City, University of London

School of Science and Technology

Department of Computer Science

MSc in Artificial Intelligence

Project Report

2022

Switchable Lightweight Anti-Symmetric Processing (SLAP) with CNN - Application in Gomoku Reinforcement Learning

Chi-Hang Suen

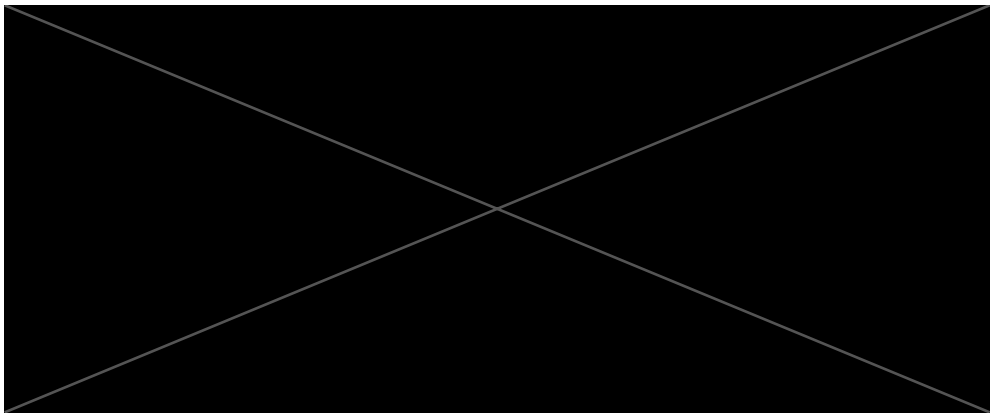
Supervised by: Dr Eduardo Alonso

30 Sep 2022

Declaration

By submitting this work, I declare that this work is entirely my own except those parts duly identified and referenced in my submission. It complies with any specified word limits and the requirements and regulations detailed in the assessment instructions and any other relevant programme and module documentation. In submitting this work I acknowledge that I have read and understood the regulations and code regarding academic misconduct, including that relating to plagiarism, as specified in the Programme Handbook. I also acknowledge that this work will be subject to a variety of checks for academic misconduct.

Signed:



Chi-Hang Suen

Abstract

I created a novel method called SLAP to speed up convergence of machine learning. SLAP is a model-independent protocol to produce the same output given different transformation variants. It can be used upon any function or model to produce outputs that are invariant with regard to specified symmetric properties of the inputs. It can be viewed as standardization of symmetry, as opposed to standardization of scale. In a preliminary stage with experiment by synthetic states, CNN with SLAP had smaller validation losses than baseline (CNN without SLAP) for 77% of groups of hyperparameter and architecture combinations, despite that baseline was trained using 8 times the number of training samples by data augmentation. Among selected models from this preliminary stage and upon further training, it was found that SLAP improved the convergence speed of neural network learning by 83% compared with its baseline control. In reinforcement learning for a board game called Gomoku, AlphaGo Zero/AlphaZero algorithm with data augmentation was used as the baseline, and the use of SLAP reduced the number of training samples by a factor of 8 and achieved similar winning rate against the same evaluator.

Keywords: AlphaGo Zero, symmetric CNN, reinforcement learning, SLAP, Gomoku

Acknowledgements

I would like to express my gratitude to my supervisor Dr. Eduardo Alonso for his insightful feedback, encouraging support and timely response during the course of this research. Furthermore, I would like to thank all teaching staff and assistants of MSc in Artificial Intelligence Programme for their education and support, collectively paving the way to this project.

Table of Contents

1. Introduction and Objectives

1.1 Problem Background

Convolutional neural network (CNN) is now the mainstream family of models for computer vision, thanks to its weight sharing mechanism to efficiently share learning across the same plane by so-called kernels, achieving local translational invariance. However, CNN is not reflection and rotation invariant. Typically it can be addressed by data augmentation to inputs by reflection and rotation if necessary, but the sample size would increase substantially. Hinton G. E., Krizhevsky A. and Wang S. D. (2011) criticised CNN that it could not learn spatial relationships such as orientation, position and hierarchy and advocated their novel capsule to replace CNN. Sabour S., Frosst N. and Hinton G. E. (2017) improved capsule using routing by agreement mechanism and outperformed CNN at recognising overlapping images, but they also admitted that it tended to account for everything in the structure. This implies capsule is too heavy in computation. Inspired by the idea of capturing orientation information in capsule network (Sabour S., Frosst N. and Hinton G. E. 2017), I would like to introduce a novel method called Switchable Lightweight Anti-symmetric Process (SLAP), a protocol to produce the same output given different transformation variants. My research question is: can transformation variants be exploited directly by SLAP to further improve and combine with CNN for machine learning?

Very often, given a certain machine learning task, we know in advance whether the task is invariant to certain types of transformation, e.g. rotation, reflection and translation. For example, in the game Gomoku (See Fig. 1, also called “Five in a Row”), the state is rotation (perpendicularly) and reflection (horizontally and vertically) invariant from the perspective of winning probability, and “partially” translation invariant. This symmetry is often exploited by data augmentation for deep learning. But this can greatly increase the dataset size if all rotation and reflection variants are included – for example there are 8 such variants for each state of Gomoku. To keep the same sample size, I created SLAP, which is, in the simplest case, just an encoder function to always output the same unique variant regardless of which of those 8 variants (using Gomoku as example) is the input, and also output the transformation information (e.g. rotation angle), though the latter not necessarily needed. Though this novel method is not a type of capsule network, but I give capsule the credit for the inspiration.

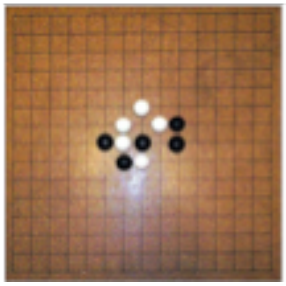


Fig. 1 Gomoku
Photo Credit: Wikipedia

1.1.1 What is Switchable Lightweight Anti-symmetric Process (SLAP)?

SLAP is a model-independent protocol and function to always produce or choose the same variant regardless of which transformation variant (by specified symmetry) is given, and if required also output the corresponding transformation. It can be used upon any function or model to produce outputs that are invariant with regard to specified symmetric properties of the inputs. If some (type) of the outputs are not invariant but follow the same transformation, the corresponding transformation information from SLAP may be used to transform these outputs back. It can be viewed as standardization of symmetry, as opposed to standardization of scale. After processing, symmetric variants are filtered out – that's why it is named 'anti-symmetric process'. Ironically, with this anti-symmetric process, the function or model (e.g. CNN) to be fed would look as if it is symmetric with regard to whichever the symmetry variant is the input, and the same output is produced. It is my novel method to exploit symmetry variants in machine learning without increasing the number of training samples by data augmentation. The motivation is to concentrate experience to speed up learning. See implementation details in Chapter 3 Methods.

1.2 Choice of Project and Beneficiaries

My research question is: can transformation variants be exploited directly by SLAP to further improve and combine with CNN for machine learning? The specific scope of this research is Gomoku reinforcement learning. To control the risk of running out of time, I planned to focus on rotation and reflection variants, and time permitting, also translation variants which are a groupoid for Gomoku. Gomoku was chosen because this game is rotation and reflection

invariant, but only "partially" translation invariant, so ideal for SLAP to test different transformations (see more technical considerations in Ch 2.2 Why Gomoku?). The beneficiaries would be AI practitioners in computer vision and reinforcement learning, in domains where symmetric properties are applicable.

1.2.1 What is Gomoku?

Gomoku, also called 5-in-a-row, is a 2-player board game, traditionally played with Go pieces (black and white stones) on a Go board (19x19), nowadays on a 15x15 board. It can be viewed as an advanced version of Tic-Tac-Toe.
Gomoku game rules of the freestyle version (i.e. no standard rules or professional rules which are mainly aimed at balancing first-mover advantage, especially for tournaments):

- Black and white place stones of his colour alternatively at an unoccupied intersection point of the board (of size 15x15 intersection points). Black first.
- Winner: the one who first forms an unbroken chain of 5 stones of his colour in a line (horizonal, vertical or diagonal)
- Draw happens if there is no winner when all intersection points are filled up.

Throughout this research, the freestyle version of Gomoku was adopted. To save computation, mini Gomoku board 8x8 (same size as chess) was used instead of the standard board 15x15.

1.3 Objectives and Products

Overall objective: by exploiting transformation variants, create novel method SLAP added to CNN that can improve machine learning. Objectives were split as follows:

- Objective 1: Implement SLAP function for the case of Gomoku, without using any domain-specific knowledge except the required symmetric properties.
- Objective 2: Test whether SLAP can benefit neural network learning, measured by validation losses and/or training time for convergence.
- Objective 3: Test whether SLAP can benefit reinforcement learning in playing Gomoku, measured by winning ratios and/or training time for attaining certain winning ratios.

Objective 2 in the above was added to initial plan.

Products of the work: novel creation of SLAP algorithm and function, and whether it improves machine learning, specifically Gomoku reinforcement learning in this case.

1.4 Methods and Work Plan

One way to implement SLAP is simply flattening the pixels of 8 variants to 8 lists, compare the lists and always choose the largest. I expected that, say in Gomoku, this could intensify the learning in certain Gomoku regions by 8 times (and other 7 variants can be ignored as we will keep the same model structure in the evaluation/testing mode) and learning could converge much faster than without using SLAP.

To test whether SLAP can benefit Gomoku reinforcement learning, the baseline followed AlphaGo Zero and Alpha Zero algorithms (almost identical algorithms except for applying to different games) and used data augmentation to produce more challenging baseline results. In short, the baseline algorithm used CNN to learn from the game states, actions and rewards during self-play, in which CNN was used to estimate expected reward and prior probability of taking each action, which were used as inputs for Monte Carlo Tree Search (MCTS) to search and take a stronger action. SLAP was added to wrap the CNN and compared with baseline control results. To save computation, smaller board 8x8 was used instead. The metrics were the winning ratios against an evaluation agent to play Gomoku against baseline and SLAP agents respectively. See details in 3. Methods.
The milestones included:

- Literature review
- Setup of baseline model (see above)
- Setup of environment of Gomoku game (see 1.2.1 What is Gomoku?)
- Setup of evaluation agent – to play Gomoku against baseline and SLAP agents respectively
- Analysis and evaluation
- Completion of report

The initial work plan with dependency graph were as follows in Fig. 2 and Fig. 3:

	Jul				Aug					Sep			
	4	11	18	25	1	8	15	22	29	5	12	19	26
*Literature review**	■	■											
Baseline model setup			■	■									
Gomoku environment setup				■									
*Evaluation agent setup**					■								
Novel structure – phase 1					■								
Self-play training – phase 1						■							
*Analysis/evaluation phase 1**						■							
Novel structure – phase 2							■	■					
Self-play training – phase 2								■	■				
*Analysis/evaluation phase 2**									■	■			
*Report**	■	■	■	■	■	■	■	■	■	■	■	■	

**Milestones of a stage/phase; 1: rotation and reflection variants; 2: translation variants*

Fig. 2 Timeline and milestones

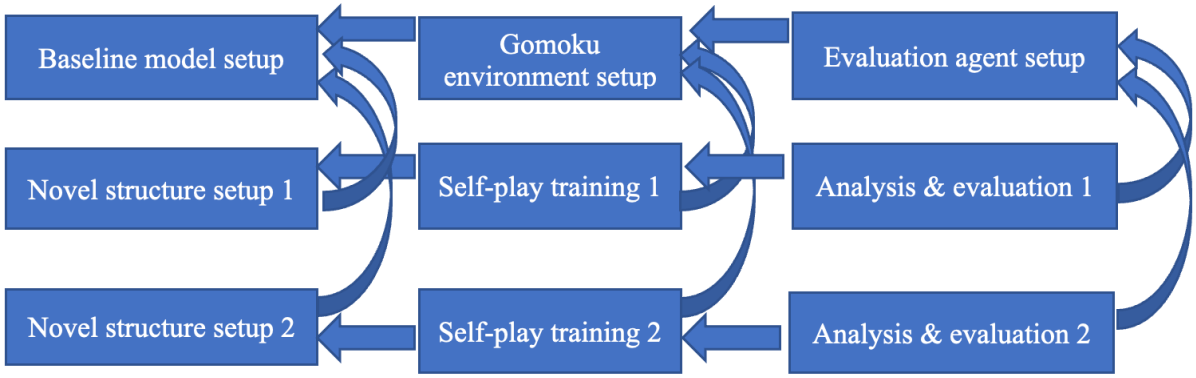


Fig. 3 Dependency Graph

The work plan was basically followed except that objective 2 was added (see 1.3 Objectives and Products) and tested by synthetics states (see details in 3. Methods), and that experiments on translation variants were postponed and carried out more briefly than planned.

