UNIVERSITY OF MACEDONIA
DEPARTMENT OF APPLIED INFORMATICS
POSTGRADUATE PROGRAMME IN ARTIFICIAL INTELLIGENCE
AND DATA ANALYTICS

A NOVEL EVOLUTIONARY ALGORITHM FOR HIERARCHICAL NEURAL ARCHITECTURE SEARCH

A dissertation
by

Aristeidis Christoforidis

Thessaloniki, June 2021

A NOVEL EVOLUTIONARY ALGORITHM FOR HIERARCHICAL NEURAL ARCHITECTURE SEARCH

Aristeidis Christoforidis

Bachelor of Applied Informatics, University of Macedonia, 2019

Dissertation

Submitted in partial fulfilment of the requirements for
THE POSTGRADUATE PROGRAMME IN ARTIFICIAL INTELLIGENCE AND DATA ANALYTICS

Supervisor Professor
Konstantinos Margaritis

Approved by the three-member Examination Committee on ../../2021

Konstantinos Margaritis Ioannis Refanidis Nikolaos Samaras

....

Aristeidis Christoforidis

.....

Contents

Acknowledgements

I would like to thank my supervisor Dr. Konstantinos Margaritis and George Kyriakides for their guidance and assistance in writing this thesis.

Abstract

In this thesis, we propose a new neural architecture search algorithm that performs network discovery in global search spaces. We introduce a novel network representation that organizes the topology on multiple hierarchical levels of varying abstraction and develop an evolution based search process that exploits this structure to explore the search space. Our approach involved a curation system that selects well performing network components and uses them in subsequent generations to build better networks. Next, we investigate how the proposed method performs on different types of data. First, we apply our method on an activity recognition time series dataset and manage to discover a topology with impressive performance. We also test the method on two image classification datasets, Fashion-MNIST and NAS-Bench-101 and achieve accuracies of 93.2% and 94.8% respectively in a small amount of time.

The code for this project can be found in `https://github.com/ArisChristoforidis/Dynamic-Hierarchical-NAS`

Introduction

The recent advances in the compute capabilities of modern hardware have resulted in an increased interest in machine learning based approaches in the artificial intelligence community, especially in the domain of neural networks. Using GPUs, deeper networks with many parameters can now be constructed and trained on more sophisticated datasets to solve harder problems. One cumbersome aspect in neural network design is determining the topology of the network, i.e. what neural layers should be used and how they should be connected to provide adequate performance. Neural architecture search aims to address this issue by providing a set of techniques that automatically find well performing network topologies for sets of data. NAS has successfully been applied to image classification problems, where it managed to build networks with better performance than the state of the art hand built ones. However, other types of datasets, such as time series data have been neglected. For instance, only one [1] other publication examines the effect of NAS on time series datasets. In this work, we develop a new NAS approach that tackles some common problems present in other algorithms in the field and examine how our algorithm works for time series datasets, while simultaneously evaluating our approach in conventional benchmark problems.

Theoretical Background

Neural Networks

An artificial Neural Network (ANN) is a directed network of neurons organized into layers and connected to each other with weighted edges. ANNs operate by receiving an input, propagating it to the different layers where it is transformed using the activation function defined in the neurons of each layer, and producing an output signal. ANNs are optimized through the training process, during which a set of training data is processed by the network multiple times in order for the weights of the neuron connections to be adjusted through back-propagation of the error between the network's output and the ground truth, typically calculated using an distance metric.

A Deep Neural Network (DNN) is an artificial neural network that has more than one hidden layers. Through training, DNNs learn to extract important information from the input data in stages - the first layers of the network learn to identify low level patterns while the subsequent layers use these patterns to identify progressively higher level abstract concepts that are relevant for producing an accurate output.

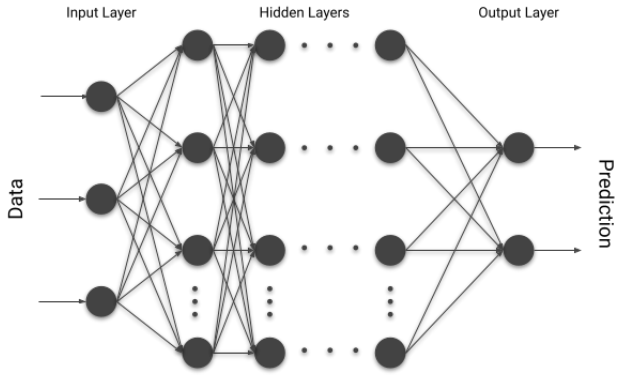


Figure 1: A deep neural network.

Convolutional Neural Networks (CNN) are deep neural networks that use convolutions in the neurons of their hidden layers. These operate by sliding a kernel through the input data and then calculating the activation function result on the transformed input. Convolutional layers usually operate alongside pooling layers, which reduce the size of their input based on a criterion. CNNs have many applications relating to real life problems, such as vision (e.g. object detection, face recognition) and natural language processing (e.g. speech to text, automatic text summarization).

Evolutionary Algorithms

An evolutionary algorithm (EA) is a meta-heuristic, stochastic algorithm that attempts to solve an optimization problem by employing techniques and methodologies inspired by the evolution processes of biological organisms. EAs operate on a population of candidate solutions, whose quality is assessed by a fitness heuristic, and try to iteratively improve these by applying a set of operators that produce new solutions. Population members are called chromosomes and contain information about the parameters of the solution in an encoded format. Usually chromosomes are represented as a fixed length binary number or a string, but many other representations are also possible, depending on the problem at hand. Operators are functions that manipulate members of the population, sometimes producing new chromosomes. One of the most fundamental operators is the mutation operator, which has a small chance to randomly modify a characteristic of a member of the population at each iteration (also known as a generation). Another core operator that defines the subset of genetic algorithms is the crossover operator, which works by selecting 2 members of the population (known as parents) and producing 2 new solutions (known as offsprings) by combining characteristics from the parents. If the offspring solutions are good enough, they are added to the population. The typical flow of an evolutionary algorithm is the following:

Algorithm 1 General evolutionary algorithm

```
1: initializePopulation()
2: i ← 0
3: while i ≤ generations_limit do
4:     applyOperators()
5:     evaluateFitness()
6:     applyFitnessThreshold()
7:     i++
```

Through this process, evolutionary algorithms try to adapt the members of the

population to the environment, i.e. the problem. The fitness function is mainly used in two ways. First, when applying an operator, the chromosome(s) to which it is applied are usually selected at least partially due to their fitness scores. Second, it is often used to simulate the existence of a filter, which acts as a threshold that removes the weakest members of the population, keeping its size constant. This mechanism not only ensures that the memory and CPU demands of the simulation remain the same throughout the execution, but also removes undesirable solution characteristics that should not be potentially reproduced in a subsequent generation by the use of an operator. The evolution loop terminates when either the generation limit is reached, or when at least one chromosome offers a satisfactory solution to the problem. As generations pass, chromosomes should converge to a small set of characteristics that construct the best solutions. Evolutionary algorithms, while offering a versatile methodology which can produce good solutions to hard problems, are very dependent on their parametrization. The hyperparameters of the algorithm vary from problem to problem, and the resulting solutions may converge prematurely or not at all, if these are not assigned correctly.

Reinforcement Learning

Reinforcement Learning (RL) is a machine learning domain in which the training data is not provided by humans, but rather by dynamic simulations of the problem that are generated during training. RL implements intelligent agents that operate on the simulated environment through actions in order to maximize their reward, obtained when they reach the goal. Typically RL is modeled as a Markov Decision Process (MDP): An agent exists in the environment in a state s, defined by its properties (e.g. position in space), equipped with a set of available actions A that allow them to transition to state s' The best action is selected by observing which transition has the best expected reward $r(s, s')$. In classical RL, this information is stored on a transition table which contains transition-rewards associations. The transition table is calculated by simulating the interactions of the agent with the environment many times, until the reward values converge. During this stage, agents have to balance exploitation and exploration. During an exploitation step, agents perform the optimal action as it appears on the incomplete transition table. During exploration, agents select a less than optimal action in hopes of finding new, well performing strategies.

It becomes apparent that the size of the transition table can be a serious problem, since a large number of transitions can be generated from a few states and actions. Calculating all possible transition rewards is often time consuming and likely unhelpful, since only one action per transition will be deemed optimal. To overcome this issue, in Deep Reinforcement Learning (DRL) the transition table is replaced by a deep neural network that learns the best policy by updating its weights. The

agent states are encoded internally, which allows the model to assess states not previously encountered. Even though DRL offers very good results on hard problems, the simulation and training procedures require much time.

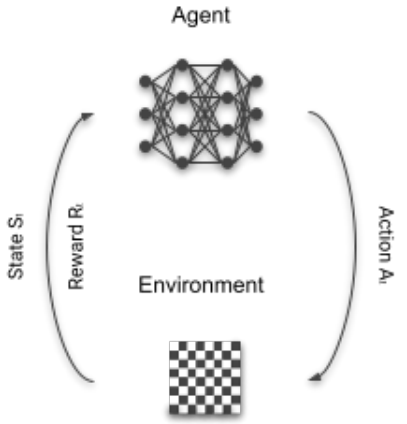


Figure 2: Deep reinforcement learning.

Neural Architecture Search

Neural Architecture Search (NAS) is a research domain within the machine learning field concerned with automated neural network design techniques and algorithms. NAS treats the problem of neural architecture design as a search process where the optimal neural network for a problem must be found in a search space defined by the various allowed network parameters and other properties. NAS can be analyzed into two smaller processes: search space design and optimization methods.

Search space design is the process of selecting which parameters of the network should be constrained or fixed and which ones should allow for variability. A common way of limiting the search space is manually selecting a network size and graph configuration for the layers, performing NAS to find the optimal neuron type for every layer. In a smaller search space, an exploration algorithm may be able to find good networks more easily and terminate faster. However, setting bounds on the search space makes it difficult to ensure that the optimal network for the problem will be included within it. Moreover, since such an approach usually relies on empirical knowledge, it is unlikely that the algorithm will produce novel architectures.

Optimization methods encompass all those techniques that guide the search towards better networks. These include both a transition strategy, i.e. finding a new, unevaluated network from an evaluated one, and evaluation, which is the process of assessing a network's performance. The first is usually an evolutionary or RL algorithm although alternative approaches such as Bayesian Inference or One-Shot methods have also been proposed. The evaluation stage changes depending on the optimization method. While normally network evaluation entails training said network sufficiently and evaluating its accuracy (or any other metric) on the validation set, some approaches perform partial training or even inference to assign a performance value. Since training multiple networks is a very time consuming procedure, such approaches have the potential to provide significant speedup on the algorithm, at the expense of a less accurate evaluation.

Time Series

A time series is a sequence of data ordered by their observation time. There are many real world domains that deal with problems where data is modeled in this fashion. Examples of such problems include weather forecasting, stock price prediction, voice recognition and disease spread modeling.

Following the traditional machine learning taxonomy, problems with labeled data (that is, data where both input and target data is provided) are characterized as either classification or regression problems. In classification, the predictive model must select the target class to which the input data corresponds. In regression, the model must predict a target numerical value from the given input. As far as the number of variables per time step are concerned, a time series can be univariate, meaning that only one variable is tracked, or multivariate, with more than one variables recorded per step. While both input and target data can be either univariate or multivariate, it is very common for problems to offer multivariate input data and univariate target data. When time series contain human interpretable patterns, for example seasonal fluctuation in the values of a variable, the data is said to be structured. Such patterns help humans make sense of the data, but are often problematic when working with machine learning systems. These are some of the core characteristics that must be taken into account when designing predictive models for time series.

Related Works

Search Space

There have been a number of works [2] concerning search space design for NAS. Search space refers to the set of possible networks that can be constructed from a defined set of parameters. There are three main approaches to this problem: global search space, micro search spaces and hierarchical search spaces.

Global Search Space

In a global search space, also known as a macro-architecture search space, network architectures can be formed from graphs of any size. Even though there are no constraints in the graph structure, nodes in the graph should only represent neural layers selected from a curated set of compatible layers for the target problem. It is also a common practice to limit the ranges of the different layer parameters and also the global hyperparameters of the problem. This is demonstrated in DeepNEAT [3] where they show how they are able to converge to an optimal set of parameters and automatically build dynamic networks of different sizes, outperforming the hand built ones. A number of other publications show algorithms operating on global search spaces, such as the works of Elsken et. al. [4] and Byla and Pang [5] that show respectively how network morphism and ant colony optimization can produce well performing networks. However, this approach is generally avoided due to its training complexity, compared to other, simpler approaches that yield similar results.

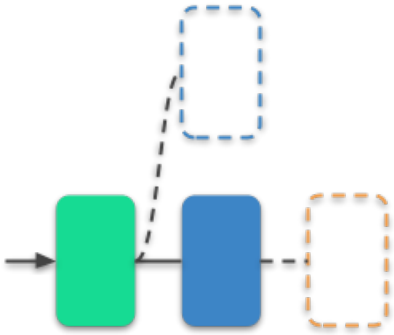


Figure 3: In global search space, the network is formed by iteratively searching for the optimal layer or connection to maximize the network accuracy.

Micro Search Space

In micro search spaces, instead of searching the optimal architecture of the entire network, the search is performed for a much smaller partition. This approach is based on the empirical knowledge gained from architectures with remarkable performance, such as the ResNet [6] and InceptionNet [7] architectures, where the network is made up of repeating blocks or cells of neural micro-architectures. Knowing that good networks consist of repeating blocks, architectures may be searched on a much smaller set of networks by trying to find the optimal block, which is then repeated manually, in accordance to the top architectures for the problem at hand. Zoph et al. [8] operate on such space (named NASNet search space) by defining two types of cells, Normal and Reduction cells that are repeated sequentially many times to form a full network. They show that using this method, the search produces networks that can outperform the current state of the art on the CIFAR-10 dataset.

Another remarkable work related to micro-architecture search is the composition of the NAS-Bench-101 database. Ying et al.[9] define a constrained search space for an Inception block that is repeated to create full networks, and then perform training on all neural models in that search space on the CIFAR-10 dataset. They record the accuracy of each network in a lookup table, and associate it with the micro-architecture that was used to produce it. This allows other researches working on the CIFAR-10 problem with a cell search approach to skip the training of networks and simply lookup their accuracy values on the table (provided that they are searching on the same space). Micro-architecture search is preferred by many researchers due to its ability to dramatically reduce search space size and can be used in a variety of

problems. However, it required substantial external knowledge about the domain of the problem and the general architectures that perform well.

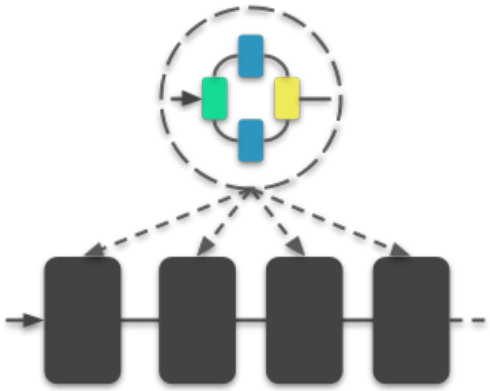


Figure 4: In micro search space, building a network entails finding a cell or set of cells that are repeated on a fixed broader architecture.

Hierarchical Search Space

Hierarchical search spaces attempt to find a middle ground between macro and micro search spaces by conducting an augmented search at both levels. The micro search space contains cells with graphs comprised of neural layers, while the macro search space has graphs where nodes are replaced by cells by the optimization algorithm.In [3], Miikkulainen et al. update their DeepNEAT algorithm by restructuring the search space as hierarchical, on CoDeepNEAT, and apply an evolutionary algorithm to develop better populations in each set. At each evolution step they combine networks from both populations by replacing the nodes at the networks of the macro population with networks from the cell population to build the final networks. Their approach is pretty straightforward, with only two search levels.

Liu et al. [10] present a more dynamic methodology, proposing an algorithm that creates populations on arbitrary levels of up to a certain depth. Their modular approach allows the to tackle the problem of image classification efficiently, iteratively producing populations of performant networks trained on the CIFAR-10 dataset. After training with 200 GPUs for 1 hour they manage to produce a state of the art network with 3.6% error, which is a 97% decrease in search time.

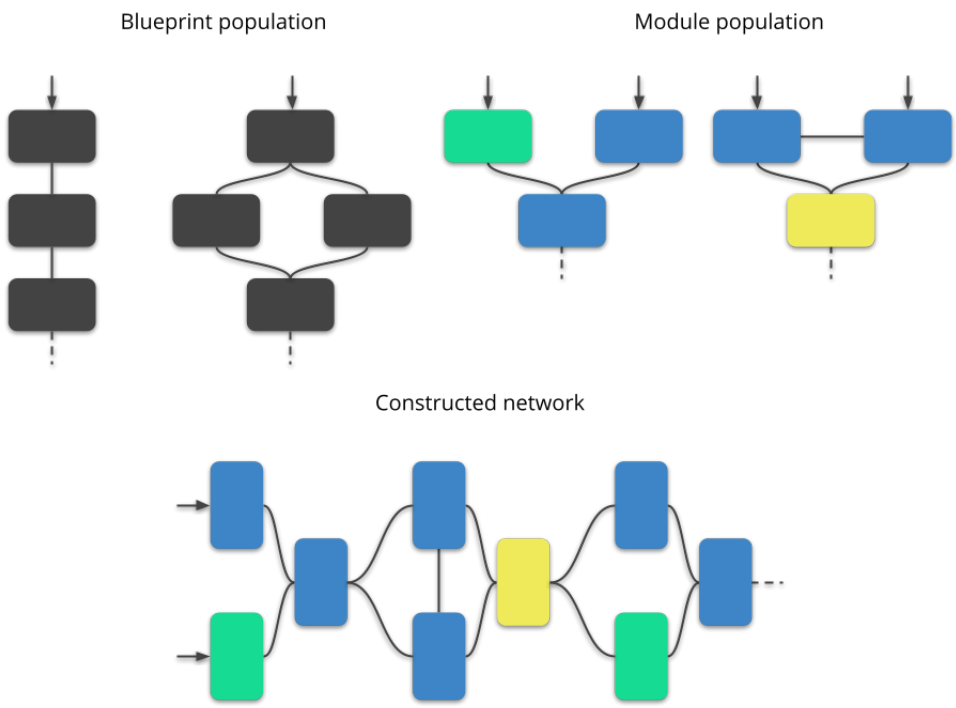


Figure 5: In CoDeepNEAT, networks are built by combining subgraphs from two populations of blueprints and modules.

Optimization Methods

Optimization methods dictate how the search is conducted. The most common approaches are an evolutionary algorithm or reinforcement learning, due to their ability to produce good networks. Lately, however, different publications explore alternative methodologies like bayesian optimization or one-shot methods in an attempts to reduce search times.

