Exploring the Latest Neural Network Architectural Components in AlphaZero

Bachelor thesis by Tim-Michael Krieg (Student ID: 2794482)
Date of submission: February 7, 2024

1. Review: Prof. Dr. Kristian Kersting
2. Review: M.Sc. Johannes Czech
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Artificial Intelligence and Machine Learning Lab

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, *Tim-Michael Krieg*, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

English translation for information purposes only:

Thesis Statement pursuant to § 22 paragraph 7 of APB TU Darmstadt

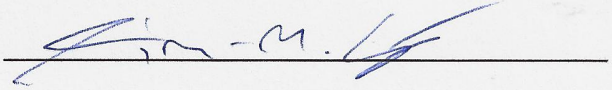
I herewith formally declare that I, *Tim-Michael Krieg*, have written the submitted thesis independently pursuant to § 22 paragraph 7 of APB TU Darmstadt without any outside support and using only the quoted literature and other sources. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I have clearly marked and separately listed in the text the literature used literally or in terms of content and all other sources I used for the preparation of this academic work. This also applies to sources or aids from the Internet.

This thesis has not been handed in or published before in the same or similar form.

I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.

For a thesis of the Department of Architecture, the submitted electronic version corresponds to the presented model and the submitted architectural plans.

Datum / Date: 05.02.2024

Unterschrift/Signature: 

Abstract

Neural network based chess programs have long been inferior in comparison to other engines until the introduction of AlphaZero. The central building block of it's underlying neural network, used for position evaluation in chess, is a dense residual block. As AlphaZero is fundamentally a convolutional neural network (CNN), advancements in domains dominated by CNNs, namely computer vision, yield potential improvements for playing strength in chess - a traditional benchmark in the field of AI.

In our work, we use ProxylessNAS to explore various recent architecture advancements across multiple domains in a *neural architecture search* (NAS) experiment. After finding a truly novel chess architecture, *ChessNAS*, we conduct another search while introducing a latency loss to the original loss function. After training both models from scratch, we not only compare their individual latency during chess-play, but also assess their playing strength in head-to-head matches. While ChessNAS-L does in fact achieve lower latency than ChessNAS, both models perform poorly in comparison to the baseline AlphaZero algorithm. In terms of training metrics and playing strength, ChessNAS-L is considerably worse than the other models. ChessNAS comes close to AlphaZero with regard to the metrics, but still loses all but a fraction of the games against AlphaZero.
Although the search ultimately have produced inferior results, potential improvements might allow for the discovery of better engines to conquer the domain of chess.

Zusammenfassung

Schachprogramme, die mithilfe neuronaler Netze Schachpositionen evaluieren, waren lange Zeit im Vergleich zu anderen Programmen unterlegen. Dies änderte sich mit der Entwicklung von AlphaZero, dessen zentraler Baustein zur Positionsbewertung ein dichter Residualblock ist. Somit ist AlphaZero im Wesentlichen ein faltendes neuronales Netzwerk (CNN). Daher können Fortschritte in Bereichen, die von CNNs dominiert werden, insbesondere in der Computer Vision, mögliche Verbesserungen für die Spielstärke im Schach - einem traditionellen Benchmark im Bereich der künstlichen Intelligenz - bieten.

In unserer Arbeit verwenden wir ProxylessNAS, um verschiedene kürzlich entwickelte Architekturfortschritte mehrerer Bereiche in einem *Neural Architecture Search* (NAS)-Experiment zu erkunden. Nachdem wir eine wirklich neuartige Schacharchitektur, *ChessNAS*, gefunden haben, führen wir eine weitere Suche durch, indem wir einen Latenzverlust in die ursprüngliche Verlustfunktion einführen. Nach dem erneuten Training beider Modelle vergleichen wir nicht nur ihre individuelle Latenz während des Schachspiels, sondern bewerten auch ihre Spielstärke in direkten Duellen. Während ChessNAS-L tatsächlich eine niedrigere Latenz als ChessNAS erreicht, schneiden beide Modelle im Vergleich zum Baseline-Algorithmus AlphaZero schlecht ab. Hinsichtlich der Trainingsmetriken und der Spielstärke ist ChessNAS-L deutlich schlechter als die anderen Modelle. ChessNAS kommt AlphaZero in den Metriken nahe, verliert jedoch alle bis auf einen Bruchteil der Spiele gegen AlphaZero.
Obwohl die Suche letztendlich minderwertige Ergebnisse hervorbracht hat, können potenzielle Verbesserungen es ermöglichen, bessere Programme für die Domäne des Schachs zu entdecken.

Contents

1 Introduction

1.1 Motivation

Chess has been a critical benchmark in artificial intelligence for decades. Before the introduction of *AlphaZero* by Silver et al. [26] in 2019, neural network based approaches were not successful due to their computational overhead. Ever since, numerous engines inspired by AlphaZero were created in order to rival Stockfish[1] and other dominant alpha-beta pruning algorithms. The playing strength of such models heavily depends on the underlying neural architecture. The original AlphaZero architecture is a convolutional neural network (CNN). Thus, the question arises whether recent developments in other domains featuring CNNs also apply to chess. As the introduction of the *Transformer* architecture has shown success in computer vision [30, 9], a field previously dominated by CNNs, the potential for chess is given. But other improvements, including variations of existing convolutional blocks, might also improve the playing strength in chess.
As the search for an improved chess architecture features the exploration of numerous architecture spaces, this is a perfect match for Neural Architecture Search (NAS). The overarching goal of NAS is to find the most suitable neural network architecture for a given problem. As a subfield of automated machine learning (AutoML), it has gained a lot of popularity over the past decade [32]. The methods in NAS allowed for improvements in architecture structures across various domains, including computer vision [32, 11]. Thus, we incorporate NAS to find a truely novel chess architecture, as no one has applied the underlying NAS algorithms to a chess based search space.
Finally, neural network strength and accuracy alone are not sufficient to make assumptions of the underlying playing strength of chess models. As chess engines work under time constraints and competing alpha-beta based models evaluate millions of board positions each move, latency is another valuable metric to consider when designing the neural network architecture. This ultimately outlines our motivation: Exploring recent neural network architecture components using neural architecture search while incorporating latency as a performance indicator and subsequently as a search heuristic.

1.2 Outline

The thesis begins with an overview of the background, featuring explanations of crucial underlying concepts. These include Neural Architecture Search (NAS), TensorRT, and various chess architectures. The exploration of AlphaZero's Residual Block [26, 28], KataGo's Nested Bottleneck Residual Block [33], Mobile Convolutional Block [25], Next Convolutional Block, and Next Transformer Block [16] sets the stage for the subsequent methods.

[1] `https://stockfishchess.org/`, accessed 17 January 2024

In Chapter 3, the methodology is presented, outlining the general procedure of the study. The three key stages—*Search*, *Training*, and *Comparison*—are detailed, summarizing the experimental process. Additionally, the chapter covers other aspects such as the dataset, hardware specifications, and specific parameters employed during training and search.
The Neural Architecture Search using Neural Network Intelligence (NNI) is also introduced in this section, shedding light on the search space, search strategy, and the incorporation of latency considerations in performance evaluation. Model comparison is discussed afterwards.

Chapter 4 features the evaluation of the research. Section 4.1 provides insights into the Neural Architecture Search, presenting latency measurements of individual blocks and revealing the outcomes of the search. Section 4.2 focuses on the training of neural networks, while Section 4.3 concludes the evaluation with the comparison of the trained networks.

Finally, Chapter 5 concludes the thesis, summarizing key findings and their implications.

2 Background

In this section, we will give a brief overview of the underlying concepts, tools and frameworks we used in order to conduct the experiments.

2.1 Neural Architecture Search (NAS)

Neural Networks have become a crucial part in state of the art chess programs [14, 26] and even Stockfish, an engine not based on Monte-Carlo tree search, utilizes `NNUE` [22] - a small neural network originally used in computer shogi - for position evaluation. As such, improving current neural architectures has been investigated by [5]. Therefore, employing neural architecture search (NAS) to discover architectures with the potential for enhancements, both in terms of speed and complexity, appears to be a promising approach.
Originally, NAS first made an appearance as a part of *automated machine learning* (*AutoML*) with the goal of automating all machine learning steps. Nowadays, it shows great potential for finding reliable and scalable architectures suited for old and new problems [11, 32, 19, 37].
Generally, NAS is composed of three main components: the *search space*, *search strategy* and *performance evaluation strategy*.

2.1.1 Search Space

The *search space* composes the set of possible architectures that is searched through during the neural architecture search. This set of architectures can i.e. feature a set of mutations from an underlying *super*-network architecture [19, 3] or be composed of several vastly different architecture types, including CNNs [11] and Transformers. Some spaces feature a huge number of possible variations, with around 10^{20} possible architecture variants [32], while others only contain a very distinct set of hand-picked models. At this point, domain knowledge heavily influences the architectures. While this domain knowledge can effectively filter out some architectures that do not fit to the task, it limits the possibility of finding a truly novel architecture [23].

2.1.2 Search Strategy

The *search strategy* refers to the actual algorithm executed, in order to determine the best performing architecture from the search space. We mainly differentiate between *black-box optimization techniques* and *one-shot techniques* [32].
Black-box optimization techniques generally train and evaluate architectures from the search space individually and compare their performance afterwards to determine the best model. A simple example is randomly choosing architectures from the search space, fully training all of them on a training dataset and finally

comparing their performance on a validation or test dataset, in order to find the best architecture. Denoted as *random search* this search strategy yields surprisingly good results compared to more complex algorithms [17] on image (CIFAR-10) and electrocardiographic (PTB) datasets. This is especially true for smaller search spaces, as employing k iterations will find some of the top $100/k\%$ of the architectures on average [32]. Yet, random search performs poorly on bigger, less engineered search spaces [1]. Zoph and Le introduced *reinforcement learning* as a means to find the best performing architecture by executing a reward-based search on the search space [37]. More sophisticated methods include evolutionary or genetic algorithms, Bayesian optimization and Monte Carlo Tree Search.

As the previously mentioned methods generally involve training up to some thousand architectures from scratch and comparing them afterwards, black-box optimization techniques require immense computational power [37, 17, 19]. In order to avoid training each architecture from scratch, *one-shot techniques* were introduced. One-shot approaches focus on a single network - a hyper- or supernetwork - that includes all architectures from the search space and trains them simultaneously [32]. Here, a *hypernetwork* refers to a network that generates the weights of other networks. It can thus be used to assign optimized weights to architectures from the search space. On the other hand, a *supernetwork* is a network that consists of other networks. All architectures from the search space are represented as subnetworks within the supernetwork [19, 3]. The weights of the individual subnetworks can then simply inherited once training is finished or re-learned in a subsequent training phase.

One of the prominent one-shot techniques in neural architecture search is Differentiable Architecture Search (DARTS) [19]. In DARTS, a supernetwork is represented as a directed acyclic graph (DAG) whose edges contain N operation candidates $\mathcal{O}$. While typical one-shot strategies involve each edge featuring a *categorical* choice over corresponding operations $o \in \mathcal{O}$, DARTS takes a unique approach. In DARTS, each operation o_i is associated with an *architecture parameter* α_i, that is learned and adjusted during the search phase. Now, unlike a categorical choice, DARTS defines a softmax function s_i over the candidate operations and their associated parameter α_i of each edge, as proposed by Liu et al. [19]:

$$\bar{o_i}(x) = \sum_{i=1}^{N} s_i o_i(x) = \sum_{i=1}^{N} \frac{\exp(\alpha_i)}{\sum_j \exp(\alpha_j)} \cdot o_i(x) \tag{2.1}$$

as a means to assign a probability to each candidate, depending on α_i.
Consequently, the task of discovering the optimal architecture within the search space is reformulated as a *dual-optimization problem* [19]. This entails concurrently seeking a) the best architecture and b) the optimal weights for the chosen architecture through alternating gradient descent. Figure 2.1 illustrates this. Specifically, the process involves updating a) the weights of the operator candidates concerning the loss function $L(o)$ and b) the architecture parameters α_i, with the weight updates preceding the adjustments to the architecture parameters.

