Magnus Poppe Wang

Evolving Knowledge And Structure Through Evolution-based Neural Architecture Search

Master's thesis in Artificial Intelligence
Supervisor: Massimiliano Ruocco, Stefano Nichele
June 2019

Magnus Poppe Wang

Evolving Knowledge And Structure Through Evolution-based Neural Architecture Search

Master's thesis in Artificial Intelligence
Supervisor: Massimiliano Ruocco, Stefano Nichele
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

Meta learning is a step towards an artificial general intelligence, where neural architecture search is at the forefront. The methods dominating the field of neural architecture search are recurrent neural networks and evolutionary algorithms. The state-of-the-art evolution-based neural architecture search algorithms evolves only the structure of the neural networks. This thesis proposes an evolution based neural architecture search method focused on transferring both structure and knowledge together through the generations. Experiments are conducted to review multi-objective optimization for evaluating neural networks through both their knowledge and structure. A trade-off when transferring knowledge is also transferring bad traits such as overfitting. An alternative pattern-based representation is tested to explore how much of the knowledge should be transferred. A comparison between local search hill climb and evolution is also conducted to find the effects of having a population by it self. The proposed system finds a top performing architecture in short time. Transfer learning proves to increase the both the stability and speed of the evolution-based neural architecture search. Optimizing neural networks through multi-objective optimization proves to work well given good objectives. Optimizing on structure yields a much more diverse population than optimizing on knowledge. The population is important to have as the choices taken by the search are crucial for its success.

Preface

This thesis was written starting the fall of 2018 until spring of 2019 at the Norwegian Institute of Science and Technology, Faculty of Information Technology and Electrical Engineering, Department of Computer Science under the Norwegian Artificial Intelligence lab.

A special thanks goes to my supervisors, Massimiliano Ruocco and Stefano Nichele for their guidance, insightful tips and discussions. Their advice made this thesis possible.

Magnus Poppe Wang

Trondheim, June 18, 2019

Contents

List of Figures

List of Tables

List of Acronyms

EA:	Evolutionary Algorithms
NAS:	Neural Architecture Search
NN:	Neural Network
CNN:	Convolutional Neural Network
RNN:	Recurrent Neural Network
NEAT:	Neuroevolution of Augmented Topologies
RL:	Reinforcement Learning
DAG:	Directed Acyclic Graph
ML:	Machine Learning
AutoML:	Automatic Machine Learning
AGI:	Artificial General Intelligence
CPU:	Central Processing Unit
GPU:	Graphical Processing Unit
vRAM:	Video Random Access Memory

Chapter 1

Introduction

This chapter is an introduction to the to this thesis, beginning with describing the motivation behind doing this research in section 1.1. Section 1.2 elaborates on what goals this thesis aims to achieve and what research questions needs answering. These research questions are addressed using the research methods found in section 1.3. A short list of contributions are listed in section 1.4 but further elaborated on in chapter 6. Finally, the thesis structure is described in section 1.5.

1.1 Background and Motivation

In recent years there has been a growth in the field of AutoML research both in universities and industry. The focus of these studies are building deep neural networks perfectly designed for one or more specific datasets [26; 8; 23; 39; 5; 30; 24]. The goal of these studies is beating the human expert designed deep neural networks (DNN).

As described in [22], there are two forms of automatic machine learning (AutoML), narrow AutoML and generalized AutoML. The narrow AutoML is an algorithm where automatic machine learning is achieved with some expert knowledge required by a user. This expert knowledge is mainly used for configuring the algorithm and optimizing the datasets. The algorithm will do the rest. Generalized AutoML requires no expert knowledge and may be configured by any user.

Generalized AutoML can be seen as a big step towards an Artificial General intelligence (AGI) [22]. Such a system can explore by it self and interpret the data collected from exploration. When able to interpret the collected data, learning algorithms can learn from the experiences creating a better AutoML algorithm. For such a system to be a true AGI, the system will have to learn by doing then improve upon it self on a higher level than the reinforcement learning algorithms of today. This can be seen as a reinforcement learning algorithm where the reward function is learned. The system may be configured by an expert human initially.

Whether such a system will ever be discovered is a disputed topic among the top AI researchers [37]. While some believe that this technology will be discovered and usable within 50-100 years, others believe it might never be discovered. Such a system is not possible today due to a lack of compute power. By comparison, simulating the neurons in of a human brain will require about 10^{17} floating point operations per second (FLOPS), which is about the same amount of power as the Sunway TaihuLight supercomputer. This computer was the worlds fastest in 2016 [37].

Some of the biggest advancements of learning algorithms in recent years has come through Deep Learning (DL) [21]. DL is behind advancements in natural language processing (NLP), image classification, recommendation systems and more. Many of the top companies has already started using this technology in their products. Google has based made huge progress in their machine translation software Google Translate using recurrent neural networks. Many of the top car manufacturers are using the technology to make their cars safer and more automated. Amazon is basing their product recommendations on Deep learning recommendation systems. This is just a few examples.

Deep learning has some problems that has become huge research fields within AI. The main problem is that DL is seen as a black box operation. Industry hesitates to adopt deep learning for this reason. This has resulted in the field of explainable AI. Designing a Neural network that performs well is also hard. The process of designing them is more art than science [18]. This spawned the field of neural architecture search (NAS).

Deep neural network performance is heavily dependant on their design. The best man-made DNNs are made by trial and error. Many of the top DNN designs are also re-used and re-trained for new tasks [14; 39]. The most recent advances in DNN architecture and design is however made by algorithms [39; 24]. Evolutionary algorithms has seen great success in this field.

The state-of-the-art evolution-based neural architecture search algorithms are only focused on evolving the structure or architecture of a neural network. This is inline with reproduction in nature, where genes are joined from both parents to make the child. The genes is in this case the architecture. For humans and some animal species like the orangutan, the child also learns from the parents after birth - passing on knowledge. This is an aspect that the current neural architecture search algorithms have mostly overlooked so far.

A meta learning algorithm which can adapt to multiple learning tasks has to be discovered before we achieve generalized AutoML [22]. Neural architecture search is a step in the right direction. The bleeding edge NAS algorithms search for neural networks that specializes in learning a single [26; 8; 23; 39; 5; 30; 24] or a handful of tasks [15].

The problem with these algorithms is that they're slow to find good architectures. Many of the algorithms are however scalable, opening the possibility of massively parallelizing the workloads over distributed compute systems. Some search algorthms use up to 500 graphics processing units to achieve better than human performance [39; 24]. These GPUs runs for days. This kind of compute power requires cloud computing systems or large data centers.

The NAS component in AutoML can be seen as the learning to learn component in this process. Interpreting explored data not in the scope of NAS, since the algorithm learns the structure of the agent model but does not automate how the data is structured and interpreted. Only one part of AutoML is then automated, meaning expert knowledge is still required to use these algorithms.

1.2 Goals and Research Questions

There are many different methods for discovering neural network architectures. Progress has been made by reinforcement learning [8; 5; 23; 39; 30], evolutionary algorithms [24; 15; 26; 31] and many other methods such as proxyless tasks [6] and monte-carlo treesearch [27]. All of these algorithms shows immense promise in the field and all of them beat the top human expert designed neural networks in terms of performance.

Evolutionary algorithms have the advantage of being easily parallelized. There are also a lot of freedom both in how the algorithm is implemented and how the neural networks are represented. A good representation makes changes fast and easy to apply. Transfer learning has also been tested in creative ways, like to learn multiple tasks [15]. This showed a huge speedup can be achieved by first training the neural networks on a simple structure before training on a more advanced one.

Goal 1 *Explore and compare what effects diversity has on a traditional explo-*

ration based evolutionary algorithm evolving neural architectures.

Research Question 1.1 *What objectives can be used to maintain diversity in a population of neural networks when using multi-objective optimization?*

Research Question 1.2 *How does an evolutionary algorithm compare to hill climbing where no population is maintained?*

The state-of-the-art neural architecture search algorithms include implementations based on evolutionary algorithms. These implementations uses immense computing power to parallelize the learning tasks which is not viable for many use cases. There are many ways to optimize evolution based search. One can try different measures of performance in the elitism step or use local search within the evolutionary algorithm to enhance the choices made when mutating or performing crossover. These optimization may speed up the search process.

Maintaining diversity in a population is a challenging aspect of evolutionary algorithms. What constitutes diversity for neural network classifiers? Differences can be both found in the topology of the networks and in the weights used with the topology. These research questions aims to answer whether good diversity measures can speed up neural architecture search or if diversity yields any positive effects at all?

Goal 2 *Explore transfer learning as a method to speed up the evaluation step of the evolutionary algorithm.*

Research question 2.1 *What effects does transfer learning have on the evaluation step?*

Evolutionary algorithms are based on transferring good features of full solutions down through generations of solutions. Mutation and crossover is used to improve upon existing solutions. For improvements, good features of the predecessor solution needs to be transferred over to the successor.

When augmenting the topology of a neural network, some of these features are the components of which the topology is built. Other features include the weight configuration each component holds. This research question will experiment with transferring weights as well as topological features down through generations.

For neural networks, evaluation requires training which is a compute intensive tasks that runs for a long time. Evaluation is what slows down evolution based neural architecture search. Can transfer learning from a predecessor to a successor with only small changes to its architecture yield a speedup in the evaluation step?

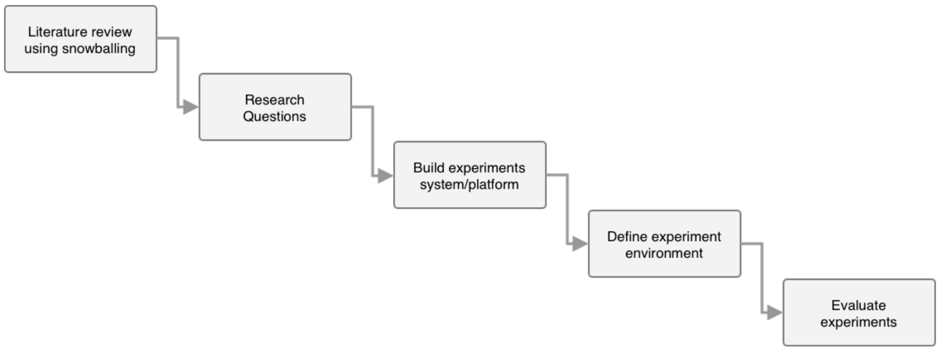

Literature review using snowballing
Research Questions
Build experiments system/platform
Define experiment environment
Evaluate experiments

Figure 1.1: Research process steps

Research question 2.2 *What is the effects of transferring fewer weights trained with different networks?*

Combining knowledge from different neural networks would be a very good crossover operator. When both changing the structure of the neural network and it's knowledge, a neural network is truly evolved. Transfer learning is however not always positive as it might transfer bad features such as overfit. Using an alternative representation with a fewer transferable weights gathered from different, will transfer learning avoid transferring memorized data-feature links? Can weights from different networks be joined together?

1.3 Research Method

This is an experiment based thesis as described by [Oates] where experiments is used as a strategy to connect cause-effect relationships. When connected, these relationships should either try to prove or disprove the hypothesis.

To address Goal 1, a system was created that evolves neural architectures using a traditional evolutionary algorithm combined with stochastic gradient decent. The elitism step has different modes for regular- or multi-objective sorting. To answer research question 1.1, two different sets of objectives will be evaluated. The first with a focus on diversity in how the neural networks are structured and the other focusing on diversifying the knowledge of a neural network. These objective-sets with both be evaluated against the baseline which uses a weighted sum of test accuracy and overfitting described in 3.2.5.

Modifications to the core evolutionary algorithm was made to answer research question 1.2. The modifications adapted the system to use a hill climbing local search method instead of evolution. This aims to answer whether a population is useful at all. Only iterating upon a single solution may be faster at finding a good architecture than when iterating on multiple solutions.s

For Goal 2, transfer learning was added into the systems. Addressing Research question 2.1, two simulations of the baseline will be tested. One with and one without transfer learning enabled. Research question 2.2 needed a new representation for testing what weights should be kept. The genotype representation changed from being closely modelled after a neural network to being only a subset of the neural network in the form of a pattern. Within the pattern, weights are saved, but between each pattern, weights are set randomly. These two different transfer learning techniques will be compared to see whether transfer learning is best suited if all possible weights are transferred or not.

1.4 Contributions

The main contributions of this thesis are:

1. The evolution based neural architecture search system described in chapter 3. An evolutionary algorithm that evolves neural network architectures and trains them using stochastic gradient decent.
2. A scalable training system for use with the TensorFlow machine learning framework that can be used either through SSH or MPI.
3. Two different representations to use with the NAS system, one being closely modelled after a neural network, described in chapter 3.2.1 and one pattern based, described in chapter 3.4.1.
4. Three different sorting techniques, one based on weighted sum and the others based on multi-objective optimization. This is described in chapter 3.2.5
5. A hill climbing local search variant of of the neural architecture search system as described in chapter 3.3

1.5 Thesis Structure

Chapter 2 contains the background theory required to read and understand this thesis. This includes Evolutionary algorithms and it's various characteristics, Hill climbing local search, deep learning and neural networks. Chapter 2.2 contains a summary of some of the most promising the state-of-the-art neural architecture search algorithms reviewed for this thesis.

Chapter 3 goes into detail about the architecture of the proposed system(s). EA-NAS is the main evolutionary algorithm tested here, with a indirect representation used as a genotype. This genotype is very similar to the structure of a neural network. Pattern-NAS changes this representation by abstracting it the neural network-like genotype representation into patterns. Local-NAS is based on the same system as EA-NAS but using a hill climbing local search instead of evolution.

Chapter 4 discusses the plans and details for the experiments of this thesis and why the experiments are done. Details like parameters used, what sorting objectives have been included and everything else needed to recreate the results of the experiments.

Chapter 5 is about results. Starting with a comparison with state-of-the-art and baseline before going through the results in more detail. What kinds of networks are created and what are the effects of diversity and transfer learning? A discussion on these subjects is found in the last section of chapter 5.

Chapter 6 concludes by connecting the experiments with the research questions stated in section 1.2, goes through the contribution of this thesis and what further work can be done on this subject.

Chapter 2

Background Theory and Motivation

This chapter will cover the theory behind this thesis. Starting with the background theory surrounding evolutionary algorithms and neural networks, section 2.1. A short description of how the literature review was conducted follows in section 2.2. The literature review follows in section 2.3, reviewing the state-of-the-art neural architecture search and its origins.

2.1 Background Theory

This section provides an overview of what the basic theory needed to understand this thesis. Starting with evolutionary algorithms and how they work. EA is followed by artificial neural networks, deep learning, transfer learning and details about the architectural components used with this thesis. The section serves as a theoretical component the reader can fall back on if needed.

2.1.1 Evolutionary Algorithms

Evolutionary algorithms are a set of optimization algorithms for optimizing in an unknown environment. These algorithms are inspired by biological evolution, with its search operators being natural selection, mutations and reproduction. This algorithm is based on making small changes to each gene and testing their fitness. For this algorithm to work properly, these changes needs to be computationally inexpensive so that the time wasted on bad changes are minimal. All individuals in a population used by evolution is always complete solutions. Evolutionary algorithms are very loosely defined because each optimization problem is different. This implies that the implementation also is very different for each problem. The following is the theory behind the most common operations without any implementation specific details. All information is gotten from [12].

Algorithm outline

```
$population \leftarrow initializeRandom()$
$fitness \leftarrow calculateFitness(population)$
while $generation \leq generations$ do
    $selection \leftarrow tournamentSelection(population)$
    for $individ \in selection$ do
        if $chance$ then
            $offspring \leftarrow crossover(individ, selection.next())$
        else
            $offspring \leftarrow mutate(individ)$
        end if
        $population \leftarrow offspring$
    end for
    $fitness \leftarrow calculateFitness(population)$
    $population \leftarrow elitism(population)$
end while
```

Representation

Since an evolutionary algorithm is a form of local search, the representation is always complete solutions. How the population is represented depends on the implementation. There is often a genotype representation and a phenotype representation. The genotype is an abstract model of the phenotype. The phenotype is the real model that is being optimized and is required for calculating the fitness. When using the genotype, the representation is called indirect. Using only the phenotype representation is allowed and is called a direct representation.

The genotype representation describes the genes that are changed by the mutation and crossover operators. The way the genotype is represented is up to the implementation, but some forms of evolutionary algorithms holds a special representation for the genotype. For genetic algorithms, the genotype is often a binary string where each number holds a special meaning for the phenotype. A change in the genotype will cause a change in the phenotype. Creative representations are then beneficial to the performance of the evolutionary algorithm as changes made to the genotype will decide the form and performance of the phenotype.

The process of converting the genotype to the phenotype is called decoding. Depending on the implementation, this might be a one-way conversion. Some representations allow for encoding the phenotype into a genotype, although it might be un necessary due to the flow of the algorithm.

Selection

There are two stages within the traditional evolutionary algorithm where selection occurs. The first is selecting what individuals in the population is to be mutated and the second is selecting which individuals survive.

- **Roulette selection**, also often referred to as *fitness proportionate selection* is a stochastic selection operator. Each individual in the population gets assigned a probability for being selected where probability is their fitness value proportionate the fitness of the rest of the population. A probability distribution is created by normalizing the individual fitness value on the fitness of the entire population.
- **Tournament selection** selects two or more randomly selected individuals and pits them against each other. The one with the best objective fitness value is selected. This makes the selection method stochastic as the probability of each individual is equal for being selected, but the probability of being the best in the tournament is not.
- **Truncation selection** is a form of selection where all individuals in the population is sorted on the objective fitness value. The individuals with the best fitness are selected. This operator is common to use with elitism and often used in combination with multi-objective optimization algorithms.

Mutation

Mutations are based on making a small change to a single individual in the population. The type of change is problem dependant. Changing a number in sudoku or changeing what city comes next in a traveling salesman problem are examples of mutations. These changes do not need to be legal for the solution as illegal solutions can be filtered out on the elitism stage. An example of this is for optimizing a solution to a sudoku board. A mutation might mean changing a the value of a cell on the board. Changing a cell to 9 when there already exists a

9 on the same row would be illegal for the final solution. These sort of mutations might be dealt with using a punishment in the fitness operator.

Crossover

The crossover operator is inspired by the sexual reproduction found in nature. This operator is used to join two solutions into a single new solution. unlike mutations, there are some commonly used ways of doing crossover:

- **Striped crossover** uses every other gene from each of the parent genomes. If the first gene is selected from $parent_1$, the second gene will always be selected from $parent_2$
- **K-Point crossover** is based on selecting $K = 1$ or more points. These points determine which of the parent genomes is copied over to the offspring genome. For each point, one of the two parents are selected. Genome information for the next point is always gotten from the other parent. For $K = 1$, a single point is selected in the genome. If this point is in the middle of the genome, half of each parent will be used in the offspring genome.

Fitness

The fitness operator in evolutionary algorithms scores each individual in the population. This score is used to rank each individual in the selection and elitism stages. This is of course heavily dependant on each implementation. Like mentioned in the mutation section above, punishments can be applied here for incomplete or illegal solutions.

Elitism

Elitism is inspired by Darwinian evolution, survival of the fittest. Here, the best individuals in the population survives while the worst dies. By doing this, the algorithm can ensure that the population is improving.

2.1.2 Deep Learning and Deep Neural Networks

This section contains a brief overview of the theory needed to understand the phenotype of the proposed evolutionary algorithm for neural architecture search. The phenotype is a neural network. Deep learning [17] in the field of machine

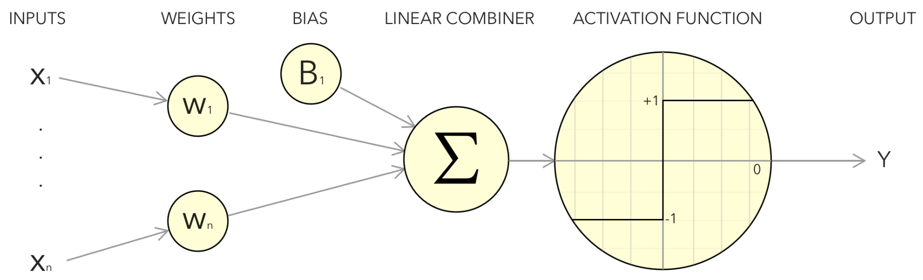


Figure 2.1: The perceptron model using the step activation function. $Y = f(\sum(x \cdot w) + b)$ The formula is another way to denote this model, where the function f is the activation function, in this case step.

learning is based on deep neural networks. Neural networks, also referred to as multilayer perceptrons are statistical models. The goal of a neural network is to approximate some function $f*$. For a classifier, $y = f * (x)$ means mapping the input x to a class y. The $f*$ function described here only contains a single function. Deep neural networks contains multiple functions chained up. The order these functions are chained describes the topology of the network. The name ”Deep learning” stems from having many of these functions chained. The number of functions chained describes the depth of a neural network.

Neural networks are loosely based on the biological neural networks found in brains. A neural network consists of neurons or nodes connected together in an acyclic directed manner. The neurons are also often grouped together in ”layers”. Before going into layers, lets start by understanding the perceptron model.

The neuron, also referred to as the perceptron, visualized in figure 2.1. The perceptron takes some inputs x_i that are weighted by weight w_i into the linear combiner. The inputs and weights are vectors, so the process of weighting the input is vector multiplication. The linear combiner then sums up the weighted input and adds in the bias. The activation function adjusts the values according to the function used before delivering the output.

Activation Functions

There are many activation functions to choose from. The most commonly used are Sigmoid, Tanh and ReLU. These three activation functions are plotted in figure 2.2. Sigmoid and tanh are both non-linear functions which functions very similarly. They can be seen as a non-linear version of the step function. Both of

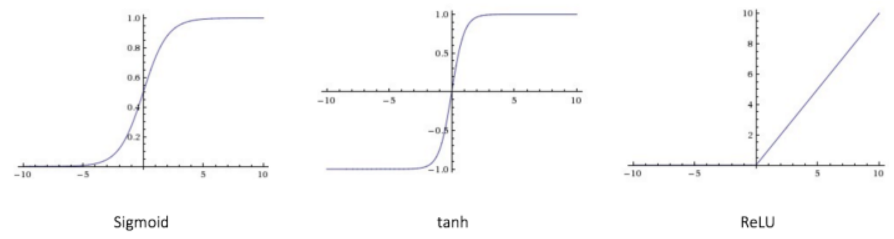


Figure 2.2: Plotted functions Sigmod, Tanh and ReLU.

these have the problem of vanishing gradients which occurs when the input value to the activation function outside the $[-6, 6]$ interval for sigmoid and outside the $[-3, 3]$ interval for tanh. Tanh has much steeper gradients than sigmoid, and the usage of these are determined by how steep gradients is needed.

ReLU is also a non-linear function. It has the advantage of giving sparsity to the output due to its horizontal line. Sparsity is desired in a deep neural network due to a speedup when not all neurons are activating. Sigmoid and tanh both activates in an analog way, meaning almost all input gives an output value. For ReLU, only inputs above 0 will activate giving less values to compute. The ReLU function in it self is also less compute intensive to calculate.