Evolutionary Algorithms

Evolutionary algorithms are widely used in NAS to explore the search space. Angeline et al. [11] are one of the first groups of researches to develop an evolutionary algorithm for the purposes of neural architecture search. They initialize their populations by creating graphs randomly, sampling from uniform distributions to set the various network parameters, such as the node count or the initial edge weights. The networks

are then evaluated and the selection operator is applied, which discards the bottom 50% of the networks (using fitness as the criterion). The remaining networks produce offsprings by mutation, classified as either parametric or structural mutations. Parametric mutations add gaussian noise to the weights of the network, while structural mutations add or remove nodes and edges. The mutation severity of a network is controlled by a temperature parameter which is a function of the fitness, and dictates that more drastic mutations be made on networks with poorer performance. They test their method on the William's trigger problem, a regular language inducing task and the ant problem, three progressively more difficult tasks. While these problems are trivial for today's systems, and the solutions to all of them can be modeled with a simple finite state automaton, the performance of their algorithm is nevertheless impressive, managing to construct networks that achieve near-perfect (and sometimes perfect) scores on the datasets of the different problems within 1000 generations.

In 2002, Stanley and Miikkulainen [12] manage to incorporate the crossover operator for NAS, presenting NEAT, a genetic algorithm that can evolve neural topologies. They too want to operate on a global search space and produce graphs of different sizes, so they define a novel genome representation with dynamic length to encode each network. These track the information for the nodes and edges separately. Node genes are quite simple and contain neural layer information. Edges, on the other hand, contain additional information besides their source and destination node indices, which allows the different operators to be applied to the network. Edges can be enabled or disabled depending on the evolution process and the mutations that occur. The most crucial piece of information on the edge genes is the innovation number, an integer which tracks the generation that each edge was added to the network graph. When a node is added to the graph between two other nodes the edge connecting them is disabled and two new edges are added. The innovation number is used to line up the edge genes of the two parent chromosomes in order to apply the crossover operator. Genes with the same innovation number are matched and inherited "as is" by the resulting offsprings. In order to increase the efficiency of NEAT, the population is segmented into species based on network similarity which serve in maintaining local topological innovations and optimizing them by performing crossover between the members of each group. The performance of the algorithm is evaluated by two tasks: the XOR problem, where the neural network, given its input, must predict the output of a XOR gate, and the pole balancing problem, where the network must learn how to correctly apply force to a vertical moving pole of variable height to prevent it from falling flat. The algorithm manages to find optimal topologies that solve both problems by performing much fewer network evaluations than comparable methods ($\sim$ 33.000 evaluations compared to $\sim$ 169.000 and $\sim$ 84.000 evaluations)

The algorithm is extended in their most recent work [3], where the authors present DeepNEAT, a more complete version of their evolutionary algorithm that also opti-

mizes for global training parameters (e.g. learning rate, momentum) and node hyperparameters (e.g. kernel size, number of filters), that in turn get their values from bounded sets. Networks are assigned continuous parameters via gaussian distribution sampling and discreet parameters by random bit flipping. The best networks pass on their parameters to their offsprings which results in good parameter combinations in the evolved networks after many generations. While this extension on the original algorithm does not introduce any new ideas, it updates the methodology so it is applicable to deep neural networks.

Miikkulainen then adapts the algorithm to operate on hierarchical search spaces with CoDeepNEAT, which works on two distinct populations, one for the macro-architecture and one for the micro-architecture, named blueprints and modules respectively. The populations are evolved separately, and their members are combined randomly to construct full networks. The fitness for each blueprint or module is the average fitness of all the networks that they exist in. In a first evaluation, the algorithm succeeds in finding a performant network topology for the CIFAR-10 dataset, which, even though doesn't surpass comparable methods, manages to converge much faster with minimal error difference. Next, using LSTM neurons, they show the performance of their approach by tackling the problem of image captioning using the MSCOCO dataset. After training with 100 GPUs for 37 generations (each generation taking about an hour to complete) the evolved model manages to outperform the hand-designed baseline model.

Overall, evolutionary algorithms are a good fit for NAS. Their open-ended nature allows researchers to easily implement virtually any strategy through the population representation and the operators that are applied to it. They are generally relatively fast (compared to other NAS techniques) and manage to discover optimal or near optimal architectures on most problems. However, these algorithms often require manual tuning of their parameters in order to perform well in different types of problems.

Reinforcement Learning

Reinforcement Learning can be applied to NAS to produce state of the art networks. Instead of using hand made algorithms to automatically build networks, neural architectures are developed by a separate neural network acting as an agent. In its basic form, at each iteration, the agent receives the current network topology (which serves in defining the current state) and expands it through its available actions. These are typically characterized by the different layers and connections that can be added to the network in an attempt to improve it. The agent learns the optimal actions (i.e. modifications) to a network in order to improve its existing state.

Zoph and Le use Deep Reinforcement Learning to design deep neural networks

for the problem of image classification on the CIFAR-10 and ImageNet datasets, with impressive results. In their first work [13], they propose using an RNN as a supervisor agent that receives the output of the last layer and uses it to construct the next layer in a series of 5 steps: first, predict the filter height and width, then the stride height and width and finally the number of filters. This set of basic steps is expanded with two anchor steps, that are used to receive and incorporate the signal of skip connections from previous cells and also from the internal components of the current cell. The agent is trained using the REINFORCE algorithm. When trained, it will design architectures that maximize its expected reward. In order to limit the network size, a layer number threshold is used that increases as training progresses and determines when the agent should stop adding layers to the network. The search is conducted on 800 GPUs, training on 800 network topologies in parallel. When evaluated on CIFAR-10, the resulting optimal network achieves an error rate of 3.65%. The algorithm is also tested on the Penn Treebank dataset, where it exhibits a test set perplexity of 62.4, a 3.6 perplexity improvement over the state of the art, while simultaneously producing a new novel cell architecture that outperforms the LSTM cell on that dataset.

In their succeeding publication [8] they improve upon this algorithm by defining the NASNet search space, which consists of two distinct types of cells, normal and reduction cells, the architectures of which must be optimized. In the full network architecture, the cells are repeated many times to form a deep neural network. The controller network that acts as the agent is again a recurrent neural network that receives the outputs of the two previous cells and outputs a series of 5 actions that dictate how the next cell should be formed. Each action takes into account the previously selected action. The 5 actions are: select a hidden state from the two previous cells, select a second hidden state, select an operation to apply to the first hidden state, select an operation to apply to the second hidden state and finally, select a combination operator to merge the results into one layer and output the cell signal. The operations are selected from a set of neural layers that contains identity, convolutional, average and max pooling layers with different kernel sizes. At every iteration, the agent makes 2 predictions sequentially, one for the normal cell and one for the reduction cell. The agent itself is trained through proximal policy optimization. After training on 500 GPUs for 4 days, the agent manages to produce state of the art topologies that achieve a remarkable 2.4% error rate on CIFAR-10. When the same architecture is transferred over to the ImageNet dataset and trained from scratch, it achieves 82.7% top 1 accuracy and 96.2% top 5 accuracy. The authors also make remarks about the efficiency of the classification models, which offer a 28% computational demand reduction over the previous state of the art model. As for the search process itself, it is 7 times faster than similar previous approaches.

Cai et al.[14] further build upon this algorithm by incorporating a methodology

that allows for the creation of branching networks. They propose using their Net2Net operators [15] that replace a neural layer with an equivalent multi branch cell, a technique which is shown to improve network efficiency. This is done recursively and produces a tree like structure for every layer that is replaced. A network produced from the original algorithm undergoes this secondary procedure which changes one layer type at each step. The new network inherits the weights from the original network and is further trained on the additional layers. The method is evaluated on the CIFAR-10 dataset by replacing layers on pre-trained networks, training them, and evaluating their performance. The replacement is done on the convolutional and pooling layers, which are replaced with branching equivalents. Starting from a small network with 87.07% accuracy, they manage to increase its performance to 95.11% with about 2 days of training on 5 GPUs. On a second experiment on the Street View House Numbers dataset, the algorithm manages to find a state of the art architecture that achieves an accuracy score of 98.17%.

To summarize, even though RL is an effective method for discovering state of the art networks in a topology search space, it has very high computational costs that stem from the need to train not only the topologies, but also the agent. There is also the need for manual restriction on the properties of the model (mainly its size) as it is expected of the agent to keep adding layers in an attempt to increase the network performance. Finally, even though the resulting networks usually outperform those built by other methods, the performance difference does not justify the difference in resource demands.

Bayesian Optimization

Bayesian Optimization (BO) in the context of NAS typically refers to methods that attempt to speed up the underlying algorithm by employing an estimation model to replace the processes that are time consuming. In most cases, the high temporal costs stem from the need to sufficiently train and evaluate the candidate models on the target dataset.

Liu et al. [16] propose using an LSTM model to estimate network performance instead of conventional training on the NASNet space in order to discover the optimal architecture faster. In order to train this surrogate network, they modify the cell architecture generation constraints as following: starting from 1, they set the maximum number of blocks per network. At each iteration, this number is increased by 1. This means that initial networks will be small and easy to train. The surrogate model uses a population of candidate networks with their actual accuracies computed to train. At each subsequent iteration, one extra block is added to each candidate network via the RNN controller. The network performance is estimated by the LSTM model, after which the K most promising cells are used to build, train networks, and update

the weights of the surrogate model. When evaluating the algorithm on the CIFAR-10 and ImageNet datasets, the algorithm manages to achieve state of the art error rates (at the time of publication) of 3.41% on CIFAR-10 and 74.2% (top 1) and 91.9% (top 5) on ImageNet. The use of the surrogate network also contributed to a 8x speedup and 5x efficiency increase over the vanilla RL approach.

Lopes and Alexandre [17] employ BO along an evolutionary algorithm in their HMCNAS algorithm, which evolves populations of hidden markov chains that are used to produce deep neural networks. In their implementation, each hidden markov chain (HMC) is an acyclic graph where the nodes are layer types and edge weights contain the probability of the source node having the destination node as the next layer. They create an initial population by making the HMC graphs for a selection of 34 handmade architectures, such as the ResNet, DenseNet, MobileNet v2 and others, each model trained for 1 epoch. The evolutionary algorithm keeps the best 15% of the networks intact, while the rest of the networks are replaced by new ones generated using BO as following: starting with a default input layer, a random HMC is selected and a second layer is connected to the input, sampled based on the transition probabilities for that chain. The process is repeated until a new hidden markov chain is built. The corresponding model is formed and trained partially to get the fitness of the HMC. The algorithm manages a test error of 9.41% on CIFAR-10 with just 0.24 GPU days (evaluated on a single GTX 1080) of training, which is quite a bit lower than comparable NAS methods.

Bayesian optimization techniques have the potential to speedup otherwise low efficiency algorithms in NAS. Incorporating such methods in the standard neural architecture search algorithms can potentially speed up the convergence times, aiding in the faster discovery of good networks. However, the speedup that these methods offer often results in decreased accuracy in the best found network, compared to that of other methods, such as reinforcement learning.

One-shot Methods

One-shot methods operate on the principle that different network topologies may share weights, since the inner layers of networks are used to perform generic feature extraction. The idea typically involves building a hyper-graph containing every possible network topology using a set of layers, and then extracting select architectures by sampling edges along a path from the input node of the hyper-graph to the output node. Common edges between two paths translate to layer connections that have shared weights in the two corresponding networks, and thus only need to be calculated (trained) once. This form of weight sharing allows the topology evaluation procedure to exploit the training of previously examined networks and reduce training times on the newer networks by only training the weights of the layers that are connected with

new edges on the hyper-graph for which the weights have not yet been calculated.

Pham et al. [18] show the effectiveness of the approach by applying a straightforward one-shot methodology on the NASNet search space, which was known to have notoriously high resource demands when explored with deep reinforcement learning methods. Their proposed approach utilizes a controller LSTM network acting on the directed acyclic hyper-graph of topologies for the convolutional and reduction cells. The search process consists of two alternating training steps: training shared weights in the DAG and training the controller. In the first step, a set of child models are sampled from the hyper-graph and trained in order to calculate the weights for the parameters of the models. The value of shared weights should be updated using the Monte Carlo estimate, which calculates the average of the gradient weighted by the loss of each model using the weights. However, the authors find that the weights in the hyper-graph can be set by training any single network that incorporates the corresponding connection, removing the need to sample a set of networks in order to get more general values. In the second step, the weights of the LSTM controller are trained using the REINFORCE algorithm in order to maximize the expected reward, which is a function of the performance of the selected model on the validation set. The LSTM controller learns to build networks by sampling paths on the hyper-graph that lead to progressively better performance, without having to train each network from scratch. Using a single GTX 1080Ti GPU, they manage to find a performant network for the CIFAR-10 dataset, achieving a 3.54% error which can be further reduced to 2.89% if the training data is augmented wit cutout regularization. Even though this is not a state of the art score, it is very close to the 2.65% state of the art error, with the approach being about 1000 times faster. Similarly, on problems such as the Pen Treebank dataset, the algorithm manages a state of the art 55.8 test perplexity by designing custom RNN cells using a DAG with neural layers and operators.

Bender et al. [19] suggest that there is little need for complex systems for selecting well performing networks from a hyper-graph. While studying the effects of weight sharing on the search process, they come to the conclusion that through random network sampling, the weights in the hyper-graph can be calculated and good networks are bound to be discovered. Training the best performing networks in the one-shot context from scratch results in competent models with weights that better suit the topology. Training with 16 P100 GPUs for 80 GPU hours results in an average accuracy of 95.9% on the CIFAR-10 dataset, using about 37.5% less parameters than comparable models.

Liu et al. [20] propose DARTS, a one-shot methodology that relies on relaxation of the search space to discover architectures fast. Instead of selecting a single neural layer to be used to combine a set of edges in the hyper-graph, a softmax outputs the weight of every available neural operation. This essentially leads to the training of a hyper-network comprised of all possible topologies for the problem. The weights of

the softmax operations are updated iteratively until convergence. When this process completes, the edges with the biggest weight are preserved, defining the final network, while edges with small weights are pruned. They evaluate their method on CIFAR-10 with a small search space in order to design convolutional and reduction cells for NASNet and manage a 2.76% error with a cost of 6.5 GPU days.

In conclusion, one-shot methods offer a way to deal with the high computational demands of training multiple networks during a topology search at a cost. Sharing weights between topologies may not accurately reflect the performance of the architectures using the weights, but it can be used as a heuristic to rank them. After the search process is complete, the topology with the best relative performance must be trained from scratch in order to produce a competitive network. The main disadvantage of the one-shot approach is the restrictions imposed on the search space by the hyper-graph. As the number of nodes (neural layers and operators) at each "level" of the graph increases, the graph naturally becomes more dense, which in turn leads to high memory demands for the saving of the shared weight matrices. This can be mitigated if some edges between nodes are removed based on a set of rules, which shrinks the search space. Using the hyper-graph also means that only networks up to a certain size can be formed, as there is no way to design topologies that exceed the length of the biggest path in the hyper-graph. These problems are evident by reading through the literature on one-shot methods, where researchers often work on micro-architecture topologies with a relatively small search space to achieve results.

Theoretical Description

Goals

It is evident that NAS is a very computationally demanding field. Discovering and evaluating an architecture is a slow process that often requires specialized infrastructure in order to be applied to problems. The works of many teams of researches attempt to work around the main bottlenecks by employing a variety of methods. The most common approach is to shrink the search space by imposing limitations on the size of networks and the variety of their layers, reducing the expressiveness of the design algorithm, but at the same time simplifying the search process. This is usually achieved indirectly, by applying NAS to micro search spaces and building modular networks comprised of repeated architecture blocks. Using empirical knowledge to conduct search in a restricted search space is an effective technique that works on many problems, but is also limited in its ability to produce extraordinary results. The vast majority of discovered networks will have similar performance, while the best topologies of each iteration will deviate slightly from the mean. Also, it can only be applied to domains where prior knowledge regarding good network macro-architectures are available.

Another interesting idea is to replace the conventional network scoring, done through training and evaluating each architecture, with a predictive model, that can produce an estimation of the performance of the model in a fraction of the time it would take to evaluate it properly. Such systems are good additions to most NAS algorithms, but their inclusion translates to yet another component that needs to be configured and evaluated.

Proposed NAS techniques tackle the problems of the field in wildly different ways. The key to designing an effective neural architecture search algorithm is to define a rigid set of goals that should be achieved with it. Naturally, the first goal of any NAS method is to discover relatively fast, with low computational costs. For this reason, an evolutionary algorithm is is used as a base though which networks will be produced as members of a population. Evolutionary algorithms have a very low overhead cost for producing solutions and have been proven to be effective in designing good networks.

They are also very versatile, allowing for easy and unrestricted network manipulation. In comparison, RL methodologies perform better in finding state of the art networks, but also require much more time and resources for training the supervisor network. The second goal that was defined for the approach presented in this thesis is to allow the search process to discover new network patterns that are not necessarily based on empirical knowledge. Working with NAS at the cell level is shown to be effective in other works; however the resulting architectures provide little knowledge about the general rules of what makes a network effective. Artificial constraints guide the discovery process towards a certain direction that the researchers believe contain the optimal network for a problem, but there is no way to be certain about the fact unless the entire search space is searched. On the other hand, global search spaces can contain an infinite number of networks, which can hinder the algorithm by offering too many choices in the search. NAS on global search spaces has been successful in the past, but better and cheaper approaches exist. Hierarchical search spaces are a decent middle ground, providing a structured way to explore a large search space with few compromises. Most approaches work on two hierarchical levels, building full networks by combining cell architectures with general macro architectures. The proposed method is based on hierarchical search, performed on an arbitrary number of levels. The approach is most similar to the work of Liu et. al. [10] where instead of using two levels, they define N levels on which topologies are evolved. Here, networks use a tree-like representation that allows them to have a variable number of levels. This allows for the third goal of the algorithm to be satisfied: find the intermediate architecture that are beneficial to the performance of the topology on the given data. Typically seen in cell search spaces, researchers try to find good configurations of layers that can be repeated as is, and increase network performance. By studying architectures at higher levels of abstraction, it is possible that new abstract blocks be discovered that are beneficial to the network overall. Studying how different blocks of layers can be combined to maximize network performance helps in the advancement of the field of deep neural networks.

Algorithm Overview

By combining an evolutionary approach with a novel network representation, an unrestricted, iterative network architecture search algorithm that can produce deep neural networks of increasing complexity and performance may be built. The evolutionary process operates on a population of candidate network architectures that start simple and are expanded through mutation. The algorithm does not use the crossover operator for producing new offspring networks - even though it would be technically possible, it is very hard to implement a system compatible with the network repre-

sentation that would guarantee an increasing trend in the fitness of the population.

Candidate topologies are initialized by creating a random graph and replacing each node with a neural module sampled from a notable modules list. The neural module is the fundamental abstraction unit for the algorithm. A neural module can either be a neural layer, such as a convolutional or pooling layer or a hierarchical graph, as explained in the next section. The combination of these two terms into a single concept allows for the representation of networks as polymorphic structures that can be easily manipulated with the mutation operator. In fact, the whole candidate topology can be represented as a neural module and is treated as such.

After initialization, the iterative process of the algorithm executes and proceeds to evolve the population for a predefined number of generations. Each iteration begins by applying the mutation operator to a number of members in the population. Mutating networks changes their structure by adding nodes and edges to their graph. In similar works, such as Mikkulainen et. al. [12] [3] the algorithm makes a single change to the topology per mutation, by adding or removing a neural layer or a connection. This is not the case in this work; at each mutation step, the network can change dramatically and have multiple nodes and edges added to it. Another difference between the two algorithms is that while CoDeepNEAT the option to remove nodes and edges exists (in the form of disabling them in a candidate network), here mutation only increases the size of the networks. This convention is set like that because it is expected that larger networks perform better that small ones. However, a large network may have inferior performance due to its topology, at which case it will be removed at the appropriate stage.

