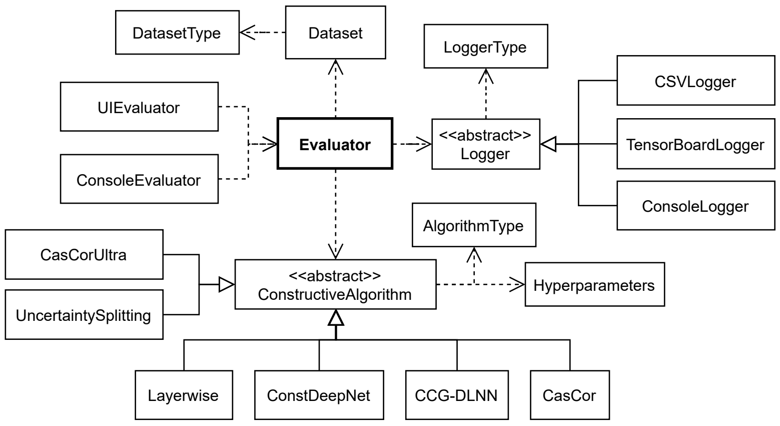


Figure 5.2: Overview of the most important classes in the evaluation system and their dependencies.

To achieve the above-mentioned software principles such as modularity and extensibility, we incorporated several design patterns. We apply class inheritance to enable polymorphism for algorithms and loggers, allowing for a flexible structure that can easily adapt to new requirements or changes. The factory method pattern is utilized for instantiating the specific subtypes of the abstract classes in the system. This approach decouples the process of object creation from the specific classes, thereby improving modularity. To be able to have the UIEvaluator receive continuous updates from the Evaluator, the observer pattern was implemented. Finally, we integrated the behavioral template method pattern to enable parts of the algorithmic procedure of CoNN and GPNN algorithm to be lifted into the parent class while keeping the algorithm-specific functionality in the subclasses. This approach reduces code duplication and thus makes the architecture simpler and more reusable.

5.1.2 User Interface

The evaluation system software provides two ways of interaction for the user, each with distinct strengths and limitations. First, the graphical application offers a simple and intuitive interface that provides the user with direct visual feedback. This mode of interaction is especially useful for testing implementations and researching algorithmic behaviors as it allows adjustment of specific algorithmic configurations and presents unique visual information. Secondly, the console application allows users to send instructions to the evaluation system using a config file and the command line. This approach is well-suited for performing large-scale evaluations of multiple algorithms, datasets, randomizations, etc. The specifics of the GUI and console evaluators are presented in 5.1.2 and 5.1.2 respectively.

GUI Evaluator

The GUI evaluator provides an easy-to-use interface as well as visual feedback for the user. As shown in figure 5.3, the main screen of the GUI evaluator is separated into three areas: configuration on the top left, result metrics on the bottom left, and training visualizations on the right.

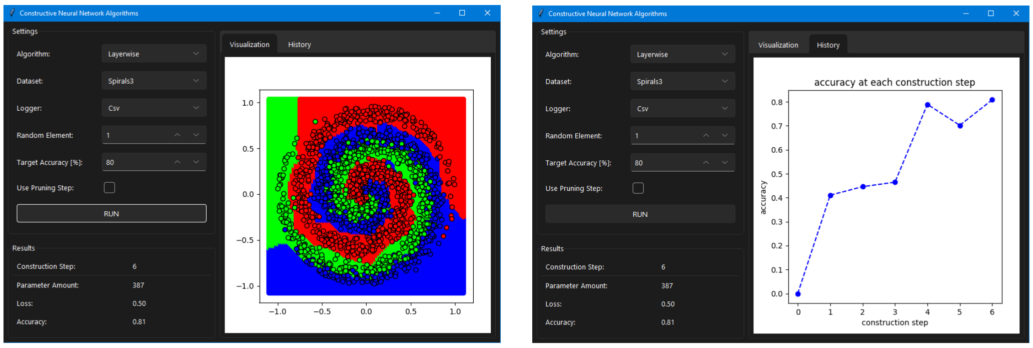
The configuration area provides input elements such as dropdown selects and text fields to allow the user to specify the desired evaluation setup. These configurable parameters include the type of algorithm, dataset, result logger, random seed, and target accuracy. Both the result and graphic areas provide information about the current state of training and network construction by showing scalar result metrics as well as visualizations. The diagrams are divided into different tabs where each tab contains unique visual information about the dataset and training process. Currently, two tabs exist, where the first shows a visualization of the current decision boundary, and the second displays a line graph with the history of accuracy values at each construction step. A simple button press starts the evaluation of the current configuration. To maintain the usability of the interface throughout the evaluation, the training process is executed on an independent background thread. Nevertheless, buttons and text fields are disabled while training is performed to avoid complications. The result metrics and diagrams are updated at each construction step using the latest data throughout the training process.

The graphical interface was implemented using the standard Python GUI toolkit Tkinter[3]. Tkinter provides a wide range of graphical widgets to simplify Python GUI implementation on most Unix platforms, macOS, as well Windows systems. To improve the design and appearance of the graphical interface, we employed Tkinter-themed widgets and applied the Sun Valley theme[4].

Console Evaluator

The console evaluator is designed to facilitate large-scale CoNN and GPNN evaluations using a simple command line interface. The user provides a configuration file that specifies a list of CoNN algorithms, hyperparameters, datasets, random seeds, and result metrics for the evaluation. The software subsequently performs the needed evaluations

[3]https://docs.python.org/3/library/tkinter.html
[4]https://github.com/rdbende/Sun-Valley-ttk-theme


Constructive Neural Network Algorithms
Visualization
History
accuracy at each construction step
accuracy
construction step

Settings
Algorithm: Layerwise
Dataset: Spirals3
Logger: Csv
Random Element: 1
Target Accuracy [%]: 80
Use Pruning Step:
RUN

Results
Construction Step: 6
Parameter Amount: 387
Loss: 0.50
Accuracy: 0.81

Data visualization Result history

Figure 5.3: The GUI application of the evaluation system running on a Windows 10 machine. The left image shows the data visualization tab, which is the current classification boundary of the 3-Spirals-Random dataset. On the right, the history of the training accuracies for each of the construction steps is displayed.

automatically and generates a log file with the specified metrics as a result. An example of the JSON configuration file is presented in listing 5.1. The user provides this configuration file as a command line argument, which might look like this: `python evaluator.py evaluation_config.json`. The configuration file is parsed using the standard Python json package and then an evaluation loop is set up, which runs an evaluation once for each combination of algorithm type, hyperparameter set, dataset, pruning mode, and random seed.

5.1.3 Logging

The evaluation system offers a total of three distinct logging types with each having its advantages and drawbacks for specific use cases. These types are the CSV logger, the TensorBoard logger, and the console logger. The software architecture for this logging functionality follows an object-oriented approach. A *Logger* base class can be used by the other parts of the system and it provides abstract methods such as *log_scalar()* that the specific subclasses can implement. The *LoggerTypes* enumeration provides a set of available logger types, which the system then translates into the corresponding subclass

The TensorBoard[5] logger uses a toolkit provided by TensorFlow to track and visualize metrics. It allows to track metrics 1:1 such as accuracy per epoch or loss per construction step and directly generates useful visualizations viewable through a web interface. As result data is stored in a special file format and its CSV data export is limited, it can be difficult to use the logged metrics for further processing. TensorBoard was initially chosen for its straightforward logging interface and visualization capabilities, but as more unique logging was needed, it became necessary to switch to a different solution.

Logging to a CSV file provides more flexibility and freedom in data handling and post-processing. Additionally, as CSV is a well-known file format, many existing data analytics tools can work with it. Each evaluation run leads to a single row in the CSV file

[5] https://www.tensorflow.org/tensorboard

```json
{
    "algorithms": [
        {
            "algorithm_type": "Layerwise",
            "hyperparameter_combinations": [
                { "n_starting_hidden_nodes": 2, "n_max_layer_expansions": 4 },
                { "n_starting_hidden_nodes": 3, "n_max_layer_expansions": 4 },
                { "n_starting_hidden_nodes": 5, "n_max_layer_expansions": 3 },
                { "n_starting_hidden_nodes": 7, "n_max_layer_expansions": 3 }
            ],
            "pruning_modes": [
                { "is_pruning_active": false },
                { "is_pruning_active": true, "magnitude_threshold": 0.1 }
            ]
        }
    ],
    "datasets": [
        { "dataset_type": "Spirals2", "target_accuracy": 0.9 },
        { "dataset_type": "Spirals3", "target_accuracy": 0.8 },
        { "dataset_type": "Curves", "target_accuracy": 0.99 },
        { "dataset_type": "Compound", "target_accuracy": 0.85 }
    ],
    "random_numbers": [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
}

```

Program 5.1: The JSON configuration file used for the evaluation.

containing a variety of metrics with a total of 20 data columns. These columns provide general evaluation information such as date and time, algorithm type, dataset, random seed, target accuracy, and hyperparameters, as well as result metrics such as construction accuracy and loss, construction parameters, construction step training epochs, and test accuracy and loss. We use the CSV logger solution for the evaluation in this thesis as it allows the merging of many different evaluation iterations into a single file that can be easily reused later to calculate and combine result metrics as well as visualize parts of the data.

The console logger provides a simple interface to print results to the terminal in a structured and readable way. The output of this logger differs from the CSV approach, in that only the metrics that are directly relevant are output. Results that need further pre-processing are unnecessary in this case. This logger is useful for providing quick evaluation insight on manual test runs without any post-processing, which is especially important for providing feedback for algorithm debugging. However, the result data is not persisted and possibilities for providing visualizations are very limited using this logging approach.

5.2 Algorithms and Hyperparameters

The evaluation procedure of this thesis contains a total of six algorithms. These are our three proposed algorithms (CasCor-Ultra, Layerwise, and Uncertainty-Splitting) as well as three state-of-the-art CoNN algorithms (CasCor [22], CCG-DLNN [60], and Const-DeepNet [105]). The following subsections discuss the relevant algorithmic details and hyperparameters of all evaluated approaches and present the selected hyperparameter sets for evaluation.

5.2.1 Proposed Algorithms

This subsection presents the hyperparameters for each proposed algorithm. To be able to show and compare hyperparameter sets that specifically impact the construction behavior and performance of the network, we only select a specific variety of construction-related hyperparameters in the evaluation of each algorithm. During pre-evaluation, we tested a larger variety of configurations, but due to spatial constraints, the subsequent evaluations only include four hyperparameter sets for each algorithm. The sets are selected such that they give a reasonable overview of the hyperparameter space, as much as this is possible using only four configurations. The results for each of these hyperparameter sets are presented in chapter 6 and only the best-performing set is selected to be used for the comparisons with the state-of-the-art algorithms. The rest of the hyperparameters, such as learning rates, decay, and momentum are fixed for the complete evaluations. We preselected well-performing values for each of these hyperparameters by performing multiple evaluations. The next paragraphs present the values for the fixed and varied hyperparameters for all the proposed algorithms.

Layerwise

The fixed hyperparameters for the Layerwise algorithm consist of the learning parameters for the optimization algorithm and the selected pruning threshold. We found in the pre-evaluation phase that the Adam optimizer performs best for this algorithm setup when configured using a learning rate of 0.001 and no weight decay. The magnitude pruning step has shown stable pruning results with minimal performance reduction with a threshold value of 0.1. For the variable hyperparameters, we focused on the two construction-specific parameters $startingHiddenNodes$ and $maxLayerExpansions$. We constructed four sets of combinations for these two parameters with the aim to provide results that give insight into the strength and weaknesses of different algorithm configurations. The values for the four sets are (2, 4) (3, 4) (5, 3) (7, 3), where the first value is the number of $startingHiddenNodes$ and the second value shows $maxLayerExpansions$.

CasCor Ultra

The CasCor Ultra algorithm has the configuration of the Adam optimizer, the pruning threshold, and the minimum corrective steps as fixed parameters. The optimizer was found to work best using a learning rate of 0.05, a learning rate decay of 0.0001, and a momentum of 0.6. The minimum corrective steps are set to 10, whereas more steps will be performed if performance continues to get better. Finally, the pruning threshold is set to 0.1 has proved very effective in pre-evaluations. The variable hyperparameter sets are composed of two elements: $nCandidates$ and $correlationThreshold$. These two parameters are combined to create four hyperparameter sets for evaluation. These are

(7, 0.5), (15, 0.4), (25, 0.3), and (50, 0.2). These sets were created to provide a good mixture of small or large candidate pools and narrow or wide hidden layers.

Uncertainty Splitting

Similar to the other proposed algorithms, the fixed hyperparameters for the Uncertainty Splitting algorithm are the optimizer parameters and the pruning threshold. Pre-evaluations showed a learning rate of 0.03 and no decay to be a well-performing configuration. For the pruning threshold, 0.1 was selected as it provided stable pruning most of the time. The variable hyperparameters consist of four elements: The *nStartingHiddenNodes*, the *maxLayerSize*, the *nUncertainNodes*, and finally the *nReplacementNodes*. Using these variables, the following four hyperparameter sets were created: (3, 32, 3, 2), (4, 64, 3, 3), (5, 32, 5, 4), and (7, 64, 5, 5). The composition of the set aims to provide types of algorithm configurations that grow at varying speeds and create networks with different widths.

