Dynamic Growing and Shrinking of Neural Networks with Monte Carlo Tree Search*

Szymon Świderski and Agnieszka Jastrzębska

Warsaw University of Technology, Warsaw, Poland

Abstract. The issue of data-driven neural network model construction is one of the core problems in the domain of Artificial Intelligence. A standard approach assumes a fixed architecture with trainable weights. A conceptually more advanced assumption is that we not only train the weights, but also find out the optimal model architecture. In this paper, we present a new method that realizes just that. We show how to create a neural network with a procedure that allows dynamic shrinking and growing of the model while it is being trained. The decision-making mechanism for the architectural design is governed by a Monte Carlo tree search procedure which simulates network behavior and allows to compare several candidate architecture changes to choose the best one. The solution utilizes a Stochastic Gradient Descent-based optimizer developed from scratch to realize the task of network architecture modification. The paper is accompanied with a Python source code of the prepared method. The proposed approach was tested in visual pattern classification problems and yielded highly satisfying results.

Keywords: neural network · changing architecture · training · Monte Carlo tree search · shrinking · growing · Stochastic Gradient Descent.

1 Introduction

Neural network training algorithms development is an essential theoretical and practical problem of Artificial Intelligence. The underlying task is network architecture design. Typical methods assume that a programmer specifies subsequent components of the network and uses an optimization algorithm of choice to find out weight values. At the same time, there is a pressing need to deliver effective methods that relieve programmers from the task of neural network architecture specification. One trend is to use predefined designs, tested by others on some benchmark datasets. A more conceptually advanced scenario is to offer training algorithms that optimize the network architecture during the training procedure. In this scenario, the training algorithm is responsible not only for setting the weights but also for modifying the model design.

The idea of delegating neural model design to an optimization algorithm has been present in the literature domain for some time now. Unfortunately,

* This research was supported by the Warsaw University of Technology within the Excellence Initiative: Research University (IDUB) programme.

up to this day, the practical use of this approach is quite limited. This is first and foremost due to the modest effectiveness of the available approaches. Most of the existing studies, such as the ones of Zhang et al. [16], were delivered for plain feed-forward neural networks. The demands of contemporary data analysis, especially in the field of image classification, are not matched when such models are used. There are some studies, such as the very recent paper by Evci et al. [5], that offer more insights. The aforementioned paper shows an approach when a network is grown/shrunk neuron-by-neuron. Comprehensive theoretical and empirical studies are overall rare.

In light of the facts outlined in the previous paragraph, in this paper, we contribute a novel method for neural network training. We are publishing it as an open-source package. We chose the name *growingnn*. It was uploaded to PyPi. Its description is under https://pypi.org/project/growingnn/. The delivered method uses error backpropagation as a base for weight update. The action of architecture design is carried out by enabling a scheduler that after each K epochs allows the neural architecture to change. The change can be realized by adding or removing a layer of a predefined type from the network. The current implementation covers three types of neural layers: (i) a plain, feed-forward layer, (ii) a convolutional layer, and (iii) a layer with residual connections. These three layer types are typical in contemporary advanced models for visual pattern recognition.

The key novelty of the presented work is the use of Monte Carlo tree search to simulate network performance. We use it to determine the optimal decision with regard to the design change. To achieve that, the neural model evolution strategy is represented as a tree. The network is redesigned in such a way that a change in the structure has a minimal impact on the already-learned weight values. In this manner, instead of offering a trivial wrapper solution that trains from scratch a new model using a library algorithm after an architecture update, we develop a novel optimizer that performs continual (progressive) training and reuse of the already-learned neural connections. Furthermore, the changes to the network architecture concern entire layers, not single neurons.

The proposed approach was empirically evaluated in applied visual pattern recognition problems involving standard benchmark datasets: MNIST [3] and FMNIST [14]. The outcomes of the proposed model were compared with baseline strategies for neural architecture change orchestration: random and greedy approaches. In both cases, the new method involving the Monte Carlo tree search yielded much better results.

The remainder of this paper is structured as follows. Section 2 addresses relevant literature positions in the domain of architecture-changing neural network training methods. Section 3 outlines the theoretical background of our approach. Section 4 shows the results of empirical tests of the new method. Section 5 concludes the paper.