ReLU is not without faults. The Dying ReLU problem is a problem that occurs when the activations of a neuron is 0. The neuron weights will never update because the gradient calculated from 0 is 0. This occurs for all negative inputs to the ReLU function. Variations of ReLU combat this problem, like Leaky ReLU.

Bias

Bias is a very important feature of a neural network which allows for using 0 value inputs to the perceptron model. The bias is learnable which is critical to the success of a neural network as it helps the network converge.

Layers

A layer in a neural network consists of many perceptrons grouped together. Each perceptron is a model like described above with its own activation function and weighted inputs. In a layer topology, the inputs to each individual perceptron is the outputs of the entire previous layer. The naming convention for the different layers are inputs, describing the data vector, hidden layers and output layers giving the output features.

Loss

The loss function - also referred to as the cost function - is used to calculate the error of predictions made by the neural network. The loss function used for the system proposed in this thesis and for most classifier tasks is maximum likelihood. This function is described as the cross-entropy between the training data and the model distribution. The function is given by:

$$\widehat{\ell}\,(\theta\,;x) = \frac{1}{n}\sum_{i=1}^{n} \ln f(x_i \mid \theta),$$

When the maximum likelihood is calculated, the difference between its output and the real features are used as loss for the neural network.

Optimizer

The optimizer's role in a neural network is to update the weight and bias parameters so that the loss function is minimized. To picture how this works, imagine a terrain where you want to find the lowest point. The optimizer acts as a guide that points to lower terrain. Note lower not lowest. Gradient decent is the most commonly used optimizer. It calculates the partial derivatives that points towards the steepest hill in the terrain.

Since gradient decent always points down hill, local minimas and plateaus are problem areas. Local minimas can trick gradient decent away from the global minima. plateaus are also difficult as these areas are flat, making the calculation of the gradient flat. Momentum is a way to avoid such problem areas, where the momentum found from previous gradients are used to decide how far to move. Momentum can often push through local minimas and plateaus.

Training Neural Networks

When training neural networks, data is fed through the input layer and predictions are gotten from the output layer. The loss of the predictions is calculated using the truth of the prediction as described in 2.1.2. The loss the propagated backwards through the network in a process called backpropagation, where the optimizer updates each of the weights and biases. Thus, the network trains.

When the network has trained on the entire training set a single time, an epoch as been completed. A neural network needs to train on the training set for many epochs for the training to converge on to a good set of weights. How the data is fed to the network is also important. Using a single sample of data, calculating the loss and updating the weights is not the most efficient or effective way to train a neural network. Mini-batches are batches of multiple samples of data being fed through the network. For each mini-batch, the loss is only calculated once, using the average error of predictions on the mini-batch of data. This also means the weights are updated once per mini-batch. This both helps the neural network converge smoother on a good set of weights and is faster to calculate. The ensure the training doesn't overfit the network on single mini-batches, the data is shuffled for each epoch of training.

Dividing Up The Data

There are three common categories of data used with neural networks, training set, validation set and test set. These are often split from the same pool of data which contains both input and the corresponding output. This is at least true for supervised learning.

- **Training Set**: The training set is used during the training procedure. This dataset is the largest of the three categories.
- **Validation Set**: Validating progress in the training procedure is important to discover errors early. The validation set is used to validate the training progress. This data remains unseen to the neural neural network meaning it's not trained on using backpropogation. If no validation is needed for the training procedure, this set can be skipped giving more features to the training set.
- **Test Set**: The test set is a portion of the dataset used for testing the prediction of the neural network. This data is used after training is completed and will tell whether the neural network has managed to generalize the knowledge gained from the training step.

Regularizers

A common problem when training neural networks is overfitting. Overfitting occurs when the neural network memorizes the training set. There are very common clues to see whether the neural network is overfitted, mainly looking at the difference in training set prediction accuracy and validation set prediction accuracy. If the accuracy is very high on the training set and the validation set accuracy is low, the neural network is overfitted. Regularizers exist to combat overfitting. There are two regularizers used in this thesis:

- **Dropout** is a simple method that can be used to avoid overfitting. Dropout turns on/off neurons in a layer by a probability p meaning the output value of the neuron is set to 0. The probability for this to happen is often user defined.

- **Batch normalization** was developed to handle internal covariate shift. Internal covariate shift describes the change in the distribution of network activations due to the change in network parameters during training [3]. Batch normalization layers handles this problem by shifting its to zero mean and unit variance for every mini-batch. This results in a normalized input.

2.1.3 Convolutional Neural Networks

Only feed forward neural networks has been described up to this point. The focus will now shift over to convolutional layers. For describing convolutional layers, the basic convolution theory needs to be described. Convolution is commonly found in image manipulation techniques such as filtering, smoothing, edge detection and more and is useful for finding features in an image.

Convolution uses a filter to calculate a new value out of a matrix. This process can be seen in figure 2.3. A source index is used as the center where the convolutional filter is applied. The filter is then matrix multiplied with the source matrix that is covered by the filter. For a 3x3 filter, the source index and all the surrounding indexes will be filtered over. The output of this is a single convoluted output value.

A convolutional layer works exactly like this. A layer contains multiple filters, where the value of the filter is learned and can be seen as the weights described for the perceptron. The convolutional layer also has activation function and biases.

Pooling layers are also an important part of convolutional neural networks. Pooling layers also use filters and common operators are max/min pooling, taking the maximum or minimum value from the source image overlapping the filter of the source matrix accordingly. Average pooling works the same way and takes the average of all values from the source image overlapping the filter. Pooling layers are not trained and does not have activation functions. The effect of using pooling

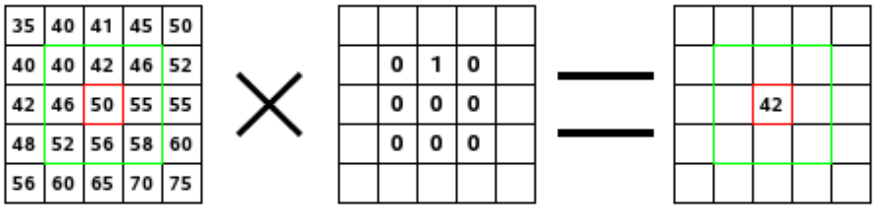


Figure 2.3: Image of the convolution operation [2]. The source matrix is to the left and is multiplied by the convolutional filter giving the output value to the right. This 2 dimensional convolutional filter will shift all values of the matrix down by one.

is that the representation becomes approximately invariant to small translations of the input.

2.1.4 Data Augmentation

With data augmentation, the training set is expanded using the existing data. This is one way to enhance a dataset. Not all datasets allow this, but for image classifiers, augmentations are quite common. Augmenting images would mean for example flipping the images 180, rotating the images just a bit, filtering the images with contrast-, saturation-, smoothing-, sharpening and brightness filters. Doing this would expand the training dataset exponentially with each filter applied. This may allow the neural network to easier generalize due to more data.

2.2 Structured Literature Review Method

The snowballing technique described in [38] was used to gather the literature. The start set contained articles handed out by my supervisors on evolutionary algorithms combined with deep learning for various purposes. The corpus was gathered using backwards snowballing from the start set combined with a few key word searches on specific subjects through the Google Scholar service. The articles was connected using forwards snowballing, finding what articles referenced other articles in the corpus. There was a huge overlap due to the narrow research field of neural architecture search.

2.3 Literature Review

Neuroevolution both searches for the optimal weights of a neural network and it's optimal structure. Neural Architecture Search started as a bi-product of neuroevolution. Finding architectures for neural networks is both an expert task and not an exact science [18]. Some of the best human crafted neural network architectures have already been outperformed by architectures crafted by algorithms. These architectures require immense compute power to make and may use up to 48,000 GPU hours (roughly 2000 days on a single Nvidia Telsa P100) [39]. This massive compute usage is due to retraining neural networks hundreds of times. There is room for improvement. A recent publication from MIT has reached human expert level performance in just 200 GPU hours [6].

2.3.1 Neuroevolution, The Inspiration For NAS

Neuroevolution is a viable alternative to stochastic gradient decent for training neural networks [34]. When training, Neuroevolution comes in many different variants, and classic neuroevolution uses an evolutionary algorithm to evolve the weights of the neural network as well as the architecture. This is especially promising since the parallelization potential of an evolutionary algorithm is much greater than for parallelizing stochastic gradient decent. The method has proven especially effective on reinforcement learning tasks.

For neural architecture search, the inspiration comes in through the life cycle of a synapse (weight connection between two nodes in the network architecture). A synapse lives as long as its it has activations, and if it activates too seldom its removed. New synapses and neurons are added through evolution.

Using neuroevolution in tandem with deep learning is rapidly gaining popularity as a field of research. This is especially apparent within the sub field of neural architecture search where evolution and stochastic gradient decent are both used to evolve the neural networks. The evolutionary component is often the architect of the neural networks while stochastic gradient decent trains them [26; 32; 24; 31]. Neural Architecture search using evolution can be seen as a hybrid between neuroevolution and deep learning [34]. The following section will describe these hybrid methods.

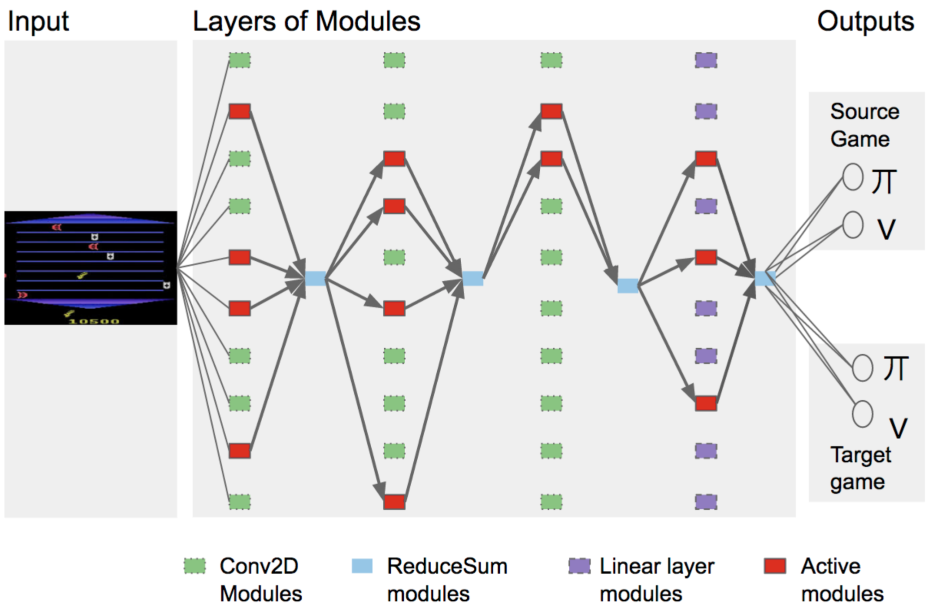


Figure 2.4: A pathway through PathNet [15]

2.3.2 Evolution Based Approaches

Evolutionary algorithms has been proved viable for finding neural architectures [26; 24; 32; 15]. Creative genotype representations described in section 2.1.1 are one of the benefits of using evolutionary algorithms for NAS. [24] uses a hierarchy of neural network operation to build their neural networks.

[15] uses a deep neural network consisting of many deep neural networks and evolves paths through it, making multitask learning possible. Here, evolution is used to evolve what path is taken through the network for each learning task. The network as a whole is trained on a multitude of tasks asynchronously, sharing the knowledge across different learning tasks when possible.

In this study, the evolutionary agents representing the pathways through the network can be chosen using a multitude of different approaches, for example a complex reinforcement learning algorithm. Their genetic algorithm generates the paths used for a specific learning task and is evaluated through training a few timesteps on the selected path using reinforcement learning. The results of these timesteps of learning is used as the fitness function for the given path. Learning

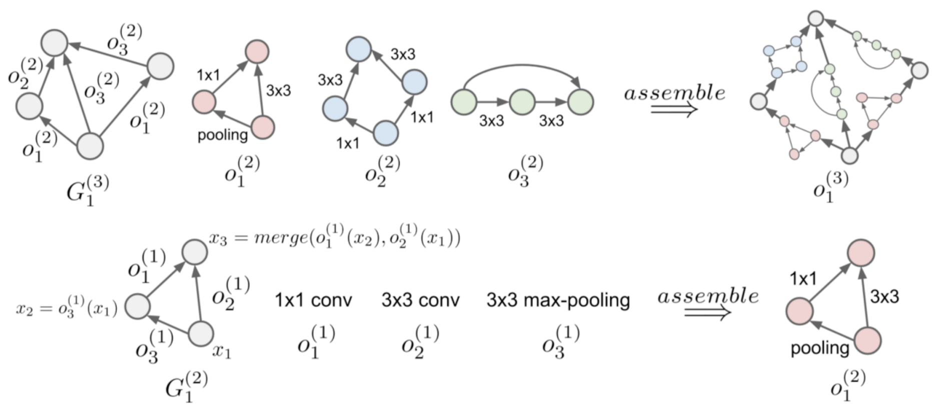


Figure 2.5: Networks assembled using hierarchies [24]

occurs in conjunction with evolution. The giant neural network consists of a set of layers, each containing a set of modules. Each module in this case is its own neural network. A path is selected by choosing one module per layer through to the end of the network. This is illustrated in figure 2.4.

An hierarchical representation was studied in [24]. The representation used is based on flat sets of primitive convolutional neural network operations as shown in figure 2.5. Each of these operations are connected together in an acyclic graph. Each of the primitive operations are assembled in a hierarchy consisting of l levels to create a bigger convolutional neural network. The hierarchical representation evolutionary algorithm uses only a single mutation operator with many effects. The mutation operator is based on replacing everything between two $selected_x$ and $selected_y$ nodes with a single new node. The replacement node $replacement$ is constrained have $l \geq 2$ levels of nodes in the hierarchy. The side-effects of this mutation operator is:

- May add new edge to the directed graph if only one of $selected_x = none$ and $selected_y \neq none$.
- May alter an existing edge if $selected_x = selected_y$.
- May remove an existing edge $selected_x \neq none$ and $selected_y = none$.

Training the models required a distributed system as the workload is extreme. The asynchronous implementation contains a receiver that takes the genotype, converts it into a neural network and then trains and evaluates it. The results are returned to the main process. This distributed system used 450 GPUs to achieve the state-of-the-art results.

CoDeepNEAT [26] uses evolution to iterativly grow a graph based genotype. The graph represents individual layers in the neural network. Here, the evolutionary algorithm starts with the least complex graph possible. For each mutation, an egde is added to the graph. CoDeepNeat uses the traditional genetic algorithm which is in the familiy of evolutionary algorithms. The representation of a genetic algorithm is unique, and is always a bit-string. Each bit-string represents a neural network layer where the bits describe both the type of neural network operator, the hyperparameters it will use and other properties such as number of neurons in a given layer.

Due to repeated patterns being used in some of the most successful neural networks, CoDeepNeat uses "blueprints" which is pre-built patterns of neural network operations to build larger neural networks. During fitness evaluation, the blueprints are combined into larger neural networks and then trained for a while.

Neural architecture search using evolutionary algorihtms manages to discover architectures that beat state-of-the-art human designed architectures [24]. The natural parallelization of the algorithms allows for massive distributed jobs utilizing cloud compute for scaling these workloads. Unfortunatly, compute hours for these algorithms makes these techniques unavailable for the majority of usecases.

2.3.3 Reinforcement Learning Based Approaches

Reinforcement learning for Neural architecture search is a form of meta-learning. Here a learning algorithm will learn how to structure a neural network such that it performs well on the designated learning task. This section will go through some reinforcement learning techniques which has seen success in neural architecture search.

Hidden Markov Models are considered one of the most basic techniques one can use for reinforcement learning. [5] uses a Markov decision based Q-learning model to build sequential neural networks with the hypothesis that what works for one dataset will work for another. The policy of the Q-learning model is created using experience replay, where exploration and exploitation is done in two different steps. First step, exploration creates neural network architectures and evaluates them. This particular implementation uses the validation accuracy as a reward. Both the architectures and the validation accuracies are stored in the experience replay module. In the exploitation step, the Q-learning model

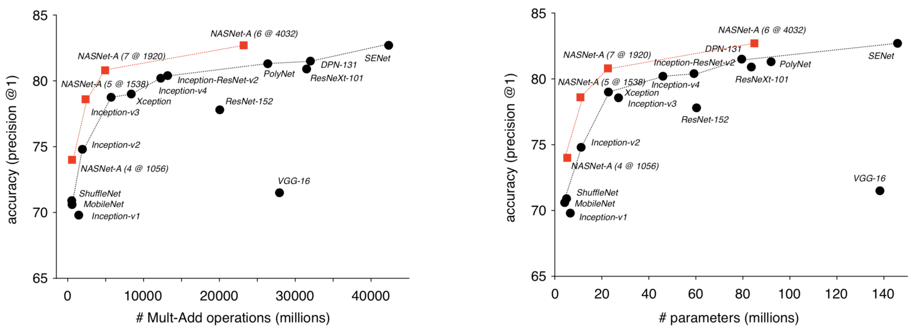


Figure 2.6: NASNet performance compared to multiple human designed state-of-the-art neural networks.

learns from the networks created. These steps are repeated multiple times to create a good model. The neural networks will be created at random for the first few steps, but after a while the Q-learning model will start to make better decisions.

Recurrent neural networks (RNN) have also seen a trend recently [30; 39]. In the study by [39], a RNN designs a neural network based image classifier known as NASNet. NASNet achieves lower error rates on both the ImageNet[14] and cifar-10[20] datasets. NASNet also outperforms top human expert designed neural networks such as ResNet [19], MobileNet v2 [33], inception v2/v3 [35] and VGG-16 [25], see figure 2.6.

To build the successful NASNet, a RNN controller was used as the architect. The algorithm works by letting the RNN design networks by selecting what neural network operation should be used in what order. The action space that the controller uses is very restricted, containing only a few neural network operations. The designed networks are trained until convergence and evaluated. The performance achieved in the evaluation step is then used as a policy gradient to train the RNN on. The RNN will then improve over time, creating better architectures for the given learning task. NASNet is included as a template architecture in some of the most used machine learning tools including Keras [7] and TensorFlow [4]. The architecture can be trained for other image classification tasks than it was originally intended for.

Search Space

Running any of the state-of-the-art algorithms found in sections 2.3.2 and 2.3.3 requires immense compute power. [30] proposes the algorithm Efficient Neural Architecture Search (ENAS) that reduces training time by letting all produced architectures share their weights. The architectures discovered by ENAS have comparable performance to NASnet [39] while claiming 1000x less expensive search.

ENAS lets all generated architectures share their weights. It does this by first producing a directed acyclic-graph (DAG) of computations. All neural networks designed by ENAS will be sub-graphs of the aforementioned DAG. The process of selecting what operations the architecture should consist of uses a recurrent neural network. The RNN selects nodes from the DAG that will be made into a neural network consisting of the computations selected in order. The RNN can only select successors of the previously selected node.

Each node in the DAG has its own parameters that are trained each time its included in a selected sub-graph. This way, all shared parameters are reused, creating a good base that can be used for the next architecture sub-graph selected by the RNN. The training of the DAG parameters happens for each created sub-graph.

The use of a DAG also vastly reduces the search space. Not only does this restrict what types of operations that can be explored but also in what order. ENAS allows for manually controlling how many operations each sub-graph should consist of, allowing them to manually tune the search space.

The action space for a Q-learning model needs to be limited for it to work. In [5] the action space is set to only contain convolutional-, fully connected-, pooling- and softmax layers. [30] uses an action space of only four convolutional operations and two pooling operations giving an action space of 6 possible operations per layer. The search space also consists of what activation function each layer should use.

Search space is also a consideration for the evolutionary algorithm based approaches. [24] has an action space consisting of 5 predefined compute operations. The search space for [15] is different, the giant neural network is predefined, only a select number of paths can be chosen by the evoluionary algorithm. This gives a much smaller search space.

2.3.4 Other Approaches

The architectures generated by NAS algorithms are generally not focused on reducing the parameters of the neural networks. [6] focuses on both finding an optimal architecture and tailoring the network to the hardware its supposed to run on. The algorithm has been named ProxylessNAS.

ProxylessNAS starts of with an oversized network and starts pruning away excess parameters. To do this, they have implemented binary gates which turns on/off parameters within each network operation. These are called the binary architecture weights. The training of the neural network being optimized happens in two stages.

Stage 1: The network weights is trained using stochastic gradient decent. The binary architecture weights are frozen during this stage.

Stage 2: The binary architecture weights are trained. The network weights are frozen during this stage.

After both these stages are complete, the parameters belonging to the binary architecture weights that are off will be pruned away. ProxylessNAS achieves comparable results to all aforementioned NAS algorithms while containing much fewer parameters. The only exception to this is ENAS, which also has small architectures.

Chapter 3

Architecture

*The following chapter will explain the architecture behind the experiments. There are three different main experiments, **EA-NAS**, **Local-NAS** and **Pattern-NAS**. EA-NAS and Local-NAS uses two different algorithms, evolutionary algorithms for EA-NAS and hill climbing search for Local-NAS. They share the same genotype representation, directed acyclic graphs (DAG) representing a whole Neural network. Pattern-NAS is also based on evolutionary algorithms but with a different representation based on sub-graphs combined into whole DAGs which represent neural networks. The chapter starts off with the training loop which is common for all of the architectures, before diving into each experiment's architecture in detail following the UML diagrams, figures 3.1, 3.3, 3.4 and 3.5*

3.1 Training Loop

The training loop is used by all of the experiments. This program is always runs asynchronously on a subprocess either locally or in a distributed system. The training loop converts the genotypes into phenotypes – or Keras models, then trains and evaluates them. It also stores all phenotype models trained for later use.