1.5 Report Structure

This report includes 6 chapters:

1. Introduction – outline of problem background regarding machine learning on different transformation variants, objectives of this research and motivations, expected products, planned methods and work, and structure of this report.

2. Context – description and commentary of related academic work regarding reinforcement learning (especially about board games) and different transformation variants, and relationship to my research.
3. Methods – description of implementation of SLAP and baseline models, and how to evaluate results.
4. Results – display of results of experiments with tables and charts.
5. Discussion – discussion of results with respect to objectives.
6. Evaluation, Reflections and Conclusions – evaluation of the project as a whole, and reflection on what has been learnt and what might be proposed for further work, and conclusions.

Note that Chapter 3 Methods relied heavily on the literature mentioned in Chapter 2 Context, and that domain-specific methods were avoided (except for representing input states, listing available actions and checking terminal status of game and except using required symmetric properties) such that results (Chapter 4) could be used to answer boarder research question in Chapter 1 Introduction, and to facilitate discussion (Chapter 5), evaluation, reflections and conclusions (Chapter 6) in both Gomoku-specific and domain-independent manners.

2. Context

2.1 The Need for SLAP with CNN

CNN (convolutional neural network) has been widely used for computer vision neural network learning but it is known that CNN is weak to deal with changes by rotation/orientation unless at the cost of much larger sample size by data augmentation. To address this problem, Hinton G. E., Krizhevsky A. and Wang S. D. (2011) proposed that neural network should make use of their then novel capsule, learning to recognize an implicitly defined visual entity and output probability of its existence and instantiation parameters such as pose; they showed that a transforming auto-encoder could be learnt to force the output (which is a vector instead of scalar) of a capsule to represent an image property that one might want to manipulate. Sabour S., Frosst N. and Hinton G. E. (2017) showed that a discriminatively trained, multi-layer capsule system achieves state-of-the-art performance on MNIST and was considerably better than a convolutional net at recognizing highly overlapping digits, using the so-called routing by agreement mechanism. However, the same paper (Sabour S., Frosst N. and Hinton G. E. 2017) also admitted that one drawback of capsule is that it likes to account for everything in an image. This implies the capsule might be too "heavy" for computation and here I would like to introduce SLAP. Also note that Peer D., Stabinger S. and Rodriguez-Sanchez A. (2021) proved that the capsule network with routing by agreement algorithm is not a universal approximator, i.e. not fit to learn all kinds of problems. As such, I did not attempt to replace CNN by capsule, but simply introduced SLAP to combine with CNN. Instead of forcing the output to represent certain transformation information (e.g. orientation angle), the novel SLAP forces the input of different variants (e.g. different rotation angle) to give the same output variant (and output the transformation information e.g. angle, if needed). So SLAP is not a capsule, but I give credit to capsule for the inspiration. I argue that, for example in the case of Gomoku, this can intensify the learning experience of certain region by 8 times and thus speed up learning.

2.2 Why Gomoku?

Reinforcement learning notoriously requires huge training sample size, so it is expected to benefit significantly from the novel SLAP. Gomoku was chosen to demonstrate the benefit of SLAP.

The reason for choosing Gomoku:

- Gomoku has simple rules.
- Gomoku has huge number of state representations ($3^{225} \sim= 2x10^{107}$), which justify the use of neural network for learning.
- Gomoku is rotation and reflection invariant, but only "partially" translation invariant, so ideal for SLAP to test different transformations
- Gomoku is Markov Decision Process as it does not need information about previous states to make decisions, which meets the basic assumption of the mathematical framework of reinforcement learning
- Important literature has shown a general effective reinforcement learning algorithm for board games (Silver D. et al 2017b)

2.3 Gomoku and Groupoid

There are different Gomoku states of the same groupoid (see Fig. 4), which means having local symmetry but not necessarily global symmetry of the whole structure (Vistoli A. 2011). Note that groupoid is much more challenging than symmetry or group, as some groupoids may not have the same status. E.g. States A & B above have 2 winning positions for white, and they can lead to white winning immediately after black's move regardless of what action black will take, but not for State C. On the other hand, the potential for learning is huge as there are much more variants of the same local structure, e.g. 156 variants by translation in Fig. 4.

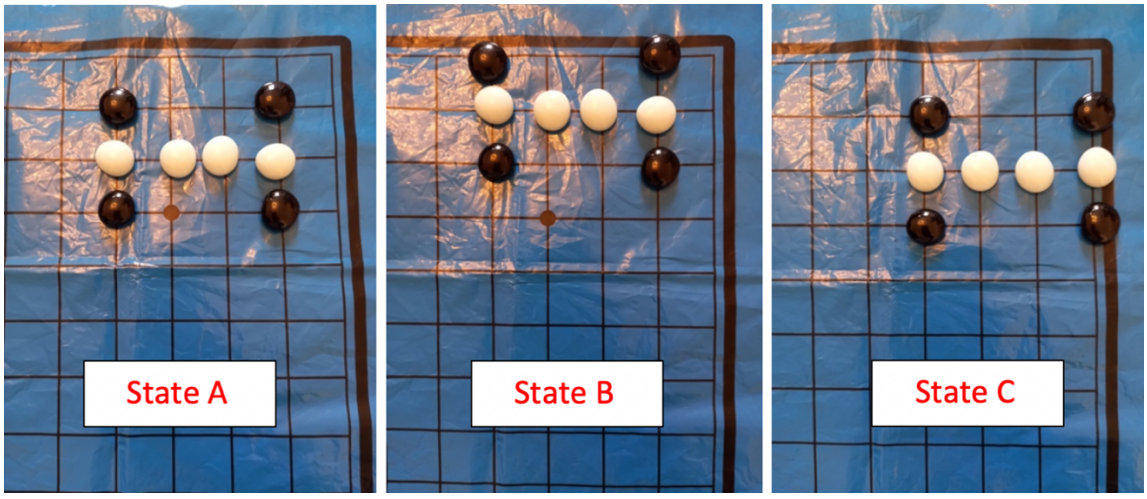


Fig. 4 Gomoku groupoid

2.4 Reinforcement Learning and AlphaGo Zero / Alpha Zero

2.4.1 Reinforcement Learning

Reinforcement learning is a machine learning method to yield a policy (can be probabilistic) to choose optimal actions in response to an environment, maximizing total rewards; it assumes the environment is Markov Decision Process, in which the next state of the environment only depends on the current state and the action of the agent(s) for each discrete time-step t, with the following basic elements (Alonso E. 2022):

State s ϵ S, where S is the set of all states

Action a ϵ A, where A is the set of all actions

Transition function $P(s_{t+1} \mid s_t, a_t)$

Policy π : how to map states to actions

Reward r : a numerical signal representing how good a state is

State-value function V(s): expected value of state (total rewards) given state s

Action-value function Q(s, a): expected value of state given state s *and* action a

Policy evaluation: making the value function consistent with the current policy

Policy improvement: making the policy greedy with respect to the current value function

Policy iteration: improving policy as better π' by policy evaluation and policy improvement

Classic reinforcement learning have various types of algorithms, but they usually have one common weakness: not scalable (or practical) with high number of state representations. Here comes deep reinforcement learning, which became popular after the DQN paper of DeepMind (Mnih V. et al 2013). Deep reinforcement learning combines deep neural network with reinforcement learning techniques to address issue of high number of state representations. As explained in Ch. 2.2, Gomoku reinforcement learning is justified to use neural network. Further note that there are additional difficulties to learn Gomoku because it is a zero-sum 2-player game, meaning that algorithms directly maximizing rewards of an AI agent won't work, as high reward of one AI player implies the opponent, also the AI itself, has done poorly in self-play.

2.4.2 AlphaGo Zero / Alpha Zero

Silver D. et al (2017a, 2017b) published AlphaGo Zero and AlphaZero papers (almost the same algorithm, the latter applied to different games incl. chess and Shogi), in which the AI learnt games from scratch (without using domain specific knowledge) based on neural

network of CNN structure for computing prior action probabilities and estimated value of state, with Monte Carlo Tree Search used as policy improvement operator and self-play of games as policy evaluation operator. They defeated their predecessor, AlphaGo, which had defected the world's number one player in Go. Since then AlphaGo Zero / AlphaZero became popular baseline algorithm for board game reinforcement learning.

For reinforcement learning of Gomoku, my baseline algorithm mainly followed that of AlphaGo Zero and Alpha Zero papers (Silver D. et al 2017a, 2017b). Specifically, it was initially planned to adopt the general code with implementation of Alpha(Go) Zero for any game (but need to customize for Gomoku environment by Pytorch) under the MIT license and well documented by Stanford University accompanied by "A Simple Alpha(Go) Zero Tutorial", summarizing the model and algorithm used in Alpha(Go) Zero as follows (Nair S. 2017) :

Neural network

The neural network feature extractor is CNN architecture. It takes state s_t as input and yields value of state $v_\theta(s_t) \in [-1, 1]$ and policy $\overrightarrow{p_\theta}(s_t)$) as probability vector over all possible actions. It has the following loss function (excl regularization terms):

loss = $\sum_t (v_\theta(s_t) - z_t)^2 - \vec{\pi}_t \cdot \log(\overrightarrow{p_\theta}(s_t))$

, where z_t, $\vec{\pi}_t$ are final outcome {-1,0,1} and estimate (to be discussed below) of policy from state s_t respectively, with 1, 0, -1 representing win, draw, lose respectively for current player.

Monte Carlo Tree Search (MCTS) as policy improvement operator

At each node, action is chosen by maximizing U(s, a), the upper confidence bound of Q-value Q(s, a), calculated by: $U(s, a) = Q(s, a) + C * P(s, a) * \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$

where N(s, a) = no. of times taking action a from state s in MCTS simulation, P(s, .) = $\overrightarrow{p_\theta}(s)$, and the policy estimate of probability is improved by using $\vec{\pi}_t = N(s, .) / \sum N(s, b)$

When a new node (not visited before from parent node) is reached, instead of rollout, the value of new node is obtained from neural network and propagated up the search path. Unless the new node is terminal node, the new node is expanded to have child nodes.

Self-play training as policy evaluation operator

For each turn of the game, a fixed number of MCTS simulations are done from the state s_t, and action is selected by sampling from the improved policy estimate of probabilities. So,

training sample data can be obtained from the environment. At the end of an iteration, new neural network is updated by learning from the training sample data.