2 Literature survey

In recent years, the topic of dynamic neural network architecture change has attracted noticeable attention. We shall start this discussion by mentioning the method known as GradMax [5]. It is a method capable of growing a neural architecture during the training procedure without costly retraining. The idea is very similar to the one discussed in this paper, but the methods that handle each change are very different from our methods. GradMax operates on the level of a single neuron. In our algorithm, there is a very wide spectrum of possible changes for a network that allows architecture to grow and shrink. GradMax maximizes gradients for new weights and efficiently initializes them using singular value decomposition (SVD). This approach makes new neurons not impact existing knowledge which is contradictory to our method for which neural network has a short period of instability. The idea of changing the structure using gradient information is relatively common in the literature. One of the first of this kind was a model called resource-allocating network [11]. In this method, when a given pattern was unrecognizable, new neurons were added. An analogous idea was published by Fahlman and Lebiere under the name Cascade-Correlation Architecture [6]. Miconi [10] proposed an approach that uses information about gradient value to adjust the number of layers and layers' size. However, his approach works only for residual levels. Neural networks can grow not only by adding new neurons but also by splitting existing neurons. The article by Kilcher et al. [9] about escaping flat areas via structure modification introduces a new strategy of this kind. When the change of loss is slowing down and the network encourages a flat error surface, the proposed method adds new neurons by splitting the existing ones. This work has two important elements in common with our method. The changes to the structure are made when the network's ability to learn decreases and we also believe that their method of splitting the neuron is in a few ways similar to our method that uses quasi-identity matrices. A very important question in the field of neural networks that can change their structure is how much the neural network can adapt to the problem. A study about Convex Neural Networks [1] shows that these networks can adapt to diverse linear structures by adding neurons in a single hidden layer in each step of training, which also forces the network to grow.

3 The method

The proposed algorithm consists of two components. The first component performs weight adjustment. The second component is the orchestrator, which launches a procedure to change the network architecture. The change takes place each K epoch and it works in a guided manner. Its decisions are made based on the outcome of the Monte Carlo tree search. A rough outline of the new routine is given in the Algorithm 1.

The algorithm is fundamentally built upon Stochastic Gradient Descent (SGD) [4]. It relies on low-level computations, avoiding elaborate tools used in some contemporary training algorithms. This simplicity makes it ideal for research that focuses on fundamental machine-learning principles. For us, a model is a main structure that stores layers as nodes in a directed graph structure, it operates on the identifiers of layers. The layer is assumed to be an independent structure that contains information about incoming and outgoing connections. The default starting structure is a graph that has an input and output layer and one connection between those. In each generation, a new layer may be added or an existing layer may be removed. As the structure grows, each layer gains more incoming and outgoing connections. In the propagation phase, the layer waits until it receives signals from all input layers. Once received, the signals are averaged, processed, and propagated through all outgoing connections.

Algorithm 1 Training Algorithm

```
 1: Input: Dataset
 2: Output: Model
 3: Initialization:
 4: SimSet ← Create simulation set
 5: Model ← Create a model with basic structure
 6: for each generation do
 7:     GradientDescent(Model, Dataset, epochs)
 8:     if canSimulate() then
 9:         Action ← MCTS.simulation(Model)
10:         Model ← Action.execute(Model)
11:     end if
12: end for
```

The function $canSimulate()$ called in Line 8 in Algorithm 1 represents a module that is later referred to as *simulation orchestrator*. The orchestrator determines the point in the training procedure when a simulation is executed during the learning process to change a current neural architecture. At the end of each generation, the simulation orchestrator checks if a simulation is needed. The moment at which the simulation is executed is very important because it helps maintain a balance between the exploration and exploitation of potential structures. A model may retain a particular architecture for several epochs, or it may require frequent changes. Too frequent changes may prevent a specific architecture from being fully trained, while infrequent changes may lead to constantly running into local minima and significantly increase learning time, rendering the method inefficient. In the section dedicated to parameter exploration, we examined various approaches, but ultimately, we decided to use a method known as *progress check*. In this method, after each generation, we check whether there has been an improvement in the model's learning. If there is no improvement, then the simulation is run.

In our algorithm, the learning rate plays a crucial role. We implemented a custom modification of the progressive learning rate in line with the work of Schaul et al. [12]. In this approach, the progressive learning rate ensures that the learning rate is very close to zero in the first epoch after the structure change. Thereafter, the learning rate grows to a constant value through the training process. When the maximum constant value of the learning rate is reached, the learning rate slowly decreases before the next action. This minimizes the negative impact of introduced changes on the already learned information in the network.