3.1.1 Choice of Machine Learning Framework

The training loop uses Keras [7] as it's machine learning framework with TensorFlow [4] as the Keras backend. The machine learning models and training

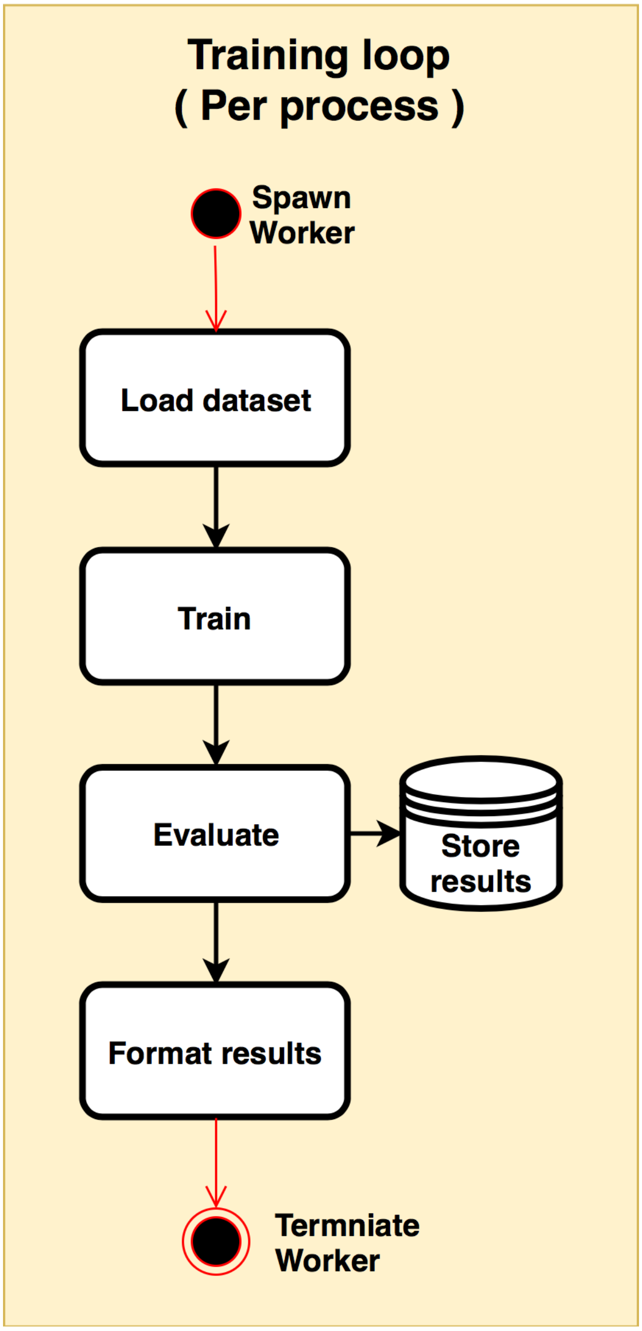


Figure 3.1: UML diagram of the training loop.

loops are defined in the Keras framework but is converted behind the scenes to run in Tensorflow. TensorFlow supports both CPU- and GPU based training of machine learning models. The CPU training is written in C++ with either a OpenMP based parallelization technique or MPI based for distributed systems. Tensorflow's GPU support is restricted to the Nvidia Cuda framework. CUDA is proprietary software that has limited support for common OSes like macOS. Cuda only support Nvidia's own GPUs, leaving AMD and Intel based parallel hardware as a non-option. This also makes TensorFlow support unavailable on macOS.

The Keras framework supports multiple backends, meaning the models created in keras can be converted to any of the supported machine learning frameworks. PlaidML [Ng] is one of the supported frameworks. It's open-source and aims at supporting all major parallel computing solutions available on the market meaning this system may run on hardware supported in PlaidML.

3.1.2 Spawn Worker

Parallelization is one of the key advantages of evolutionary algorithms. As such, multiple parallelization techniques have been tested out for this thesis. This had to be done as the servers available during the writing period had different supported methods for parallelizaiton.

Using SSH For Distributed Processing With The Execnet Framework

Secure Shell (SSH) is a common method for connecting to another computer on the network through a terminal interface. Through the execnet package, setting up a connection between two machines on the same network for distributed processing was simple. This framework allows the algorithm to run on multiple machines without them being in the same cluster. This was the first configuration used for testing this system.

File synchronization was a requirement for transfer learning. All of the files created asynchronously on each server had to be synced back to the server running the main process. This was a time-consuming task as machine learning models may take up many gigabytes of storage. To minimize the transfer time overhead, a system for ensuring minimal transferring of data was implemented. This was meant to ensure that all machine learning models were trained on same the machine they had previously been trained on meaning transfer was unnecessary. This system was later scrapped as an MPI system could be used with a cluster.

Handling TensorFlow GPU Memory Allocation

TensorFlow's implementation of GPU memory allocation does not allow for de-allocation during a process lifetime. Forced de-allocation of memory will lead to unwanted exception being raised during runtime. After allocating memory for the training session on the GPU, memory was nearly full. When the second training session was allocated on the GPU, the program would fail or run very slow due to memory being swapped from the servers main memory and the dedicated GPU memory. To work around this, a child process has to be spawned for the training loop.

Using a child process raises serialization concerns. Neither TensorFlow nor Keras supports serializing the machine learning models. A work around for this is to save the Keras models to disk and keeping the absolute path available in memory for later use.

Using Message Passing Interface With Python

The original implementation of Message Passing Interface (MPI) was intended to use with spawning all of the worker processes with the main process to save time on the overhead of spawning a new process. As seen in the previous section 3.1.2, each training session process needs to be killed to free GPU memory. The original implementation is not a good fit for this task.

Newer versions of MPI allows for spawning processes dynamically, while still being able to use the same communication as with the original implementation. Further, the package MPI4PY [10; 11; 9] contains functional programming concepts like "map" for easily distributing the workload over multiple child processes. Serialization is also automatic using the pickle package for serializing Python classes.

3.1.3 Converting Genotype To Phenotype

Converting the Genotype over to a Phenotype is done using a recursive loop and Keras models. This process starts by first creating the Input Keras layer and then locating the input nodes in the DAG. A recursive process is then started which goes through the DAG, node for node in a queue-based fashion. Each node seen in this process, is converted into a corresponding Keras layer and added to the Keras model. There are some special cases that needs to be handled:

- For ends of branching, a concatenation layer are added concatenating all of the nodes previous to the current node.

- A Dense node following a Conv2D or Pooling node needs to have a Keras flattening layer between it self and the 2-dimensional layers.

Finally, the outputs nodes are discovered, converted and concatenated. A Dense output layer using the softmax activation function is added to complete the classifier. A complete Keras model has been created.

3.1.4 Transferring Knowledge

Transfer learning occurs after the phenotype Keras model has been generated from the genotype. The predecessor of the current phenotype has to be written to disk in the previous generation for transfer learning to activate. The predecessor phenotype is loaded into memory. Transfer learning goes through every layer in the newly generated Keras model and finds the matching layer from the predecessor Keras model. The matching is done by using custom IDs which are applied during phenotype to genotype conversion.

3.1.5 Load Dataset

As this system needs to be able to use multiple datasets, a generalized way of importing data had to be made. Using the built-in python module ”importlib”, python functions can be loaded through an absolute path to the ”.py” file containing the functions. There are three functions needed to import data to the training algorithm:

1 **get_training_data():** Gets the entire training dataset. The training data is used both with training the models and with calculating how ”overfitted” the models are.

2 **get_validation_data():** Gets the entire validation dataset used for calculating the validation accuracy. The validation accuracy is needed to test if a neural network has converged. When not training until convergence, this data can be used as training data instead as its not used for anything else.

3 **get_test_data():** Gets the test data. This data is used in the evaluation of the model. Accuracy scores on this data is the main way to identify good models over bad models.

Importing the data in this way also allows for experiments with transfer learning to learn from multiple datasets.

3.1.6 Train

There are two different training methods that can be chosen between, "training" and "training until convergence". "Training" trains the neural network for a fixed number of epochs while "training until convergence" uses a condition for when to quit training. Otherwise, the training loops are the same. The default parameters for the training are listed below. All of those parameters may be user defined and applies to both methods.

Optimizer	:	Adaptive Momentum (ADAM) with a learning rate of 0.001
Loss	:	Categorical cross entropy
Dataset order	:	Shuffled every epoch
Batch size	:	60

Convergence Condition

The training method "train until convergence" runs for a set number of epochs. After this loop has completed, the convergence condition will be checked. The convergence condition tests whether the training has made any improvement for this round of training and whether the training has made any improvement overall. The former uses the 10 last training epochs as a sample and averages them. This gives the acc_{avg}.

$$\text{converged} = (acc_{avg} \cdot 0.985) \leq acc_i \leq (acc_{avg} \cdot 1.015)$$

This condition is checked for both the validation accuracy and the training accuracy. Both of the training and validation expressions has to be true for the condition to be met.

The ladder checks if training has made any improvement overall since it started. This condition samples the accuracy achieved 30% into the total training epochs, yielding acc_{early}. This sample is used in a similar fashion as with the *converged* condition:

$$\text{improved} = (acc_{early} * 0.90) \leq acc_i \leq (acc_{early} * 1.10)$$

The training will go on until either one or both of the conditions *converged* or *improved* are true.

3.1.7 Evaluate

The evaluation lets each neural network predict all of the samples found in the test dataset. These predictions are used to calculate the confusion matrix which all performance is measured from. From the confusion matrix, a classification report is created with the following metrics:

Precision	The total number of correct predictions out of all predictions.
Recall	The number of correct predictions in a certain category.
F1-score	Uses both prediction (p) and recall (r) to create a mean between them: $2 \cdot \frac{p \cdot r}{p+r}$

3.1.8 Format And Store Results

The results found in the evaluation step is stored into each genotype as fitness. For EA-NAS and Local-NAS, this means just storing the value directly into the genotype. For Pattern-NAS each pattern receives the score collectively found in the training step. Patterns saves results in a list, where a optimal result is used for comparison. Since each pattern is used multiple times per training loop, there will be multiple results to choose from.

Limitations With Writing To Disk

This step also saves the Keras models to disk. This is a required step for transfer learning to work. Since a Keras model may take up many gigabytes of disk space, storage for a huge simulation may pose a problem. OSErrors have also been encountered when multiple child processes are writing to the same network drive concurrently. This is a known issue with Keras. The saving process may therefore try saving up to 5 times with a 5 second pause between each try. Although the retry process solves this error in most cases, 1 in 500 models are lost.

3.2 EA-NAS

The EA-NAS experiment is modelled closely after a traditional evolutionary algorithm. The initialization step creates random genotypes representing neural networks. These genotypes makes up the initial population. The genotypes are iterativly mutated and crossed over for random changes. Only the best scoring neural networks are kept and iterated on, while the worst ones are removed. Deciding which genotypes to keep or remove is controlled by multi-objective sorting. Iterating will go on until a goal state is found, the last generation is reached or the algorithm runs out of time.

3.2.1 Genotype Representation

The representation of genotypes in EA-NAS is a directed acyclic graph. The graph can contain either sub-graphs consisting of whole genotypes or individual operations. These DAGs are supposed to have a tight coupling to Keras while still being an indirect representation. A genotype and its corresponding phenotype gathered from the preliminary results can be seen in figure 3.2.

Operations/Nodes

The individual operations are nodes containing the properties of the neural network operations they represent. The following operations are used in the genotype:

- **Fully connected (Dense)**: The 1-dimensional Dense nodes comes in three different sizes, Small with 250 neurons, Medium with 750 neurons and Large with 1500 neurons. The activation function of these are always rectified linear units (Relu) as they have proven to be more efficient when used with the backpropogation than other activation functions.
- **Convolution (Conv2D)** Convolutional nodes represents 2 dimentional convolutional layers. They come in two different filter sizes, either 3x3 or 5x5. They have 50 filters and ignores edges. Ignoring the edges was a simple way to enable branching without errors. As with Dense nodes, Convolutional nodes also use Relu.
- **Pooling** Pooling layers comes in two types, Maximum Pooling and Average Pooling.

With this list, the total search space is 7. Each of these nodes have a direct counterpart in the Keras model phenotype. The sizing of all the operations are fixed to reduce the search space. The search space grows exponentially with each type of Node.

Regularizers are also a very important part of a neural network as they help keep the network from overfitting. Both the Dense and Conv2D nodes have a 80%

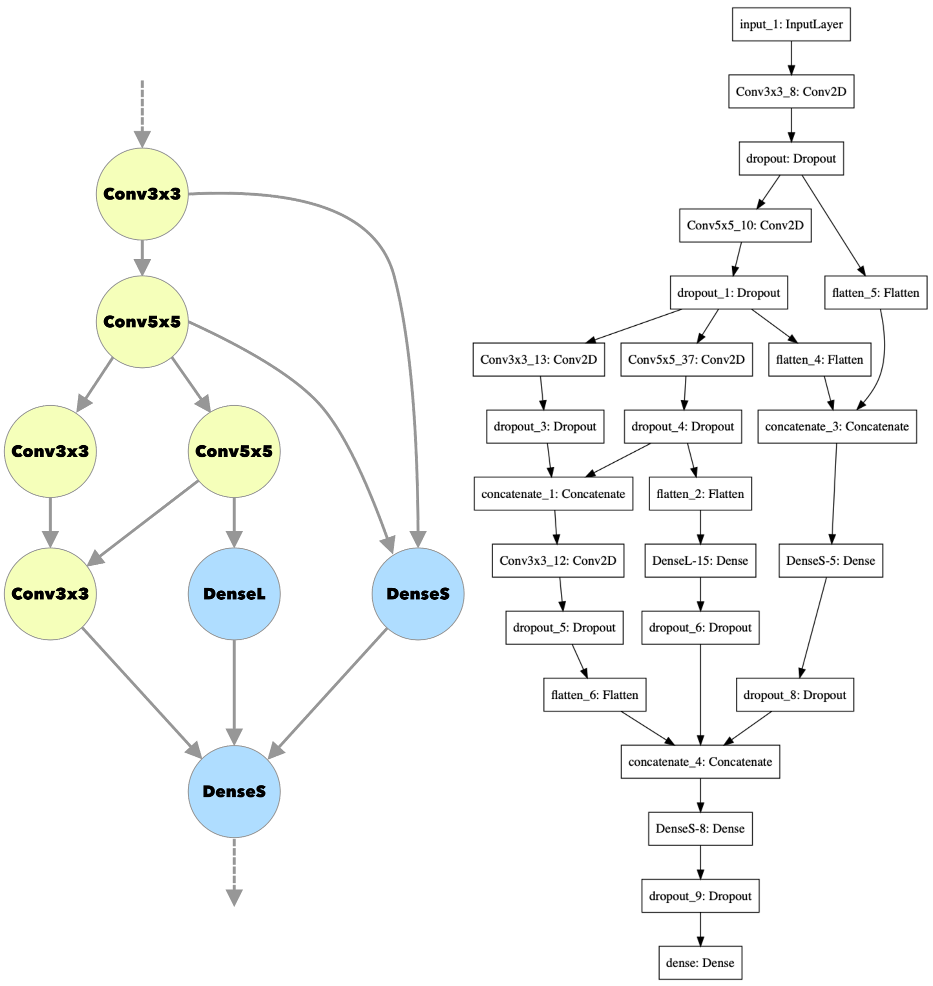


Figure 3.2: The model to the left is the genotype directional-acyclic graph while the model to the right is the Keras Model. The Keras model is generated using the built in tool for visualizing network models. This example also have 100% regualizers used. The regularizers here are dropout only. Convolutional layers may also use batch normalization.

chance of having a regularizer connected to their outputs. This is chosen during the initialization of each individual node. The default setting for Dense layers are Dropout, while convolutional layers have batch normalization as default but may also use dropout.

Genotypes/Directed-Acyclic Graph

The full DAGs contains at minimum 4 operations or sub-graphs. When building such a graph, the first 4 nodes added into the DAG must be added with the appending operator (see section 3.2.3 for appending). After the minimum nodes are appended, the graph can be extended using any of the other mutation operators. There are some constraints on how the operators are added. A Dense node may never come before a Conv2D or Pooling node.

The genotype does not contain nodes for inputs or outputs. All nodes without any previous nodes in the DAG are seen as "input" nodes. Similarly output nodes are all nodes without any next node in the DAG. This genotype allows for branching as is in the nature of DAGs. This branching will be carried over to the phenotype upon conversion.

3.2.2 Initialize

The initialization step creates and trains the initial population. Its important that randomness is maintained in this step to create a diverse population. To spawn a random new individual, the following process is performed:

1 Generate empty genotype. This is an empty directed acyclic graph.

2 Draw the number changes to be applied to the new genotype. This number will be between two user defined boundries.

3 Apply the changes using the mutation operator. The mutation operator is described in the next section **??**.

This process is repeated for each individual, resulting in a very random population. Every operator in the mutation operator can be applied with the same probability as with regular mutations, even the "remove" operator. After the entire population has been generated, the training loop is launched for evaluating the fitness of the new population.

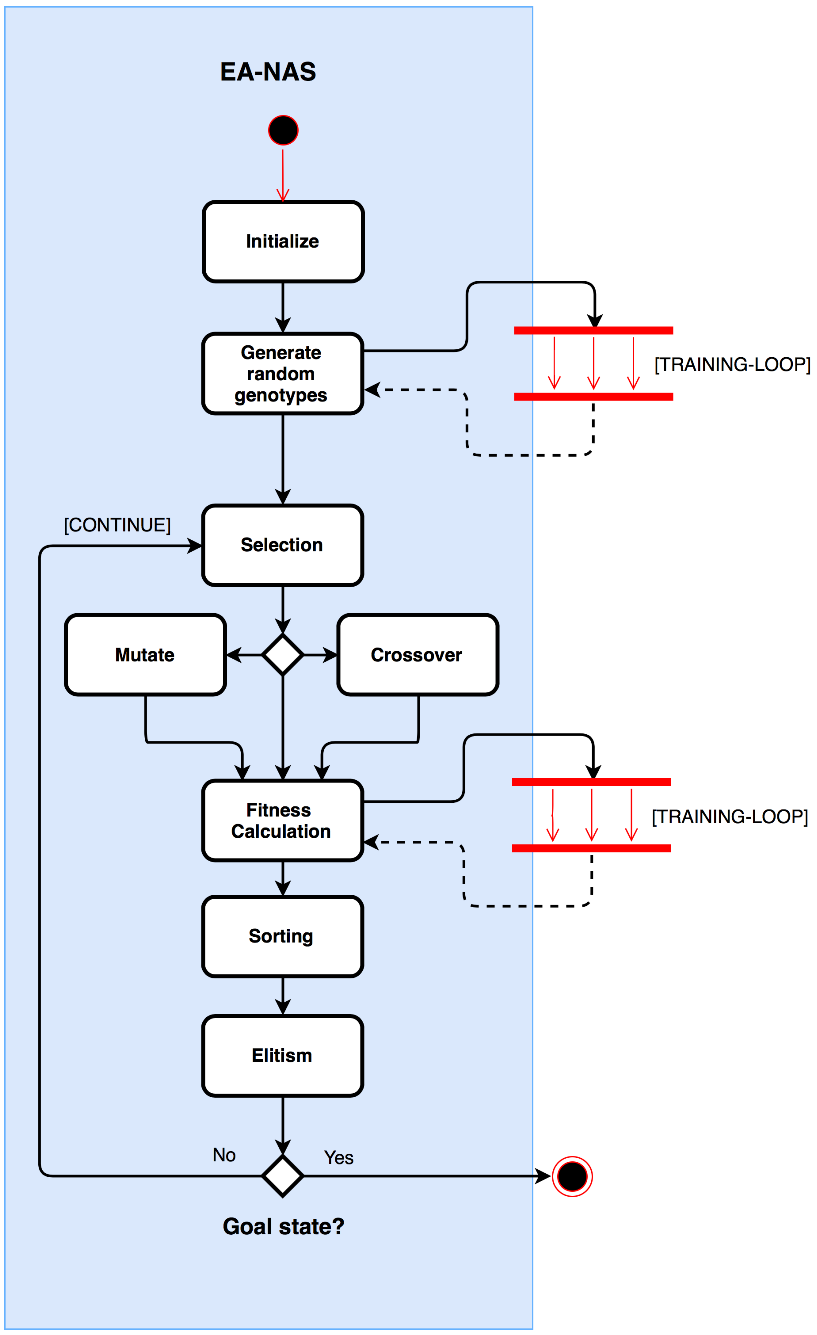


Figure 3.3: UML diagram of the EA-NAS experiment.

3.2.3 Selection, Mutation and Crossover

Selection is used to select the individuals that are to be either mutated or crossed over. The selection will select 50% of the population. The operator used is Tournament selection. Tournament selection is based on having 2 or more individuals of the population compete against each other. The operator will always select the best of the tournament. Which individuals that will be used for tournament is randomly selected. The tournament size for this implementation is 2. This means there is a probability of selecting two "bad" individuals for tournament. This gives the worst performers in the population a chance to improve. The whole previous generation will be trained for some more epochs before elitism. This will ensure that the unselected are not discarded without a second chance at the evaluation step.

The second chance is important as smaller neural networks perform better with fewer training epochs than larger ones does. In the long run however, the smaller neural nets that had the most promising scores might not be large enough achieve the desired accuracy score on the test set. The bias towards smaller neural networks is also good as smaller networks will require less compute power to both train and use for predictions. If a smaller neural network can perform on par with the larger neural networks, the smaller one is preferred.

Mutation

The mutation operator applies a single change to a given genotype. The mutations are mostly removal or insertion of neural network operators. There are some placement constraints to adhere to. A dense layer may not come before a convolutional- or pooling layer as there are no features to extract from a dense layer. The different mutations that can be performed are selected using the probability distribution in listed below with the descriptions:

Append (Probability: 0.0303) Creates a new neural network operator and appends it at the end of the genotype.

Insert (Probability: 0.0303) Inserts a new neural network operator between two nodes in the network. Note that this insertion does not replace the connections between the two nodes, but rather adds a new one. An insertion can happen to any two nodes in the genotype as long as the placement constraints applies.

Insert between (Probability: 0.4545) Similar to the Insert operator, this operator selects two nodes with direct connections to each other and replaces their connection with a new neural network operator between them. Placement constraints applies here too.

Skip-connection (Probability: 0.0303) Creates a connection between two nodes in the genotype. No new network operator is created or insterted for the new connection. This method is based on the "skip-connection" method in ResNet [19].

Remove (Probability: 0.4545) Removes a selected node from the genotype and reconnects each of the nodes connected to the removed node. This is the exact opposite of "insert-between".

To keep everything random, nodes selected for mutations within the genotype is always random. For the insert and insert-between operators, the two nodes used for insertion points are also selected randomly. These also use the probability distribution below for selecting each type of layer:

0.3947	:	Dense layer (of various sizes)
0.2631	:	Convolutional layer (3x3)
0.1842	:	Convolutional layer (5x5)
0.0789	:	Max Pooling layer
0.0789	:	Average Pooling layer

Crossover

The crossover operator combines two genotypes into a new genotype as described in chapter 2.1.1. For this particular implementation, deciding on a crossover operator proved diffcult. One cannot simply join two directed acyclic graphs together as they may have very different structures. There is also transfer learning to consider. If doing a striped crossover, all of the trained weights from the predecessors will be lost. Two solutions to this problem was tested:

Crossover operator 1: Inserting a full genotype into another genotype. This method would allow for the weights to be almost entirely kept the same for both genotypes. For this method however, all possible insertion-positions has to be tested, selecting the best overall performing position. This step was time consuming. There are also constraints that needs considering, these are the same as for the mutation operator. This method proved to perform poorly as the successor of the two genotypes was very large. There also seemed to be little improvement in any of the experiments with this method.

Crossover operator 2: Placing two genotypes "side-by-side". In this crossover operator, both genotypes share the same inputs and are concatenated on their individual outputs using a softmax layer. This means there are no physical connections between the two genotypes. They can do their predictions individually. The softmax layer joining their outputs will create a probability distribution of the outputs for each model. The one with the best guess will hopefully be selected. The best configuration for this crossover operator would be two networks which perform well on different classes in the classifier task. Is has however proven to be a difficult task as most networks performs quite similar on the same classification tasks, with only minor variations as seen in the results 4. This operator also have the same model size problem as with the former crossover operator.

Both of the crossover operators were disabled due to their problems. This leaves the algorithm without any crossover operator. All selected individuals that were bound for crossover are mutated instead.