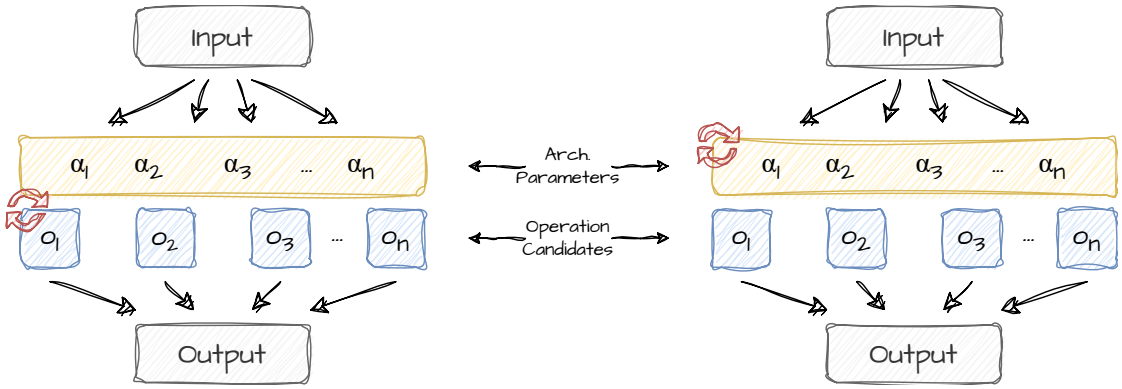


Figure 2.1: Continuous relaxation of candidate operation choice, as proposed by Liu et al. [19]. First, candidate weights are updated according to the loss. Afterwards, architecture parameters α_i are updated as well.

Updating the weights of the candidates simply involves the derivative of the loss function with respect to the corresponding weights:

$$\frac{\partial L(o)}{\partial \boldsymbol{w}_o} = \frac{\partial L(o)}{\partial o(x)} \frac{\partial o(x)}{\partial \boldsymbol{w}_o} \tag{2.2}$$

while updating the architecture parameters α follows a similar pattern:

$$\frac{\partial L(o)}{\partial \alpha} = \frac{\partial L(o)}{\partial s} \frac{\partial s}{\partial \alpha}. \tag{2.3}$$

2.1.3 Performance Evaluation Strategy

Lastly, the *performance evaluation strategy* defines the evaluation method in order to provide a comparison between two architectures [32]. In order to obtain an evaluation of a network, one could fully train and run an evaluation on the fully trained network. As that would require great computational costs, an architecture is typically evaluated during the training process.
The actual evaluation function highly depends on the problem at hand [32]. Generally, performance evaluation strategies try to predict the actual evaluation using an estimation. Those include learning curve extrapolation [8] or zero-cost proxies [20]. But other methods that can give some evaluation metric during the training process also apply.

2.2 TensorRT

Chess engines operate under time constraints to compare performance, demanding fast inference and low latency for position evaluations. While Stockfish, an alpha-beta search engine, evaluates around 80 million positions, modern CNN and MCTS-based engines evaluate mere thousands per second [26]. Due to their millions of parameters contributing to evaluations, inference time on these large models is rather slow. In

order to ensure competitive relevance, optimizing inference speed thus becomes a critical aspect in boosting performance.

CNN-based chess engines, characterized by repetitive architectures and parallelizability, benefit from efficient GPU utilization [15]. NVIDIA's TensorRT[1] framework offers a series of optimizations on pre-trained neural networks in order to addresses these needs:

1. **Precision Reduction:** TensorRT reduces the typical float32 precision of weights and activations in PyTorch models to int8 while preserving high accuracy for deep neural networks [36].
2. **Layer Fusion:** Recognizing the repetitive structure of CNNs, TensorRT fuses routine computations like convolution + normalization + ReLU, or entire convolutional blocks, further optimizing network performance.
3. **GPU-Centric Optimizations:** TensorRT customizes GPU utilization, optimizing algorithm choice, dynamic memory allocation, and leveraging CUDA for parallelization of input streams, ultimately maximizing evaluation efficiency.

TensorRT is widely used across domains to increase the evaluation speed of machine learning models [34, 36, 5].
Zhou and Yang's paper [36] advocates for ONNX[2] to TensorRT Conversion as the most effective method, offering superior throughput, reduced latency, and efficient GPU memory utilization. This approach thus ensures maximal efficiency in evaluating trained models.

2.3 Chess Architectures

The game of chess has long posed a problem for artificial intelligence in it's history. Deep Blue [4], a chess playing engine based on an IBM supercomputer, was the first computer to win a game against current world champion Garry Kasparov. Deep Blue utilized the most iconic algorithm to successfully play chess: Minimax, or more specifically, it's alpha-beta pruning optimization.
Up until the end of 2017, neural networks have largely been ignored in the domain of chess, as they were inferior in terms of latency, elo rating and overall playing strength in comparison to highly optimized alpha-beta pruning variants. This was completely flipped around, as work by Silver et al. from Google's DeepMind proposed a chess engine - *AlphaZero* - with a fundamentally different approach [26]: Instead of relying on an alpha-beta pruning algorithms to crunch millions of board positions a second, AlphaZero uses a deep neural network for position evaluation combined with Monte-Carlo Tree Search (MCTS) to search for the best move. This approach, although only evaluating some thousand subsequent moves a second, could outperform state-of-the-art chess engines [26].
From that point on, the popularity of neural networks in chess engines has only grown. Even Stockfish, the best chess engine to date, adopted a neural network to make a position evaluation in 2020. Here, the engine incorporates efficiently updatable neural networks (NNUE) first introduced by Nasu for Shogi [22]. This network follows a more *shallow* architecture, only featuring 4 fully connected layers of which the first is highly over-parameterized with $\approx$ 10 million weights, taking a (Half)King-Piece encoded board representation as the input.
While NNUEs generally perform well in combination alpha-beta pruning based search algorithms [22], due to

[1] `https://developer.nvidia.com/tensorrt`, accessed 03 February 2024
[2] `https://onnx.ai/`, accessed 03 February 2024

their effective and fast computation on a CPU, we omitted them from the search. This is because we would have to change the input representation fed into the network based on what initial layer is used in the search process. As we use the AlphaZero architecture as a baseline, the input representation is drastically different [26, 5] (see table 3.1 for reference) to the (Half)King-Piece encoding [22]. Furthermore, we use the previously mentioned TensorRT framework for efficient GPU utilization. This also contributes to the exclusion of NNUE, as it is optimized for CPU usage.

As previously mentioned, the neural architecture proposed by Silver et al. [28, 26] is fundamentally a convolutional neural network (CNN) architecture. This implies: Advancements in the field of computer vision, a domain dominated by CNNs, ultimately also affect research in the domain of chess engines. One such advancements is the *Transformer* architecture first introduced by Vaswani and colleagues in [30]. Here, the authors completely omit recurrence and convolutions in favor of the underlying attention mechanism and put more focus on self-attention. This ultimately enabled a key feature of the transformer architecture: The usage of global information. Initially constructed as a translation model, the transformer architecture quickly dominated the field of Natural Language Processing (NLP) but also made it's way to the field of computer vision [9, 35]. Most notably, in [9] the authors utilize the same transformer architecture as initially proposed in [30] and apply it to embedded (16 by 16 pixel sized) patches of images. As such, it seemed like the transformer architecture should be the perfect addition to the domain of chess, as global information could - in theory - improve playing strength beyond what is capable with CNNs, as they have more *local* information extraction. But is changing the original CNN-based architecture to a transformer worth it for chess engines? Building upon the idea of using hybrid strategies [16], Czech et al. [5] show that using a transformer based hybrid architecture does not necessarily improve playing performance of chess engines. Although great efforts have been made to address the issue of latency [18, 16, 34], faster hybrid architectures do not seem to increase playing strength for chess. However, the developers of Leela Chess Zero (Lc0)[3] - an engine inspired by AlphaZero - are investigating the extend to which at least attention mechanisms in general can be applied to chess.

2.3.1 AlphaZero Residual Block

AlphaGo was the first truly successful integration of a Monte-Carlo Tree Search (MCTS) algorithm in combination with a neural network architecture to play the game of Go at human expert level [27]. Since then, the authors first changed the supervised learning paradigm of AlphaGo to starting *tabula rasa* while using reinforcement learning from self-play and then subsequently apply the same principle to both chess and shogi in a follow-up paper [28, 26].
AlphaZero takes the current board configuration (see table 3.1 for reference) as an input and returns both a win probability and a probability distribution over subsequent moves. At the heart of it's convolutional network lie 19 (or 39) dense residual blocks [10] whose depiction can be seen in 2.2.

[3] `https://lczero.org/`, accessed 17 January 2024

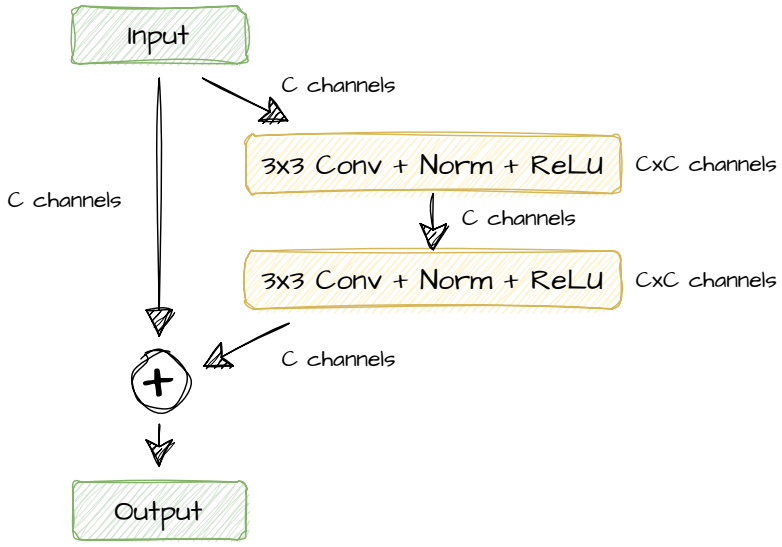


Figure 2.2: A dense residual block [10] as applied in AlphaGo Zero [28] and subsequently in AlphaZero [26]. In both cases, 19 or 39 of these blocks were stacked sequentially and $C = 256$ channels were used.

2.3.2 KataGo: Nested Bottleneck Residual Block

In the appendix of Danihelka et al.'s [6] work on the application of the Gumbel-Max trick [31] as a policy improvement algorithm for AlphaZero, the authors mention a modification made to the original AlphaZero network architecture. Here, the dense residual blocks (2.2) are replaced by *bottleneck* residual blocks [10]. In bottleneck residual blocks, the number of channels is first reduced by a 1x1 convolution. The result is then parsed through a standard 3x3 convolution with the expensive bulk of computations, followed by another 1x1 convolution which increases the channels to the original amount. This decrease in computational complexity within the bottleneck block allows for an increase in the number of channels or in the total number of blocks. In [6] the authors simply half the number of channels in the bottleneck block.
The Go engine KataGo [33, 7] further changes this modification by stacking more 3x3 convolutions in between the bottleneck 1x1 convolutions, resulting in the block architecture seen in 2.3. This change is made, as 1x1 convolutions still produce large computational costs and a single 3x3 convolution at a reduced dimension will not pay back enough of it. Thus, they argue, introducing more convolutional layers with added residual connections might improve the performance in comparison to a standard (dense) residual block [7].

