An Empirical Study on the Efficacy of Evolutionary Algorithms for Automated Neural Architecture Search

Andrew D. Cuccinello
Rowan University

AN EMPIRICAL STUDY ON THE EFFICACY OF EVOLUTIONARY ALGORITHMS FOR AUTOMATED NEURAL ARCHITECTURE SEARCH

by
Andrew D. Cuccinello

A Thesis

Submitted to the
Department of Computer Science
College of Science and Mathematics
In partial fulfillment of the requirement
For the degree of
Master of Science in Computer Science
at
Rowan University
December 16, 2021

Acknowledgements

We thank the Rowan University Department of Computer Science for their support and the GAANN fellowship funded by DOE award-P200A190033 which also includes funding for computing resources to perform the computational experiments completed in the thesis.

Abstract

Andrew D. Cuccinello
AN EMPIRICAL STUDY ON THE EFFICACY OF EVOLUTIONARY ALGORITHMS
FOR AUTOMATED NEURAL ARCHITECTURE SEARCH
2020-2021
Shen-Shyang Ho, Ph.D.
Master of Science in Computer Science

The configuration and architecture design of neural networks is a time consuming process that has been shown to provide significant training speed and prediction improvements. Traditionally, this process is done manually, but this requires a large amount of expert knowledge and significant investment of labor. As a result it is beneficial to have automated ways to optimize model architectures. In this thesis, we study the use of evolutionary algorithm for neural architecture search (NAS). Moreover, we investigate the effect of integrating evolutionary NAS into deep reinforcement learning to learn control policy for ATARI game playing. Empirical classification results on the NASBench image dataset using different population selection and drop methods for the evolutionary NAS are presented to validate the usefulness of simple evolutionary algorithms in optimizing neural architectures, showing that even basic evolutionary algorithms are a well performing and easy to use approach. We also show the feasibility of using evolutionary NAS to extract good features that can improve the pong game playing score when the computational resource is limited.

Table of Contents

Table of Contents (Continued)

Table of Contents (Continued)

List of Figures

List of Tables

Chapter 1
Introduction

Neural networks must be carefully configured to maximize performance, but this is often a difficult and time consuming task. Not only does it rely on an abundance of knowledge of the domain in question, but there are often no good ways of predicting the performance of an architecture without training the model, a process that can take significant computational power, which equates to significant financial cost.

As such, it is desirable to have an automatable way of finding an architecture for a neural network, and ideally an optimal or near optimal one at that. This is the underlying motivation behind the area of research known as neural architecture search. By defining methodology to efficiently search through architectures, we can discover architectures that can potentially surpass human tuned models. Current research has already demonstrated models using architectures discovered by these techniques can surpass manually engineered architectures on a number of tasks such as semantic segmentation, object detection, and most commonly image classification [1, 2].

This thesis therefore seeks to survey and taxonomize existing research in the area using already established terminology, as well as demonstrate the potential for these techniques by illustrating that even simple methodologies can discover architectures that can perform at or above baseline levels through way of example. In this thesis, we study the use of evolutionary algorithm for neural architecture search (NAS). Moreover, we investigate the effect of integrating evolutionary NAS into deep reinforcement learning to learn control policy for ATARI game playing. Empirical classification results on the NASBench image dataset using different population selection and drop methods for the evolutionary NAS are presented to validate the usefulness of simple evolutionary algorithms in optimizing neural architectures, showing that even basic evolutionary algorithms are a well performing and easy to use approach. We also show the feasibility of using evolutionary NAS to extract good features that can improve the pong game playing score when the computational resource is limited.

This work has six chapters, including this introduction. In the proceeding chapter 2, we review different types of methodologies and approaches to build a taxonomy of this problem. In chapter 3 we provide background on evolutionary algorithms which are used in this work and describe how they are integrated into the deep reinforcement learning problem of ATARI Pong. In chapter 4 we describe the tasks of the two related experiments performed — (i) using NAS to develop an image classification architecture and (ii) discovering a feature extraction architecture for improving performance of an agent on ATARI Pong. In chapter 5 we analyze the results of the preceding chapter and compare the results of various trial runs. Finally in chapter 6 we conclude the work and describe one interpretation for the future of NAS as a field.

Chapter 2

Literature Review

The area of research dedicated to automated engineering of neural network architectures is generally referred to as either neural architecture search, or more generally as architecture engineering. The latter often carries a connotation of handmade architectures, so in this work we use the term neural architecture search.

The main area of research in this field is the reduction of computational cost for the problem. With endless resources, the problem of finding an optimal architecture is trivially solvable by brute force. However, the difficulty comes from the fact that not even a large corporation has truly infinite resources, and we would like to use the model sometime before the sun swallows the earth. The trick then, is to reduce the time and cost enough such that the problem becomes solvable and cost efficient.

While neural architecture search is an area of current research, existing methodology can have one-to-three components which each target the problem of computational time in a different way. We use the terminology coined by Elsken et. al. [3] in this thesis. In particular, they categorize neural architecture search methods into three groups, namely: search space, search strategy, and performance estimation.

2.1 Search Space

Methods that target the search space as a way to reduce cost use carefully formulated sets of possible parameters to limit the size of the search space. These tend to incorporate some knowledge of architecture that leads to positive correlation with performance. In theory, if this is done carefully enough and with enough sophistication, a search space can be created that only contains models with a high probability of competitive performance.

The largest issue with this approach is that it introduces a human bias, which may prevent the discovery of truly novel architectures [3].

2.2 Search Strategy

When a method attempts to optimize the strategy in which the search space is explored, it falls into this category. This can be as simple as altering the exploration/exploitation tradeoff. In practice however, many search spaces are unbounded and require more sophisticated methods of exploration.

The experiment we performed in chapter 4 and chapter 5, an evolutionary algorithm approach, targets this as a primary method of optimization [3].

2.3 Performance Estimation

Since training a model is almost always the primary source of computation cost, rather than searching the architecture space itself, this category offers the most promise for efficiency improvements by estimating the performance of a model without fully training it.

The largest issue with this approach is that it is not generally understood why some architectures work better than others, so estimations of performance have the potential to be wildly inaccurate. The common “black box” terminology remains true here [3].

2.4 Types of Search Spaces

In the case of neural architectures, the search space is to the possible operations and how they may be interconnected. Searching different spaces can have different benefits and drawbacks, and a carefully designed search space can reduce training time by ensuring that only highly performing models may be visited.

2.4.1 Sequential Layer-Wise Operations

There are a number of ways to design the search space for neural network architectures, but the most readily apparent is with a list of sequential operations. Zoph et. al. [4] and Baker et. al. [5] are some of the earliest to make use of this type of space. These spaces often require some additional validation rules to make sure generated architecture is valid.

2.4.2 Cell-Based Representation

This search space is made to emulate architectures like ResNet [3], which perform well on vision tasks. The goal of this space is then to design a cell that can be repeated a number of times, with more or less repetitions depending on the desired model size [2].

2.4.3 Hierarchical Structures

Some algorithms take advantage of a hierarchical structure to optimize the size of the search space. These algorithms construct small graphs out of basic operations then reuse the graphs recursively to form larger graphs. Such hierarchical structures allow the algorithms to avoid the cost of exploring minute changes to large models consisting of dozens if not hundreds of basic operations [3].

2.4.4 Memory-Bank Representation

Algorithms such as SMASH create a slightly different representation of a neural network. They reformulate the structure of a neural network as a set of blocks of memory that can support read or write operations. The layers then read, compute, and write from subsets of these blocks. In a sequential model one block would get read from and written to over and over. More complex formulations are possible [3].

SMASH performs this reformulation for the purpose of easy encoding for the conditioning vector of a HyperNet, which is then used to predict model weights as a proxy for actual training [6].

2.5 Types of Search Strategies

2.5.1 Random Search

These algorithms use random sampling to find the best architectures. An offshoot of this category is methods like grid search. These algorithms are very naive but have been shown to perform well in hyperparameter optimization, which is a closely related field [7].

2.5.2 Reinforcement Learning

An agent is trained as a controller that proposes models which are then evaluated. This boils down to a standard reinforcement learning task, where the action space is the choices for parts of the architecture, the reward is the accuracy of the generated model, and any loss is calculated as normal [4].

2.5.3 Evolutionary Algorithms

An initial population of individuals is created, usually randomly. Each individual represents a single architecture. After evaluating this population's performance, individuals can generate children through mutation and mating, much like animals in nature. Generally only the strongest performers are allowed to reproduce, but some researchers found other methods beneficial such as eliminating older individuals from the population [1]. As the population reproduces and mutates, survival of the fittest ensures an increasing performance.

