APPLICATIONS OF NEURAL ARCHITECTURE SEARCH TO DEEP REINFORCEMENT LEARNING METHODS

An Undergraduate Research Scholars Thesis

by

WILLIAM ALLEN[1], ZACHARY LINDSEY[2]

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Sarah J. Witherspoon

May 2023

Majors: Applied Mathematics[1]
Mathematics[2]

Copyright © 2023. William Allen[1], Zachary Lindsey[2].

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

We, William Allen[1], Zachary Lindsey[2], certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with our Faculty Research Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

Page

ABSTRACT 1

DEDICATION 2

ACKNOWLEDGMENTS 3

NOMENCLATURE 4

1. INTRODUCTION 5

   1.1 Modern Machine Learning & Types of Neural Architectures 6
   1.2 Project Details 8

2. OTHELLO 9

   2.1 Game Rules 9
   2.2 History and Computational Methods 10

3. REINFORCEMENT LEARNING & ALPHAZERO 12

   3.1 Formal Description of Reinforcement Learning 12
   3.2 AlphaZero 13
   3.3 Monte Carlo Tree Search 14
   3.4 Integration of MCTS in AlphaZero 15
   3.5 Exploration and Exploitation with AlphaZero 16

4. NEURAL ARCHITECTURE SEARCH 17

   4.1 Overview 17
   4.2 Applications 18
   4.3 Implementations 20
   4.4 Challenges and Future Directions 21

5. METHODS 22

   5.1 Neural Architecture Search 22
   5.2 Self Play 23
   5.3 Training 24
   5.4 Ranking 25
   5.5 Performance Considerations 26

6. RESULTS 27

7. CONCLUSION 34

REFERENCES 36

ABSTRACT

Applications of Neural Architecture Search to Deep Reinforcement Learning Methods

William Allen[1], Zachary Lindsey[2]
Department of Mathematics[1, 2]
Texas A&M University

Faculty Research Advisor: Dr. Sarah J. Witherspoon
Department of Mathematics
Texas A&M University

This research aims to investigate the impact of various Neural Architectures (NAs) on the performance of machine learning models in the context of the deterministic game of Othello, with the goal of providing insights into selecting optimal neural architectures for a given problem description. We constructed several fundamentally different neural architectures, including fully connected networks and convolutional networks, and assessed their performance across various metrics such as win rate, training time, and computational resource consumption. By correlating these performance changes between NAs with their relevant structural differences, we sought to offer technical insights that can guide the selection and composition of ML architectures for other problem-solving contexts with deterministic constraints analogous to those in Othello. Furthermore, this research emphasizes the importance of understanding the trade-offs between different NAs in terms of resource efficiency, learning capability, and adaptability to specific problem domains, thereby facilitating more informed decisions in the development of machine learning models.

DEDICATION

To our families, instructors, and peers who supported us throughout the research process. Nurturing every venture, embracing relentless growth, offering nurturing advice - gracious individuals valuing effort, yielding outstanding understanding, unlimited potential.

ACKNOWLEDGMENTS

Contributors

We would like to thank our faculty advisor, Dr. Sarah J. Witherspoon, for her guidance and support throughout the course of this research.

The data used for APPLICATIONS OF NEURAL ARCHITECURE SEARCH TO DEEP REINFORCEMENT LEARNING METHODS was provided by William Allen. The analyses and illustrations thereof depicted in APPLICATIONS OF NEURAL ARCHITECURE SEARCH TO DEEP REINFORCEMENT LEARNING METHODS were conducted in part by William Allen & Zachary Lindsey and the data is unpublished.

All other work conducted for the thesis was completed by the students independently.

Funding Sources

No funding was received for this project.

NOMENCLATURE

CNN	Convolutional Neural Network
ENAS	Evolutionary Neural Architecture Search
DARTS	Differentiable Architecture Search
FC	Fully Connected
GCN	Graph Convolutional Networks
LSTM	Long Short Term Memory
MCTS	Monte Carlo Tree Search
ML	Machine Learning
MLP	Multilayer Perceptron
NLP	Natural Language Processor
NA	Neural Architecture
NAS	Neural Architecture Search
ReLU	Rectified Linear Units
RL	Reinforcement Learning
RNN	Recurrent Neural Network
UCB	Upper Confidence Bound
$v_{\pi^*}(s)$	Optimal policy
$\pi^*(s)$	Improved policy

1. INTRODUCTION

The growing popularity of Machine Learning (ML) along with the advancing techniques and technologies that power its application have continuously and exponentially diversified its application in many fields. This has led to the invention of many different methodologies for constructing ML systems. In principle, each ML algorithm involves a type of Neural Network (NN) along with a companion learning algorithm.

A NN can be visually represented as a bidirectional graph with nodes called "neurons" and connections called "weights." In a NN, there are several layers of neurons, with each neuron in a layer connecting to neurons in other layers. Figure 1 shows a fully connected Multilayer Perceptron (MLP) Neural Network. The figure on the right is colored to show the value of weights as the network is trained.

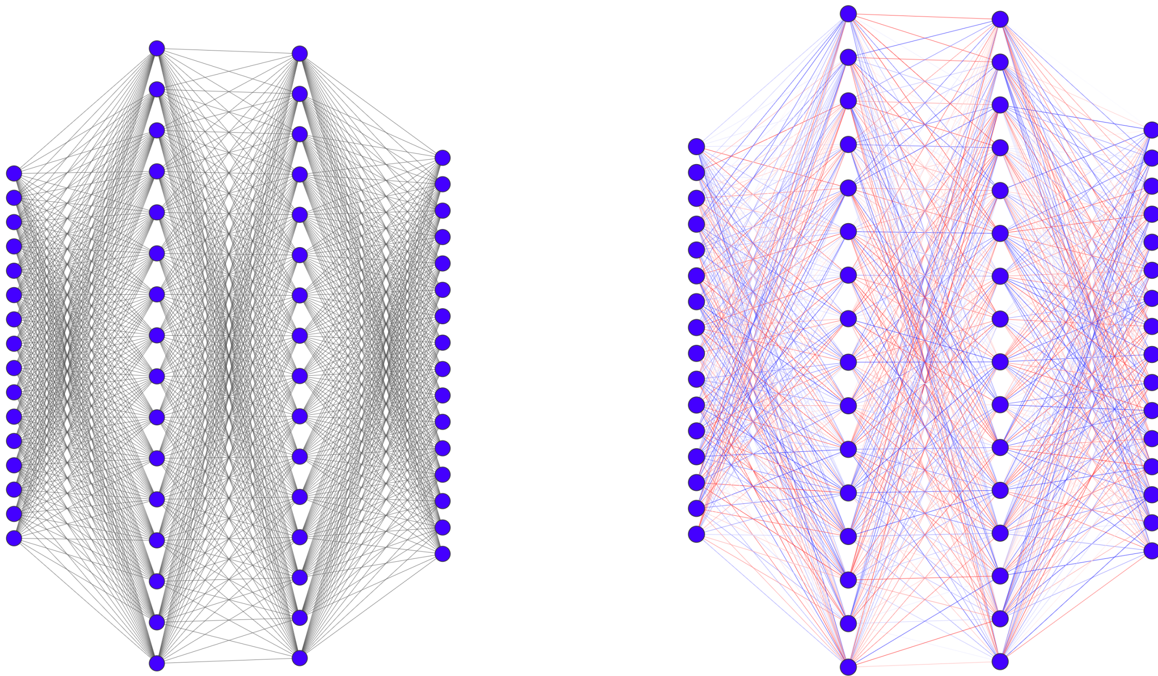

Figure 1: Multilayer Perceptron (MLP)

The input layer receives an encoded input, such as the numerical description of a board game's configuration at a particular moment, and the final output layer returns an encoded description associated with the chosen output space, such as the weighted values of each potential move given a game state. Take the example of Chess. In Chess, the encoded input is the current state of the board translated into a numerical string, and the various layers perform an algorithmic transformation of this information into an encoded description of the output space: a probability distribution mapping each legal move to the probability (or "value") of the move being played.

The learning algorithm that accompanies the NN is often categorized into one of three types: supervised, unsupervised, and reinforcement-based learning. Based on the individual goal of the NN implementation, this learning algorithm will "train" the NN.

1.1 Modern Machine Learning & Types of Neural Architectures

In practice, there are many different types of Neural Networks that are employed. Specific constructions of NN are typically called Neural Architectures (NA). For example, Google DeepMind's AlphaGo agent is implemented as a deep Convolutional Neural Network (CNN) [1]; Natural Language Processors (NLPs), such as chatGPT, are typically implemented as transformer networks [2]. For tasks where data is represented sequentially, an engineer may opt to additionally include either a recurrent neural network (RNN) [3] or a long-short-term-memory network (LSTM) [4]. Situations where the input data is mostly categorical make use of embedding layers [5], and so on.