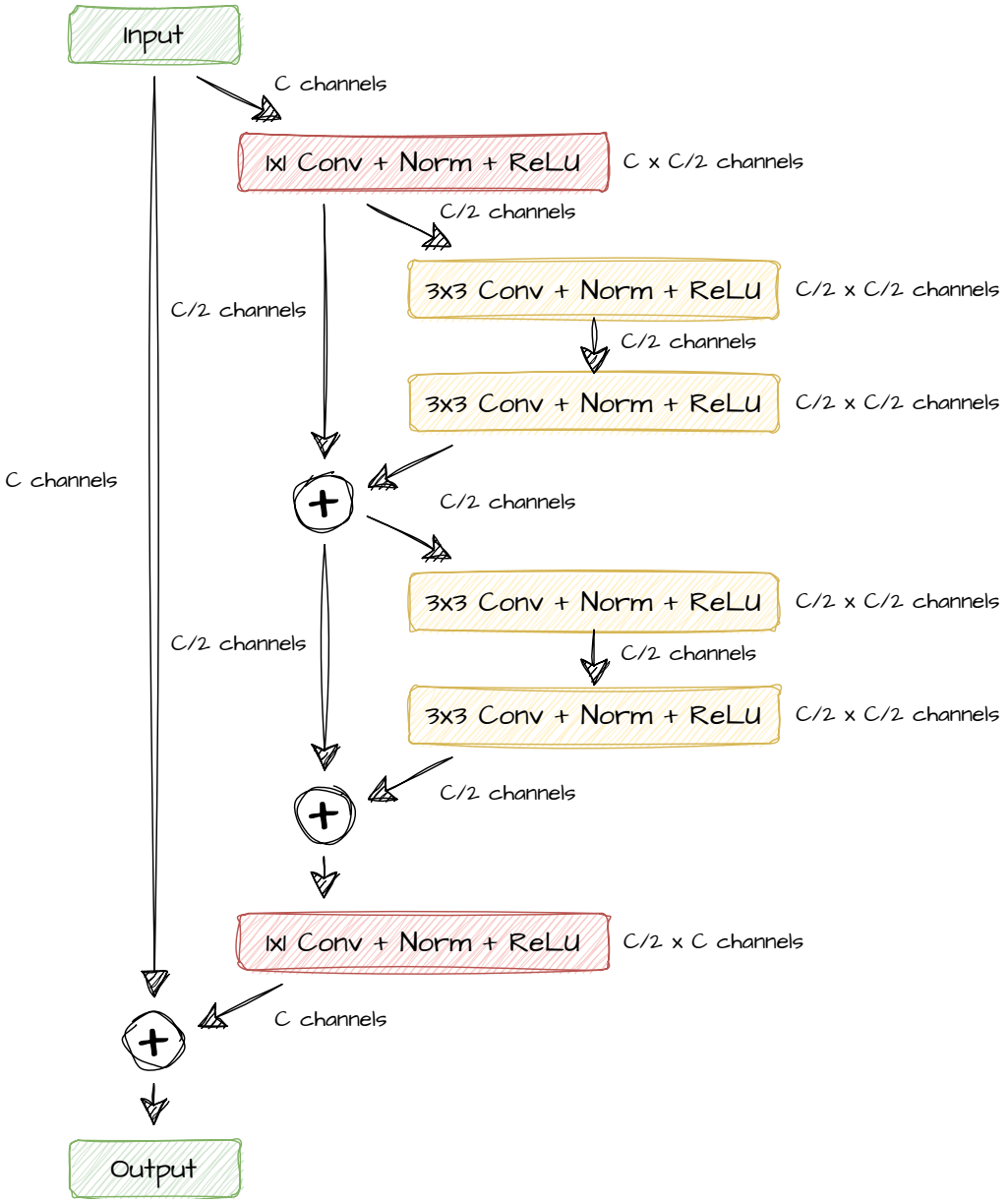


Figure 2.3: The Nested Bottleneck Residual Block (NBRB) inspired by Danihelka et al's work and modified to use in KataGo [33, 6, 7]. Architecture was proposed in seperate KataGo methods on Github (link) and adopted in KataGo as of 2022/23 in their "b18c384" variant.

2.3.3 Mobile Convolutional Block

Since convolutional neural networks (CNN) in computer vision have gained a lot of popularity over the years, the demand for models that run on smaller scaled platforms (such as drones, etc.) has increased. Larger scaled CNNs typically perform better [24], requiring more computational power as a consequence. However, there has been numerous attempts to create more efficient architectures that can also perform well on mobile devices. The works of Howard et al [12], namely MobileNets, not only achieve a smaller overall architecture, but also less latency, while preserving accuracy compared to a dense residual block (2.2) [10, 25]. This

perfectly fits into the domain of chess, which is heavily reliant on this performance indicator. Subsequent work by Sandler et al. and Howard et al. build upon the MobileNet architecture, which is based on depthwise seperable convolutions, and include linear bottlenecks with inverted residuals [25, 11]. An illustration of the architecture dubbed the “Mobile Convolution Block” (MCB) can be seed in 2.4.

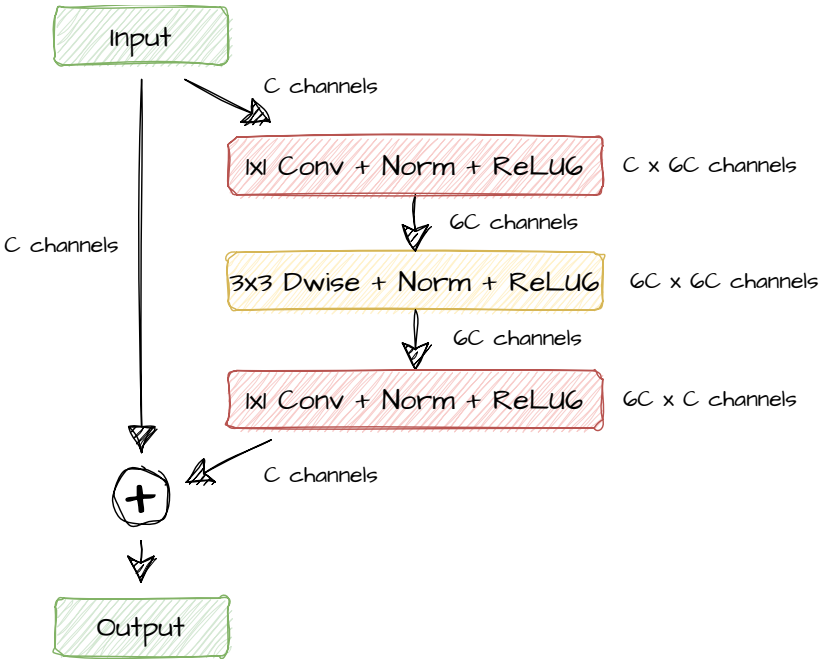


Figure 2.4: The mobile convolutional block as introduced in MobileNetV2 and later reutilized in MobileNetV3 [25, 11].

2.3.4 Next Convolutional Block

Following the MCB from the previous section, we include yet another block architecture that is based on convolutions. In [16], Li et al. introduce the Next Convolution Block, as a modification of the MetaFormer architecture [35]. This block features it’s own Multi-Head Convolutional Attention (MHCA) module, who’s purpose is to implement a token mixer proposed in the MetaFormer architecture. The MHCA module is followed by a Multi-Layer Perceptron (MLP) in order to finish the transformer-inspired adaptation. The full architecture can be seen in 2.5.

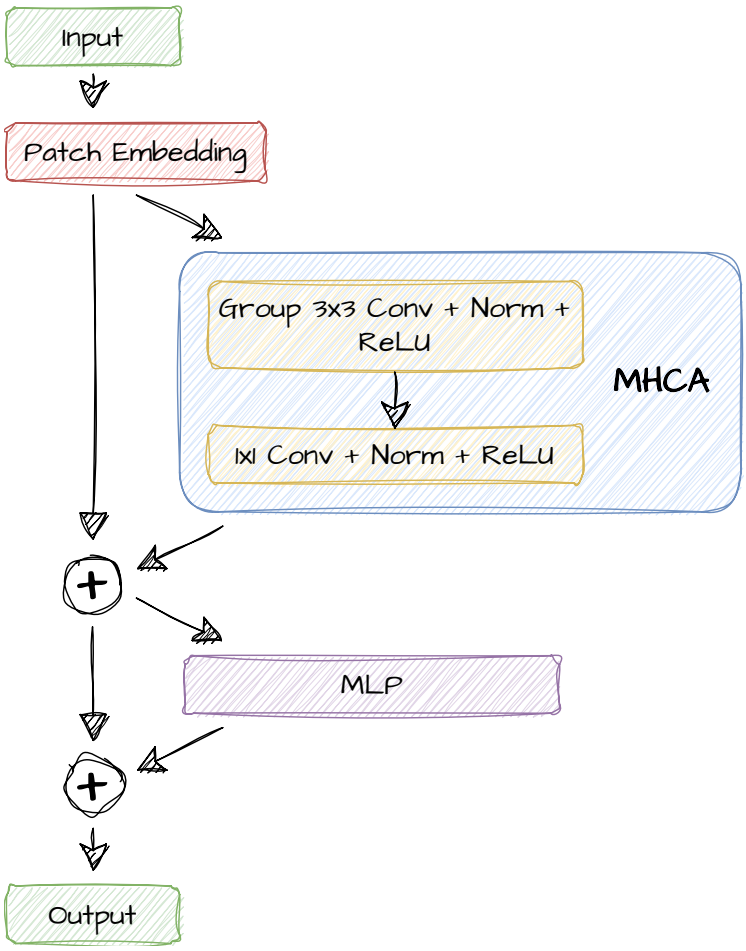


Figure 2.5: Next Convolutional Block (NCB) as introduced by Li et al. [16]. The block features a Multi-Head Convolutional Attention (MHCA) module to act as a token mixer [35] and a Multi-Layer Perceptron (MLP). As opposed to previous block architectures, channel sizes are constant and thus omitted from this illustration for clarity.

2.3.5 Next Transformer Block

As mentioned in the introduction to this section, transformer-inspired architectures have not been hugely successful in MCTS and neural network based search algorithms in chess [5]. Full blown transformer architectures also perform considerably worse than hybrid strategies. This is also due to their increased latency in comparison to other, CNN based components [25, 11, 18]. Especially the evaluation of a given board position suffers from increased latency, as fewer positions can be evaluated in total. As such, strictly using transformer blocks as opposed to the standard residual block 2.2 utilized in AlphaZero will inevitably lead to worse performance in comparison to other models.
However, one can not ignore the massive spike in attention these models have gained over the past years. Recent work also addressed the issue of latency by better utilizing TensorRT or changing components of the architectures in order to obtain comparable latency to convolutional networks [34, 16]. Further, hybrids that use both transformer blocks and convolutional blocks have gained more popularity [16, 18]. As such, we are

going to include the Next Transformer Block (NTB), as introduced in work by Li et al [16] and illustrated in 2.6, in our search.