Members of the population that are new or have been mutated at the current iteration must be evaluated. This means that the corresponding networks have to be constructed, trained and tested on the validation dataset. Training the candidate architectures is an integral component of most NAS algorithms. The training procedure serves in assessing the performance of each topology which is then utilized by the iterative process in different ways. For example, in a genetic context, the performance of each network can be used in the selection operator to determine which members will be used to produce new candidate offspring topologies. Alternatively, in reinforcement learning based NAS publications, the performance of each network is usually used to train the controller network by incorporating the accuracy of the produced network in the loss function and backpropagating the error on the controller's weights. Training populations of networks is a slow process, and recent publications often replace the procedure entirely by opting to use a predictive model. Usually, the predictive model is another deep neural network that receives a topology as a graph, calculates an internal embedding (in order to support graphs of different sizes) and outputs a predicted accuracy score. That model is trained by providing it with topology graphs and architectures that have been trained conventionally, with the explicit

purpose of providing training data for it. It is worthwhile training the predictive network in conjunction with the rest of the NAS process, feeding it sampled models from each step so it can continuously improve its accuracy as the process progresses, which also removes the need for the existence of a scored topology dataset at the start of the algorithm.

While it is clear that an accuracy estimation model is effective, such a component is not used in this work. Instead, each model is trained partially for a number of epochs determined by a complexity heuristic. The complexity refers to the size of the graph in terms of number of nodes, i.e. the total numbers of neural layers in it. For a given graph $G = (V, E)$ the complexity heuristic is defined as:

$$complexity = \frac{|V|}{C} \tag{1}$$

where C is a constant positive integer, typically between 1 and 4. A network with more neural layers is bound to be more complicated and require more training than a smaller network in order for the algorithm to acquire a representative accuracy score. The constant C can be adjusted as needed to control the amount of additional training that should be performed for every extra neural node. After the complexity of a topology has been calculated, the number of training epochs is calculated as follows:

$$training\_epochs = \max\left(1, \min\left(\frac{module.complexity}{\max\left(log(generation + 1), 1\right)}, max\_epochs\right)\right) \tag{2}$$

where MAX_EPOCHS is the maximum number of training epochs allowed. This value is not just a ceiling for the number of epochs that a network can be trained for, but also indirectly acts as an implicit threshold that penalizes networks with too many parameters. If a network is too large, it will not be trained to its full potential and its recorded accuracy will be lower than its true accuracy. This translates to a worse fitness which affects how the topology is utilized in later stages. The intuition behind this heuristic is that smaller networks with less parameters need less time to converge, while bigger networks require more training time to accurately exhibit their performance. The purpose of the complexity heuristic is to speed up the training process in the population by quickly estimating the amount of training each network requires. If all networks were trained for a constant number of epochs, a significant amount of time would be lost in the training of small, already converged networks, due to the need to set the epoch count to a high number in order to accommodate for the larger networks that are created in later iteration and demand sufficient training. Training the population is the most time intensive part of the algorithm.

After training the appropriate networks, the notable modules list is updated to contain the most performant parts of the topologies in the population. Updating the

notable modules is a multi-step procedure that is analyzed later in the thesis. In the last part of the evolution loop, the networks with the lowest fitness scores are eliminated. The number of networks to be eliminated is expressed as a percentage over the total number of networks in the population. At the end of each iteration, the threshold fitness value for the population is calculated based on this percentage, after which point all networks with fitness scores less than the threshold are deleted. In order to keep the population size constant, an equivalent number of new candidate networks as those deleted is generated and inserted into the population in order to be evaluated at the next generation. Replacing networks is a key component in the algorithm because it filters out topologies that are either too simple, too complex or just don't have a suitable layer and connection configuration to produce good results. Furthermore, the percentage of the population that is replaced directly influences the rate at which new modules enter the notable modules list. The frequency at which the replacement process runs is itself controlled by a separate parameter which determines how many iterations should pass between every replacement pass. If the number is high, multiple mutations are applied to the same networks, increasing their size and their sophistication. If the number is low, the notable modules list is updated faster which leads to a faster convergence on the network topology. Depending on the problem and the available layers, this value needs to be tunes to provide optimal results.

This high level overview of our proposed algorithm serves in gaining an initial intuition about the different systems at play. The snippet below contains pseudocode for the algorithm in a more compact form, highlighting the structure of the underlying evolutionary process. Certain components are abstracted behind functions that will be analyzed in later. In the following sections, we will begin expanding on the different routines and structures that are used throughout the algorithm and the ways they are interlinked.

Algorithm 2 Dynamic Hierarchical NAS - Overview

▷ Create initial population with random topologies.
$population \leftarrow \{\}$
for $i \leftarrow 0$ **to** $population\_size$ **do**
    $module \leftarrow createNeuralModule\,()$
    $population.add\,(module)$
for $generation \leftarrow 0$ **to** $generation\_count$ **do**
    for each $module \in population$ **do**
        $mutate(module)$
        ▷ Clamp the training time between 1 and the maximum number of epochs.
        $training\_epochs = \max\,(1, \min\,(\frac{module.complexity}{\max\,(log(generation+1),1)}, max\_epochs))$
        $accuracy \leftarrow trainAndEvaluate(module, training\_time)$
        $module.setFitness(accuracy)$
    updateModulesList(population)
    ▷ Apply threshold to eliminate weaker topologies.
    if $generation \mod delete\_interval == 0$ **then**
        $networks\_to\_replace \leftarrow network\_replace\_pct * population\_size$
        ▷ Sort population by fitness(ascending).
        $sort(population)$
        for $i \leftarrow 0$ **to** $networks\_to\_replace$ **do**
            $population[i] \leftarrow createNeuralModule\,()$

Hierarchical Network Representation

Evolutionary algorithms operate on members of the population in order to find a near optimal solution to a specific problem. Inspired by traditional biology, the population contains entities called chromosomes, each of which is a solution to the problem. Chromosomes use a user-defined encoding in order to contain the properties of the solution. Thus, its format should be sufficiently descriptive so that its information can be used to reliably solve the problem. The encoding should also be flexible, to allow the different operators to change parameters in the solution easily. A secondary requirement for the encoding is to be lightweight, especially in cases where populations are large. Other requirements may be set in a case by case basis, depending on the particular characteristics of the problem.

In the domain of NAS, a chromosome or solution is the topology of a neural network which, when trained, performs well in the given dataset. It is very common for researchers to adopt a representation that uses a fixed length array to encode the graph information. In this case, each element in the array usually holds information

about neural layers which are combined to form the candidate solution. This approach is often preferred because of its ease of implementation and manipulation with the evolutionary operators. However, it is also very restrictive in the allowed network structure and size. Even if "no-operation" layers are used, there is still a maximum size limitation on the constructed topologies. A better solution is introduced in [12] where chromosomes take the form of lists that can be expanded. This somewhat complicates the implementation of the different operators, but the authors manage to make it work successfully.

In this work, an original, novel representation scheme is used to encode network information in the chromosomes. The inspiration behind it was the idea that neural networks are black box models that receive an input and produce an output, while their internal processes remain hidden. Neural networks are computational graphs; in their most basic form they can be comprised of a single hidden layer. In a similar manner, the black box idea can be extended to the notion of the neural layer: even though their computations are well defined and understood, they can still be treated as a model that receives an input and calculates an output. Naturally, in a neural network graph, a group of connected layers form a sub-graph that can adhere to the same principle. At a higher level, different sub-graphs can form even larger sub-graphs, and eventually the full network. Treating every entity in this fashion allows for the definition of a universal abstraction construct, the neural module. A neural module is defined as a computational graph of any size that can process an input and produce an output. Each node in a neural module can represent either a neural layer or another neural module. Starting from a full neural network, neural layers and then neural modules can iteratively be organized into sub-graphs, ultimately building a hierarchical graph structure. At the top, a graph with only just one node represents the entire network. It is connected to an input and output nodes which provide and receive the data and the result respectively. Lower level neural modules also get auxiliary input/output nodes to direct the flow of the data. The input data is propagated into the lower level neural modules, until it reaches the neural layers at the bottom of the hierarchy. The data flows from an abstract graph of a module to the abstract graphs of the neural modules corresponding to each node by being directed to every node in the internal neural modules' abstract graph recursively. After all the neural layers in a neural module complete their data processing, their combined result is formed and sent on the next hierarchical level as the result of the neural module in which they belong.

One of the implicit consequences of this representation is that it is possible to have hierarchical graphs with varying depths. This removes the need to introduce "no-operation" or identity layers to the set of available layers for the optimization algorithm. Another important result is that the maximum depth of the graph does not have to be defined. Depending on the execution of the evolution process, the

hierarchical graph will always result in a valid network. Of course, many different hierarchical graphs can result in the same network. The hierarchical structure is only utilized by the algorithm, while the final result is a normal neural network graph, i.e. a computational graph with nodes representing neural layers. The process which extracts a neural network from a neural module operates by replacing each node in its computational graph, also called the abstract graph, with the computational graph of the corresponding neural module. When this is done recursively on all hierarchical branches, the resulting graph is the full neural network.

The hierarchical representation is designed to satisfy the 3rd goal set earlier, which is to find intermediate architectures that provide good performance on the problem. This is an extension of searching for micro architectures, which are sub graphs that contain neural layers. If a neural module containing just neural layers is considered a block or cell equivalent (using the terminology of other NAS publications) then its higher level neural module would be an intermediate level block containing neural modules and possibly some neural layers on its graph. Considering that a key feature of deep neural networks is repeating cells in different ways, combining higher level cells should achieve the same result.

It should also be noted that the hierarchical representation can be used to organise layers in existing, handmade, state of the art topologies. Consider for instance the topology of InceptionNet, a deep neural network , initially designed for image classification datasets. It is constructed by repeating a cell architecture called the Inception Module that performs features extractions at different resolutions, along with a selection of intermediate layers that link the different cells.

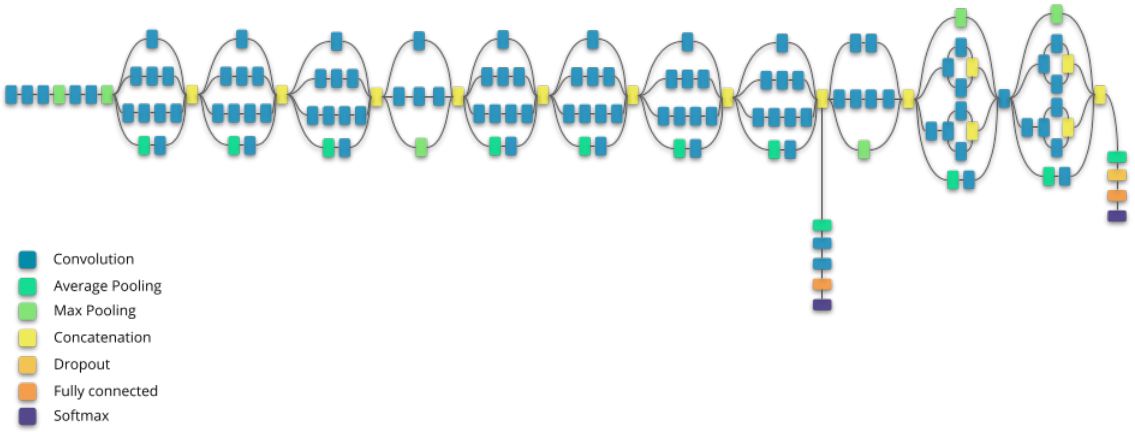


Figure 6: The InceptionNet model.

The authors of the original paper understandably focus more on the novel archi-

tecture of the Inception Module. However, by observing the complete architecture, it is clear that there are multiple other repeating sub graphs that can form cells, such as the classification heads, or the modified Inception Module on the first and second classification branches. These 2 sub graphs, along with the Inception Module are clear examples of sub graphs that can be represented as a low level neural module. However, it is possible to add more abstraction to the construct by defining yet another neural module one level higher than the previous ones, whose computational graph describes the way the three lower level neural modules are connected together. In the original InceptionNet, this high level, abstract module is repeated twice. Continuing this process will result in the creation of a rigid hierarchy with only one module at the top, containing information for the entire network. The same can be applied to all network architectures. The hierarchical representation is designed for the construction of networks with repeating patterns at multiple levels, which describes many well performing models such as the VGG, ResNet, EfficientNet and others.

Network Generation

In the previous section we saw how an existing network may be organised into different levels of neural modules using the hierarchical representation. The process of organising a network such as the InceptionNet by hand is useful for acquiring a basic intuition about the representation structure, but doesn't really explain how the system can be utilized by a generative routine in an evolutionary context. This section analyzes the evolutionary process' network generation algorithm, and how it takes advantage of the hierarchical representation to initialize networks with increasingly more complex topologies organically.

A significant weakness of successful NAS publications, such as the works of Zoph et. al. and Mikkulainen et. al. is the slow pace with which topologies are constructed and improved. In both cases, candidate networks start with a single layer and are iteratively improved by adding nodes and edges to the graph, either via a reinforcement learning model in the first case, or with the use of a genetic algorithm in the second. An important consequence of the hierarchical representation is that it allows for an improvement on the first part of the problem, which is the topic of this section. There are 2 points at which network generation occurs in the algorithm. The first one is before the evolution loop, where the initial population of networks is created. The second one is at the end of a generation, when new population members are generated to replace those just deleted by the threshold filter. In both cases, the generation algorithm produces a neural module in two steps.

The process begins by selecting the number of nodes N in the neural module's abstract graph. This parameter can be set as a constant value, or be sampled from

a distribution to add variance to the members of the population. In either case, the number should not be too high because it is very easy to exceed computational resource limits. On the first step of the network generation, N neural modules are sampled from the notable modules list. Each neural module in the list has an average fitness which is used as the weight for the sampling step. Neural modules with higher average fitness values are more likely to be part of the generated neural module. During the second step, a random computational directed graph is created for the new module. The graph has N nodes, each one representing a randomly assigned neural module from the set of modules sampled in the first step. Instead of using one of the many available network generation algorithms that are available, a custom one was designed to ensure that the resulting graphs adhere to two basic rules guaranteeing that the resulting neural networks are valid and do not waste computational resources. The first rule is that every node in the graph must have at least one input edge and at least one output edge. In the case where a node has no incoming edges (source node), no signal can reach the neural layers of the corresponding neural module, so it never participates in the training and is in no way useful to the resulting network. On the other hand, if a node has no outgoing edges (sink node), even though it can receive a signal from previous nodes and perform a calculation, the result is not used in any way by the network, rendering it useless. This also means that its incoming weights are not being trained through loss backpropagation, offering no contribution to the preceding layers. Furthermore, its inclusion adds a computational burden that slows down the training process for no reason.

The graph generator creates graphs focusing on this rule. The first step in graph generation is to create one node for the input signal and one for the output signal. These are auxiliary nodes that are added to control the flow of the graph, and do not actually correspond to a neural layer or module. After that, N unconnected nodes are placed on the graph space. Next, a random number of edges are created, starting from the input node and directed at a random subset of the unconnected nodes. At least one edge must connect the input node with the neural module nodes in order to allow the information to be transmitted through the network. A similar procedure happens for the output node. A number of module nodes are randomly selected and connected to the output node in order to propagate the data to the rest of the network. At this point, some nodes in the graph have no incoming edges, some have no outgoing edges, and some are not connected at all. The next step is to distinguish unconnected and partially connected nodes into two sets, depending on the type of connection they are missing (the two sets can have common elements). At each case, an edge is added to complete the missing connection: nodes with no incoming connections get an edge from a random node to them and nodes with no outgoing connection get an edge from them to another random node. This method guarantees that all nodes in the graph will be connected properly. However, there is a second issue that may arise:

there might be cycles in the graph. Even though there are cases where cycles are used in deep neural networks (e.g. RNNs) they are avoided in convolutional neural networks (which is what NAS mainly focuses on), as they tend to run into issues with vanishing gradients very often, which are normally not present in sequential neural networks. For this reason, the second rule states: the directed computational graph must be acyclic. To enforce this, after the topology has been generated, a routine checks the graph for cycles [21]. If there are any, the previous step runs over again, generating a brand new graph to be evaluated for cycles. If a generated graph has no cycles, it is valid and the neural module can be created.

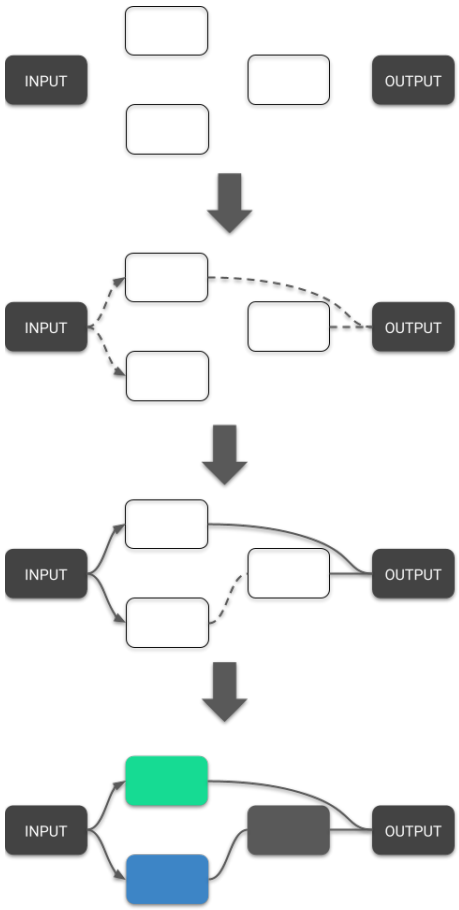


Figure 7: The graph generation occurs in 3 stages: first, N nodes are created. Next, a random number of nodes are connected with the input and output nodes. Then, nodes with no incoming or outgoing connections get random edges with a randomly assigned node in the graph. If there are no cycles, the graph is valid and nodes can get assigned neural modules.

At training time, each member of the population recursively replaces each node in its abstract graph with the abstract graph of the neural module it represents, until the resulting complete graph is comprised of just neural modules representing layers. The novelty of this approach results from the fact that each neural module can represent a sub graph of any size. Contrary to other approaches where nodes and edges are assigned one by one, associating a node of an abstract graph with a neural module adds an entire new segment to the neural network. Unlike micro architecture search, it is not just a small topology being added - each module can be deeply complex, itself combining other, lower level topologies to be formed. Even though this behaviour is highly desirable, it can be the source of a significant problem: if the number of nodes in the abstract graph is anything other than small, an explosion in the number of layers in the network may occur. This happens because all lower level neural modules except the lowest level ones (those that represent a single neural layer) will also have the same amount of nodes (or a similar amount, if the number is sampled from a distribution. Thus, the number should be sufficiently small, as to expect this immediate formation of a moderate sized network right at initialization. The actual initial network size is tied to the iteration number - as the algorithm progresses, more complex network will be initialized each generation.