Modern ML architectures are often implementations of an amalgam of several different architectures. For example, DeepMind's StarCraft II agent AlphaStar [6], an agent trained to play the board game Catan [7], an agent trained to play the board game Monopoly [8] and others all use a variety of different machine learning architectures as interconnected pieces of the entire network. Figure 2 shows the Neural Architecture of AlphaStar in which 8 different neural architectures are employed. The purpose of integrating different architectures is typically optimization. Singular architectures may be strategically inadequate at handling the task's scope, making them resource-inefficient. Combining multiple architectures that address the shortcomings of a singular

architecture can yield significant optimization.

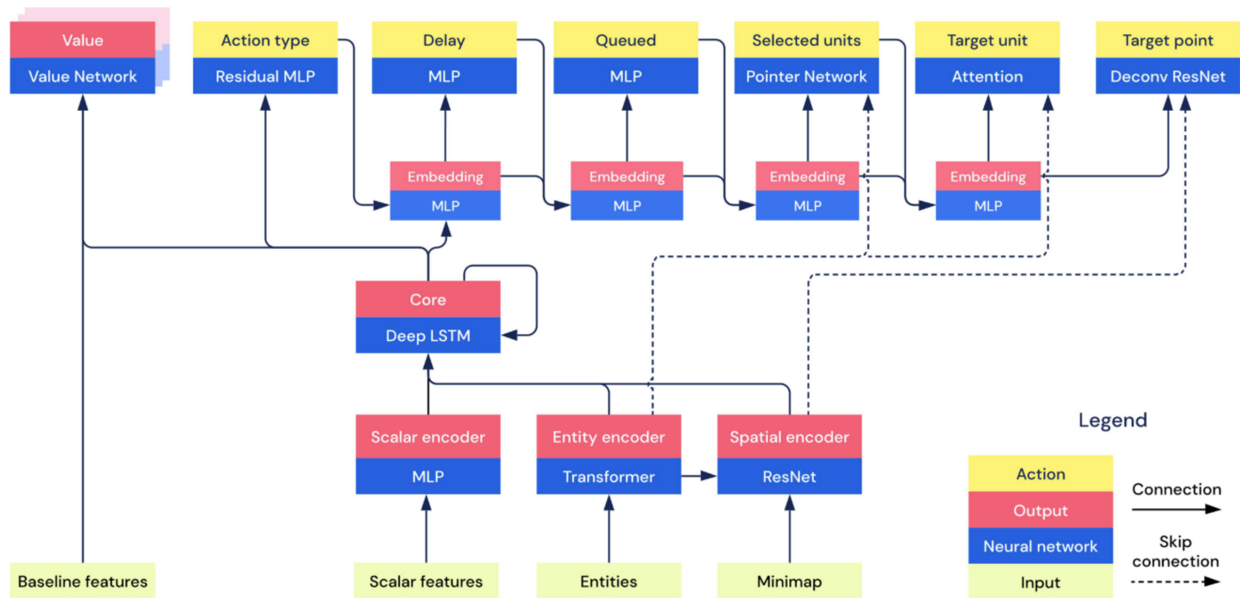

Figure 2: Neural Architecture of AlphaStar

Constructing many modern and highly successful task-oriented architectures has largely been guided by intuition [9] In fact, current ML convention views the selection of which architectures to include and how to implement them as a general, intuitive process. This means that there is no generic way to determine the correct architecture a priori, given only a problem description. This is concerning, as training a fundamentally suboptimal NN on a supercomputer cluster – only to replace it with a refined or entirely different network after performance analysis – can be extremely expensive in terms of time and financial resources.

In essence, selecting an appropriate Neural Architecture for a given problem description remains an open problem, and solving it can yield substantial benefits. Both creating an optimized NN for a problem and refining the accuracy of the initial composition of a Neural Architecture would significantly reduce a project's resource consumption and improve the rate at which a NN achieves desired performance [10].

1.2 Project Details

This research aims to provide technically appropriate literature regarding intuitive methods for selecting problem-fit Neural Architectures. To accomplish this, we will produce a pool of architectures to train the reinforcement learning algorithm AlphaZero with. The board game Othello will represent the problem we intend to solve. We will construct a programmatic model of Othello to represent the environment for our agent. Then, we will construct several categorically distinct NAs, along with permutations of these NAs within their categories, to compete against one another in search of the best performer.

The information obtained during this competition will allow for a comparative analysis between NAs across various performance metrics such as win rate, win margins, training time, iterative score, board control, computational resource consumption, and more. By correlating these performance changes between NAs with their relevant structural differences, we aim to provide technical insight into the selection and composition of ML architectures for other problem-solving contexts that are isomorphic to the deterministic constraints of Othello.

Lastly, we will perform this training on consumer-grade computational resources, i.e., processing power within the average range of a home desktop. The secondary purpose of this restriction (as opposed to purchasing or otherwise acquiring time with a supercomputer for training) is to address the issue of scale that permeates ML applications. Although ML applications are more diverse than ever, demonstrating new ML technologies and training new successful agents are typically limited to contexts with substantial computational resources. By recording the aforementioned analytics and integrating modern optimization techniques, we also aim to present literature regarding which NAs are performance/resource consumption viable for contexts with limited resources.

2. OTHELLO

Othello, a modern variant of the game Reversi invented in 1883, is a board game played on an 8x8 tiled grid with up to 64 disks, each having a dark side and a light side. The primary distinction between the two games is Othello's initial state, which has the center 2x2 square populated with four pieces, two belonging to 'light' and 'dark' respectively, arranged diagonally with same-color pieces. There is no standard with respect to which color is in a certain diagonal. In our project, we use both openings (Figure 3).

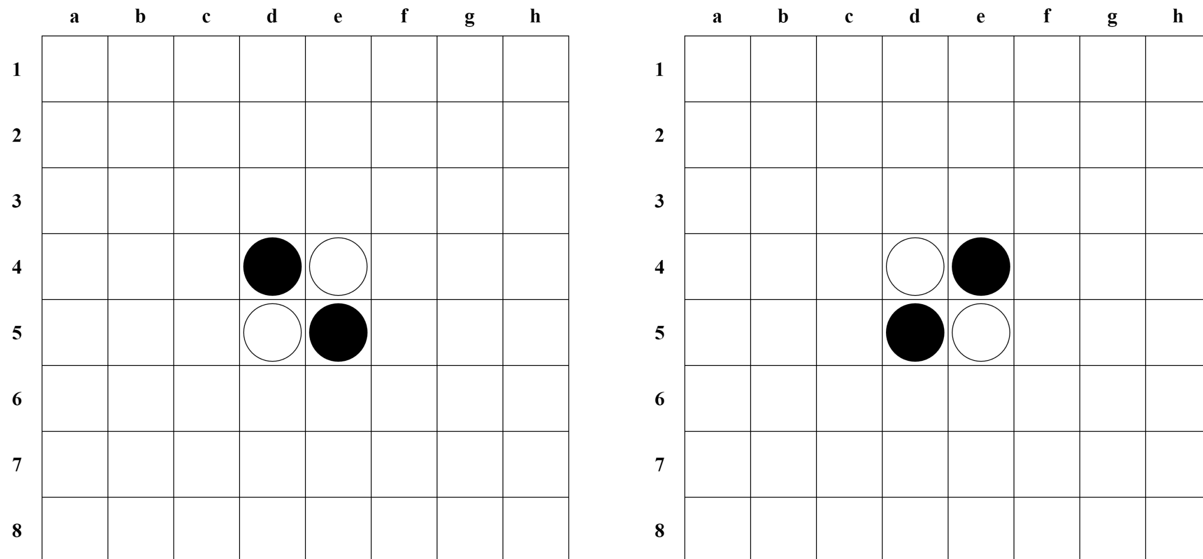

Figure 3: Two possible opening layouts for Othello.

2.1 Game Rules

To start the game, four disks are placed in the center of the board, with two disks of each color placed diagonally across from each other. The player using the dark pieces moves first, and the players take turns placing disks on the board. A move is considered legal if it results in

at least one of the opponent's disks being sandwiched between two of the player's disks, either horizontally, vertically, or diagonally. When a disk is placed in a position that sandwiches one or more of the opponent's disks, those disks are flipped over to the player's color.

If a player cannot make a legal move, they must pass their turn. The game ends when both players have passed, or when the board is full. The player with the most disks of their color on the board at the end of the game is the winner. Ties are also possible.

Figure 4 demonstrates an example of how the first three moves might play out. The legal moves for the current player are indicated by the small blue circles. Black moves first by placing a tile in position e3, flipping the tile along the vertical at e4. On the next move, white places a tile at f3, flipping the tile along the diagonal at e4.