3.1 Neural architecture design changes

The algorithm draws inspiration from the achievements of ResNet-50 [8] and ResNeXt [15], particularly from the success of residual connections. In the presented algorithm, the model's structure is treated as a graph, in which layers are nodes and connections between layers are directed edges. The structure built from combining residual and sequential connections shapes a directed acyclic graph, which also functions as a flow network [7]. The algorithm allows adding and removing layers from the model without losing the residual structure of layers. All layers operate asynchronously, and signals move through the network in a manner akin to recursion. Such a structure has key properties, as it allows for unlimited network size, ensures that data always flows through the network without supervision, prevents deadlocks, and prohibits cyclic or unnecessary layers.

In general, a network has a tendency to add new layers, which allows the network to grow and learn new features. Changes to the network structure are added in the form of actions. An action concerns essentially either an addition or a removal of a layer. The algorithm generates all of the possible candidate connections for a given type of layer. Each possible connection for a given layer type in structure defines one single action that can be run on the current model.

The current implementation supports four different types of layers.

1. sequential dense layer;
2. residual dense layer;
3. sequential convolution layer;
4. residual convolution layer.

Dense layers can be connected to and from any layer kind as long as a residual structure is preserved. Convolution layers have a specific rule they can only be added after another convolution layer as input. In our experiments, the initial layer is always convolutional as the model is primarily developed to deal with computer vision tasks.

3.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [13] is a search algorithm used in decision processes, particularly in games and simulations. It builds a tree structure by simulating different possible moves, evaluating their outcomes, and expanding the tree. Each iteration in this simulation is divided into four parts.

1. **Selection.** Starting from the root of the tree, one child is selected. The main difficulty is to maintain a balance between exploration and exploitation. This balance is controlled by upper confidence bound applied to trees [2]:

$$a^* = \arg \max_{a \in A(M_s)} \left(Q(M_s, a) + C_s \cdot \sqrt{\frac{\ln N(s)}{N(M_s, a)}} \right). \quad (1)$$

For a given set of actions $A(s)$ generated for a current model structure M_s, the formula selects the action chosen in the child node during the selection. $Q(M_s, a)$ denotes the average result of scores from the rollout phase. $N(s)$ is a number denoting how many times model structure M_s has been analyzed. $N(M_s, a)$ denotes the number how many times action a has been processed for model structure M_s. Initially, the root node consists of a model structure for the current generation.

2. **Expansion.** The algorithm adds new children of a node. It executes all possible actions for a model structure, creating a set of children nodes, each having a different model structure.
3. **Rollout** or playoff. For a given node which is a leaf, in the simulation tree, the algorithm is trying to play a random game. In our adaptation, the rollout executes n random actions on a given model, to simulate future possible changes after a given action. After the rollout, the resulting structure is passed to the score function. The score function trains the resulting structure on the simulation dataset and the resulting accuracy represents the score from the rollout.
4. **Backpropagation.** All nodes are updated according to the score function after the rollout.

MCTS in this implementation is time-limited. After a user-specified time, the simulation returns a single action. The assumption is that the action performed on the current structure should set the stage for subsequent actions to converge toward an optimal structure. In each generation, MCTS identifies changes in the structure toward an optimal configuration. This behavior is analogous to the gradient descent algorithm, which in each epoch determines the change in weight for improvement.
For each structure, the algorithm has seven possible action types to generate

1. **Action: Add a dense sequential layer.** Executing this action adds a sequential layer between two other layers. Unlike residual layers, the sequential layer does not have the ability to determine the initial state of weights in the layer.
2. **Action: Add a dense residual random layer.** A residual layer between two layers does not need to be added between layers that are directly connected. A residual layer can be added between any two layers between which there is a path in the same direction. The initial state of the weights is very important because it determines what will happen to the network's knowledge after adding a new layer.
3. **Action: Add a dense residual "zero" layer.** This action involves adding a residual layer, for which the initial value of all weights is zero. Since the weights are zero, the residual layer should have almost no effect on the network output in the first forward propagation execution after adding this layer. The assumption is that the network may be at a point in space where it cannot move towards the global minimum, as it lacks a dimension in which it could move.

4. **Action: Add a dense residual identity layer.** The idea in adding this layer is that the value of the input layer to this layer is enhanced. The weight matrix in this layer is an identity matrix and the bias values are zero, which means that the output from this layer is the same as the output from the input layer to the newly added layer because it is a residual layer.
5. **Action: Add a sequential convolution layer.** Adding a sequential convolution layer works the same as it was in a dense layer, but convolution layers can be only connected to another convolution layer as input, as output it can be a convolution or dense layer. In the convolution type of actions, there are no predefined weights initial state.
6. **Action: Add a residual convolution layer.** Adding a residual convolution layer works the same as it was in a dense residual layer, with the same constraints as it was with convolution sequential action.
7. **Action: Remove a layer.** Removing a layer cannot change the main principles in the structure, the graph that the layers create must be directed and acyclic. The algorithm allows to removal of any layer except the initial input and output layer of the model. The algorithm may create additional connections to maintain the established structure.