The evaluation metric in this research would be based on winning percentage and drawing percentage of the trained AI (against an independent evaluation agent), which was also one of the key evaluation metrics (strictly speaking, numbers of winning, drawing and losing out of total 100 games instead of percentages were presented in their papers) in the two papers (Silver D. et al 2017a, 2017b).

2.5 Novelty and related work

To the best of my knowledge, the proposed switchable lightweight anti-symmetric process is novel. On lightweight capsule, DSC-CapsNet was proposed as lightweight capsule network, which focused on computing efficiency and reducing number of parameters (Dan S. et al 2021); Sun K. et al (2021) proposed dense capsule network with fewer parameters – neither had the novel structure proposed in this study. On symmetric CNN, Hu X. S., Zagoruyko S. and Komodakis N. (2018) proposed to impose symmetry in neural network parameters by repeating some parameters and achieved 25% reduction in number of parameters with only 0.2% loss in accuracy using ResNet-101, a type of CNN; but unlike SLAP, symmetry was not imposed in the inputs. Bergman D. (2019) incorporated symmetry into neural network by creating symmetry (of specific type) invariant features, but no implementation or idea similar to SLAP was used.

2.6 Symmetry, Disentangled Representation and Gomoku Representation

Symmetry is one of the natures of the real world. Animals can detect the same object or the same prey being moved (translated), or even rotated after being slapped (my novel method was deliberately abbreviated as SLAP). Recognising symmetry can also speed up learning patterns, a typical trick used for playing some board games. On the other hand, failing to recognise or exploit symmetric variants can increase the number of training samples significantly, both to human and machine. To facilitate research exploiting symmetry in machine learning, Irina Higgins et al connected symmetry transformations to vector representations using the

formalism of group and representation theory to arrive at the first formal definition of disentangled representations (Higgins I. et al 2018), expected to benefit learning from separating out (disentangling) the underlying structure of the world into disjoint parts of its representation. Upon this work, Caselles-Dupré H., Garcia-Ortiz M. and Filliat D. (2019) showed by theory and experiments that Symmetry-Based Disentangled Representation Learning could not only be based on static observations: agents should interact with the environment to discover its symmetries. They emphasized that the representation should use transitions rather than still observations for Symmetry-Based Disentangled Representation Learning. This was taken into account when the representation of Gomoku was designed for reinforcement learning in this research.

2.7 SLAP, Symmetry and Artificial General Intelligence

One may expect that an artificial general intelligence (AGI) system, if invented, should be able to learn unknown symmetry. Researchers have worked on this, for example Anselmi F. et al (2017) proposed learning unknown symmetries by different principles of family of methods. But it is equally important to learn by exploiting symmetry more effectively. Imagine in the future that an AGI system can interpret the rules of Gomoku and realize from the rules that Gomoku is reflection and rotation invariant. Will it still waste time learning Gomoku by assuming such symmetry is not known? Of course not and it should directly exploit such symmetry. My vision is that such exploitation should be switched on easily if one wishes, and hence the term 'switchable' in SLAP, which can be used upon any function or model. If transfer learning in CNN is analogous to reusing a chair by cutting the legs and installing new legs to fit another, such 'switchable learning' in SLAP is analogous to turning the switch of an adjustable chair to fit certain symmetries. Such kind of 'switch ' in design can also help AI be more explainable and transparent, and more easily reused or transferred, while an AGI system should be able to link and switch to different sub-systems easily to solve a problem . SLAP can also reduce memory other than training samples. For example, AlphaGo Zero used a transposition table (Silver D. et al 2017a), a cache of previously seen positions and associated evaluations. Had SLAP been used instead of data augmentation, such memory size could be reduced by a factor of 8, or alternatively 8 times more positions or states could be stored. Indeed memory plays an important role in reinforcement learning as well by episodic memory, an explicit record of past events to be taken as reference for making decisions, improving both

sample efficiency and speed in reinforcement learning as experience can be used immediately for making decisions (Botvinick M. et al 2018). It is likely that an AGI system would, just like human, use memory to solve some problems rather than always resort to learning from scratch. And in the real word, a continuous space, there can be much more than 8 equivalent variants. Recently, Higgins I., Racanière S. and Rezende D. (2022) suggested in a paper that symmetry should be an important general framework that determines the structure of universe, constrains the nature of natural tasks and consequently shape both biological and artificial general intelligence. In the same paper (Higgins I., Racanière S. and Rezende D. 2022), they argued that symmetry transformations should be a fundamental principle in search for a good representation in learning. Perhaps my novel method SLAP may contribute a tiny step towards AGI (though the exact journey to AGI is unclear to the academic community), by shaping input representations directly by symmetry transformation. Note that SLAP can be used upon any function or model and even if some (types) of the outputs are not invariant but follow the same transformation, these may be broken down and use the transformation information output from SLAP to make appropriate transformation back later for these parts only. A little kid often mistakes *b* for *d* at the beginning of learning alphabets, and it appears that human learning types of objects by vision might naturally assume symmetry first and then learn non-symmetry later. If a machine learning problem is to be split into stages or parts by specified symmetry as a guide, SLAP might help by wrapping certain parts of a function or neural network model.

3. Methods

In this research, Pytorch[1] was used because of its object-oriented and pythonic style, and it is one of the most widely supported deep learning framework. Experiments were done by single GPU of Hyperion from City, University of London, and of Jarvislabs, a GPU cloud provider (see Appendix for hardware details), depending on availability and budget. To save computation, mini Gomoku board 8x8 (same size as chess) was used instead of the standard board size 15x15.

3.1 Switchable Lightweight Anti-symmetric Process (SLAP)

SLAP was the novel method to be tested. It forces the input of different variants (e.g. different rotation angle) to give the same output variant (and output the transformation information e.g. angle, though not necessarily used). There can be multiple ways to achieve this. For rotation and reflection variants of Gomoku states, one way to implement this is simply flattening the pixels of 8 variants to 8 lists, compare the lists and always choose the largest. Abbreviated as "slap" in coding, meaning an object after a slap is still the same object. Below (Fig. 5) was the algorithm used for SLAP in dealing with rotation and reflection variants of Gomoku states, but the concept may be applied to other symmetries as well. See example in Fig. 6.

Algorithm SLAP

1: Generate symmetry variants of input image/state, store required transformation
2: Convert each variant to a list
3: Compare each list and find the 'largest' list
4: **return** the 'largest' *variant* & required transformation of the variant

Fig. 5 SLAP algorithm

Suppose the specified symmetry is reflection and rotation and below is the sample input:

```
0 0 0 0
0 0 0 0
0 0 1 0
0 0 1 0
```

Generate all 8 variants of the input image or state:

```
0 0 0 0    0 0 0 0    0 1 0 0    0 0 0 0
0 0 0 0    0 0 1 1    0 1 0 0    0 0 0 0
0 0 1 0    0 0 0 0    0 0 0 0    1 1 0 0
0 0 1 0    0 0 0 0    0 0 0 0    0 0 0 0
```

[1] Completely reproducible results are not guaranteed across PyTorch releases, individual commits, or different platforms. Results may not be reproducible between CPU and GPU executions, even when using identical seeds.

0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Then convert to list for comparison. Note that they are compared as if each is flattened and the first element of each list is compared first.

So the output variant from SLAP (i.e. the 'largest' variant) is:
0 1 0 0
0 1 0 0
0 0 0 0
0 0 0 0 , where required transformation is rotation (90 degrees anti-clockwise) by 2 times

Fig. 6 SLAP example

If the image or state has multiple input channels or planes in one sample, the first channel/plane is compared first during list comparison.

SLAP was implemented by numpy instead of torch tensor for faster speed, because internally numpy uses view for rotation and reflection while torch tensor uses copy. The output variant replaced the input state when SLAP was applied to Gomoku reinforcement learning. During inference time, output action probabilities from neural network would be transformed back using the transformation information (i.e. rotation and reflection) from SLAP.

3.1.1 Invariance

Denote s, t = slap(x_i), where slap is SLAP function, s is the symmetry (of certain group G) variant and t is the corresponding transformation information. Equality holds for all i given property of slap, i.e. s, t = slap(x_1) = slap(x_2) = slap(x_3) …

Denote s = slap(x_i)[0], t = slap(x_i)[1], the pythonic expression to capture first and second return variables of a function respectively.

Denote h(slap(x_i)[0]) as $h^{slap}(x_i)$ for any function h.

Given an arbitrary function y = f(x).

$y = f^{slap}(x_i) \Rightarrow y = f(slap(x_i)[0]) \Rightarrow y = f(s)$ for all i

Hence, $y = f^{slap}(x_i)$ is invariant with respect to i, i.e. invariant to symmetry (of group G).

When this arbitrary function f is the neural network function, the composite function resulting from the neural network, f^{slap}, is invariant to symmetry (of corresponding group G).

3.1.2 Pixel-wise Analysis and Differentiability

Here SLAP specifically refers to the algorithm or function used for mini Gomoku.
Suppose there is only one non-zero pixel in mini Gomoku board (8x8) at position (h, w).
First consider the case when the pixel value is positive.
Denote new position of the 'largest' variant by only reflections (if needed) as (a, b).
Then (a, b) = (min(h, 7-h), min(w, 7-w))
Denote new position of the 'largest' variant further by transpose (if needed) as (x,y).
Then (x, y) = (min(a, b), max(a,b))
i.e. x = min(min(h, 7-h), min(w, 7-w))
y = max(min(h, 7-h), min(w, 7-w))
Note that (x, y) is also the new position of the pixel after SLAP, because reflection and transpose variants collectively are the same set as the 8 reflection and rotation variants.
Consider when the pixel value is negative. The calculation for (x,y) is the same except reversing min and max functions.
Also note that min & max functions can be expressed as ReLU functions:
max(a, b) = max(a-b, 0) + b = ReLU(a-b) + b
min(a, b) = a + b – max(a, b) = a – ReLU(a-b)
Hence, new position (x, y) after SLAP is differentiable like ReLU except at the point(s) where input of ReLU is zero after transformation to ReLU functions.

When there are multiple non-zero pixels, if there is unique largest pixel value that is positive, simply calculate the new position of this pixel using above formula, and follow the corresponding transformation (if unique) for all other pixels. If there are multiple largest pixels value that are positive, choose the one with earliest position (if unique) and follow its corresponding transformation (if unique) for all other pixels. In other cases, it would become more complicated to consider the positions pixel-wise.

Choosing the largest pixel would involve argmax, which is not directly differentiable, although it can be handled by using softmax and/or Gumbel-softmax as differentiable approximation to argmax (Jang E., Gu S. and Poole B. 2017). Another related technique is differentiable sorting and ranking by approximation and/or torchsort (Blondel M. et al 2020; Koker T. and Betz R. 2021). Since SLAP was not applied to intermediate layers of neural networks for Gomoku and so its differentiability was not required in this research, I would leave differentiability

implementation of SLAP to future work, which might involve techniques such as Gumbel-softmax and torchsort.

3.1.3 Groupoid and SLAP-CC

It is also interesting to experiment with translation variants, which are considered to be groupoid instead of group because Gomoku is only 'partially' invariant to translation. It was less prioritized in the work plan due to limited time and its more challenging nature. There can be many more translation variants than rotation and reflection variants, see Ch. 2.3. To save computation, different algorithm (specifically, crop and centre) was used to 'standardize' translation variants, denoted as SLAP-CC in the below, to emphasize that it shared the same general idea as SLAP, but just different way for implementation. This function was denoted as 'cc' in the code.