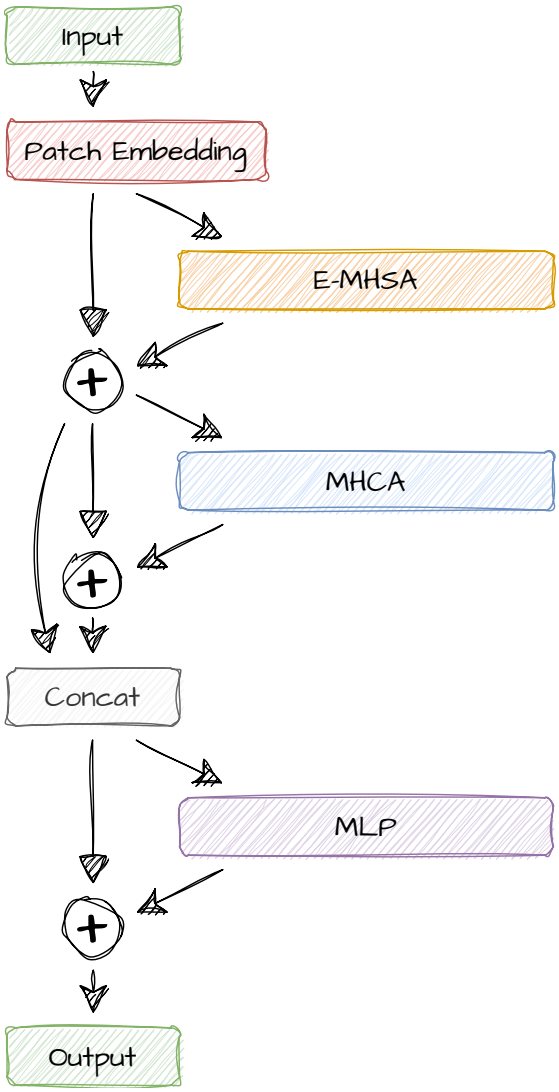


Figure 2.6: Next Transformer Block (NTB) as introduced by Li et al [16]. As opposed to other blocks, channel sizes are constant and thus omitted from this illustration for clarity.

3 Methodology

Here we will outline what components played a major part in our experiment and how we integrate key concepts previously mentioned.

3.1 General Procedure

The experiment comprised distinct stages, delineated as follows:

3.1.1 Search Stage

In the *search stage*, a Neural Architecture Search (NAS) was executed on AlphaZero, encompassing all architectures introduced in Section 2.3. The goal was to identify an enhanced version of AlphaZero incorporating more recent architectures. Initially, NAS was conducted without considering latency, followed by a subsequent iteration that included latency considerations (as detailed in Section 3.4.3), enabling a direct comparison.

3.1.2 Training Stage

Following the completion of NAS, all discovered models, along with a baseline AlphaZero version, underwent training from scratch. Upon the conclusion of the training process, the models were exported in ONNX format to facilitate the last stage.

3.1.3 Comparison Stage

The final models were then optimized for inference using TensorRT [36] and subjected to a conclusive *comparison stage*. First, models were tested for inference speed (that is, nodes per second and latency) and subsequently, this stage also entailed a round-robin tournament involving all competing models, providing a comprehensive assessment of their relative performance and elo ratings.

3.2 Dataset

For both the Neural Architecture Search (NAS) and the subsequent training of the discovered architectures, we used the KingBase Lite 2019 dataset[1]. This dataset consists of over 1 million chess games played by individuals with an ELO rating of 2200 or higher and spans the period from 2000 to the release date. Specifically for the NAS portion, a subset of approximately 48 thousand games was utilized to find the best architecture. This was done in order to reduce search time. Conversely, the entire dataset was employed for training AlphaZero and the best models found during NAS.
Partitioning the subset used for NAS into training and validation subsets was not necessary, as the implementation of ProxylessNAS by NNI [21] does not use validation steps by design. For the training portion however, we excluded a subset of the entire dataset for validation.

3.2.1 Input Representation

The original board position representation used for AlphaZero in [26] utilizes 8×8 planes, each conveying specific aspects of the board configuration. These planes include boolean or integer values, capturing information about piece locations, persisting castling rights, last moves played and more.
In a recent study by Czech et al. [5], the input representation was expanded with additional features (see table 3.1 for reference). These include opposite color bishops and material differences, while excluding color and move count features. Notably, the authors observed the significance of two boolean planes with masks for each player's pieces, demonstrating their high average importance in model evaluation. Moreover, the modified representation led to an actual improvement in playing strength compared to the original representation in [26]. Hence, we adopt this enhanced representation as the input for our neural networks.

3.3 Hardware and Parameters

3.3.1 Hardware and Software

Both, the neural architecture search and training of individual networks was performed on NVIDIA `V100` GPUs. For the search process, 4 GPUs were used in parallel, while for training a single network only was GPU was utilized. All GPUs ran CUDA version 11.4. Additional information about software and frameworks can be seen in table 3.2.

[1] `https://archive.org/details/KingBaseLite2019`, accessed 18 January 2024

Feature	Planes	Type	Comment
P1 pieces	6	bool	order: {PAWN, KNIGHT, BISHOP, ROOK, QUEEN, KING}
P2 pieces	6	bool	order: {PAWN, KNIGHT, BISHOP, ROOK, QUEEN, KING}
Repetitions*	2	bool	how often the board positions has occurred
En-passant square	1	bool	the square where en-passant capture is possible
~~Colour~~*	1	bool	all zeros for black and all ones for white
~~Total move count~~*	1	int	integer value setting the move count (UCI notation)
P1 castling*	2	bool	binary plane, order: {KING_SIDE, QUEEN_SIDE}
P2 castling*	2	bool	binary plane, order: {KING_SIDE, QUEEN_SIDE}
No-progress count*	1	int	sets the no progress counter (FEN halfmove clock)
Last Moves	16	bool	origin and target squares of the last eight moves
is960*	1	bool	if the 960 variant is active
P1 pieces	1	bool	grouped mask of all P1 pieces
P2 pieces	1	bool	grouped mask of all P2 pieces
Checkerboard	1	bool	chess board pattern
P1 Material difference*	5	int	order: {PAWN, KNIGHT, BISHOP, ROOK, QUEEN}
Opposite color bishops*	1	bool	if they are only two bishops of opposite color
Checkers	1	bool	all pieces giving check
P1 material count*	5	int	order: {PAWN, KNIGHT, BISHOP, ROOK, QUEEN}
Total	39 / 52		

Table 3.1: Modified input representation as described in work by Czech et al. [5] and used for the experiments.

Hardware / Software	Description
GPU	NVIDIA Tesla V100 SXM3
GPU Driver	Version: 470.141.03
CUDA	Version: 11.4
Training Backend	NAS: pytorch-lightning-2.0.9.post0 Training: PyTorch-1.12.0a0+8a1a93a
Evaluation Backend	TensorRT-8.2.5.1 (tag 22.05[2])
NAS Framework	Microsoft NNI v3.0
CrazyAra Commit	83dfacf

Table 3.2: Hardware and Software used in the search and training process. Note that the architecture search was performed on 4 GPUs simultaneously, while training a single network was done on a single GPU.

[2] https://github.com/NVIDIA/TensorRT/releases/tag/22.05, accessed 24 January 2024

3.3.2 Training and Search Settings

The *training stage* encompassed two main aspects: a) training the architectures discovered during the search stage, and b) training the AlphaZero architecture for comparative purposes. The entirety of the KingBaseLite dataset was employed for both scenarios. Each model underwent training from scratch for 7 epochs, utilizing a batch size of 1024 games. The decision to halve the batch size, in comparison to the work by Silver et al. [28], aimed to accommodate larger models within memory constraints.

The increased number of epochs during the *search stage* was chosen to facilitate the selection of optimal operations for each layer. The decision to halve the batch size, in comparison to the training stage, was made to address limitations in memory capacity. As such, the learning rate was also adjusted accordingly.

Moreover, various additional parameters related to training the search space or individual models were carefully considered. For detailed insights into these parameters, please refer to the configuration files available in the project repository[3]. A list of differing parameters can be found in the appendix (1).

3.4 Neural Architecture Search using NNI

In order to efficiently incorporate neural architecture search (NAS) into the `CrazyAra`[4] chess framework, we make use of the Neural Network Intelligence (NNI)[5] toolkit by Microsoft [21]. This is because a) the framework effectively implements all necessary components of NAS for our use case and b) CrazyAra (as of December 11th 2023) mainly supports PyTorch[6] as the machine learning framework, which is fully supported by NNI.

3.4.1 Search Space

The standard AlphaZero neural network architecture for position evaluation, following AlphaGo Zero [26, 28], features a residual tower followed by two seperate *heads*. The residual tower starts with a single convolutional block (denoted as *stem*), featuring a 3×3 convolution with batch-normalization and rectifier non-linearity. This *stem* is followed by 19 dense residual blocks (2.2). Lastly, two *heads* are applied to the final output of the tower: A policy head, which outputs a (logit) probability distribution $\boldsymbol{p}$ over the subsequent moves, and a value head, that outputs a scalar position evaluation v in the range of $[-1, 1]$. Developers from Lc0 introduced a third head, the *moves left head*, whose purpose is to potentially prevent random moves and apply dynamic movetime allocation in tournaments. We are omitting this head, as it is not totally clear to what extend it is useful [14].

Generally, in the context of this experiment, we are only substituting the 19 intermediate blocks by architectures previously mentioned in section 2.3. In terms of the number of channels, we adopt the 256 channels proposed by Silver et al. and keep them consistent over all block choices.
In order to incorporate the different architectures into our experiment, we utilize the `LayerChoice` feature of NNI. Here, a `LayerChoice` implements the operation choice featured in a supernet-layer. All blocks

[3] `https://github.com/QueensGambit/CrazyAra/tree/master/DeepCrazyhouse/configs`, accessed 24 January 2024
[4] `https://github.com/QueensGambit/CrazyAra`, accessed 18 January 2024
[5] `https://github.com/microsoft/nni`, accessed 18 January 2024
[6] `https://pytorch.org/`, accessed 04 February 2024

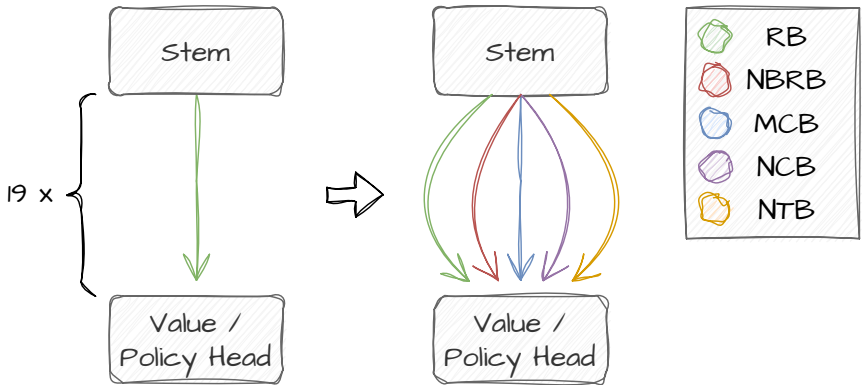


Figure 3.1: Illustration of the search space. The original AlphaZero [28, 26] architecture is displayed on the left with 19 dense residual blocks (2.2) in between the *stem* and the *heads*. The search space with 19 $\times$ `LayerChoice` of blocks proposed in 2.3 is shown on the right.

referenced in section 2.3 are implement in PyTorch and added to the search space by wrapping them inside a `LayerChoice`. As such, during the actual search process, all 19 blocks get assigned a specific candidate. An illustration of the search space can be seen in 3.1.