Initializing New Random Individuals

Each generation, there is also a probability of 0.01 of a new single individual being added into the population. This functionality exists to increase exploration. This is useful in cases where there is no genotype in the population which can learn the dataset properly.

3.2.4 Calculating Fitness

For calculating fitness, the training-loop is launched using MPI. First, all parameters are packed into zipped pickled binary strings. Then, MPI distributes the workload to each server. Once on the servers, each genotype is converted into its corresponding phenotype which is represented by a Keras model. The phenotype of the predecessor is also loaded into memory for transfer learning. The networks are either trained for a fixed number of epochs each or the training is scaled based on size of the network. There is no empirical evidence so far that the scaling makes the results more fair, thus EA-NAS uses the fixed number of epochs favoring the smaller networks.

3.2.5 Sorting and Elitism

There is two sorting functions available to use, the first is based on weighted sum and the second is based on the multi-objective optimization algorithm NSGA-II [13]. These two sorting functions serve the same purpose but have very different goals. For the weighted sum, the best of the population needs to be chosen in terms of raw performance, where each goal is optimized and the strictly best solutions are selected while for the multi-objective optimization algorithm, the goal is not only to select the best solution but also to maintain diversity within the population.

Weighted Sum

For this sorting algorithm, the weighted sum of the overfit penalty and the test accuracy is calculated. The overfit penalty is calculated by:

$$overfit = |acc_{training} - acc_{test}|$$

The weights are set to 30% overfit penalty and 70% test accuracy, giving the formula:

$$score = overfit \cdot 0.30 + (1 - acc_{test}) \cdot 0.70$$

The compliment is taken of the test accuracy. The *score* is sorted ascending. The best individuals have lower scores. When only using the test accuracy as a sorting metric, the best individuals were the most overfitted in the population. This was not desirable as individuals with lower test accuracy but a much more balanced training accuracy may need some more epochs of training time to get better. With overfit penalty included in the weighted sum, differences in overfitting helps the algorithm select more desirable solutions.

Multi-Objective Optimizaiton Using NSGA-II

The multi-objective sorting algorithm NSGA-II [13] is implemented for this neural architecture search algorithm. Multi-objective sort is often used with evolutionary algorithms to maintain diversity in the population. Maintaining the diversity means keeping solutions that may be worse than the others in raw performance, but have some major differences that are worth keeping. For algorithms like Evolutionary algorithms, diversity is key to avoid local maximas and plateaus. This is important to keep the population from becoming entirely the same solution.

NSGA-II sorts by first creating frontiers of solutions, then sorting each frontier using the crowding distance assignment algorithm. Each frontier contains solutions in which the condition "no worse" is held between all solutions. No worse means no one solution is dominated by another by being either equally good or better in all objectives of the multi-objective optimization. The frontiers are ranked with all solutions in $frontier_i$ dominating all solutions in $frontier_{i+1}$. We can then say that a solution in $frontier_i$ are strictly better than solutions $frontier_{i+1}$. Within each $frontier$, all solutions are seen as equal because no solution is worse than another in all objectives.

For every objective that uses floating point numbers, the "no worse" condition can be hard to measure. All floating point objectives is therefore rounded to use at most 3 decimals. This is especially important in the classification accuracy objectives, as two networks where $net_0 = 0.900000000$ and $net_1 = 0.900000001$, should be measured as equals.

In comes crowding distance assignment (CDA) which sorts each $frontier$ based on score in all objectives. CDA tries to prioritize each frontier based on differences between each solution. The most polarizing solutions, i.e. the worst and the best are given max score while everyone in the middle is given the score based on the cuboid difference between the solutions closest to them [13].

$$S_i.distance = S_i.distance + (s_{i+1}.measure - s_{i-1}.measure)$$

Where S is a list of all solutions, *measure* is the solution's score on a given objective and *distance*' is the crowding distance factor. The objectives sorted on can be read about in chapter 4.2.1.

Elitism

Elitism removes the part of the population which is deemed the least likely to succeed. The population size has to be maintained before starting each generation. For weighted sum described above, this means the worst performers in the population are removed. For multi-objective optimization, both the worst and the most similar solutions are removed. The best of the most similar are kept.

3.3 Local-NAS

Local-NAS builds upon EA-NAS. Instead of using an evolutionary algorithm for evolving the best architecture, Local-NAS uses the local search algorithm hill climb. Only the single best individual in the population is kept for the next generation. This causes the most changes in initialization, selection and elitism. The only main difference in initialization is that only the best of the tested initial

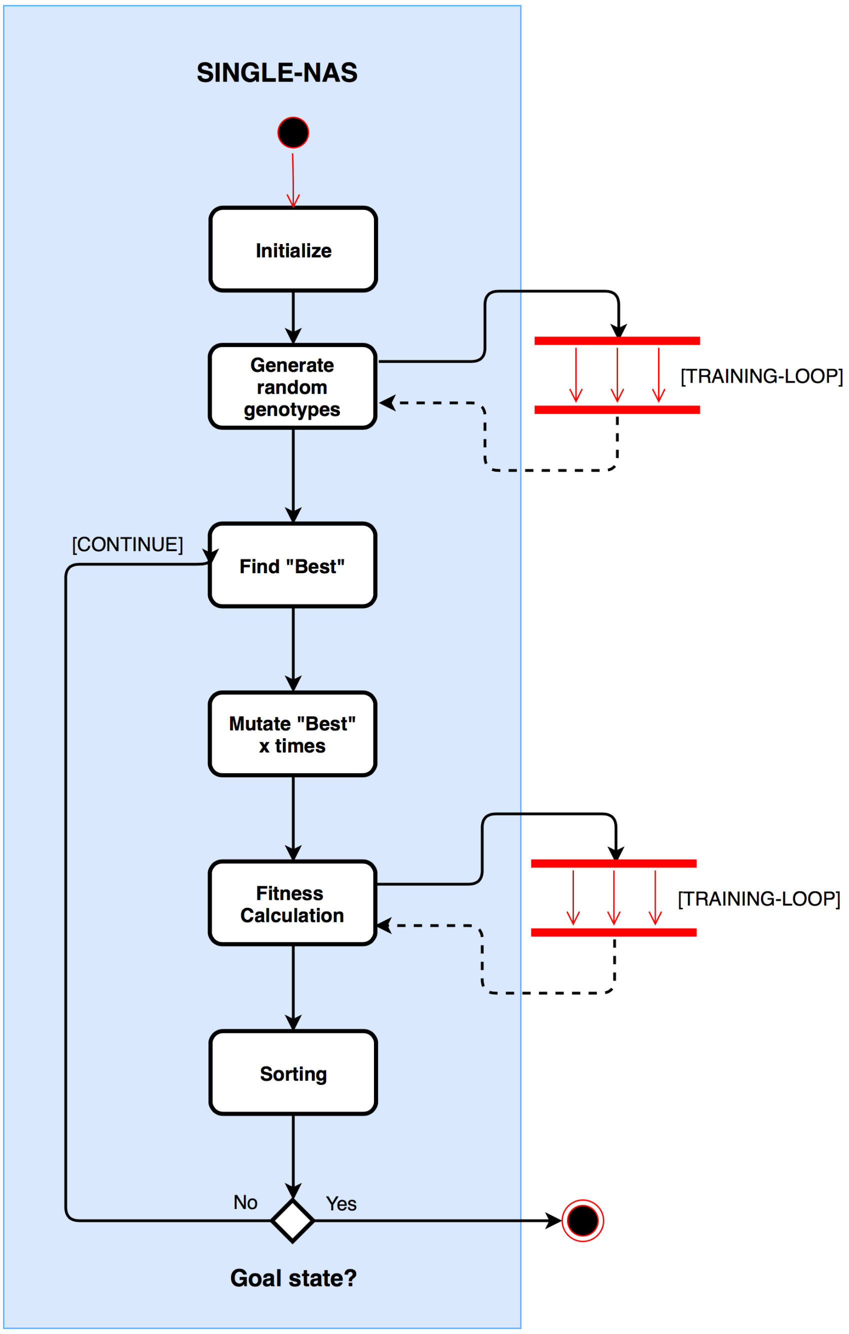


Figure 3.4: UML diagram of the Local-NAS experiment.

population is kept. This means the rest of the algorithm remains the same as EA-NAS. The list contains the steps within Local-NAS that are identical to EA-NAS:

- Representation, section 3.2.1
- Generate random genotypes, section 3.2.2
- Mutation and Crossover, section 3.2.3. Crossover is not used in Local-NAS.
- Calculating fitness, section 3.2.4
- Sorting, section 3.2.5. For Local-NAS, the weighted sum is always used.

3.3.1 Selection

As only the single best individual from the population is kept each generation, the selection for mutation is very different from EA-NAS. Here, the single best is mutated numerous times before each of the new mutated solutions are evaluated for fitness. Both the mutation operator and the evaluation is exactly the same as for EA-NAS.

3.3.2 Elitism

The multi-objective sorting is not usable with Hill-climbing as there is no diversity to maintain. The weighted sum sorting with overfit penalty described in section 3.2.5 is always used here. The single best performer revield from the sort is kept for the next generation.

3.4 Pattern-NAS

Pattern-NAS is vastly different from the two aforementioned experiments found in sections 3.2 and 3.3. This is largely due to the difference in representation. EA-NAS and Local-NAS uses a very direct representation similar to the phenotype as described in section 3.2.1. Pattern-NAS on the other hand uses a meta representation of the EA-NAS/Local-NAS representation. This new representation, described in section 3.4.1 causes many changes to the core evolutionary algorithm.

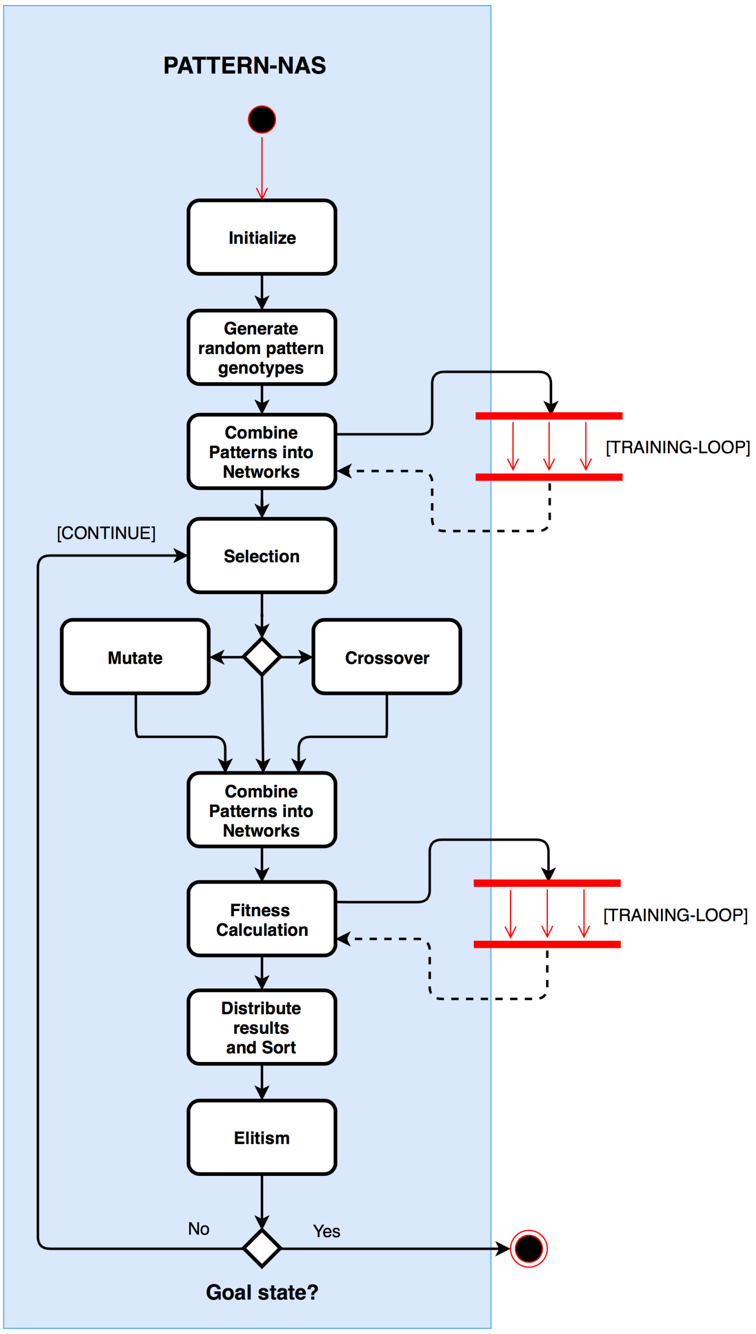


Figure 3.5: UML diagram of the Pattern-NAS experiment.

3.4.1 Representation

The representation of Pattern-NAS consists of many smaller patterns represented by graphs. A pattern is a DAG in the same way that the EA-NAS and Local-NAS representation are DAGs. There is a few key differences:

1 The Pattern DAGs may contain at most 4 operations.

2 All operations within a Pattern DAG have to be either 1-dimensional or 2-Dimensional operations. Cannot consist of both.

3 No sub-graph can be inserted into a Pattern.

4 Patterns may have disconnected nodes as seen in Pattern 2 and 3 of figure 3.6

3.4.2 Initialization

Each pattern is initialized by first selection what proportion of the population will be 2D patterns (i.e. Conv2D or Pooling operations). The rest of the population will become 1D patterns (Dense). Then, the patterns will be randomly assigned operations. The operations are connected together with a random number of connections in the range 0-4. All though these connections are set randomly, too much randomness can cause the graphs to be cyclic. Constraints has been added to prevent this.

3.4.3 Joining Patterns

Although the Pattern representation supports training by it self, any pattern would be too small to learn a complex dataset. The patterns therefore have to be combined into a larger EA-NAS genotype before training commences. There are two methods of joining patterns, one which tries to place each pattern to their optimal position and one based on randomness. Before training, one network is joined using the optimal position technique and the rest is joined using randomness.

For joining optimally, the EA-NAS genotype is build using only the best performing patterns in the population. When the best performing of the population has been selected, each pattern must also know where it's commonly used and where it fits best within a neural network. This is important for the weights not to be too badly fitted.

The placement is measured by a normalized distance metric. This metric is a number between 0 - 1 which describes the distance from the input node to the start of the pattern normalized on the critical path through the neural network. This metric is stored along side the result for each training session. Since each training session will result in its own neural network weights, one can use the distance metric in combination with a pretrained set of weights. This configuration is selected upon the joining optimally phase.

Joining randomly starts of by creating an empty EA-NAS genotype. This genotype is incrementally filled with patterns. A fully joined EA-NAS genotype is defined as one that has either met the user defined limit in size or contains all patterns in the population.

After using any of the two mentioned combination methods, the patterns must be connected together to create the bigger EA-NAS DAG. For joining two patterns, the output nodes of one patterns are directly connected to the input nodes of the next pattern. This creates a sequential connection between two patterns. Since patterns can both have multiple inputs and multiple outputs, the inputs and outputs of the two patterns are joined randomly. The number of connections created will match the larger number of either number of inputs or number of outputs. This process is visualized in figure 3.6.

3.4.4 Mutation

There are four mutation operators for mutating the Pattern-NAS system. As with EA-NAS and Local-NAS, these are primitive operations for a directed acyclic graph.

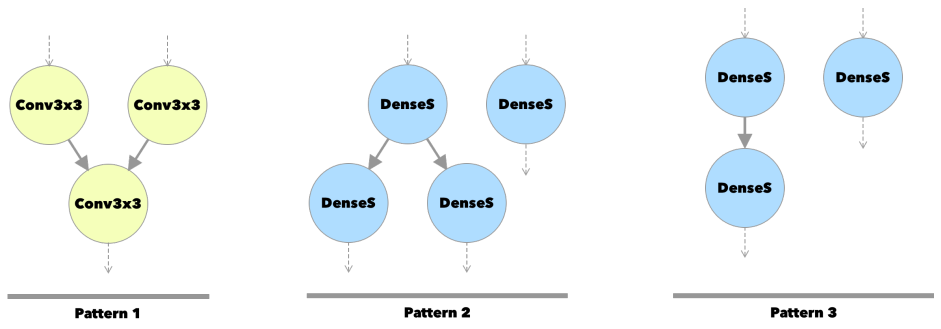
Insert Adds a node to the graph in a random order. A connection may also be set.

Remove Removes a node in the graph and any connections it may include.

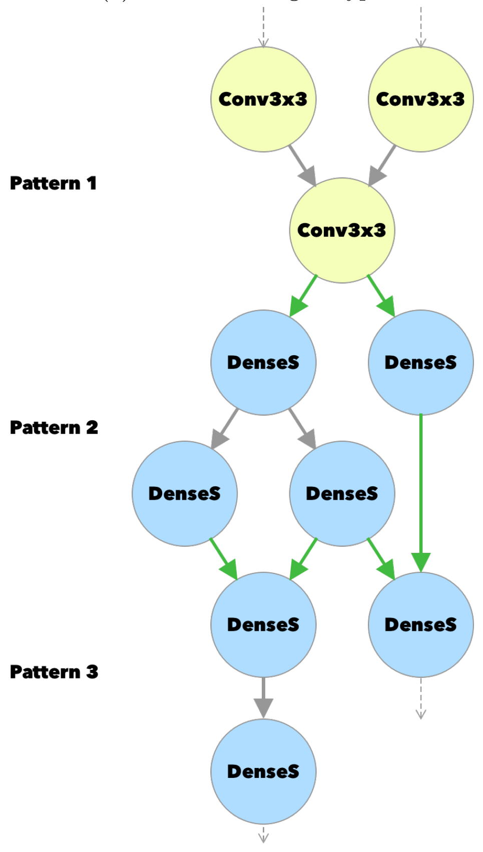
Connect Adds a connection between two nodes, adhering to the positional constraints that keeps the graph acylic.

Swap Replaces one op with a new other op. This removes all weights that could have been transferred with transfer learning.

These mutation operators are only conditionally available. If number of nodes in a pattern is at minimum, the remove operation is disabled. Similarly, if the number of nodes in the pattern is at maximum, the insert is disabled. Maximum



(a) Pattern-NAS genotypes



(b) Patterns joined together into a EA-NAS genotype

Figure 3.6: Pattern combination method. The three patterns shown in *a* are joined together in a sequential manner depicted in *b*. Whole arrows show connections while in-going striped arrows shows input nodes and striped out-going arrows show output nodes. The green arrows in *b* represent the new connection made between the patterns.

number of connections disables the "Connect" operator. Swap is always allowed. Note that there is no "remove connection" operator. This is because "remove" operator will clean up connections when removing the node, making the pattern disconnected. All of the available operators have the same probability of being selected.

3.4.5 Crossover

The Pattern-NAS crossover operator is based on the common "striped" crossover operator described in section 2.1.1. To apply this, one of the two predecessor patterns selected for cross over is selected as "main". The process starts with taking every other node from each of the patterns and applying them to a new empty successor pattern. The pattern then gets connected using the connection pattern found in the "main" pattern. The size of the main pattern is kept. If the "non-main" pattern is larger, the extra nodes will be discarded. Similarly, if the "non-main" pattern is smaller, more nodes from the "main" pattern will be added into the new successor.

3.4.6 Fitness Calculation, Evaluation and Elitism

The fitness step for Pattern-NAS uses the training loop described in section 3.1. Before this the training can start, patterns must be joined using the joining method found in section 3.4.3. When these to steps are complete, the training loop will run until convergence for all the combined networks.

Since the results achieved from the training loop are for multiple patterns as a collective, the patterns must also share the results achieved. It's critical that all patterns are included in multiple training sessions by being joined in multiple networks. If not, the evaluation step will fail due to multiple patterns having the exact same performance. The evaluation method uses the weighted sum function described in 3.2.5. The fitness can now come from multiple training sessions. The best performing training session a pattern was included in is used to evaluate it. When the Pattern-NAS search finishes, the best joined EA-NAS model will be returned.

3.5 NAS front-end

A view was created to follow the progress of the NAS searches live. This view is a single-page application for the web browser built using the Vue.js framework in

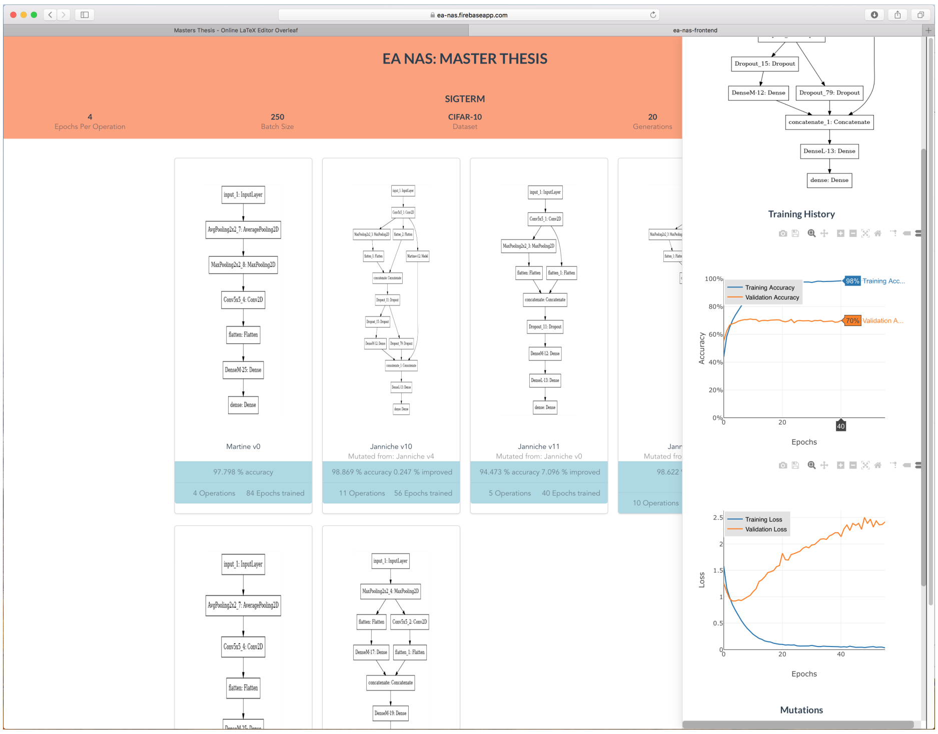


Figure 3.7: Screenshot of NAS Front-end showing detail view of one of the individuals in the population. The website can be accessed at `https://ea-nas.firebaseapp.com`

combination with semantic UI. A Firebase based back-end server was also created to store the data for use with the view.

The NAS search uploads it's progress to Firebase on certain checkpoints within the application. If possible, a generated image of the Keras model is also uploaded for debugging purposes. What the algorithm creates, what changes it does and what works is important. The website is updated automatically when any new data becomes available through the Firebase realtime database.

The view displays overview data on the main page including image of the neural network, the current validation accuracy, size of the genotype and how many epochs the phenotype has been trained for. When clicking on one of the individuals, a detail side panel appears with more information like plots of training-, validation- and test accuracy, loss and a log of all mutations and crossovers that have been applied to this genotype.