5.2.2 Benchmark Algorithms

This subsection focuses on the three state-of-the-art CoNN algorithms: CasCor [22], ConstDeepNet [105], and CCG-DLNN [60]. These three approaches were selected to be evaluated and compared with our proposed algorithms. For more information and an algorithmic breakdown of each method see chapter 3. This subsection presents the selected hyperparameter for the algorithms as well as specific implementation details that are noteworthy for the evaluation. Each of the algorithms underwent a pre-evaluation phase where multiple hyperparameter combinations were tested to determine the best-performing set of parameters for each algorithm. These selected hyperparameters are fixed for all subsequent evaluations of the algorithms. Additionally, it is worth noting that small implementation details of the algorithms used in our evaluation software deviate from their original paper for various reasons. These deviations are explained in the following subsections for each algorithm.

Cascade Correlation (CasCor)

The CasCor algorithm [22] sequentially adds new one-node hidden layers to the network which maximizes correlation with the output error. CasCor was chosen as a benchmark algorithm for this evaluation as it is the most well-known algorithm and is still in use today. Thus, providing us with a valuable baseline for assessing the performance of our proposed algorithms.

The algorithm was implemented in the evaluation software system as closely as possible to the specifications in the original paper. One difference in our implementation is the usage of the default stochastic gradient descent for optimization, instead of the Quickprop algorithm. This choice was made because Quickprop requires second derivatives of the loss function, which is not always possible in our scenarios. Other details about the implementation include: The activation function of the hidden nodes are Sigmoid functions, while the output nodes use Softmax activation. Finally, the weights in the network are initialized using random samples from a standard normal distribution.

Using the experience and results gained from the pre-evaluation phase of the algorithm, we selected the following parameters as the final hyperparameter set for this evaluation: The learning rate is fixed to 0.1 with a decay value set to 0.0001, which showed the best convergence speed overall in preliminary evaluations. A candidate pool size of 25 provides a good balance between computational effort and generating effective nodes for selection. The pruning threshold was set to 0.1 to allow for optimal removal of parameters without impacting performance notably.

ConstDeepNet

ConstDeepNet [105] uses the classic backpropagation algorithm to incrementally construct a multi-layer hidden network one node at a time. The reasons for selecting this approach include that it was published in the last few years and has shown impressive performance results in its original paper. Additionally, it integrates a different learning approach than CasCor and CCG-DLNN as it does not use a correlation measure to add new nodes to the network. This approach is similar to the method of the proposed Layerwise algorithm and thus a comparison in performance and behavior is valuable.

Our implementation of the ConstDeepNet algorithm follows the description in the original paper as closely as possible. Some detail such as the activation and the optimization algorithm are not directly specified. Our implementation uses the following setup: All nodes in the network employ the ReLu activation function and the error function is the cross-entropy loss. The weights and biases are initialized using the Kaiming initialization [39]. The optimization process is performed using the Adam optimizer. The final hyperparameters that were selected in the pre-evaluation phase are as follows: A maximum layer size of 20, striking a good balance between depth and width. A learning rate of 0.005 with no rate decay and finally a pruning threshold of 0.1.

CCG-DLNN

CCG-DLNN [60] is an extension of the CasCor algorithm that builds hidden layers including multiple hidden nodes using the existing correlation maximization procedure. We selected this approach as a benchmark algorithm as it is a recent improvement to the well-known CasCor algorithm that showed good performance in the evaluation presented in the original paper. As our proposed CasCor Ultra algorithm also extends CasCor and allows for multi-node hidden layers, we see a comparison between these approaches as useful.

The algorithm was implemented into the evaluation software system by closely following the specifications given in the original paper. Key choices in our implementation where no information was given in the paper are: The activation functions of the network nodes, where we selected the Sigmoid function for the hidden nodes and the Softmax function for the output nodes. The optimization algorithm was chosen to be the Adam optimizer and the loss function is the cross-entropy loss. The weights in the network were initialized by sampling from a standard uniform distribution. The pre-evaluation phase yielded the following hyperparameters to be used for the final evaluations: A maximum layer size of 12 with a candidate pool size of 25. The optimizer is configured with a 0.005 learning rate and no decay. Finally, the pruning step is performed using a threshold of 0.1.

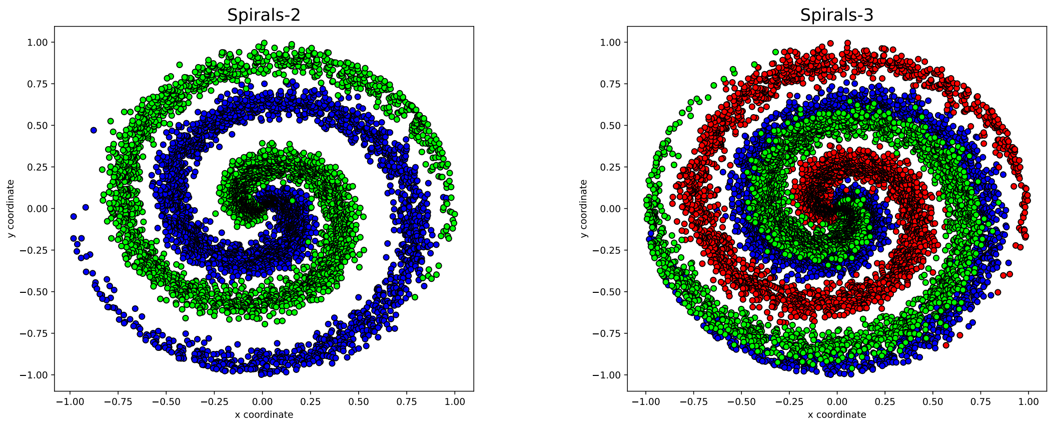
5.3 Benchmark Datasets

The proposed constructive algorithms, as well as the state-of-the-art algorithms, are evaluated on a set of benchmark problems to measure and compare performance and analyze their behavior. We selected a total of four datasets (Spirals-2, Spirals-3, Moons, and Classification), which are discussed in detail in the following subsections. Even though only four datasets were selected for this evaluation, several other datasets are implemented in the evaluation system software and can be used in the future.

We acknowledge that the selected datasets do not perfectly represent real-world data. However, these datasets were chosen specifically because they allow for simple control over the number of data points, the problem complexity, and the amount of randomness. This makes the evaluations easier, more controllable, and more consistent. Furthermore, several CoNN and GPNN approaches in literature were also evaluated using generated data sets such as these [22, 60].

5.3.1 Sprials-2 and Sprials-3

The spiral datasets are widely used benchmark datasets in the space of CoNN and GPNN algorithm [22, 60]. The datasets usually consist of 2-dimensional input points arranged in a spiral pattern, each belonging to a distinct class. The network is trained to predict to correct class given the point coordinates. We created implementations of spiral datasets, called Sprials-2 and Sprials-3, which add random noise to the constructed spirals to increase classification difficulty. In addition, the added noise increases susceptibility to overfitting, which shows how the algorithms handles such problems. Our spiral implementation works by generating the spirals for each class separately. Polar coordinates, which means radii and angles, are generated and then transformed into Cartesian coordinates using sine and cosine. During generation random offsets are added to the angle to create the noise. Examples of the two datasets can be seen in figure 5.4.



Spirals-2 Spirals-3

Figure 5.4: Examples for the Spirals-2 and Spirals-3 datasets. The different classes are colored in green and blue for Spirals-2 and red, green, and blue for Spirals-3.)

The main reason for using these types of datasets is that they provide complex and nonlinear decision boundaries that the model has to learn to solve the problem. In addition, as they are typically 2-dimensional, the data and the corresponding decision boundary can be visualized easily, providing more insight into the model performance.

5.3.2 Moons

The Moons dataset[6] is a simple binary classification problem with two interleaving half circles. It was selected for this evaluation mainly because of its simple setup, while still creating a non-linear problem for the algorithm to solve. This combination allows us to assess how the evaluated algorithms handle such problems by constructing small, well-performing networks. The datasets also provide a gap between the classes which helps to test the algorithm's capability of finding good decision boundaries when no errors are present. Additionally, its ease of visualization aids the development and testing of new algorithms. The dataset is available through the scikit-learn library and can be dynamically generated by supplying the number of data points, the amount of random noise, and a random seed element. A visual example of the dataset is displayed in figure 5.5.

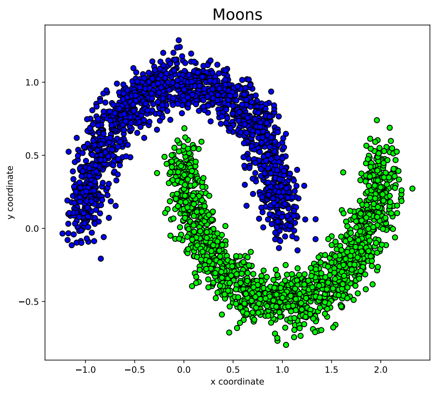


Figure 5.5: Example of the Moons dataset.

5.3.3 Classification Generator

The Classification[7] dataset generator from the scikit-learn library is able to create random n-class classification problems. The generator creates clusters of points normally distributed about vertices on a hypercube and assigns the clusters to specific classes. The exact problem type can be specified by providing a variety of parameters to the generator such as feature types, class balance, data scaling, and randomness. The configuration used for this evaluation is as follows: A total of six features where five are informative and one is redundant, four different classes, and two clusters using a class

[6]https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_moons.html
[7]https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html

separation of 1.5. This setup creates a moderately hard classification problem that resembles real-world datasets. We selected this generator for our datasets because of its many configuration options and its ability to create datasets resembling real-world data.

5.3.4 Preprocessing and Stopping Criteria

As each of the selected datasets is dynamically generated, the number of data points can be adjusted based on our needs. We selected a total of 5000 data points for each problem. After generation, the data is given to the network without performing any preprocessing or feature selection. This decision was made because the goal is to compare the different algorithmic approaches and not to achieve high accuracies on these datasets. All datasets are shuffled using the random number seed of the current evaluation to avoid biased results through the ordering of the data. After shuffling, the datasets are split into train, evaluation, and test data using split percentages of 56%, 14%, and 30% respectively. As the number of data points of each problem is large enough the relatively small percentage of evaluation data is no problem. The selected target accuracies for each dataset are as follows: 90% for Sprials-2, 80% for Sprials-3, 99% for Moons, and 85% for the classification generator dataset.

5.4 Metrics

The task of selecting useful metrics for evaluating and comparing CoNN and GPNN approaches is non-trivial. This difficulty arises because each algorithm constructs its network in a different way using various growth strategies. Well-designed metrics for these types of algorithms should not only provide information about the performance of the constructed model but also give insights into network architecture and the efficiency and speed of the training and construction process. Only by providing multiple or all of these metrics can constructive neural networks be evaluated and compared in a reasonable way.

In related literature, various metrics have been used to assess and compare the performance of constructive algorithms for neural networks. The most common metric combination in this area is to use training and testing accuracy as well as network size defined by the neuron count as metrics [67, 86]. However, the neuron count metric may be misleading because the parameters that are actually optimized are the weights and biases. This is the reason [57] select network parameter count as a metric instead. Fahlman [22] incorporates the average epochs per construction step as an additional measure to provide insight into the efficiency of the training process. In a similar attempt, Lin et al. [53] recorded the training time and the inference time. Although this metric provides information about the training speed, it is highly dependent on the underlying computing architecture of the evaluation. Finally, Mohamed et al. [60] present a cost metric to measure total training effort, but no calculation method was provided in the paper.

Inspired by the metric found in the existing literature and using the expertise we gathered in the development and testing phase, we selected the following metrics for our evaluation.

- Final loss value on training and testing data

- Final accuracy on training and testing data
- Training accuracy at each construction step
- Total parameters at each construction step
- Trainable parameters at each construction step
- Pruned parameters at each construction step
- Number of training epochs performed at each construction step
- Total construction effort of the network training process

These specific metrics were selected, because they give a measure of performance (accuracy) while also giving information about the network architecture and the network construction process. The loss and accuracy provide insight into the performance of the network on the benchmark datasets, which is generally the most important metric. We only use the accuracy metric for performance, as all datasets for the evaluation are classification based (see section 5.3). In addition to the final accuracy, we record the accuracy at each construction step to show how the network learns and is able to provide performance history graphs. In an effort to gain more understanding of the network architecture, we decided to record the number of total parameters (weights and biases), trainable parameters (not frozen and not pruned), and pruned parameters for each construction step. These three metrics give detailed feedback about the network architecture and can be used for further processing. In combination with the metrics above, the number of training epochs at each construction gives ideas about the speed and efficiency of the construction process.

Finally, we combine the about mentioned trainable parameters and the number of training epochs to calculate a novel constructive effort (CE) metric. By combining these recorded metrics we aim to accurately measure the effort expended in training and constructing the network. Additionally, because this metric does not use time measurements, the underlying evaluation infrastructure is irrelevant. Equation 5.1 demonstrates how CE is calculated, where n denotes the current construction step.

$$CE = \log\left(\sum_{1}^{n}(\text{trainableParameters}_n \cdot \text{nrOfEpochs}_n)\right) \tag{5.1}$$

5.5 Evaluation Procedure

All of the results shown in chapter 6 were acquired using an automatic evaluation procedure. As can be seen in figure 5.1, a total of four main input variables are needed for a run of the evaluation system. These are the selected dataset, the used algorithm type, the hyperparameter set for this algorithm, and finally the random seed used for all random processes during the evaluation and training (e.g. shuffling datasets or initializing weights and biases). Figure 5.6 shows all specific variations of the four input variables and the order in which they are evaluated. Each box symbolizes a loop where the contained procedure is iterated once for each input variable. We evaluate a total of six algorithms using multiple hyperparameter sets (see section 5.2) on four benchmark datasets (see section 5.3). Each of these combinations is evaluated ten times using different random seeds (numbers from 1 to 10). This ensures results are less likely to be influenced by good or bad random processes such as data splits or weight and bias initializations during training. The variation for each input parameter can be easily specified using the JSON config file and the console interface of the evaluation system (see subsection 5.1.2). The possible parameter combinations are defined in the config and the evaluations are then performed automatically according to it. Multiplying the loop iterations for each input parameter we get a total of 1280 evaluations to generate all the results of this thesis. Using the CSV logging functionality of the evaluation system (see subsection 5.1.3), this results in a single CSV file containing 1280 rows.