Network Mutation

Every evolutionary process needs to incorporate a set of operators that improve the initial population by modifying the candidate solutions. While there are many different functions that can be applied to a set of chromosomes, they have to be compatible with their structure or alternatively be designed with their properties in mind. The proposed hierarchical representation is designed to provide many benefits at the evolutionary algorithm at a cost. The drawback to this chromosome formulation is that it is very restrictive in terms of what operations can be applied to it. Due to its novel structure, most typical evolutionary functions, such as the crossover function, cannot be applied out of the box. As a result, custom operators must be designed to allow solutions to improve and produce better results.

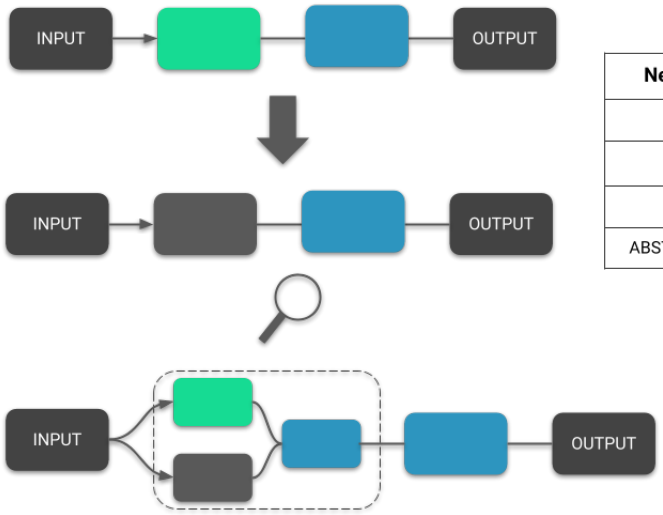
Modifications to candidate topologies are performed through the mutation operator, which is the standard operator for evolutionary algorithms. The mutation operator has a chance to get triggered in each candidate at every iteration, randomly changing a set of properties of the solution. Typically, mutating a graph would mean making a modification in its nodes or edges, such as adding or removing a node or an edge. Mikkulainen et. al. implement the mutation operator in their work by introducing functions that can temporarily disable a node or edge in a network, while preserving the information so it can be enabled again in a later iteration in the same

manner. In this work, mutation functions operate in two ways: in the node level by replacing a simple neural module for a more complex one, and in the edge level by adding an edge or set of edges to the candidate network. Yet again, the hierarchical architecture allows for the rapid modification of networks by incorporating the level of the neural module being modified in the hierarchy.

Starting with the node mutation, this operation is the most transformative of the two, because it adds an entire sub graph to a select position in the topology. This action is performed by selecting a random neural module representing a single layer from any level in the hierarchy of a candidate and replacing it with a neural module sampled from the notable modules list. The process is simple: starting from the top level of a candidate, a node in its abstract graph is chosen at random. If the selected node represents a neural layer, it is replaced with another neural module from the notable modules list. Otherwise, the process repeats for the nodes of the abstract graph of the chosen node.

Algorithm 3 Node mutation

```
1:  procedure MUTATENODE(module : NeuralModule)
2:      ▷ Sample a random index from a uniform distribution.
3:      idx ← U(0, module.child_modules.size())
4:      selected_node ← module.child_modules[idx]
5:      if selected_node.module_type == ABSTRACT_MODULE then
6:          mutateNode(selected_node)
7:      else
8:          selected_node.module_type ← ABSTRACT_MODULE
9:          generateRandomGraph(selected_node)
10:         sampleChildModules(selected_node)
```



Notable modules list

Neural module	Average fitness
CONV_3.N	0.29
POOL_2.M	0.18
CONV_1.H	0.23
ABSTRACT_MODULE	0.45

Figure 8: Node mutation in an abstract graph. A neural node in the hierarchy is randomly selected and replaced with a neural module from the notable modules list. In this example, the green node is replaced with the abstract module with an average fitness of 0.45. Expanding the new node reveals its internal components.

The replacement of the neural layer with a neural module doesn't necessarily mean that that new addition will be more complex than the old module - especially in the first iterations, it is very possible that a neural layer will be replaced by another neural layer. This implementation of node mutation creates balanced hierarchical graphs that are comprised of abstract modules at the top and more neural layers at the lower levels of the hierarchy. The reason this happens is because it is less likely for lower level neural modules to be mutated due to the iterative sampling that would have to occur for such an event to take place. Neural layers in higher levels naturally have a higher chance of being converted first. As a result, the overall structure of the network is more rigid, with the different sub graphs manipulating the data cooperatively, similarly to the way it would be processed in a man-made graph. In contrast, if the node mutation were to be performed at a neural layer at a uniformly sampled level in the hierarchy, the network density could differ significantly from sub graph to sub graph. This is not desirable behaviour as it could lead to erroneous

topologies.

The second operation is the edge mutation, which is far simpler than the previous one. This operation adds a directed edge between two neural modules at a level in the hierarchy of a candidate network. Even though it is similar to the previous operation, this one can fail if there is no way to add an edge to the selected sub graph without creating a cycle. The mutation function operates as follows: starting from the top level of the hierarchical graph, decide in a stochastic manner whether an edge is to be added at this level or on deeper level, on the abstract graph of a child neural module. If the edge is to be added on the current level, generate a list with all the possible edges that can be added to the graph of this module without creating a cycle. If there are no such edges, the mutation fails. If there are valid directed edges, one is selected at random from the list and added to the graph. In the case that the edge is to be added to a child module, a non trivial neural module (i.e. a neural module that has more than just a single layer) is selected and the procedure runs for its abstract graph instead.

Algorithm 4 Edge mutation

```
1: function MUTATEEDGE(module : NeuralModule)
2:     nodes ← module.abstract_graph.getNodes()
3:     ▷ Track whether an edge can be added on this hierarchical level.
4:     can_add_edge ← true
5:     while true do
6:         visit_child ← U(0, 1)
7:         if visit_child == true || can_add_edge == false && nodes.size() > 0
   then
8:             idx ← U(0, module.child_modules.size())
9:             selected_node ← module.child_modules[idx]
10:            result ← mutateEdge(selected_node)
11:            if result == FAILURE then
12:                nodes.remove(selected_node)
13:            else
14:                return SUCCESS
15:        else if can_add_edge == true then
16:            ▷ Get all edges that can be added to the graph.
17:            possible_edges ← getAllPossibleEdges()
18:            if possible_edges.size() == 0 then
19:                can_add_edge ← false
20:            while possible_edges.size() > 0 do
21:                ▷ Sample an edge
22:                edge_idx ← U(0, possible_edges.size())
23:                edge ← possible_edges[edge_idx]
24:                ▷ Create a copy of the abstract graph.
25:                graph_copy ← copy(module.abstract_graph)
26:                graph_copy.addEdge(edge)
27:                if hasCycles(graph_copy) then
28:                    possible_edges.remove(edge)
29:                else
30:                    module.abstract_graph ← graph_copy
31:                    return SUCCESS
32:            can_add_edge ← false
33:        else
34:            return FAILURE
```

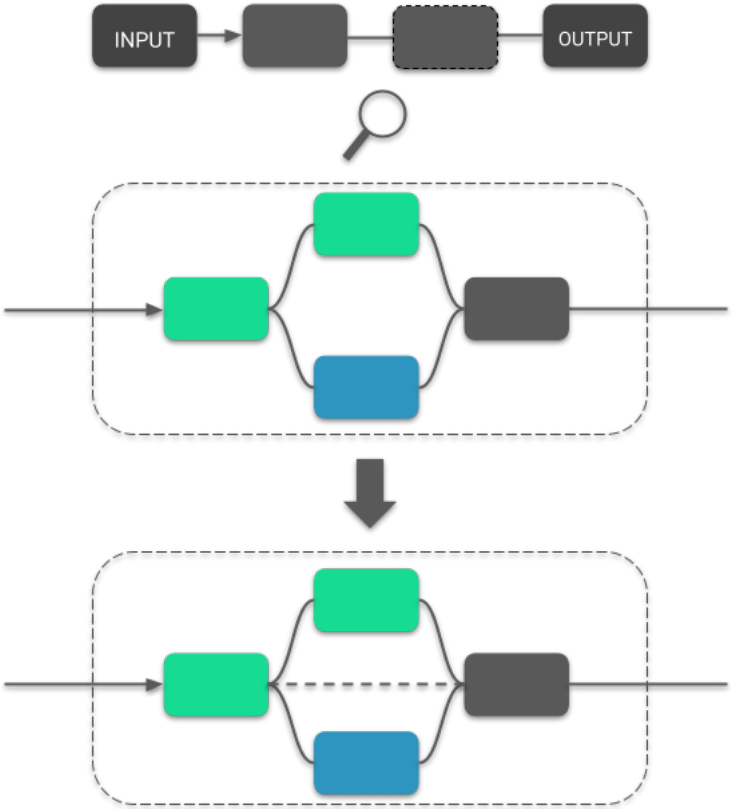


Figure 9: Edge mutation in an abstract graph. In this example hierarchy, the second abstract module in the upper level is selected for edge mutation. Expanding its topology, a random edge is added between two neural modules.

It is apparent that the stochastic element determined whether the function will place an emphasis on creating edges at higher or lower levels. If, for example, there is a 50% chance of adding the edge at the current level and 50% chance of adding it on a child module, the algorithm will greatly favor adding edges high in the hierarchy. This can be beneficial because edges on abstract graphs of higher levels translate to more actual edges in the final network topology. An edge between two nodes in an abstract graph means that all neural modules of the abstract graph of the neural module of the source node will be connected with all neural modules of the abstract graph of the neural module of the destination node. If the edge placement is favored on lower level modules, the changes to the computational graph will be less severe and more localized in the sense that the connection will be established between nodes that share a common parent. High level edge additions allow for the connection of more disconnected areas of the network, which may prove to be useful in some cases.

However, they are also more likely to connect many different areas in a way that slows down the training process without offering a benefit in performance.

In order to determine the type of mutation to be applied to a candidate network, a random number is sampled from a uniform distribution. Then, a decision is taken depending on the sampled value: whether to mutate a node, edge or leave the module as is. In other methods with more than one operator, mutation is very rarely performed, and is used in order to offer some external variation to solutions that may tend to converge. In our case, since mutation is the only operator, there is little justification for not using it on every network at each iteration. As such, the chance for skipping mutation on a network could be set to zero, which leads to a more rapid network expansion. Alternatively, some networks will not be mutated at each iteration, and thus not be reevaluated, which reduces the amount of time needed per loop. As for the chance of the other two events, a node mutation is undoubtedly a more transformative operation and should probably have a lower chance to occur compared to edge mutation, which only affects the flow of data in the network.

Fitness Evaluation

After the networks are built, they must be assigned a fitness value. The fitness value is a scoring metric that determines how performant each network is in relation to the other candidate networks. All evolution methods implement a scoring function that evaluates a chromosome and calculates its fitness for the problem at hand. The fitness value determines the degree to which the evolution process exploits each solution's characteristics in subsequent iterations. In other works, the properties of the best solutions at each iteration tend to be incorporated in other candidate solutions in an attempt to further improve the population. Furthermore, a good fitness value of a solution translates to a longer survival in the population, by virtue of not being affected by techniques such as fitness cutoff thresholds which aim to purify the population by removing its worst performing members and replacing them with new ones.

In the context of neural architecture search, where solutions are neural topologies, the fitness function must relate to network performance on the given dataset. The obvious way of assessing network performance is to adequately train a network and calculate its accuracy and other relevant metrics. Another approach is to use a predictive model, such as a graph neural network, to estimate the performance of a network. While that method is initially attractive because it greatly reduces the assessment time for the population at each iteration, it has two disadvantages: the estimations are prone to a certain degree of error, and the predictive model requires training. For this work, the fitness of each topology is equal to the accuracy of the corresponding trained network when evaluated on the validation data set. The degree

of training for each network varies depending on its complexity in order to save time by avoiding training topologies for more than they need to be trained. Training a network is a straightforward process that can yield fitness values for the candidate topologies. Using the hierarchical network representation, the calculated fitness value will correspond to the entire topology, which can be represented by the top level neural module of each candidate member in the process. This however, is not the full extent to which the fitness value is used. It is important to know which parts of the network are significant and beneficial to the entire topology. This refers to the lower level neural modules that form sub graphs in the architecture and serve a purpose in improving the performance of the network by transforming the input data in various ways. In a sense, these child neural modules can be considered properties of the overall solution - they are smaller networks with their own set of properties and solve a particular, internal problem. Consequently, these can too be evaluated indirectly in relation to the fitness of the entire network. Assigning a fitness value to all modules in the hierarchy of a candidate reveals the most useful sub graph schemes in the population, similar to how some properties are highlighted for their ability to produce good solutions. Neural modules that are present in many well performing networks should be considered useful and used by the mutation process more often than modules that, when combined with others, create problematic topologies. Finding these modules is a two part process which begins by calculating values for al neural module components in topology's hierarchy and then using them to rank and promote modules in the notable modules list (which will be analyzed in the next chapter).

Three different approaches were considered for carrying out the process of assigning fitness values to child modules. The first approach operates by taking a parent module's fitness and splitting it evenly on the child modules of its abstract graph. This is done recursively, which results in all modules from the top to the bottom of the hierarchy getting assigned their fitness values. Due to the fitness division that occurs at every hierarchical level, higher level modules have higher fitness values while neural layers in the bottom have much lower fitness values. The intuition of this method is that the contribution of a single neural layer or small sub graph to the entire topology is smaller than that of a much larger sub graph. Taking this fact into account, complex, higher level modules should be preferred for the construction of candidate topologies.

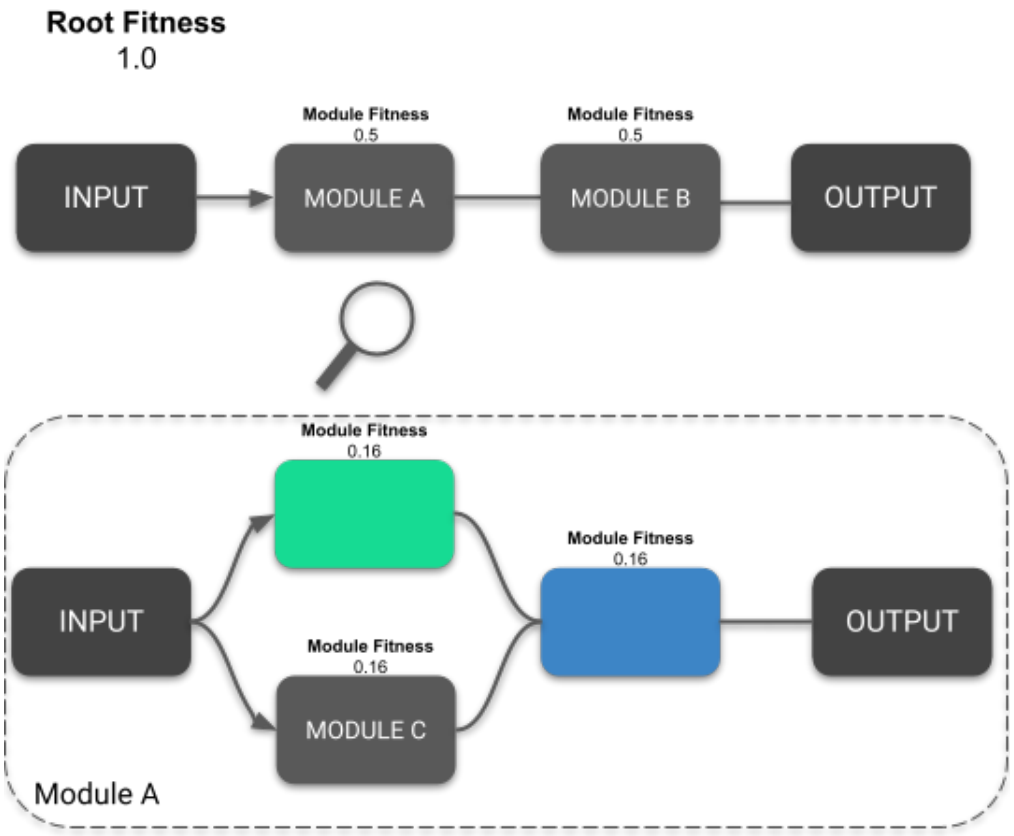


Figure 10: In the first fitness distribution policy, the fitness of a module is split evenly among its children.

The second approach is more complicated, taking into account the structure of the hierarchical sub graphs in a candidate. A fitness flow mechanism is responsible for evenly distributing the fitness of a module in an abstract graph, as if it had the properties of a liquid. Once again, starting at the top, the process begins in the first abstract graph by assigning the entire fitness value to its input node. The value is then spread evenly among its outgoing edges, producing the fitness values for the next set of nodes. If a node has more than one incoming edges, it receives the sum of the fitness values of the nodes that feed into it. This continues for every node in the abstract graph, until the entire fitness is aggregated in the output node (this approach dictates that the input and output nodes must have the same fitness values for the process to have worked correctly). Each module in the hierarchy gets its fitness and continues by performing the same procedure on its abstract graph using the fitness value assigned by its parent as the initial value that will be distributed. The logic behind this approach is that connections are important and should be taken into

account when evaluating modules in the hierarchy. Nodes that have multiple incoming edges are more significant because they combine different signals together. Topologies with known patterns such as skip connections boost the evaluation of internal nodes, indicating their importance in the network. This also means that the same neural module can exhibit different fitness values, depending on how it is connected when placed at different points in the graph.

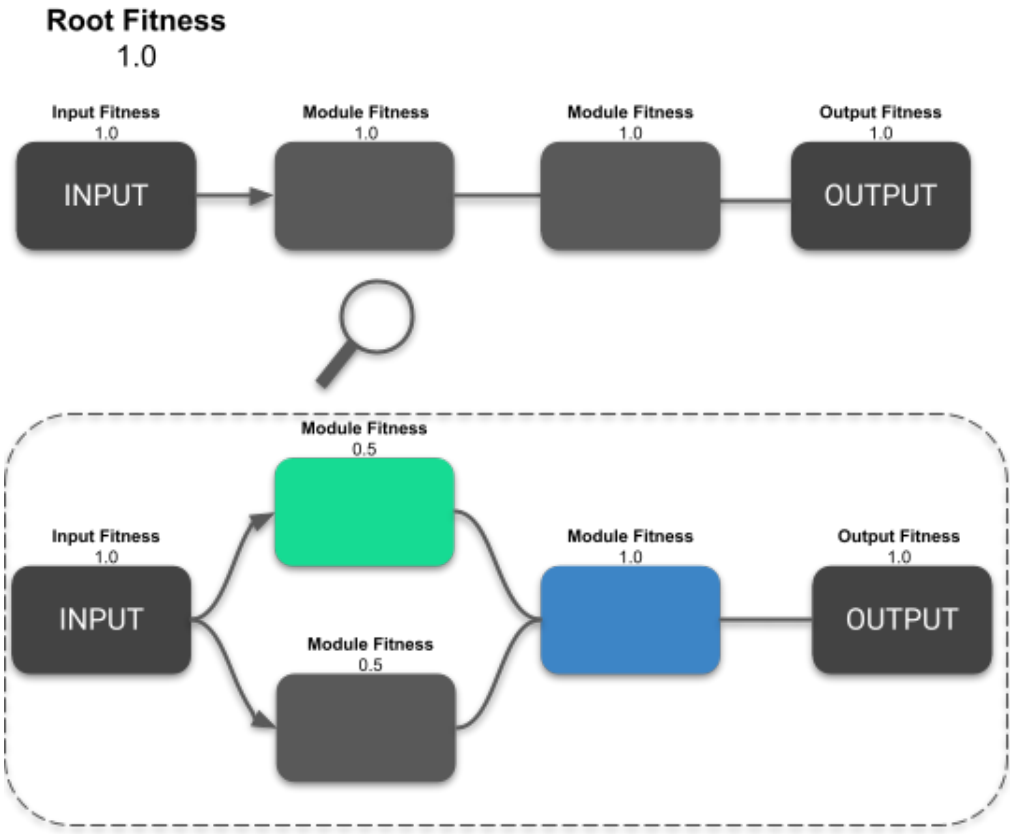


Figure 11: In the second fitness distribution policy, the fitness of a module flows evenly through the different paths.