Chapter 4

Experimental Plans and Setups

Out of the three systems described in chapter 3, EA-NAS is the main experiment. Multiple different sorting algorithms will be tested against the weighted sum sort described in section 3.2.5. Can a sorting algorithm maintain diversity in the population of neural network genotypes? Further, is using a population fast at all? When only optimizing a single architecture, may changes converge faster than when optimizing a population of architectures? When using transfer learning with evolution, the weights are evolved with the architecture using both evolution and gradient decent. Can transfer learning speed up NAS? How much transfer learning should be applied? Experiments for these questions will be described in this chapter. First, the plans and purpose for each experiment will be discussed, then the specifics of each experiment. The results will be evaluated in chapter 5.

4.1 Why Evolve Neural Network Architectures

The idea behind using evolution to generate good neural network architectures is incrementally manipulating the architecture to improve upon it over time. Adding a layer to the architecture of a neural network may or may not yield better predictions from the neural network. Since there is no exact science for composing the architectures of neural networks [18], an optimization algorithm like evolution is a good fit.

The Lottery Ticket Hypothesis. “A randomly-initialized, dense neural network contains a subnetwork that is initialized such that - when trained in isolation - it can match the test accuracy of the original network after training for at most the same number of iterations.”
- Frankle, J. and Carbin, M. (2019) [16]

Reviewing the lottery ticket hypothesis which was also proven [16]. It’s clear that pruning is an important step when creating a neural network to keep it from being over-parameterized or too large. As reviewed in chapter 2.3.4, Proxyless-NAS [6] gets their exemplary results from taking an over-parameterized neural network and pruning it, following the lottery ticket hypothesis. They do not however explore as the over-parameterized network architecture used is static.

Can an evolutionary algorithm find a neural network architecture that is the subnetwork described in the lottery ticket hypothesis? The evolutionary algorithm proposed in this thesis may both prune and add to a neural network architecture, giving it the tools needed to find the subnetwork. Also to note, the lottery ticket hypothesis is only applied to ”Dense” neural network architectures and not convolutional which the contribution of this thesis aims to optimize.

4.2 Experimental Plans

The experiment for this thesis is exploring how a traditional evolutionary algorithm fairs when applied to neural architecture search. There are many factors that can improve upon the performance of EA’s. The representation is key– and a different representation for the EA will also be tested with Pattern-NAS. How these algorithms considers performance is controlled by sorting and elitism components. Multiple sorting techniques will be experimented with. There are four experiments planned. The experimental plan will go through each experiment and what the experiments aims to answer.

- **Experiment 1**: Performance of a traditional evolutionary algorithm for neural architecture search. How does different sorting algorithms affect performance? What are the best objectives for a multi-objective sorting algorithm when applied to NAS? What objectives will maintain diversity best? Is diversity required at all to achieve top accuracy results?
- **Experiment 2**: Hill climbing vs evolution, only changing a single genotype vs changing an entire population. Will hill climbing converge on a better architecture faster than evolution?
- **Experiment 3**: To truly find the differences in using or not using transfer

learning can only be found by isolating transfer learning during the search. Experiment 3 will take the exact baseline experimental setup and turn off transfer learning. What are the effects of using transfer learning with evolution-based neural architecture search?

- **Experiment 4**: Experimenting with another representation and explore performance differences. This more abstract representation will have less weights to use for transfer learning. Can knowledge be transferred when trained in multiple neural networks?

4.2.1 Experiment 1: Evaluating and Comparing Neural Networks

This experiment is based on testing if a traditional evolutionary algorithm based on exploration will perform well on neural architecture search. Using a direct representation means doing mutations on the phenotype. The genotype representation used in EA-NAS is not a direct representation. It's however very similar in structure to the phenotype. While a clever representation like the one used in [24] can give a faster search, it can also make the algorithm explore less and exploit more.

Using local search within evolutionary algorithms is quite common to enhance performance. Local searching typically hinders exploration while exploiting more by for example changing how the mutation and crossover operators work. These types changes might include doing specific optimization to the representation that is known to give better results. Since there is no exact science for how to build a neural network architecture optimally [18], local search is left out of all of the evolution based experiments.

One of the key goals of this experiment is to see if an exploration based evolutionary algorithm is viable for neural architecture search. A viable NAS algorithm have to find a well performing architecture in a short amount of time.

How Does Different Sorting Algorithms Affect Performance

For experimenting with sorting algorithms, there are three different configurations used in combination with EA-NAS. The first, experiment 1.1 is the weighted sum scoring described in section 3.2.5. The two others, experiments 1.2 and 1.3 are based on the multi-objective optimization algorithm NSGA-II [13] described in section 3.2.5, with a very different set of objectives.

The usual measure of how well a neural network performs is usually measured by the validation- or test accuracy. Is prediction accuracy the best measure of performance for a neural network or are there other factors that also describes the network? This experiment aims to compare different measures of performance for neural networks through multi-objective optimization.

Experiment 1.1: Baseline

This experiment will test sorting based on test-accuracy and overfit of the genotype as described in chapter 3.2.5. Here, the test-accuracy is maximized while the overfitting is minimized. The test or validation accuracy a widely used metric for both evolution-based and reinforcement learning based NAS algorithms [26; 31; 24; 6; 39; 5] and should therefore be part of the series of experiments for this thesis. Through the preliminary results, it was apparent that the most overfitted in the population always prevailed when only sorting on the test accuracy. The algorithm got stuck on a local maxima. The overfit score was therefore added with improved results. This will be compared against the following two sorting experiments:

Experiment 1.2: Knowledge Sort

The first multi-objective sorting experiment is based on the performance of each individual classifier task. This means the score of each class airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck of the CIFAR-10 dataset will be individually scored and maximized. The scores used are the F1-scores gotten from the classification report, described in chapter 3.1.

The goal here is to see if diversity can be measured in the knowledge held within a neural network instead of measuring something like structure. Here, transfer learning becomes even more important as evolution aims to pass good features through the generations. This is the foundation of any evolutionary algorithm.

Experiment 1.3: Structure Sort

Sorting based on architectural properties aims to optimize the architecture faster by focusing on well known layer-patterns. The objectives set here based on studying the state-of-the-art networks such as [19; 25; 39] and examples listed on the Keras website [7]. There are some clear patterns that are repeated and should be considered. The objectives are:

- **Test accuracy** is a required metric as this is the only real indicator of a good neural network architecture.
- **Max branching** A branch is where the outputs of one layer in the Keras model is used as input for two other layers. At the genotype's thickest point, how many branches are there? This objective is minimized.
- **Convolution count** counts how many convolutional nodes are included in the genotype. This objective is maximized
- **Double pooling**, Pooling layers may be placed in sequence. This is undesirable and is therefore minimized. What is desired is patterns of convolutional layers followed by Pooling layers as it is seen quite a lot in well performing CIFAR-10 examples.
- **Overall size**, the overall size of the network should be minimized. This objective contradicts the "convolution count" objective. Minimizing the overall size of the network is important as described for experiment 1.1, Baseline.

Two more objectives were considered but left out. An objective to control how many one-dimensional Dense layers are included in the genotype could be a good objective. There was little evidence of either good results with few dense layers or good results with many dense layers in the preliminary results. The other objective considered was based on searching through the genotype for convolution-pooling sequences. These patterns are desired as they are found in many of the state-of-the-art neural architectures [19; 25; 39; 35], and with enough exploration, the evolutionary algorithm will figure that out. The number of objectives should be minimized as this will make the frontiers easier to form.

Using the double pooling objective may also backfire as diversity is measured by differences. If the other objectives are the common denominators and double pooling is the difference, networks with double pooling will be kept to maintain diversity. This objective might have been better as a constraint on the mutation operator.

4.2.2 Experiment 2: Evaluate Performance Difference In Hill Climb vs Evolution

One of the main traits of evolutionary algorithms is that they are based on making small changes to the genotype that are fast to evaluate. This is not the case with NAS. Evaluation is slow. Small changes therefore needs to be precise to achieve a good architecture fast. The Local-NAS experiment, where evolution is replaced with hill climbing changes only a single genotype multiple times per generation.

Here, each change is a mutation to the best of the previous generation. Changes will be much more precise. Multiple changes are made and evaluated in parallel and only the "best" change is kept. This also means no diversity metric is required. The weighted sum sorting as described in chapter 3.2.5 will be used for this experiment. The sorting is exactly the same as used in Experiment 1.1, Baseline.

Although there may be a speedup in making correct changes to the architecture, Hill climbing local search is well known for getting stuck in local minimas or plateaus. This is exactly why maintaining diversity in the population is important for evolutionary algorithms. This experiment will test how stability and performance of the search is impacted by having a population of networks as opposed to a single network to optimize.

4.2.3 Experiment 3: Importance of Transfer Learning

The main contributions of this thesis is using transfer learning in neural architecture search. As mentioned earlier in this chapter, passing good features through the generations and improving upon them is the foundation of evolution. The survivors of each generation should be as good or better than the previous generation. For neural networks, the architecture is one of these features. A good architecture should be passed through the generations.

The architecture of a neural network is worthless without the proper set of weights. These weights represent the knowledge within each neural network, a feature that must be passed through the generations as it's key for a neural network to perform. This experiment will compare transfer learning baseline described above in section 4.2.1 to an identical experiment without transfer learning.

4.2.4 Experiment 4: Test Performance Differences With Representation

Genotype representation matters. When the fitness calculation per generation may take hours to complete it does not matter whether the rest of the algorithm runs in milliseconds or nanoseconds. The representation may however change how many generations is required to find a good architecture. An alternative representation is tested which is less similar to the phenotype. The Pattern-NAS representation described in chapter 3.4.1 will be compared to the EA-NAS representation. The Pattern-NAS representation is quite similar to the representation used with DeepCoEvolution [26] described in chapter 2.3.2.

The idea behind the representation is to use knowledge from other human created neural networks [19; 25; 35] which have repeating patterns that is known to perform well. The algorithm will have the freedom to evolve how these patterns are designed both in topology and nodetypes. Each pattern will also know their own preferred position in the network.

There is a clear trade-off in transfer learning when using a less direct representation. Because the patterns needs to be joined, only the weights within each pattern can be used with transfer learning. The rest of the trained weights of the predecessor will be lost. How much knowledge should be transferred from the previous generation? What are the effects of joining weights trained in separate neural networks?

4.3 Experimental Setups

For the experiments to be comparable, they have to have the same amount of time to find good architectures. Each experiment gets 72 hours to find a good neural architecture.. Over the course of these 72 hours, each experiment will have access to 12 GPUs. This gives each experiment 864 GPU hours.

The dataset used for the experiments is the open CIFAR-10 [20] dataset. It's is used in many of the experiments found in the state-of-the-art section 2.3. This dataset was selected as its common to use as a benchmark for image classifiers. Using the ImageNet dataset[14] was also a possibility but was discarded due to the immense compute power used for searching in the state-of-the-art NAS algorithms [6; 24; 39]. That amount of compute was not available while writing this thesis.

Please note:
All configurations used for the experiment can be found on the github repo for this thesis: `https://github.com/MagnusPoppe/NAS/tree/master/configurations/cifar-10/experiments`

Parameter	**1.1**	**1.2**	**1.3**	**2**	**3**	**4**
Dataset	CIFAR-10	CIFAR-10	CIFAR-10	CIFAR-10	CIFAR-10	CIFAR-10
Training samples	50000	50000	50000	50000	50000	50000
Validation samples	0	0	0	0	0	0
Test samples	10000	10000	10000	10000	10000	10000
Epochs	10	10	10	50	10	30
Batch size	90	90	90	90	90	90
Learning rate	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
Train until convergence	OFF	OFF	OFF	OFF	OFF	OFF
Scale epochs by layers	OFF	OFF	OFF	OFF	OFF	OFF
Use Transfer Learning	ON	ON	ON	ON	OFF	ON
Population size	8	16	8	11	8	12
Generations evolved	150	150	150	50	150	50
Use multi-objective sort	ON	ON	ON	OFF	OFF	ON
Sort by accuracy	ON	OFF	OFF	OFF	OFF	ON
Sort by classifier tasks	OFF	ON	OFF	OFF	OFF	OFF
Sort by architecture	OFF	OFF	ON	OFF	OFF	OFF
Min mutations	7	7	7	7	7	3
Max mutations	14	14	14	14	14	8

Table 4.1: Parameters used for all experiments. The table headers are experiment numbers.

4.3.1 Experiment 1: Elistism Sorting

The first experiment is to test how well a traditional evolutionary algorithm performs for neural architecture search. Below is a table of all parameters used by the different sorting experiments. There are three categories of parameters. First category describes the dataset used, the second category describes parameters for training the neural network phenotypes and the third category describes the parameters set for evolution. Please note that the only differences are within the "sort by" parameters and the population size.

For EA-NAS, the population size is increased by 50% after mutations are complete. Its important to keep occupancy of the GPUs at maximum during the fitness calculation step to run the algorithm as fast as possible. Meaning a population size of $8 \cdot \frac{3}{2} = 12$ neural networks training concurrently, maximizing occupancy. There are variations in the architectures which makes some networks train faster than others, so perfect occupancy is not possible. The workloads are sorted so that the biggest loads are queued first. For when the population size is bigger than the amount of GPUs available, smaller loads will be queued on free gpus when some other workloads finishes. Some GPUs will then only see one large workload while others might run multiple smaller workloads in the same amount of time, maximizing occupancy.

4.3.2 Experiment 2: Hill Climb vs. Evolution

The parameters used with experiment 2 is based on being comparable to baseline, depicted as accuracy sort in table 4.1. The parameters are mostly the same. Note that to maximize occupancy, the population size is different due to how populations are built in a local search algorithm as compared to evolution. Local search doesn't use population and only holds one complete solution at a time. Population size in this context means the number of mutations being performed to the current solution held by the local search algorithm. Every mutated solution will also be evaluated in the fitness step. Occupancy of the GPUs should therefore be considered with setting the population size. Here, 11 in population size means 11 mutated predecessors plus the previous best trained on 12 GPUs simultaneously per generation. This is maximum occupancy.

4.3.3 Experiment 3: Turning Off Transfer Learning

This experiment is an exact copy of the Baseline experiment 1.1 described in section 4.1, only without transfer learning. Continuous training will however occur meaning for each generation if a neural network survives and is not mutated, training will continue with the weights from the previous training session. This is the same behaviour as with Baseline experiment 1.1. If a mutation or crossover operation occurs, transfer learning will not be used. The weights of the newly mutated network will have its weights randomly initialized.

4.3.4 Experiment 4: Using The Pattern Representation

Experiment 4 is based on running the Pattern-NAS system. There are a few key differences for occupancy and population size compared to the EA-NAS experiment. The number of networks trained will always at max occupancy. For a 12 GPU setup, the patterns will be joined into at least 12 networks. The population size parameter here describes how many patterns are in use at any time. Similarly, the Min and Max size replaces the "initialization min/max mutations". Min- and max size represents the number of patterns a phenotype can be joined by. The largest phenotype for this experiment is $8 * 4 = 32$ layers, excluding regularizers. Remember from section 3.4.1, a pattern may contain between 2 and 4 nodes.

Chapter 5

Results and Discussion

This chapter evaluates the results achieved by each of the experiments described in chapter 4. First some remarks about the results to properly understand the context. Raw results follows with evaluation. The results are discussed in the last section.

5.1 About The Experimental Results

The number of generations reached for each experiment must be discussed before diving into the results. Although all simulations runs on the same number of compute nodes with the same number and models of attached GPUs for the exact same amount of time, there will be large differences in number of generations reached by each experiment. There are some factors that will determine the number of generations reached:

- The size of each neural network makes a huge difference in training time and evaluation time. This is especially a factor with Pattern-NAS, experiment 2 where the average size of each neural network generated is much larger than what EA-NAS produces.
- The different types of layers has vastly different backpropagation compute times. A dense layer is much faster to compute backpropagation for than a convolutional layer.
- The population size is not the same for every experiment. Most of the experiments will have a population size that equals the number of compute

nodes available with the exception of Experiment 1.2. Experiment 1.2 has a population size double that of the other EA-NAS based experiments. Experiment 4 can with a high probability create a few more jobs than the compute nodes available.

- Exceeding the GPU memory limit. This occurred in all of the experiments at least once. Exceeding the GPU memory occurs when the neural network is so big that it uses more than the 16 GB of vRAM available on the Nvidia P100 GPUs used for these experiments. GPU Memory has to be dumped to system memory when this problem occurs.

Also to note, the fitness step, as described in chapter 3.1 has a runtime of the critical path of all jobs. The critical path known from job-shop scheduling problems is the longest overall timed job. Measures was taken to prevent this for experiments where there are more jobs than compute nodes. This is the case with experiment 1.2 Knowledge sort. For the rest of the job, the measures taken are irrelevant due to every job always being started at once. The job that takes the longest time to finish will always be the critical path.

Because the number of generations reached varies, the x-axis on the different plots are often depicted in hours making all of the experiments comparable. The common denominator is 72 hours of runtime.

Please note: All plots of the results were created using a Jupyter notebook found at: `https://github.com/MagnusPoppe/NAS/blob/master/LAB/evaluation.ipynb`. As mentioned before, multiple different configurations was experimented with to compare what methods works best. There are therefore multiple experiments to plot. Table 5.1 describes the naming translation.

Baseline	Name of experiment 1.1 which All other experiments are compared to. This experiment uses weighted sum sorting.
Knowledge Sort	Experiment 1.2, knowledge sort uses Multi-objective-optimization with knowledge based objectives as described in section 4.2.1.
Structure Sort	Experiment 1.2, structure sort uses Multi-objective-optimization with structure based objectives for optimizing neural architecture as described in section 4.2.1.
Local	Experiment 2, the hill climb local search comparison found in section 4.2.2
Baseline w/o TL	Experiment 3 (section 4.2.3), turning transfer learning off for the baseline experiment.
Patterns	Experiment 4 (section 4.2.4), experimenting with a different, more abstract representation.

Table 5.1: The naming scheme of the individual experiments for the results.

System	**% error**	**# Parameters**	**GPU-Time**	**Augmented**
Exp 1.1: Baseline	13.53	2,051,564	864h	No
Exp 1.2: Knowledge sort	14.29	1,573,252	864h	No
Exp 1.3: Structure sort	11.75	2,913,066	864h	No
Exp 2: Local	19.82	23,955,856	864h	No
Exp 3: Baseline w/o TL	16.80	2,033,252	864h	No
Exp 4: Patterns	19.95	77,369,478	864h	No
cuda-convnet [Cif]	18.00	Not Disclosed	Not Applicable	No
DeepCoEvolution [26]	7.30	Not Disclosed	Not Disclosed	Yes
Hierarchical Rep [24]	3.75	Not Disclosed	7200h	Yes
NasNet [39]	2.40	27,600,000	48000h	Yes
ENAS [30]	2.89	4,600,000	8h	Yes
ProxylessNAS [6]	2.08	5,700,000	Not Disclosed	Not Disclosed

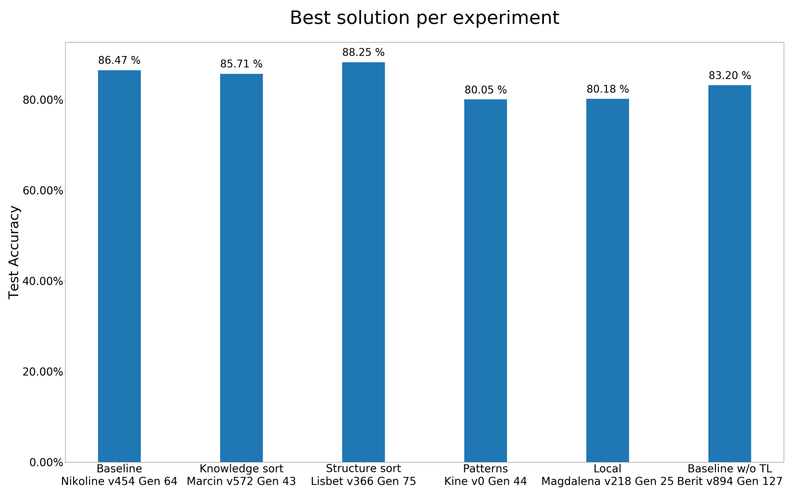
Table 5.2: Comparison table between the experiments, baseline and state of the art neural architecture search algorithms. The different categories are separated with a horizontal line, where the first category is the results from these experiments, the second category is baseline scores from CIFAR-10 website [Cif] and the third category is state-of-the-art neural architecture search algorithms.

5.2 Experimental Results

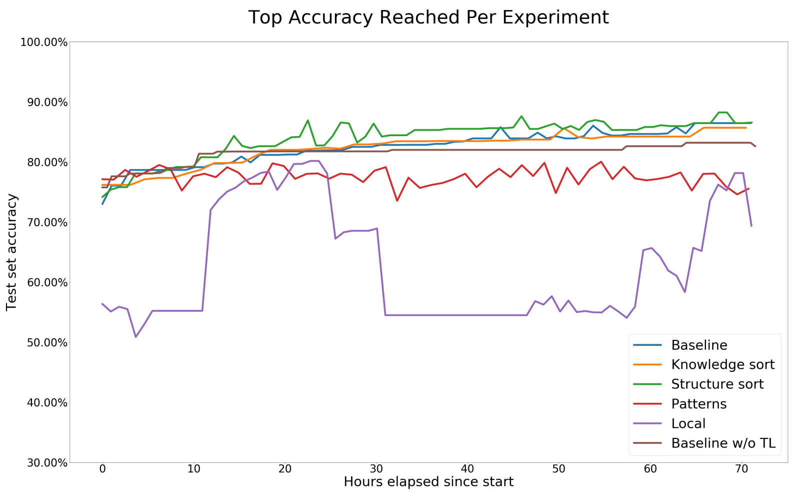
The results from figure 5.3 visualizes top prediction accuracy achieved for every experiment. Table 5.2 compares these results to both the baseline from the CIFAR-10 website called cuda-convnet and studies from the literature review. Looking at the table, the results achieved by the systems proposed in this thesis is far from beating the state-of-the-art neural architecture search algorithms. These are not comparable to the proposed system due to dataset augmentation which is further discussed in section 5.4.4. Comparable results are however found in the human expert designed neural network cuda-convnet which is as mentioned above the CIFAR-10 baseline. Here, all EA-NAS based experiments (1 and 3) outperforms cuda-convnet, while Pattern-NAS and Local-NAS based experiments (2 and 4) performs worse. Structure sort is the strongest performer of the experiments run for this thesis with the top score at 88.25% accuracy on the test set which is a 6.25% improvement over the CIFAR-10 baseline. A decisive best performer of the experiments cannot be proved before multiple runs of each experiment has been performed as discussed in section 5.4. From figure 5.1b we can see that the experiments are fairly stable at the $75\% - 85\%$ range. The pattern experiment scores decreased over time, while the rest of the experiments improved but only slightly from what the initial population achieved.