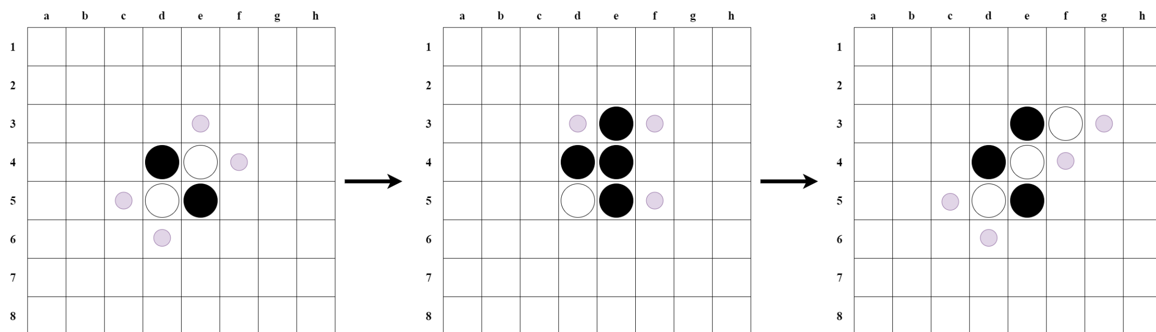

Figure 4: Example of game progression for the first three moves.

2.2 History and Computational Methods

Othello's deterministic nature, straightforward rules, and gameplay make it an attractive target for various dimensions of computer play and research. A deterministic game is one without outcomes dependent on chance, and it can be solved. Solvability refers to the ability to determine optimal play from any position. In other words, from any configuration of a deterministic game's state, it is knowable which moves to play to guarantee victory, a tie, or determine if the game state inevitably results in a loss.

The solvability of various perfect games remains an active concept in computer and human play. For instance, the most advanced mathematical conjecture for Chess has come close to predicting the solution but has not perfected it. As a result, computer programs playing Chess are not playing perfectly, but only highly optimally, and can still be both beaten and improved upon. This further refinement and optimization towards perfect play is where the significance of an optimally curated NA arises.

Othello played on a 4x4 and 6x6 board has been weakly solved [11]. The rising game tree complexity, referring to the number of possible ways to arrive at each unique board configuration, has prevented the solving of Othello on a full 8x8 board. The state space complexity of (8x8) Othello, referring to the total number of unique configurations, is 10^{28} [12] The game tree complexity of (8x8) Othello is 10^{58} [12].

In comparison, the state space complexity of Chess is 10^{46} [13], and its game tree complexity is $10^{10^{50}}$ [13]. The differences between the rules and dynamics of these two games, despite their comparable dimensions, account for the tremendous difference in complexity. However, fundamentally, they can be approached by the same ML algorithms and techniques. For the purpose of demonstration, agents trained to play Othello will exhibit evolutionary growth and improvement significantly faster. Othello's low complexity makes it an ideal space for the development and training of new architectural models, and by extension, also for our research.

3. REINFORCEMENT LEARNING & ALPHAZERO

Reinforcement Learning (RL) is a widely-used machine learning technique that typically involves an environment, an agent, an action space for the agent to sample from, and a reward function. The agent learns the relationships between the environment state, action, and reward through exploration and aims to maximize the accumulated reward.

RL is a powerful approach within machine learning that largely eliminates the need for data acquisition. The agent's independence and its freedom to make decisions and develop policies enable the creation of unique and robust strategies. For instance, chess models trained through reinforcement learning are known to play unorthodox moves. Moreover, reinforcement learning's reliance on a reward function instead of a finite goal allows it to be applied to a wider range of problems, including efficient resource management and personalized recommendations, which are highly valuable in various business domains.

However, without proper moderation, RL-based training can lead to an overload of states and eventually decrease generational performance. In other words, it is possible for an agent to learn detrimental habits that hinder or even reverse its progress.

3.1 Formal Description of Reinforcement Learning

A more precise definition of reinforcement learning involves an agent learning an optimal policy π for taking an action a in a state s to maximize the expected value of a reward R_t at any time step t. Specifically, R_t is the sum of actual discounted rewards r_t at time step t. The reward can be a negative value to indicate undesired behavior or action.

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{1}$$

Where $\gamma \in (0, 1]$ controls the degree to which we value potential future rewards. A low γ results in a greedy agent that prioritizes immediate rewards, while a high γ leads to an exploratory agent that engages in more risk-seeking behavior.

The policy π is a probability distribution that maps all available actions at a given timestep A_t and indicates the likelihood of choosing a specific action a given s.

$$\pi(a \mid s) = P[A_t = a \mid S_t = s] \tag{2}$$

One critical concept in reinforcement learning is the state-action value function, or Q-value, which estimates the expected return of taking action a in state s under policy π.

$$q_\pi(s, a) = E_\pi[R_t \mid S_t = s, A_t = a] \tag{3}$$

Furthermore, we can define the state-value function as the expected reward, given a policy π and state s.

$$v_\pi(s) = E_\pi[R_t \mid S_t = s] \tag{4}$$

Ultimately, we seek the optimal policy v_{π^*} that provides the maximum reward:

$$v_{\pi^*}(s) = \max_\pi v_\pi(s) \tag{5}$$

3.2 AlphaZero

AlphaZero [14], developed by researchers at Google DeepMind, is a sophisticated and generalized version of AlphaGo. AlphaGo utilizes a reinforcement learning technique, primarily employing a Monte Carlo Tree Search (MCTS) to perform guided searches in the state space of the board game Go. AlphaZero expands upon this concept by eschewing the need for supervised learning or pre-existing player data used in AlphaGo's training process. Instead, AlphaZero solely relies on reinforcement learning and MCTS.

This innovative approach necessitates only the rules of the game as input, rendering AlphaZero versatile and applicable to a broad array of finite deterministic domains. By employing self-play, AlphaZero can efficiently explore and learn from the vast state space of complex games, discovering optimal strategies and continually refining its gameplay.

3.3 Monte Carlo Tree Search

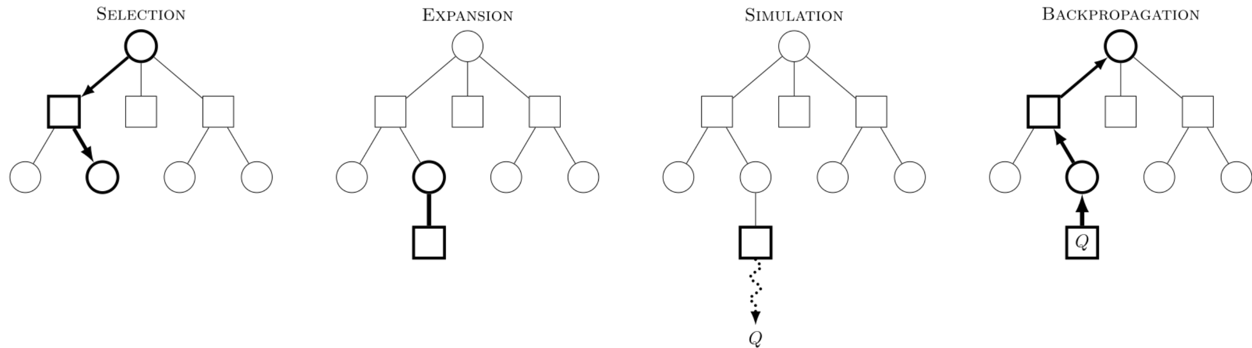

Figure 5: Diagram of four phases of Monte Carlo Tree Search.

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm that involves a tree search and random sampling. It is widely used in artificial intelligence applications, particularly for games and planning problems. MCTS performs a best-first search, guided by the outcome of simulated random playouts, to explore the search space effectively.

In a MCTS, the search tree is iteratively expanded by adding new nodes to the tree. Each node represents a unique state in the search space, and the edges between nodes correspond to the legal actions that lead to new states. MCTS consists of four primary steps: selection, expansion, simulation (or rollout), and backpropagation (Figure 5).

During the selection step, the algorithm traverses the search tree from the root node, choosing the child node with the highest upper confidence bound (UCB) until an unexpanded node is reached. The UCB balances exploration and exploitation by taking into account both the estimated value of the node and the number of times the node has been visited.

Once an unexpanded node is reached, the expansion step occurs. A new child node is added to the tree by applying a legal action to the current node, resulting in a new game state. Following expansion, the simulation (or rollout) step is performed. The algorithm simulates a

random sequence of actions from the new child node until a terminal state is reached or a predefined depth limit is hit. The outcome of this simulation is used to estimate the value of the new node.

Finally, in the backpropagation step, the algorithm updates the value estimates and visit counts of all the nodes along the path from the newly added node back to the root node. This process is repeated for a fixed number of iterations or until a predefined depth limit is reached. The algorithm then selects the action corresponding to the most visited child node of the root node as the best action to take.

3.4 Integration of MCTS in AlphaZero

The integration of Monte Carlo Tree Search (MCTS) in AlphaZero can be summarized in three main points:

1. AlphaZero does not perform rollouts. Instead, it relies on its neural network to evaluate the nodes in the search tree Specifically, when the algorithm reaches a new state $s^{'}$ that is not present in the search tree, it uses its neural network f_θ to estimate the value $v(s)$ of the new state and a probability distribution $\mathbf{P}(s)$ across all legal actions from the new state.