Before executing the method, a default number of neurons in layers, denoted as def_{neu}, can be set. This default value does not force all layers to have the same number of neurons, but rather most of them will align with it. Initially, the output layer will have the number of incoming connections set to def_{neu}, which subsequently influences most layers to adopt this neuron count. This alignment occurs because each added layer adapts to the layers it connects to.

3.3 Training scheme – the complete algorithm

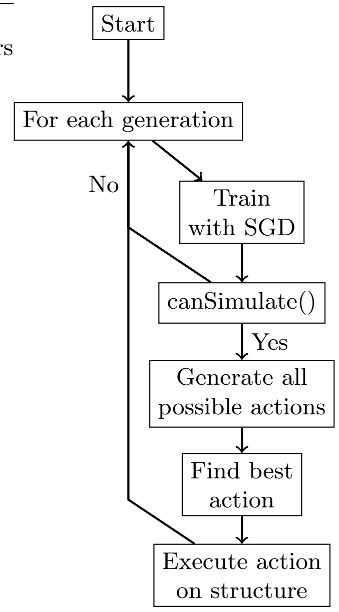
In the presented algorithm, iterations for training the model are divided into generations and then into epochs. In every generation, the structure of the model can change. In every epoch, the model changes its weights. For each generation, the algorithm runs a gradient descent algorithm for a specified number of epochs. While training learning rate changes progressively. For the first and last epoch in a single generation, the learning rate is close to zero. In the middle of training, the learning rate gets to some maximal value, this approach makes it easier for the structure to adapt to new changes in the network. Although the key properties of gradient descent are preserved, there is a big change in data flow in forward and backward propagation. If a layer has more than one input signal, it waits until all information is gathered, after that all input signals are averaged and then processed.

The biggest advantage of the residual structure is that there is no need to supervise the data flow. When a signal is sent to the input layer of the model, it is guaranteed that the signal will travel through the network up to the final output layer and then return to the caller.

The model starts forward propagation by sending a signal to the input layer as a forward signal. Similarly, after calculating the loss, backward propagation starts by sending a signal to the output layer as a backward signal. Forward propagation for a given layer waits until the layer receives inputs from all incoming links. The layer assumes that the received input is in the correct form. When one layer sends the input to another layer, the latter converts it to the desired size. These steps are summarized in the algorithm 2.

Algorithm 2 Forward propagation in dense layers

```
 1: Input: $I_{L_k}$ Input from one incoming connection
 2: Output: Output from all outgoing connected layers
 3: Initialization: $R \leftarrow Null$
 4: $N_I \leftarrow$ Amount of incoming connections
 5: $N_O \leftarrow$ Amount of outgoing connections
 6: inputs.append($I_{L_k}$)
 7: if len(inputs) $< N_I$ then
 8:     return R
 9: end if
10: $\bar{I} \leftarrow \frac{1}{N_I} \sum_{i=0}^{N_I} I_{L_i}$
11: $Z \leftarrow Weights * \bar{I} + Bias$
12: $A \leftarrow ActivationFun(Z)$
13: for each output connection do
14:     $L_O \leftarrow$ Connected outgoing layer
15:     $Q_I \leftarrow Matrix(A.rows, L_O.columns)$
16:     $R_{L_O} \leftarrow L_O.ForwardPropagate((A \cdot Q_I)^T)$
17:     if $R_{L_O} != Null$ then
18:         $R \leftarrow R_{L_O}$
19:     end if
20: end for
21: inputs $\leftarrow$ []
22: return R
```



After all input signals are received, the layer calculates output using learned weights and activation function. This output is adjusted to each layer on the outgoing connection by Quasi-Identity matrices (Q_I). A quasi-identity matrix is created to mimic identity matrices as closely as possible, by resizing the identity matrix to fit a specific size, we can quickly adjust a vector to a different size while maintaining its essential characteristics. It is an efficient way to adjust vector size without losing their fundamental features.

At this point in the algorithm, a layer can send input signals to all connected outgoing layers. This happens by running forward propagation recursively on each of the connected layers. Because the structure of the graph of connections between layers is a directional non-cyclic graph built on residual or sequential connections, we know that the last outgoing layer will return the output from the output layer of the whole model.