2.5.4 Progressive Decision Processes

These types of algorithms start by generating very simple architectures and progressively increasing complexity. Some algorithms also learn a surrogate model to predict the performance of a model without the need to train it. This surrogate model allows a sort of pathfinding through the search space, much in the same way that one may find a path through a directed graph using an algorithm like A* (or other heuristic based pathfinding) [8, 9].

2.5.5 Gradient Descent

Methods using gradient descent usually combine the learning of weights and architecture into a single problem; this is often referred to as "one-shot" learning. Generally speaking, these models share weights by treating children architectures of a parent architecture as subgraphs of a supergraph [6, 10, 9]. We note these methods typically offer exceptional performance at the cost of substantially increased setup effort.

2.6 Types of Evaluation Strategies

Evaluation of performance is almost always the vast majority of computational cost. Dozens of hours of GPU time may be taken training a single model, and to find a truly novel architecture thousands of architectures must be evaluated. As such, optimization of this task offers the largest potential performance improvement.

2.6.1 Training from Scratch

This is the most readily available and apparent methodology, but also the least efficient. As stated, training a single model can take dozens of GPU hours, so training a model from nothing to convergence, while it does provide the truest estimation of performance, poses significant computational barriers. A few strategies are available to reduce this cost, most notably partial training.

2.6.2 Proxy Tasks

Rather than training on a full and complete problem, such as a complete data set, we can train a model on a proxy to our desired task that is smaller in scope. The easiest example is using image datasets of two different sizes, the target dataset and the proxy dataset can differ in size by orders of magnitude but still provide a representative approximation of the task in question. Training time on the proxy task is therefore reduced accordingly. This is the methodology used by Zoph et al. [2] to transfer learned architecture from a small dataset to a large dataset.

There are a number of other more sophisticated methods that fall into the proxy task category. A proxy task can also be defined as a similar but more efficient task performed to evaluate architecture performance. Baker et al. [11] predict the learning curve of training to enable earlier stopping, resulting in a up to 6x speed up.

2.6.3 Parameter Sharing

One of the most complex but interesting approaches is to create a dependency between models and then discover ways to reuse weights between dependent models. Some algorithms such as Efficient Architecture Search (EAS) by Cai et al. [12] take the approach of incrementally growing the network so as to reuse the previous model's weights. Future models then only need a small amount of training.

Similarly, Efficient NAS (ENAS) treats the problem as a large computational graph search, and reframes the solution as finding the optimal subgraph of this supergraph. This allows ENAS to almost ubiquitously share weights and results in a speedup of 1000x over NAS [10].

2.6.4 Predictive

This method of evaluation shows potential as a significant source of performance improvement. The goal is to predict the final weights of the architecture based on it's configuration, without the computational cost of from-scratch training. This is often done with a HyperNet [13], and the generated weights are then directly validated. Algorithms such as SMASH have shown promising results by learning a mapping from a binary encoded representation of the network to the weight space [6]. The disadvantage of this method is its inability to generate novel architectures; rather, it operates by exploring a pre-designed space.

2.6.5 One-Shot

A one-shot approach takes weight sharing and combines architecture generation. These algorithms effectively train the model during its generation. Then by zeroing out certain connections of the model, we can treat child models as subgraphs of a supergraph [9].

2.7 Representative Architecture Search Approaches

To demonstrate the wide variety of approaches for automated engineering of neural network architectures, we highlight some representative architecture search algorithms.

2.7.1 NeuroEvolution of Augmenting Topologies (NEAT)

One of the earliest algorithms for generating architectures is NeuroEvolution of Augmenting Topologies (NEAT) [14]. NEAT was introduced in 2002 and is impressively ahead of its time. Existing algorithms far surpass it in performance, both in terms of speed and accuracy of final generated models. However, as a forerunner in the area it deserves discussion.

NEAT is novel in its inclusion in this thesis as it is the only algorithm we discuss that directly evolves the weights of the network in addition to the network topology. While this does create less of a need to combine multiple works, a task that can add difficulty to experimental setup, it also means it is hard to use in combination with other algorithms which improve upon its performance, as you must make modifications to the algorithm.

There are several key features of NEAT that lead to its massive success with thousands of citations. It is often cited as one of the first successful algorithms in the field of NAS, although its use is now limited as there are more efficient ways to evolve model weights, and it requires a very specific data structure to use.

We summarized three key implementation features [14]:

1. **Speciation:** NEAT subdivides a population into different species, which allows individuals to compete within their own niches rather than with the entire population. This is done to protect innovation, and allow individuals time to optimize their structure, which is important as NEAT also learns weights in addition to architecture. NEAT places individuals into groups based on how similar their genes are. As two genomes grow more and more disjoint, it indicates less and less shared evolutionary history.

2. **Minimal Initial Populations:** NEAT starts the initial population with the smallest possible architectures, in order to force any pieces of architecture created to justify their existence. The designers note that with an initial population, it is quite easy to have nonsensical or unconnected architecture, and forcing additional elements to be added during the run prevents that in a clear and non pernicious way.

3. **Historical Markers** To enable crossover between architectures, NEAT tracks where a gene (e.g. a specific connection) comes from by assigning the gene a unique identifier. Subsequent architectures then inherit identification, and architectures that share identifying genes are able to perform crossover safely. Any genes that are not shared are taken from the most fit parent.

2.7.2 Neural Architecture Search (NAS)

The algorithm arguably responsible for the explosion of popularity of the Neural Architecture Search field is the titular Neural Architecture Search (NAS). The algorithm is fairly straightforward, it uses reinforcement learning to train a controller agent that predicts the architecture of a network. Then, the predicted architecture is trained and the resulting accuracy is used as a reward for the controller network. Over time, this causes updates to the controller to improve the quality of the predicted model parameters [4].

2.7.3 One-Shot Architecture Search

The goal of One-Shot Architecture Search is to allow the evaluation of more models by avoiding repeated training. The algorithm accomplishes this by training one very large model that contains every possible operation that can be performed within the network. At evaluation time, some of these operations are then zeroed out, and the resulting model is used to evaluate the equivalent model without any additional training requirements.

Bender et al. [15] observed that the accuracies of models evaluated using this technique decreased by 5-10% for the best performing models, but as much as 60% for worse performance models. They hypothesized that the one shot architecture is in reality learning which operations are the most important to model accuracy, rather than individual weights.

2.7.4 Differentiable Architecture Search (DARTS)

A promising approach to solve the search problem is Differentiable ARchiTecture Search (DARTS), which changes the formulation of the problem to make the architecture space to be differentiable, which in turn allows the use of gradient descent when performing the search.

DARTS searches for a computation cell, following the examples of Zoph et al [2], Real et al. [1], and Liu et al.[8], who have all shown it has very competitive performance on CIFAR-10. In order to allow the problem to be differentiable, the search space must be made continuous. First a zero operation is added, so nodes may be unconnected. DARTS then relaxes the categorical choice of a particular operation to a softmax over all possible operations. This turns the problem then into a probabilistic one, in which the choice of operation can be made by selecting the most likely operation at each connection.

The performance of DARTS is quite impressive, reducing the benchmark from thousands of GPU hours like that of AmoebaNet-A [1] to just a few GPU days.

2.7.5 Progressive Neural Architecture Search (PNAS)

Progressive Neural Architecture Search (PNAS) [8] reduces the computational complexity of searching for an architecture by performing the process as a series of progressive decisions. The algorithm starts by searching smaller models first, progressively making decisions as to how to evolve and expand the model. During this process it learns a heuristic to guide the search.

Each decision then limits the search space in a way that reduces the number of models that must be explored in the future.

Chapter 3
Methodology

In this chapter, we first provide background on evolutionary algorithm. Then, we describe how the evolutionary algorithm is used for neural architecture search. Finally, we describe in detail how the evolutionary algorithm is integrated into deep reinforcement learning for ATARI game playing.

3.1 Background: Evolutionary Algorithms

Evolutionary algorithms are a class of performance optimizing algorithms which mimic the way natural selection improves the ability of a population to survive in an environment, allowing only the most fit individuals to reproduce. Instead of survival in the jungle however, evolutionary algorithms target a problem and we say they select for fitness in solving the problem. If natural selection itself were an evolutionary algorithm, we would say the fitness that was being optimized for was survival. In other cases, it's the ability to solve a particular problem, like playing Pong or classifying images.

First, we define some key technical concepts for evolutionary algorithms used throughout the thesis [16, 17].

- **Fitness:** A score of how well a task was performed. Selection of the fitness function is one of the key ways in which a population's evolution may be guided towards desired results.
- **Individual:** A single entity which may complete a task or be used to complete a task.
- **Population:** A group of X unique individuals.
- **Generation:** A set of individuals which exist at the same time between iterations of

reproductions or mutations.