In the case that the new state $s^{'}$ already exists in the search tree, then the MCTS proceeds as normal. When the MCTS reaches a terminal node, the game reward $r \in \{-1, 0, 1\}$ is propagated up the search tree as the value v. This value estimation, provided by the neural network, serves as a more informed heuristic compared to a random rollout, allowing AlphaZero to focus its search more efficiently.

2. The UCB is based on the output of the neural network and takes into account both the estimated value of the state $v(s)$ and the probability distribution across legal moves $\mathbf{P}(s)$. During MCTS, the algorithm selects the action a^* that maximizes the UCB among all legal actions:

$$U(s,a) = Q(s,a) + \lambda \cdot \mathbf{P}(s,a) \cdot \frac{\sqrt{\sum_\alpha N(s,\alpha)}}{1 + N(s,a)} \tag{6}$$

where

$$Q(s,a) = \frac{N(s,a) \cdot Q(s,a) + v}{N(s,a) + 1} \tag{7}$$

is the expected reward for taking action a from a given state s, $N(s,a)$ is the cross-simulation count of how many times this action has been taken. The parameter λ is a hyperparameter that controls the degree of exploration.

3. AlphaZero uses a self-play training approach where it plays against itself using the current neural network to generate moves. As usual, after the MCTS has finished its iterations, we take a normalized count of $N(s,a)$ to produce the improved policy vector $\pi^*(s)$:

$$\pi^*(s) = \left\{ \frac{N(s,a)^{1/\tau}}{\sum_{\alpha} N(s,\alpha)^{1/\tau}} \right\} \tag{8}$$

for each legal action a of a state s. τ is the temperature parameter and controls the degree of exploration during evaluation. The resulting gameplays are used to update the neural network weights, which are then used for the next iteration of self-play. This approach allows AlphaZero to continually improve its performance through experience gained from playing against itself. Specific details of our implementation can be found in Methods.

3.5 Exploration and Exploitation with AlphaZero

Balancing exploration and exploitation is a vital aspect of reinforcement learning. Exploration refers to the agent's attempts to discover new information about its environment by trying different actions, while exploitation refers to the agent's use of its existing knowledge to make decisions that maximize rewards. Striking the right balance between exploration and exploitation is crucial for the agent's performance, as excessive exploration may lead to inefficient learning, while overemphasis on exploitation may result in suboptimal solutions.

The use of the neural network f_θ to infer the value of new nodes not only speeds up the search process but also reinforces the balance between exploration and exploitation. The neural network learns from the game trajectories generated by MCTS, continuously refining its understanding of the state-action value function $q_\pi(s,a)$. This learned knowledge is then employed to guide the MCTS search, allowing AlphaZero to concentrate on more promising areas of the search space.

4. NEURAL ARCHITECTURE SEARCH

The field of deep learning has experienced remarkable growth and success over the past decade, with applications spanning various domains such as computer vision, natural language processing, and reinforcement learning. Central to this success is the design and optimization of neural network architectures, which often require considerable human expertise and trial-and-error experimentation. As deep learning continues to evolve, there is an increasing need for automated methods that can discover and optimize neural architectures more efficiently and effectively.

One of the goals of Neural Architecture Search (NAS) is to automate the process of finding the most suitable neural network architecture for a given task. Traditionally, this process has been performed by human experts, who manually design and optimize architectures based on their experience and knowledge. However, NAS aims to leverage computational power and advanced search algorithms to automatically discover architectures that can achieve superior performance compared to those designed by humans.

4.1 Overview

Systems that employ neural networks (NN) have the property of dynamic interchangeability, which allows for the structure of the layers (the NN itself) to be easily swapped out, as long as the domain of the encoded input and desired output remain the same. This property is especially useful for adapting a pre-existing neural network to a new problem or for exploring various architectures to identify the optimal NN implementation for a given task. The potential performance differences between various architectures make this exploration worthwhile, as it can lead to improved efficiency and effectiveness in solving complex problems.

To understand how dynamic interchangeability works, consider the following example: Suppose we have a neural network designed for image classification using a Convolutional Neural Network (CNN) architecture. If we wanted to test whether a different architecture, such as a Recurrent Neural Network (RNN), could perform better on the same image classification task, we

could swap out the CNN layers with RNN layers while keeping the input and output encoding the same. This would allow us to compare the performance of the two architectures on the same problem without making significant changes to the overall structure of the neural network.

Another example could be the adaptation of a neural network for a new problem within the same domain. For instance, if a neural network was originally designed for sentiment analysis using a Transformer architecture and we wanted to adapt it for a machine translation task, we could simply swap out the Transformer layers with another architecture, such as an RNN with an attention mechanism or a CNN, while keeping the input and output encodings consistent. This interchangeability allows for rapid experimentation with various architectures and can help identify the most suitable architecture for a specific problem.

Dynamic interchangeability can also be applied at the level of individual layers within a neural network. For example, a researcher might experiment with different types of activation functions, such as ReLU, sigmoid, or tanh, within the same architecture to determine which function leads to the best performance. Similarly, various types of normalization layers (e.g., batch normalization, layer normalization) or attention mechanisms (e.g., self-attention, cross-attention) can be interchanged within a single architecture to explore their impact on the overall performance of the neural network.

Though specific neural architectures tend to show better performance in certain categories of problems, there is nothing preventing an architecture of one type from being used for any type of problem. For example, while a transformer network is typically used for large language models, it could be replaced with a convolutional network without sacrificing its ability to perform the task, though it would likely affect efficiency. The general purpose of NAS is to establish methods for quickly determining which neural networks, or combinations thereof, are best suited for a particular problem.

4.2 Applications

Several papers have demonstrated the benefits achieved through NAS in recent years. For instance, the EfficientNet model [15] achieved state-of-the-art accuracy on ImageNet while being

significantly smaller and faster than previous models. The model achieved this by using a novel compound scaling method that uniformly scales the network's depth, width, and resolution.

Another example is the use of Graph Convolutional Networks (GCNs) [16] for graph-based classification problems. GCNs are designed to learn from graphs by performing convolutions on their adjacency matrices. By using GCNs, a model can learn from non-Euclidean data, which is prevalent in many real-world scenarios, such as social networks, biological networks, and recommendation systems.

In a separate study, researchers used a transformer architecture for predicting protein structure from amino acid sequences [17]. The transformer model outperformed other methods by a significant margin, further demonstrating the importance of choosing the appropriate architecture for a given problem. This has implications for various fields, including drug discovery, personalized medicine, and understanding the fundamental mechanisms of life.

NAS has also been applied to speech recognition tasks, as seen in the development of RNN-T [18]. This model, which uses a Recurrent Neural Network Transducer (RNN-T) architecture, has achieved state-of-the-art performance in speech recognition, outperforming traditional systems that rely on separate models for acoustic language modeling.

Most relevant to our project is in the development of AlphaGo and its successor, AlphaZero [19, 14]. In the transition from AlphaGo to AlphaZero, the researchers explored different neural network architectures to enhance the system's ability to learn and generalize across various board games. AlphaGo employed two separate neural networks, the policy network and the value network, both of which were CNN-based. The policy network was used to predict potential moves, while the value network estimated the probability of winning in a given game state. These networks were combined with Monte Carlo Tree Search (MCTS) to effectively explore and evaluate game states.

In contrast, AlphaZero incorporated a single deep neural network architecture with residual connections [20], which consisted of both policy and value heads. This unified architecture enabled AlphaZero to simultaneously predict moves and estimate the probability of winning.

4.3 Implementations

Various methods have been proposed for performing NAS, ranging from gradient-based approaches to reinforcement learning and evolutionary algorithms. These approaches aim to explore the vast search space of possible neural architectures to identify the most effective ones for a given task.

One approach to NAS is Evolutionary Neural Architecture Search (ENAS) [21], which employs evolutionary algorithms to guide the search for optimal architectures. In ENAS, an initial population of neural architectures is created, often randomly. This population then evolves over time through genetic operations such as mutation and crossover, which generate new architectures by combining or modifying existing ones. The fitness of each architecture is evaluated based on its performance on the given task, and the fittest architectures are selected to produce offspring for the next generation. This process continues for a predetermined number of generations, or until a satisfactory architecture is found.

Another NAS implementation involves using reinforcement learning (RL) to guide the search process. In this approach, a controller generates candidate architectures by sampling from a discrete search space. The performance of these candidate architectures is evaluated on the task, and the resulting rewards are used to update the controller through policy gradient methods. This process is iteratively repeated, with the controller learning to generate architectures that lead to higher rewards over time.

Gradient-based NAS methods, such as DARTS (Differentiable Architecture Search) [22], offer an alternative approach by formulating the search problem as a continuous optimization problem. In DARTS, a continuous version of the architecture search space is used, allowing for efficient gradient-based optimization. The architecture is then updated using gradient descent.