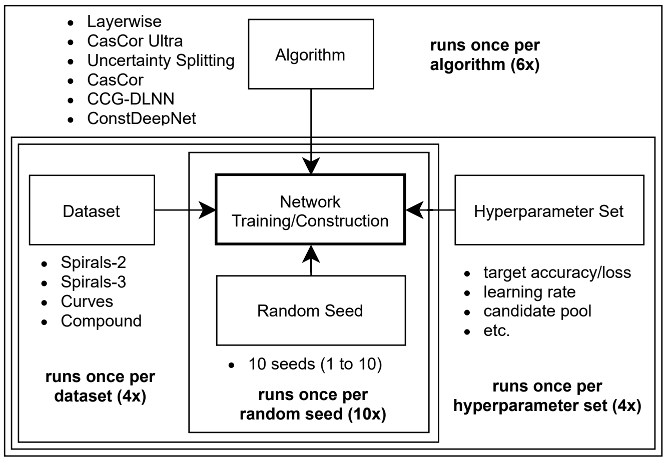


Figure 5.6: Overview of the procedure for evaluating the proposed algorithms of this thesis.

All of the evaluations have been performed on a personal computer using the Windows 10 operating system. The main hardware specifications of the machine include an AMD Ryzen 5 1600 processor, a Radeon RX 560 graphics card, and 16 gigabytes of RAM. In the software area, Python version 3.7 and the PyCharm Community Edition 2021.3 development environment were used for the evaluation. These specifications are provided for completeness as the impact of the underlying hard- and software on the results should be negligible. This is because we selected the result metrics to not rely on any time measurements, but use the actual number of computations which are independent of the computation speed.

Chapter 6

Results

In this chapter, we present the results obtained from evaluating the proposed CoNN algorithms on multiple machine learning benchmark problems. Refer to chapter 5 for more details on the specific datasets, metrics, and configurations used for the evaluations and chapter 4 for in-depth explanations about each evaluated algorithm. The chapter is structured such that the results for each of the three proposed algorithms are discussed individually. Here, the focus lies on the comparison between different hyperparameter sets to give insight into the effects of hyperparameter selection and show algorithm robustness. The fourth and final section provides comprehensive comparisons among the proposed algorithms and with three state-of-the-art CoNN algorithms. These comparisons provide crucial information about the performance of the proposed algorithms in the context of multiple well-known solutions. This allows us to define the strengths and weaknesses of each algorithm and gather information for possible improvements.

6.1 Layerwise Algorithm Results

This section presents the evaluation results of the Layerwise algorithm on multiple benchmark datasets. The algorithm was evaluated using four different sets of hyperparameters as specified in the evaluation chapter 5. The results are divided into three subsections which cover the performance of the algorithm on training and test data, the final network size and construction effort, and the comparison of the results of the pruning and non-pruning variants of the algorithm. Finally, the results are reviewed and discussed in subsection 6.1.4, where the best-performing hyperparameter set is determined. This configuration is then used for the comparisons with the other proposed algorithms and the state-of-the-art algorithms in section 6.4.

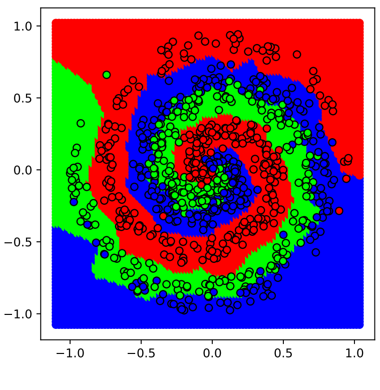
6.1.1 Algorithm Performance

The performance results of the Layerwise algorithm on each dataset are illustrated in table 6.1. They consist of the final network accuracy on the train and test data set for each benchmark problem. Each row represents a set of hyperparameters ($startingNodeAmount$ and $maxLayerExpansions$) shown in the hyperparameters (HP) column. The results show that target accuracies (displayed next to the dataset name) were reached for all hyperparameter sets. In general, all target accuracies were overshot by 1% up to some-times more than 5%. This is likely due to the large construction steps in the later stages of layer widening. The simple Moons dataset was solved nearly perfectly. The missing percentages seem to be due to the problem of the algorithm having no errors left on the evaluation set and thus finishing the algorithm. This means that the decision boundary will be very close to one class of the dataset and the randomness in the test set will cause errors at these points.

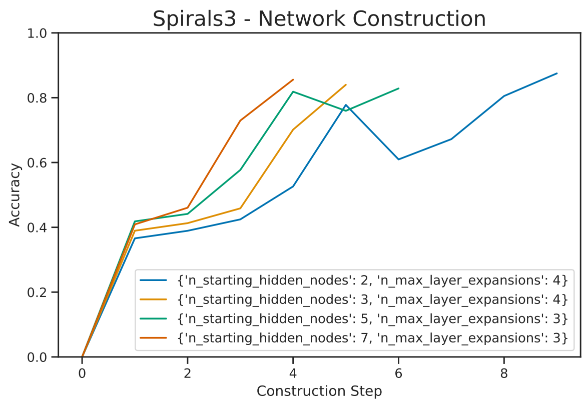
HP	Spirals-2 (0.9)		Spirals-3 (0.8)		Moons (0.99)		Class (0.85)	
	Train	Test	Train	Test	Train	Test	Train	Test
$(2, 4)$	0.946	0.947	0.826	0.819	0.999	0.998	0.875	0.863
$(3, 4)$	0.938	0.942	0.833	0.846	0.998	0.998	0.871	0.860
$(5, 3)$	0.963	0.961	0.817	0.815	0.999	0.998	0.883	0.871
$(7, 3)$	0.960	0.955	0.841	0.854	0.998	0.998	0.888	0.879

Table 6.1: Layerwise algorithm performance on benchmark problems.

A somewhat peculiar observation in the results is the closeness of train and test accuracies, for which we however have a plausible explanation. Counting the ten random repetitions of each evaluation we find that about half of the time, the training accuracy is higher than the test accuracy, while it is reversed for the other half. We believe that this closeness of train and test accuracy stems from the good generalization capabilities of the algorithm. This can be further supported when looking at the decision boundaries produced by the Layerwise algorithm shown in figure 6.1, where the boundary for Spirals-2, as well as Spirals-3 perfectly encapsulate the target classes with no overfitting. We see an error in the evaluation system as unlikely, as other algorithms such as CasCor Ultra do not show the same closeness between train and test accuracies.



Spirals-3 decision boundary (a)



Spirals-3 accuracy per construction step (b)

Figure 6.1: The left figure (a) shows an example of the resulting decision boundary of the Layerwise algorithm on the Spirals-3 dataset. Figure (b) depicts the median training accuracy histories for each hyperparameter set on the same dataset.

The median performance curve in the construction process of the Layerwise algorithm can be seen in figure 6.1 (b). The curves for each hyperparameter set clearly show

how the expansion of the layers happens in differently-sized steps. The small variation in accuracy in the first step is due to the number of starting hidden nodes. In the next steps, we find that the accuracies grow at similar rates but the main acceleration in accuracy increase start at different times with lower starting hidden nodes increasing later. It can also be observed how accuracy dips at a certain construction step, which indicates that the maximum size of the layer is reached and a new layer is initialized.

6.1.2 Network Complexity and Efficiency

The data displayed in table 6.2 gives insight into to ability of the Layerwise algorithm to produce compact, high-performing networks in an efficient way. The three columns for each dataset display the test accuracy, the total parameters of the final network, and the construction effort (CE) metric.

HP	Spirals-2 (0.9)			Spirals-3 (0.8)			Moons (0.99)			Class (0.85)		
	Acc	Param	CE	Acc	Param	CE	Acc	Param	CE	Acc	Param	CE
$(2,4)$	0.947	162.0	13.71	0.819	378.0	14.49	0.998	42.0	11.82	0.863	92.0	12.31
$(3,4)$	0.942	122.0	13.23	0.846	291.0	14.57	0.998	62.0	11.79	0.860	70.0	12.01
$(5,3)$	0.961	202.0	13.65	0.815	243.0	14.37	0.998	77.0	11.89	0.871	59.0	11.88
$(7,3)$	0.955	142.0	13.30	0.854	339.0	14.87	0.998	72.0	11.82	0.879	81.0	12.03

Table 6.2: Layerwise algorithm results on benchmark problems showing network complexity and construction efficiency.

The results show that the overall parameter counts and construction effort results fit the problem complexity with the problems having high amounts of non-linearities needing more parameters and more time to construct. The differences between the hyperparameter sets on both these metrics are visible but not enormous with the best and worst performers varying no more than 50%. When ranking the number of parameters needed to solve the problem from one to four and using the average over all datasets, we find that the (3,4) hyperparameter set achieves the top rank of 1.75. The other three sets trail behind with (5,3) having the average rank of 2.5, (7,3) with 2.75, and finally (2.4) with an average of 3. Performing the same ranking for the CE metric we again find set (3,4) on top with an average rank of 1.75 and (2,4) as the worst performer. The good performance of the (3,4) set seems to be due to it having a good mixture between the size of the construction steps and the width of its layers. The hyperparameter set (2,4) on the other hand adds fewer nodes each step resulting in higher construction effort, but small networks for simple problems such as the Moons dataset.

6.1.3 Pruning Step Results

Table 6.3 presents the differences in the results of the default Layerwise CoNN approach with the results of the GPNN Layerwise variant where the magnitude pruning step is integrated. The columns for each dataset are similar to the ones seen in subsection 6.1.2, except that they show the percentage difference between the respective results of the pruning and non-pruning Layerwise algorithm.

We find that even with the pruning step activated, the test accuracies for each hyperparameter combination stay mostly the same, only showing decreases or even increases

HP	Spirals-2 (0.9)			Spirals-3 (0.8)			Moons (0.99)			Class (0.85)		
	Acc	Param	CE	Acc	Param	CE	Acc	Param	CE	Acc	Param	CE
$(2,4)$	-0.4%	-11.4%	-16.0%	-1.6%	12.9%	0.7%	0.0%	-9.5%	-8.7%	-0.3%	-6.5%	-8.6%
$(3,4)$	0.1%	-7.3%	-11.9%	-0.6%	-7.9%	-8.7%	0.0%	-9.6%	-9.5%	1.2%	-9.2%	11.3%
$(5,3)$	-0.1%	-5.9%	-7.9%	1.3%	-8.0%	-0.3%	0.0%	-38.3%	-20.8%	-1.4%	-9.3%	-18.8%
$(7,3)$	0.1%	-7.4%	-12.0%	0.2%	-7.2%	-7.7%	0.1%	-9.7%	-4.2%	0.1%	-10.4%	-9.3%

Table 6.3: Layerwise algorithm results on comparing the integration of a pruning step with the regular results.

by at most 2%. The cases of accuracy increase can be explained by the last construction step being too large in some cases, which causes overfitting, and the pruning step removing certain parameters, after which the network size fits the problem complexity better. The parameters as well as the construction effort also show decreases of around 10% in most cases. The observed decrease in total network parameters, combined with the stable test accuracy, suggests the presence of redundant parameters which can be pruned without impacting performance in a negative way. The decrease in CE occurs due to there being fewer trainable parameters in the network, which reduces the number of calculations for the optimizer.

6.1.4 Discussion

We find that the Layerwise algorithm solves all the selected benchmark problems successfully while creating reasonably sized networks. We observed good generalization capabilities without overfitting the data on the difficult Spirals-3 dataset. The integration of the pruning step provided a noticeable decrease in both the final parameters and construction effort. The four tested hyperparameter sets provided similar performance overall, which shows the stability of the algorithm under different conditions. Still, the configurations using three starting hidden nodes and four layer expansions surpassed the three other configurations in terms of final network size and construction efficiency. Thus, we select this setup for the comparisons in section 6.4.

6.2 CasCor Ultra Algorithm Results

This section presents the evaluation results of the CasCor Ultra algorithm on multiple benchmark datasets. The algorithm was evaluated using four different sets of hyperparameters as specified in the evaluation chapter 5. The results are divided into three subsections which cover the performance of the algorithm on training and test data, the final network size and construction effort, and the comparison of the results of the pruning and non-pruning variants of the algorithm. Finally, the results are reviewed and discussed in subsection 6.2.4, where the best-performing hyperparameter set is determined. This configuration is then used for the comparisons with the other proposed algorithms and the state-of-the-art algorithms in section 6.4.

6.2.1 Algorithm Performance

The performance results of the Layerwise algorithm on each dataset are illustrated in table 6.4. They consist of the final network accuracy on the train and test data set for each benchmark problem. Each row represents a set of hyperparameters (*nCandidates* and *correlationThreshold*) that are specified in the HP column. The results show that the target accuracies (displayed next to the dataset name) were reached for all hyperparameter sets. The train accuracy has in general around 1% overshoot with the furthest outlier reaching 2.6%. The impact used hyperparameter set on the final accuracy results seems to be minimal overall.