The algorithm of SLAP-CC, shown in Fig. 7, would concentrate experience around the centre, as input variant was centred to become output variant. If it could not be exactly centred, the algorithm would make it slightly lean to top left.

Algorithm SLAP-CC

1: Find non-empty min & max row index, min & max column index in input image
2: r_shift = (no. of rows – 1– min row index – max row index) // 2
3: c_shift = (no. of columns – 1– min column index – max column index) // 2
4: **return** numpy.roll(image, (r_shift, c_shift), axis=(-2, -1))

Fig. 7 SLAP-CC algorithm

Note that since Gomoku is not completely invariant to translation, SLAP-CC was used to add information as additional planes instead, as opposed to replacing the input state when SLAP was applied. 2 planes representing stones of different colours (current and opponent players respectively) centred together by SLAP-CC, followed by 2 planes representing original indices for vertical and horizontal positions respectively (scaled linearly to [1, -1]) were added along with original 4 planes in Gomoku state representation (see Ch. 3.2). The scaled position indices for whole plane were to give neural network a sense of original positioning.

3.2 Representation of Gomoku

In this research, the representation of Gomoku followed the style of AlphaGo Zero / AlphaZero, which used two 2D-planes to represent position of Go stones of current player and opponent player respectively by one-hot-encoding, repeated over the last 8 time-steps, finally followed by one plane indicating the colour of current player, all filled with 1 if black otherwise 0; information over a few time steps was required because Go has specific rules regarding repeated game situations (Silver D. et al 2017a, 2017b). In Gomoku, there are no such similar repeated game situations, so state at the latest time-step would be complete information for any player to decide an action. Initially, I planned to simplify to only one time-step with colour plane. But Caselles-Dupré H., Garcia-Ortiz M. and Filliat D. (2019) showed that for symmetry-based disentangled representation (o_t, a_t, for observation and action respectively at time t), one should not use a training set composed of still samples (o_t , o_{t+1} , ...), but rather transitions ((o_t, a_t, o_{t+1}), (o_{t+1}, a_{t+1}, o_{t+2}), ...). So a plane representing last action (i.e. position of last stone) was also included in Gomoku state representation in this research. Information at previous time-step was not included as it would always be a linear combination of planes of current time-step and last action given the rules of Gomoku, i.e. just a redundant input feature for Gomoku.

Together there were 4 planes for each sample of Gomoku states, representing current player stones, opponent stones, last action and current colour respectively by one-hot-encoding. See Fig. 8 for a typical Gomoku state representation in this research, which used 8x8 board instead.

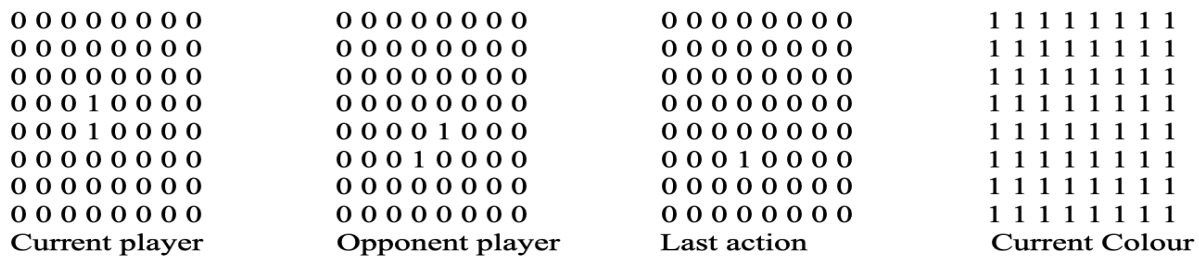


Fig. 8 Gomoku state representation example at time t = 4

On label representations, probability of each move was represented by 8x8 flattened vector. Final outcome (or value) of each state was represented by 1, 0, -1 respectively for win, draw, lose, from the perspective of current player.

3.3 Testing Benefits for Neural Network Learning

Before testing SLAP in reinforcement learning, we should test whether it would benefit neural network learning at all. Completely random states would be difficult to learn, so synthetic states of Gomoku were created for testing neural network learning with SLAP vs with typical data augmentation (by rotation and reflection), the latter of which had 8 times the number of training samples. Self-play was not involved in this testing.

Synthetic states were generated by first creating states each with only 5 stones connected in a straight line (i.e. win status) for all combinations for current black player, then removing one stone (to be repeated 5 times with different stones to create 5 different states) and randomly adding 4 opponent stones to become one about-to-win state. Together these were one set of 480 about-to-win states. Different sets of about-to-win states could be created since white stones were merely random and the combinations far outnumbered those of black stones placed as mentioned. Each set was mixed with 1000 purely random states, also with 4 stones for each player. Roughly two-third were purely random states. 8 mixed sets were created, i.e. 11,840 samples. 15%, i.e. 1,776 samples, were reserved for validation test.

Labels were assigned as follows: if there were one or more possible moves to win immediately (include some purely random states, though the chance would be very remote), the value of state would be labelled as 1 and the wining move(s) would be labelled with probability of move = 1/no. of winning moves, while those of other moves were labelled 0; otherwise the value of state would be labelled as 0 and the probability of move for each available move would be random by uniform distribution, normalizing and summing to 1.

Neural networks with SLAP vs with data augmentation would learn from training samples of states and labels to predict labels of validation data given the input states. Validation loss and its speed of convergence would be the key metrics. To better utilize limited computation resources, more iterations of training were reserved for promising models from first stage.

First, at preliminary stage, for each set of hyperparameters the neural network ran 1000 iterations each with batch size 512 sampled from training samples of size 10,064 and 80,512 respectively for neural networks with SLAP and neural networks with data augmentation. Sampled with replacement, same as during reinforcement learning. There were 2400 combinations of hyperparameters by grid search, shown in Fig. 9:

Hyperparameter	Tested values	Remarks
use_slap	True, False	False: data augmentation instead of SLAP
extra_act_fc	True, False	True: add extra layer (size 64) to action policy
L2	10^{-3}, 10^{-4}, 10^{-5}	weight decay of optimizer
Num_ResBlock	0, 5, 10, 20	no. of residual blocks
SGD	True, False	False: Adam optimizer instead of SGD
lr	10^{-1},10^{-2},10^{-3},10^{-4},10^{-5}	Learning rate
dropout	0, 0.1, 0.2, 0.3, 0.4	

Fig. 9 Hyperparameters tested at preliminary stage

If Num_ResBlock > 0, the residual blocks replaced the common CNN layers and added a convolutional layer of 256 filters (3x3 kernel, stride 1, padding 1, no bias, ReLU activation) as the first layer. Autoclip was not used in the optimizer, unlike reinforcement learning.

At stage 2, selected models from previous stage would run for 10,000 iterations instead of 1,000 iterations.

3.4 Baseline Reinforcement Learning Algorithm

The baseline algorithm for learning Gomoku followed AlphaGo Zero/AlphaZero algorithm briefly described in Ch. 2.4, but there were additional implementation details as there were subtle differences between AlphaGo Zero and AlphaZero, taken into account as in Fig. 10:

	AlphaGo Zero (Silver D. et al 2017a)	AlphaZero (Silver D. et al 2017b)
Pitting models	Yes, model with new weights plays against previous one; new weights are adopted only if it wins 55% or above	No, new weights are always adopted after each iteration of neural network learning
Symmetry	Data augmentation by rotation and reflection to increase sample size by 8 times for training; transform to one of 8 variants randomly in self-play for inference	Not exploited, as it is intended for generalization
Action in self-play	Sampled proportional to visit count in MCTS in first 30 moves, then selected greedily by max visit count (asymptotically with highest winning chance) in MCTS	Sampled proportional to visit count in MCTS

Outcome prediction	Assumes binary win/loss, estimates & optimises probability of winning	Also considers draw or other outcomes, estimates & optimises expected outcome

Fig. 10 Differences between AlphaGo Zero and AlphaZero

Among these differences, the baseline algorithm in this research followed the better version, and thus followed AlphaZero except on symmetry exploitation. Like AlphaGo Zero, my baseline exploited symmetry by data augmentation to increase number of training samples by 8 times, but random transformation was not done in self-play as it seemed not to improve results but more time-consuming in initial informal trials. Note that it might seem intuitive to better retain the practice of pitting models, but AlphaZero paper showed that time spent on pitting models (quite significant time) would be better spent on continuous self-play for better performance as it outperformed AlphaGo Zero despite not using data augmentation (Silver D. et al 2017b). It was decided not to use AlphaGo Zero's 2-stage sampling for action in self-play and to simply follow AlphaZero's simpler approach after some small trials, and better spent efforts tuning other hyperparameters.

3.5 Code Implementation

3.5.1 Code to be adapted

As the objective was to test the novel method SLAP applied to a certain neural network, the codes available for baseline model would be adopted and modified for this research, instead of building from scratch. Among many open source codes implementing AlphaZero, a popular well-documented code by Song J. (2017) using AlphaZero algorithm for Gomoku was found and adopted for upgrade, replacing the initial planned code mentioned in Ch. 2.4, which required building Gomoku environment by Pytorch from scratch. The adopted code had 6 components:

Game – setting the environment of the game allowing players to take actions to effect changes on the board, checking winner and terminal status of game, with capability to display the game if needed.

Policy – setting the neural network structure, with function to optimize the neural network in a step, and function to process and output estimated probability of moves and value of state by neural network from input board state.

MCTS by AlphaZero – Monte Carlo Tree Search (MCTS) conducted by AlphaZero agent, using policy-value neural network to guide the tree search by probability of move as initial prior probability and to evaluate leaf nodes.

Pure MCTS – Monte Carlo Tree Search (MCTS) conducted by evaluation agent, using purely random policy for initial prior probability.

Train – self-play training, by policy and MCTS to take actions in games, updating policy after each (batch of) game(s), with regular evaluations by playing against random MCTS.

Human play – allowing human to play against the AI agent or random MCTS agent.

3.5.2 Upgrade on adapted code

The adapted code could already implement the basic algorithm mentioned in 3.4, but about a hundred small trials were conducted for upgrade, testing and fixing bugs in this research before the major experiments. Major changes or upgrade to the adopted code were as follows:

SLAP

- Added this new component to implement SLAP, see Ch. 3.1.

Game

- Aligned with AlphaGo Zero to allow temperature parameter (affected sampling /exploring actions in self-play) to be fed differently at 2 stages within a game (Silver D. et al 2017a) (though finally followed AlphaZero and not used it in major experiments, one of few differences between AlphaGo Zero and AlphaZero).
- Fixed bugs that mixed up width and height (though not affect boards of square shape).
- Speeded up checking of game terminal status by using the fact that previous state must be non-terminal, and checking only lines connected to last action (stone).

- Used array-index style instead of coordinate style for location of a move, to avoid flipping upside down for some internal array or tensor computations in policy component, and changed graphic display accordingly.

Policy

- Added option to use SLAP and configure noise
- Added network options: evaluation mode, dropout, number of residual blocks (0: same as original adopted code; AlphaGo Zero (Silver D. et al, 2017a, p. 27) used 19 or 39 residual blocks), extra FC layer for action, SGD & AdamW optimizer options in addition to Adam
- Calculated validation loss under evaluation mode
- Used autoclip to gradients for regularization (Seetharaman P. et al 2020)
- Doubled the speed by making output consistent as numpy instead of tensor
- Added option to normalize or not the prior probabilities before MCTS (adapted code missed to normalize this but still could learn, perhaps because relative prob could still guide the search of MCTS somehow) .
- Added option to vary noise linearly against leaf value, i.e. underdog played more randomly, so as to try breaking predicted outcome to facilitate new findings.
- Applied noise after (instead of before as in adapted code) masking out illegal actions.
- Fixed deprecation warning regarding log_softmax, tanh.
- Removed flipping upside down as game state was no longer stored upside down.