3.4.2 Search Strategy

In the background section, we introduced various *black-box optimization* and *one-shot* search strategies. While the former provide many advantages, such as robustness, easier parallelism and others [32], they require immense computational resources [37, 17, 19]. This is, because algorithms from this category usually train many networks from the search space before comparing their performance. As our search space yields $5^{19} \approx 19$ trillion different combinations, we can not afford to train even a fraction of these potential architectures from scratch. Thus, we will apply a one-shot strategy.

The most popular one-shot strategy is Differentiable Architecture Search (DARTS) [19], due to its simplicity concerning the supernetwork approach [32]. However, a heavy drawback of DARTS is the high memory consumption during the search process. As a supernetwork typically consists of many smaller networks, keeping all candidates and associated weights in memory is nearly impossible for larger search spaces. For example, the softmax over all N candidate operations in 2.1 requires all of their weights during computation, resulting in roughly N times the memory and time cost. As the training batch also needs to be stored in memory during each training step, even modern GPUs can not store all of it during run-time.
Furthermore, as with nearly all one-shot NAS algorithms, the problem of *rank disorder* arises. Usually, one-shot algorithms make the assumption, that the ranking of the architectures within the supernetwork is consistent with the ranking one would obtain by training the architectures independently. If this assumption is not met, the algorithm suffers from rank disorder. While one could argue for and against rank disorder in DARTS, more recent advancements in one-shot strategies actively address this issue.

In order to mitigate these unwanted drawbacks, while preserving the effectiveness of the search, we instead apply a ProxylessNAS search as proposed by Cai et al. [3]. This alteration of DARTS addresses the memory issues by applying *path binarization* to the training of the network. Instead of strictly utilizing the proposed softmax s over all candidates (2.1), the authors introduce binary gates for each path:

$$g = \text{binarize}(s_1, \dots, s_N) = \begin{cases} [1, 0, \dots, 0] & \text{with probability } s_1, \\ & \dots \\ [0, 0, \dots, 1] & \text{with probability } s_N \end{cases} \tag{3.1}$$

which in turn influences the operator choice:

$$\bar{o}_i(x) = \sum_{i=1}^{N} g_i o_i(x) = \begin{cases} o_1(x) & \text{with probability } s_1, \\ & \dots \\ o_N(x) & \text{with probability } s_N. \end{cases} \tag{3.2}$$

The binarized version of the probability distribution over operator candidates thus reduces the computation at run-time to a compact model. Conversely, using DARTS would yield an increase in memory consumption proportional to N.
While down-scaling the model or learning a single block that is repeated a number of times address the issue of rank disorder, this ultimately reduces the potential of the underlying search space. As ProxylessNAS effectively reduces the computational needs during the search process to searching a compact model, the strategy removes the need to alter the search space while also reducing the chance of *rank disorder* in comparison to DARTS and other one-shot search strategies.

Just like in DARTS, instead of exploring every possible combination of `LayerChoice` options, the previously defined search space over all architecture modifications is relaxed in a continuous manner. Every operator within the `LayerChoice` is assigned an architecture parameter α and sampled according to equations 3.1 and 3.2. Updating architecture weights is equivalent to 2.2, while updating α changes slightly in comparison to DARTS (equation 2.3). In ProxylessNAS, only two candidates are sampled randomly and used for calculating the softmax s, instead of using all N candidates. An illustration can be found in figure 3.2.

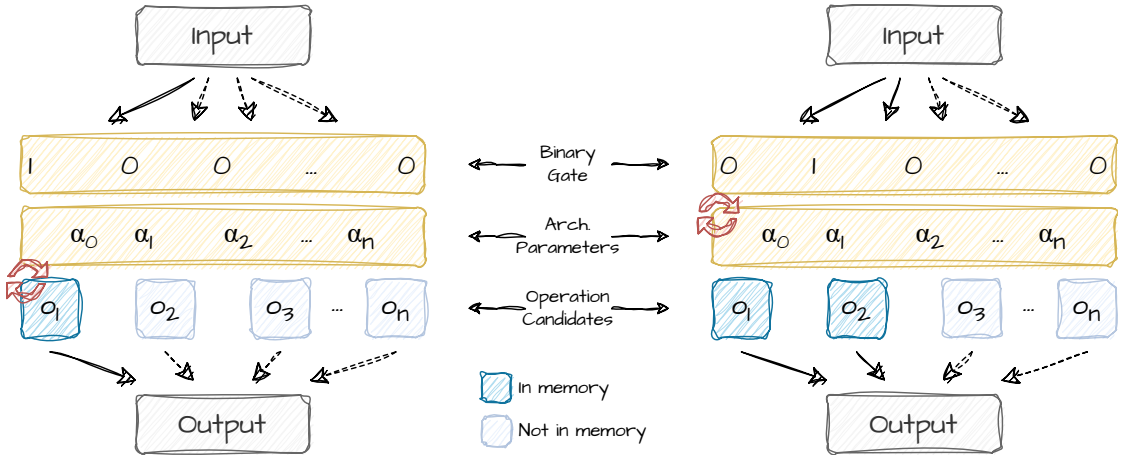


Figure 3.2: Binary modification to the continuous relaxation in DARTS [19], as proposed by Cai et al. [3]. In comparison to 2.1, only two candidates (including weights) are kept in memory during a step. Like before, the weights of the operation are updated first (left), followed by architecture weights (right).

3.4.3 Latency in Performance Evaluation

In terms of chess engines, evaluating the actual performance is not straight forward. While value, policy and combined loss as proposed in [28, 26] offer a measurable performance metric, these values could ultimately be misleading. As explained in section 2.2, chess engines need to be optimized for quick position evaluation (that is, *latency*) as well. Thus, latency in and of itself is a valuable performance predictor that should be incorporated into the search process on top of the other losses.
In addition to the aforementioned improvements of ProxylessNAS [3] over DARTS [19], the authors also integrate latency to the loss function in a differentiable way. That is, they calculate the expected latency of the model, $\mathbb{E}$[latency], multiply it by a scaling factor λ and add this term to the loss function. Instead of relying on a pre-trained latency prediction model however, we measured the latency $L(o)$ of each `LayerChoice` candidate o prior to the search process. As such, we calculate the expected latency of the searched model following the method in [3]:

$$\mathbb{E}[\text{latency}] = \sum_i \mathbb{E}[\text{latency}_i] \tag{3.3}$$

where $\mathbb{E}[\text{latency}_i]$ corresponds to the measured latency of operator candidate i multiplied by it's probability s (3.1):

$$\mathbb{E}[\text{latency}_i] = \sum_j s_j^i \cdot L(o_j^i). \tag{3.4}$$

Finally, the loss function of a model during the search process is a combination of the value, policy and latency loss (3.3):

$$Loss = \alpha \cdot [z - v]^2 - \boldsymbol{\pi}^\top \log \boldsymbol{p} + c \cdot \|\theta\|_2^2 + \lambda \cdot \mathbb{E}[\text{latency}]. \tag{3.5}$$

Just like in [5], the overall loss is kept consistent with the original AlphaZero design. We also adopt $\alpha = 0.01$ to mitigate overfitting. However, to balance the introduction of the latency loss, we also add a loss factor λ.

3.5 Model Comparison

In order to acquire a comprehensive comparison of the trained networks, we transformed the models into ONNX format and optimized their speed using TensorRT, following the approach proposed by Zhou et al. [36]. The final latency information was obtained by using the *ClassicAra* engine[7] to calculate nodes per second.

Additionally, we conducted several Cute Chess[8] matches between all models to assess their playing strength. This approach not only provided insights into relative performance but also allowed for the comparison of elo rating differences among the models.
Each match consisted of 500 rounds with 2 games each. The full `cutechess` commands can be found in appendix (5.2).

[7] `https://github.com/QueensGambit/CrazyAra/tree/master/engine`, accessed 02 February 2024
[8] `https://github.com/cutechess/cutechess/tree/master`, accessed 01 February 2024

4 Evaluation

Following the preceding section, we will give insight into the actual results of the search, training and comparison stages.

4.1 Neural Architecture Search

4.1.1 Latency Measurements of Blocks

Prior to the architecture search, the latency of all blocks in section 2.3 was measured on the same hardware. The average latency over 100 test-runs for each block can be seen in table 4.1. Notably, the standard residual block (RB) used in AlphaZero performed best with a latency of ≈ 0.304 milliseconds. The Next Transformer Block showed the worst performance with ≈ 2.567 milliseconds. This is due to the complexity of the block architectures, featuring two attention modules and a multi-layer perceptron, in comparison to the two 3×3 convolutions of the RB. In general, all blocks perform in relation to their architectural (and thus computational) complexity seen in section 2.3.

Block	**Latency (ms)**
Dense Residual Block (RB)	0.304
Nested Bottleneck Residual Block (NBRB)	1.013
Mobile Convolutional Block (MCB)	0.426
Next Convolutional Block (NCB)	0.668
Next Transformer Block (NTB)	2.567

Table 4.1: Average latency of each block measured in milliseconds. Latency values were sampled independently using random inputs. The measurements were conducted on a NVIDIA V100 GPU, consistent with the experimental setup. Note that the blocks were not sped up using TensorRT.

4.1.2 Search Results

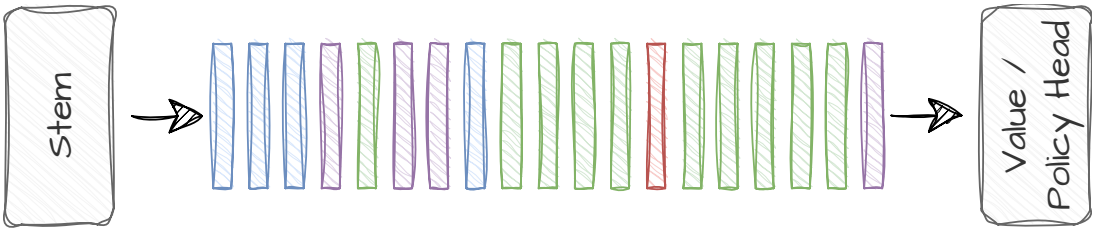
3 architecture searches were performed in total. The first search was executed without adding latency to the loss function. This was done in order to acquire a direct comparison between a search with and without considering latency.
The second and third search both included latency in the loss function. However, we omitted the search results of the first run with $\lambda = 0.2$, as proposed in the official implementation of ProxylessNAS[1]. That is due

[1] `https://github.com/mit-han-lab/proxylessnas`, accessed 23 January 2024

to the fact that the run showed clear signs of bias towards the provided latency-measurements. The overall architecture only changed with respect to two blocks in comparison to base AlphaZero, resulting in a model with 17 Dense Residual Blocks (RB). Thus, we ran a third search, with $\lambda = 0.05$ to mitigate this effect.

The figure in 4.1 showcases the most effective block configurations. Notably, the second model (denoted *ChessNAS-L*) shows great utilization of Residual Blocks (RB). In contrast, the first model (denoted *ChessNAS*) demonstrates the Nested Bottleneck Residual Block (NBRB) and the Mobile Convolutional Block (MCB) as the most valued contenders, each appearing 7 and 6 times, respectively. The Next Convolutional Block (NCB) secures the third position with 3 occurrences, followed by the Next Transformer Block (NTB) with only 2 instances. Lastly, the standard Residual Block (RB) is represented once.
As already mentioned, ChessNAS-L features more occurrences of the RB, 10 in total. Both, the MCB and the NCB share the second place with 4 instances each and only one NBRB made it into the selection. Finally, the NTB was completely excluded by this search.