HP	Spirals-2 (0.9)		Spirals-3 (0.8)		Moons (0.99)		Class (0.85)	
	Train	Test	Train	Test	Train	Test	Train	Test
$(7, 0.5)$	0.918	0.905	0.804	0.741	0.997	0.996	0.857	0.832
$(15, 0.4)$	0.908	0.912	0.821	0.757	0.996	0.997	0.868	0.835
$(25, 0.3)$	0.915	0.910	0.814	0.765	0.996	0.998	0.868	0.840
$(50, 0.2)$	0.913	0.905	0.812	0.773	0.998	0.998	0.861	0.850

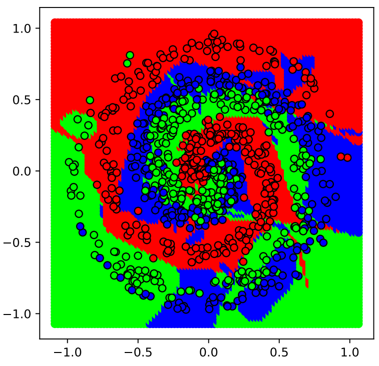
Table 6.4: CasCor Ultra algorithm performance on benchmark problems.

The difference in train and test accuracy values is generally small for the problems where low or medium amounts of non-linearities are needed such as Spirals-2 and Moons. Here, the test accuracy is never further than 1.5% from the training accuracy, which indicates good generalization and little overfitting for these problems. However, the Class dataset and the Spirals-3 dataset show larger gaps between training and testing results. The Spirals-3 problem in particular displays drop off between train and test accuracy of up to nearly 10%. This signifies that the generalization capabilities decrease when the complexity of the problem increases. Also suggesting that the corrective step, which was integrated to specifically target this problem of the original CasCor algorithm, has not fully achieved its aim. The decision boundary for the Spirals-3 datasets displayed in figure 6.2 (a) also supports this claim, as we can see areas of overfitting that stem from the greedy addition of highly non-linear hidden nodes. The number of candidates or the correlation threshold seems to make no notable differences in this behavior.

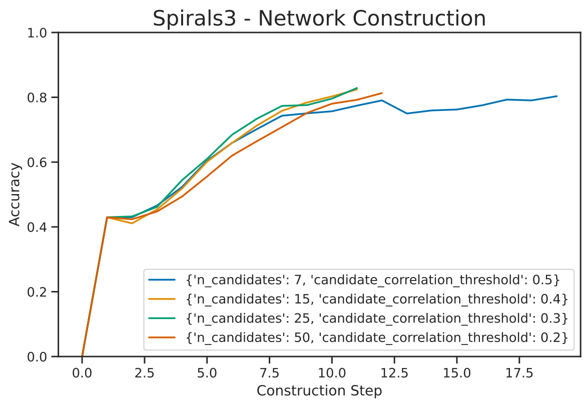
Figure 6.2 (b) provides insight into the training process by displaying the median accuracy at each construction step for each set of hyperparameters. The graph shows that each set of parameters starts at the same accuracy when no hidden nodes are present. Then, the accuracy curves move toward the target of 0.8 in a similar fashion with hyperparameter sets (15,0.4) and (25,0.3) reaching the goal first. Set (7,0.5) seems to take the longest to reach the target accuracy, which is likely due to the low amounts of candidate nodes it can choose from to improve performance. Also, the high correlation threshold will include more redundant nodes in each layer.

6.2.2 Network Complexity and Efficiency

The data displayed in table 6.5 gives insight into to ability of the CasCor Ultra algorithm to produce compact, high-performing networks in an efficient way. The three columns for each dataset display the test accuracy, the total parameters of the final network, and the construction effort (CE) metric.



Spirals-3 decision boundary (a)



Spirals-3 accuracy per construction step (b)

Figure 6.2: The left figure (a) shows an example of the resulting decision boundary of the CasCor Ultra algorithm on the Spirals-3 dataset. Figure (b) depicts the median training accuracy histories for each hyperparameter set on the same dataset.

HP	Spirals-2 (0.9)			Spirals-3 (0.8)			Moons (0.99)			Class (0.85)		
	Acc	Param	CE	Acc	Param	CE	Acc	Param	CE	Acc	Param	CE
$(7, 0.5)$	0.905	385.0	14.87	0.741	1121.5	16.04	0.996	26.0	12.48	0.832	139.0	13.55
$(15, 0.4)$	0.912	126.5	14.61	0.757	487.5	16.06	0.997	16.0	12.60	0.835	103.0	13.98
$(25, 0.3)$	0.910	89.5	14.95	0.765	279.5	16.13	0.998	11.0	13.01	0.840	76.5	14.23
$(50, 0.2)$	0.905	84.5	15.64	0.773	237.5	16.85	0.998	11.0	13.63	0.850	77.0	15.05

Table 6.5: CasCor Ultra algorithm results on benchmark problems showing network complexity and construction efficiency.

The results in table 6.5 show that the final network architecture is highly dependent on the selected hyperparameter set. Looking at the total network parameters for each problem, we see that the higher the number of candidates and the lower the correlation threshold, the smaller the final network will be. The reasons for this are likely that more candidate nodes provide a better option for selecting good hidden nodes and the lower threshold adds fewer redundant nodes to the network. Interestingly, the construction effort results show the reverse trend by needing less effort for a lower number of candidates. This makes sense as the candidate training makes up the bulk of the needed computation and increasing their number also increases construction effort, even if the network has fewer parameters overall. To find the optimal set of hyperparameters we rank the total parameters and the construction effort from one to four and use the average over all datasets. Unsurprisingly, we find that for the number of total parameters, the hyperparameter set (7,0.5) achieves an average rank of 4.0 followed by set (15,0.4) with 3.0. Sets (25,0.3) and (50,0.2) are close with ranks of 1.5 and 1.25 respectively. The ranking for the CE metric is nearly exactly reversed except that set (15,0.4) has an average rank of 1.75. The hyperparameters selected for sets (15,0.4) and (25,0.3) seem to strike a balance between construction effort and the total network parameters.

6.2.3 Pruning Step Results

Table 6.6 presents the results for the evaluation of the CasCor Ultra algorithm with the magnitude pruning step integrated. The columns for each dataset are similar to the ones seen in subsection 6.2.2, except that they show the percentage difference between the respective results of the pruning and non-pruning Layerwise algorithm. Negative values indicate that the metric for the pruning variant decreased in comparison to the non-pruning variant, while positive values mean the opposite.

HP	Spirals-2 (0.9)			Spirals-3 (0.8)			Moons (0.99)			Class (0.85)		
	Acc	Param	CE	Acc	Param	CE	Acc	Param	CE	Acc	Param	CE
$(7, 0.5)$	-0.9%	-77.4%	-16.8%	0.1%	-75.6%	3.8%	1.1%	-48.0%	-2.2%	-0.2%	-51.4%	3.9%
$(15, 0.4)$	0.6%	-60.8%	3.7%	0.0%	-74.1%	-15.5%	0.0%	-37.5%	-0.5%	0.1%	-50.4%	-3.7%
$(25, 0.3)$	0.5%	-59.2%	-4.8%	0.7%	-57.4%	13.5%	0.0%	-27.2%	-0.0%	-0.7%	-44.4%	-4.2%
$(50, 0.2)$	0.7%	-57.3%	28.5%	-0.2%	-60.6%	0.3%	0.1%	-27.2%	-0.1%	-0.5%	-41.5%	4.4%

Table 6.6: CasCor Ultra algorithm results on comparing the integration of a pruning step with the regular results.

The results in table 6.6 that magnitude pruning is a highly effective approach for the CasCor Ultra algorithm to reduce the number of parameters in the network. For most datasets, the pruning step removes around 50% or more from the originally needed parameters. The trend discussed in subsection 6.2.2 is also visible in these results as the hyperparameter set (7,0.5) generally removes the most parameters from the network. Throughout pruning, all test accuracies stay very close to their original values never losing more than 1%, which indicates that the removed parameters really are redundant. The changes in the construction effort seem more dependent on the dataset, as the results vary from needing more or less effort depending on the problem. The reason for this is likely that the pruning step only impacts the construction effort by prolonging or shortening the construction process, which seems to happen somewhat randomly depending on the hyperparameters and the dataset. Pruning only impacts CE in that way, because the bulk of computation is done when training new candidates, for which the pruning step does not make a difference.

6.2.4 Discussion

The findings presented in this section show that the CasCor Ultra algorithm displays good performance and reached all target accuracies on the training data. The algorithm shows good performance, especially on simpler datasets with low to moderate non-linearities. However, for more complex datasets such as Spirals-3, its generalization capabilities are limited. The total network parameter results are highly dependent on the selected hyperparameters as a result of a better selection of nodes due to candidate pool size and the redundancy introduced by the correlation threshold. These two parameters come with a trade-off, as an increased number of candidates also leads to higher computational effort. The integration of the pruning step was highly effective and led to substantial reductions in the final network size across all datasets. Overall, we find that the algorithm excels in creating compact, efficient networks for simple and moderate problems but shows limitations when used on more complex problems.

For selecting the optimal hyperparameter configuration we focus on the architecture-based results as there were no notable differences in the accuracies. Taking into account the ranking of total parameters and construction effort as well as the absolute differences of these metrics we find that only sets (15,0.4) and (25,0.3) are worth considering for selection. Including the small differences in accuracy between the two hyperparameter configurations we determine set (25, 0.3) to be best and thus it is used for the comparisons in section 6.4.

6.3 Uncertainty Splitting Algorithm Results

This section presents the evaluation results of the Uncertainty Splitting algorithm on multiple benchmark datasets. The algorithm was evaluated using four different sets of hyperparameters as specified in the evaluation chapter 5. The results are divided into three subsections which cover the performance of the algorithm on training and test data, the final network size and construction effort, and the comparison of the results of the pruning and non-pruning variants of the algorithm. Finally, the results are reviewed and discussed in subsection 6.3.4, where the best-performing hyperparameter set is determined. This configuration is then used for the comparisons with the other proposed algorithms and the state-of-the-art algorithms in section 6.4.

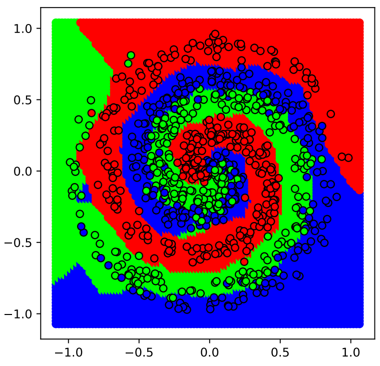
6.3.1 Algorithm Performance

The performance results of the Uncertainty Splitting algorithm on each dataset are illustrated in table 6.7. They consist of the final network accuracy on the train and test data set for each benchmark problem. Each row represents a set of hyperparameters (*nStartingNodes*, *maxLayerSize*, *nUncertainNodes*, and *nReplacementNodes*) shown in the HP column. The results show that target accuracies (displayed next to the dataset name) were reached for all hyperparameter sets. The target accuracy goals were overshot by 1% to 5% depending on the dataset, which is due to larger replacements at the end of the widening of a layer. The overshoot seems to be independent of the used hyperparameter configuration. The algorithm reached perfect accuracy for both the train and test set on the simple Moons dataset. This perfect score is because the random weight and bias sampling during inference used for training seems to provide decision boundaries that are more centered between the classes.

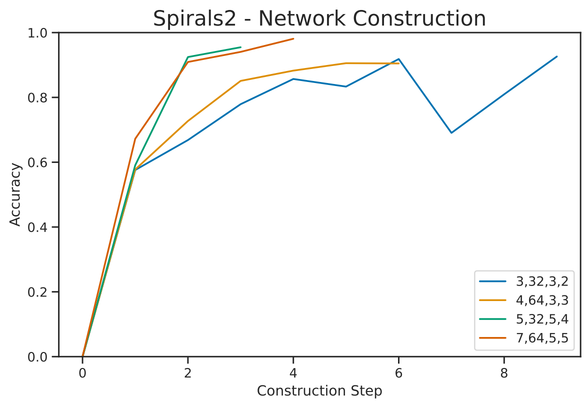
HP	Spirals-2 (0.9)		Spirals-3 (0.8)		Moons (0.99)		Class (0.85)	
	Train	Test	Train	Test	Train	Test	Train	Test
$(3, 32, 3, 2)$	0.958	0.955	0.820	0.813	1.000	1.000	0.895	0.893
$(4, 64, 3, 3)$	0.926	0.923	0.832	0.831	1.000	1.000	0.885	0.883
$(5, 32, 5, 4)$	0.954	0.953	0.824	0.822	1.000	1.000	0.897	0.881
$(7, 64, 5, 5)$	0.969	0.966	0.827	0.821	1.000	1.000	0.888	0.887

Table 6.7: Uncertainty Splitting algorithm performance on the benchmark problems.

Similar to the results of the Layerwise algorithm, we see that train and test accuracies are very close together, indicating good generalization capabilities and low amounts of overfitting. The smooth decision boundary for an evaluation using the Spirals-3 datasets displayed in figure 6.3 (a) supports these claims. The line chart in figure 6.3 (b) presents the training history for the Spirals-2 datasets. We can see that the different hyperparameter sets only vary slightly in the first construction step, which is due to the different numbers of starting neurons. Then, the number of uncertain and replacement nodes comes into play, where for example the first set needs more steps due to a lower number of nodes being replaced. The graphic shows overall that the construction steps are rather large as the target accuracy is generally reached after only four or five steps.



Spirals-3 decision boundary (a)



Spirals-2 accuracy per construction step (b)

Figure 6.3: The left figure (a) shows an example of the resulting decision boundary of the Uncertainty Splitting algorithm on the Spirals-2 dataset. Figure (b) depicts the median training accuracy histories for each hyperparameter set on the same dataset.