Random search is another simple yet surprisingly effective method for NAS. In this approach, random architectures are generated and evaluated, with the best-performing ones being selected as the final solutions. While this method lacks the guided exploration of other NAS techniques, it can serve as a useful baseline and has been shown to perform competitively in some

cases.

4.4 Challenges and Future Directions

One significant challenge in NAS is the computational cost associated with searching and training various architectures. As the search space expands, it becomes increasingly difficult to explore all possible combinations efficiently. Developing more efficient search algorithms and leveraging techniques such as transfer learning and knowledge distillation can help mitigate this issue.

Scaling laws, particularly in the context of large language models (LLMs), provide a valuable approach to predicting the performance of smaller models as they scale up. These laws help identify trends in model performance and resource requirements, enabling more informed decisions when designing and selecting architectures. Incorporating scaling laws into NAS methodologies can further reduce computational costs and enhance the overall effectiveness of the search process.

Another challenge lies in creating better performance metrics to evaluate the suitability of different neural architectures for specific tasks. Existing metrics like accuracy and computational efficiency may not comprehensively assess an architecture's effectiveness. Developing new metrics that consider factors such as robustness, interpretability, and fairness can provide a more holistic evaluation of neural architectures.

Future directions for NAS research may include the development of more efficient search algorithms and exploration of multi-objective optimization techniques, which consider multiple performance criteria simultaneously. This could help identify architectures that strike a balance between competing objectives, such as accuracy and computational efficiency. Additionally, incorporating domain-specific knowledge and constraints into the NAS process may lead to more specialized and effective architectures for specific tasks.

5. METHODS

We implemented a deep reinforcement learning model similar to the one defined in the AlphaGo/AlphaZero. Our iterative process matches that of AlphaZero as it consists of only two steps: self-play and training. We do not include an evaluation step between iterations to choose which model will produce training data for the next iteration. Instead, we augment the most recent model with training data. In parallel, we employ an ELO rating system to synchronously track the generalized performance of the models across iterations.

5.1 Neural Architecture Search

Our research primarily focuses on how the structural differences between neural architectures affect performance metrics. To this end, we constructed several fundamentally different neural architectures and include variations of these to address scaling, such as keeping the type of layers the same while doubling their number. Every neural architecture is passed the encoded input for processing, and the decoding of the policy vector π and the value v remains the same.

Baseline (A1) and A2: The first architecture, referred to as the baseline (A1), is a CNN consisting of four convolutional layers followed by three fully connected (FC) layers. The input to the model is an 8x8 board, with each cell representing the game's state. The model uses 256 channels in the convolutional layers. Each convolutional layer is accompanied by a batch normalization layer, which helps in stabilizing and accelerating the training process.

The first two convolutional layers use a kernel size of 3x3 with stride 1 and padding 1. The last two convolutional layers also use a kernel size of 3x3 but with stride 1 and no padding. These layers aim to capture local patterns and spatial information on the board.

The fully connected layers consist of 1024, 512, and the final action size (64) neurons, respectively. Each FC layer, except the last one, is followed by a batch normalization layer. The model also employs dropout regularization with a specified dropout rate of 0.3 during training.

The A2 architecture is identical to the baseline (A1) architecture, with the only difference

being the number of channels in the convolutional layers, which is increased to 512.

B1 and B2: The B1 architecture is a fully connected neural network without any convolutional layers. It consists of six fully connected layers, with the input being an 8x8 board flattened to a 64-element vector. The FC layers have 2048, 1024, 512, action size (64), 256, and 1 neurons, respectively. Each layer, except the last two, is followed by a batch normalization layer. Dropout regularization is employed during training, with a specified dropout rate 0.3.

The B2 architecture is an extension of the B1 architecture, with more neurons in the first three FC layers. Specifically, the number of neurons in these layers is 4096, 2048, and 1024, respectively. Similar to the B1 architecture, each layer, except the last two, is followed by a batch normalization layer, and dropout regularization is applied during training.

All the architectures use rectified linear units (ReLU) as the activation function for the hidden layers. For the output layers, the models employ log softmax for policy (action probabilities) and tanh for value (estimated game outcome). These outputs are used to guide the agent during training and gameplay.

5.2 Self Play

During a self-play episode, the initial state of the game s_0 will be set to one of two possible states as defined in the introduction, with a 50% chance of being either. We use 25 MCTS simulations and play 100 games for each episode. Othello is symmetric across 7 transformations and we augment our example data with these symmetries. The MCTS exploration parameter λ is set to 1. The temperature τ is set to 1 on the first turn and remains that way until it is set to 0 at turn 25. These measures are to ensure a diversity of training examples. At the end of the episode, all n articles of example data are written to a file as tuples (s_i, π_i^*, r_i) for $i = 1..n$. Figure 6 illustrates this procedure.

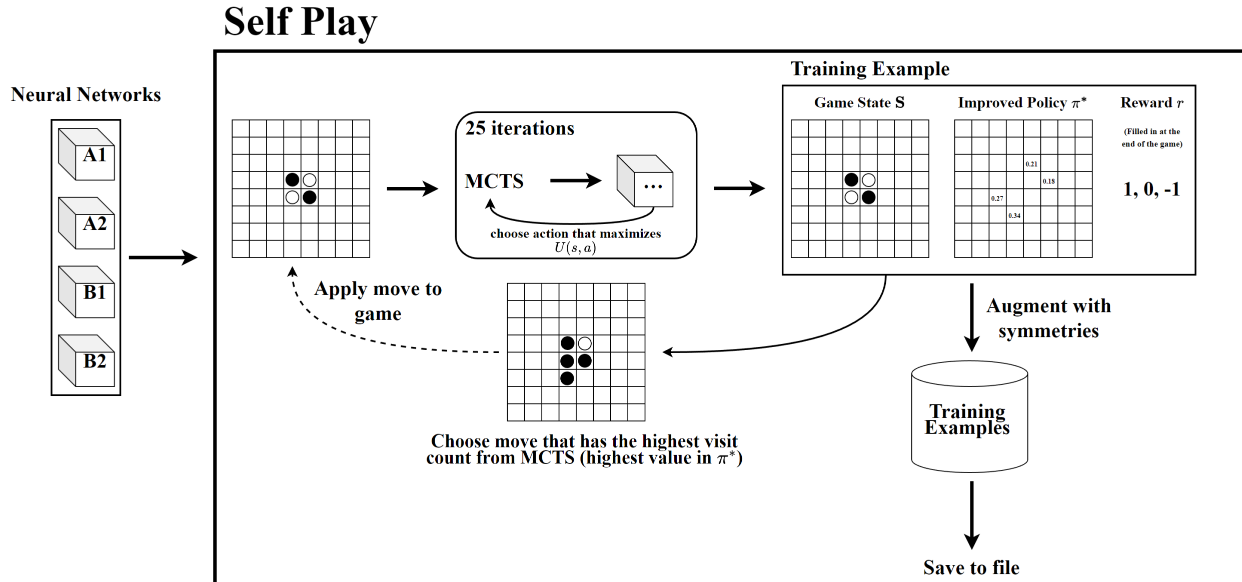

Figure 6: Diagram of the Self Play loop.

5.3 Training

Example data from the most recent self-play episode is loaded and shuffled. We do not remove duplicates resulting from symmetries, since many are concentrated during the first few moves of a game. A queue of 200,000 training examples is kept. When the queue is full, the oldest examples are freed first.

The current neural network f_θ is trained according to the Adam [23] optimizer to minimize the loss function:

$$l = \sum_i (v_\theta(s_i) - r_i)^2 + \pi_i^* \log(p_\theta(s_i)) \tag{9}$$

Training time is set to 10 epochs for all Neural Architectures. Once this is finished, the new Neural Network is saved and the process returns to the self-play routine with the most recent model. Each model iteration is saved individually for later analysis. All models were trained for 160 iterations. Each model a took a different amount of time to complete one full self play and training loop given the respective size and complexity of the model.

5.4 Ranking

We determine the capabilities of each model iteration by employing the ELO ranking system. The ELO system is a popular mechanism for rating relative skill levels in a variety of domains such as chess, tennis, and video games. The ELO system employs a probabilistic model given by the following equation:

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}} \tag{10}$$

Here, E_A represents the expected score for player A, while R_A and R_B are the ELO ratings of players A and B, respectively. After a game between two players, their ELO ratings are updated according to the following formula:

$$R'_A = R_A + K(S_A - E_A) \tag{11}$$

In this equation, R'_A is the updated ELO rating for player A, K is the k-factor (set to 30 in our case), S_A is the actual score of player A (1 for a win, 0.5 for a draw, and 0 for a loss), and E_A is the expected score for player A calculated using equation (10).

When a new model is introduced to the ranking system, it is initialized with an ELO score of 1200. We uniformly select unique models to play each other, without regard for match quality. The chosen models play a sequence of 10 games. The ELO score gain/loss is summed for each of the games in the series and then applied.