(a) Final *ChessNAS* architecture, acquired by NAS results without using the additional latency loss.



(b) Final *ChessNAS-L* architecture, acquired by NAS results using latency loss and loss factor $\lambda = 0.05$ (see 3.5 for reference).

Figure 4.1: NAS results of both runs without (a) and with latency loss (b). Green: Dense Residual Block, Red: Nested Bottleneck Residual Block, Blue: Mobile Convolutional Block, Purple: Next Convolutional Block, Yellow: Next Transformer Block.

Individual probability distributions for each block can be seen in 4.2. Every step contains the average probability of one block over all 19 `LayerChoice` options. We can clearly see a rapid change over the first 8 thousand steps ($\sim$ 5 epochs). Afterwards, no significant changes happen in terms of the probabilities. However, one can notice slight deviations for model ChessNAS, while the probabilities of it's competitor show little to no deviation. This is due to the strong effect of the latency loss with respect to the blocks with higher latency.
The block-probabilities of the first search are relatively consistent with the number of occurrences in the final result. Same applies to the second search, where we observe little to no instance of blocks with low probability. Thus, adding a latency constraint certainly increases the chance of lower latency blocks.

Results concerning the individual losses of both models can be seen in figure 4.3. Here, both models show similarities, as the value and policy loss are nearly identical, while the model with latency shows an overall higher total loss than it's competitor. That is due to the introduction of the latency loss, without which both models would also have the same overall loss.

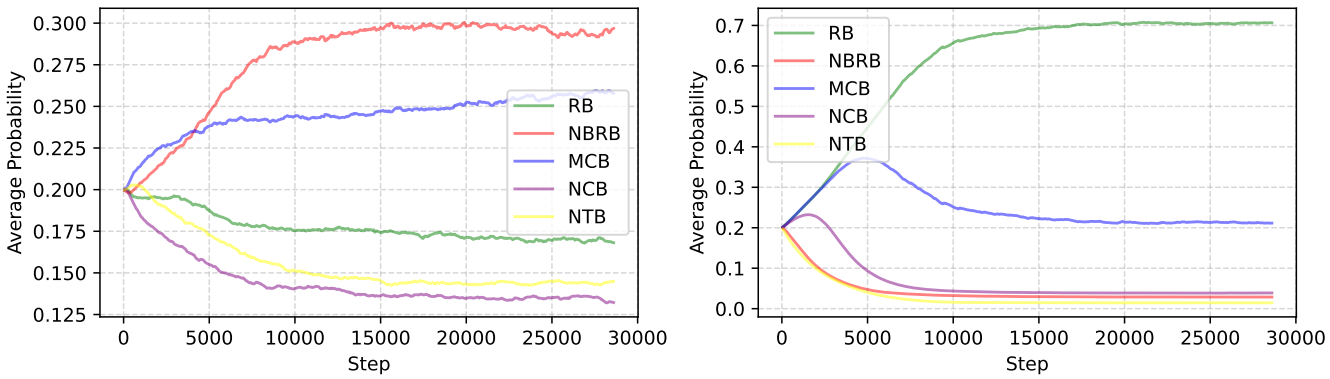


Figure 4.2: Individual block probabilities for ChessNAS (left) and ChessNAS-L (right) during the search process. The displayed values depict the average probability across all 19 blocks for each search step. It's important to note that the y-axis scale differs between the two graphs.

Further, both models show the same trend in the shape of the loss values. First, all losses (except for the value loss) show a rapid decrease which is also consistent with the distribution over the block probabilities in figure 4.2. Here, we also see a rapid change in the individual probabilities at first. After that, the individual probabilities stay relatively consistent, just like the loss values.

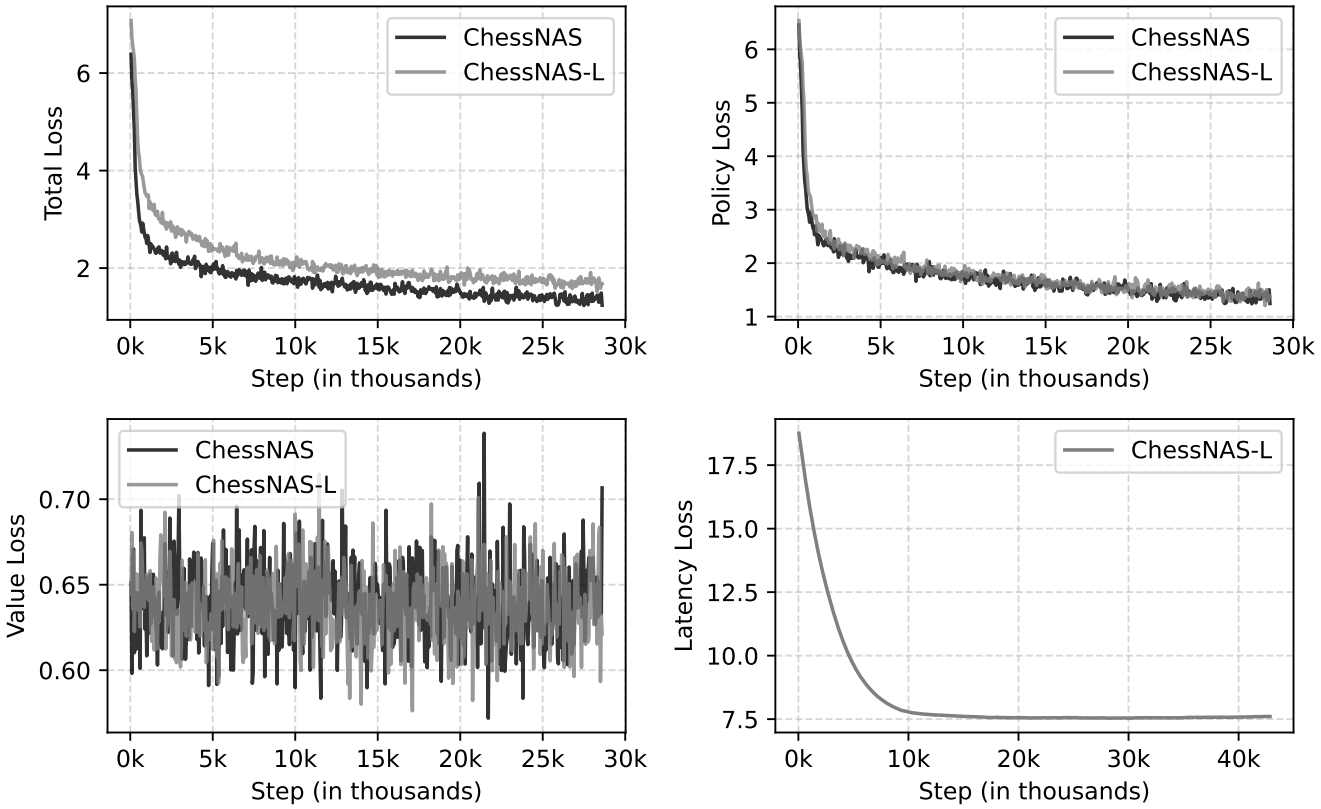


Figure 4.3: Losses of ChessNAS and ChessNAS-L during the neural architecture search. Note, that the latency loss was only calculated for ChessNAS-L. Final values are depicted in table 4.2. Losses are calculated according to equations in [28, 26, 5].

Individual final values, including mean deviation, of the losses can be found in table 4.2. Most notably, the final policy loss of the search with latency is lower than the policy loss of it's competitor, with a consisted mean deviation. The value loss is not significantly different, as one could see in the plot before. Differences in the total loss are, again, due to the latency loss in the second model.

Model	Total Loss	Policy Loss	Value Loss	Latency Loss
ChessNAS	1.238 ± 0.299	1.246 ± 0.303	0.706 ± 0.017	-
ChessNAS-L	1.323 ± 0.371	0.998 ± 0.342	0.617 ± 0.016	7.602 ± 1.199

Table 4.2: Final values and mean deviation of losses for the searched models without and with latency (ChessNAS and ChessNAS-L). Plots of all values can be seen in figure 4.3. Losses are calculated according to equations in [28, 26, 5].

4.2 Network Training

During the training stage, we measured the policy, value and total losses for each of the three models. The latency loss was omitted from training, as it only served as an indicator during the search and would not contribute to performance indication during training. All losses were recorded for both training and validation datasets and can be seen in figures 4.4 and 4.5. Final values plus mean deviations are listed in table 4.3. In terms of policy loss, the latency model (ChessNAS-L) performs considerably worse than the other two models. Both AlphaZero and the model without latency considerations (ChessNAS) perform rather similar, with AlphaZero keeping the upper hand in both training and validation policy loss. However, we see around double the loss values for ChessNAS-L. All three models have a rapid decrease for both training and validation losses for the first 80 steps (first epoch), followed by relatively stable values over subsequent training iterations.

Model	**Total Loss**	**Policy Loss**	**Value Loss**
Training:			
AlphaZero	1.119 ± 0.175	1.126 ± 0.177	0.417 ± 0.036
ChessNAS	1.227 ± 0.170	1.235 ± 0.171	0.421 ± 0.040
ChessNAS-L	2.151 ± 0.124	2.167 ± 0.125	0.519 ± 0.097
Validation:			
AlphaZero	1.221 ± 0.179	1.229 ± 0.180	0.432 ± 0.022
ChessNAS	1.279 ± 0.185	1.288 ± 0.187	0.435 ± 0.028
ChessNAS-L	2.171 ± 0.129	2.188 ± 0.131	0.535 ± 0.088

Table 4.3: Final values and mean deviation of both training and validation losses for the trained models. Plots of all values can be seen in figure 4.4 and 4.5. Losses were calculated according to [28, 26, 5].

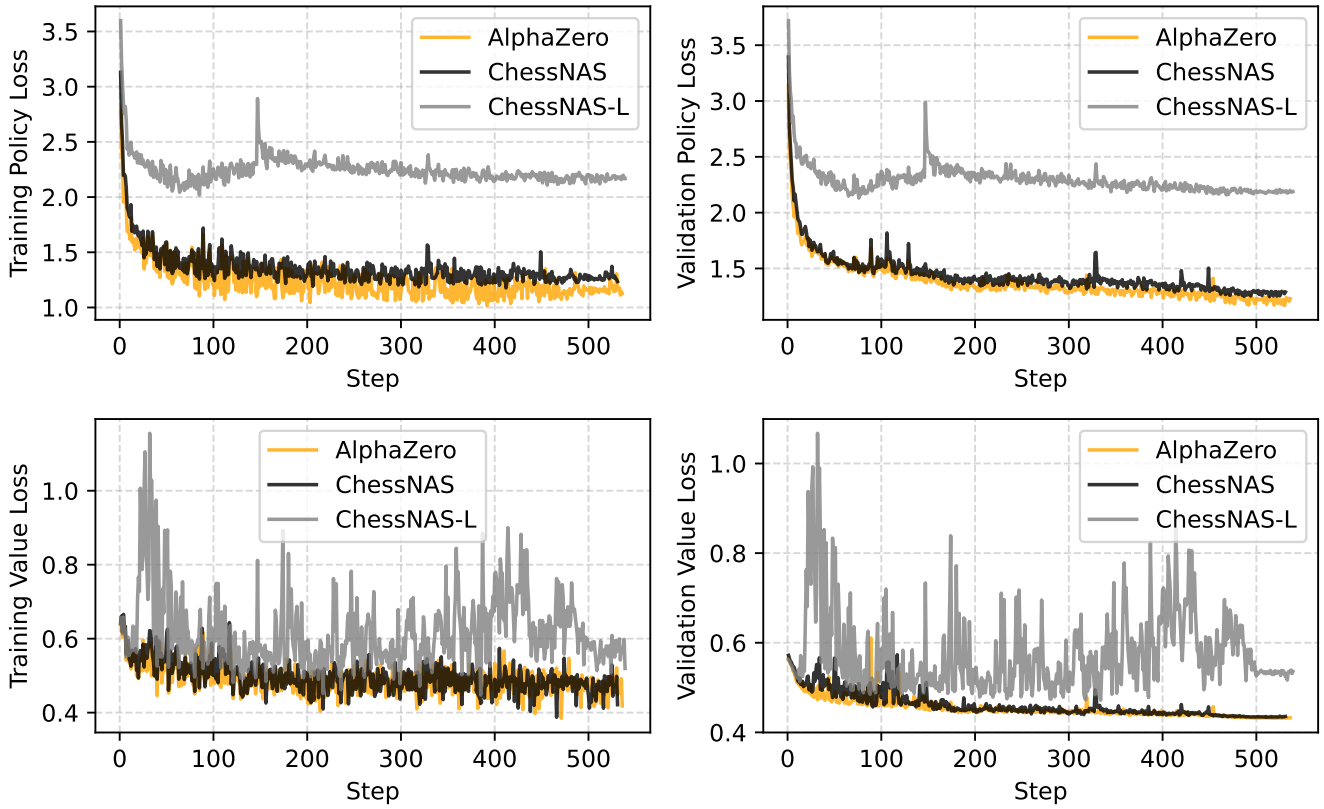


Figure 4.4: Policy and value losses while training from scratch for all models. Depicted are the values for the losses according to [28, 26, 5] in training and validation steps.