The parameters of the phenotype models for EA-NAS is much smaller than the state-of-the-art neural architecture search algorithms seen in table 5.2. The all of the EA-NAS top generated networks from experiment 1 and 3 were under half the size of the smallest state-of-the-art network. This is positive as with fewer parameters, the networks are both faster to train and run inference on. These networks may however be too small to truly generalize. This might have constrained the score. For the experiment 2, Local and experiment 4, Patterns the model sizes were very different. The local experiment had exactly the same parameters as experiment 1 and 3 for network sizes and probability distribution for selecting different node/layer types. It still ended up with the giant model. The Pattern experiment used an entirely different sizing of each neural network which led to the truly gigantic size of the top individual through the generations.

72 hours of GPU time is much less than the state-of-the-art. This is with the exception of ENAS, which only used 8 hours on the same dataset. As discussed in the state-of-the-art, the search space of ENAS is really restricted - leading to much less exploration and also less GPU time. When comparing against Hierarchical representations for Efficient Neural Architecture Search [24], the Baseline experiment used only 12% of the GPU hours and when comparing to NasNet [39], only 1.8% of the GPU hours. The scope of these searches is then much larger than what was possible with the experiments of this thesis.

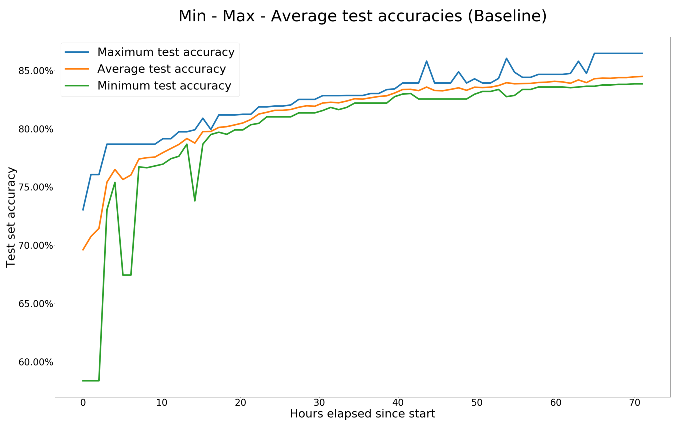


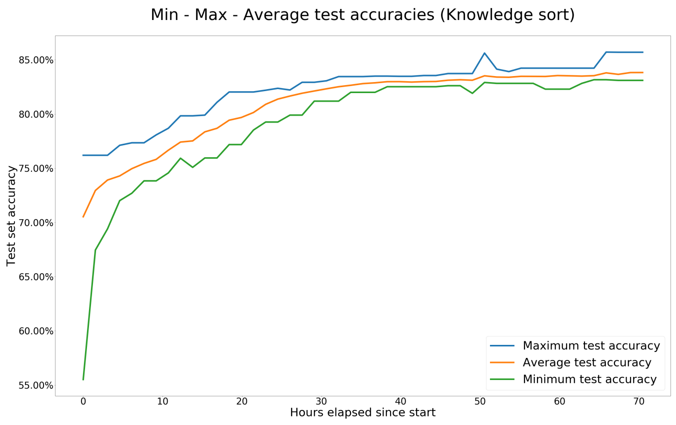
(a) This is the best prediction scores reached for each of the experiments. The score is measured in prediction accuracy on the test set of CIFAR-10. The text below each bar describes what experiment and which specific individual holds the score and at what generation in the evolution this score was achieved.



(b) Timeline of test set accuracies reached for the 72 hours of running for all experiments

Figure 5.1: Overall accuracies on the CIFAR-10 test set displayed in two formats.





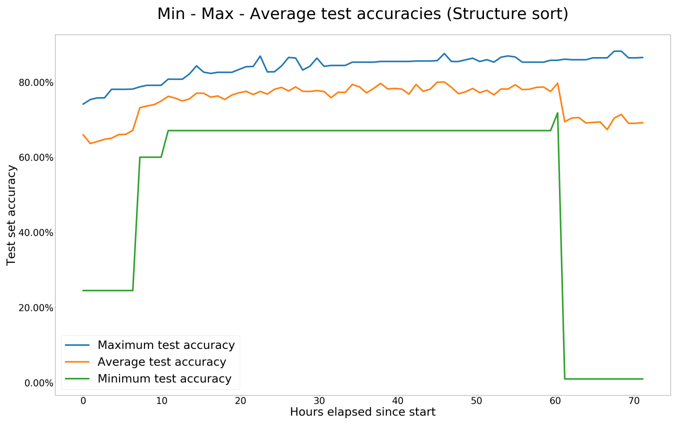


Figure 5.2: Comparing the minimum, average and maximum accuracies reached on the test set over hours of runtime

5.3 Evaluation

This section evaluates each of the experiments in turn, starting with experiment 1, sorting where performance and architectural proprieties are in focus to see whether the objectives can maintain diversity and a stable performance improvement over time. Further we will look at the significance of diversity through the local experiment 2 before going through a comparison of baseline with and without transfer learning. This section ends with an evaluation of the results of transfer learning using a different representation through experiment 4.

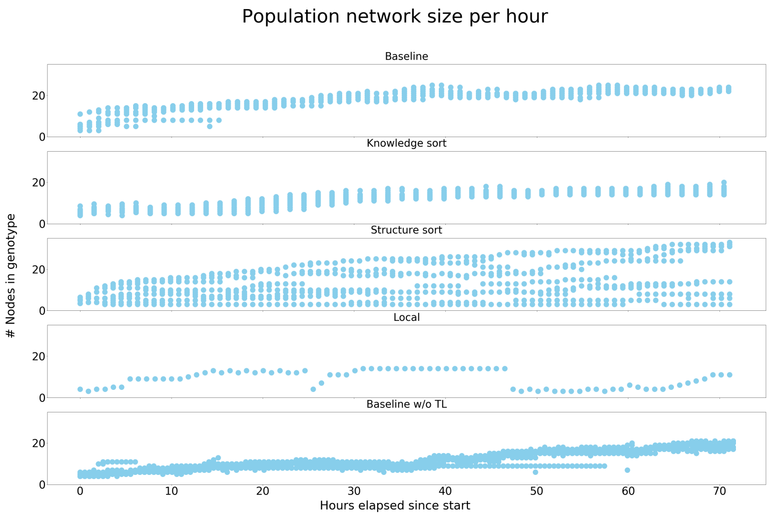
5.3.1 Experiment 1: Evaluating Sorting Techniques

All three of the sorting techniques tested scored well above the cuda-convnet. Although all of these experiments performed well, the stability in the search was somewhat different between them. From figure 5.3 we can see that for the Baseline and Knowledge Sort experiments have a more stable increase in both the minimum, average and maximum test accuracies compared to the Structure sort experiment 1.3. This is seen through the distance between the lines. For Baseline and Knowledge sort, the lines grows closer and closer with some spikes. Structure sort on the other hand hold a much higher distance between them. This might be a sign of diversity in the population. The minimum of Structure sort is the same individual from hour 11 to hour 59. This genotype is scoring high on the architectural objectives but not on the accuracy objectives. Scoring the worst in the accuracy category also yields a high diversity score which makes this network desirable.

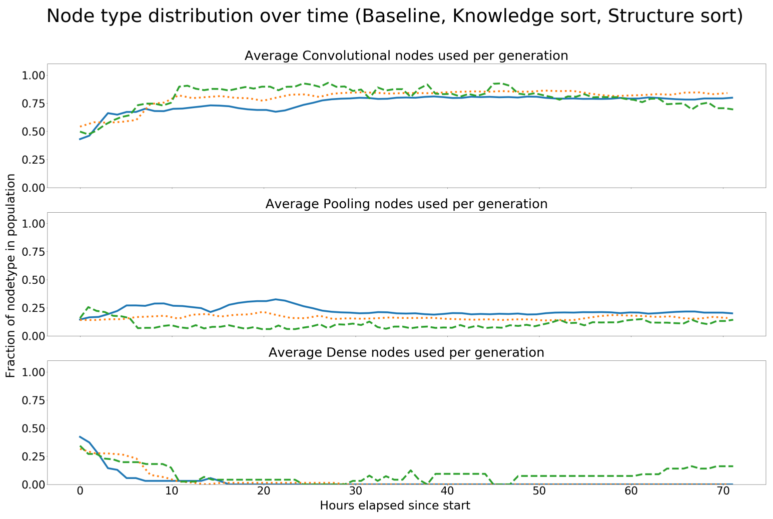
Architectural Properties

The neural architectures generated by any of the EA-NAS based experiments starts out small. As seen in figure 5.3b, almost all of the addition done to these initially small architectures are convolutional nodes. The average convolutional nodes used grows almost immediatly to above 60% of the nodes in use. This is common for the three experiments. The algorithm finds by it self that using convolutional layers gives the best results for the CIFAR-10 dataset. Similarly, pooling layers are kept at a minimum but are in use for all experiments. Dense layers however remain completely unused after just a one third of search time. The dense layers are further discussed in section 5.4.3.

From figure 5.3a we can see the number of nodes each individual genotype holds per generation. Each dot is the size of one individual genotype. Diversity



(a) Each single blue dot represents the number of nodes held by a single genotype at one point in time.



(b) Node type distribution used by the evolutionary algorithm over time. Green is Structure sort, Orange is Knowledge sort and Blue is Baseline. The hour axis is shared for all meaning that the sum of a particular hour across all three plots is 1 as each hour represents the average.

Figure 5.3: These two plots show how the population changes in stucture in terms of size and node types over time.

in sizing for Baseline, Knowledge sort and Baseline w/o TL are fairly similar with each of them growing over time. There is not much diversity in the sizing of the genotypes as there is little spread of the blue dots. Structure sort on the other and has an objective to minimize the size of the genotypes. This objective is being diversified as the spread of the blue dots grows over time. There is then high diversity in the sizing of the population. Similarly, the use of pooling nodes is lower in the Structure sort experiment 1.3 than with any of the other EA-NAS experiments. This is clearly shown in figure 5.3b. The same figure also shows that the the usage of convolution nodes is maximized as compared with pooling and dense node usage. The usage is however not much higher than with the other sorting algorithms.

Using branches is also favored in all of the EA-NAS based experiments. All of these experiments have a large factor of branching when compared to the overall size of the network. The figure 5.4 shows the most branching individual in the population for each generation along with the size of the same individual. From the plot, the most branching in the population is very short and wide. At one point in the "baseline without transfer learning", both the lines meet. This means all of the nodes in the network are in parallel.

Sorting on Knowledge

The different classification classes was used as objectives in experiment 1.2, knowledge sort. All of the individual classes in the CIFAR-10 dataset are plotted in figure 5.5 for experiment 1. These plot shows the difference between minimum and maximum compared to average accuracy achieved per generation. When comparing Baseline to Knowledge sort, there is a larger difference in spacing between the minimum, maximum and average. This difference is however minor. This metric compared to just using test- or validation accuracy is the same in terms of performance.

5.3.2 Experiment 2: Hill Climb Local Search Compared To Evolution

This subsection evaluates some of the results connected to the "Local" experiment 2. By looking at figure 5.6 it's clear that at around hour 30, a poor decision was made by the elitism. This decision was choosing a genotype which was less overfitted, compromising on the test accuracy score. Local search only holds a single genotype. All later discoveries made by the search was affected by this. The weighted sum (section 3.2.5) might be the wrong operator for this experiment.

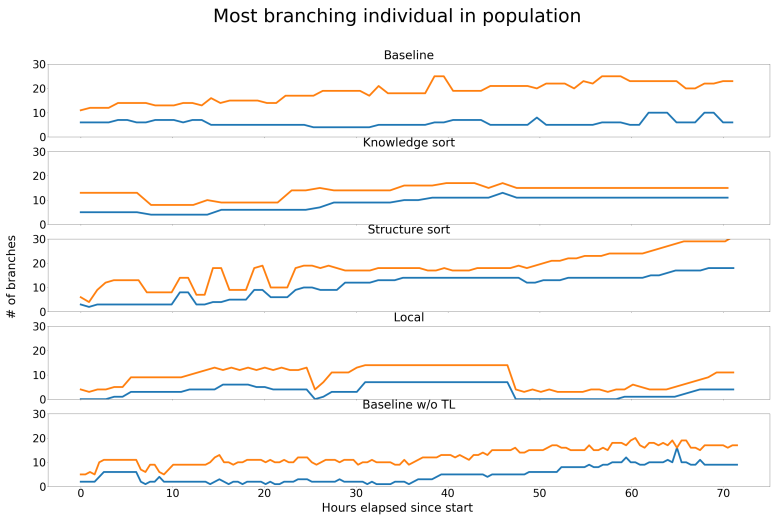
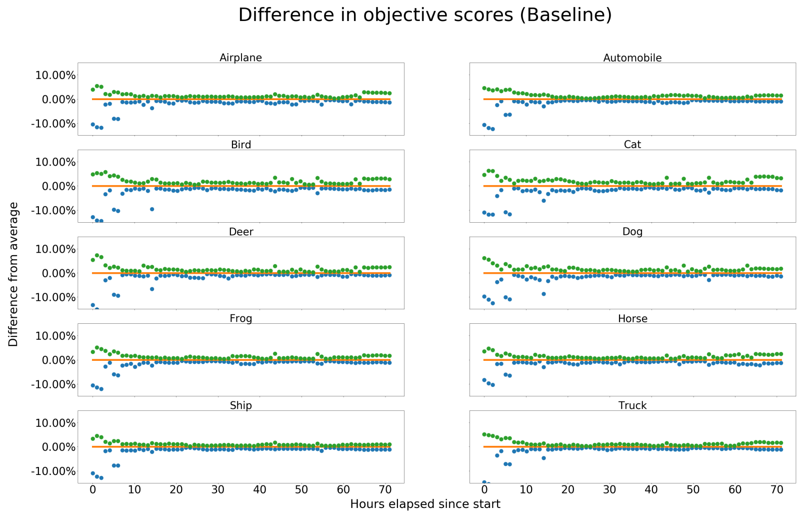


Figure 5.4: Branches within each neural network appear when the computational graph is forked. Here, maximum branching, the widest point in the network is measured for the single most branching individual in the population per generation. Blue line indicates number of branches while orange line indicates the number of nodes held by the same individual.



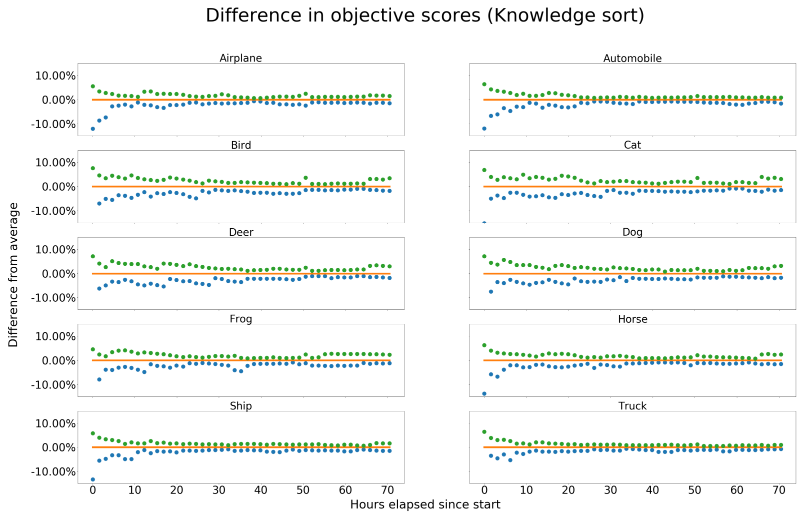


Figure 5.5: This figure shows difference in accuracies within each class for the classifer tasks in CIFAR-10. For each of the plots, blue is minimum accuracy and green is maximum accuracy. The orange line is average accuracy contained at 0. Each line of dots represents a single generation.

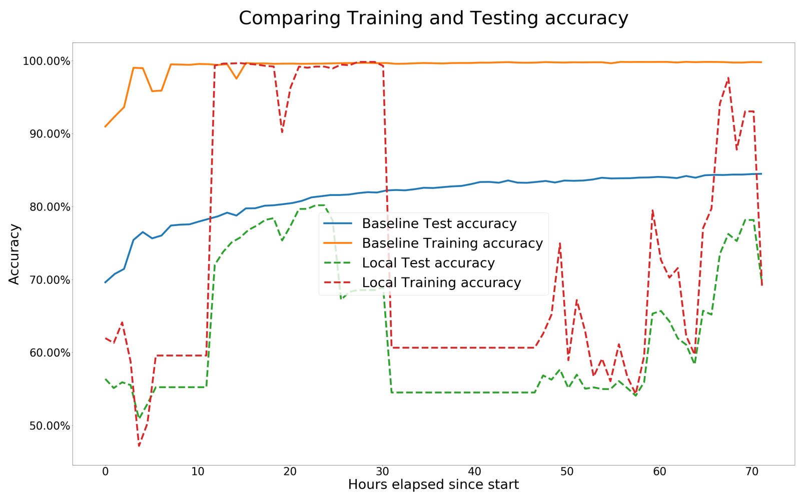


Figure 5.6: The performance of Local compared with Baseline.

This decision wouldn't have been destructive if there had been more solutions to mutate in the later generations. The Bad results that follows hour 30 are a direct result of this. Comparing this to Baseline which uses the same function for elitism - weighted sum as described in chapter 3.2.5 - the results of Baseline is much more stable. These kinds of decisions has much less impact when holding a population. It's also apparent from figure 5.6 that baseline has a much more stable search.

This really shows the value of holding a population when searching through computationally expensive environments. None of the other experiments suffered from this kind of bad decisions.

From figure 5.8c we can see that the training set accuracy for the best individual of the population was much more unstable than what was seen in figure 5.8. This individual was also less overfitted than what has been previously seen as shown by the distance from green dot representing test set accuracy and the blue line representing training set accuracy.

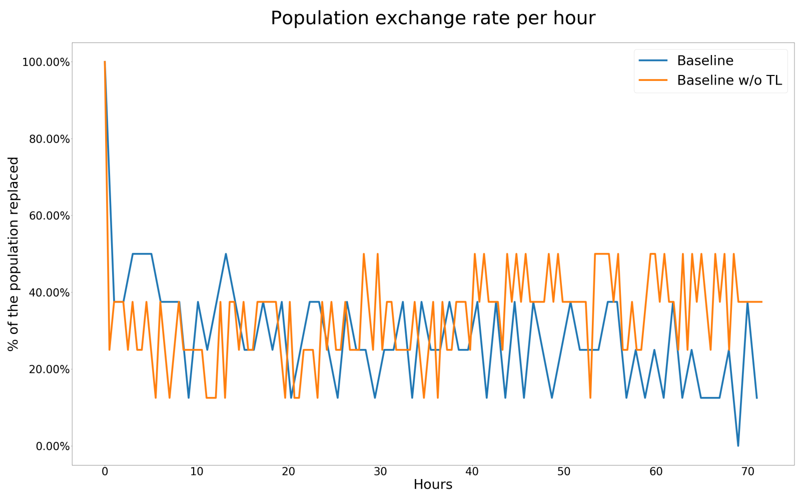


Figure 5.7: Comparing the baseline to the baseline without transfer learning. This plot shows the percentage of the population that is replaced per generation. The X-axis is set to hours for comparability.

5.3.3 Experiment 3: Turning Off Transfer Learning

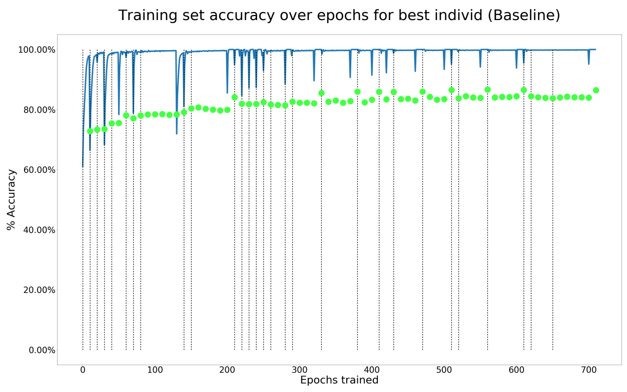
When studying figure 5.7, the population exchange rate which describes how many individuals in the population dies and are replaced by new offsprings show a clear decline when using transfer learning. New offsprings have a much slimmer chance of surviving without transfer learning due to the disadvantage of having less training epochs when spawned. The new offspring phenotypes will then compete with phenotypes that has been trained over many generations. This might mean that good genotypes are discarded due to bad performance when in reality they are not finished training. A solution to this is training each phenotype (Keras model) for more epochs per generation. This would lead to much more time spent on the fitness step. The transfer learning does in fact yield a speedup in the fitness step per generation given that under-training is the problem.

Training is also more unstable when not using transfer learning. Figure 5.8a plots the lifetime of the single best genotype and shows clearly that over time, the training- and test accuracy stabilizes when using transfer learning. When not using transfer learning, the training remains unstable throughout the generations. As seen by the black striped lines representing each mutation performed, mutations also becomes less frequent over time. With improvements over time, good mutations becomes harder to achieve. This is only the case when using transfer learning and may be related to the point above, the non-transfer learning baseline individuals may be trained to little to prove their performance and are therefore replaced with a new mutation.

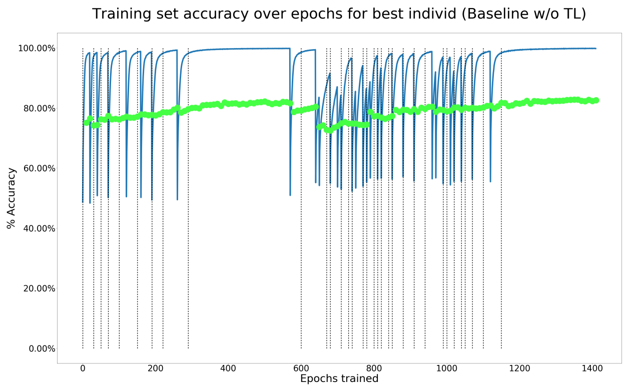
5.3.4 Experiment 4: Using An Alternative Representation With Patterns

The accuracy achieved by the neural networks generated by joining evolved patterns was steady at 78-80% accuracy on the test set throughout the 72 hours. The patterns used almost immediately shrunk down to 2 layers for the entire population. When the patterns are that small, transfer learning will have very little effect on the training as this is no more effective than randomly initializing the neural networks.