MCTS by AlphaZero

- Added Dirichlet noise at the root node to align with papers (Silver D. et al 2017a, 2017b) (adapted code applied noise after MCTS instead of during MCTS; I applied both noise options)
- Reused sub-tree in other moves within the same game during evaluation as well, in addition to during self-play

Pure MCTS

- No important changes.

Train

- Added option to apply SLAP to transform training samples.
- Used multi-processing to speed up evaluation by multiple cores; not applied to self-play as it slowed down, suspectedly due to frequent switching memory in MCTS.

- Calculated validation loss for game states freshly generated after one (batch of) self-play game(s) and immediately before sampled for training; games states from playing with evaluation agent were neither used for calculating validation loss nor training as the action probabilities of evaluator agent's pure MCTS had different distribution.
- Changed evaluation to multi-tiers.
- Allowed more hyperparameters to change instead of being hard-coded.
- Added option to adaptively decrease learning rate when validation loss increased significantly, measured by standard score
- Exported data buffer, loss and result data.
- Sampled different batches instead of using same batch for multiple iterations of neural network update.

Human Play

– Updated script files above and changed configuration accordingly.

3.5.3 Additional codes

AI vs AI – a script was created by allowing two AI agents to play against each other, with same setting as if each was playing against an evaluator in the evaluation

Synthetic states – a script was used to generate synthetic Gomoku states for testing neural network learning (see Ch. 3.3 Testing Benefits for Neural Network Learning), with another script training the network given these synthetic states, calculating and finally exporting training and validation loss.

Optimizing speed – I created a separate version to pre-compute game states and variant variables and re-use symmetry variants by the view nature of numpy rotation and flip (the view nature enabled updating one game state to automatically update other variants during self-play), aiming at optimizing the speed to use SLAP. With time constraint and limited GPU resources, this version was not fully tested and used in major experiments, but it showed that overhead cost of SLAP on reinforcement training could become further insignificant.

Analysis – additional coding was created in JupyterLab to process exported data for analysis and illustration by charts and tables.

3.6 Neural Network Architecture and Configurations

Unless otherwise specified, the following architecture and configurations were used for experiments:

The neural network consisted of 3 common convolutional layers (32, 64, 128 filters respectively) each with 3x3 kernel of stride 1 and padding 1 with ReLU activation, followed by 2 action policy players and in parallel 3 state value layers. The input was 8 x 8 x 4 image stack comprising of 4 binary feature planes. The action policy layers had one convolutional layer with 4 filters each with 1x1 kernel of stride 1 with ReLU activation, followed by a fully connected linear layer to output a vector of size 64 corresponding to logit probabilities for all intersection points of the board. The state value layers had one convolutional layer with 2 filters each with 1x1 kernel of stride 1 with ReLU activation, followed by fully connected linear layer to a hidden layer of size 64 with ReLU activation, finally fully connected to a scalar with tanh activation. Dropout, if any, would be applied to all action policy layers and state value layers except output layers; not applied to common layers.

Optimizer: Adam with autoclip (Seetharaman P. et al 2020)
Batch size per optimisation step: 512 (2048 in AlphaGo Zero)
Data buffer size: 10,000 for data augmentation, 1,250 for SLAP
No. of network optimisation steps per policy iteration: 10
No. of self-play games per policy iteration: 1
No. of playouts per move in MCTS: 400 (1600 in AlphaGo Zero, 800 in AlphaZero)
C_{puct} (constant of upper confidence bound in MCTS) : 5
Temperature parameter: 1 (same as AlphaZero)
Dirichlet alpha of noise: 0.3 (same as chess in AlphaZero)

Smaller batch size and number of playouts per move in MCTS were used because Gomoku is less complex than Go. Dirichlet alpha was initially set at 0.3 because mini Gomoku (8x8 board) has same board size as chess and similar number of available action choices per move.

3.7 SLAP in Gomoku Reinforcement Learning

SLAP was used to pre-process states as training samples for network training, and also used to pre-process input states for network inference. Transformation information from SLAP was only used with network inference to convert the probabilities (but not estimated state value/ outcome) back to corresponding game board positions for MCTS to choose action. See Fig. 11.

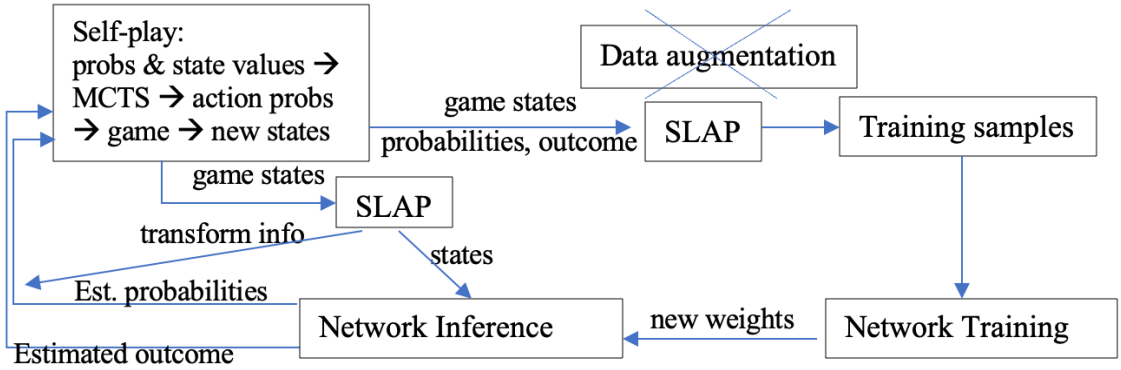


Fig. 11 SLAP used in Gomoku reinforcement learning

In SLAP-CC model, SLAP-CC was applied at the same place as SLAP in the above flow chart, but data augmentation was kept instead of being replaced and no transformation information was used to transform probabilities output of the network. See methods in Ch. 3.1.3.

3.8 Evaluation method

Unlike other reinforcement learning that can use reward as evaluation metrics, rewards during self-play would not be suitable for this 2-player zero-sum game. Independent agents would be needed to play against the AI agent for evaluation. The winning percentage (draw counted as half win) would be the evaluation metrics for the performance of the model. Data were generated through self-play, similar to those in the DeepMind papers (Silver D. et al 2017a, 2017b).

Independent agent(s), also called evaluation agent or evaluator, was built by pure Monte Carlo Tree Search (MCTS) with random policy to play against this AI. The strength of a pure MCTS agent depends on the number of playouts (aka. simulations) in each move. To facilitate observation during very initial stage, multi-tier evaluation was built by playing 10 games against 3 pure MCTS agents (30 games total), each with 1000, 3000, 5000 playouts respectively. The overall winning percentage (draw counted as half win) against these three agents would be the metrics.

For analysis, number of games needed in training for achieving certain winning percentage would be presented, showing how fast (or slow) the novel method learnt compared with control. Since our objective was to introduce SLAP to improve machine learning, we would focus on the relative performance of the two agents instead of the absolute performance playing Gomoku.

The choice of tested hyperparameters in reinforcement learning would take into account the results from testing on learning synthetics states. At first stage, each model would be trained by self-play of 250 games. Selected models would be trained by self-play of 5000 games in the second stage. Evaluation by playing against evaluator was done every 250 games.

Further, a script was created by allowing two AI agents to play against each other for 100 games, with the same setting as if each was playing against an evaluator in the evaluation.

For evaluation on learning synthetic states, see Ch. 3.3 Testing Benefits for Neural Network Learning.

4. Results

4.1 SLAP Function Outputs

In Fig. 12, left shows symmetry variants while right shows corresponding SLAP outputs.

```
[[ 0  1  2  3]   (array([[15, 14, 13, 12],
 [ 4  5  6  7]          [11, 10,  9,  8],
 [ 8  9 10 11]          [ 7,  6,  5,  4],
 [12 13 14 15]]         [ 3,  2,  1,  0]]), 'no_flip', 2)

[[ 3  7 11 15]   (array([[15, 14, 13, 12],
 [ 2  6 10 14]          [11, 10,  9,  8],
 [ 1  5  9 13]          [ 7,  6,  5,  4],
 [ 0  4  8 12]]         [ 3,  2,  1,  0]]), 'no_flip', 1)

[[15 14 13 12]   (array([[15, 14, 13, 12],
 [11 10  9  8]          [11, 10,  9,  8],
 [ 7  6  5  4]          [ 7,  6,  5,  4],
 [ 3  2  1  0]]         [ 3,  2,  1,  0]]), 'no_flip', 0)

[[12  8  4  0]   (array([[15, 14, 13, 12],
 [13  9  5  1]          [11, 10,  9,  8],
 [14 10  6  2]          [ 7,  6,  5,  4],
 [15 11  7  3]]         [ 3,  2,  1,  0]]), 'no_flip', 3)

[[ 3  2  1  0]   (array([[15, 14, 13, 12],
 [ 7  6  5  4]          [11, 10,  9,  8],
 [11 10  9  8]          [ 7,  6,  5,  4],
 [15 14 13 12]]         [ 3,  2,  1,  0]]), 'flip', 2)

[[ 0  4  8 12]   (array([[15, 14, 13, 12],
 [ 1  5  9 13]          [11, 10,  9,  8],
 [ 2  6 10 14]          [ 7,  6,  5,  4],
 [ 3  7 11 15]]         [ 3,  2,  1,  0]]), 'flip', 3)

[[12 13 14 15]   (array([[15, 14, 13, 12],
 [ 8  9 10 11]          [11, 10,  9,  8],
 [ 4  5  6  7]          [ 7,  6,  5,  4],
 [ 0  1  2  3]]         [ 3,  2,  1,  0]]), 'flip', 0)

[[15 11  7  3]   (array([[15, 14, 13, 12],
 [14 10  6  2]          [11, 10,  9,  8],
 [13  9  5  1]          [ 7,  6,  5,  4],
 [12  8  4  0]]         [ 3,  2,  1,  0]]), 'flip', 1)
```

Fig. 12 Symmetry Variants and SLAP Outputs

Not surprisingly, SLAP produced the same output variant for each variant in Fig. 12 as they belonged to the same group, but different transformation information required. SLAP function produced the correct 'largest' variant with the corresponding required transformation information. For example, the first one ('no_flip', 2) shows no flipping horizontally required, but rotation (anti-clockwise 90 degrees) by 2 times would be required to produce the output variant in the right, from the input variant in the left. See methods in Ch. 3.1.

```
array([[0., 0., 0., 0., 0., 0., 0., 0.], array([[0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 1., 1., 0., 0., 0., 0.],        [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 1., 0., 0., 0., 0.],        [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],        [0., 0., 1., 1., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],        [0., 0., 1., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],        [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],        [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.]])       [0., 0., 0., 0., 0., 0., 0., 0.]])

array([[0., 1., 1., 1., 0., 0., 0., 0.], array([[0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 1., 0., 0., 0., 0.],        [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],        [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],        [0., 0., 1., 1., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],        [0., 0., 1., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],        [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],        [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.]])       [0., 0., 0., 0., 0., 0., 0., 0.]])
```

Fig. 13 SLAP-CC Output

In Fig. 13, left shows input image while right shows output image by SLAP-CC. The non-zero elements in the input image shifted to centre, but slightly towards left as they were impossible to be exactly centred horizontally. As there were many translational variants, it would be too verbose to compute and display all variants, so only two were shown. See methods in Ch. 3.1.3.