In contrast to the self play training phase, during evaluation we set the temperature according to the distribution

$$\tau = 0.65^{(t+1)} \tag{12}$$

where t is the turn number. This ensures that the first move will be chosen pseudo-uniformly and by move 6 the agents will choose moves based on the highest number of visits as usual. By introducing stochasticity into the first few moves of play, this prevents strictly deterministic

evaluation and allows for a broader sampling of the models overall performance.

5.5 Performance Considerations

As stated above, performing this research using consumer-grade computational resources is an important factor. All computation was conducted on a single machine, using an i7-7800k CPU and a GTX 1080 TI GPU. The innately compute intensive nature of ML paired with this computational restriction makes the performance of our implementation critical. Pervasively throughout out research, extra consideration was given to the efficiency of our methodology.

The Othello engine and the AlphaZero implementation were written in C++, and the MCTS for AlphaZero was constructed asynchronously using root node parallelization. The self-play, training, and ranking phases were all run in parallel.

6. RESULTS

We evaluated the performance of the four neural architectures (A1, A2, B1, and B2) by plotting their ELO scores against the number of iterations (Figure 7), as well as considering the training time required for each model iteration. Our analysis revealed that B1 outperformed all other models in virtually every dimension, including having the highest ELO peak, most efficient ELO gain per training time, and best overall performance. B2 closely followed B1, with similar performance trends but consistently lower peaks, and across all iterations both A1 and A2 had lower average performance and longer training times.

We applied the methods of the Bayesian Information Criterion (BIC) to select our linear regression models and decided on a fourth-degree polynomial representation of best fit across the data sets.

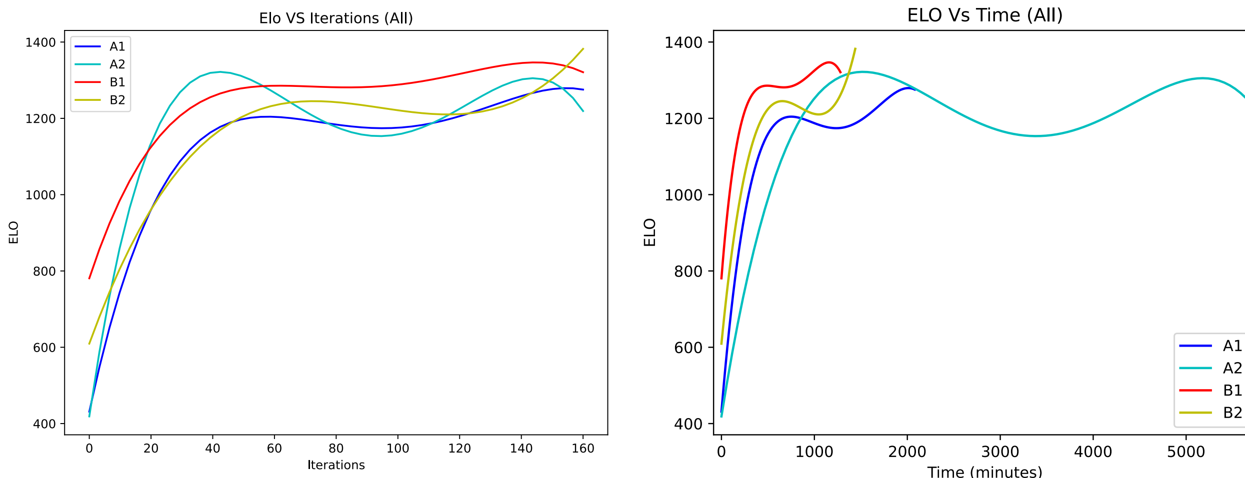

Figure 7: Left: ELO vs. Iterations for all models. Right: ELO Vs Time for all models.

Even a superficial inspection of the training data immediately validates the significance of the research question. A2 performed significantly worse than B1 in every capacity, while taking nearly four times as long to train to the same number of iterative generations. While a relatively

inexpensive mistake on a home desktop, this trial-and-error testing of the A2 architecture would have induced as serious waste of resources if a larger model were trained on a super-computer or computer cluster. Consider the graphs of A2 and B1 (Figure 8).

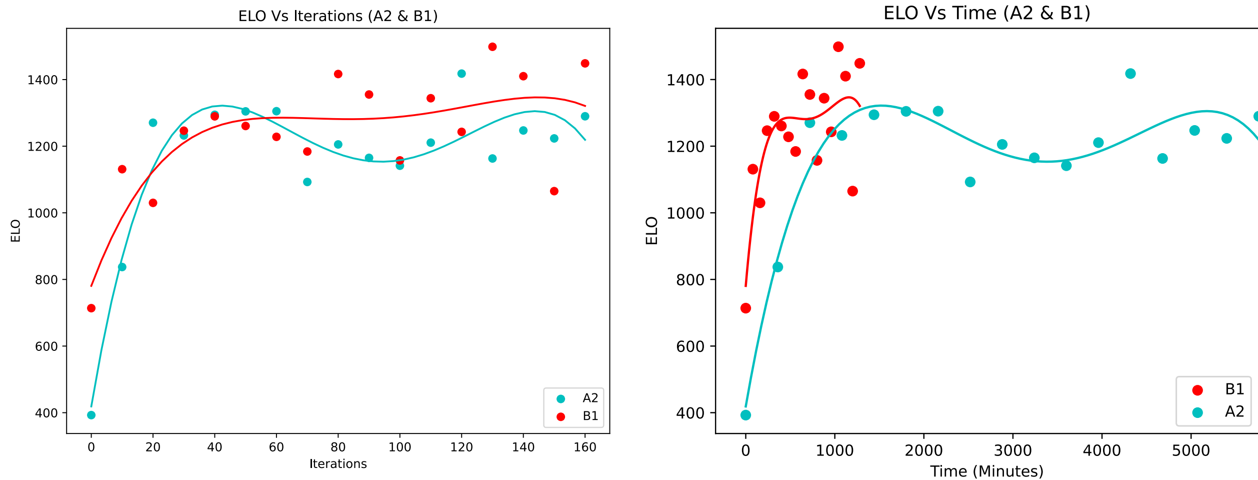

Figure 8: Left: ELO vs. Iterations. Right: ELO Vs Time (A2 and B1).

At early iterations, B1 well outperformed A2. The models then became competitive before B1 again emerged as the prevailing architecture. The discrepancy between their mean performances steadily grew during the training cycle. When factoring in training time, these performance results become even more striking. B1 produced the highest performing model that we encountered in the training of all models (1498) at 1000 minutes into training. The A2 peak, 1418, was not produced until 4500 minutes into training. The resulting argument is clear: when considering finite computational resources, A2 is sub-optimal, and there are no attributes of it that are preferable to those of B1. A comparison of A1 and A2 solidifies this argument (Figure 9).

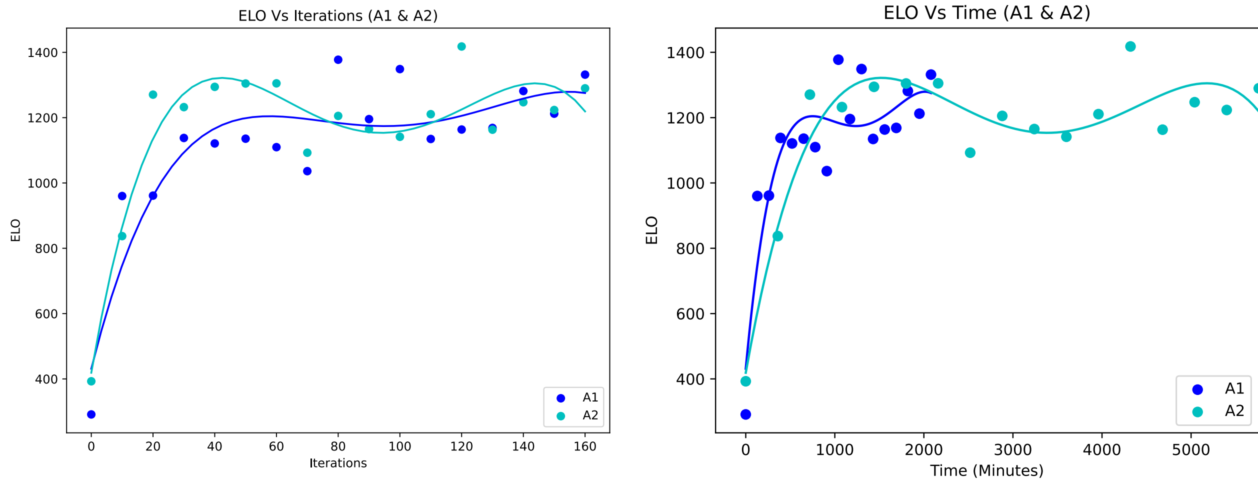

Figure 9: Left: ELO vs. Iterations. Right: ELO Vs Time (A1 and A2).