3.4 Complexity Analysis

Monte Carlo three search is time limited but it must analyze all actions at a depth of one in the searched tree. For a structure with k hidden layers, it is possible to generate $k * (k-1)/2$ actions to add sequential layers, as this is the maximum number of edges that can be added in a directed graph with k vertices without introducing cycles. The maximum number of actions for adding residual layers is $(k-1)!$, as the highest number of possible residual connections occurs in a graph resembling a path, where each vertex can connect to all subsequent vertices except itself. This means that time complexity for one generation is $O(((k-1)! + k * (k-1)/2) * n * e)) + C_{SGD} = O(k! * n * e) + C_{SGD}$, where C_{SGD} denotes the complexity of training the model using SGD, n is the size of simulation set, e is the number of epochs for training the model in simulation. In our experiments simulation set had 10 examples per class which means that $n = 100$ and the training time in the simulation was 10 epochs.

4 Empirical analysis

4.1 Datasets and empirical setup

The experiments reported in this paper were conducted on the widely used MNIST [14] and Fashion MNIST [14] datasets. MNIST is a dataset of handwritten digits, while Fashion MNIST contains images of various items of clothing. These datasets are suitable for testing both convolutional and non-convolutional models. We have deliberately reduced the size of the initial neural architecture in order to efficiently evaluate the validity of the algorithms. The initial network configuration consists of a single convolutional input layer with a 3×3 filter and a dense output layer with 10 hidden neurons. In addition, we conducted experiments using different random seeds to ensure the robustness and generalizability of our results across different training scenarios.

In what follows, we address the overall quality of the designed algorithm. Furthermore, we inspect the most critical parameters present in the method. In the conducted experiments, the training set was divided into training and testing subsets, with the testing set comprising 20% of the training data. The distribution of images per class was even. More specifically, we used stratified sampling implemented in the *train_test_split* function in the sklearn library. As a result, the testing set for MNIST comprised 8,400 images, with the training set consisting of 33,600 images. For FMNIST, the testing set included 12,000 images, and the training set comprised 48,000 images.

4.2 Classification quality of the new approach

The conducted experiment aimed to empirically validate two hypotheses:

RH1 The first hypothesis states that the algorithm induces neural network growth that leads to an optimal structure.

RH2 The second hypothesis states that the Monte Carlo simulation can identify optimal changes in the network structure.

To verify the first hypothesis RH1, the experiments were conducted on the discussed MNIST and FMNIST datasets, with the initial network structure deliberately reduced to stimulate network growth.

The second hypothesis RH2 was verified by comparing the Monte Carlo algorithm with greedy and random approaches. The random approach randomly selects a change to execute on the network. The greedy approach evaluates each potential change in a single step. The learning process using the Monte Carlo simulation is characterized by a stable and persistent drive to improve quality, each subsequent network change was selected to optimize the model's overall performance. After introducing a change to the network, there are a few epochs during which the network is unstable, but it quickly returns to a stable state and achieves a higher score than before the change. Because the presented data analysis problem is relatively simple and the network has a very small number of neurons, it quickly falls into a state of procrastination. In this state, the network has learned everything it could within its current structure.

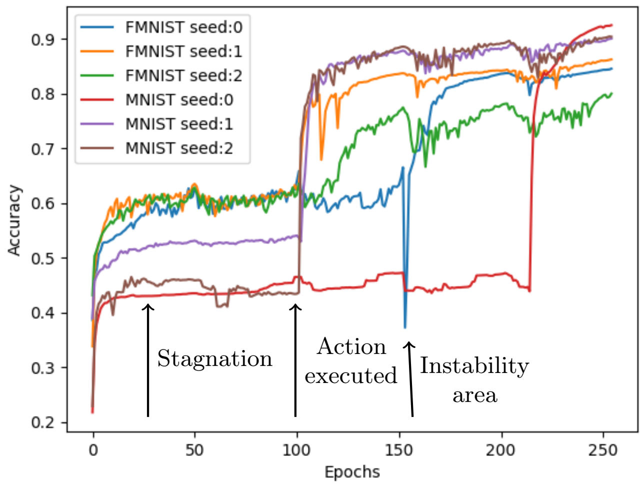


Fig. 1. Neural network learning process with the use of the Monte Carlo simulation algorithm.