6.3.2 Network Complexity and Efficiency

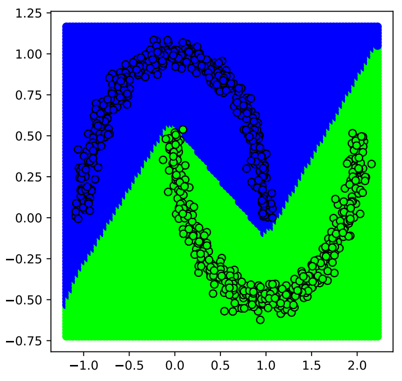
The results in table 6.2 reveal information about the construction efficiency and the network architectures created using the Uncertainty algorithm. Each dataset has three columns displaying the test accuracy, the total parameters of the final network, and the construction effort (CE) metric.

HP	Spirals-2 (0.9)			Spirals-3 (0.8)			Moons (0.99)			Class (0.85)		
	Acc	Param	CE	Acc	Param	CE	Acc	Param	CE	Acc	Param	CE
$(3, 32, 3, 2)$	0.955	214.0	12.74	0.813	1314.0	14.28	1.000	124.0	10.53	0.893	206.0	11.39
$(4, 64, 3, 3)$	0.923	224.0	12.60	0.831	522.0	14.30	1.000	104.0	10.05	0.883	162.0	11.04
$(5, 32, 5, 4)$	0.953	254.0	12.43	0.822	546.0	13.43	1.000	154.0	10.18	0.881	118.0	11.23
$(7, 64, 5, 5)$	0.969	274.0	12.59	0.821	570.0	13.88	1.000	274.0	10.75	0.887	162.0	11.02

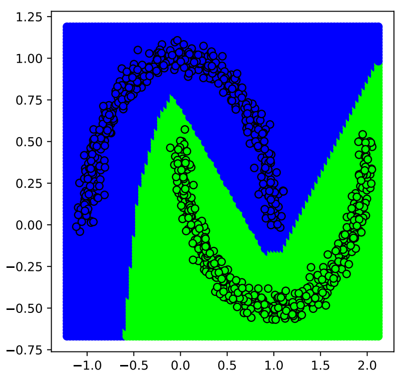
Table 6.8: Uncertainty Splitting algorithm results on benchmark problems showing network complexity and construction efficiency.

We can see that the hyperparameter configuration (3,32,3,2) produces small well-performing networks for the Spirals-2 dataset but struggles with the harder Spirals-3 problem. The rest of the hyperparameter sets create reasonable-sized networks for

Spirals-2, Spirals-3, and the Class problems, which indicates that the stability of the algorithm is largely independent of the configuration. The relatively simple Moons dataset, where only a few nodes would suffice, also needs comparably large amounts of parameters. This is likely because the randomness in the inference takes longer to achieve the high target accuracy of 99%. Although more parameters are needed, a better decision boundary is produced overall as can be seen in figure 6.4. By ranking the hyperparameter sets on the number of total parameters for each dataset we see that set (4,64,3,3) achieves the best average rank with 1.5, followed by set (5,32,5,4) with 2.25, set (3,32,3,2) with 2.75, and set (7,64,5,5) with 3.25. The same ranking for the construction effort has set (4,64,3,3) again at the top with an average rank of 2.0 closely followed by sets (5,32,5,4), (7,64,5,5), and (3,32,3,2). Note that the total parameters include a mean and a standard deviation value for each weight and bias, which doubles the number of parameters in comparison to conventional neural networks. This means that by fixing the weights and biases, using the mean value or sampling from the distribution, the total network parameters can be halved without losing any performance. As the target accuracies were reached for all datasets, we can say that the hyperparameter set (4,64,3,3) has shown the best performance overall including architecture and construction efficiency.



Layerwise Curves decision boundary (a)



Uncertainty Splitting Curves decision boundary (b)

Figure 6.4: The two figures show examples of the produced decision boundaries for the Layerwise algorithm (a) and the Uncertainty Splitting algorithm (b). We can see that Uncertainty Splitting produces better class separation, which is likely due to the randomness used in the inference process.

6.3.3 Pruning Step Results

The integration of a magnitude pruning step into the Uncertainty Splitting algorithm was evaluated for all datasets. The percentage differences for the result metrics between pruning and non-pruning variants are presented in table 6.9. Negative values indicate that the metric for the pruning variant decreased in comparison to the non-pruning variant, while positive values mean the opposite.

The pruning data in table 6.3 shows varying results with hyperparameter set (3,32,3,2)

HP	Spirals-2 (0.9)			Spirals-3 (0.8)			Moons (0.99)			Class (0.85)		
	Acc	Param	CE	Acc	Param	CE	Acc	Param	CE	Acc	Param	CE
$(3,32,3,2)$	-0.6%	18.6%	-0.8%	0.5%	64.2%	60.5%	0.0%	-9.6%	-11.3%	0.0%	-7.2%	-3.4%
$(4,64,3,3)$	0.2%	-8.0%	-4.2%	-0.2%	-13.7%	-19.6%	0.0%	-3.8%	-6.3%	-0.6%	-8.6%	-7.9%
$(5,32,5,4)$	0.0%	28.7%	13.5%	1.4%	14.2%	32.4%	0.0%	-7.7%	-9.5%	0.5%	-10.1%	-12.3%
$(7,64,5,5)$	-0.1%	-7.6%	-9.0%	-0.2%	-2.2%	-3.8%	0.1%	-10.5%	-13.5%	-0.3%	-8.0%	-8.2%

Table 6.9: Uncertainty Splitting algorithm results on comparing the integration of a pruning step with the regular results.

and (5,32,5,4) needing more parameters and increased construction effort for Spirals-2 and Spirals-3 when magnitude pruning is active. The reason for this is likely to do with the smaller layer width of 32 nodes, which sets (3,32,3,2) and (5,32,5,4) have. The removal of some important weights or biases causes the network to have to build another layer and the increase in parameters is caused as a new layer always requires a few construction steps to be large enough to achieve good performance. This would also explain why pruning is effective for the Moons and Class datasets as they are simpler and a new layer is not required in this case. The other two hyperparameter sets show noticeable reductions in parameters and construction effort for all benchmark problems. The reductions generally lie in the area of 5% to 15%. The test accuracies for all hyperparameter sets and all datasets stayed mostly stagnant only showing variations of less than 1% in either direction. This shows the algorithm is able to recover and find a well-performing solution, even if important weights and biases are pruned.

6.3.4 Discussion

The Uncertainty Splitting algorithm performs well on all datasets reaching all target accuracies while generally creating small networks. The smooth decision boundaries and high test accuracies indicate good generalization while not overfitting to the noise in the data. Further, the algorithms showed a unique of producing a more centered decision boundary due to the random weight and bias sampling during inference making it a preferable choice for such use cases. The integration of pruning steps into the Uncertainty Splitting algorithm had varied impacts across datasets showing that this algorithm, or at least certain configurations might not be well suited for a magnitude pruning approach. Finally, with the ability to fix the probability distributions, the total parameters can be halved without impacting performance, therefore improving this approach even further.

The different hyperparameter sets showed similar accuracy levels for each problem but varied in total parameters and the construction effort needed to build the network. By ranking each of these metrics for all datasets we determined that the hyperparameter set (4 starting nodes, 64 maximum layer size, 3 uncertain nodes, 3 replacement nodes) performed best overall. Thus, this configuration is used in the comparisons with the other evaluated algorithm in section 6.4.

6.4 Comparisons and Discussion

This section presents an in-depth analysis of the results of each of the six evaluated algorithms. We compare algorithm performance, network complexity, efficiency, and pruning integration on four benchmark datasets. In addition, we use random reruns of the evaluation with median result metrics to reduce bias. After the results, we provide an extensive discussion where the strength and weaknesses of our proposed algorithms are outlined and the overall performances of the different algorithms are compared. We also explain what the results mean for the field of CoNN and GPNN algorithms as well as the research questions of this thesis. The algorithms were evaluated using its best found hyperparameters with the termination conditions being a specified target accuracy per dataset and the stagnation of the evaluation accuracy. Note that the Uncertainty Splitting algorithm is abbreviated with U-Splitting in the following tables. Further, we mark the top results for each dataset in bold to improve readability in the tables.

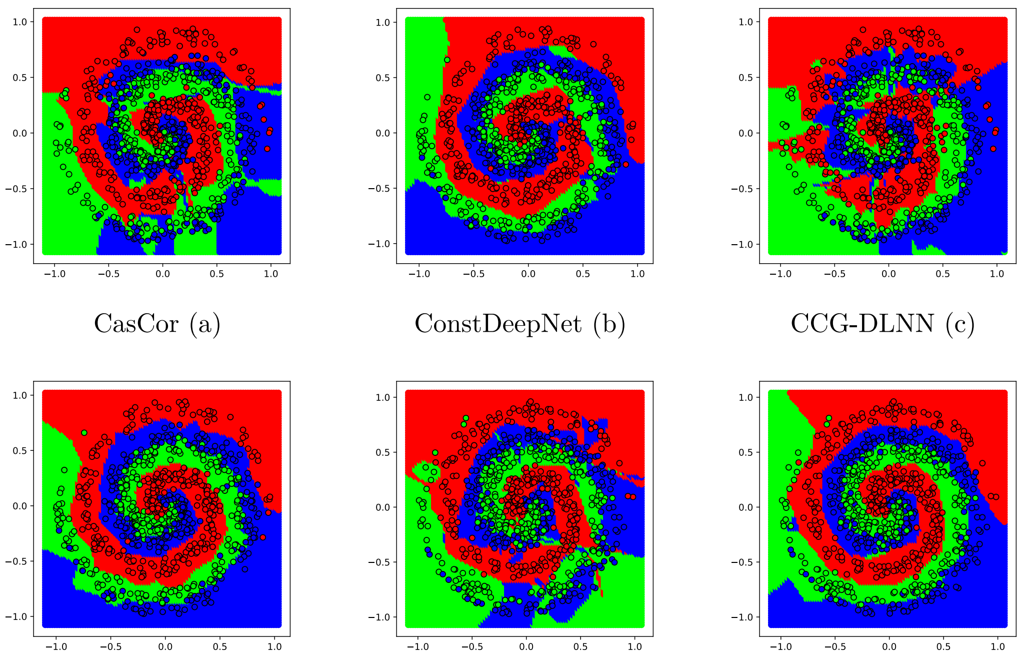
6.4.1 Performance

The performance results of the evaluated algorithms on each dataset are illustrated in table 6.4. The datasets are displayed in four columns, where each of them consists of two sub-columns showing the final accuracy on the training and on the testing data. The results show that each algorithm managed to reach target accuracy (displayed next to the dataset name) for the testing set. While the algorithms such as Layerwise, Uncertainty Splitting, and ConstDeepNet show larger overshoots, the other algorithms remain closer to the specified target accuracy.

Algorithm	Spirals-2 (0.9)		Spirals-3 (0.8)		Moons (0.99)		Class (0.85)	
	Train	Test	Train	Test	Train	Test	Train	Test
Layerwise	**0.938**	**0.942**	**0.833**	**0.846**	0.998	0.998	0.871	0.860
CasCor Ultra	0.915	0.910	0.814	0.765	0.996	0.998	0.868	0.840
U-Splitting	0.926	0.923	0.832	0.831	**1.000**	**1.000**	**0.885**	**0.883**
CasCor	0.909	0.904	0.804	0.776	0.998	0.995	0.854	0.833
ConstDeepNet	0.934	0.915	0.821	0.802	**1.000**	**1.000**	0.869	0.837
CCG-DLNN	0.912	0.903	0.814	0.765	**1.000**	**1.000**	0.853	0.842

Table 6.10: Train and test accuracy results of all evaluated algorithms on the benchmark problems.

All algorithms achieved test accuracies surpassing the target accuracy for the Spirals-2 dataset, with Layerwise and Uncertainty Splitting exceeding it the most. We also see good generalization as the differences in training and testing accuracies are minimal for this problem. The Spirals-3 dataset displays larger gaps between training and testing accuracies, especially for the CasCor-based algorithms, which are the CasCor Ultra, CCG-DLNN, and CasCor itself. This gap indicates overfitting and this is supported by the visualization of the decision boundaries for these three algorithms in figure 6.5 (a), (c), and (e), where we see areas with clear errors in the boundary. The Spirals-3 results for the Layerwise, Uncertainty Splitting, and the ConstDeepNet algorithms show better generalization when looking at the train and test result gap and their respective decision boundaries in figure 6.5 (b), (d), and (f). The simple Moons dataset was solved with nearly perfect accuracy by all algorithms and differences in training and testing results are minimal. Finally, the Class dataset demonstrates a similar trend as the Spirals-3 dataset, where the Uncertainty Splitting and Layerwise algorithms generalized best and show the least drop off in accuracy when comparing train and test results.



CasCor (a) ConstDeepNet (b) CCG-DLNN (c)

Layerwise (d) CasCor Ultra (e) Uncertainty Splitting (f)

Figure 6.5: The figures above display an example of the created decision boundary for the Spirals-3 dataset for each of the six evaluated algorithms.

Overall, we conclude that our three proposed algorithms show comparable performance to the state-of-the-art algorithms on the four selected benchmark datasets. The Layerwise and the Uncertainty Splitting algorithm even indicate better generalization capabilities and less overfitting, especially for more complex datasets. After analyzing multiple results from different CoNN and GPNN algorithms, we can also say that CasCor-based algorithms often lead to overfitting for more complex datasets, even if multiple nodes are present in each layer or corrective training epochs are applied.