The third approach is the simplest of the three: given a fitness value of a candidate in the population, all internal neural modules in the hierarchy get assigned that value too, independently of their complexity or depth. This is the same technique used by [3], although it is applied to a population of chromosomes with different structure. than those proposed here. The idea is that good modules should consistently appear on well performing networks, therefore their fitness should not be penalized by their position or the depth in which they appear. However, an important problem may

arise here: how will more complex modules be highlighted for their performance when compared to lower level modules, since all modules in a topology are assigned the same value? This issue is handled organically by the algorithm through the candidate and notable modules lists in a way that will be explained in the next section.

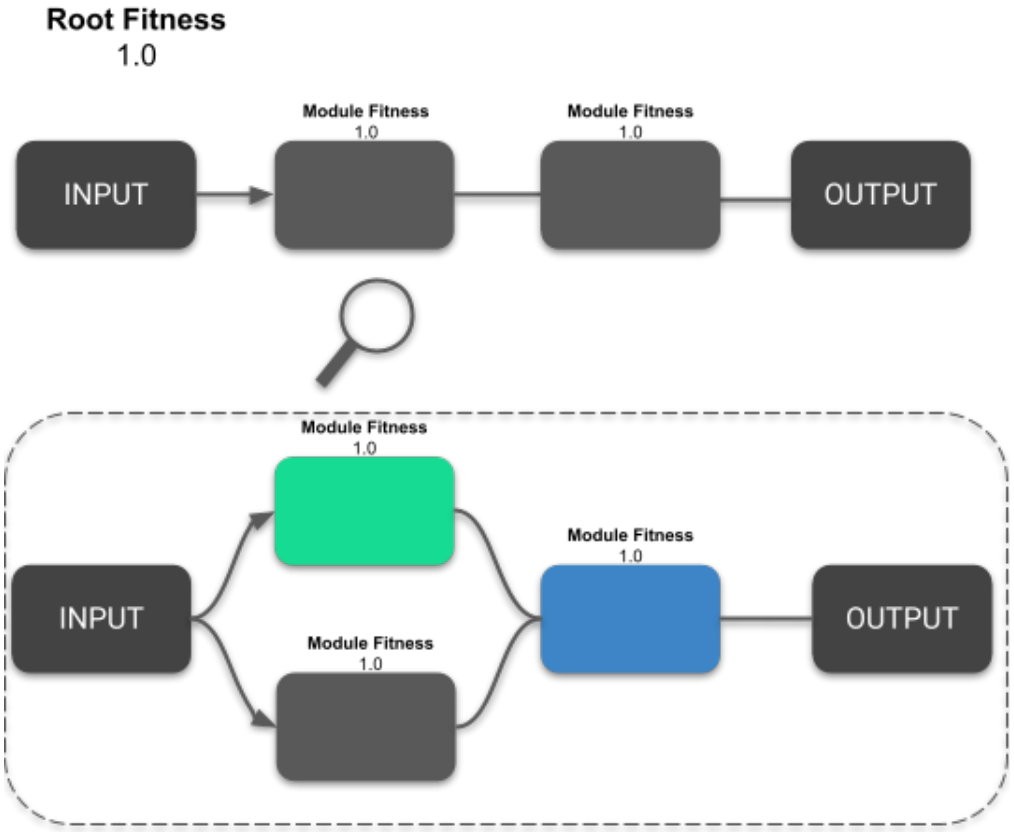


Figure 12: In the third fitness distribution policy, all modules in the hierarchy of a neural module get the root fitness.

Out of the three approaches, the one utilized in the algorithm is the third one, since it provided better results in the conducted experiments. Even though each fitness assignment technique has certain advantages and drawbacks, applying the same fitness to all modules in a hierarchy produces the best results in terms of finding the best performing architectures. Fitness evaluation of each module is controlled by an internal flag in the chromosome that marks whether a neural module requires a fitness evaluation. All new candidate architectures initially require evaluation. After that, networks may need to be reevaluated in the case of a mutation, at which point the resulting network is treated as an entirely new topology. The modified or added

module triggers a recursive flag update to its parent that travels all the way to the top of the hierarchy. When it is time for an evaluation, the algorithm only checks the top module for each candidate and acts accordingly. If an update is needed, all modules of the network get updated fitness values. Otherwise, the candidate fitness is not reevaluated.

Candidate and Notable Modules

The notable modules list has been mentioned a few times already. It is the final component of the algorithm and its function is the cornerstone of the entire process. Its use it to index those neural modules that consistently provide good performance when included in a computational graph. In reality, this data structure is an associative table that contains neural modules and their corresponding properties, such as their average and cumulative fitness, their complexity values and other info which is used to perform the sampling operations. There are 2 additional lists that are implemented along the notables list: the candidate modules list and the banned modules list (also called blacklist). This section analyzes their purpose and the way these 3 structures interface with each other and the rest of the evolutionary process in order to push the search forward.

The evolutionary process operates by creating populations of candidate topologies, evaluating each one, distinguishing the modules that have a better than average performance and using them to construct the next iterations of populations. In order to create topologies, the network generation function samples neural modules from the notable modules list, and then assigns each module to a node in a randomly generating abstract graph. The graph, along with the sampled modules, form a topology which is also considered a new neural module with the aforementioned as its properties. The notable modules list is initialized by creating and inserting into it a set of predefined neural modules, each representing a single neural layer (essentially graphs with one node). These should be layers that are typically used to solve the type of problem the dataset poses. Usually, convolutional and pooling layers with different properties make up the layer list. The first population members are created by combining these simple neural layers in different permutations, building slightly more complex neural modules. The new networks are trained and evaluated, and all the modules under the hierarchy are assigned fitness values. Modules that are already recorded in the notable modules list have their average fitness values updated. In order to update the average fitness of a module in the list with the fitness of the module in a new network, the following snippet is used:

Algorithm 5 Average fitness update

1: **procedure** RECORDFITNESS($module : NeuralModule, new\_fitness : int$)
2: $\quad module.average\_fitness \leftarrow \frac{module.occurrence\_count * module.average\_fitness + new\_fitness}{occurrence\_count+1}$
3: $\quad module.occurrence\_count++$

This initially concerns simple modules, like the modules representing neural layers that make up the computational graphs of the candidates. As for the new, higher level modules that have not yet been recorded in the notables list, they have to first be placed in the candidate modules list, where they will remain until it can be determined whether or not they are of quality. The key feature of the notable modules list is its constrained size - it can only contain a predefined number of modules. In order to insert a new, well performing module in the list, if the list is full, another module with worse average fitness must be removed. This has two purposes: the first is to create an evolutionary pressure that forces the process to only maintain the best performing modules, which will eventually also lead to population convergence. The second is to force the algorithm to work with progressively higher levels of abstraction in its modules, by replacing simpler modules with more complex ones as more generations of populations are produced. Considering the fitness evaluation scheme that was selected in the last section, a valid question may arise: how can the process replace simple modules for more complex ones, if all modules in the hierarchy get the same fitness value? This can be answered by considering the probability of two new modules of different complexity being created. The most trivial neural module, an independent neural layer, can occur very frequently in many networks in the population. As such, its average fitness will be affected by all topologies in which it is included. This includes not only good networks, but also bad topologies that fail to exploit the layer properly due to the structure of their graph. A more complex module will be harder to create: the abstract graph and the sampled layers would have to be identical to the prototype. As such, a much smaller subset of networks will contain it. The quality of these architectures is bound to show less variance: depending on the quality of the module (and the other modules at use), these topologies will lean towards a better or worse performance. As a result the fitness of the neural module, if it is actually good, will be averaged by a set of networks with better average fitnesses. This ensures that more complex architectures will be replacing the simpler ones as the discovery process continues.

Even though new neural modules can be directly compared to existing modules in the notables list, using their fitness values, there is a potential pitfall that must be avoided: it is not fair to compare the fitness of a new module that only just appeared with the average fitness of a module that has appeared multiple times in the population. In the first case, the performance of that module is heavily tied

to their specific position in the topology. Even if it is a top level module, its fitness does not necessarily reflect how well it could work as an internal module with just one sample. As a result, if for example its fitness is good, it could replace a module on the notables list that has a lower average fitness but is actually quite adaptable. In order to have a proper comparison between a newly formed neural module and those on the notable modules list, an average fitness must be evaluated for the former, aiming to provide a more representative picture about its quality and deal with edge cases where a module is assigned an exceptional score due to the inclusion of other modules in the network. This is where the candidate modules list comes in. This list is yet another associative table for modules where new chromosomes are initially placed before a comparison between them and the notable modules is deemed valid. The candidate modules list has a much larger size compared to the notable modules in order to allow a large set of modules to be stored. When a new module is first encountered in the population it is recorded in the candidate modules list. New modules occur at initialization and with each mutation: when a module in the hierarchy is modified either by changing a node or an edge, its parent modules and all other modules from that one to the top (following the parent chain) are also considered modified and could possibly make up new modules. All new modules in a chromosome are recorded in the candidates list. Each module is associated with a block of information similar to that of the notable modules, concerning the fitness, module occurrence count and also a time-to-leave (TTL) counter. The last data object is unique to this particular list and its existence is crucial to the practical application of the algorithm. The purpose of the candidate modules list is to hold modules until they are generated naturally in the population more times than a predefined threshold, at which point their average fitness of those occurrences can be compared to the average fitness of the modules on the notables list. Ideally, the size of the candidate modules list would be set to infinite, to allow all encountered modules to be stored there, until it could be evaluated whether they should be promoted to notables or not. Unfortunately, this is not physically possible as realistically the algorithm is constrained by computational limitations. The workaround for this issue is to introduce a maximum size for the candidate modules list and also add an expiry timer to modules in it. Each module has a limited number of generations after the time it was first recorded to reach the threshold occurrence count. It is reasonable to assume that if a neural module scheme of a certain complexity has not occurred in a number of generations, it is probably not going to occur again, because the modules in the notables that would be needed to create it have probably been replaced with new modules of higher complexity. The actual TTL value is individually calculated for each module by multiplying its complexity with a base TLL factor:

$$TTL = BASE\_TTL \cdot module.complexity \tag{3}$$

The TTL is reduced by 1 for all candidate modules whenever a generation is complete. If the TTL of a module reaches 0 and the module has no yet been promoted to notable, the module is removed from the candidates list and placed in the banned modules list. The blacklist contains all modules that are not to be recorded in the candidates list, should they be encountered in the population. All modules in a network hierarchy are compared against those in the blacklist, and if there is a match, that module is skipped. The existence of the banned modules list allows for the continuous insertion and deletion of elements from the candidates list, which is necessary for introducing new modules to it and the notable modules list.

At the end of each generation, candidate modules that have exceeded the occurrence threshold have a single chance of being promoted to notable modules. If there is space for N more modules in the notables list, the top N candidate modules with the best average fitness are promoted to notable modules. Then, if there is no free space left in the notables but there are still candidate modules with enough occurrences and a better fitness than that of some notables, those candidates are promoted and the previously notable modules are placed in the blacklist. This is done to indicate to the algorithm that it should not consider modules that were considered good in the past, but are now too simplistic and obsolete. Finally, if a candidate module has achieved the occurrence requirement but does not exhibit an average fitness good enough for the notables list, it is placed in the blacklist. This is the expected case of a configuration that is simply not good enough, and should therefore not be considered.

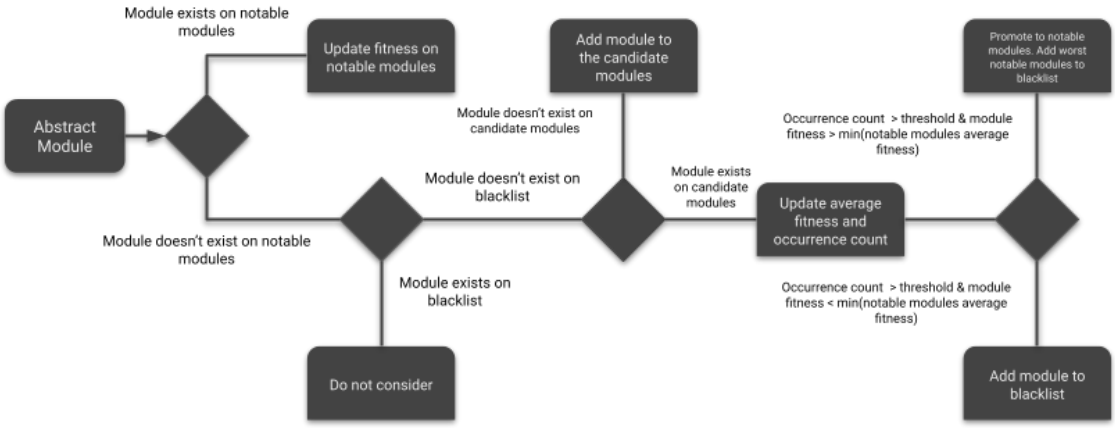


Figure 13: The process of curating a module using the 3 records.

Even though the mutation operator is the only one used in the algorithm, the function of the candidates and notables list could be considered an alternative to a

crossover operator. The concept is common in both this implementation and that of a typical crossover operation: parts of well performing chromosomes are combined together with the use of a rule to produce better solutions that combine characteristics from their parents. In our case, the desirable properties (the good neural modules) have to be added to the notables list in order to be sampled at later generations through mutation, while crossover is usually applied directly to two or more members in the population. This indirect implementation of a familiar idea is designed to circumvent the problems that would arise when trying to crossover two hierarchically represented networks, when the depth and the structure of those networks differs. They key difference between the two approaches is that when sampling from the notables list, the information about the origin of the module is not transferred to the new network - the topology is selected because it is considered generally good. Even though the mutation implementation is roughly equivalent to a genetic algorithm with crossover, the algorithm is not classified as such: The three external data structures are the main source of new modules to networks, not other "parent" chromosomes.

This concludes the theoretical description of the algorithm. The three lists of modules are at the core of the search process, continuously recording topologies of increasing complexity, comparing them with the best found architectures and promoting them to notable in order to change the set of building blocks used by the network generation process. The search changes, starting small and eventually using abstract module topologies to synthesize deep neural networks that provide superior performance when compared with their predecessors. With the appropriate configuration, the algorithm should be able to traverse the search space efficiently and find well performing architectures for many current datasets and machine learning problems.

Technical Description

This chapter concerns the implementation of the algorithm and its features in order to conduct performance assessment. For this project the programming language of choice is Python 3 due to its ease of use and the abundance of machine learning libraries that are available, allowing those interested to develop new novel algorithms efficiently by mitigating the need for writing core functionality from scratch. In the following sections, the overall structure of the search process will be laid out through the analysis of the three most significant that define it. By the end of this chapter, it will be apparent how the different obstacles that arise during the implementation are overcome, making the process computationally viable.

Representing Neural Modules

The representation of the population members is arguable one of the most important details in an evolutionary algorithm. Using a standard type to represent the complex hierarchical modules would be a very difficult task - there are too many complex properties to encode in a string, for example. For this reason, the custom class of *NeuralModule* was defined, containing all the relevant properties in an easy to use and programmatically modify format. This class is used to create objects during network initialization that automatically assume a random topology, ready to be expanded. A neural module accepts an optional parent module during its creation in order to establish a link between the two and calculate its depth, which is equivalent to that of its parent incremented by 1. During initialization, no parent module is provided, indicating that that module is the root of a neural hierarchy and has a depth of 1. If the module occurs as a result of a mutation, it also receives a module template object called *ModuleProperties*, which is a minimal packaging of the necessary neural module characteristics in order to be replicated. In the case that no template is provided (the module is a top level module), a random module is sampled from the notable modules using the normalized fitness of each of the modules as weights. In both cases, the aquired template is used to set the rest of the parameters in the neural module. The first such parameter is the module type. The module type is implemented as an

enum and indicates whether the neural module represents a single neural layers or a complex abstract module. The next two parameters are the layer type and abstract graph, and their assignment depends on the module type. When dealing with the module type of "neural layer", the layer type is a string describing the selected layer and its properties, while the abstract graph is undefined. Additionally, the number of children in case of a mutation is sampled at random here, from a distribution of choice. If the module type is "abstract module", the layer type is undefined and ultimately ignored, while the abstract graph is a directed acyclic graph (DAG). In order to work with graph objects, the networkx package is used. Networkx provides a set of convinient classes and functions that allow for the creation and manipulation of graphs. A DAG in networkx is defined as a set of nodes along with a set of edges represented as node pairs, where the first node in each pair is the source node for the edge and the second node is the destination. Nodes are referenced using a unique integer id, each referring to a specific child module. Next, a dictionary is initialized to retain associations between graph node indices and neural module object references. Dictionaries are python's default version of associative arrays that organize data in key/value pairs offering fast access speeds at the expense of memory. There are two distinct cases when building the dictionary: if there is a *ModuleProperties* instance provided in the constructor, it supplies the new chromosome with a sorted list of module properties for the children of the template network, allowing the module to create identical child modules recursively and associate them with the correct node index. If the properties are not provided, which happens in the case of mutation, the association dictionary is initialized by creating a new random module (with no properties) for each node and associating it with it through its id. These new modules will randomly sample their topologies from the notables list and embed them in the full graph of the candidate. The abstract graph must be generated only in the case of mutation, where a simple neural module is converted to an abstract module. This is achieved by employing algorithm 3 presented in the previous chapter. Separate from the regular mutation phase of the algorithm ,an initial mutation is triggered automatically when initializing a candidate with the module properties of a template that is a single neural layer. This is a case that occurs at the start of the search process, while the notable modules list only contains the templates for the allowed neural layers, the lowest form of abstraction. The mutation aims to jumpstart the process by immediately forcing it to make candidates one level of abstraction higher than that of the notable modules. Each neural module in the population dynamically updates and caches its module properties for later reference by the search processes. The properties update is triggered by a mutation of a neural module of a certain depth in the network: similarly to the fitness update, the modified module sends a signal upwards that notifies each module in the chain between the mutated one and the top module that the topology has changed and their module properties need to

be recalculated. Modules not in the affected branch will not be modified. They are treated as inner topologies that remained intact from the mutation, which results in considerable computational savings, compared to, say, recalculating the module properties for all modules in a hierarchy whenever a mutation occurs. The update entails dereferencing the old module properties object and making an new instance from scratch, using the new values for the data of the module.