- **Mutation (M):** The morphing of an individuals genome to produce a different individual with an altered genome.
- **Crossover (C):** The mixing of two individuals genomes to produce a third unique individual.
- **Genome:** The set of parameters which can be combined to produce a unique individual. For a CNN, this may be the size of the hidden layers, their stride, what type of activation function(s), etc.

The basic structure for an evolutionary algorithm (see Figure 1) follows the same skeletal structure with maximum number of generations, *G*, with variations coming from how fitness is evaluated or how individuals are selected to be mutated or removed from the population.

Figure 1

Pseudocode for the Basic Structure of an Evolutionary Algorithm

```
population = []
Initialize population with X randomly generated architectures
Evaluate architectures in initial population to obtain fitness
for generation 1 ... G
    drop D individuals in population using some criteria
    while population not full
        mutate the individual with a probability of M
        crossover the individual with a probability of C
        add new individuals to the population
        evaluate individuals in the population
```

One notes that for the pseudocode in Figure 1, there are a number of points where one line changes can have severe behavioral effects, i.e., change in the algorithm's learning behavior [16]. In particular, we highlight two significant algorithm characteristics: fitness function and candidate selection.

- **Fitness function:** By changing how we evaluate fitness, we can have the evolutionary algorithm evolve different properties. By returning higher fitness values for quicker completing models for example, we can evolve models which compute faster rather than more accurately. The number of possible fitness score calculations is really only limited to values which can be computed, which provides incredible versatility for potentially minimal effort.

  Additionally, it should be noted that dual-optimization is a rather trivial task with evolutionary algorithms by using weighted fitness scores. By calculating a fitness score using both accuracy and inverse calculation time, we can evolve models which have better accuracy but also compute faster.

- **Candidate Selection:** We may choose how we select individual candidates to be mutated or interbred. The methodology we do so with changes how much exploration and exploitation we do. We can also change how we remove individuals from future consideration, which further targets the exploration and exploitation trade-off. The number of individuals and the size of the population can also have effects on the evolution and its performance.

3.2 Evolutionary Neural Architecture Search

Figure 2 shows the general structure of an evolutionary neural search algorithm [18]. We note that this diagram is nearly identical to the pseudocode presented in Figure 1. As mentioned, all evolutionary algorithms share the same bones and this remains true in the evolutionary neural architecture search case, where only method of fitness evaluation changes.

Figure 2

A Conventional EA Based Neural Architecture Search [18]

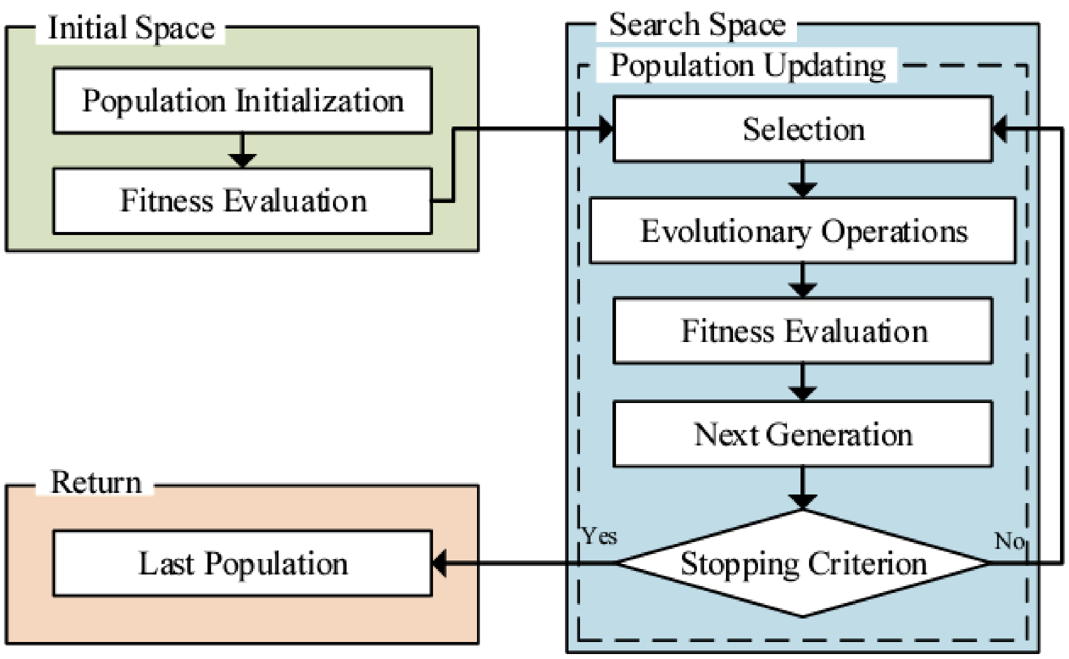


Figure 3

Overview of Reinforcement Learning for Atari Game Playing Using EA Discovered Feature Extraction Models

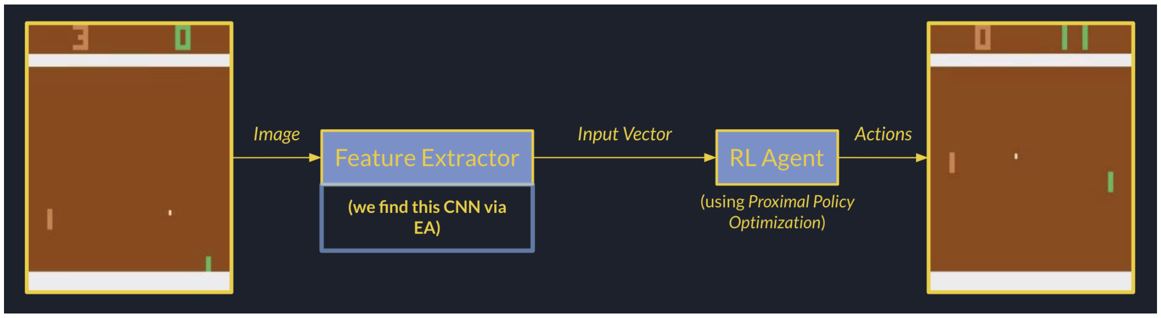


Image
Feature Extractor
(we find this CNN via EA)
Input Vector
RL Agent
(using Proximal Policy Optimization)
Actions

3.3 Evolutionary Neural Architecture Search for Feature Extraction

Figure 3 shows an overview of the pipeline used to perform the Atari experiment. The goal is to discovered a model to perform feature extraction — that is, turning the raw pixel data from the game into an input vector of weights for the game playing agent — for the game Pong. This model is discovered using an evolutionary algorithm which shares pseudocode with the above Figure 1. For posterity, we reproduce the pseudocode below in Figure 4 with modifications for clarity.

Figure 4

Pseudocode for the Atari Experiment

```
Population = []
Initialize population with X randomly generated architectures
For each model in the population
    Train feature extractor and game agent concurrently for
        up to 3 hours, or until they reach a score of -15
While total time spent < 30 days
    Drop the 5 worst performing individuals from the population
    While population not full
        Select an individual using tournament selection with 20%
            tournament size
        Mutate the individual
        Add the new individual to the population
        Evaluate the new individual individual
```

In training the feature extraction and game playing agent, we use a reinforcement learning algorithm called Proximal Policy Optimization (PPO). For a complete mathematical description of the algorithm see [19]. A description of how it is used in the Atari pipeline is shown in Figure 5. We note that it is not used for anything except the evaluation of the fitness of a model. The evolutionary algorithm and PPO never interact; the evolutionary algorithm proposes models to be trained, which are then trained using PPO.

This simplistic approach proves sufficient, as shown later in this work. We do note however, that many improvements could be made to make the process more efficient. We note some of these improvements in chapter 6.

Figure 5

Reinforcement Methodology for the Atari Experiment

```
evaluate(candidate):
    If candidate already evaluated, return prior result
    Initialize game playing agent with candidate as the feature
        extractor
    While score < -15 and time spent < 3 hours
        extract features from game pixel data using candidate
        predict next action using extract features in game agent
        update game agent and feature extractor using PPO
    Perform 10 trials of the final model
    Store results of trials as mean score and standard deviation
    Return mean score of 10 trials
```

Chapter 4
Application Scenarios and Experimental Setups

In this chapter, we describe the tasks of (i) using NAS to improve the performance of image classification and (ii) improving control policy learned by reinforcement learning of ATARI game playing via NAS-improved feature extraction from raw image inputs. We describe the data used for the two tasks and our experimental designs and setups to investigate the two tasks.