6.4.2 Network Complexity and Efficiency

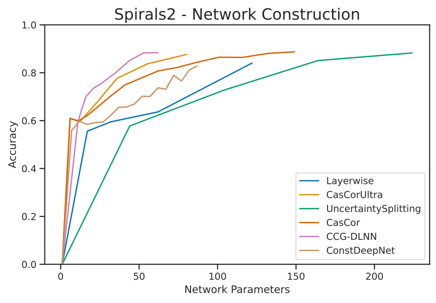
This subsection focuses on the network architecture and construction effort of the evaluated algorithms and thus gives insight into their ability to produce compact, high-performing networks in an efficient way. Table 6.11 displays three columns for each dataset with sub-columns for the test accuracy, the total parameters of the final network, and the construction effort (CE).

The results show that the differences in final network parameters are generally rather

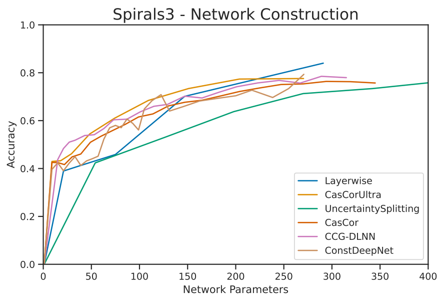
Algorithm	Spirals-2 (0.9)			Spirals-3 (0.8)			Moons (0.99)			Class (0.85)		
	Acc	Param	CE	Acc	Param	CE	Acc	Param	CE	Acc	Param	CE
Layerwise	0.942	122.0	13.23	0.846	291.0	14.57	0.998	62.0	11.79	0.860	70.0	12.21
CasCor Ultra	0.910	89.5	14.95	0.765	**279.5**	16.13	0.998	**11.0**	13.01	0.840	76.5	14.23
U-Splitting	0.923	224.0	**12.60**	0.831	522.0	**14.30**	1.000	104.0	**10.05**	0.883	162.0	**11.04**
CasCor	0.904	158.0	15.47	0.776	372.5	16.78	0.995	**11.0**	12.48	0.833	**64.0**	14.08
ConstDeepNet	0.915	100.5	13.92	0.802	283.5	14.82	1.000	32.0	11.24	0.837	73.5	12.16
CCG-DLNN	0.903	**78.5**	14.05	0.765	576.0	16.96	1.000	**11.0**	11.66	0.842	177.0	13.95

Table 6.11: Results of all evaluated algorithms on benchmark problems showing network complexity and construction efficiency.

small except for some outliers such as the Uncertainty Splitting algorithm on the Moons dataset. By ranking the network parameters for each dataset lowest first we find that the CasCor Ultra algorithm manages to construct the smallest networks with an average rank of 1.75 while the algorithm creating the networks with the most parameters was the Uncertainty Splitting algorithm with a rank of 5.5. However, note that the parameters for Uncertainty Splitting can be halved without degrading performance by fixing the weights and biases to the mean. Our Layerwise algorithm ranks more toward the middle with a rank of 3.75. For problems of low or medium difficulty such as Moons and Spirals-2, the CasCor-based algorithms manage to generate especially small networks, while they perform more around the average for the more complex problems. On the other hand, we find that the construction effort for the CasCor-based algorithms is a lot higher than for the algorithms in general independent of the dataset. This is because the training of the candidate pool at each construction step requires many extra computations in comparison to the other algorithms. When ranking the CE metric we find that the Uncertainty Splitting and the Layerwise algorithms sit more toward the top end having average ranks of 1 and 2.5 respectively, while CasCor Ultra is placed on the lower end with a rank of 5.25.



Spirals-2 (a)



Spirals-3 (b)

Figure 6.6: The figures depict the mean training accuracy with the corresponding network parameters for each algorithm. Figure (a) shows this graph for the Spirals-2 dataset and figure (b) for Spirals-3.

Figure 6.6 gives insight into the training process by showing the mean accuracy and the associated mean number of network parameters throughout construction for each of the evaluated algorithms. We can see that the Uncertainty Splitting algorithm takes larger growing steps which is a main factor in its low construction effort. CasCor Ultra on the other hand uses smaller construction steps enabling easier exploration of compact models.

We conclude from the results in this subsection that our proposed algorithms can compete and even outperform the benchmark algorithms on the evaluated datasets in terms of the smallest number of network parameters and best construction efficiency. The results show that the Layerwise algorithm finds a good balance between generating small networks and providing good construction speed and efficiency, while the Uncertainty Splitting algorithm specializes in fast network construction and CasCor Ultra excels in constructing very compact networks.

6.4.3 Pruning Step Results

In addition to the basic CoNN variants, every evaluated algorithm was also evaluated on the same datasets with a magnitude pruning step integrated. Table 6.12 presents the results for the comparisons between pruning and non-pruning variants of each algorithm. The table is structured similarly to table 6.11 except that each cell shows the percentage difference after integrating pruning.

Algorithm	Spirals-2 (0.9)			Spirals-3 (0.8)			Moons (0.99)			Class (0.85)		
	Acc	Param	CE	Acc	Param	CE	Acc	Param	CE	Acc	Param	CE
Layerwise	0.1%	-7.3%	**-11.9%**	-0.6%	-7.9%	-8.7%	0.0%	-9.6%	**-9.5%**	1.2%	-9.2%	**11.3%**
CasCor Ultra	0.5%	**-59.2%**	-4.8%	0.7%	**-57.4%**	13.5%	0.0%	**-27.2%**	-0.0%	-0.7%	**-44.4%**	-4.2%
U-Splitting	0.2%	-8.0%	-4.2%	-0.2%	-13.7%	**-19.6%**	0.0%	-3.8%	-6.3%	-0.6%	-8.6%	-7.9%
CasCor	-0.9%	-16.4%	-17.1%	-2.4%	-12.3%	12.2%	0.0%	0.0%	0.0%	0.1%	0.0%	9.1%
ConstDeepNet	-0.1%	-8.9%	-10.3%	-2.6%	-5.1%	-5.8%	0.0%	-3.1%	-7.7%	-0.7%	-9.3%	-9.2%
CCG-DLNN	-0.2%	-4.4%	-2.0%	-2.1%	5.3%	3.7%	0.0%	-2.5%	-1.9%	-0.5%	-2.5%	-5.6%

Table 6.12: Results of all evaluated algorithms comparing the integration of a pruning step with the regular results.

The pruning results display overall that the integration of a magnitude pruning step can reduce the number of network parameters and the construction effort without negatively affecting accuracy. This means that all evaluated algorithms generate, at least to some extent, redundant network architecture. The reduction of construction effort results from the weights and biases that are pruning during construction not needing to be optimized anymore which overshadows the extra computational effort performed by the pruning logic itself. However, a more sophisticated pruning algorithm would possibly increase the overall construction effort. Some algorithms such as Layerwise do show increases in CE for certain problems, but this likely has to do with the bad timing of the removal of important parameters, which causes a new layer to be needed to solve the problem. When only examining the number of pruned network parameters, the clear winner is the CasCor Ultra algorithm, which reduced the architecture size by up to 60% while not losing any accuracy. The other algorithms mostly reduce the parameter amount by only 10% or less.

6.4.4 Summary

This section presented and analyzed the results of our three proposed algorithms Layerwise, CasCor Ultra, and Uncertainty Splitting compared to the established algorithms CasCor, ConstDeepNet, and CCG-DLNN. Overall, the investigation and comparisons show promising results in terms of performance, network size, and training efficiency. Each of our proposed algorithms demonstrated distinct advancements in different result metrics which serves as evidence of the potential viability in future research and for real-world machine learning use cases.

When it comes to performance, the Layerwise and Uncertainty Splitting algorithms displayed superior generalization capabilities, especially for complex datasets such as Spirals-3. This is evidenced by the low difference in training and testing accuracies that these two algorithms demonstrate in comparison to the benchmark algorithms. For the Spirals-3 dataset, the benchmark algorithms showed around 5% to 10% less testing accuracy compared to their training accuracy, while Layerwise and Uncertainty Splitting kept similar performance or even improved it slightly. This strengthened generalization is notable as approaches such as CasCor, CasCor Ultra, and CCG-DLNN have the tendency to overfit on the more complex datasets such as Spirals-3. This improvement of the Layerwise and Uncertainty Splitting algorithms suggest a higher potential for better performance in real-world situations, where the ability to find the correct patterns in data while disregarding noise is vital.

The CasCor Ultra algorithm, while showing signs of overfitting for more complex datasets, displayed a remarkable ability for constructing compact networks, reflected in its low parameter count across most datasets. For two out of the four evaluated datasets, CasCor Ultra produced the smallest networks while still providing competitive performance. The parameter counts on the remaining two datasets are also only a few percentage points away from the top-performing algorithm. Although CCG-DLNN and CasCor create similarly small networks on one or two problems, CasCor Ultra is to only algorithm providing minimal parameters across the board. This particular trait could make it valuable in settings where computational resources and memory are limited.

Although the Uncertainty Splitting algorithm constructed networks with a larger architecture size, its construction process proved very efficient due to targeted network extensions using uncertainty and larger construction steps. It used the least effort for all four evaluated datasets while showing high performances overall. The gap to the second-best algorithm in terms of the construction effort, which was either the Layerwise or the ConstDeepNet algorithm, was as big as a whole order of magnitude for two of the four problems and not much smaller for the rest. Additionally, it is important to note that the rather high parameter counts produced by the Uncertainty Splitting algorithm can be halved without losing any performance by fixing the weights and biases to their mean.

The Layerwise algorithm strikes a good balance between creating efficient, small, and high-performing networks. This is shown by its high accuracy values across all datasets while total parameter counts and construction effort were among the best all of the time. Overall, the results of the Layerwise algorithm are very similar to ConstDeepNet, which also displayed good stability with high performance for all evaluated datasets. Additionally, in comparison to all benchmark algorithms, Layerwise showed better generalization and less overfitting supported by low training and testing gaps and good decision boundaries. These good performances across all metrics and top generalization capabilities may make it a useful approach for many machine learning problems in the real world as well as in research.

The results also demonstrate the effects of integrating a magnitude pruning step with each algorithm. We found that the pruning step leads to a noticeable decrease in final parameters in the network without negatively impacting accuracy. The integrated method often also leads to a decrease in construction effort as fewer parameters have to be optimized at each step. The CasCor Ultra algorithm showed especially good pruning results by halving the number of parameters in the network for multiple datasets. The effectiveness of a simple pruning step such as magnitude pruning also suggests that more sophisticated pruning methods would prove even more fruitful. However, these methods would likely increase construction effort noticeably.

In summary, the three novel algorithms proposed in this thesis display potential for advancing the field of constructive neural network algorithms. Each of them shows unique strengths in areas such as generalization, network compactness, construction efficiency, or a balanced combination of these factors. Additionally, the evaluations and comparisons presented in this chapter provide helpful insights and benchmarks for the research in the fields of CoNN and GPNN algorithms.

Chapter 7

Conclusion

This thesis proposes three novel constructive neural network (CoNN) algorithms: Layerwise, CasCor Ultra, and Uncertainty Splitting. Each of them introduces novel ideas or extends existing approaches in the field, intending to advance the knowledge around CoNN algorithms. We evaluated each algorithm on multiple benchmark datasets to assess overall performance, generated network sizes, and construction efficiency. The datasets Spirals-2, Spirals-3, Moons, and Classification for the evaluations were selected as benchmark tasks. To measure the computational cost of construction in a robust way, we additionally developed the novel construction effort (CE) metric. Three state-of-the-art CoNN algorithms, CasCor, ConstDeepNet, and CCG-DLNN, were also evaluated on the same datasets to enable comparisons with existing approaches in the field. Furthermore, we added a simple magnitude pruning step to our algorithms and evaluated variants with and without pruning to give additional insights into the effectiveness of each approach. By using pre-evaluation phases to find well-performing hyperparameters for each evaluated algorithm and random reruns for all evaluations we try to reduce bias in the results for this thesis. To simplify the evaluation process we created an open-source software system that facilitates the automatic evaluation of multiple constructive algorithms on several benchmark datasets using various hyperparameter configurations. The results indicate overall that the three proposed algorithms show potential for advancing the field of constructive neural network algorithms. Each of them shows unique strengths in areas such as generalization, network compactness, construction efficiency, or a balanced combination of these factors. Additionally, the evaluations and comparisons presented in this thesis provide helpful insights and benchmarks for the research in the fields of CoNN and GPNN algorithms.

7.1 Answers to Research Questions

The results presented in chapter 6 provide insights into the performance, behavior, and strengths and weaknesses of our proposed CoNN algorithms. Comparisons with state-of-the-art algorithms contextualize these results in the view of the whole field of CoNN algorithms. Using this knowledge, we can provide precise and informed answers to the research questions stated in the introduction of this thesis.

RQ 1: How well do our proposed algorithm perform in terms of training efficiency, network architecture, and final performance compared to state-of-the-art CoNN algorithms, and under what conditions does improvement occur?

The three proposed algorithms, as well as the three state-of-the-art algorithms, were configured using the best hyperparameter arrangements discovered in the pre-evaluations for this final analysis. Each of these six algorithms was evaluated on all four benchmark datasets to gather metrics such as training and testing accuracy, total network parameters, and construction effort. These metrics enable us to make meaningful comparisons between our proposed algorithms and the state-of-the-art algorithms as well as draw conclusions about their strength and weaknesses.