In short, SLAP and SLAP-CC functions produced expected outputs.

4.2 Impact on Neural Network Learning

4.2.1 Preliminary Stage Testing

Fig. 14 shows validation losses for synthetic states (see methods in Ch. 3.3) over different hyperparameters. 2400 combinations of hyperparameters (see notation explanation and set of

values in Fig. 9) were tested and by choosing the best learning rate and dropout rate for each sub-group, 96 sub-groups of results were presented in Fig. 14, in which SLAP outperformed data augmentation counterparts in 77% of these sub-groups and underperformed in 23%, at comparison precision +/- 0.001.

		use_slap	False				True			
		Num_ResBlock	0	5	10	20	0	5	10	20
extra_act_fc	SGD	L2								
False	False	0.00001	2.879	3.817	4.346	4.371	2.873	3.725	4.370	4.336
		0.00010	2.896	3.676	4.340	4.368	2.881	3.665	4.330	4.330
		0.00100	2.904	3.529	4.366	4.371	2.883	3.544	4.360	4.328
	True	0.00001	2.945	3.544	4.364	4.413	2.954	3.519	4.395	4.383
		0.00010	2.921	3.804	4.431	4.373	2.910	3.332	4.371	4.374
		0.00100	3.050	3.855	4.382	4.390	2.886	3.336	4.341	4.362
True	False	0.00001	2.956	3.887	4.365	4.363	2.945	3.465	4.339	4.360
		0.00010	2.955	3.819	4.365	4.365	2.948	3.458	4.317	4.330
		0.00100	3.034	3.290	4.360	4.364	2.951	3.415	4.332	4.330
	True	0.00001	3.265	3.252	4.389	4.376	2.947	3.467	4.377	4.410
		0.00010	2.982	3.520	4.372	4.379	2.970	3.485	4.386	4.373
		0.00100	4.314	3.338	4.354	4.370	2.975	3.328	4.362	4.388

Fig. 14 Best validation losses for synthetic states – preliminary stage

In Fig. 14, models with Num_ResBlock 0 and extra_act_fc were better or no worse than counterparts, so they were selected for stage 2 testing. For comparison and caution purposes, models with Num_ResBlock 5 were also included. If we relaxed loss comparison precision to within +/- 0.01, SLAP models shared the same best learning rate and dropout rate with baseline counterparts, so these same set of learning rates and dropout rates were selected for further testing for apple-to-apple comparison.
(lr = 0.1 if SGD else 0.001; dropout = 0.3 if Num_ResBlock==5 else (0.1 if SGD else 0))

4.2.2 Stage 2 Testing

At stage 2, 24 selected models from previous stage ran for 10,000 iterations instead, with losses recorded every 10 iterations. See validation losses in Fig. 15.

	use_slap	False		True	
	Num_ResBlock	0	5	0	5
SGD	L2				
False	0.00001	2.816	3.088	2.817	2.944
	0.00010	2.805	3.059	2.813	2.882
	0.00100	2.813	2.893	2.814	3.132
True	0.00001	4.329	3.132	3.923	3.155
	0.00010	4.273	3.281	2.810	3.025
	0.00100	2.817	3.419	3.032	3.817

Fig. 15 Best validation losses for synthetic states – stage 2

In Fig. 15, the best few SLAP and baseline models converged to loss around 2.81. All these had Num_ResBlock 0, i.e. no residual blocks. So we would focus on these models, see Fig. 16 and Fig. 17.

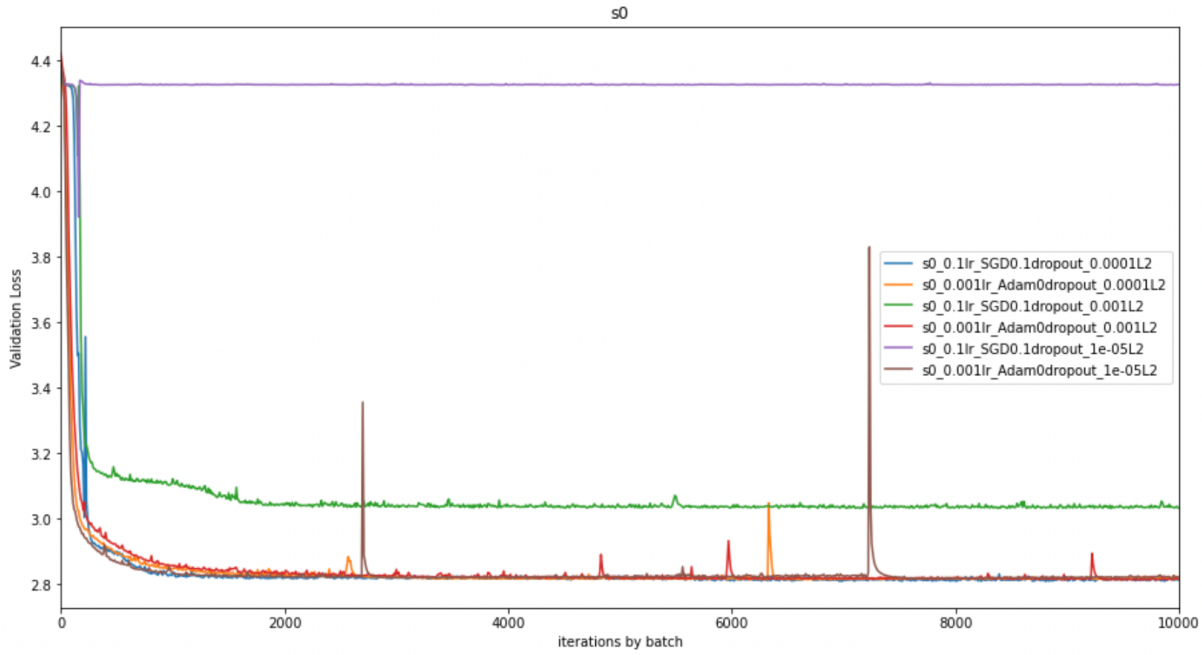


Fig. 16 Validation loss of s0 (models with SLAP, 0 Num_ResBlock) for 10,000 iterations

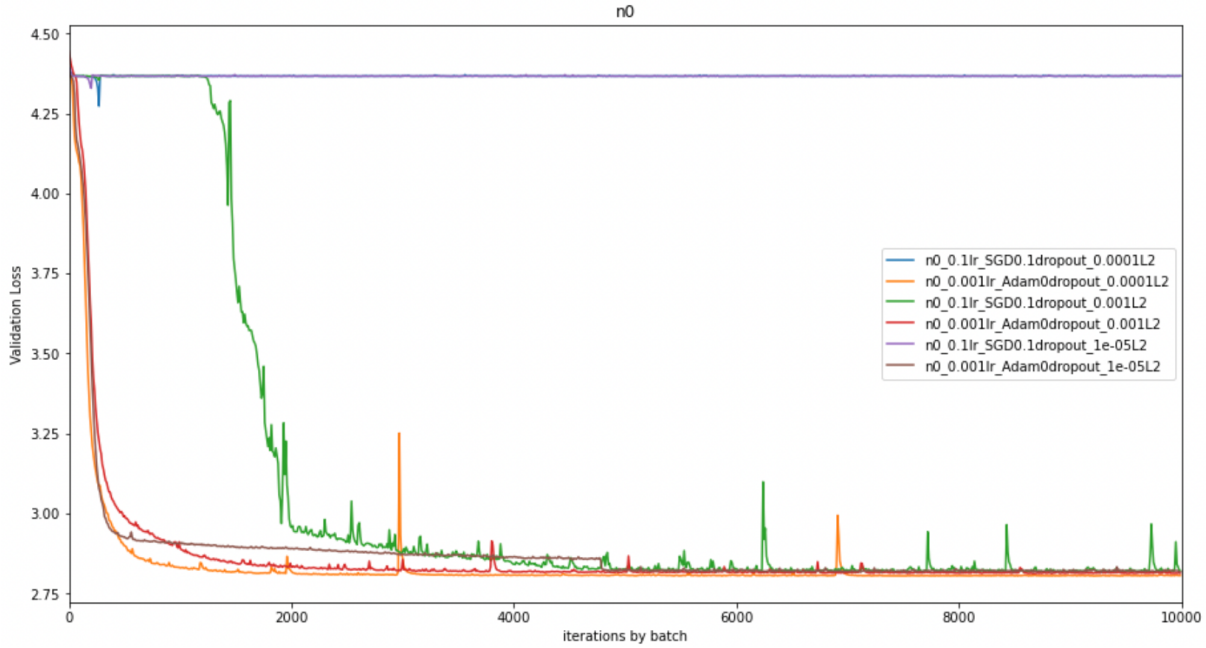


Fig. 17 Validation loss of n0 (models with no SLAP, 0 Num_ResBlock) for 10,000 iterations

Not all models with SGD optimizers converged well, while all models with Adam optimizers in Fig. 16 & Fig. 17 converged to near 2.8, so let's focus on the 6 models with Adam only for first 5000 iterations. In Fig. 18a, each model had Adam optimizer, same learning rate 0.001, no dropout , no residual blocks, but different values of L2.

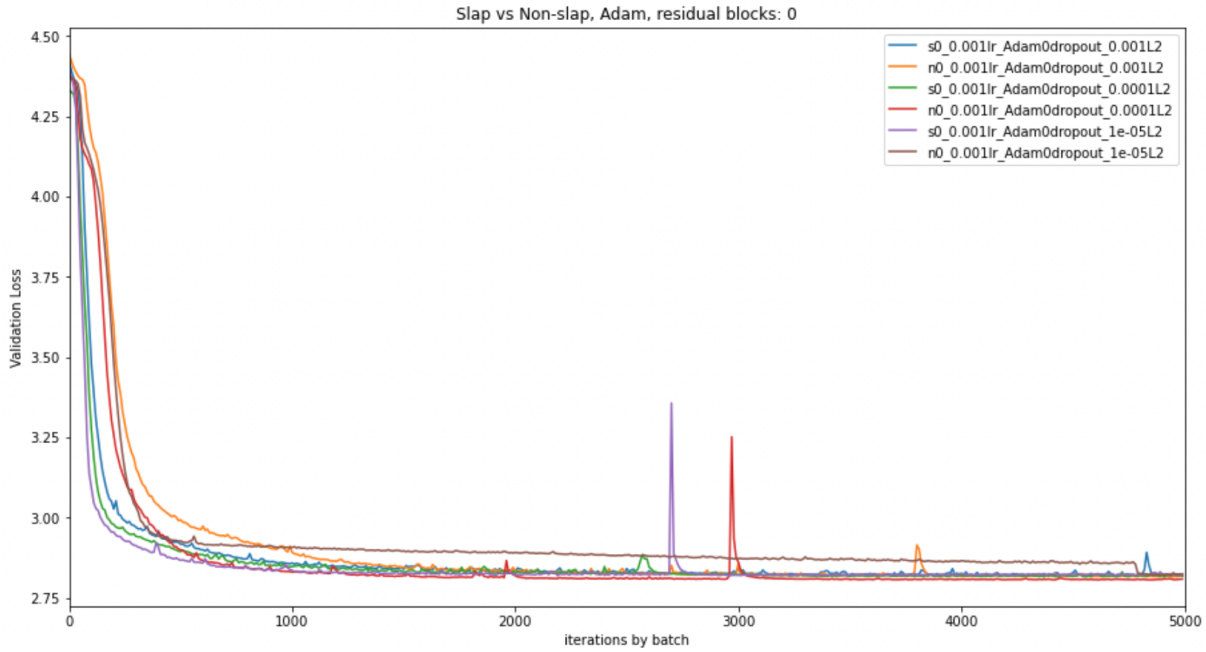


Fig. 18a Validation loss with SLAP vs non-SLAP (0 Num_ResBlock, Adam optimizer)

Let's check if training loss also converged to similar value, average over 3 records to roughly match sample size of validation dataset. See Fig. 18b.