4.1 Image Classification

Image classification is a common and important task in many modern automated applications such as medical diagnosis and autonomous driving. It is critical to have high accuracy for such applications, especially in the provided two examples where patient treatment protocols may be influenced based on a false diagnosis, or a falsely identified obstacle may cause a car to crash injuring or killing passengers. Many approaches have been proposed to tackle these example problems, let alone the many other applications of image classification. See the references herein for a few examples of medical diagnosis [20] and autonomous driving [21]. We note autonomous driving is a far more sophisticated problem than simple image classification — involving many types of problems such as decision making and risk assessment — but necessitates image classification use in its solution.

4.1.1 NASBench Image Dataset

NASBench dataset [22] contains 423k precomputed models on the CIFAR-10 image dataset [23], making the architectural search space of the dataset finite but large. It is limited to small feedforward architectures referred to as cells, as discussed earlier in chapter 2. Final models then have a complete architecture as described below:

```
(input) 3x3 convolution, 128 output channels
[3x cell from search space]
Width/height are halved w/ max-pooling, channel count doubles
[3x cell from search space]
Width/height are halved w/ max-pooling, channel count doubles
[3x cell from search space]
Width/height are halved w/ max-pooling, channel count doubles
Global average pooling
(output) Dense softmax
```

We again note only the cell architecture is searched for. This stacked cell architecture is a common pattern in NAS spaces and hand-made classifiers [3, 9, 24]. The entire space consists of cell architectures which are directed acyclic graphs of V nodes, where each node has one of L operations. The possible set of operations is 3x3 conv, 1x1 conv, 3x3 max-pool and V is limited to $<= 7$. There can be no more than 9 edges. As long as these constraints are maintained, every possible combination of cell architectures is present in the dataset, which allows us to fully experiment without the need to train the models, as they are precomputed for us.

This dataset allows us to experiment and compare various techniques, specifically with evolutionary algorithms, without having to train 423,624 models, which would cost an amount of money which is infeasible to us.

Table 1

NASBench 101 Dataset Statistics - Full Training

	Minimum	**Maximum**	**Median**	**Mean**	**Std. Dev.**
Trainable Parameters	227,274	49,979,274	5,356,682	8,459,762	8,558,503
Training Time	284.22	5,535.54	1,560.50	1,932.40	918.82
Train Accuracy	10.02%	100.00%	100.00%	99.19%	7.38%
Validation Accuracy	9.47%	95.11%	91.25%	90.25%	6.99%
Test Accuracy	9.98%	94.55%	90.65%	89.69%	6.91%

Note. Table obtained through statistical analysis of NASBench dataset.

We first note some statistics about the dataset in Table 1. The best possible validation accuracy in this dataset is around 95%, so that is our target when performing this experiment.

4.1.2 Experimental Design and Setup

In generating architectures, we note we do not use crossover during evolution. Crossover is notoriously difficult to implement on architectural problems such as this, due to the concerns of generalization of genes and their compatibility. We note that Stanley and Miikkulainen [14] call these excess or disjoint genes. NEAT solves this problem through historical marking, described in the survey section; however, fundamentally the problem is that it is hard to correctly choose which genes should be mixed with which when there are excess or missing genes in a genome [14]. Since only performing crossover between identical architecture models will allow only very rare occurrences of crossover and may introduce unknown bias, we elect not to include crossover at all.

Additionally, we use half training during our experiment execution, but in reporting final results compare the completed training measures. This ideally helps us mimic the real world performance, where we would not expect to fully train models during evolution due to its computational infeasibility.

4.2 Learning Control Policy for ATARI Game Playing

Mnih et al. [25] first used a convolutional neural network to extract important features from image pixels obtained during playing of ATARI games. These extracted features are used as input to a variant of Q-learning [26] to estimate future reward during the learning process of control policy used for playing ATARI games. Schulman et al. [19] replaces Q-learning with proximal policy optimization (PPO) with a better sample complexity compared to trust region policy optimization (TRPO) [27, 19], while maintaining properties of TRPO.

4.2.1 CNN-Driven Feature Extraction for Reinforcement Learning

Mnih et. al. [25, 28] proposed a CNN architecture which takes in state representation as input to the neural network and output a vector with elements corresponding to the Q-values of the possible actions for the input state. This input states is represented by the features extracted from the convolutional layers. Hence, the architecture provides value function estimation of future rewards given the input states. The CNN computes the Q-values with only a single forward pass through the network.

The architecture proposed by Mnih et. al. [25, 28] is significant in that it is the first architecture to our knowledge proposed that is generally applicable to many vision-based feature extraction tasks on Atari 2600 games, and is often a standard configuration used during first-pass research or included as a default configuration in libraries. This first deep learning architecture are often cited as initiating the boom of interest in deep reinforcement learning.

We select this problem because of its personal interest to us, but also because it is a comparatively simple space without many complexities upon which the problem of NAS can be explored. We choose a single game, Pong, to run our experiments on, again acknowledging the incredible computational cost of model training as part of our performance evaluation. More complex games, let alone multiple games, are simply beyond our computational resources.

The goal of this experiment is very modest, simply to investigate whether evolutionary NAS can be used efficiently and effectively to find an architecture that outperforms the architecture proposed by Mnih et. al [28] at Pong.

4.2.2 Experimental Design and Setup for the Game of Pong

As discussed, the biggest problem with NAS is the incredible computational complexity of searching models. In this problem we spent a total of 775 GPU hours (approximately 1 month) and had mild success. In this time, we trained a total of 369 models for an average of approximately 2 hours each. Exact training time for an individual model varied, with a standard deviation of approximately 45 minutes, but no model was allowed to train for more than 3 hours. Additionally, training was stopped when a score threshold of -10 was reached, which is approximately 25% of the way through training. The maximum score threshold would be +21, which would be considered a perfect game.

We count training speed as approximately 30% of total fitness, while score is counted as the other 70%. Training time is scored using an inverse logarithmic function; e.g. as training time increases the portion of score training time is responsible for decreases logarithmically. This encourages very quickly training models with high performance to bubble to the top, while not as heavily penalizing longer training models, as the logarithmic nature of scoring tends to flatten the weighting of this time factor after a certain point. The exact breakdown of these two portions of fitness could be trivially adjusted to favor smaller models or ignore model size altogether.

Due to computational cost we were unable to evaluate metrics of selection and fitness in this experiment. The evaluation is only performed on the NASBench dataset described in subsection 4.1.1.

Our primary means of reducing this complexity is to reduce the size of the search space. Due to limited resources we constrain this significantly. We limited our architecture to exactly 3 convolutional layers. Layer size is limited to a power of 2 from 2^1 to 2^7. Kernel size is limited to a power of 2, up to a maximum of half the layer size. Activation functions were chosen from RELU, GELU, and CELU with an alpha of 1. The final model output size is 512.

We select models from the population using tournament selection (see section 4.3). Pseudocode for tournament selection is as follows:

```
SelectTournament(pop, k, n):
    randomly choose k candidates from the population
    return the top n candidates from the random selection
```

We randomly choose 20% of the total population and return the top 5 candidates amongst the randomly selected candidates.

Due to the computational complexity required to run reinforcement learning algorithms to convergence, we cannot realistically use the Pong experiment to compare and analyze various evolutionary algorithm approaches as we would rapidly exceed budget. As such, we use the NASBench dataset (see subsection 4.1.1) to perform these experiments, and rely on the assumption that the results provide insights to the usage of evolutionary NAS to pong and other control policy learning tasks using reinforcement learning.

4.3 Compared Population Selection Methods

We will compare empirically a variety of population selection methods. Each of these methods is used to select candidates for the next generation, which are then mutated.

1. **Random Selection:** There is no special consideration for candidate selection. This, in theory, gives a high degree of exploration but discourages any exploitation. In a way it is very similar to random search, since candidates are picked randomly, except we explore local areas around the candidates via mutation.

2. **Greedy Selection:** The best candidates are selected in order. This approach entirely favors exploitation over exploration. We will, while using this approach, find a good performing architecture quickly. However it becomes easy to get trapped in local maxima while using this method, and there is little to no way to escape.

3. **Tournament Selection:** We use random selection to select groups of potential candidates, then select the best candidates from these groups. This encourages an exploration/exploitation

trade-off that is easily alterable by how many random candidates we select. The higher the percentage of the total population that is selected, the closer to greedy selection we get when we finally select the best candidates from the pool.

4.4 Compared Drop Methods

We maintain a consistent population size of 100 individuals. In order to perform mutation, we replace some existing models with mutations of other models. Which models get dropped from consideration depend on our choice of drop method.

1. **Greedy Dropping:** We drop the worst performing models. This is analogous to advancing the best performing models, but allows us to easily mix and match methodology. When mixed with selecting the best candidates, we refer to this as "pure greedy".

2. **Tournament Dropping:** We use random selection to select groups of potential candidates to drop, then drop the worst performing candidates from these groups. This is exactly the same as tournament selection for mutation candidates, except for the purposes of dropping models.