When comparing pure performance, the Layerwise and Uncertainty Splitting algorithms display higher final accuracies and superior generalization capabilities compared to the other algorithms, especially for complex datasets such as Spirals-3. While the benchmark algorithms showed around 5% to 10% less testing accuracy compared to their training accuracy for Spirals-3, Layerwise, and Uncertainty Splitting kept similar performance or even improved it slightly. Although the Uncertainty Splitting algorithm constructed networks with rather large architectures, its construction process proved very efficient due to targeted network extensions using uncertainty and larger construction steps. The gap to the second-best algorithm in terms of the construction effort was a whole order of magnitude for two of the four datasets. The CasCor Ultra algorithm, while showing signs of overfitting for more complex datasets, displayed a remarkable ability for constructing compact networks, reflected in its low parameter count across most datasets. Its final network sizes were smaller than its predecessor CasCor and even the recent CasCor extension CCG-DLLN. Finally, the Layerwise algorithm proved to find a good balance between creating efficient, small, and high-performing networks. This is shown by its high accuracy values across all datasets while total parameter counts and construction effort were among the best all of the time. The results showed that no other algorithm managed to achieve a better performance on average over all metrics.

Using the results and comparisons discussed above we can answer the research question: Our proposed algorithms demonstrate competitive performance compared to state-of-the-art CoNN algorithms in terms of training efficiency, network architecture, and final performance. Each of them shows unique strengths in areas such as generalization, network compactness, construction efficiency, or a balanced combination of these factors. In scenarios where good generalization capabilities are needed our Layerwise and Uncertainty Splitting algorithms outperform all other evaluated algorithms. In addition, the Layerwise algorithm manages to show the best balance of performance, final network size, and construction effort. The CasCor Ultra algorithm on the other hand shows improvements over the state-of-the-art when it comes to producing very compact networks.

One limitation of this evaluation is that only three state-of-the-art algorithms are considered. Additionally, the best hyperparameter configurations were discovered during pre-evaluations and may not necessarily be the optimal ones.

RQ 2: How does the introduction of a magnitude pruning step influence the training process, network architecture, and performance of our proposed algorithms? What trade-offs exist between purely constructive and grow-and-prune algorithms?

Each of the six algorithms of the final evaluation was additionally tested with an integrated magnitude pruning step. This pruning approach is executed once per construction step and considers low weight and bias values (below a specified threshold) as irrelevant and removes these parameters. The percentage differences for each metric were calculated before and after the pruning step to show the effectiveness of the integration into each algorithm.

The introduction of the pruning step demonstrated substantial effects on the network architecture of the proposed algorithms as well as the state-of-the-art algorithms. By far the most successful integration of pruning was done by the CasCor Ultra algorithm, which showed reductions in final network parameters of up to 60%. The rest of our proposed algorithms as well as the state-of-the-art algorithm all showed reductions in the range of 5% to 20%. These noticeable decreases in final parameters did not affect the final network accuracy negatively with changes mostly staying under the -1% mark with some even increasing in accuracy slightly. The construction effort was also decreased in most cases after the integration of the pruning step. We found reductions of around 10% in general, but a small number of algorithms also showed increases in construction effort for some datasets around the same percentage values. Interestingly, for the more complex datasets, the CasCor-based algorithm all displayed an increase in construction effort while still showing large decreases in the final network size.

After gathering the above-mentioned results and comparisons we can answer the research question as follows: The introduction of a pruning step can, depending on the algorithm, significantly influence the efficiency of training and the architecture of the network, without compromising performance. As for the trade-offs between a purely constructive approach and the grow-and-prune algorithms, we find that a magnitude pruning step is such a lightweight operation that integration shows nearly no impact on computational complexity. Thus, if a useful pruning threshold is selected and the algorithm is suited for this approach, we only see benefits to the usage of such a method.

It should be noted that only a single pruning threshold, discovered when testing thresholds in the pre-evaluation phase, was fully evaluated for each algorithm. This means that other pruning thresholds would possibly yield different results.

7.2 Future Work

Despite the promising results achieved in this work, several potential avenues for future exploration and development still remain. The evaluation of the proposed algorithms, while extensive, could be expanded in numerous ways. More state-of-the-art CoNN algorithms could be evaluated to provide more comprehensive comparisons with our proposed algorithms. The selection of benchmark datasets could be extended by also including real-world datasets to provide a less theoretical perspective on the results. Neural network are generally able to solve both classification and regression problem, however, this evaluation only focused on classification datasets. Therefore, exploring the performance of the algorithm on multiple regression datasets is important.

Another intriguing prospect is the evaluation without target accuracies and only a convergence criterion. While target accuracy provides a simple measure of successfully solving a problem, it may limit the potential for higher performance. Removing it in future evaluations could bring further insight into the capacity for the final performance and the potential trade-offs between accuracy, computational cost, and network complexity.

As for the proposed algorithms, further tuning based on these evaluation results could improve their performance. Investigating more hyperparameter sets could lead to more finely-tuned algorithms and superior network constructions. As currently only a simple magnitude pruning step was implemented, incorporating additional pruning steps such as parameter importance ranking methods or checking performance degradation could prove interesting. Finally, an important extension would be to expand the algorithms to include convolutional layer construction, which could be a powerful extension for image-based tasks.

The open-source software system developed in this thesis has proved to be a valuable tool. Future enhancements could include implementing a wider range of CoNN algorithms, providing more visualizations and result metrics in the GUI evaluator, and enabling GPU computation for speeding up evaluation speed.

References

Literature

[1] Oludare Isaac Abiodun et al. "State-of-the-art in artificial neural network applications: A survey". *Heliyon* 4.11 (2018), e00938 (cit. on pp. 1, 4).

[2] Ajith Abraham. "Artificial neural networks". *Handbook of measuring system design* (2005) (cit. on pp. 4, 7, 8).

[3] Edoardo Amaldi and Bertrand Guenin. "Two Constructive Methods for Designing Compact Feedforward Networks of Threshold Units". *International journal of neural systems* 8 5-6 (1997), pp. 629–45 (cit. on p. 25).

[4] Dave Anderson and George McNeill. "Artificial neural networks technology". *Kaman Sciences Corporation* 258.6 (1992), pp. 1–83 (cit. on pp. 5, 8, 9).

[5] Timur Ash. "Dynamic node creation in backpropagation networks". *Connection science* 1.4 (1989), pp. 365–375 (cit. on pp. 12, 15, 17, 18, 25).

[6] M Augasta and Thangairulappan Kathirvalavakumar. "Pruning algorithms of neural networks—a comparative study". *Open Computer Science* 3.3 (2013), pp. 105–115 (cit. on pp. 13, 14, 33).

[7] Thomas Bäck and Hans-Paul Schwefel. "An overview of evolutionary algorithms for parameter optimization". *Evolutionary computation* 1.1 (1993), pp. 1–23 (cit. on p. 10).

[8] Sarkhan Badirli et al. "Gradient boosting neural networks: Grownet". *arXiv preprint arXiv:2002.07971* (2020) (cit. on p. 36).

[9] Imad A Basheer and Maha Hajmeer. "Artificial neural networks: fundamentals, computing, design, and application". *Journal of microbiological methods* 43.1 (2000), pp. 3–31 (cit. on p. 5).

[10] Dimitris Bertsimas and John Tsitsiklis. "Simulated annealing". *Statistical science* 8.1 (1993), pp. 10–15 (cit. on p. 10).

[11] Monica Bianchini and Franco Scarselli. "On the complexity of neural network classifiers: A comparison between shallow and deep architectures". *IEEE transactions on neural networks and learning systems* 25.8 (2014), pp. 1553–1565 (cit. on p. 7).

[12] Davis Blalock et al. "What is the state of neural network pruning?" *Proceedings of machine learning and systems* 2 (2020), pp. 129–146 (cit. on pp. 13–15).

[13] Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. "Audio Chord Recognition with Recurrent Neural Networks." In: *ISMIR*. Citeseer. 2013, pp. 335–340 (cit. on p. 8).

[14] Neil Burgess. "A Constructive Algorithm that Converges for Real-Valued Input Patterns". *Int. J. Neural Syst.* 5.1 (1994), pp. 59–66. DOI: 10.1142/S0129065794000074 (cit. on p. 25).

[15] Heather A Cameron and Lucas R Glover. "Adult neurogenesis: beyond learning and memory". *Annual review of psychology* 66 (2015), p. 53 (cit. on p. 1).

[16] Dami Choi et al. "On empirical comparisons of optimizers for deep learning". *arXiv preprint arXiv:1910.05446* (2019) (cit. on pp. 9, 36).

[17] Pádraig Cunningham, Matthieu Cord, and Sarah Jane Delany. "Supervised learning". In: *Machine learning techniques for multimedia.* Springer, 2008, pp. 21–49 (cit. on pp. 8, 9).

[18] Xiaoliang Dai, Hongxu Yin, and Niraj K Jha. "Incremental learning using a grow-and-prune paradigm with efficient neural networks". *IEEE Transactions on Emerging Topics in Computing* (2020) (cit. on pp. 13, 14, 26, 29, 33).

[19] Xiaoliang Dai, Hongxu Yin, and Niraj K Jha. "NeST: A neural network synthesis tool based on a grow-and-prune paradigm". *IEEE Transactions on Computers* 68.10 (2019), pp. 1487–1497 (cit. on pp. 14, 26, 28, 29, 33).

[20] Wlodzislaw Duch and Norbert Jankowski. "Transfer functions: hidden possibilities for better neural networks." In: *ESANN*. 2001, pp. 81–94 (cit. on p. 6).

[21] Scott Fahlman. "The recurrent cascade-correlation architecture". *Advances in neural information processing systems* 3 (1990) (cit. on p. 17).

[22] Scott Fahlman and Christian Lebiere. "The cascade-correlation learning architecture". *Advances in neural information processing systems* 2 (1989) (cit. on pp. 12, 13, 15–17, 21, 25, 26, 33, 36, 37, 45, 46, 48, 50).

[23] Scott E Fahlman et al. *An empirical study of learning speed in back-propagation networks.* Carnegie Mellon University, Computer Science Department Pittsburgh, PA, USA, 1988 (cit. on p. 9).

[24] Scott E Fahlman. "fahlman1988faster". In: *Proc. 1988 Connectionist Models Summer School.* Morgan Kaufmann. 1988, pp. 38–51 (cit. on p. 36).

[25] Nader Fnaiech et al. "A modified version of a formal pruning algorithm based on local relative variance analysis". In: *First International Symposium on Control, Communications and Signal Processing, 2004.* IEEE. 2004, pp. 849–852 (cit. on p. 27).

[26] Nader Fnaiech et al. "The combined statistical stepwise and iterative neural network pruning algorithm". *Intelligent Automation & Soft Computing* 15.4 (2009), pp. 573–589 (cit. on p. 27).

[27] Leonardo Franco and José M Jerez. *Constructive neural networks.* Vol. 258. Springer, 2009 (cit. on pp. 1, 34).

[28] F.M. Frattale Mascioli and G. Martinelli. “A constructive algorithm for binary neural networks: the oil-spot algorithm”. *IEEE Transactions on Neural Networks* 6.3 (1995), pp. 794–797. DOI: 10.1109/72.377991 (cit. on p. 25).

[29] Marcus Frean. “A thermal perceptron learning rule”. *Neural Computation* 4.6 (1992), pp. 946–957 (cit. on p. 19).

[30] Marcus Frean. “The upstart algorithm: A method for constructing and training feedforward neural networks”. *Neural computation* 2.2 (1990), pp. 198–209 (cit. on pp. 12, 24).

[31] Stephen I Gallant et al. “Perceptron-based learning algorithms”. *IEEE Transactions on neural networks* 1.2 (1990), pp. 179–191 (cit. on p. 19).

[32] Stephen I Gallant. “Three constructive algorithms for network learning.” In: *Proceedings of the eighth annual conference of the cognitive science society.* 1986, pp. 652–660 (cit. on pp. 12, 18–20, 25).

[33] Stephen I Gallant and Stephen I Gallant. *Neural network learning and expert systems.* MIT press, 1993 (cit. on pp. 19, 24).

[34] Ethan Goan and Clinton Fookes. “Bayesian neural networks: An introduction and survey”. *Case Studies in Applied Bayesian Data Science: CIRM Jean-Morlet Chair, Fall 2018* (2020), pp. 45–87 (cit. on p. 10).

[35] Honggui Han and Junfei Qiao. “A self-organizing fuzzy neural network based on a growing-and-pruning algorithm”. *IEEE transactions on fuzzy systems* 18.6 (2010), pp. 1129–1143 (cit. on pp. 2, 13, 14).

[36] Stephen Hanson. “Meiosis networks”. *Advances in neural information processing systems* 2 (1989) (cit. on pp. 24, 25).

[37] Shayan Hassantabar, Xiaoliang Dai, and Niraj K Jha. “STEERAGE: Synthesis of neural networks using architecture search and grow-and-prune methods”. *arXiv preprint arXiv:1912.05831* (2019) (cit. on pp. 14, 26, 29).

[38] Shayan Hassantabar, Zeyu Wang, and Niraj K Jha. “SCANN: Synthesis of compact and accurate neural networks”. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.9 (2021), pp. 3012–3025 (cit. on pp. 26, 29).

[39] Kaiming He et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE international conference on computer vision.* 2015, pp. 1026–1034 (cit. on pp. 35, 47).

[40] Torsten Hoefler et al. “Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks”. *The Journal of Machine Learning Research* 22.1 (2021), pp. 10882–11005 (cit. on p. 14).

[41] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. *Neural Networks* 2.5 (1989), pp. 359–366. DOI: https://doi.org/10.1016/0893-6080(89)90020-8 (cit. on pp. 1, 4).

[42] David Hunter et al. “Selection of proper neural network sizes and architectures—A comparative study”. *IEEE Transactions on Industrial Informatics* 8.2 (2012), pp. 228–240 (cit. on p. 11).