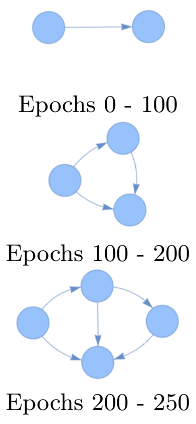


Fig. 2. Graphs representing history of structure for FMNIST seed 2

In Figure 1, we observe the model learning process using our algorithm that employs Monte Carlo simulation. The graphs illustrate classification accuracy, with architectural changes in the neural network introduced every 50 epochs. Epoch number is indicated on the OX axis.

Visible "jumps" (see Figure 1) in the learning process occur shortly after the introduction of a modification to the network. The perturbations and instabilities during learning represent a transition phase in which the learning rate gradually changes in the first epochs after the modification. Each generation lasts for

50 epochs, so the observed phases of stability and procrastination are prominent in full cycles. The structure in each generation learns all possible features it can acquire with the structure available in the given generation. A modification was needed to extend the structure so that it could learn a new feature in the existing data set. Figure 1 shows that when the model was unable to learn more, it fell into the stability region. After the Monte Carlo simulation selects the best action, it allows the network to be better fitted. The results obtained confirm the hypothesis that the algorithm induces a growth of the neural network that leads to an optimal structure.

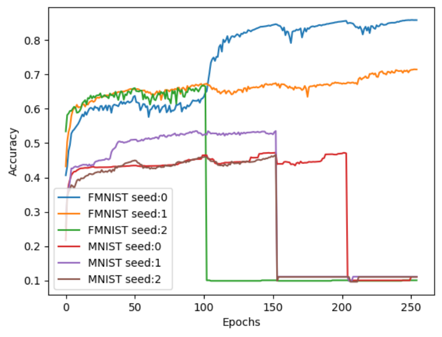


Fig. 3. Ablation study: learning process with a random simulation algorithm.

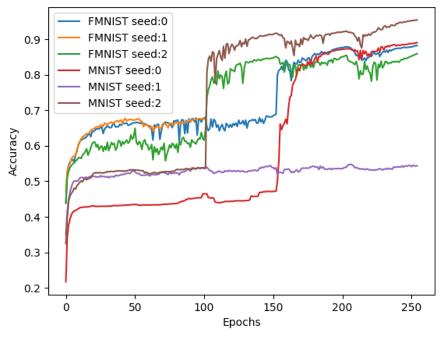


Fig. 4. Ablation study: learning process with a greedy simulation algorithm.

Subsequently, we address the experiments conducted to verify the second research hypothesis concerning the justifiability of using the MCTS algorithm. We test two what-if scenarios. In the first scenario, we replace the MCTS method with a random architecture modification. In the second scenario, we test a greedy search method in place of the MCTS method. The results are illustrated in Figures 3 and 4, depicting the learning progress in terms of accuracy across epochs during training.

Table 1. Classification accuracy reported in a comparative study assessing the impact of a simulation algorithm. The procedure was repeated nine times with different seeds.

	FMNIST			MNIST			**mean**
algorithm	seed 0	seed 1	seed 2	seed 0	seed 1	seed 2	
greedy	83%	67%	82%	86%	55%	91%	77.67%
Monte Carlo	82%	83%	79%	90%	87%	87%	88.35%
random	83%	70%	9%	11%	11%	11%	11.43%

In the plot concerning a randomized scenario (Figure 3), we observe that the learning process is highly unstable. It is possible that the algorithm may eventually reach a well-working structure. However, it is much more likely that the chosen changes will cause a total collapse of the predictive power. Observable sudden drops in the learning process indicate that the selected change prevented further learning.

In Figure 4, we illustrate the learning process for the greedy approach. It shows significantly better results than the random simulation.

Subsequently, let us examine the numerical quality scores of different processing pipelines. These are summarized in Table 1. It becomes apparent that the overall score of the greedy simulation is worse than the score achieved by the Monte Carlo simulation. Changes introduced by the greedy algorithm mostly had a positive impact on the algorithm's performance but did not lead it to achieve structures as good as those obtained through the Monte Carlo simulation.

Since the Monte Carlo simulation was able to look further into the future compared to the greedy simulation, chosen actions had a better long-term impact on the network structure. The Monte Carlo simulation selects the best change in the current generation but also considers subsequent ones, thereby determining the direction of changes in each generation leading to an optimal structure. In contrast, the greedy simulation only analyzes all possible steps in a single generation and does not consider potential future changes.

The random simulation effectively illustrates that not all changes are good for the structure, choosing the best action is crucial for the learning process, emphasizing the necessity of simulation. It is evident that random changes to the network result in the development of a structure that is unfortunately not favorable for the presented problem. This intuitively indicates that there exists an optimal structure for the given problem, and the Monte Carlo simulation is the best for discovering this structure. The obtained results confirm the hypothesis that Monte Carlo simulation can identify optimal changes in the network structure.