3. **Random Dropping:** We perform no special consideration for which models to drop, instead dropping a number of models at random. This has a heavy lean towards exploration, but harms exploitation by potentially dropping highly performing models. However, combination with a highly exploitative selection method such as greedy selection was an intriguing enough notion that we included it for purposes of gathering empirical data.

4. **Drop Oldest:** We keep track of the order in which model specifications are encountered, and drop the oldest specifications. This technique is inspired by the same in AmoebaNet [1], however here we vary the number of dropped individuals. This provides some trade-off between exploration and exploitation, because the natural intuition is that models improve over time.

Chapter 5

Experimental Results and Discussions

In this chapter, we first present empirical results on the NASBench Dataset using different population selection and drop methods for the evolutionary NAS. We compare the effect of the variations on the image classification accuracies. Finally, we present the Pong game playing mean scores and discuss the results.

5.1 Comparison Results of Selection and Drop Methods

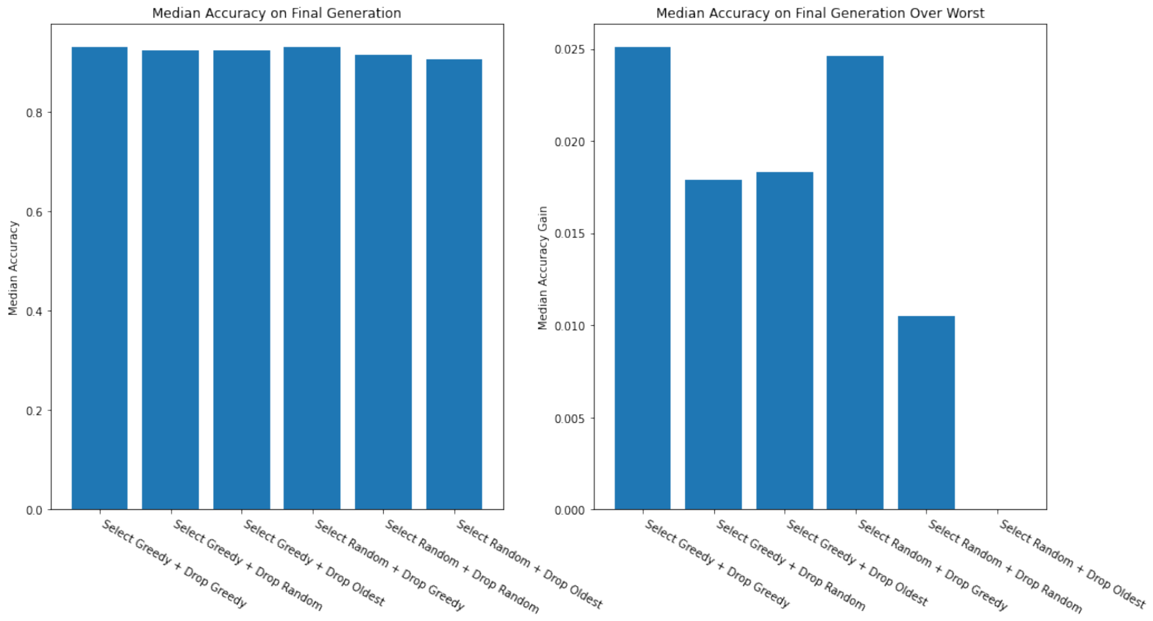
Figure 6 shows the comparison results of the mean accuracies for different selection/drop methods for evolutionary NAS after 120 generations. In particular, we present results for the following variations (i) Select 5 Greedy + Drop 5 Greedy, (ii) Select 5 Greedy + Drop 5 Random, (iii) Select 5 Greedy + Drop 5 Oldest, (iv) Select 5 Random + Drop 5 Greedy, (v) Select 5 Random + Drop 5 Random, and (vi) Select 5 Random + Drop 5 Oldest.

From Figure 6, we observe that there are two comparable methods — (i) Select 5 Greedy + Drop 5 Greedy and (iv) Select 5 Random + Drop 5 Greedy — while the other four methods lag behind by a fairly significant margin. The key insight here is that a methodology of candidate removal which considers the fitness of each model is critical. We discuss each method in more detail in the next subsection.

Next, we present the learning curves for using different variations of select/drop methods for evolutionary NAS from Figure 7 to Figure 11 and discuss the learning behavior for each variation. Each figure shows median accuracy of a generation over time for one month of computational time. Each trial represents a separate random seed, shared across all experiments with the same trial number.

Figure 6

Mean Accuracy Comparison of Different Selection/Drop Method for Evolutionary Algorithm for the NAS After 120 Generations



5.1.1 Greedy Selection

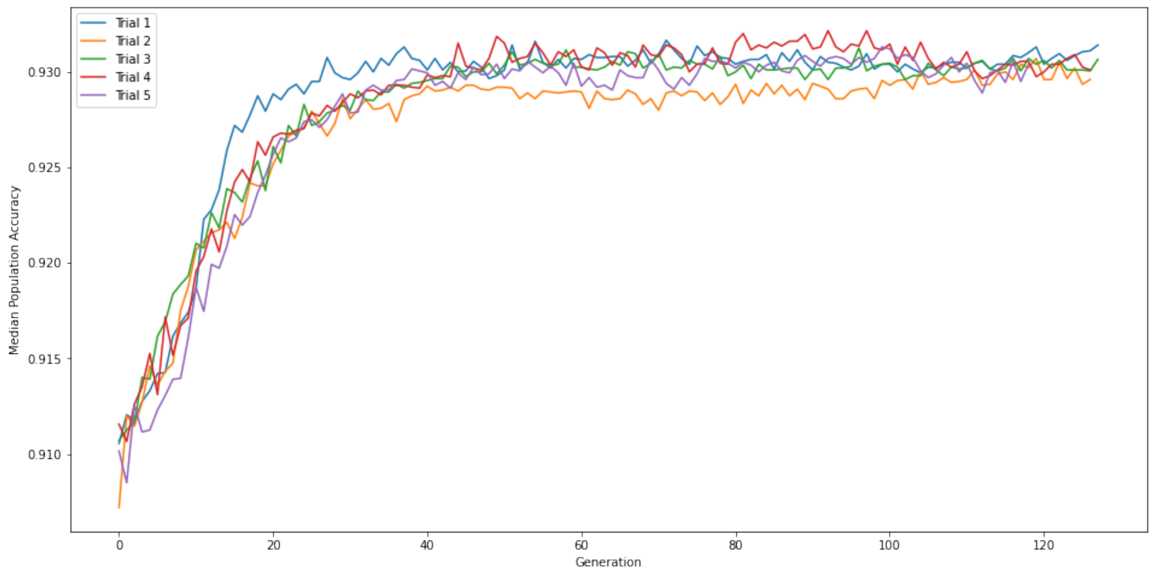
We note that pure greedy evolution performs acceptably, as shown in Figure 7. However, there is significant falloff in the improvement rate after generation ~ 20 as the poor performance models are weeded out, and the greedy selection has little way to escape the local maximum. However, as noted before, due to the simplicity of this search space, purely greedy selection performs with high efficiency. There is a small but present amount of performance that could be gained, but there are a large number of high performance models, as noted by the median test accuracy of the entire dataset being 90.65%. We expect purely greedy selection to perform acceptably on more significant datasets, but the problem of it becoming trapped in local maximums will be increased.

As shown in Figure 8 and Figure 9, the training curve of greedy selection with random dropping is slightly sharper than greedy dropping with random selection, however the greedy selection and random dropping trials have significantly higher variance. This follows logically from the premise. With random dropping, there is the possibility of dropping highly performing models, so we occasionally lose these models from consideration and as a result have very unstable performance improvement. With random selection and greedy dropping, the possibility of dropping a highly performing model is eliminated, and we only drop the least performant models. As such we create an implicit lower bound for the performance of the evaluation that improves over time.

This approach may be sufficient if any well performing architecture is needed, e.g. we do not require the best performing architecture, any will do, as it demonstrates fairly remarkable growth in the earliest portion of training. In these cases, pure greedy selection could be used for a shorter amount of time to find an acceptably performing model.

Figure 7

Learning Curves for Using Select 5 Greedy + Drop 5 Greedy



5.1.2 Oldest Dropping

Performance using the methodology of dropping the oldest architectures is abysmal, to say the least. There is simply too much variance between individual models to create a situation where older models perform worse than newer models, since there is no guarantee at all that a mutation will lead to better or equal performance. As such it is incredibly easy to throw away our highest performing model, as displayed in Figure 10.