The performance trends of A1 and A2 per iteration are similar, and it can even be inferred that the scaling of A1 successfully improved the base performance of the architecture type, as the initial iterations of A2 far outperformed A1. A2's 0-20th iterations achieved just below a 1300 ELO rating, while A1's 0-20th iterations were well under a 1000 rating. The subsequent training results, however, demonstrated that while scaling A1 did allow for additional productive detail to be captured, there was not a linear relationship between this additional detail and the resulting increase in computational intensity. A1 averaged 13 minutes per completed iteration, whereas A2 required 36, and the peak performance gain of A2 over A1, 1417 to 1377 respectively, does not justify the resource demand. Further emphasizing this point is the mean performance of A2 to A1, 1146 to 1043, a gain of only 9.9 percent. It is fair to conclude that the scaling of A1 into A2 added enough resolution that performance was notably impacted, but at a disproportionate cost to training time, which is a primary focus of this study.

The results indicate that the CNN approach of A1 and A2 is poorly suited to the problem of Othello, at least in contrast to the FC approach of B1 and B2. (Figure 10) shows the effect of scaling between B1 and B2.

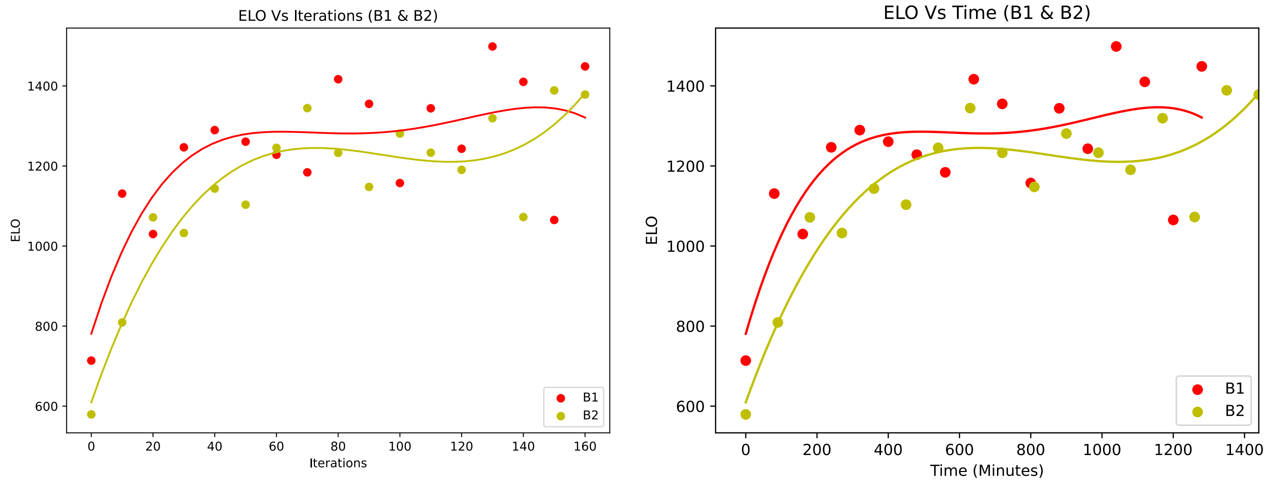

Figure 10: Left: ELO vs. Iterations. Right: ELO Vs Time (B1 and B2).

Contrary to the initialization and early iterations of the scaled A2 architecture, which outperformed the early iterations of A1, B1's early iterations outperformed B2. This suggests that the initial size of FC network in B1 was closer to the optimal scale than B2. The type of comparative performance degradation observed during the training of B2 is typically attributed to overfitting. B1 had a higher mean and higher peak, at 1195 and the aforementioned 1498, whereas B2 only reached a mean of 1122 and a peak of 1388. However, unlike the training time issue encountered during the scaling of A1 into A2, the total minutes of training time for B2 remained comparable to B1, at 1440 to 1280. The scaling of A1 into A2 was successful in producing more detail at the cost of higher training time, but scaling B1 obfuscated the model's interpretation of the encoded data, likely by causing a fixation upon noise and other erratic data unrelated to instructive game-play trends.

With respect to iterations, A2 performed consistently better than A1. Factoring in training time places A1 ahead of A2 as a more attractive model for any context that would involve limited computational resources. B1 outperformed all other models absolutely, both with regard to training efficiency and iterative performance, as shown by the next two sets of figures. (Figure 12) .

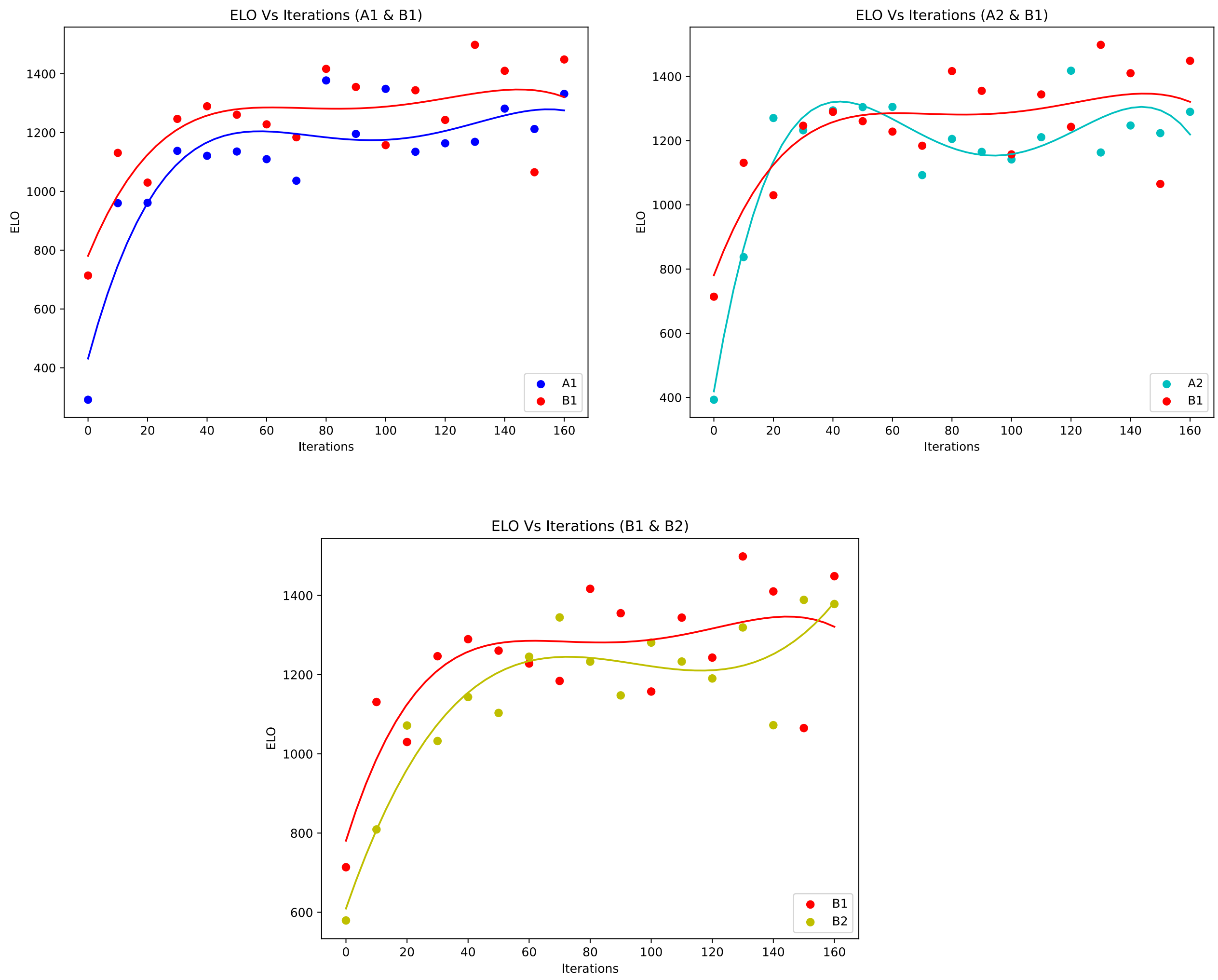

Figure 11: Left: A1 and B1. Right: A2 and B1. Below: B1 and B2 (ELO Vs Iterations).