4.3 Parameters of the method and their impact on the procedure

The development of this algorithm required some design decisions that were made based on theoretical analyses and empirical results. In this section, we discuss the most important parameters that were fixed in the discussed experiments. Below, we present the results from three experiments aimed at determining key parameters for this algorithm. The first parameter determines the mode of evaluating simulation results, the second parameter dictates the initiation moment of the simulation, and the third parameter governs how the learning rate parameter should work. All experiments in this section were run on the MNIST dataset with three different seeds.

The first parameter to be tested was the mode of operation of the *score function* during the algorithm simulation. The algorithm can use either the *loss* or *accuracy* parameter to score a model after a particular action. The choice of this parameter is particularly important when using Monte Carlo Tree Search. On the one hand, accuracy values are immediately normalized and fit well with the established and effective patterns in Monte Carlo Tree Search. On the other hand, using the loss value as an evaluation criterion has the potential to convey more information about the quality of the model after a specific action has been performed.

In that regard, we inspect and compare the impact of using loss and accuracy as two distinct evaluation criteria. The experiments were conducted on both datasets. They were repeated three times with different seeds.

Table 2. Classification accuracy depending on various score function modes.

score function	**seed 1**	**seed 2**	**seed 3**	**mean**
accuracy	89.5%	85.0%	53.7%	76.0%
loss	44.9%	89.5%	55.7%	63.3%

The experiments have shown that the mean score of the models that used accuracy in grading actions was bigger than that of those that used loss, which is shown in Table 2. We stipulate that it is because UCB1 (UCB stands for Upper Confidence Bounds) in Monte Carlo simulation works better with normalized values like accuracy.

Subsequently, we examined different operational modes for the simulation orchestrator. There are several strategies one can design for this task. We chose to investigate three specific modes: *constant*, *progress check*, and *overfit*.

Table 3. Classification accuracy and the number of changes of network architecture for various simulation orchestrator working modes.

	accuracy				number of simulatons			
orchestrator	**seed 1**	**seed 2**	**seed 3**	**mean**	**seed 1**	**seed 2**	**seed 3**	**mean**
overfit	89.3%	86.6%	84.6%	86.3%	18	18	18	18
progress check	83.6%	88.0%	90.6%	87.4%	10	7	9	8.6
constant	89.9%	88.2%	87.1%	88.3%	18	18	18	18

Table 3 showcases the average classification accuracy on test sets for various orchestrator operation modes. The "overfit" method displayed slightly worse performance compared to the other two. In this mode, the algorithm triggers simulations when there is a high probability of overfitting based on the model's learning history. The "constant" mode conducts a simulation in each generation, while the "progress check" mode verifies if the model has achieved improved accuracy compared to the previous generation; if not, a simulation is initiated. All these methods show promise, yielding similar results. Despite the "constant" method having the best mean score, the "progress check" mode attains the highest maximum score. Consequently, we analyzed the extent of changes induced by these orchestrators. The primary objective of this module is to find a balance between exploiting and exploring the model structure and adjusting the number of changes to the model's learning progress. Table 3 details the number of changes caused by the orchestrator. Both the "overfit" and "constant" orchestrators made changes in each generation. Perhaps the "overfit" mode should be better parameterized for a given problem to be less or more sensitive. The "progress check" made changes in approximately half of the possible generations. We reduced the number of neurons in the layers to close to 10, explaining why adding more layers consistently yielded high scores in all orchestrators. However, the "progress check" orchestrator, with fewer changes, emerged as the most stable and efficient method, making it the likely best choice for future experiments.

Table 4. Classification accuracy depending on different learning rate scheduler modes.

Learning rate scheduler	seed 1	seed 2	seed 3	mean
constant mode	54.3%	83.9%	81.9%	73.3%
progress mode	86.5%	87.9%	85.2%	86.5%

Subsequently, we compared the use of a progressive learning rate with a constant learning rate. Progressive means that the learning rate increases up to a certain point during one generation and then decreases almost to zero before starting the next generation. This happens so that the changes introduced by the algorithm to the model have the least impact on the features learned so far. With a constant learning rate, the same value applies throughout all generations. From the conducted experiments, we inferred that the progressive mode yields higher results.