The *ModuleProperties* object records the following information for a given module: module type, layer type (irrelevant on abstract modules), abstract graph, a list of child module properties, sorted by the index of the corresponding node in the abstract graph, and the number of total nodes and edges in the full graph. If the module type is "neural layer", the abstract graph is null, the list of child module peroperties is empty, the number of total nodes is 1 and the number of edges is 0. These attributes are used to calculate a unique hash for each combination of properties that may occur. The hash is used so that a *ModuleProperties* object can act as a key in the 3 lists later on. Python supports the hash calculation of a tuple (an immutable set of variables) provided that each variable is of a hashable data type, like an integer or string. The module type and layer are an enum and string respectively, which are hashable types. However, the abstract graph is a complex object that requires special attention. For each complex neural module, the abstract graph is generated randomly. Two graphs may only differ in the ids assigned to each node, but be essentially identical in terms of what child modules they use and how they are connected together. The hashing algorithm should be impartial to the actual node indices and edge combinations between them and be able to identify when two abstract graphs are computationally equivalent. In other words, the same hash value must be produced for any two graphs if they are structurally equivalent irregardless of their node labeling. This is necessary in order to later check if a module of a candidate has been encountered before, and take the appropriate actions in the 3 lists. Two graphs that have the same number of nodes, edges and also possess the same edge connectivity are called isomorphic. Therefore, the hash of two isomorphic graphs should be the same. This is something that can be achieved with the Weisfeiler-Lehman (WL) hash function [22]. The WL hash function has the ability to produce hashes for graphs and detect isomorphism using the WL test. Producing the hash is an iterative process in which the hash neighborhoods of each node are aggregated and used as the updated node labels many times until convergence. The final hash of the algorithm is obtained by hashing a histogram of the converged labels for all nodes in the graph. It should be noted that while this algorithm ensures that isomorphic graphs will get identical hashes, there is a small chance that non isomorphic graphs are assigned the same hash. This is hardly a problem in our case, since the WL hash only corresponds to the structural hash of the abstract graph that will be combined with hashes of the other properties of the module in order to produce a final hash

for the module properties object which it represents. Consequently, the chance of the module properties objects of two non identical neural modules being the same is very small, as all other attributes (including child modules) would also have to be identical in order to produce the same hash. After performing hashing for the structure of the graph, the module properties of the child modules must also be hashed, each executing the process above. Neural modules at the bottom only use their module type and layer type to produce their hashes, so when the recursive hash call reaches them, they can quickly calculate their hash values. The aggregated hashes are appended to the end of the attributes tuple for the module properties object being hashed, along with the module type, layer type and abstract graph (structural) hash. The tuple goes through python's default hash function and produces a unique hash which can be used to identify that specific properties object. The hash for the properties of each candidate is cached in the object to save computational resources and slightly speedup training. Other neural modules that are generated in the evolution and have the same properties construct and identical properties object, which in turn produces the same hash and can be used to identify data in the 3 lists later on.

Topology Evaluation with NORD

At each iteration, after the network mutation phase, each modified candidate member of the population must be evaluated. That means using the modules in the hierarchy of each candidate to construct the full computational graph using the abstract graph and child references of each module, then using that as a guide in order to build an actual trainable neural network, which in turn is trained and evaluated on a validation set to return the fitness of the topology. Building the full computational graph is a relatively easy process that involves starting from the top module in a hierarchy and recursively replacing each node in the abstract graph with the abstract graph of its corresponding child module until no nodes in the graph correspond to modules of type "abstract module", i.e. the graph is comprised of modules with "neural layer" types and properties. Converting that graph to a neural network entails using the information in the graph to dynamically define an equivalent topology using the interface functions and classes of an appropriate deep learning package such as Tensorflow or PyTorch. This is not a very straightforward task: building and evaluating networks is an intricate process that involves combining layers, being mindful about input and output dimensions and building a robust data pipeline to feed the network with training samples. Even though building a topology graph is simple there are many details that need to be worked out in order to build its neural network (e.g. how can the input signals of two or more nodes be combined together in order to form the input for another set of layers?). Since converting each candidate topology to a neural network

is not a very viable approach considering the sheer number of candidate topologies in the population throughout the generations, an automated topology conversion and evaluation pipeline must be developed that can receive any topology, build and train the appropriate network and produce an evaluation. Programming such a routine by hand would be rather time consuming and burdersome, as there are many edge cases that must be taken into account. Thankfully, there already is a solution in the form of the Neural Operations Research and Development (NORD) package [23], a python framework that simplifies the process of building and evaluating networks. With it, it is possible to easily convert each candidate topology to an equivalent neural network and get consistent and comparable evaluation results that guide the evolutionary process. NORD is built on top of the PyTorch library which aims to streamline the network building phase by offering a higher level interface for creating deep neural networks. It is geared towards NAS research, allowing those interested to easily train populations of networks on a set of benchmark datasets and develop their algorithms and methods accordingly. The fundamental class of the framework is the descriptor, which is used to build a neural network in steps just like a networkx graph. After initializing the object, all that needs to be done is to call two simple methods for adding layers and connections between them and the network will be constructed. The *add_layer* method receives a reference to the PyTorch layer, the list of parameters as a dictionary and a layer name, for later reference. The *connect_layer* methods receives two layer names and forms a connection between them. Arguably the most useful feature during this process is the framework's ability to automatically combine layers, resolving dimensionality inconsistencies by adding intermediate layers that merge the output signals of incoming edges with the use of a suitable operator. On our side, all that needs to be done is to associate the layer information in the abstract modules with simple layers with a PyTorch module and a specific set of parameters. In this implementation, the layer naming convention includes not only the type of layer, but also its properties. For example, the layer name "CONV_3.N" refers to a convolutional layer with a kernel size of 3 and a default number of output channels (N denotes "normal" amount of output channels, in contrast to "H" which refers to half output channels of the default). Other parameters such as the input channels and the number of strides are fixed and standard for all layers. As a result, it is possible to build a PyTorch neural network with NORD from a neural module as follows: first, build the full computational graph of the candidate module and initialize a descriptor. Then, loop through all nodes in the full graph, read their layer name and use the *add_layer* method to add a new layer in the graph with the appropriate layer reference and parameters dict. In addition, assign a representative name to each added layer. After that, loop through all edges, and for each, figure out the layer names of each end and use them to add a connection to the descriptor. It should be noted that for some nodes in a graph, more than one PyTorch layers may

be added to provide better results, e.g. a module representing a single convolutional layer in the computational graph will correspond to a convolutional layer, a RELU layer, a batch normalization layer and a dropout layer, all sequentially connected in the descriptor graph.

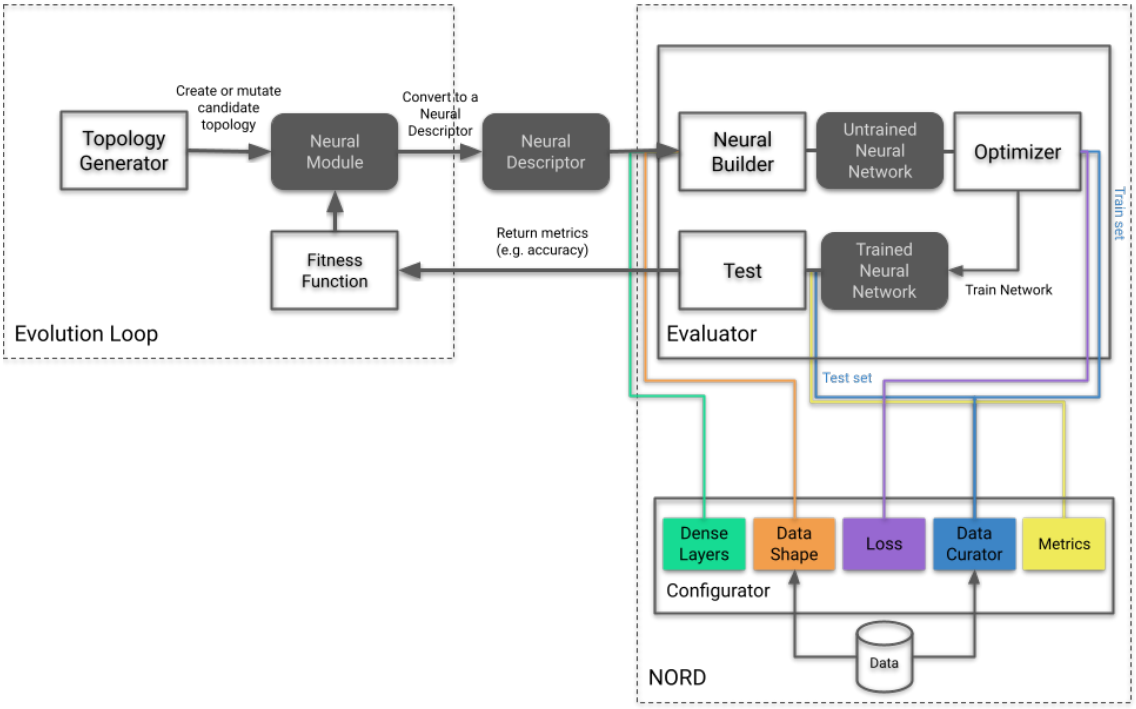


Figure 14: The process of evaluating a topology with NORD.

Before a fitness score can be obtained, it is necessary to set up the evaluation process. In a conventional PyTorch configuration, one would have to load the dataset for a local or remote repository, perform the desired transformation and augmentation operations to improve the quality of the training, split it into training and validation partitions and initialize the corresponding data loaders in order to train in batches of samples. Additionally, the optimizer and loss criterion would have to be initialized in order to have the ability to backpropagate the error on the network weights and adjust them through training. Finally, the training loop would have to be implemented, with the standard zero gradient/predict/backpropagate/optimize structure. Since this procedure is fairly standard, NORD conveniently abstracts the steps men-

tioned above in a single class, the *Evaluator*. This component allows for the direct comparison of deep learning models and methodologies by offering a platform where machine learning methods can be evaluated against each other using standardized benchmark datasets, configured in exactly the same way for each method. The same evaluator and dataset can be configured for a set of methods and evaluate each one using the same criteria, without deviating due to unrelated factors. To build this object, the only information needed is an optimizer class reference and the dataset name, along with some optional configuration parameters. The NORD package retains an internal library of common machine learning datasets which include CIFAR-10, Fashion-MNIST and others, each of which can be referenced using their names as strings. The package provides a different function for each dataset which is used internally to download the data from its official repository, perform all the standard operations to it and prepare it for training automatically. The data is stored locally, so it only downloaded once and stored to be used in consecutive sessions without the need to re-download each time. Depending on the dataset, NORD configures the proper loss function, removing the need to do it manually. Networks can be evaluated easily by making a call to the *descriptor_evaluate* function and providing a descriptor object, the number of training epochs and the dataset name. The functions takes care of the descriptor training and returns the value of the appropriate metric for the dataset (e.g. accuracy for classification problems), along with the training time that was required.

To summarize, we use NORD in order to evaluate candidate topologies with minimal friction. First, an evaluator is initialized, setting the target dataset for the NAS method. Then, for each candidate in the population, the full graph is extracted by calling the appropriate function on the top level module and is used to build a descriptor equivalent. Next, each descriptor is passed to the evaluator which constructs the corresponding network, uses the selected dataset to train it and returns its performance on the validation set. Finally, the acquired value is used as the fitness for each chromosome, following the procedure described in chapter 6.

The Module Manager

Implementing the candidate modules list, the notable modules list and the banned modules list in practice is very easy. The *ModuleManager* class wraps the three data structures and offers a set of easy to use functions that serve the various needs of the evolutionary algorithm. It is initialized by defining the three module containers as follows: the candidate modules is initialized as an empty dictionary, designed to hold associations of *ModuleProperties* and *TempPropertiesInfo* pairs. The *TempPropertiesInfo* class is an extension of the *PropertiesInfo* class, which wraps the average

fitness and occurrence count variables for an associated *ModuleProperties* object instance, extending it by including the *TTL* counter for a candidate module. The notable modules is also initialized as a dictionary, but unlike the previous container, it has content inserted into it immediately. Using an allowed layers list containing string representations of layers and their parameters, the corresponding neural modules of type "neural layer" will be build to form a set of *ModuleProperties-PropertiesInfo* where the *ModuleProperties* are extracted from the modules just formed and the *PropertiesInfo* is initialized with zeros on both their variables. These will be used to construct the first batch of new modules in the population offering a new level of abstraction to the candidates. Finally, the banned modules list is an empty list. It is not necessary to hold any type of association pairs in this list - its function is to simply contain irrelevant neural modules (or minimal identifying information for each of them, as will be shown in the next paragraphs).

The three containers are automatically updated by interacting with each other when the *on_generation_increase* callback function is triggered in the *ModuleManager* at the start of a new generation. The source of new module information is a process triggered after the evaluation of a candidate which determines if each of the modules in its hierarchy should be added or updated in the candidate modules. After a generation, the candidate modules dictionary should have several new modules, along with the corresponding *TempPropertiesInfo* instances, retaining their fitness information. The callback on the generation increase event performs the trade between the candidate and notable modules, and marks the modules of the two containers that should be removed. More specifically, modules whose *TTL* has reached 0 on this generation and can't participate in the promotion due to a low occurrence count, candidate modules that fail the comparison test due to a low average fitness and notable modules that are demoted due to being replaced by a new candidate module should be recorded in the blacklist. However, there is still the issue of size requirements for the three data structures. While the candidate and notable modules have a maximum size parameter to restrict the maximum size of their contents on RAM (and on the disk, when saving the evolution state) the banned modules list is unconstrained. Simply moving a *ModuleProperties* instance from the candidate or notable modules list to the blacklist is not feasible, since that data structure would then explode in size along with the whole process. At this point, the use of the blacklist should be considered: it is only utilized to check whether or not a module that does not appear to be in the candidate or notables list should be added to the candidate modules, or it is a module that should be ignored. This details highlights the solution to the problem of space. The actual properties (e.g. abstract graph, child module properties) of a banned module do not matter. The only information needed is a unique identifier. The hash of a module *ModuleProperties* instance conveniently happens to serve as one. The hash is a simple integer, unique for every *ModuleProperties* instance that

has a constant size, independent of the module size. Every time a new module must be evaluated regarding whether or not it should enter the candidates, we can just see if the hash of its *ModuleProperties* is in the banned modules list. If it is, it must be ignored. Using just hashes in the blacklist minimizes the space required for its contents. There is still a ceiling on how much data it can hold, but that number is much larger. On a modern computer, a few billion hashes can be recorded in the blacklist, which should be plenty of modules for the process to go through. There is still however, a speed optimization that can be easily implemented. A serial search must be performed for each module in a candidate to determine whether its hash appears in the blacklist. This means that, on an unsorted list, all elements of the list must be considered for every search invocation. As the size of the list grows, this could be a potential source of delays. To alleviate that, all that needs to be done is to keep the list sorted. Using the *bisect* package, new items can be introduced in a collection such as a list so that they occupy the proper position to keep the collection sorted. The package implements functions for finding the index of an item in order to maintain a sorted configuration in lists, arrays etc. When a new module is being searched on the blacklist, the result of running the "search" function is the index of the position that its hash should occupy, if it is to be inserted. Accessing that position directly and comparing the current hash content with the hash of the module informs the process of its next steps. If the two hashes are identical, then the module is in the banned modules list and should be ignored. Else, the module in question is to be recorded in the candidate modules. When a module from the candidate or notable modules must be placed in the blacklist, a similar procedure occurs. The bisect search function returns its appropriate position and the corresponding *ModuleProperties's* object hash is inserted there.

Experiments

Introduction

In order to assess the performance of our method, the proposed algorithm is evaluated by performing neural architecture search on a series of datasets. The problem domain consists of a time series classification dataset and 2 conventional image classification problems. The main motivation behind the choice of working with time series is to investigate how well NAS methods perform on data samples with an underlying temporal relationship. Working with time series offers a very important advantage over image datasets, which are traditionally preferred: the lower dimensionality of the data (1 dimension instead of 2 in the case of images) allows for the significantly faster training of the candidate networks, allowing researchers to evaluate their NAS algorithms in reduced time frames. Additionally, the two image classification datasets are used as benchmarks so that our method can be compared to other NAS methods, which generally illustrate their performance on such benchmarks.

The approach is similar for all datasets. For every dataset, the search space is defined by setting a list of allowed operations, which correspond to neural layers with specific properties that are to be used as the initial building blocks for the first neural networks in the search. In later stages, such operations are grouped together in graphs and are used through their inclusion in a higher level module in a candidate topology, as explained in the previous chapters. As far as the time series data is concerned, the actual neural operations are selected to form convolutional rather than recurrent neural networks. Even though historically RNN and LSTM cells have been used as the building blocks of networks in such problems in order to leverage their ability to accept sequences of variable length, they have some serious disadvantages, the main one being that RNN based networks are slow to train, which is amplified in the domain of NAS, where large populations of candidate topologies must be trained and evaluated. An alternative approach is to use CNNs with 1 dimensional convolutions which makes fitting networks on temporal data practically possible, circumventing the problem of parallelizing operations and achieving comparable results in faster time frames. In this case, the convolutional kernels operate by incorporating information

from data in the neighboring time steps to produce more elaborate internal signals of the network, much like in the case of images. As an additional benefit to choosing this approach for the time series data, there is a relative uniformity between the time series classification datasets and the conventional image classification datasets, since all problems in the experiments will be approached from the same angle (using mainly convolutional and pooling layers).

All experiments were conducted in the Google Colaboratory platform. Google colaboratory is a service which offers high end computational resources to academics for research purposes, free of charge. Certain restrictions (e.g. limited amounts of GPU access per day, lack of manual hardware choice) that are put in place to prevent abuse had an effect on the algorithm evaluation process. The produced candidate topologies were trained over a period of several days using a single T4 NVIDIA GPU. In order to allow the evolution process to operate for more than the default daily time limit imposed by the platform (which varies and can go up to 12 hours), a custom state saving system was developed to save populations of topologies and other relevant information, which is used in the next days to resume from where the search had left off, instead of starting from scratch each time. Nevertheless, even though the experiments are limited as far as NAS methods are concerned, the results clearly show the merit and performance benefits of the proposed method.

Human Activity Recognition

The first benchmark problem that was tackled in order to evaluate the proposed method was that of human activity recognition using accelerometer data from a mounted sensor. Casale et al. [24] have published a novel time series dataset posing the problem of classifying a set of 7 distinct actions performed by humans using the acceleration data provided by a chest-mounted sensor for the 3 axes. The sampling frequency of the accelerometer is 52HZ, meaning that 52 consecutive samples correspond to 1 second of data. This is used as the sampling window by NORD, with 26 lags overlapping between each pair of consecutive instances, forming 3 time series(52 logs each) per sample (one for each axis), representing a single second and corresponding to a target action. The authors collected the data by having subjects perform the same activity for 2 minute intervals where the activity would have to be stated at the beginning and would continue until the end of the time frame or the pressing of the interrupt button, ensuring that there are no border cases in the dataset, i.e. activities do not change throughout the span of the 1 second window. The 7 activities recorded were: working at a computer, standing up and walking and going up and down stairs, standing still, walking, going up and down stairs, walking and talking with someone, and talking while standing. Some classes are groups of activities with overlapping

items, which may make the task of classification slightly more difficult. The entirety of the data was collected from a group of 15 participants, who performed activities while equipped with the accelerometer sensor on their chests.