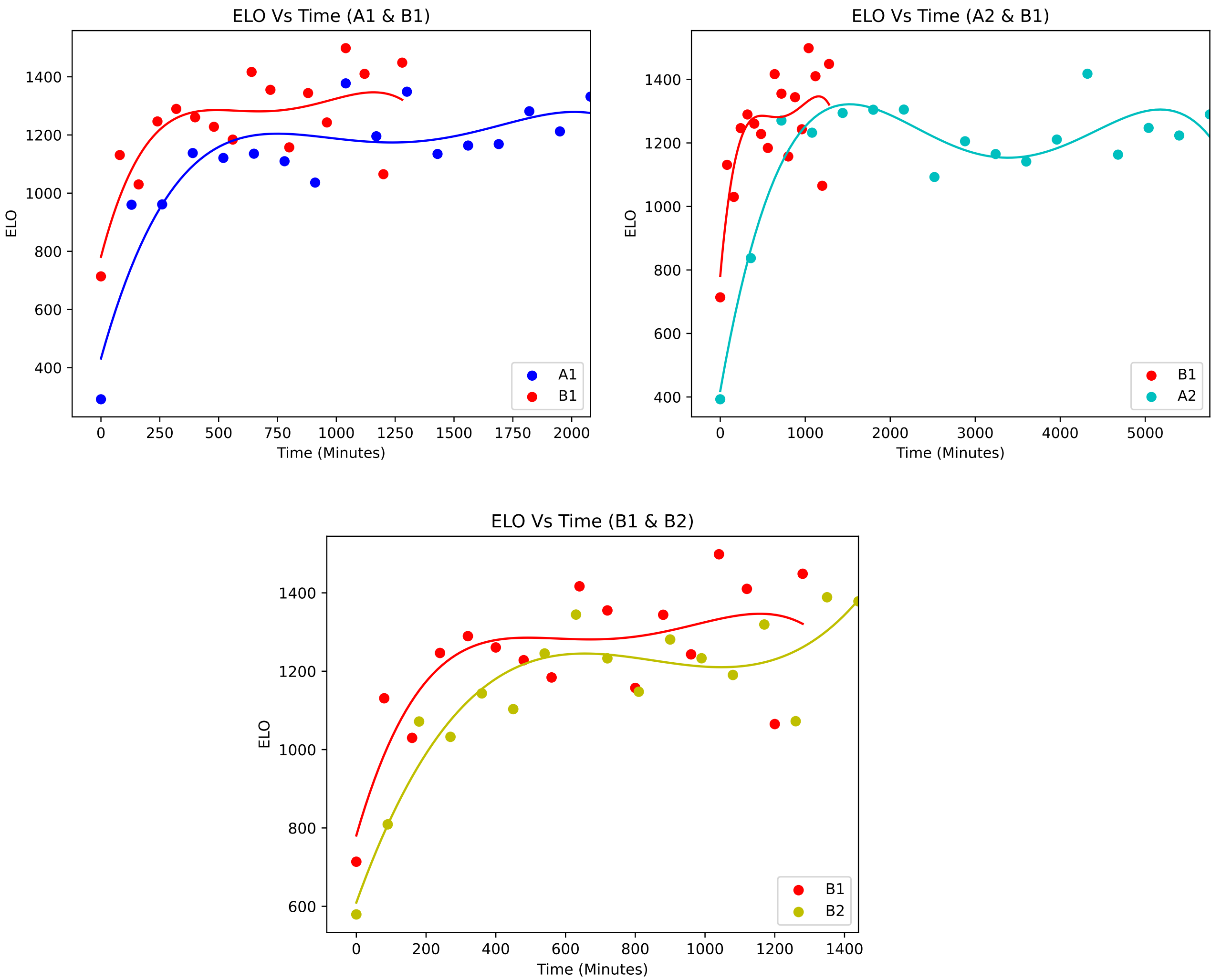

Figure 12: Left: A1 and B1. Right: A2 and B1. Below: B1 and B2 (ELO Vs Time).

The first figure references B1 against all other models symmetrically per iteration, and the second proportionally per time. It is clearly shown that B1 is the best model for the problem of Othello. The results satisfactorily demonstrated that scaling can dramatically affect a model's proficiency, both positively and negatively. In cases where scale is not the bottleneck to a model's performance, such as the case with B1, it has a vastly diminished effect. B1 had the highest mean ELO across its iterations and notably performed the strongest during the early and late stages of training. The performance of B1's 0-20th iterations was also the best of all the models. Taking in

conjunction with all of these dimensions and metrics, it is easy to conclude that the B1 architecture is more structurally optimal for the problem description of Othello.

7. CONCLUSION

In our analysis, we found that fully connected (FC) neural networks (B1 and B2) outperformed CNNs (A1 and A2) across various performance metrics in Othello. B1's fully connected architecture likely enabled it to capture complex patterns in the game state more effectively than the convolutional layers in A1 and A2. Unlike convolutional layers that focus on local patterns and spatial information, fully connected layers can learn intricate relationships between all the board positions.

Although we initially expected B2's increased network size to provide an advantage in learning complex relationships between board positions, the added complexity led to challenges in training, including overfitting. As a result, B2's performance was slightly lower than B1.

A1, the least effective model in terms of average performance, used a relatively simple convolutional architecture. The limited capacity of A1's convolutional layers likely prevented it from learning more complex patterns and relationships on the Othello board, leading to a less effective strategy and its lowered performance.

Lastly, A2, our third-best performing model, benefited from an increase in the number of channels. However, the added complexity was not sufficient to compensate for the inherent limitations of the convolutional architecture in fully understanding the Othello board. The larger model size of A2 and the resulting significant increase in training time rendered this model unsuitable as a viable architecture.

The nature of the performance changes measured when A1 was scaled to A2 is significant. The CNN approach to Othello did not fail completely but rather performed comparatively worse. In the case of 8x8 Othello, capturing enough detail about local interactions likely required too much resolution for the scale of A1. While CNNs excel at capturing spatial information and hierarchical patterns in tasks involving images and complex spatial data, the global structure of the Othello board and the importance of capturing local interactions make fully connected networks

more suitable for play.

In conclusion, the performance increase of A2 over A1 contrasted with the performance of B1 suggests that using a FC NN over a CNN for deterministic state spaces relates to a prioritization of individual pieces over the general board layout. It is predictable that for larger board sizes of Othello, the proportional value of local interactions to general board knowledge would decrease, and CNNs would perform better. This intuition is supported by machine learning attempts at more complex deterministic games, including the original AlphaZero models that play Chess and Go. Therefore, when presented with a particular problem description, the process of determining whether to use a fully connected neural network or a CNN requires an engineer to look at the prioritization of general spatial information versus specific element or piece interactions given by a problem description.

In future research we would consider exploring alternative architectures and approaches, including recurrent neural networks, graph neural networks, transformers, multi-modal architectures, curriculum learning, and different board representations to enhance learning efficiency and develop novel strategies. Additionally, we would conduct a systematic search for optimal hyperparameters, including learning rates, dropout rates, batch sizes, and regularization techniques, to understand their impact on performance.

REFERENCES

[1] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv:1511.08458 [cs]*, 2015.

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, and A. Askell, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[3] K. Ohno and A. Kumagai, "Recurrent neural networks for learning long-term temporal dependencies with reanalysis of time scale representation," in *2021 IEEE International Conference on Big Knowledge (ICBK)*, pp. 182–189, IEEE.

[4] R. C. Staudemeyer and E. R. Morris, "Understanding lstm–a tutorial into long short-term memory recurrent neural networks," *arXiv preprint arXiv:1909.09586*, 2019.

[5] C. Guo and F. Berkhahn, "Entity embeddings of categorical variables," *arXiv preprint arXiv:1604.06737*, 2016.

[6] O. V. et al., "Starcraft ii: A new challenge for reinforcement learning," *arXiv preprint arXiv:1708.04782*, 2017.

[7] B. Driss and T. Cazenave, *Deep Catan*, pp. 503–513. Springer International Publishing, 2022.

[8] T. B. et al., "Decision making in monopoly using a hybrid deep reinforcement learning approach," *arXiv preprint arXiv:2022.3166555*, 2022.

[9] A. N. Warrier and S. Amuru, "How to choose a neural network architecture? – a modulation classification example," in *2020 IEEE 3rd 5G World Forum (5GWF)*, pp. 413–417, 2020.

[10] M. Wistuba, A. Rawat, and T. Pedapati, "A survey on neural architecture search," *arXiv preprint arXiv:1905.01392*, 2019.

[11] J. Feinstein, "The perfect play line in 6x6 othello," Nov 2009.

[12] V. Allis, *Searching for Solutions in Games and Artiicial Intelligence*. Jul 1994.

[13] C. E. Shannon, *Programming a Computer for Playing Chess*, p. 2–13. Bell Telephone Laboratories, Inc., Murray Hill, N.J., 1988.

[14] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[15] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*, pp. 6105–6114, PMLR, 2019.

[16] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[17] A. Nambiar, M. Heflin, S. Liu, S. Maslov, M. Hopkins, and A. Ritz, "Transforming the language of life," *Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, Sep 2020.

[18] T. Makino, H. Liao, Y. Assael, B. Shillingford, B. Garcia, O. Braga, and O. Siohan, "Recurrent neural network transducer for audio-visual speech recognition," 2019.

[19] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[21] 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*, vol. 34, pp. 550–570, feb 2023.

[22] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," 2019.

[23] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.