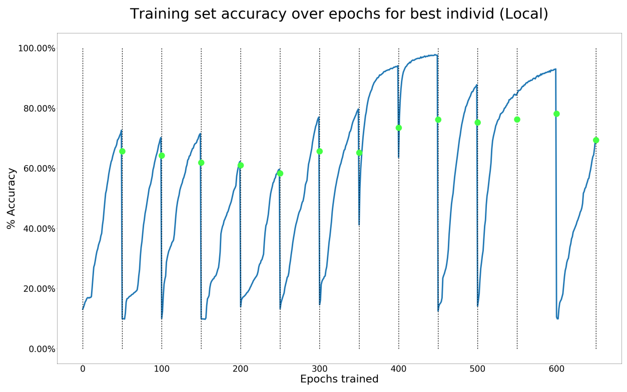
Figure 5.9b shows how much accuracy is gained for each generation by training on average. Both the baseline w/o TL experiment 3 and the Pattern experiment 4 has almost no changes in achieved accuracy per training session. For the different variations of experiment 1, Knowledge- and Structure sort and baseline on the other hand, the accuracy gained per session is low. This means less training is required for the neural network to perform. From figure 5.9a we see that training



(a)

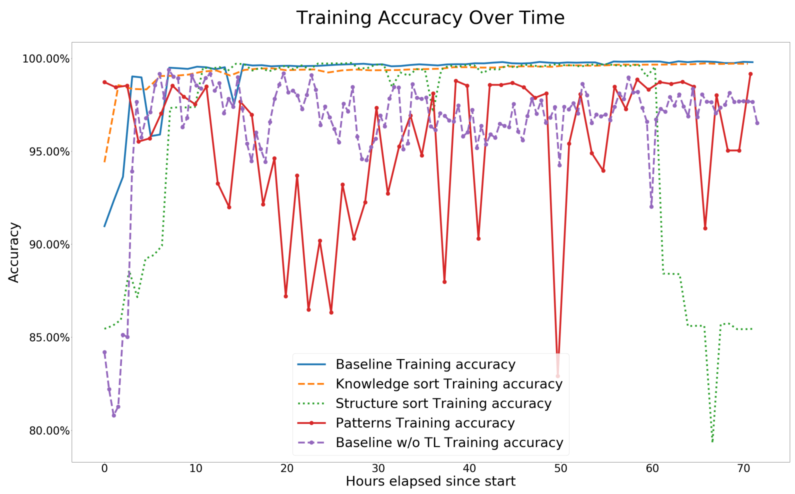


(b)

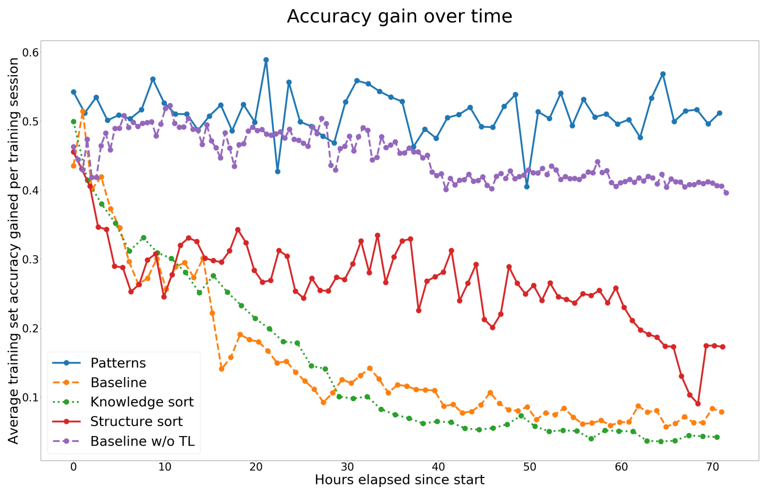


(c)

Figure 5.8: This plot shows the lifetime of the best individual seen. Blue line is training set accuracy and green markers are test set accuracy. The black striped lines represent each mutation added to the genotype.



(a) Accuracies achieved over time on Training set



(b) This figure shows how much accuracy was gained for a training session, averaged over the population for each generation. Decoding this, each point is the average difference in prediction accuracy between the first and last epoch of training for the generation. The training set is used for this particular plot.

Figure 5.9

set accuracy is high throughout the training time, with fluctuations in the range of $[80\% - 100\%]$. Since we then know that accuracies are high for all experiments, transfer learning is working. Less accuracy is lost for experiment 1.

Combining knowledge from different neural networks through patterns did not work. When also looking at the average size of each pattern, they shrunk down from an average of 3 nodes in the first generation to all containing the minimum of 2 nodes after only a few hours of evolution. This has definitely had an effect on transfer learning as only weights within each pattern is transferred. This is to few weights to make any difference in transferring knowledge.

5.4 Discussion

The first and most obvious limitation of these experiments is that they are only ran once. The systems proposed in this thesis are non-deterministic which means one input may yield multiple outputs. There is a lot of random used in these experiments, both with mutation operators and when generating the initial neural network weights which makes the algorithm non-deterministic. Running the same experiments multiple times would validate the results. As there was limited time to do these experiments and limited capacity on the compute cluster used, i was unable to do repeated runs of the same experiments. This means no conclusion can be drawn by looking at any of these results.

5.4.1 Evaluating Neural Networks Based On Learning Tasks

Using the individual classifier tasks as the performance metric in experiment 1.2, Knowledge sort gave similar or slightly worse scores than just using accuracies on the test set. Observing figure 5.5, its apparent that all networks generated performs quite similar on the different learning tasks. This might be CIFAR-10 related as some classes are much harder to learn than others. Specifically, the Cats, Horses and Dogs classes proves difficult for the neural networks while the rest got better scores. Could it be that given some specific data, the neural networks with somewhat similar structure and similar amounts of training always gives similar but not identical predictions? If so, this sorting metric is of no more use than just having the test set accuracy as a single metric for performance. This should be validated with a different dataset as the CIFAR-10 dataset gives very similar results per class for all of the networks generated across all of the experiments.

Scalability

A neural network classifier's architecture is good if it performs well over all classes. When using the classifier objectives, the crowding distance give diversity on the learned knowledge of each neural network. This is what experiment 1.2, Knowledge sort aims to achieve. This is however not a scalable metric. The more objectives used, the harder domination is to achieve. The number of objectives should therefore be at a minimum. For the CIFAR-10 dataset, 10 objectives will work as long as the population size is at least double the number multi-objective objectives. When evaluating the CIFAR-100 dataset which holds 100 different classes, this sorting metric would be useless as the population size needs to be huge.

Each class will get one network that performs worst and one that performs best. Both of these will get max score due to the design of NSGA-II. In the worst case scenario for CIFAR-10, this would mean a population of 20 every network could get max score making every network equal. This is however very unlikely to happen. For the CIFAR-10 dataset used in this thesis, this metric works. It should however not be used with dataset containing more classes.

5.4.2 The Structural Multi-Objective Optimization Objectives

Remembering chapter 4.2.4 there are 5 objectives for the Structure sort to optimize, Test Accuracy (Maximized), Max branching (Minimized), Convolution count (Maximized), Double Pooling (Minimized) and Overall size (maximized). As stated in section 5.3.1 all of the objectives are diversified and works as expected. The max branching objective on the other hand is not. Figure 5.4 shows how the branching in the experiment 1.3, structure sort is in fact higher or the same as the other experiments. As this is the only experiment which minimizes branching, the expectation was that branching would be less used with this experiment. Structure sort failed to complete this objective.

After analysis of the max branching function shows that for some genotypes, branching is calculated wrong, giving many more branches than what actually exists in the genotype. For any future work this feature should either be removed or rewritten to get correct branching on each genotype.

5.4.3 The Search Space

The baseline architectures generated, as seen in figure 5.3b ends up using zero dense layers in the final solution, with the exception of the forced output layers which is a dense layer matching the classes in the classifier task. For reducing the search space, dense layers can be removed in the future work as they remain unused anyways. This would shrink the search space from 7 to 4. Reducing the search space like this would increase the probability of the algorithm making a good mutation or change to the genotype.

5.4.4 Achieving Higher Performance

It's quite common to augment the dataset for CIFAR-10. Common augmentations include flipping all images 180 degrees, rotating images $+-(3-10)\%$, applying color and contrast filters to the images. By adding this together, the images trained on can be increased from the original 50000 up to 200000-300000 images. This will of course increase the generalization of the model in a large degree. As seen in table 5.2, all but the ProxylessNAS neural architecture search algorithms use image augmentation to achieve their very high test set accuracy. It's not stated in the paper [6] or on any of the review websites studied with this thesis whether ProxylessNAS uses augmentations or not.

Most of the other state-of-the-art neural architecture search algorithms reviewed in the literature review section 2.3 use much more compute power than what was available while writing this thesis. For comparing scores against NasNet [39], DeepCoEvolution[26] or Hierarchical representations for neural architecture search [24], much more compute power is required. More compute power along with using dataset augmentation is the key to increasing the performance the system proposed in this thesis.

5.4.5 Transfer Learning And The Lottery Ticket Hypothesis

The lottery ticket hypothesis [16] as described in section 4.1 states that any neural network contains a subnetwork which yields the same or better performance. They reason this by stating that initializing a neural network gives a few winning tickets. The winning ticket is an analogy for a good weight configuration on a set of neurons which yields good performance. These neurons may by them self be the only contributor to the prediction results. A network which holds winning tickets can be pruned to almost only contain the winning tickets while still maintaining the same or better performance and training time.

Pruning these neural networks can be done with up to an 80% reduction in number of parameters for [25]. The pruning procedure is done by first training the network to completion. Then - over multiple iterations - prune neurons with low contribution i.e. fewer activations. The training is then reset using late resetting. Late resetting is resetting the weights to the previously trained networks weights on a stage that is very close to the start of the training. This is then iterated on until a good architecture has been discovered.

The system proposed in this thesis tries to minimize the number of parameters found in the neural networks generated. If the first neural network in a strain (genotype family) is initialized such that it contains no winning tickets, it may never improve. This is due to transfer learning transferring the non-winning tickets. This is however a rare problem that would likely be discovered and handled by the elitism step. There is still a possibility of a good architecture being found by the search process, but because of the initial weights being transferred from the predecessor, it will never perform as well as it could have.

Transfer learning has yielded good results when it comes to transferring good features through the generations. Some networks will suffer the same bad traits that their predecessor had, like overfitting or low prediction accuracy. This seems like a good trade-off to make.

Transfer Learning's Effect on Exploration

As knowledge is transferred through the generations, most of the neural networks starts performing quite similarly as compared to not transferring knowledge. Predictions in different classes of the dataset are mostly the same with only minor variations. Diversity then might be worse when using transfer learning.

5.4.6 Crossover

A major limitation of the evolutionary neural architecture search algorithm proposed in this thesis is the lack of a crossover function. Crossover is a major part of what makes an evolutionary algorithm perform well. Finding a good crossover method proved hard for a couple of reasons:

1. How does one simply join two directed acyclic graphs?
2. How much learned knowledge is acceptable to lose?

The most important one of these questions is the second one. When crossing over two genotypes, many of the trained weights will be lost as a result of joining two genotypes. As this knowledge is one of the key elements that makes EA-NAS perform, losing knowledge is a hard trade-off to make. This question is closely related to the first question. The way two genotypes are joined decides how much knowledge the successor loses. As an example, striped crossover as described in section 2.1.1 takes every other node from each of the parents and joins them together in an alternating pattern. A striped crossover would mean 100% loss in knowledge as no weights are compatible. A K-point crossover implementation also described in section 2.1.1 would only mean K weights lost, keeping most of the knowledge. How does one select a point for crossover when two directed acyclic graphs have very different shapes?

Although some crossover methods were tested, all of them gave worse performance in the preliminary results for this thesis. All of them were discarded. The tested methods are described in section 3.2.3. A crossover function should be researched for EA-NAS.

Chapter 6

Conclusion

This chapter starts with going addressing the goals and research questions stated in chapter 1.2. There were two main goals for this thesis, each with their own research questions. These will be reviewed in turn. The contributions of this thesis are then described in section 6.1 followed by some suggestions for how to improve upon this work is listed in section 6.2.

The original description of this thesis was to explore the intersection between evolution and deep learning. There was many themes to choose from and after reading quite a bit, the focus landed on neural architecture search using evolution. The goals for this thesis where too see whether an evolutionary algorithm was viable for neural architecture search. Extending this, transfer learning became a focus area to speed up evolution as well as exploring how to compare neural architectures. These themes fits well with evolution as passing on features is a key function of evolution. Comparing the architectures also comes into play when performing elitism.

Goal 1: Explore and compare what effects diversity has on a traditional exploration based evolutionary algorithm evolving neural architectures

RQ 1.1: What objectives can be used to maintain diversity in a population of neural networks when using multi-objective optimization? To summarize, there were two sets of objectives used in the experiments. The first set (experiment 1.2) was based on seeing whether the diversity in the knowledge of each neural network could be used as objectives for the elitism stage of the evolutionary algorithm neural architecture search. There was little diversity in the knowledge learned by networks generated. Using the knowledge as a diversity metric was no more effective than just sorting on the test set accuracy. Important to note, this may be a CIFAR-10 specific result as other datasets might make the networks behave differently.

The second set (experiment 1.3) of objectives was based on finding diversity in the architectures of the generated neural networks. The objectives used was based on common architectural patterns found in literature. Diversity with these objectives was indeed high throughout the search, keeping architectures that were very different in shape, connectivity, node types and performance. The experiment using these objectives was also the best performer of all the experiments. Structural objectives works well for diversifying neural networks.

RQ 1.2: How does an evolutionary algorithm compare to hill climbing where no diversity is maintained? To start, having a population at all is a huge advantage. As seen from the "local" experiment 2, a bad decision can change the entire trajectory of the search. Stability of the search process is very important as all decisions made when doing neural architecture search are very costly. A bad decision has a smaller impact when there is an entire population of solutions to fall back on. Using any goal, a population based search is better suited for neural architecture search.

A good objective is key to diversifying neural network architectures. As stated above, using the correct set of objectives means a diverse population of neural networks can be discovered. Architectural objectives worked well for this task while knowledge based objectives performed well but did not maintain diversity for it's own objectives.

Goal 2: Explore transfer learning as a method to speed up the evaluation step of the evolutionary algorithm.

RQ 2.1: What effects does transfer learning have on the evaluation step? There are a few key effects from using transfer learning in neural architecture search that has been discovered through the experiments and result analysis. Transfer learning does in fact speed up the learning process by giving any offspring weights that are known to perform well. Very few if any epochs of training is required to reach the same performance achieved by the predecessor. When compared to not using transfer learning, evidence pointed to under-training in the non transfer learning based experiment, showing that more epochs of training was required to let new mutated offsprings have a chance of survival. With the same amount of epochs, the performance comparison between a long-lived genotype and a new genotype was unfair.

Stability is also improved using transfer learning as the increase in network performance is steady over generations. Networks are trained for more epochs without having the unfair comparison stated above. small changes to the architecture does not mean having a huge loss in knowledge. This stability comes at a cost. The knowledge of each network is less diverse over time as the networks has more or less the same knowledge. This is then a trade-off.

RQ 2.2: What is the effects of only transferring some of the predecessor weights using an alternative representation? Having a less direct representation gives a larger loss in knowledge between each generation. This was known before doing the experiments. Since there is a diversity trade-off to using transfer learning, how much of the knowledge should be kept through the generations? More knowledge gives better stability and higher performance. The pattern experiment had too small patterns for transfer learning to yield any better results than random initialization of weights. A more direct representation is then required.

6.1 Contributions

This thesis makes a few contributions. The first being a evolutionary algorithm for evolving neural networks using multi-objective optimization. Multi-objective optimization can be applied to a population of neural network genotypes and does in fact maintain diversity given the correct objectives. Diversity on the knowledge of each neural network has proven not to be any more effective than using test- or validation accuracies. Optimizing architectural objectives proves effective and gives diverse populations of neural networks where shapes, performance, knowledge and connectivity are diversified.

Multi-objective optimization is used for making decisions on whether or not a neural network performs well. This decision proves crucial through the Local-NAS system which only holds a single solution at a time. When not having a population to fall back on, these decisions can be fatal in the search process, making the search end up in a local maxima. A population is important for evolution-based neural architecture search.

The main contribution is using transfer learning for neural architecture search. This process requires a very direct representation to work. Transfer learning speeds up the search at the cost of diversity in the knowledge the population holds. It also provides more stability to the training process over time. All of these contributions requires more testing as each experiment was only ran once.

6.2 Future Work

This section will cover the future work discovered through writing this thesis. Themes include lower level integration with TensorFlow, parameters in the algorithm that needs tuning and steps towards generalized AutoML, starting with dataset augmentation. As stated in section 5.4.4, using augmentation on the dataset to obtain a greater larger training dataset is quite common. This should be applied to the experiments proposed here. This thesis is not comparable to the state-of-the-art neural architecture search algorithms without data augmentation. This applies for all of the experiments.

Parameter Optimization For EA-NAS

There are many parameters that can be experimented with for EA-NAS. The most important one is the probability distribution that decides what node types is chosen during mutation. The distribution is now heavily skewed towards convolutional- and dense nodes. Since the results clearly shows that dense nodes are filtered out while convolutional layers are preferred in the generated networks, all node types should have an equal probability for being selected. This might impact the speed of the algorithm significantly due to more bad choices being made which in turn means more neural networks which is known to be no better than their predecessor being evaluated for fitness. As stated in section 5.4.3, since the dense nodes are filtered out and remains mainly unused for the final solutions, experiments with removing them completely should be tried. With almost half the search space, new types of nodes/layers could be added like separable- and depth-wise convolutional nodes. These nodes/layers have seen great success in [24].

Combining Knowledge From Different Neural Networks

For combining the knowledge of multiple neural networks during evolution of network architectures, crossover is a requirement. When both trying to maintain a directed acyclic graph and keeping weights, a good crossover function was hard to achieve - as stated in chapter 3.2.3. Changing the structure of the genotypes to sequential graphs without any forks or joins, crossover would be easy to implement. A K-point crossover - as described in section2.1.1 - would be a good fit for both transferring knowledge and structure. For K-point crossover, one or more points are selected for crossing over. The knowledge within each sequence gathered from each parent genotype would have their weights transferred. Since the patterns tested with experiment 4 in this thesis were too small for transfer learning to have any effect, K-point crossover could help finding the correct amount of genes from each genotype to combine into the new successor genotype.

An alternative to this is to alter the existing pattern experiment to have much bigger patterns. This would yield the same results only with a directed acyclic graph representation. Crossover would however be hard to add in this representation which was one of the reasons the patterns were so small in the current experiment. Small patterns allowed for easier crossover in the Pattern-NAS implementation.

More Exploitation

Local search optimization should be explored as an extension to the EA-NAS system. Helping the EA algorithm to make smarter decisions may reduce time spent evaluating significantly. One idea to achieve this is to add a deeper integration with the TensorFlow [4] library. Deeper integration allows for more control over what actually happens inside the neural network. Reading out each layer's individual contribution to the prediction result might help with network pruning. If each layer's contribution can be measured to find the importance for different predictions, small contributions may be pruned first. This method is already tested in [18; 6; 16]. The TensorFlow library also has its own optimization toolkit that in version 2.0 includes a model pruning tool that prunes the network automatically [36].

Towards Truly Generalized AutoML

One of the key components to a generalized AutoML algorithm is interpreting the data. Data is gathered through exploration and is mostly unstructured. Having automatic interpretation of unstructured data combined with any of the neural architecture search systems reviewed in the literature review would be a huge step towards generalizing automatic machine learning. This is however an entirely different research field.

Bibliography

[Cif] Cifar-10 website. https://www.cs.toronto.edu/~kriz/cifar.html. Accessed: 2019-16-03.

[2] (2019). 8.2. convolution matrix.

[3] (2019). Internal covariate shift - machine learning glossary.

[4] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[5] Baker, B., Gupta, O., Naik, N., and Raskar, R. (2016). Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*.

[6] Cai, H., Zhu, L., and Han, S. (2018). Proxylessnas: Direct neural architecture search on target task and hardware. *CoRR*, abs/1812.00332.

[7] Chollet, F. et al. (2015). Keras. https://keras.io.

[8] Cortes, C., Gonzalvo, X., Kuznetsov, V., Mohri, M., and Yang, S. (2017). AdaNet: Adaptive structural learning of artificial neural networks. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 874–883, International Convention Centre, Sydney, Australia. PMLR.

[9] Dalcin, L. D., Paz, R. R., Kler, P. A., and Cosimo, A. (2011). Parallel distributed computing using python. *Advances in Water Resources*, 34(9):1124 – 1139. New Computational Methods and Software Tools.

[10] DalcÃn, L., Paz, R., and Storti, M. (2005). Mpi for python. *Journal of Parallel and Distributed Computing*, 65(9):1108 – 1115.

[11] DalcÃn, L., Paz, R., Storti, M., and DâElÃa, J. (2008). Mpi for python: Performance improvements and mpi-2 extensions. *Journal of Parallel and Distributed Computing*, 68(5):655 – 662.

[12] De Jong, K. A. (2016). *Evolutionary Computation: A Unified Approach.* MIT Press, Cambridge, MA, USA.

[13] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197.

[14] Deng, J., Dong, W., Socher, R., Li, L., and and (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255.

[15] Fernando, C., Banarse, D., Blundell, C., Zwols, Y., Ha, D., Rusu, A., Pritzel, A., and Wierstra, D. (2017). Pathnet: Evolution channels gradient descent in super neural networks.

[16] Frankle, J. and Carbin, M. (2019). The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations.*

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

[18] Gordon, A., Eban, E., Nachum, O., Chen, B., Wu, H., Yang, T.-J., and Choi, E. (2018). Morphnet: Fast simple resource-constrained structure learning of deep networks. pages 1586–1595.

[19] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

[20] Krizhevsky, A. (2012). Learning multiple layers of features from tiny images. *University of Toronto.*

[21] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521:436–44.

[22] Liu, B. (2018). A very brief and critical discussion on automl. *ArXiv e-prints.*

[23] Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L.-J., Fei-Fei, L., Yuille, A., Huang, J., and Murphy, K. (2017a). Progressive neural architecture search. *ArXiv e-prints*.

[24] Liu, H., Simonyan, K., Vinyals, O., Fernando, C., and Kavukcuoglu, K. (2017b). Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*.

[25] Liu, S. and Deng, W. (2015). Very deep convolutional neural network based image classification using small training sample size. In *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*, pages 730–734.

[26] Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Navruzyan, A., Duffy, N., and Hodjat, B. (2017). Evolving deep neural networks.

[27] Negrinho, R. and Gordon, G. J. (2018). Deeparchitect: Automatically designing and training deep architectures. *CoRR*, abs/1704.08792.

[Ng] Ng, C. Reintroducing PlaidML. `https://www.intel.ai/reintroducing-plaidml/#gs.9510bq`.

[Oates] Oates, B. J. *Researching information systems and computing.*

[30] Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. (2018). Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*.

[31] Rawal, A. and Miikkulainen, R. (2016). Evolving deep lstm-based memory networks using an information maximization objective. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 501–508, New York, NY, USA. ACM.

[32] Real, E., Aggarwal, A., Huang, Y., and V Le, Q. (2018). Regularized evolution for image classifier architecture search.

[33] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520.

[34] Stanley, K., Clune, J., Lehman, J., and Miikkulainen, R. (2019). Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1.

[35] Szegedy, C., Wei Liu, Yangqing Jia, Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9.

[36] Team, T. (2019). Tensorflow model optimization toolkitâââpruning api.

[37] Tegmark, M. (2017). *Life 3.0: Being Human in the Age of Artificial Intelligence.* Knopf Publishing Group.

[38] Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. *ACM International Conference Proceeding Series.*

[39] Zoph, B., Vasudevan, V., Shlens, J., and V. Le, Q. (2017). Learning transferable architectures for scalable image recognition.