Figure 8

Learning Curves for Using Select 5 Greedy + Drop 5 Random

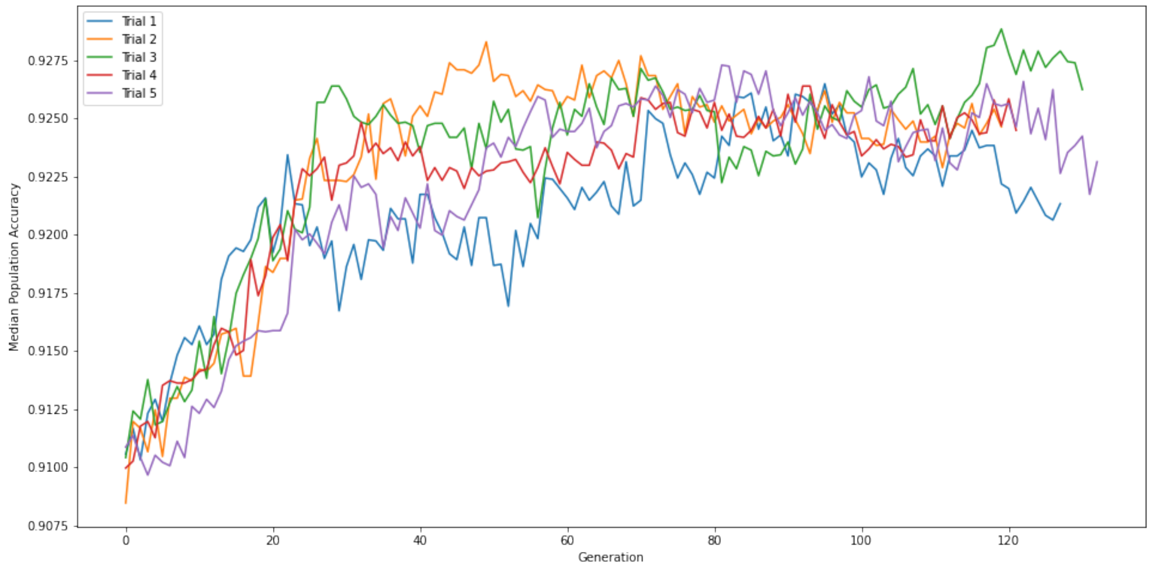
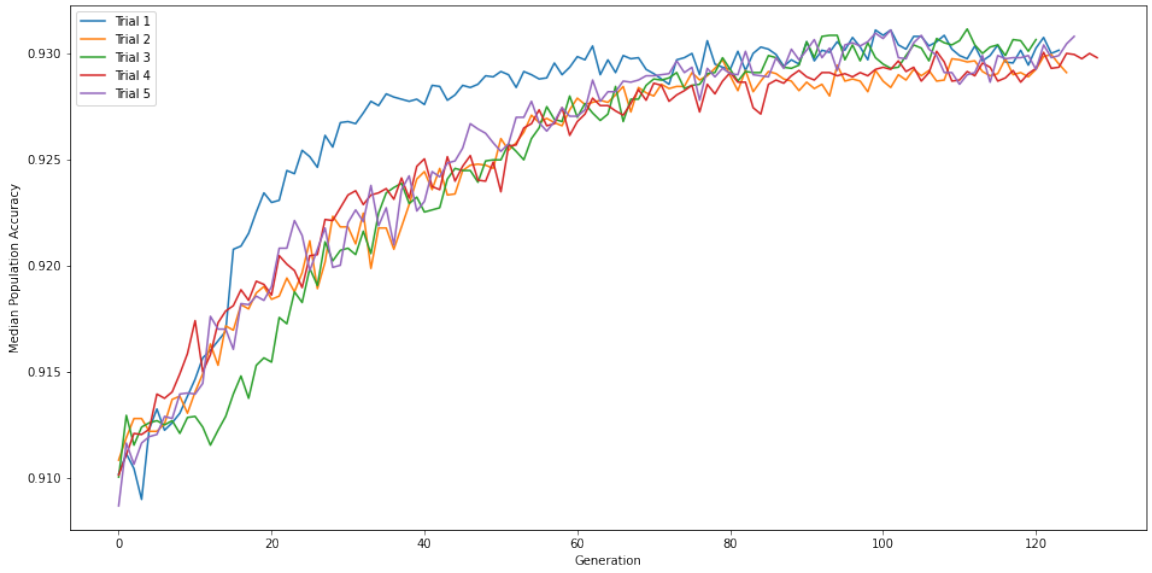


Figure 9

Learning Curves for Using Select 5 Random + Drop 5 Greedy

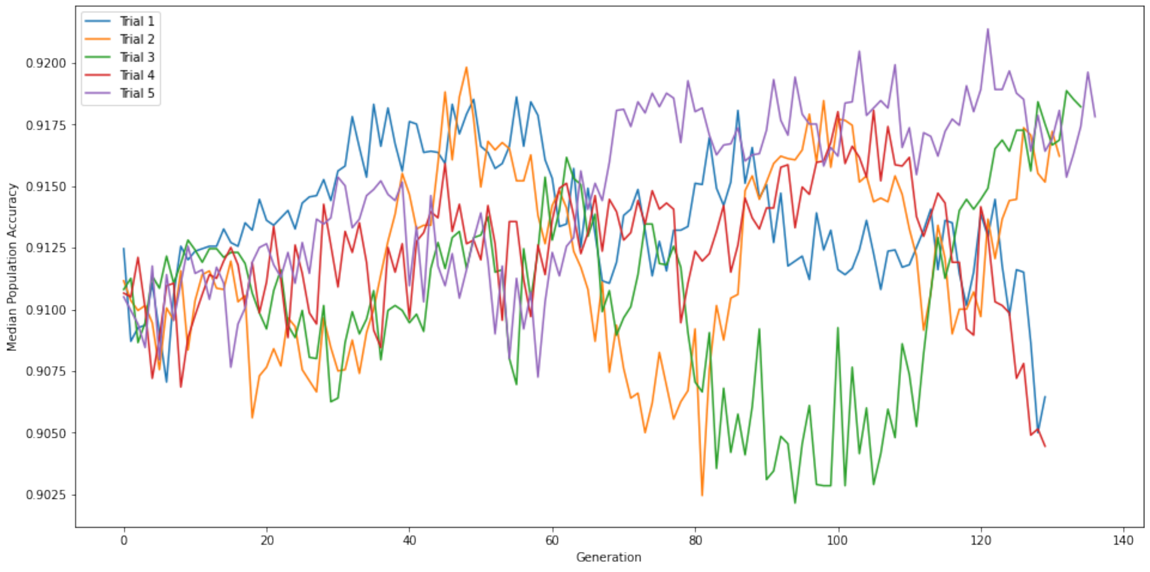


5.1.3 Random Dropping

Purely random dropping does not deserve much consideration, as it is purely random, as expected. It provides no real performance gains over time, but may accidentally stumble across a viable model. This performance is shown in Figure 11.

Figure 10

Learning Curves for Using Select 5 Random + Drop 5 Oldest



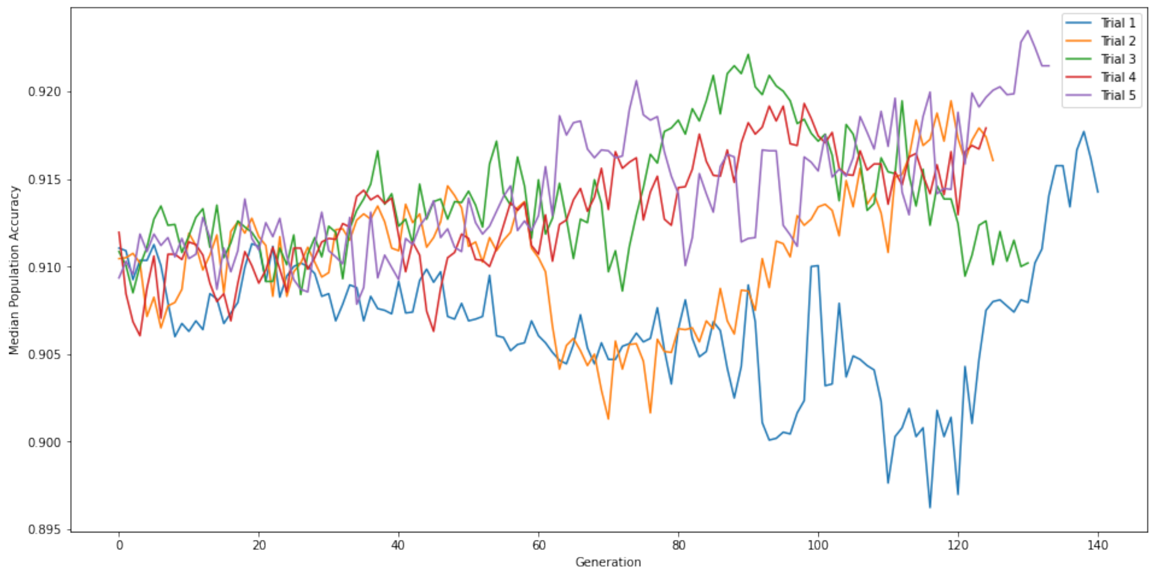
5.1.4 Tournament Selection

We next try three variations of tournament selection. We vary the size of the tournament, and scale the number of selected models with said size. For each trial, we again use a population size of 100 individuals. Figure 12 shows a graph of the hyperparameter space for tournament selection while still dropping the worst performing models. We note tournament selection's primary attractiveness is in its ability to mix exploration and exploitation naturally by configuring the size of the tournament. The larger the tournament, the closer to a fully greedy solution it is. However, there is a sweet spot where it mixes exploration and exploitation very effectively and manages to more reliably find higher performance models. The overall slope of the plane indicates that there is a slight inclination in performance as the size of the tournament grows and the number of candidates selected from the tournament decreases. However, the slope also indicates that smaller selections have a greater impact on performance than larger tournaments.