Examining the value loss does not yield different results. AlphaZero and ChessNAS both acquired similar value losses for training and validation. Fluctuation in the validation value loss are considerably less, which is to expect due to the lower number of validation steps. ChessNAS-L however achieves worse values overall and far more fluctuations than it's competitors.

Finally, the total loss is nearly identical to the policy loss values, as intended by keeping the influence of the value loss low. The same pattern applies, having a rapid decrease at first, while staying relatively stable after the first epoch. ChessNAS-L also performs considerably worse, just like before.

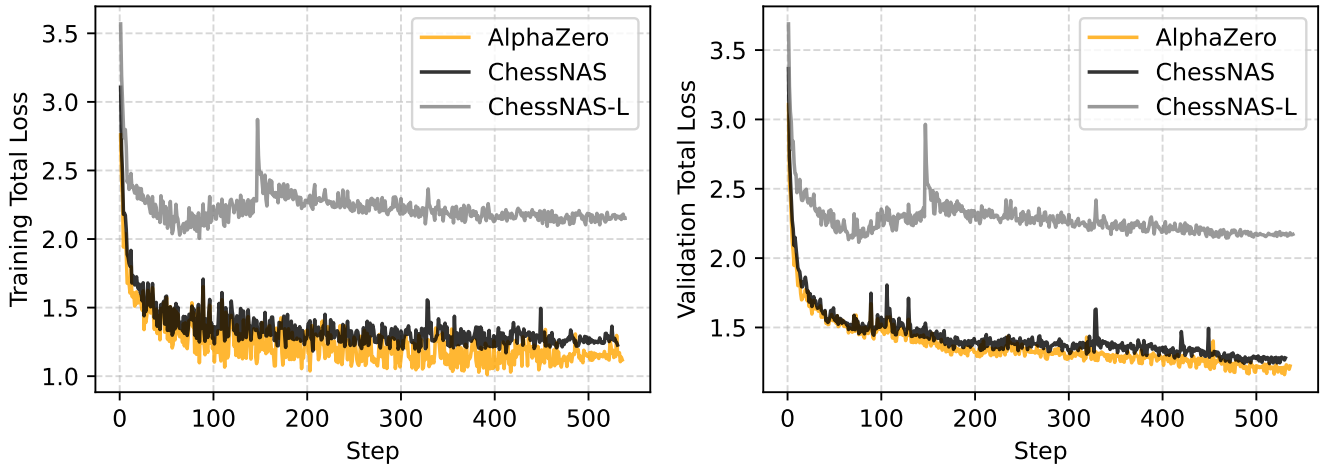


Figure 4.5: Total losses while training from scratch for all models. Depicted are the values for the total loss according to [28, 26, 5] in training and validation steps.

In addition to the losses, we also measured the policy and value accuracy for training and validation (reference figure 4.6). They show similar patterns as the losses. At first, all accuracies increase rapidly. The policy accuracy again stagnates after around 80 steps, while the value accuracy actually further increases in a linear manner.
In terms of the models, we again see inferior results for ChessNAS-L. This is especially true for the policy accuracy, which is roughly half of it's competitors. The value accuracy is similar, with more extreme fluctuations in comparison to the other models. This ultimately suggest less stability for the latency model.

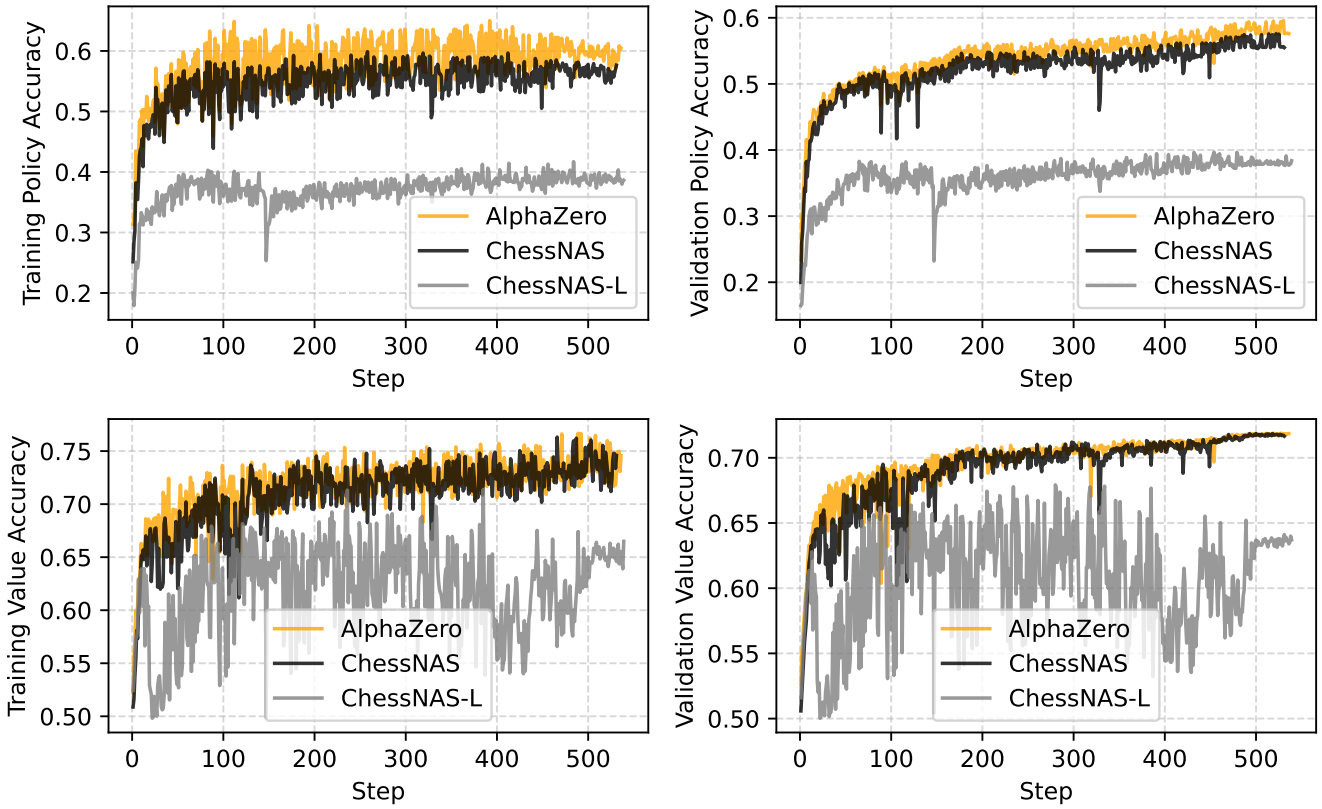


Figure 4.6: Policy and value accuracy while training from scratch for all models. Depicted are the values for the accuracies according to [28, 26, 5] in training and validation steps.

4.3 Network Comparison

For the final comparison of the proposed models (ChessNAS, ChessNAS-L) and AlphaZero, we first measured the calculation speed of the fully trained models. That is, we took the exported models in ONNX format, sped them up using TensorRT and measured the nodes per second- and subsequent latency-scores by searching for the best move from the standard starting position in chess. Results are depicted in table 4.4. Here, we can clearly see that AlphaZero shows the best results, having the lowest latency with $\approx 42\mu s$. That comes with no surprise, as the previous latency measurements in 4.1 suggest the standard residual block to perform best in terms of latency, when compared to the other blocks.
Further, we indeed see the impact of the latency loss in ChessNAS-L. It achieves considerably lower latency values ($\approx 45\mu s$) than it's direct competitor ChessNAS ($\approx 60\mu s$). This suggests positive impact of the introduction of the latency loss into the total loss function.

Finally, in order to assess true playing strength of all models, we conducted three matches consisting of 1000 games between the models. First, we let ChessNAS and ChessNAS-L compete against each other. As to be expected from the previous training results, ChessNAS-L was not successful against his strong competitor. As such, ChessNAS secured 849 wins, while ChessNAS-L won 94 of the encounters. 57 games ended in a draw. This results in a (relative) elo difference of 342.0, with an error margin of 30.8.

Model	Average NPS	Latency (μs)
AlphaZero	23973.1	41.71
ChessNAS	16595.8	60.26
ChessNAS-L	22019.8	45.41

Table 4.4: Average nodes per second (NPS) and measured latency for the searched models, without and with latency considerations respectively (ChessNAS and ChessNAS-L) and AlphaZero. Less latency and more NPS are desirable in chess, due to time constraints.

The subsequent match between ChessNAS-L and AlphaZero played out similarly. This time however, AlphaZero won all 1000 games against ChessNAS-L, which ultimately suggests the inferiority of the network architecture. This also removes the ability to calculate elo differences between the two models.

In the end, it came down to the match between ChessNAS and AlphaZero. AlphaZero dominated the match, acquiring 995 wins, while only losing 3 times and scoring two draws. That results in a (relative) elo difference of 958.5, with an error margin of 269.4. As such, both models found during the neural architecture search could not beat the baseline AlphaZero.

5 Conclusion

This section will conclude our work and give insights to potential improvements in future work.

5.1 Discussion

Using a *Proxyless* neural architecture search, we explored recent neural architecture developments on the AlphaZero chess engine [3, 26]. Specifically, within the chess engine *CrazyAra*, we created a diverse search space, offering alternatives to the basic residual block [10] employed in AlphaZero. These alternatives ranged from improved convolutional blocks (Mobile Convolutional Block [25, 11], Next Convolutional Block [16]) to enhancements found in other AlphaZero-based engines (Nested Bottleneck Residual Block in KataGo [33]) and a transformer variant (Next Transformer Block [16]).