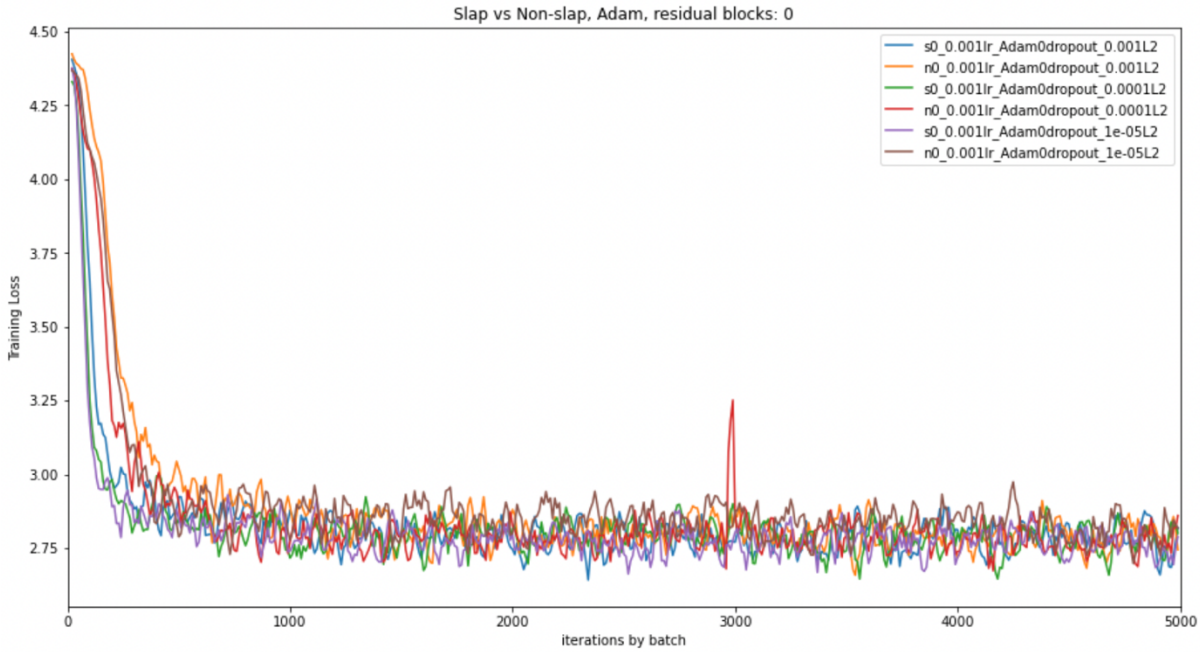


Fig. 18b Training loss with SLAP vs non-SLAP (0 Num_ResBlock, Adam optimizer)

Compared with validation loss, the training loss curves in Fig. 18b converged to similar value but it fluctuated more as it was measured based on batch instead of epoch.

Experiments were repeated 3 more times to calculate average time for convergence. In Fig. 19, the time for validation loss to reach 3.0 doubled roughly across different values of L2 consistently when data augmentation was used instead of SLAP. This consistency was also observed in the chart (Fig. 18a) during earlier iterations. The time to reach 2.9 fluctuated much more, but also improved with SLAP. On average, SLAP improved the speed of convergence by 95.1% and 71.2% measured by reaching 3.0 and 2.9 respectively.

	Time to converge to 3.0			Time to converge to 2.9		
L2	SLAP	Data augmentation	% Difference	SLAP	Data augmentation	% Difference
10^{-3}	225	485	90.2%	685	1168	70.4%
10^{-4}	180	380	111.1%	473	658	39.2%
10^{-5}	178	398	85.9%	415	868	109.0%
Average	204	397	95.1%	524	898	71.2%

Fig. 19 Time to converge, by number of iterations

In short, SLAP improved convergence speed in neural network learning by 83.2% in average.

4.2.3 Testing Sample Size

Holding validation data set unchanged, the training data sample size was reduced by holding out some samples for testing to match required size, using models with L2=10^{-4} in Fig. 18a. In Fig. 20 and Fig. 21, SLAP models were more vulnerable to decreasing number of training samples and it failed to keep converging when training sample size was 1258 or 2516.

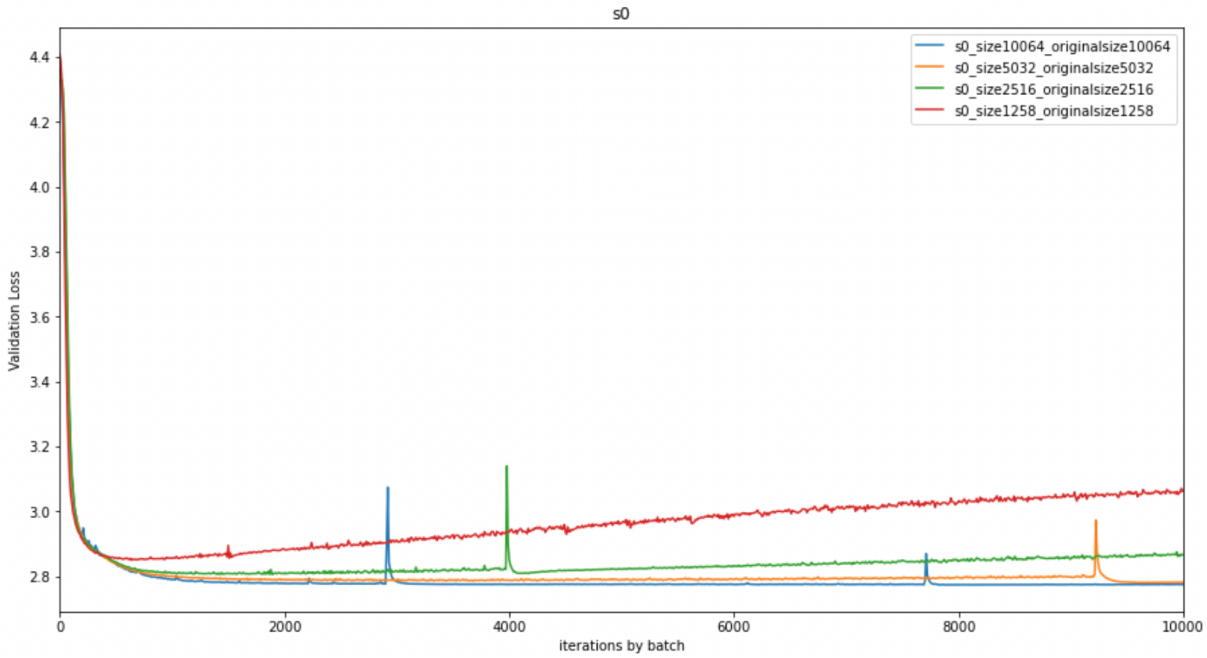


Fig. 20 Testing different training sample size for SLAP

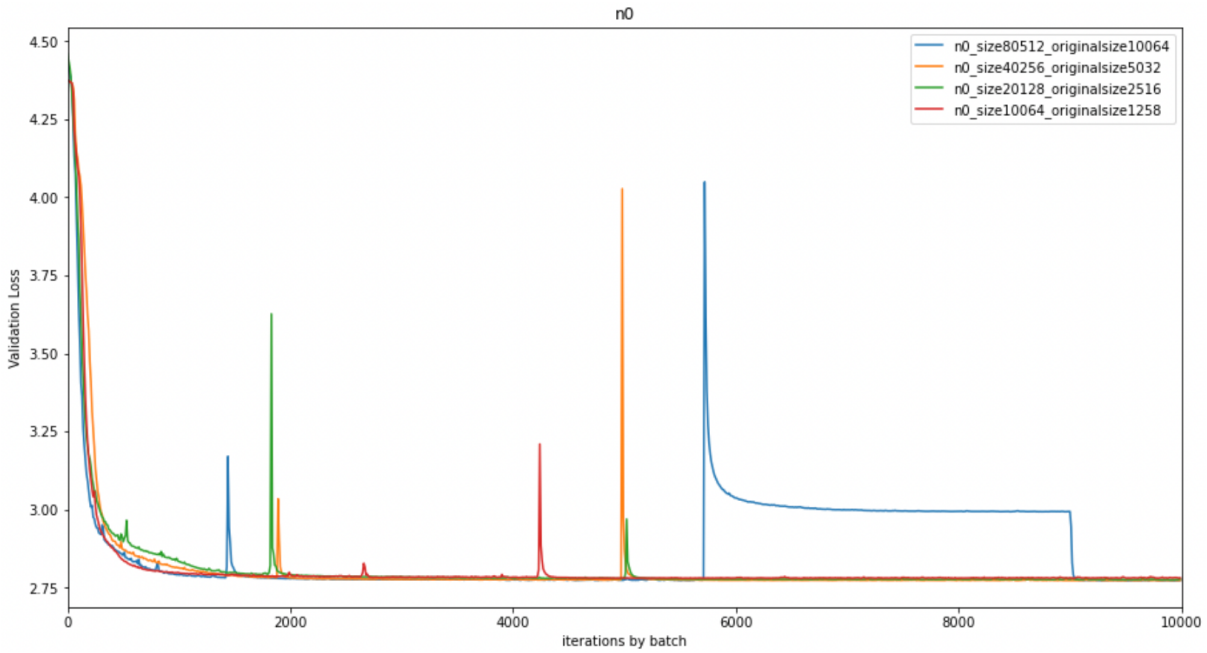


Fig. 21 Testing different training sample size for non-SLAP

4.2.4 Testing SLAP-CC

SLAP-CC was added to the 3 best baseline models in Fig. 18a of Ch. 4.2.2. See methods in Ch. 3.1.3. In Fig. 22a and 22b, both training and validation losses of SLAP-CC converged to around 2.8 for all 3 values of L2, similar to its baseline counterparts.