This is a somewhat unexpected, but welcome, performance increase over our baseline random selection with greedy dropping. An interesting observation is even with the gain in performance being clearly visible, it is incredibly small, which is a testament to the kind of performance the baseline is capable of. There is little reason to use any other metric, especially given the simplicity of the baseline. While there might be a slight performance improvement here, variance created by a real world problem with different properties may cause the exact parameters required to achieve greater performance to vary slightly, and as a result finding these parameters may actually increase the overall time spent.

Figure 11

Learning Curves for Using Select 5 Random + Drop 5 Random

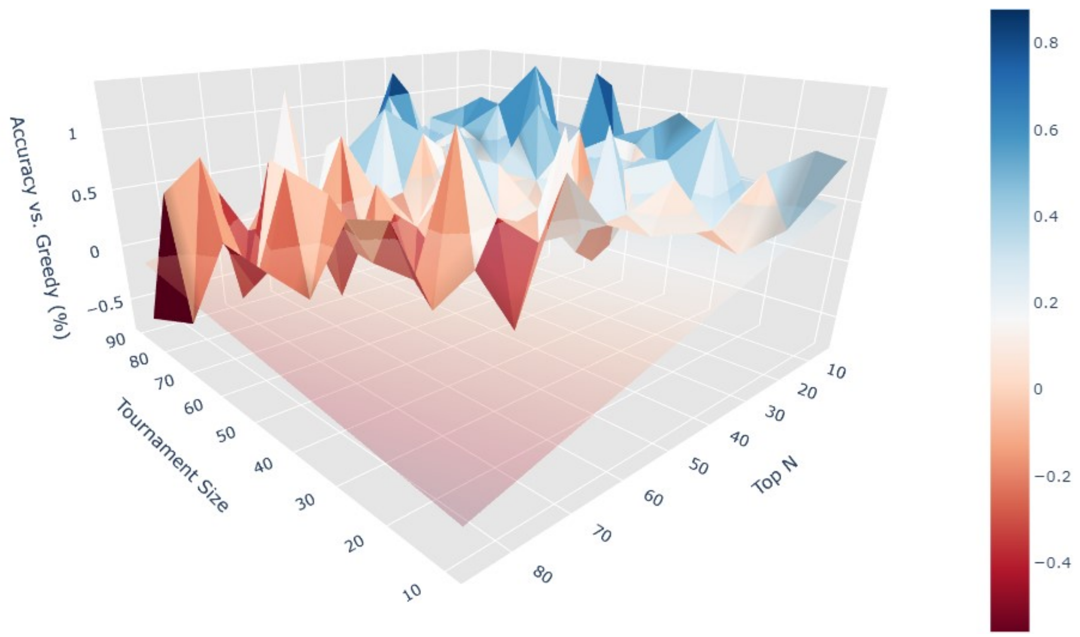


5.2 Pong - Game Playing Performance Evaluation

Early stopping was a crucial part of computational cost savings, even with this simple architecture space. Without early stopping, we estimate at least a 4x cost increase based upon the rough point at which we stop training. We do however note that training generally slows down towards the end of the period as convergence is approached, and the actual cost increase is likely to be even larger, as stopping 1/4th of the way through the raw reward threshold is not necessarily linearly correlated to training time, which may actually be much longer.

Figure 12

Tournament Selection Performance vs Random Selection + Greedy Drop



Note. Transparent plane shows is a R^2 best fit plane to demonstrate slant of parameter space.

Figure 13

Pong - Cumulative Size vs. Mean Game Score

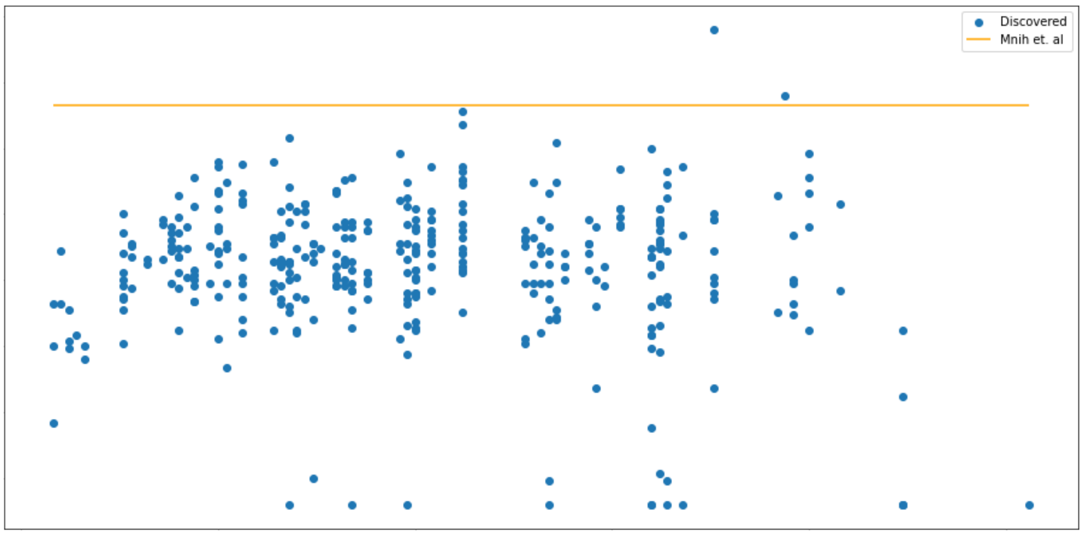


Figure 13 shows the mean model performance over the cumulative size (summing the number of neurons across all hidden layers) for playing the pong game using the learned control policy. We find there is a slight negative correlation with cumulative hidden layer size and mean reward during evaluation, and note this is somewhat expected due to the simplicity of the game Pong. The vast majority of models performed poorly.

We note a few things about this figure. Firstly, there are a few models that performed exceptionally poorly. These models all have a mean performance of -21 after evaluation, which means they did not score a single point during the evaluation period. There are nine of these such models, we have listed two of them below as representative examples.

```
2d Convolution, output size 32, kernel size 4, stride 1, padding 0,
    ReLU activation
2d Convolution, output size 128, kernel size 32, stride 1, padding 0,
    ReLU activation
2d Convolution, output size 4, kernel size 2, stride 1, padding 0,
    CELU(alpha=1) activation
Flatten to the size of the observation space
Linear layer of size 512, ReLU activation
```

```
2d Convolution, output size 64, kernel size 2, stride 1, padding 0,
    CELU(alpha=1) activation
2d Convolution, output size 128, kernel size 32, stride 1, padding 0,
    GELU activation
2d Convolution, output size 32, kernel size 4, stride 1, padding 0,
    ReLU activation
Flatten to the size of the observation space
Linear layer of size 512, ReLU activation
```

Both of these models are on the larger side of the cumulative total hidden size, with 164 and 224 hidden nodes respectively. We do note that they do not conform to a traditional architecture of increasing reduction seen in many convolutional neural network architectures, where we would expect to see each subsequent layer at least an equal in size or smaller than the previous. This is a generally accepted way to start designing sequential architectures, where we expect the increasing abstraction to pull the correct data out and eventually arrive at a decision of which action to take.

We also note that there is a range of models that performed exceptionally poorly, even some of lower sizes, so the issue cannot be explained away by appealing to the reasoning of over-complication of some architectures, but there is an underlying logic that is poorly understood. We offer this as the start of an explanation as to why these models did not perform to expectations, but can offer little proof as to this, again due to the black box nature of these networks.