The following neural layers and property combinations are selected for the operations list: 6 convolutional layers composed of kernels with size $\in \{1, 2, 3\}$ and output channel count $\in \{256, 512\}$ and 3 average pooling layers and 3 max pooling layers with kernel size $\in \{2, 3, 5\}$. As mentioned beforehand, in order to provide better results, convolutional nodes are replaced with a convolutional/relu/batchnorm/dropout(with dropout probability equal to 10%) layer group. The defined search space using this set of operations is much larger than that of many other NAS publications that are often limited to 5-7 operations [9] [8], while this experiment uses 12 distinct operations. As far as the core evolution is concerned, the population has a size of 50 candidate topologies, the notable modules list has a maximum size of 15 modules (populated by 12 core modules at the start of the search), the base TTL value for candidate topologies is 3 generations, the minimum candidate topology observation count is 2 observations, with the fitness threshold activating at the end of every generation with the 40% worst performing topologies in the population being replaced by new schemes. This last setting is fine tuned so that the well performing topologies can remain in the population and have a chance to expand their structure, improving their performance, while the ineffective networks are replaced completely, offering a drastic contribution to the candidates list. The node mutation chance is set to 15% while the edge mutation chance is set to 55%. At most, only one mutation may occur at a given topology per generation. This is implemented by sampling from a uniform distribution on the range $[0, 1]$ and choosing the type of mutation based on the value of the sample s where $s \in [0, 0.15)$ corresponds to a node mutation, $s \in [0.15, 0.7)$ corresponds to an edge mutation and $s \in [0.7, 1]$ corresponds to no mutation. Each node mutation resulted in the generation of an abstract graph with two nodes (excluding the input and output nodes), rendering the following 5 combinations possible:

Using more than two internal nodes dramatically increases the number of possible graphs, greatly extending the time needed to promote new topologies. During evaluation, the max number of training epochs was set to 20, with the actual number varying per topology, calculated using equation 3. The evolution loop was executed for 20 generations. The experiment was conducted over the course of 5 days, which amounted to $\sim$ 15 hours of GPU time. The results are presented in figure 15.

In order to test the effectiveness of our method we test it against random search (dashed lines). The random search is implemented by setting the same fitness value for all candidate topologies, regardless of the modules' performance on the validation set. This gives all neural modules an equal chance of being promoted to notables, provided that they meet the occurrence threshold (and that the notables list is not full). The green lines in figure 15 show the fitness of the best network found at each

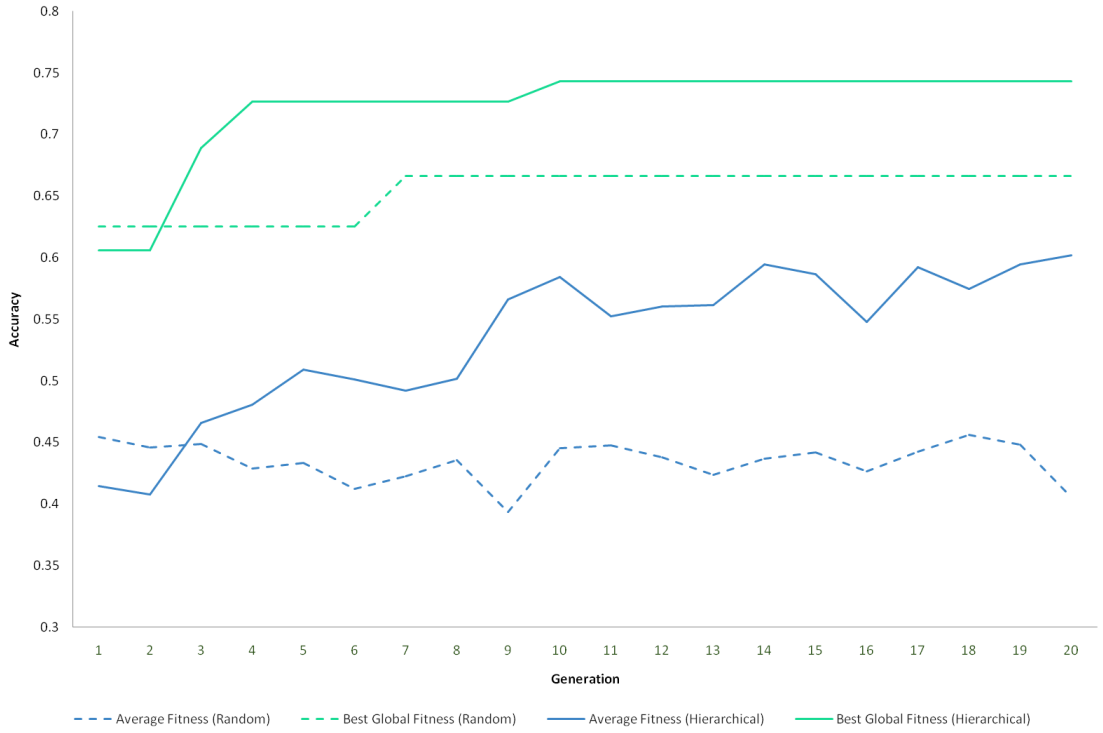


Figure 15: Average and best network accuracy for the population during the architecture search for the activity recognition problem.

point in the evolution, while the blue lines show the average fitness of the topologies in the population. Looking at the average fitness, it is evident that the proposed method works well in improving the quality of the population. The candidate and notable modules lists curate and highlight the most beneficial graph schemes, which are used in subsequent generation to build more sophisticated networks that generally perform better. This increases the average accuracy of the produced networks, raising the elimination threshold along with it. However, new topologies that replace old population members are not at a disadvantage compared to networks that survived the fitness check, since they are formed from notable modules of complexity appropriate to the current generation. On the other hand, the random search approach fails to make any improvements in the overall quality of the population. Sampling modules at random means that bad modules will not be removed from the population and will have an equal chance of being selected, leading to mediocre results. While the curve for the proposed method shows an increasing trend, the trend line for the random selection is horizontal. In fact, the average population fitness of the hierarchical approach moves into the range of the best fitness for the random method. In the

same figure, the solid green line shows a significant advantage for the hierarchical approach when the best constructed topologies are considered. In only 10 generations, the best topology accuracy improves by 14%, during which time a better performing topology is discovered on 3 occasions (generations 3,4,10). The best networks with the random method start off with similar performance as the proposed method but have just 1 slow improvement. In order for a network to improve in that case it must a) be good enough to remain in the population and b) perform a favorable mutation (which has a lower chance of happening when not taking into account the accuracy metrics). For continuous improvements, these two conditions must be met in consecutive generations, which is an event of decreasing probability as the number of operations increases. The hierarchical algorithm stores good modules into its memory (the notables list) and gives them priority over less performant ones by weighting the sampling process with the average fitness. This means that if a good module in the population gets ruined by an unfavorable mutation, its components may survive, even if it fails the fitness check at the end of the next generation.

It is worth studying the best network fitness curve for the hierarchical approach further, as it reveals details about the behavior of the algorithm. Between each improvement, there is a period of searching where better networks are not found. In those generations, the algorithm essentially generates populations of networks in order to reach the occurrence count for modules in the candidates list so that they can be tested against the notable modules. The three parameters tied to this behavior are the population size, the occurrence threshold and the base TTL. Increasing the population size would reduce these periods of inactivity where better networks don't occur, but would increase the time required to go through a single generation. Reducing the module occurrence threshold would make candidate modules perform the promotion check faster at the expense of an accurate assessment which may lead to the promotion of weak modules. Finally, while increasing the TTL may not have a noticeable theoretical effect, one should be mindful of the potential memory related problems that may arise from a very large list of candidate modules. Reducing the TTL, on the other hand, can lead to the preemptive rejection of potentially useful modules that failed to appear the required number of times in the population before they expired. Balancing these 3 parameters (along with secondary variables such as the mutation probabilities) is a crucial step that can determine the behavior of the algorithm. For our limited computational resources, a small population with low occurrence count and a moderate TTL was preferred to minimize the time needed to complete each generation, resulting in the observed behavior. If the algorithm were to be deployed at a high performance computational cluster, the generations where a better network is not discovered could be omitted by dealing with larger populations.

The best network after 20 generation achieves a validation accuracy score of 74.3% and has the following structure:

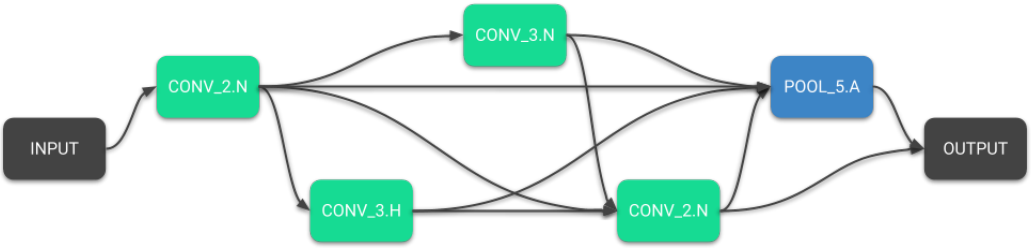


Figure 16: Discovered activity recognition network.

The final topology uses only 4 out of the 12 available operations. The evolution algorithm has built a network that uses convolutions at the start in order to produce latent signals which are then combined and passed through an average pooling layer that is again combined with the outputs of a previous convolutional layer to produce the output. This lines up with the way the deep convolutional networks for such problems are built when designed by a researcher.

When observing the graph, one can begin to visualize how the hierarchical layers are laid out. For example, the connections landing on the pooling layer are indicative of an edge connection on the abstract nodes of a higher level. It is possible to visualize the hierarchical structure by observing the abstract graphs of each level (depth):

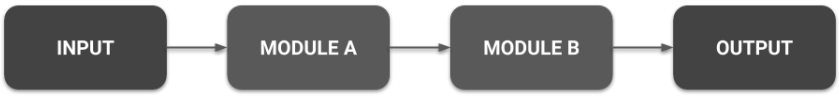


Figure 17: Root module

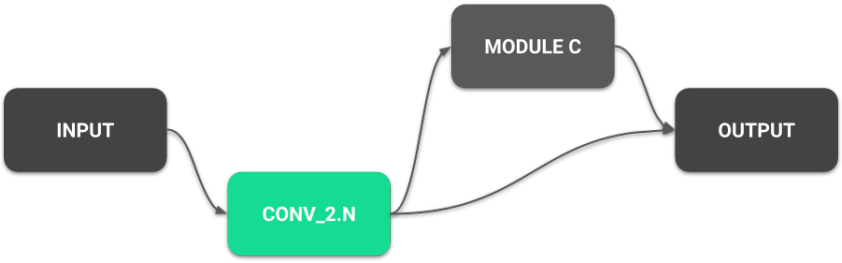


Figure 18: Module A

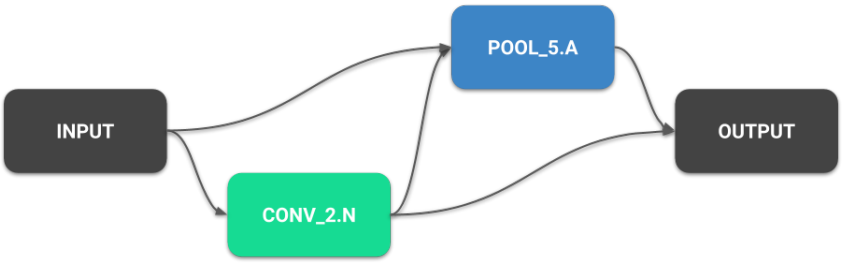


Figure 19: Module B

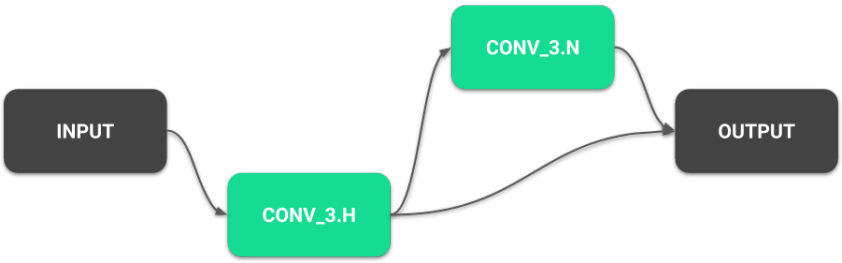


Figure 20: Module C

Each abstract graph represents a module that was sampled from the notable modules list. Different modules are combined in varying depths to construct a complete network that can perform the requested classification task. In this case, three hierarchical layers were formed as a result of the aggregation of node mutation operations into the sampled modules. At depth level 1, the network has a sequential structure with one node feeding data into the next, which in turn produces the output. Moving on depth level 2, the two nodes form a branching topology that propagates the data to a set of paths with different operations. The module on the deepest level replaces the abstract module on the first graph of depth 2, which creates a sub graph of neural operations. Admittedly, this is probably not how a human would form the hierarchy at this level, but it is very understandable for an evolutionary algorithm operating with heuristics to do so. The produced structure arises from the configuration of the hyperparameters of the graph generation: since exactly 2 nodes exist in any abstract graph, in order to add complexity to a submodule, one of the nodes must undergo mutation, increasing the depth of that branch. As far as the layer connections are concerned, the signals of many layers are combined due to a connection in the abstract graph of a parent module as expected, producing complex signals for the succeeding layers (e.g. the pooling layer near the end). The increased connection mutation

probability emphasizes edge additions, especially at the higher levels of the network, which creates topologies with many combinations of latent information in the inner layers of the network. Even though this generally produces more complex models that require more time for training and inference, it is generally a convenient way to ensure a baseline of performance in the accuracy of the model. Some connections may be unnecessary - but they rarely reduce the prediction quality.

The overall structure of the hierarchy is consistent with the hand built models. The most similar hand built architecture to our final result is probably InceptionNet. The InceptionNet model is comprised of a set of branching blocks that are connected sequentially . The produced topology resembles that form: on depth level 1, the graph is sequential, while the lower level blocks have branching structures. In this instance no abstract modules happen to be repeated (something that happens in InceptionNet). This is probably a result of the low number of generations in the evolution, although it could also be attributed to the algorithm building a more specialized network that better fits the data. In any case, the network has the core characteristics of human designed networks and performs well, which renders the proposed method satisfactory.

Fashion-MNIST

In addition to the time series datasets, NAS is performed on a set of conventional classification datasets. The first dataset, Fashion-MNIST, contains 28×28 sized grayscale images of 10 different classes of articles of clothing and accessories. The classes are the following: t-shirt, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag and ankle boot. There are 60000 images in the training set and 10000 images on the test set. In order to work with images instead of time series, all that needs to be done is to define the set of valid layers that can be used as building blocks by the algorithm. Since training on 2D data involves a higher information load on the network, the training and evaluation times are increased significantly. In a attempt to speed up the search process, we define a set of 6 neural operations: 3 2D convolutional layers with an output channel count of 32 and kernel size $\in \{1, 2, 3\}$ and 3 max pooling layers with kernel size $\in \{2, 3, 5\}$. Our algorithm is configured to operate on a population of 20 modules (to adhere to temporal and size limitations), with a maximum size of 10 modules in the notable modules list and a base TTL of 4 for candidate modules. The rest of the settings are identical to the activity recognition network experiment. We perform a limited architecture search for 20 generations and compare it with the results from a random search with the same settings. The quality of the discovered networks is pictured in the graph below:

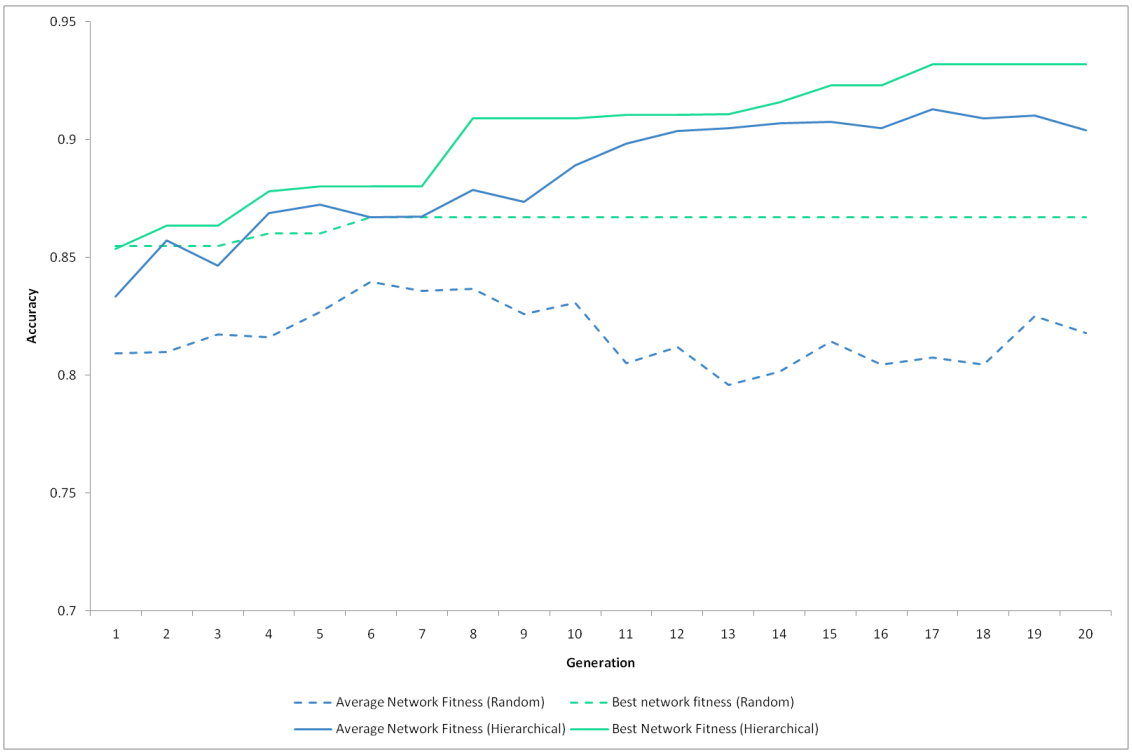


Figure 21: Average and best network accuracy for the population during the architecture search for the Fashion-MNIST problem.

Our approach manages to discover a topology with an accuracy score of 93.2% in generation 17. As expected, the random search offers inferior results in both average network accuracy and best topology performance, achieving a top accuracy of 86.7%. In fact, the average accuracy of the networks in the population on our approach surpasses the best performance of the random approach early on thanks to the sampling technique used in the notable modules. Our approach compounds knowledge from previous generations, which leads to the construction of better modules and networks possessing good predictive abilities. The accuracy score achieved using this approach is 3.71 percentage points from the state of the art score, achieved using a fine tuned DARTS based solution [25]. This is a gap that could theoretically be covered if the search runs for a higher number of generations or performing augmentation on the dataset. In figure 21, there is no sign of convergence in the curves, which indicates that a higher score is indeed possible. Additionally, DARTS works on the cell level, which means that the search space is more constrained, a side effect that limits the structure of the produced topologies.