[43] Kevin Hunter, Lawrence Spracklen, and Subutai Ahmad. "Two sparsities are better than one: unlocking the performance benefits of sparse–sparse networks". *Neuromorphic Computing and Engineering* 2.3 (2022), p. 034004 (cit. on p. 7).

[44] Laurent Valentin Jospin et al. "Hands-on Bayesian neural networks—A tutorial for deep learning users". *IEEE Computational Intelligence Magazine* 17.2 (2022), pp. 29–48 (cit. on pp. 11, 37).

[45] Diederik P Kingma and Jimmy Ba. "Adam: a method for stochastic optimization. arXiv". *arXiv preprint arXiv:1412.6980* (2018) (cit. on p. 36).

[46] Andrej Krenker, Janez Bešter, and Andrej Kos. "Introduction to the artificial neural networks". *Artificial Neural Networks: Methodological Advances and Biomedical Applications. InTech* (2011), pp. 1–18 (cit. on pp. 5, 6).

[47] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". *Communications of the ACM* 60.6 (2017), pp. 84–90 (cit. on p. 29).

[48] Tin-Yau Kwok and Dit-Yan Yeung. "Constructive algorithms for structure learning in feedforward neural networks for regression problems". *IEEE transactions on neural networks* 8.3 (1997), pp. 630–645 (cit. on pp. 1, 11–13, 33, 34).

[49] Yann LeCun, John Denker, and Sara Solla. "Optimal brain damage". *Advances in neural information processing systems* 2 (1989) (cit. on pp. 26, 33).

[50] Yann LeCun et al. "Gradient-based learning applied to document recognition". *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324 (cit. on p. 29).

[51] Jingling Li et al. "How does a Neural Network's Architecture Impact its Robustness to Noisy Labels?" In: *Advances in Neural Information Processing Systems.* Ed. by A. Beygelzimer et al. 2021. URL: https://openreview.net/forum?id=Ir-WwGboFN- (cit. on p. 1).

[52] Zhen Li, Guojian Cheng, and Xinjian Qiang. "Some classical constructive neural networks and their new developments". In: *2010 International Conference on Educational and Network Technology.* IEEE. 2010, pp. 174–178 (cit. on p. 16).

[53] Shaobo Lin, Jinshan Zeng, and Xiaoqin Zhang. "Constructive neural network learning". *IEEE transactions on cybernetics* 49.1 (2018), pp. 221–232 (cit. on pp. 1, 50).

[54] DC Liu and J Nocedal. "On the limited memory method for large scale optimization: Mathematical Programming B" (1989) (cit. on p. 21).

[55] Ioannis E Livieris. "Improving the classification efficiency of an ANN utilizing a new training methodology". In: *Informatics.* Vol. 6. 1. MDPI. 2018, p. 1 (cit. on p. 21).

[56] Mario Marchand, Mostefa Golea, and Pal Rujan. "A Convergence Theorem for Sequential Learning in Two-Layer Perceptrons". *EPL (Europhysics Letters)* 11 (July 2007), p. 487. DOI: 10.1209/0295-5075/11/6/001 (cit. on p. 25).

[57] Enrique S Marquez, Jonathon S Hare, and Mahesan Niranjan. "Deep cascade learning". *IEEE transactions on neural networks and learning systems* 29.11 (2018), pp. 5475–5485 (cit. on p. 50).

[58] Sam McKennoch, Dingding Liu, and Linda G Bushnell. “Fast modifications of the spikeprop algorithm”. In: *The 2006 IEEE International Joint Conference on Neural Network Proceedings*. IEEE. 2006, pp. 3970–3977 (cit. on p. 16).

[59] Marc Mezard and Jean-P Nadal. “Learning in feedforward layered networks: The tiling algorithm”. *Journal of Physics A: Mathematical and General* 22.12 (1989), p. 2191 (cit. on p. 24).

[60] Soha Abd El-Moamen Mohamed, Marghany Hassan Mohamed, and Mohammed F Farghally. “A New Cascade-Correlation Growing Deep Learning Neural Network Algorithm”. *Algorithms* 14.5 (2021), p. 158 (cit. on pp. 12, 15, 21–23, 25, 45–48, 50).

[61] Karim Mohraz. “FlexNet-a flexible neural network construction algorithm”. In: *4th European Symposium on Artificial Neural Networks (ESANN’96)*. 1996, pp. 111–116 (cit. on p. 12).

[62] Pavlo Molchanov et al. “Pruning convolutional neural networks for resource efficient inference”. *arXiv preprint arXiv:1611.06440* (2016) (cit. on p. 13).

[63] Pramod L Narasimha et al. “An integrated growing-pruning method for feedforward network training”. *Neurocomputing* 71.13-15 (2008), pp. 2831–2847 (cit. on pp. 26, 29).

[64] Meenal V Narkhede, Prashant P Bartakke, and Mukul S Sutaone. “A review on weight initialization strategies for neural networks”. *Artificial intelligence review* 55.1 (2022), pp. 291–322 (cit. on p. 32).

[65] Ismoilov Nusrat and Sung-Bong Jang. “A comparison of regularization techniques in deep neural networks”. *Symmetry* 10.11 (2018), p. 648 (cit. on p. 10).

[66] Rajesh Parekh, Jihoon Yang, and Vasant Honavar. “Constructive neural network learning algorithms for multi-category real-valued pattern classification”. *Dept. Comput. Sci., Iowa State Univ., Tech. Rep. ISU-CS-TR97-06* (1997) (cit. on pp. 19, 24).

[67] Rajesh Parekh, Jihoon Yang, and Vasant Honavar. “Constructive neural-network learning algorithms for pattern classification”. *IEEE Transactions on neural networks* 11.2 (2000), pp. 436–451 (cit. on pp. 1, 2, 12, 33, 34, 50).

[68] Astrid G Petzoldt and Stephan J Sigrist. “Synaptogenesis”. *Current biology* 24.22 (2014), R1076–R1080 (cit. on p. 1).

[69] Lutz Prechelt. “Early stopping-but when?” In: *Neural Networks: Tricks of the trade*. Springer, 2002, pp. 55–69 (cit. on p. 10).

[70] Lutz Prechelt. “Investigation of the CasCor family of learning algorithms”. *Neural Networks* 10.5 (1997), pp. 885–896 (cit. on pp. 12, 17, 21, 25, 36).

[71] Lutz Prechelt et al. “Proben1: A set of neural network benchmark problems and benchmarking rules” (1994) (cit. on p. 26).

[72] Douglas L Reilly, Leon N Cooper, and Charles Elbaum. “A neural model for category learning”. *Biological cybernetics* 45.1 (1982), pp. 35–41 (cit. on p. 24).

[73] Martin Riedmiller and Heinrich Braun. “A direct adaptive method for faster backpropagation learning: The RPROP algorithm”. In: *IEEE international conference on neural networks*. IEEE. 1993, pp. 586–591 (cit. on p. 17).

[74] Frank Rosenblatt. “Principles of Neurodynamics: Perceptrons and the Theory of”. *Brain Mechanisms* (1962), pp. 555–559 (cit. on p. 16).

[75] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. *arXiv preprint arXiv:1609.04747* (2016) (cit. on p. 9).

[76] David E Rumelhart et al. “Backpropagation: The basic theory”. *Backpropagation: Theory, architectures and applications* (1995), pp. 1–34 (cit. on p. 16).

[77] Ideen Sadrehaghighi. “Non-Gradient Based Optimization” (Dec. 2021). DOI: 10.13140/RG.2.2.28762.77764/4 (cit. on pp. 10, 19).

[78] Iqbal H Sarker. “Machine learning: Algorithms, real-world applications and research directions”. *SN computer science* 2.3 (2021), p. 160 (cit. on pp. 8, 9).

[79] Jürgen Schmidhuber. “Deep learning in neural networks: An overview”. *Neural networks* 61 (2015), pp. 85–117 (cit. on pp. 7, 8).

[80] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. “Activation functions in neural networks”. *Towards Data Sci* 6.12 (2017), pp. 310–316 (cit. on pp. 6, 35).

[81] Sudhir Kumar Sharma and Pravin Chandra. “Constructive neural networks: A review”. *International journal of engineering science and technology* 2.12 (2010), pp. 7847–7855 (cit. on pp. 11, 34).

[82] P Sibi, S Allwyn Jones, and P Siddarth. “Analysis of different activation functions using back propagation neural networks”. *Journal of theoretical and applied information technology* 47.3 (2013), pp. 1264–1268 (cit. on p. 37).

[83] Natalio Simon. “Constructive Supervised Learning Algorithms” (1993) (cit. on p. 25).

[84] Steen Sjogaard. “A conceptual approach to generalisation in dynamic neural networks”. *Doctoral thesis, Aarhus University* (1991) (cit. on p. 17).

[85] Joan Stiles and Terry L Jernigan. “The basics of brain development”. *Neuropsychology review* 20.4 (2010), pp. 327–348 (cit. on p. 1).

[86] Jose L Subirats, Leonardo Franco, and Jose M Jerez. “C-Mantec: A novel constructive neural network algorithm incorporating competition between neurons”. *Neural Networks* 26 (2012), pp. 130–140 (cit. on pp. 19, 20, 25, 50).

[87] José L Subirats et al. “Multiclass pattern recognition extension for the new C-Mantec constructive neural network algorithm”. *Cognitive Computation* 2.4 (2010), pp. 285–290 (cit. on p. 21).

[88] Richard S Sutton and Andrew G Barto. “Reinforcement learning”. *Journal of Cognitive Neuroscience* 11.1 (1999), pp. 126–134 (cit. on p. 8).

[89] Kenji Suzuki. *Artificial neural networks: methodological advances and biomedical applications*. BoD–Books on Demand, 2011 (cit. on pp. 7, 8).

[90] Tomasz Szandała. “Review and comparison of commonly used activation functions for deep neural networks”. *Bio-inspired neurocomputing* (2021), pp. 203–224 (cit. on pp. 5, 6).

[91] K Mike Tao. “A closer look at the radial basis function (RBF) networks”. In: *Proceedings of 27th Asilomar Conference on Signals, Systems and Computers.* IEEE. 1993, pp. 401–405 (cit. on p. 24).

[92] Jean-Philippe Thivierge, Fracois Rivest, and TR Shultz. “A dual-phase technique for pruning constructive networks”. In: *Proceedings of the International Joint Conference on Neural Networks, 2003.* Vol. 1. IEEE. 2003, pp. 559–564 (cit. on pp. 26, 27, 29, 33).

[93] Alan J Thomas et al. “Two hidden layers are usually better than one”. In: *Engineering Applications of Neural Networks: 18th International Conference, EANN 2017, Athens, Greece, August 25–27, 2017, Proceedings.* Springer. 2017, pp. 279–290 (cit. on p. 18).

[94] Nick K Treadgold and Tamás D Gedeon. “A cascade network algorithm employing progressive RPROP”. In: *International Work-Conference on Artificial Neural Networks.* Springer. 1997, pp. 733–742 (cit. on p. 17).

[95] Claudio Turchetti. *Stochastic models of neural networks.* Vol. 102. IOS Press, 2004 (cit. on p. 10).

[96] Sunil Vadera and Salem Ameen. “Methods for pruning deep neural networks”. *IEEE Access* 10 (2022), pp. 63280–63300 (cit. on p. 13).

[97] Ashish Vaswani et al. “Attention is all you need”. *Advances in neural information processing systems* 30 (2017) (cit. on p. 8).

[98] Najdan Vuković and Zoran Miljković. “A growing and pruning sequential learning algorithm of hyper basis function neural network for function approximation”. *Neural Networks* 46 (2013), pp. 210–226 (cit. on pp. 2, 13, 14).

[99] Qi Wang et al. “A comprehensive survey of loss functions in machine learning”. *Annals of Data Science* (2020), pp. 1–26 (cit. on p. 9).

[100] Bogdan M Wilamowski. “Neural network architectures and learning algorithms”. *IEEE Industrial Electronics Magazine* 3.4 (2009), pp. 56–63 (cit. on pp. 1, 7).

[101] Jihoon Yang, R. Parekh, and V. Konavar. “DistAl: an inter-pattern distance-based constructive learning algorithm”. In: *1998 IEEE International Joint Conference on Neural Networks Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98CH36227).* Vol. 3. 1998, 2208–2213 vol.3. DOI: 10.1109/IJCNN.1998.687203 (cit. on p. 25).

[102] Xue Ying. “An overview of overfitting and its solutions”. In: *Journal of physics: Conference series.* Vol. 1168. IOP Publishing. 2019, p. 022022 (cit. on p. 10).

[103] Ryad Zemouri et al. “A new growing pruning deep learning neural network algorithm (GP-DLNN)”. *Neural Computing and Applications* 32.24 (2020), pp. 18143–18159 (cit. on pp. 2, 12, 26–29, 33).

[104] Ryad Zemouri et al. "Breast cancer diagnosis based on joint variable selection and constructive deep neural network". In: *2018 IEEE 4th Middle East Conference on Biomedical Engineering (MECBME)*. IEEE. 2018, pp. 159–164 (cit. on p. 27).

[105] Ryad Zemouri et al. "Constructive deep neural network for breast cancer diagnosis". *IFAC-PapersOnLine* 51.27 (2018), pp. 98–103 (cit. on pp. 15, 22, 23, 25, 45–47).

[106] Weibao Zou, Yan Li, and Arthur Tang. "Effects of the number of hidden nodes used in a structured-based neural network on the reliability of image classification". *Neural Computing and Applications* 18 (2009), pp. 249–260 (cit. on p. 7).