Interestingly, even with this very limited experiment an architecture was found that outperformed the one described by Mnih et. al. [28] on the specific task of Pong. We describe our higher performing architecture below:

```
2d Convolution, output size 128, kernel size 2, stride 1, padding 0,
    GELU activation
2d Convolution, output size 16, kernel size 2, stride 1, padding 0,
    ReLU activation
2d Convolution, output size 32, kernel size 8, stride 1, padding 0,
    CELU(alpha=1) activation
Flatten to the size of the observation space
Linear layer of size 512, ReLU activation
```

For comparison, we also describe the model presented by Mnih et. al. [28] here:

```
2d Convolution, output size 32, kernel size 8, stride 4, padding 0,
    ReLU activation
2d Convolution, output size 64, kernel size 4, stride 2, padding 0,
    ReLU activation
2d Convolution, output size 64, kernel size 3, stride 1, padding 0,
    ReLU activation
```

```
Flatten to the size of the observation space

Linear layer of size 512, ReLU activation
```

We note that Mnih et. al. [28] designed their architecture to extract features from many different Atari games, and provide experimental results for 49 different games. We focus on a single game, again due to budget constraints. As a result, this performance improvement is interesting but niche. It does demonstrate empirically that a better architecture can be found, even with a very simple search space with many limitations. We do not expect that there would be any issues with generalizing this approach to the other 48 games in question. However, it does not disprove that the model of Mnih et. al. [28] is the best for their stated purpose - performing well at feature extraction on all 49 games.

Chapter 6

Conclusions and Future Work

In this thesis, we study the use of evolutionary algorithm for neural architecture search (NAS). Moreover, we investigate the effect of integrating evolutionary NAS into deep reinforcement learning to learn control policy for ATARI game playing. Empirical classification results on the NASBench image dataset using different population selection and drop methods for the evolutionary NAS are presented to validate the usefulness of simple evolutionary algorithms in optimizing neural architectures, showing that even basic evolutionary algorithms are a well performing and easy to use approach. We also show the feasibility of using evolutionary NAS to extract good features that can improve the pong game playing score when the computational resource is limited.

There are many directions that future work can go, however there are a few key issues that are still plaguing further progress. First and foremost, existing research is unable to provide real insight as to why some architectures work well and why others do not. The only generally accepted knowledge is that more complex tasks require larger architectures, but even this sometimes does not hold true [3].

The largest and most critical area of research in our opinion is therefore on development of understanding of the internal architectures that lead to positive performance. This knowledge would be instrumental to the future research on NAS subjects, and would likely drastically reduce the computational and human cost of architecture design. In essence, the field of NAS exists because of this particular problem. It is, however, a large way off from solving.

There is some work being done in this area though, we point attention to studies such as the ones by Geiping et al. [29] on inverse problems and the stability of DARTS, which has promise to begin to elucidate the inner workings — in the non-mathematical sense — of these algorithms and what sorts of problems they work well on and what sorts of problems they fail to produce satisfactory results on. More research on this is needed, but once the shortcomings of existing top performance methods are fully understood, they may be iterated on or new methods can be produced which do not produce these shortcomings.

As for evolutionary architecture search methodology, it has been shown to be successful: see herein [18]. Most of these methods however use fairly complex methodology in comparison to what was shown in this work, and some well known methods such as AmoebaNet-A use far greater processing power than is commonly available (450 CPUs for 2-5 days). It's performance was certainly better than that which was presented in this work, leading our simplistic methodology by 4%, it is arguable if this performance is worth the cost. AmoebaNet-A demonstrated (at the time) state of the art performance, recent focus in neural architecture search has moved away from evolutionary algorithms and into gradient based approaches due to their exceptional performance and comparatively reduced expert knowledge requirements due to gradient learning, rather than manually created fitness metrics. As such, the future of evolutionary algorithms in this space is unknown, but this work demonstrates that they can be a quick and easy methodology which provides solid performance.

References

[1] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, *Regularized evolution for image classifier architecture search*, 2019. arXiv: 1802.01548 [cs.NE].

[2] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 8697–8710. DOI: 10.1109/CVPR.2018.00907.

[3] T. Elsken, J. H. Metzen, and F. Hutter, *Neural architecture search: A survey*, 2019. arXiv: 1808.05377 [stat.ML].

[4] B. Zoph and Q. V. Le, *Neural architecture search with reinforcement learning*, 2017. arXiv: 1611.01578 [cs.LG].

[5] B. Baker, O. Gupta, N. Naik, and R. Raskar, *Designing neural network architectures using reinforcement learning*, 2017. arXiv: 1611.02167 [cs.LG].

[6] A. Brock, T. Lim, J. M. Ritchie, and N. Weston, *Smash: One-shot model architecture search through hypernetworks*, 2017. arXiv: 1708.05344 [cs.LG].

[7] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," vol. 13, pp. 281–305, Feb. 2012, ISSN: 1532-4435.

[8] C. Liu *et al.*, *Progressive neural architecture search*, 2018. arXiv: 1712.00559 [cs.CV].

[9] L. Weng, "Neural architecture search," *lilianweng.github.io/lil-log*, 2020. [Online]. Available: https://lilianweng.github.io/lil-log/2020/08/06/neural-architecture-search.html.

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

[11] B. Baker, O. Gupta, R. Raskar, and N. Naik, *Accelerating neural architecture search using performance prediction*, 2017. arXiv: 1705.10823 [cs.LG].

[12] H. Cai, J. Yang, W. Zhang, S. Han, and Y. Yu, "Path-level network transformation for efficient architecture search," in *International Conference on Machine Learning*, PMLR, 2018, pp. 678–687.

[13] D. Ha, A. Dai, and Q. V. Le, *Hypernetworks*, 2016. arXiv: 1609.09106 [cs.LG].

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

[15] G. Bender, P.-J. Kindermans, B. Zoph, V. Vasudevan, and Q. Le, “Understanding and simplifying one-shot architecture search,” in *Proceedings of the 35th International Conference on Machine Learning*, J. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, PMLR, Oct. 2018, pp. 550–559. [Online]. Available: https://proceedings.mlr.press/v80/bender18a.html.

[16] D. H. Tizhoosh. “Machine intelligence - lecture 18 (evolutionary algorithms),” Kimia Lab, University of Waterloo. (Apr. 2019), [Online]. Available: https://www.youtube.com/watch?v=3-NiZPbkr7A.

[17] P. Vikhar, “Evolutionary algorithms: A critical review and its future prospects,” Dec. 2016, pp. 261–265. DOI: 10.1109/ICGTSPICC.2016.7955308.

[18] Y. Liu, Y. Sun, B. Xue, M. Zhang, G. G. Yen, and K. C. Tan, “A survey on evolutionary neural architecture search,” *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[19] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *arXiv preprint arXiv:1707.06347*, 2017.

[20] E. S. Biratu, F. Schwenker, Y. M. Ayano, and T. G. Debelee, “A survey of brain tumor segmentation and classification algorithms,” *Journal of Imaging*, vol. 7, no. 9, 2021, ISSN: 2313-433X. DOI: 10.3390/jimaging7090179. [Online]. Available: https://www.mdpi.com/2313-433X/7/9/179.

[21] E. Yurtsever, J. Lambert, A. Carballo, and K. Takeda, “A survey of autonomous driving: Common practices and emerging technologies,” *IEEE Access*, vol. 8, pp. 58 443–58 469, 2020. DOI: 10.1109/ACCESS.2020.2983149.

[22] C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy, and F. Hutter, “NAS-bench-101: Towards reproducible neural architecture search,” in *Proceedings of the 36th International Conference on Machine Learning*, K. Chaudhuri and R. Salakhutdinov, Eds., ser. Proceedings of Machine Learning Research, vol. 97, Long Beach, California, USA: PMLR, Sep. 2019, pp. 7105–7114. [Online]. Available: http://proceedings.mlr.press/v97/ying19a.html.

[23] A. Krizhevsky, G. Hinton, *et al.*, “Learning multiple layers of features from tiny images,” 2009.

[24] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, *Learning transferable architectures for scalable image recognition*, 2018. arXiv: 1707.07012 [cs.CV].

[25] V. Mnih *et al.*, “Playing atari with deep reinforcement learning,” *arXiv preprint arXiv:1312.5602*, 2013.

[26] C. Watkins, “Learning from delayed rewards,” Jan. 1989.

[27] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, *Trust region policy optimization*, 2017. arXiv: 1502.05477 `[cs.LG]`.

[28] V. Mnih *et al.*, *Human-level control through deep reinforcement learning*, 2015. Nature: 10.1038 (cs.NE).

[29] J. Geiping, J. Lukasik, M. Keuper, and M. Moeller, “DARTS for inverse problems: A study on stability,” in *NeurIPS 2021 Workshop on Deep Learning and Inverse Problems*, 2021. [Online]. Available: https://openreview.net/forum?id=ty5XCitJfLA.