The best network topology uses 23 neural nodes and 93 connections between them. There are a total of 10 abstract modules used in the hierarchy, some of them repeating

multiple times (e.g. modules G and J in the figures below). The evolutionary process has maintained all 3 convolutional operations to be utilized at some point in the network, but only 1 pooling operation (max pooling with a kernel size of 3). Since the full network is too dense to be presented here in a meaningful way, we present the best topology hierarchically, from top to bottom:

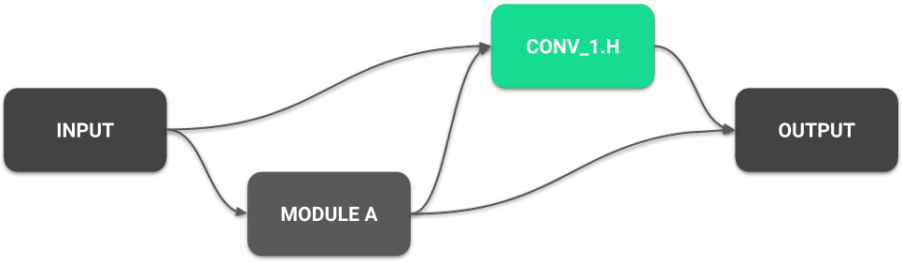


Figure 22: Root module

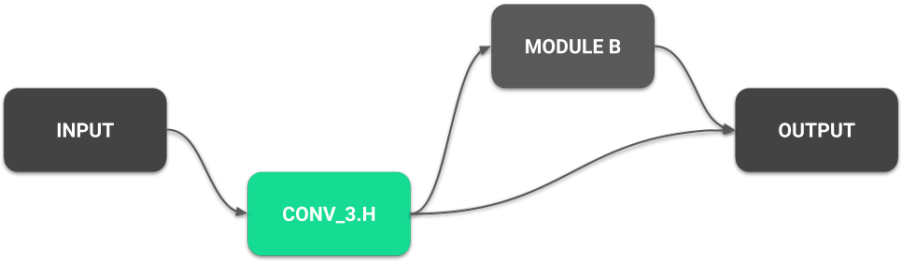


Figure 23: Module A

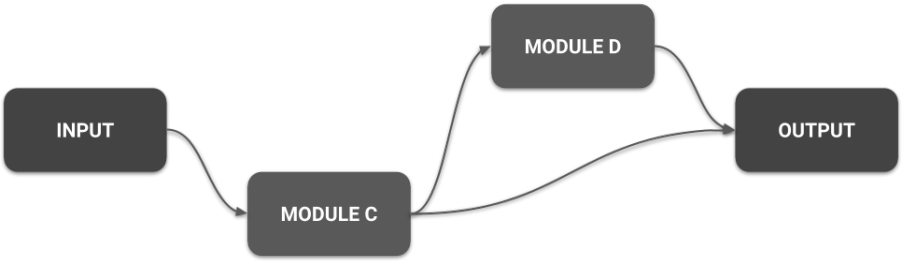


Figure 24: Module B

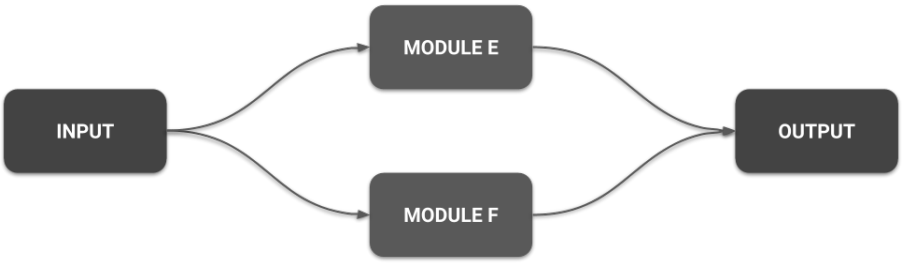


Figure 25: Module C

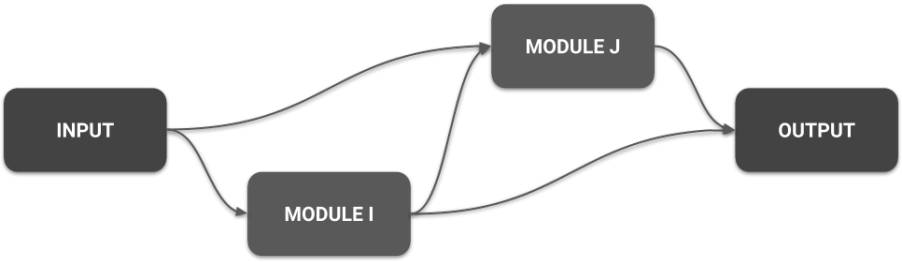


Figure 26: Module D

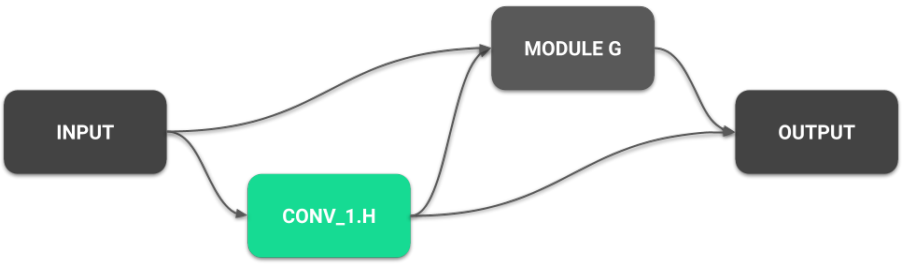


Figure 27: Module E

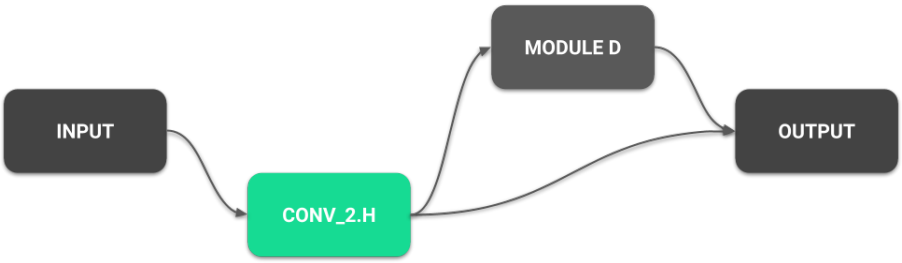


Figure 28: Module F

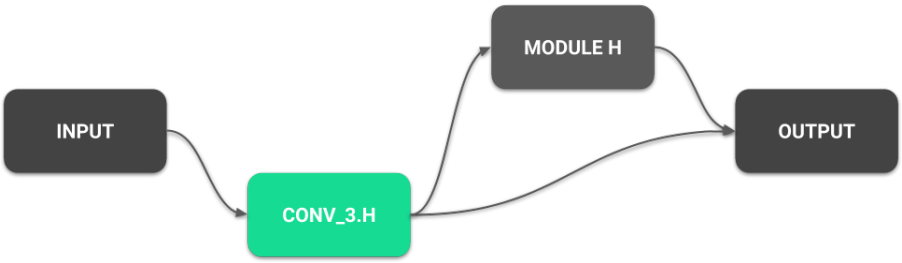


Figure 29: Module G

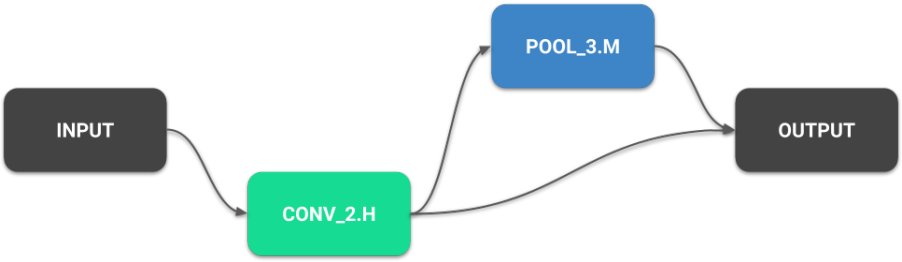


Figure 30: Module H

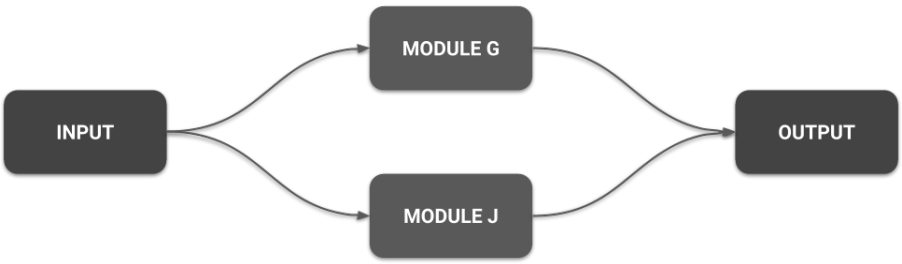


Figure 31: Module I

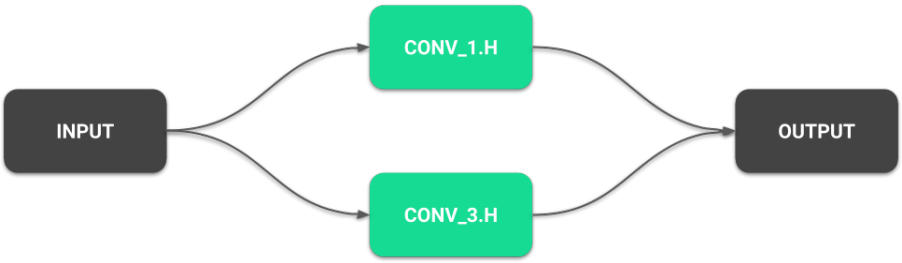


Figure 32: Module J

CIFAR-10 (NAS-Bench-101)

For the last experiment, we apply our dynamic hierarchical NAS algorithm to the CIFAR-10 dataset using the NAS-Bench-101 database [9]. While our approach is designed with global search spaces in mind, we wanted to test how well it works on cell search spaces too, where multiple restrictions are put in place. In this case, the search space comprises of all cells with up to 7 nodes (neural operations) and 9 edges (connections), using only 3 available operations: 1×1 and 3×3 convolutions (grouped with an additional batch normalization and a RELU layers) and a 3×3 max pooling operation. There are 423624 possible cells in the search space. The discovered cell is placed in designated positions in a larger, fixed convolutional topology to form a full network.

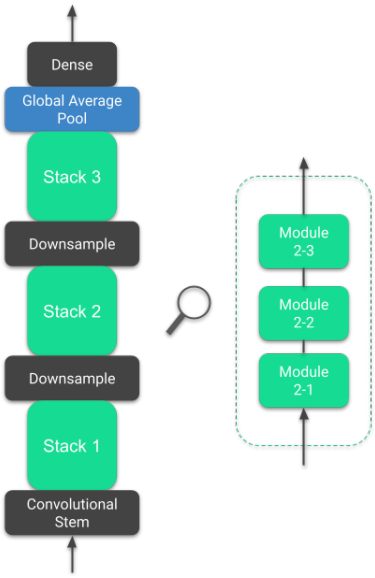


Figure 33: The NAS-Bench-101 network skeleton.

NAS-Bench-101 contains records of accuracy for all networks built by the cells in the defined search space which can be accessed using the cell topology adjacency matrix as the key. After generating or mutating a candidate topology, we build the corresponding descriptor object, pass it to NORD, which in turn performs the lookup operation and returns the accuracy record instantly, without requiring the training and evaluation of the network.

Since there are significantly fewer operations in this experiment, a smaller population size of 10 is set. Candidate modules reach their occurrence threshold much faster, so new schemes are introduced quickly to the notable modules list. One thing

that must be taken into account is the restrictions on the number of nodes and edges. If a candidate cell topology exceeds the imposed limits, it can't be evaluated. Nevertheless, the search algorithm manages to find a top performing cell topology in just 6 generations with an accuracy of 94.8%.

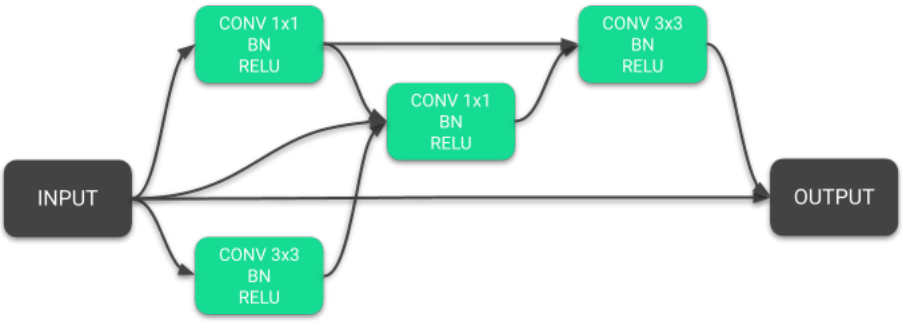


Figure 34: Discovered NAS-Bench-101 cell for the CIFAR-10 dataset.

Conclusion

In this thesis we propose Dynamic Hierarchical NAS, an evolution based NAS technique that discovers new topologies in a global search space by continuously curating a list of effective network submodules, combining them based on their fitness scores to form full networks with progressively better performance scores. Our method takes advantage of a novel hierarchical network representation that allows for the formation of subgraphs of arbitrary complexity that can be broken down into distinct parts, each of which is assigned a fitness score. Using 3 auxiliary module directories, we preserve a knowledge base of modules, gradually introducing more abstract schemes that allow for the rapid transformation of existing candidate networks, dramatically reducing the time spent on mutation operations. While the approach still suffers from some common problems present in most evolutionary algorithms, mainly the large amounts of time required to train and evaluate candidate networks such negative effects are mitigated in two ways. First, by allowing large scale mutations on networks, the population can converge faster compared to traditional mutation approaches that mutate one neural operation or connection at a time. Second, by using a simple heuristic to estimate the training time required for each candidate based on its graph complexity. Furthermore, the modular design of the method theoretically allows for the complete replacement of the training and evaluation components with a predictive model, similar to the one used in [16] that estimates the performance of the candidates much faster.

We apply the proposed algorithm to 3 datasets and show how a set of neural operations can be modified to conform to the special properties of the data. We perform topology search on a time series dataset and a limited search on 2 conventional image classification datasets and manage to develop competitive networks in short amounts of time.

The code for this project can be found in `https://github.com/ArisChristoforidis/Dynamic-Hierarchical-NAS`

References

[1] H. Rakhshani, H. I. Fawaz, L. Idoumghar, G. Forestier, J. Lepagnot, J. Weber, M. Brévilliers, and P.-A. Muller, "Neural Architecture Search for Time Series Classification," 2020.

[2] G. Kyriakides and K. Margaritis, "An Introduction to Neural Architecture Search for Convolutional Networks," pp. 1–17, 2020. arXiv: 2005.11074. [Online]. Available: http://arxiv.org/abs/2005.11074.

[3] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, and B. Hodjat, "Evolving deep neural networks," *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pp. 293–312, 2018. DOI: 10.1016/B978-0-12-815480-9.00015-3. arXiv: arXiv:1703.00548v2.

[4] T. Elsken, J. H. Metzen, and F. Hutter, "Simple and efficient architecture search for convolutional neural networks," *6th International Conference on Learning Representations, ICLR 2018 - Workshop Track Proceedings*, pp. 1–14, 2018. arXiv: 1711.04528.

[5] E. Byla and W. Pang, "DeepSwarm: Optimising Convolutional Neural Networks Using Swarm Intelligence," *Advances in Intelligent Systems and Computing*, vol. 1043, pp. 119–130, 2020, ISSN: 21945365. DOI: 10.1007/978-3-030-29933-0_10. arXiv: 1905.07350.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-Decem, pp. 770–778, 2016, ISSN: 10636919. DOI: 10.1109/CVPR.2016.90. arXiv: 1512.03385.

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 07-12-June, pp. 1–9, 2015, ISSN: 10636919. DOI: 10.1109/CVPR.2015.7298594. arXiv: 1409.4842.

[8] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning Transferable Architectures for Scalable Image Recognition," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 8697–8710, 2018, ISSN: 10636919. DOI: 10.1109/CVPR.2018.00907. arXiv: 1707.07012.

[9] C. Ying, A. Klein, E. Real, E. Christiansen, K. Murphy, and F. Hutter, "NAS-BENCH-101: Towards reproducible neural architecture search," *36th International Conference on Machine Learning, ICML 2019*, vol. 2019-June, pp. 12 334–12 348, 2019. arXiv: arXiv:1902.09635v2.

[10] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, "Hierarchical representations for efficient architecture search," *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, pp. 1–13, 2018. arXiv: 1711.00436.

[11] P. J. Angeline, G. M. Saunders, and J. B. Pollack, "An Evolutionary Algorithm that Constructs Recurrent Neural Networks," *IEEE Transactions on Neural Networks*, vol. 5, no. 1, pp. 54–65, 1994, ISSN: 19410093. DOI: 10.1109/72.265960.

[12] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002, ISSN: 10636560. DOI: 10.1162/106365602320169811.

[13] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, pp. 1–16, 2017. arXiv: 1611.01578.

[14] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, "Efficient architecture search by network transformation," *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, pp. 2787–2794, 2018. arXiv: 1707.04873.

[15] H. Cai, J. Yang, W. Zhang, S. Han, and Y. Yu, "Path-level network transformation for efficient architecture search," *35th International Conference on Machine Learning, ICML 2018*, vol. 2, pp. 1069–1080, 2018. arXiv: 1806.02639.

[16] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L. J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive Neural Architecture Search," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11205 LNCS, pp. 19–35, 2018, ISSN: 16113349. DOI: 10.1007/978-3-030-01246-5_2. arXiv: 1712.00559.

[17] V. Lopes and L. A. Alexandre, "HMCNAS: Neural Architecture Search using Hidden Markov Chains and Bayesian Optimization," pp. 1–9, 2020. arXiv: 2007.16149. [Online]. Available: http://arxiv.org/abs/2007.16149.

[18] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient Neural Architecture Search via parameter Sharing," *35th International Conference on Machine Learning, ICML 2018*, vol. 9, pp. 6522–6531, 2018. arXiv: 1802.03268.

[19] G. Bender, P. J. Kindermans, B. Zoph, V. Vasudevan, and Q. Le, "Understanding and simplifying one-shot architecture search," *35th International Conference on Machine Learning, ICML 2018*, vol. 2, pp. 883–893, 2018.

[20] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," *7th International Conference on Learning Representations, ICLR 2019*, pp. 1–13, 2019. arXiv: 1806.09055.

[21] D. B. Johnson, "Finding All the Elementary Circuits of a Directed Graph," *SIAM Journal on Computing*, vol. 4, no. 1, pp. 77–84, 1975, ISSN: 0097-5397. DOI: 10.1137/0204007.

[22] N. Shervashidze, P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-Lehman graph kernels," *Journal of Machine Learning Research*, vol. 12, pp. 2539–2561, 2011, ISSN: 15337928.

[23] G. Kyriakides and K. G. Margaritis, "Towards automated neural design: An open source, distributed neural architecture research framework," *ACM International Conference Proceeding Series*, pp. 113–116, 2018. DOI: 10.1145/3291533.3291564.

[24] J. Vitrià, J. Sanches, and M. Hernández, "Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6669 LNCS, no. October 2017, 2011, ISSN: 03029743. DOI: 10.1007/978-3-642-21257-4.

[25] M. S. Tanveer, M. U. Karim Khan, and C.-M. Kyung, "Fine-Tuning DARTS for Image Classification," pp. 4789–4796, 2021. DOI: 10.1109/icpr48806.2021.9412221. arXiv: 2006.09042.