5 Conclusion

The paper has brought forward a new approach to dynamic neural network training procedures. The proposed procedure relies on a simulation orchestrator that launches an MCTS procedure. The outcome of this procedure is a decision concerning a change in the neural architecture. The addressed solution works on the level of a layer: after each simulation, we may decide to add a layer, remove a layer, or keep the current architecture intact. The detailed formalism was presented for convolutional, plain sequential (dense) layers, and residual sequential layers. The new method, in contrast to the approaches existing in the literature, empowers more flexible model design through the use of a wide variety of neural layers.

The validity of the use of the MCTS algorithm for design was tested in an ablation and substitution study. We have replaced the Monte Carlo simulations with a random decision-making algorithm and with a greedy algorithm. The latter performed substantially worse. The random algorithm was unacceptable.

An indispensable component of the delivered study was the prepared source code. It is openly available as a Python package named *growingnn*. It was uploaded to PyPi: https://pypi.org/project/growingnn/. It contains the implementation of the training algorithm prepared from scratch in Python. We want to underline the scarcity of open-source codes in the domain of dynamic neural topology adjustment methods and we believe that our work would bring practical value to the researchers working in this area.

References

1. Bach, F.R.: Breaking the curse of dimensionality with convex neural networks. Journal of Machine Learning Research **18**, 1–53 (2017), https://jmlr.org/papers/volume18/14-546/14-546.pdf
2. Bagaria, V., Baharav, T.Z., Kamath, G.M., Tse, D.N.: Bandit-based monte carlo optimization for nearest neighbors. IEEE Journal on Selected Areas in Information Theory **2**(2), 599–610 (2021). https://doi.org/10.1109/JSAIT.2021.3076447
3. Deng, L.: The mnist database of handwritten digit images for machine learning research. IEEE Signal Processing Magazine **29**(6), 141–142 (2012)
4. Dogo, E.M., Afolabi, O.J., Nwulu, N.I., Twala, B., Aigbavboa, C.O.: A comparative analysis of gradient descent-based optimization algorithms on convolutional neural networks. In: 2018 International Conference on Computational Techniques, Electronics and Mechanical Systems (CTEMS). pp. 92–99 (2018). https://doi.org/10.1109/CTEMS.2018.8769211
5. Evci, U., Vladymyrov, M., Unterthiner, T., van Merriënboer, B., Pedregosa, F.: GradMax: Growing neural networks using gradient information. In: Proc. of ICLR 2022 (2022), https://iclr.cc/virtual/2022/poster/7131
6. Fahlman, S.E., Lebiere, C.: The Cascade-Correlation Learning Architecture, p. 524–532. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1990)
7. Goldberg, A.V., Tardos, É., Tarjan, R.E.: Network Flow Algorithms. Princeton University (1989), https://www.cs.princeton.edu/techreports/1989/216.pdf
8. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 770–778 (2016). https://doi.org/10.1109/CVPR.2016.90
9. Kilcher, Y., Bécigneul, G., Hofmann, T.: Escaping flat areas via function-preserving structural network modifications. In: Proc. of ICLR 2019 (2019), https://openreview.net/forum?id=H1eadi0cFQ
10. Miconi, T.: Neural networks with differentiable structure. CoRR **abs/1606.06216** (2016), http://arxiv.org/abs/1606.06216
11. Platt, J.: A resource-allocating network for function interpolation. Neural Computation **3**, 213–225 (06 1991). https://doi.org/10.1162/neco.1991.3.2.213
12. Schaul, T., Zhang, S., LeCun, Y.: No more pesky learning rate guessing games. Journal of Machine Learning Research **28**(3), 343–351 (2013), https://proceedings.mlr.press/v28/schaul13.html
13. Swiechowski, M., Godlewski, K., Sawicki, B., Mandziuk, J.: Monte carlo tree search: A review of recent modifications and applications. Artificial Intelligence Review **56**(3), 2497–2562 (2023). https://doi.org/https://doi.org/10.1007/s10462-022-10228-y
14. Xiao, H., Rasul, K., Vollgraf, R.: Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. CoRR **abs/1708.07747** (2017), http://arxiv.org/abs/1708.07747
15. Xie, S., Girshick, R., Dollár, P., Tu, Z., He, K.: Aggregated residual transformations for deep neural networks. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 5987–5995 (2017). https://doi.org/10.1109/CVPR.2017.634
16. Zhang, X., Yang, T., Wang, L., Liu, S., Yan, J., He, Z.: Architecture growth of dynamic feedforward neural network based on the growth rate function. In: 2022 IEEE 11th Data Driven Control and Learning Systems Conference (DDCLS). pp. 1190–1195 (2022). https://doi.org/10.1109/DDCLS55054.2022.9858492