The proposed search space was then first explored with ProxylessNAS, initially disregarding latency - a crucial performance indicator in the time controlled domain of chess. This yielded an architecture we denote as *ChessNAS*. Subsequently, in a second run, we introduced latency loss, as originally proposed by the works of Cai et al. [3], in order to establish a direct comparison: *ChessNAS-L*. The resulting architectures exhibited promising results in terms of loss metrics during the search period (see figure 4.3 for reference). While searching for ChessNAS-L, we noticed a high bias potential towards the latency loss with the latency loss factor originally used in ProxylessNAS. Hence, we reduced the latency loss factor. Even after reduction, the introduction of the latency loss showed great influence on the resulting architecture with a 70% chance of adopting the fastest block.
However, both architecture do not seem to show any signs of patterns. While only one search was executed, and thus general assessments over the existence of patterns is questionable, the provided architectures do seem to be somewhat “random”.

Training the architectures from scratch provided early insights into the results when comparing playing strength. Surprisingly, ChessNAS-L underperformed considerably compared to its competitors across all tracked loss and accuracy metrics during training. In a head-to-head match of 1000 games against ChessNAS, ChessNAS-L won only 94 games, indicating inferior results, and failed to secure a single victory against the baseline AlphaZero. Additionally, ChessNAS managed only 3 wins in a 1000-game match against AlphaZero.

The lack in playing strength of ChessNAS-L could stem from a number of factors. First of all, although the training of each architecture was performed using the same (random) seed, this does not guarantee ultimate comparability between the model training and final results. As implied by work of Bethard et al. [2], applying multiple seeds and taking the best resulting model would have been a safer procedure than relying on a single seed. This is even more relevant when looking at the initial loss metrics during the search phase. Here, ChessNAS-L did perform similarly when compared to ChessNAS. Due to time constraints, we were not able to perform multiple trainings using different seeds. However, this would also apply to the search phase altogether.

As we only used a single search for both architectures, we potentially already excluded better architectures before the actual training had begun. Running the search with more seeds might also reduce the perceived "randomness" of the resulting architectures.

ChessNAS's poor performance against the baseline AlphaZero raises questions about the effectiveness of the chosen NAS algorithm in the chess domain. The selection of ProxylessNAS initially seemed promising, given the importance of latency in strong chess engines. However, the incorporation of latency could be a critical factor. Cai et al. utilized a *latency prediction model* to estimate latency during training, adapting to diverse hardware constraints [3]. In our case, hardware constraints were consistent throughout all phases. Yet, utilizing latency measurements for single-block architectures might have negatively impacted the search.

All in all, our initial attempt to find a strong, novel chess architecture using NAS failed. The experiences gained from this exploration still offer valuable insights for future attempts at utilizing neural architecture search. The difficulties we encountered provide opportunities to fine-tune NAS methodologies, aligning them more effectively with the demands inherent in the dynamics of chess.

5.2 Future Work

As the previous section suggests, there exists room for improvement. Subsequent work could build upon these findings and experiment with alternative NAS search strategies [1]. Introducing a novel performance estimation strategy for chess could further improve the architecture search. Efficient GPU utilization of the chess architecture is also worth investigating during the architecture search [15].
Altering the overall shape of AlphaZero to adjust to the increase in number of parameters by using methods introduced in work by Tan et al. [29] and applied in [5] could also be investigated. Further, other recent advancements should be incorporated. That includes work by the developers of Lc0 concerning *Squeeze-and-Excitation* networks [13], their improvement of the value head or their introduction of a third - *moves left* - head. Additionally, picking up on the number of parameters, the Nested Bottleneck Residual Block should be further investigated. It shows great potential, as it did not only achieve the highest probability in ChessNAS (see figure 4.2), but is also employed by recent versions of the Go playing engine KataGo [33]. This includes some potential modifications regarding 5×5 convolutions.

Acknowledgements

I would sincerely like to thank everyone who has supported me throughout the academic journey this thesis proved to be. That includes my fellow students, Mia Zech, Nathalie Weger and Marc Saghir, who took the time and effort to review my thesis, but also my girlfriend and family, who emotionally supported me on the way. In addition, I especially thank Johannes Czech for providing all necessary research materials and all the needed information.
Finally, I do have to note that the art-style used for the creation of the figures is highly inspired by the KataGo methods in [7].

Bibliography

[1] G. Bender et al. “Can Weight Sharing Outperform Random Architecture Search? An Investigation With TuNAS”. In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2020, pp. 14311–14320. DOI: 10.1109/CVPR42600.2020.01433. URL: https://doi.ieeecomputersociety.org/10.1109/CVPR42600.2020.01433.

[2] Steven Bethard. *We need to talk about random seeds*. 2022. arXiv: 2210.13393 [cs.CL].

[3] Han Cai, Ligeng Zhu, and Song Han. *ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware*. 2019. arXiv: 1812.00332 [cs.LG].

[4] Murray Campbell, A.Joseph Hoane, and Feng-hsiung Hsu. “Deep Blue”. In: *Artificial Intelligence* 134.1 (2002), pp. 57–83. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(01)00129-1. URL: https://www.sciencedirect.com/science/article/pii/S0004370201001291.

[5] Johannes Czech, Jannis Blüml, and Kristian Kersting. “Representation Matters: The Game of Chess Poses a Challenge to Vision Transformers”. In: *CoRR* abs/2304.14918 (2023). DOI: 10.48550/ARXIV.2304.14918. arXiv: 2304.14918. URL: https://doi.org/10.48550/arXiv.2304.14918.

[6] Ivo Danihelka et al. *Policy improvement by planning with Gumbel*. 2022. URL: https://openreview.net/forum?id=bERaNdoegn0.

[7] KataGo developers. *Nested Bottleneck Residual Nets*. [Online; accessed 06-February-2024]. 2022. URL: https://github.com/lightvector/KataGo/blob/master/docs/KataGoMethods.md#nested-bottleneck-residual-nets.

[8] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. “Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves”. In: *Proceedings of the 24th International Conference on Artificial Intelligence*. IJCAI’15. Buenos Aires, Argentina: AAAI Press, 2015, pp. 3460–3468. ISBN: 9781577357384.

[9] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2021. arXiv: 2010.11929 [cs.CV].

[10] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[11] Andrew Howard et al. *Searching for MobileNetV3*. 2019. arXiv: 1905.02244 [cs.CV].

[12] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 [cs.CV].

[13] Jie Hu et al. *Squeeze-and-Excitation Networks*. 2019. arXiv: 1709.01507 [cs.CV].

[14] Dominik Klein. *Neural Networks for Chess*. 2022. arXiv: 2209.01506 [cs.LG].

[15] Jack Kosaian and Amar Phanishayee. *A Study on the Intersection of GPU Utilization and CNN Inference*. 2022. arXiv: 2212.07936 [cs.LG].

[16] Jiashi Li et al. *Next-ViT: Next Generation Vision Transformer for Efficient Deployment in Realistic Industrial Scenarios*. 2022. arXiv: 2207.05501 [cs.CV].

[17] Liam Li and Ameet Talwalkar. *Random Search and Reproducibility for Neural Architecture Search*. 2019. arXiv: 1902.07638 [cs.LG].

[18] Yanyu Li et al. *EfficientFormer: Vision Transformers at MobileNet Speed*. 2022. arXiv: 2206.01191 [cs.CV].

[19] Hanxiao Liu, Karen Simonyan, and Yiming Yang. *DARTS: Differentiable Architecture Search*. 2019. arXiv: 1806.09055 [cs.LG].

[20] Joseph Mellor et al. *Neural Architecture Search without Training*. 2021. arXiv: 2006.04647 [cs.LG].

[21] Microsoft. *Neural Network Intelligence*. Version 3.0. Jan. 2021. URL: https://github.com/microsoft/nni.

[22] Yu Nasu. "Efficiently Updatable Neural-Network-based Evaluation Functions for Computer Shogi". In: *Ziosoft Computer Shogi Club* (2018). URL: https://www.apply.computer-shogi.org/wcsc28/appeal/the_end_of_genesis_T.N.K.evolution_turbo_type_D/nnue.pdf.

[23] Esteban Real et al. *AutoML-Zero: Evolving Machine Learning Algorithms From Scratch*. 2020. arXiv: 2003.03384 [cs.LG].

[24] Mats L. Richter et al. "Should You Go Deeper? Optimizing Convolutional Neural Network Architectures without Training". In: *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, Dec. 2021. DOI: 10.1109/icmla52953.2021.00159. URL: http://dx.doi.org/10.1109/ICMLA52953.2021.00159.

[25] Mark Sandler et al. *MobileNetV2: Inverted Residuals and Linear Bottlenecks*. 2019. arXiv: 1801.04381 [cs.CV].

[26] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. DOI: https://doi.org/10.48550/arXiv.1712.01815. arXiv: 1712.01815 [cs.AI].

[27] David Silver et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. DOI: 10.1038/nature16961.

[28] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550 (Oct. 2017), pp. 354–. URL: http://dx.doi.org/10.1038/nature24270.

[29] Mingxing Tan and Quoc V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. 2020. arXiv: 1905.11946 [cs.LG].

[30] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL].

[31] Tim Vieira. *Gumbel-max trick and weighted reservoir sampling*. [Online; accessed 06-February-2024]. 2014. URL: http://timvieira.github.io/blog/post/2014/08/01/gumbel-max-trick-and-weighted-reservoir-sampling/.

[32] Colin White et al. *Neural Architecture Search: Insights from 1000 Papers*. 2023. arXiv: 2301.08727 [cs.LG].

[33] David J. Wu. *Accelerating Self-Play Learning in Go*. 2020. arXiv: 1902.10565 [cs.LG].

[34] Xin Xia et al. *TRT-ViT: TensorRT-oriented Vision Transformer*. 2022. arXiv: 2205.09579 [cs.CV].

[35] Weihao Yu et al. *MetaFormer Is Actually What You Need for Vision*. 2022. arXiv: 2111.11418 [cs.CV].

[36] Yuxiao Zhou and Kecheng Yang. “Exploring TensorRT to Improve Real-Time Inference for Deep Learning”. In: *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)* (2022), pp. 2011–2018. URL: https://api.semanticscholar.org/CorpusID:253882601.

[37] Barret Zoph and Quoc V. Le. *Neural Architecture Search with Reinforcement Learning*. 2017. arXiv: 1611.01578 [cs.LG].

Appendix

```
cutechess-cli -variant standard -openings file=chess.epd format=epd order=
    random -pgnout model-1_vs_model-2.pgn -resign movecount=5 score=600 -draw
    movenumber=30 movecount=4 score=20 -concurrency 1 -engine name=Model_1 dir
    =path/to/engine cmd=./ClassicAra proto=uci option.Model_Directory=path/to/
    model -engine name=Model_2 dir=path/to/engine cmd=./ClassicAra proto=uci
    option.Model_Directory=path/to/model -each tc=0/60+0.1 option.Nodes=200
    option.Simulations=400 option.Batch_Size=16 option.First_Device_ID=ID -
    games 2 -rounds 500 -repeat
```

Listing 1: Full `cutechess` command used to compare playing strength of AlphaZero, ChessNAS and ChessNAS-L in head-to-head matches. Opening book (`chess.epd`) consists of various standard chess openings. As for the engine, we used the built-in ClassicAra engine of the *Crazyara* repository (Github link).

Parameter	Search Stage	Training Stage
Dataset	KingBaseLite 48k	KingBaseLite
Batch Size	512	1024
Minimum Learning Rate	0.002	0.001
Maximum Learning Rate	0.14	0.07
Epochs	20	7

Table 1: Hyperparameters that differ in the search and training stages. Dataset and batch size for the search stage were reduced in order to decrease search time. As for the learning rate, a one-cycle learning rate scheduler was used. More epochs were used for the search stage as a means to facilitate the operation choices throughout the layers.