Appendix

This appendix shows the final architectures landed on for each of the experiments conducted for this thesis. The architectures have been printed using the built-in Keras *model.plot* function. As these networks are large, details are not visible on print. The images are uploaded to the thesis GitHub repository for viewing and zooming on a computer: `https://github.com/MagnusPoppe/NAS/tree/master/experiments/best-found-architectures`.

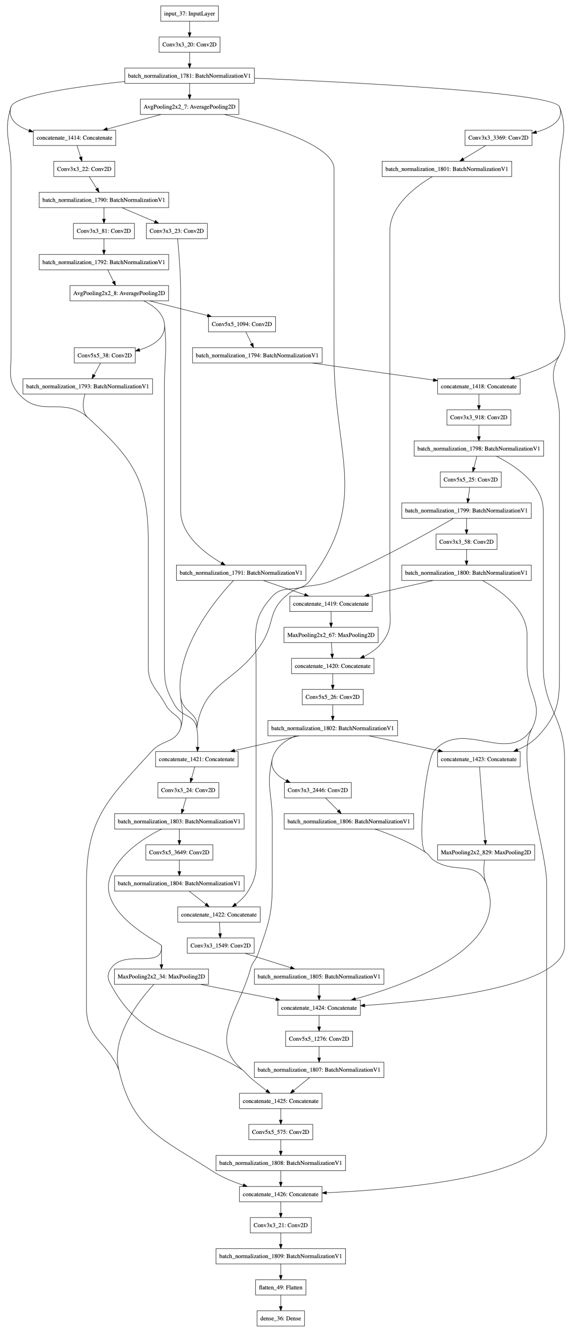

input_37: InputLayer
Conv3x3_20: Conv2D
batch_normalization_1781: BatchNormalizationV1
AvgPooling2x2_7: AveragePooling2D
concatenate_1414: Concatenate
Conv3x3_3369: Conv2D
Conv3x3_22: Conv2D
batch_normalization_1801: BatchNormalizationV1
batch_normalization_1790: BatchNormalizationV1
Conv3x3_81: Conv2D
Conv3x3_23: Conv2D
batch_normalization_1792: BatchNormalizationV1
AvgPooling2x2_8: AveragePooling2D
Conv5x5_1094: Conv2D
Conv5x5_38: Conv2D
batch_normalization_1794: BatchNormalizationV1
batch_normalization_1793: BatchNormalizationV1
concatenate_1418: Concatenate
Conv3x3_918: Conv2D
batch_normalization_1798: BatchNormalizationV1
Conv5x5_25: Conv2D
batch_normalization_1799: BatchNormalizationV1
Conv3x3_58: Conv2D
batch_normalization_1791: BatchNormalizationV1
batch_normalization_1800: BatchNormalizationV1
concatenate_1419: Concatenate
MaxPooling2x2_67: MaxPooling2D
concatenate_1420: Concatenate
Conv5x5_26: Conv2D
batch_normalization_1802: BatchNormalizationV1
concatenate_1421: Concatenate
concatenate_1423: Concatenate
Conv3x3_24: Conv2D
Conv3x3_2446: Conv2D
batch_normalization_1803: BatchNormalizationV1
batch_normalization_1806: BatchNormalizationV1
Conv5x5_3649: Conv2D
MaxPooling2x2_829: MaxPooling2D
batch_normalization_1804: BatchNormalizationV1
concatenate_1422: Concatenate
Conv3x3_1549: Conv2D
MaxPooling2x2_34: MaxPooling2D
batch_normalization_1805: BatchNormalizationV1
concatenate_1424: Concatenate
Conv5x5_1276: Conv2D
batch_normalization_1807: BatchNormalizationV1
concatenate_1425: Concatenate
Conv5x5_575: Conv2D
batch_normalization_1808: BatchNormalizationV1
concatenate_1426: Concatenate
Conv3x3_21: Conv2D
batch_normalization_1809: BatchNormalizationV1
flatten_49: Flatten
dense_36: Dense

Figure 6.1: The final architecture of the Baseline experiment 1.1

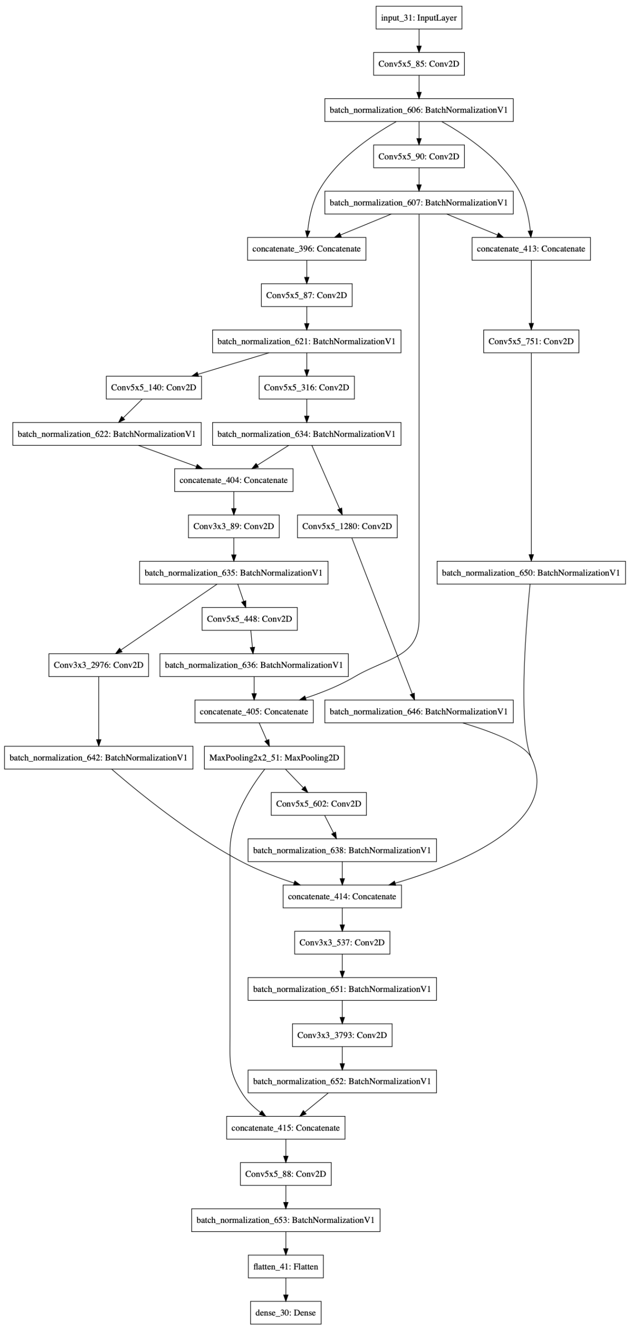

input_31: InputLayer
Conv5x5_85: Conv2D
batch_normalization_606: BatchNormalizationV1
Conv5x5_90: Conv2D
batch_normalization_607: BatchNormalizationV1
concatenate_396: Concatenate
concatenate_413: Concatenate
Conv5x5_87: Conv2D
batch_normalization_621: BatchNormalizationV1
Conv5x5_751: Conv2D
Conv5x5_140: Conv2D
Conv5x5_316: Conv2D
batch_normalization_622: BatchNormalizationV1
batch_normalization_634: BatchNormalizationV1
concatenate_404: Concatenate
Conv3x3_89: Conv2D
Conv5x5_1280: Conv2D
batch_normalization_635: BatchNormalizationV1
batch_normalization_650: BatchNormalizationV1
Conv5x5_448: Conv2D
Conv3x3_2976: Conv2D
batch_normalization_636: BatchNormalizationV1
concatenate_405: Concatenate
batch_normalization_646: BatchNormalizationV1
batch_normalization_642: BatchNormalizationV1
MaxPooling2x2_51: MaxPooling2D
Conv5x5_602: Conv2D
batch_normalization_638: BatchNormalizationV1
concatenate_414: Concatenate
Conv3x3_537: Conv2D
batch_normalization_651: BatchNormalizationV1
Conv3x3_3793: Conv2D
batch_normalization_652: BatchNormalizationV1
concatenate_415: Concatenate
Conv5x5_88: Conv2D
batch_normalization_653: BatchNormalizationV1
flatten_41: Flatten
dense_30: Dense

Figure 6.2: The final architecture of the Knowledge sort experiment 1.2

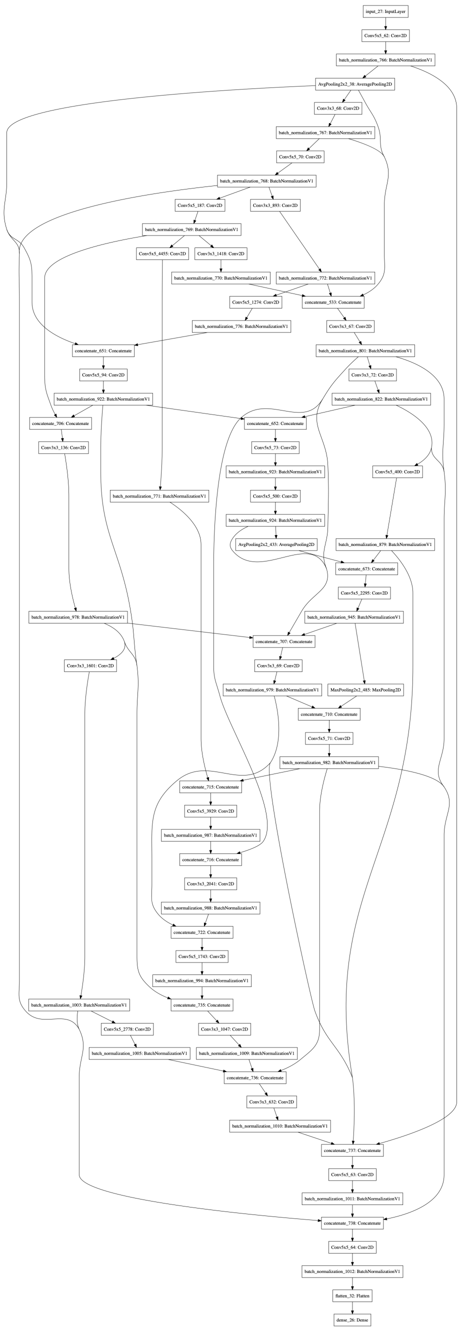


Figure 6.3: The final architecture of the Structure sort experiment 1.3

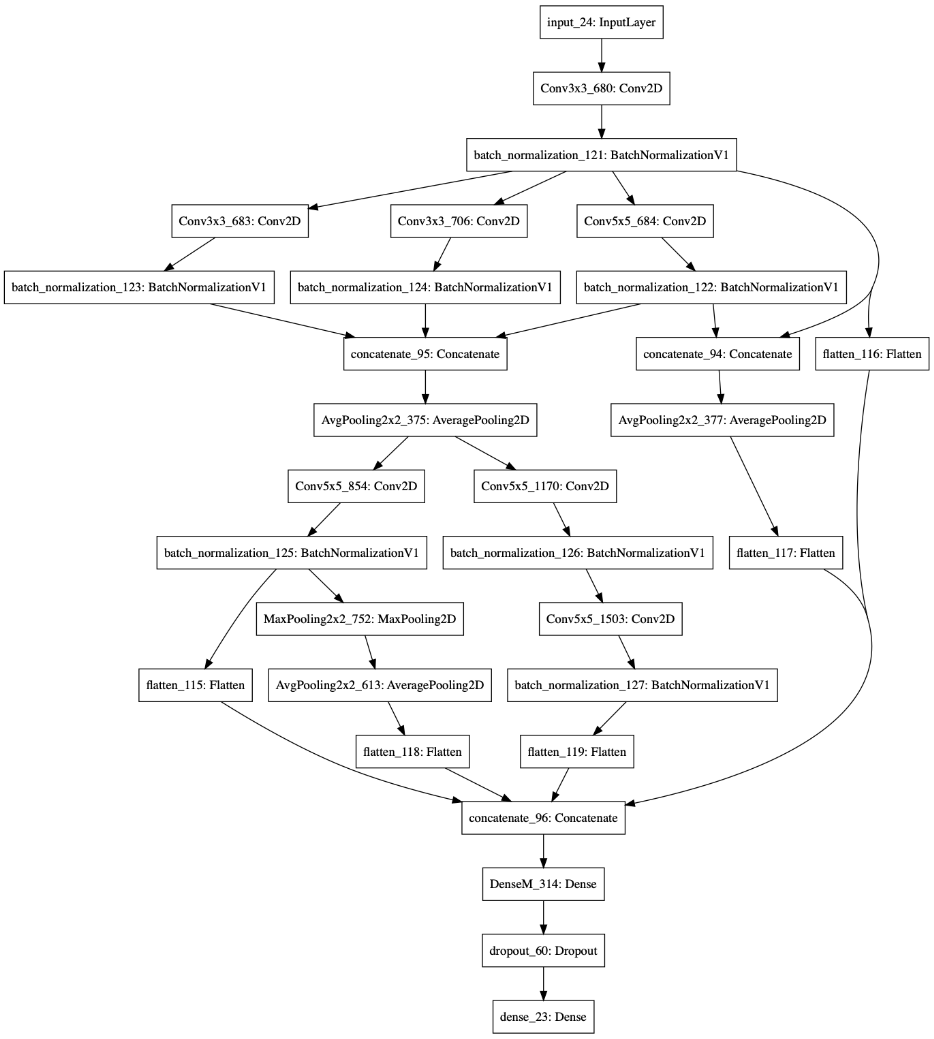

input_24: InputLayer
Conv3x3_680: Conv2D
batch_normalization_121: BatchNormalizationV1
Conv3x3_683: Conv2D
Conv3x3_706: Conv2D
Conv5x5_684: Conv2D
batch_normalization_123: BatchNormalizationV1
batch_normalization_124: BatchNormalizationV1
batch_normalization_122: BatchNormalizationV1
concatenate_95: Concatenate
concatenate_94: Concatenate
flatten_116: Flatten
AvgPooling2x2_375: AveragePooling2D
AvgPooling2x2_377: AveragePooling2D
Conv5x5_854: Conv2D
Conv5x5_1170: Conv2D
batch_normalization_125: BatchNormalizationV1
batch_normalization_126: BatchNormalizationV1
flatten_117: Flatten
MaxPooling2x2_752: MaxPooling2D
Conv5x5_1503: Conv2D
flatten_115: Flatten
AvgPooling2x2_613: AveragePooling2D
batch_normalization_127: BatchNormalizationV1
flatten_118: Flatten
flatten_119: Flatten
concatenate_96: Concatenate
DenseM_314: Dense
dropout_60: Dropout
dense_23: Dense

Figure 6.4: The final architecture of the Local experiment 2.

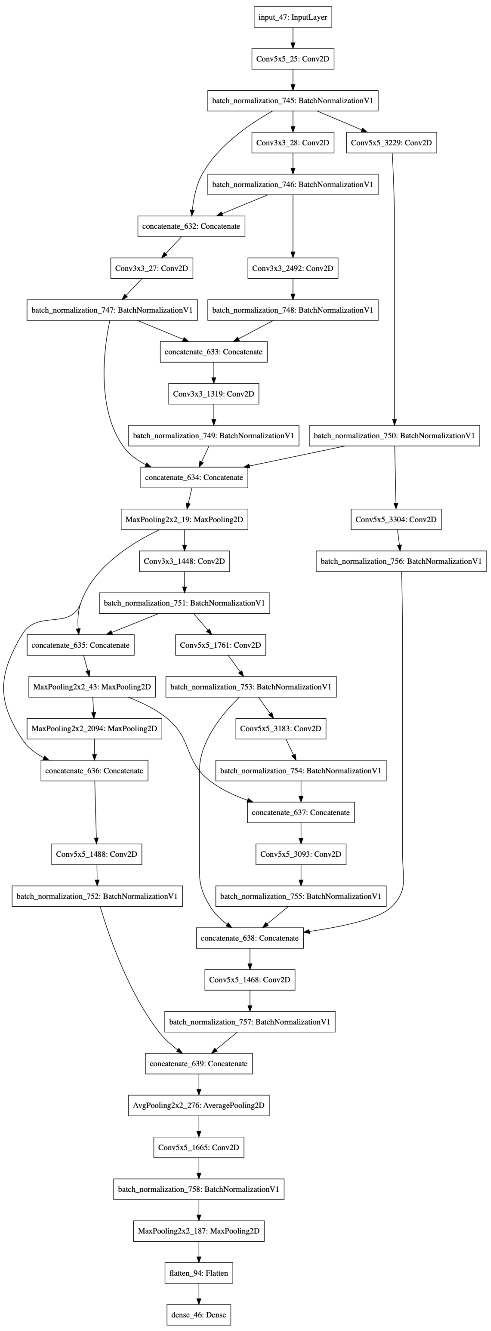

input_47: InputLayer
Conv5x5_25: Conv2D
batch_normalization_745: BatchNormalizationV1
Conv3x3_28: Conv2D
Conv5x5_3229: Conv2D
batch_normalization_746: BatchNormalizationV1
concatenate_632: Concatenate
Conv3x3_27: Conv2D
Conv3x3_2492: Conv2D
batch_normalization_747: BatchNormalizationV1
batch_normalization_748: BatchNormalizationV1
concatenate_633: Concatenate
Conv3x3_1319: Conv2D
batch_normalization_749: BatchNormalizationV1
batch_normalization_750: BatchNormalizationV1
concatenate_634: Concatenate
MaxPooling2x2_19: MaxPooling2D
Conv5x5_3304: Conv2D
Conv3x3_1448: Conv2D
batch_normalization_756: BatchNormalizationV1
batch_normalization_751: BatchNormalizationV1
concatenate_635: Concatenate
Conv5x5_1761: Conv2D
MaxPooling2x2_43: MaxPooling2D
batch_normalization_753: BatchNormalizationV1
MaxPooling2x2_2094: MaxPooling2D
Conv5x5_3183: Conv2D
concatenate_636: Concatenate
batch_normalization_754: BatchNormalizationV1
concatenate_637: Concatenate
Conv5x5_1488: Conv2D
Conv5x5_3093: Conv2D
batch_normalization_752: BatchNormalizationV1
batch_normalization_755: BatchNormalizationV1
concatenate_638: Concatenate
Conv5x5_1468: Conv2D
batch_normalization_757: BatchNormalizationV1
concatenate_639: Concatenate
AvgPooling2x2_276: AveragePooling2D
Conv5x5_1665: Conv2D
batch_normalization_758: BatchNormalizationV1
MaxPooling2x2_187: MaxPooling2D
flatten_94: Flatten
dense_46: Dense

Figure 6.5: The final architecture of the Baseline w/o TL experiment 3.

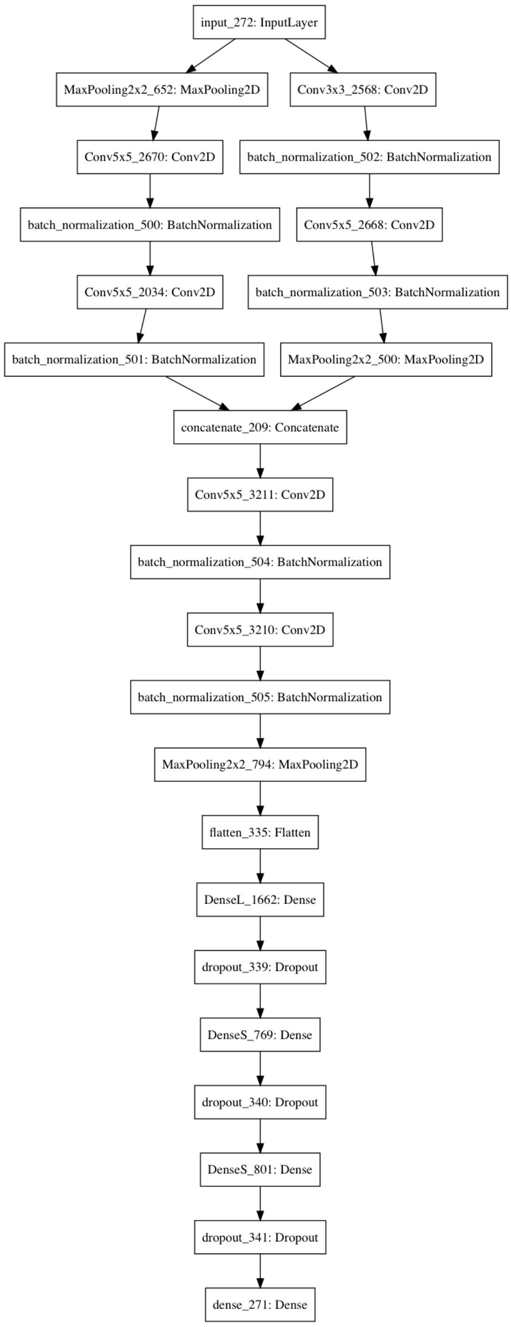

input_272: InputLayer
MaxPooling2x2_652: MaxPooling2D
Conv3x3_2568: Conv2D
Conv5x5_2670: Conv2D
batch_normalization_502: BatchNormalization
batch_normalization_500: BatchNormalization
Conv5x5_2668: Conv2D
Conv5x5_2034: Conv2D
batch_normalization_503: BatchNormalization
batch_normalization_501: BatchNormalization
MaxPooling2x2_500: MaxPooling2D
concatenate_209: Concatenate
Conv5x5_3211: Conv2D
batch_normalization_504: BatchNormalization
Conv5x5_3210: Conv2D
batch_normalization_505: BatchNormalization
MaxPooling2x2_794: MaxPooling2D
flatten_335: Flatten
DenseL_1662: Dense
dropout_339: Dropout
DenseS_769: Dense
dropout_340: Dropout
DenseS_801: Dense
dropout_341: Dropout
dense_271: Dense

Figure 6.6: The final architecture of the Pattern experiment 4.

Magnus Poppe Wang

Evolving Knowledge And Structure Through Evolution-based Neural Architecture Search

NTNU
Norwegian University of
Science and Technology