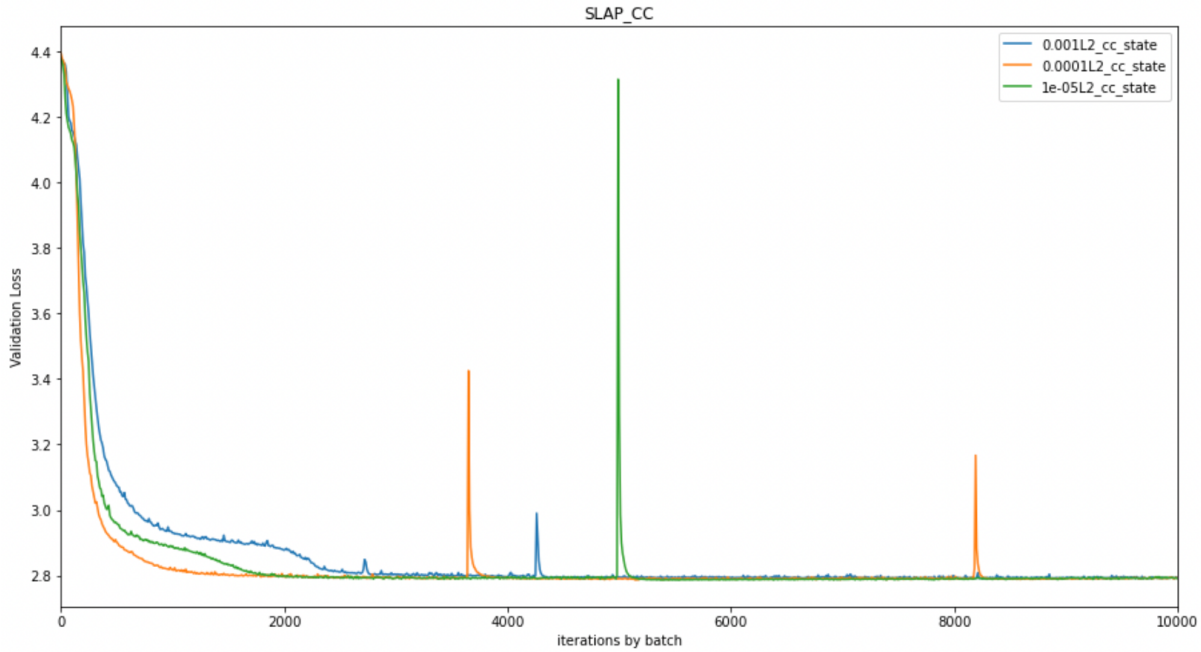


Fig. 22a Validation loss with SLAP-CC (0 Num_ResBlock, Adam optimizer)

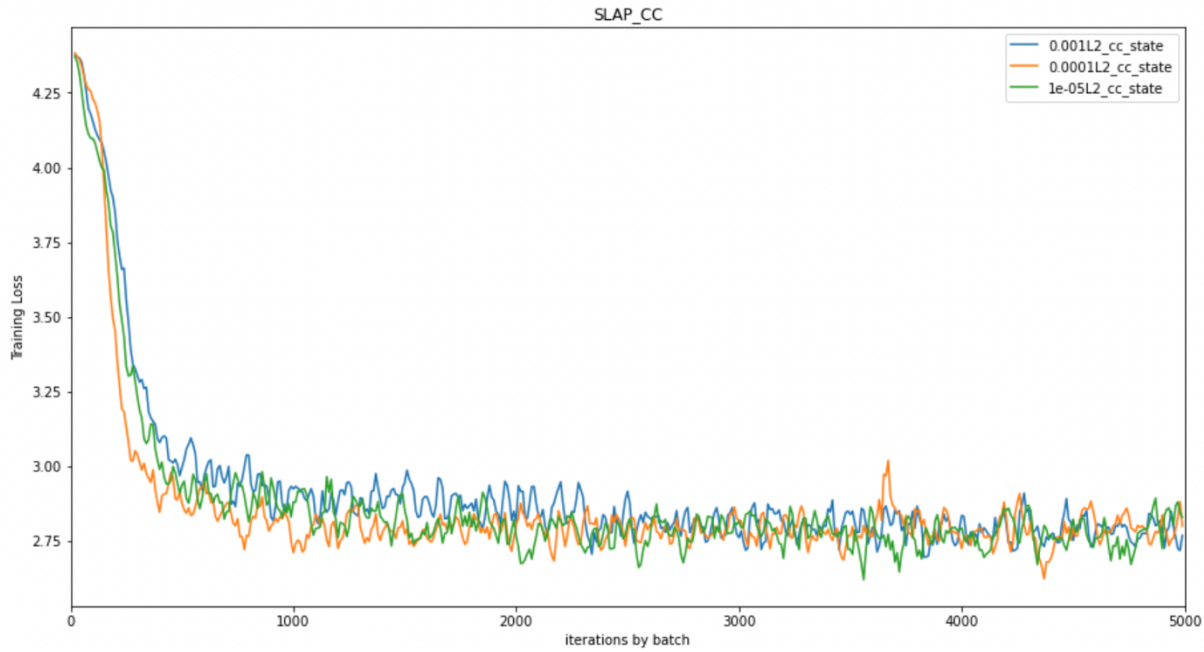


Fig. 22b Training loss with SLAP-CC (0 Num_ResBlock, Adam optimizer)
Moving average over 3 records to roughly match sample size of validation loss dataset.

Experiments were repeated 3 more times to calculate average time for convergence. In Fig. 23, the time for validation loss to reach 3.0 and 2.9 both worsened across different values of L2 when SLAP-CC was added.

	Time to converge to 3.0			Time to converge to 2.9		
L2	SLAP-CC	Data augmentation	% Difference	SLAP-CC	Data augmentation	% Difference
10^{-3}	728	485	-33.3%	1713	1168	-31.8%
10^{-4}	503	380	-24.4%	960	658	-31.5%
10^{-5}	495	398	-33.3%	1210	868	-28.3%
Average	575	397	-30.7%	1294	898	-30.7%

Fig. 23 Time to converge, by number of iterations

In short, SLAP-CC decreased convergence speed in neural network learning by 30.7% in average.

4.3 Impact on Reinforcement Learning

4.3.1 Stage 1 Testing

Reinforcement learning required much more computation than neural network learning in Ch. 4.2, so to save computation, the choice of hyperparameters for testing would be based on best models in Ch. 4.2, with some deviations for exploration. SLAP and baseline models (see methods in Ch. 3.1 & 3.4) would be tested by grid search over the same set of hyperparameters, except learning rate due to observations in small testing for upgrade, see Fig. 22.

Hyperparameter	Tested values	Remarks
use_slap	True, False	False: data augmentation instead of SLAP
explore	0, 0.25	Dirichlet noise weight *after* MCTS
noise	0, 0.25, (0.1, 0.4)	Dirichlet noise weight *at root node of* MCTS
Dirichlet	0.15, 0.3	Dirichlet alpha of noise *at root node of* MCTS
L2	10^{-3}, 10^{-4}	weight decay of optimizer
dropout	0, 0.2	
lr (learning rate)	SLAP: 10^{-3}, $5\text{x}10^{-4}$, $2.5\text{x}10^{-4}$ no SLAP: 10^{-3}, $2\text{x}10^{-3}$, $4\text{x}10^{-3}$	

Fig. 22 Hyperparameters tested for reinforcement learning, stage 1

Note: for models with noise 0, Dirichlet would not be tested over two different values as above because the noise would be disabled. For other hyperparameters, see configurations in Ch. 3.6.

At the first stage, each of 240 models would be trained by self-play of 250 games. Data buffer size was 1250 and 10,000 for SLAP and non-SLAP models respectively, both roughly equivalent to storing latest 60 games. Recall that non-SLAP models used data augmentation to

increase training sample size by factor of 8. For results, see models with highest winning ratios (see Ch. 3.8) for SLAP & non-SLAP respectively in Fig. 23.

SLAP models

Model	Win ratio	lr	dropout	L2	explore	noise	Dirichlet
s0_1	36.7%	$5x10^{-4}$	0.2	10^{-3}	0.25	(0.1, 0.4)	0.3
s0_2	33.3%	$2.5x10^{-4}$	0.2	10^{-4}	0	(0.1, 0.4)	0.15
s0_3	30.0%	$2.5x10^{-4}$	0	10^{-4}	0	(0.1, 0.4)	0.15
s0_4	30.0%	$2.5x10^{-4}$	0.2	10^{-3}	0	0	N/A

Non-SLAP models

Model	Win ratio	lr	dropout	L2	explore	noise	Dirichlet
n0_1	60.0%	$2x10^{-3}$	0	10^{-3}	0	0.25	0.3
n0_2	60.0%	$4x10^{-3}$	0	10^{-4}	0.25	0.25	0.3
n0_3	56.6%	10^{-3}	0	10^{-3}	0.25	(0.1, 0.4)	0.3
n0_4	56.6%	$4x10^{-3}$	0	10^{-4}	0	0.25	N/A

Fig. 23 Best reinforcement learning models for SLAP & non-SLAP, stage 1

Note: noise (0.1, 0.4) means that noise weight varies from 0.1 to 0.4 during self-play, linearly against estimated value of the state.

4.3.2 Stage 2 Testing

At the next stage, the models in Fig. 23 would be trained by self-play of 5000 games. With more games arranged for training, larger data buffer size could be used. In Ch. 4.2.3, both SLAP and non-SLAP models converged when original training sample size was roughly 5000. So data buffer size was increased to 5000 and 4000 respectively for SLAP and non-SLAP models, roughly equivalent to storing latest 250 games. To align with stage 1 testing initially, the initial data buffer size was kept same as stage 1 for first 1000 games. This also had the benefit of getting rid of initial poor-quality game state data quickly.

Learning rate would decrease adaptively by half if validation loss increased beyond 3-sigma limit, measured every 100 games.

Let's see their winning rate curves in Fig. 24 and Fig. 25.

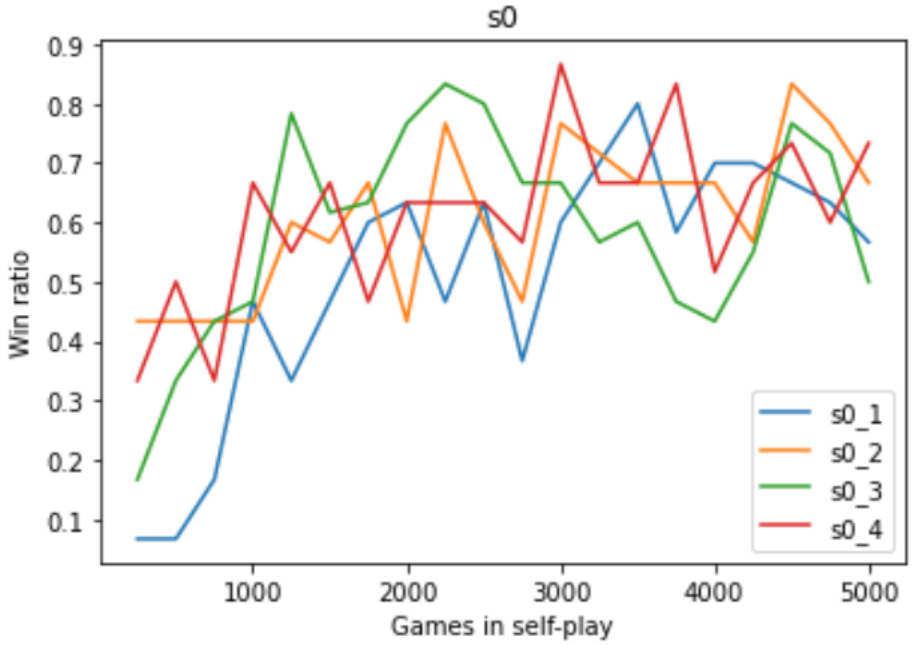


Fig. 24 Winning rate of s0 (models with SLAP, 0 Num_ResBlock), stage 2

In Fig. 24, the best among SLAP models was s0_4, by both highest and final winning rates. Its highest winning rate was 86.7%, equivalent to winning 26 games out of 30.

It was often either win or loss in evaluation, and seldom a tie. Assuming tie could be neglected, it simplified as Bernoulli distribution and the standard deviation could be approximated by $\sqrt{p(1-p)/30}$ to calculate confidence interval, where 30 is number of trials. So its 95% confidence interval would be 86.7% +/- 12.2%, i.e. (74.5%, 98.9%)

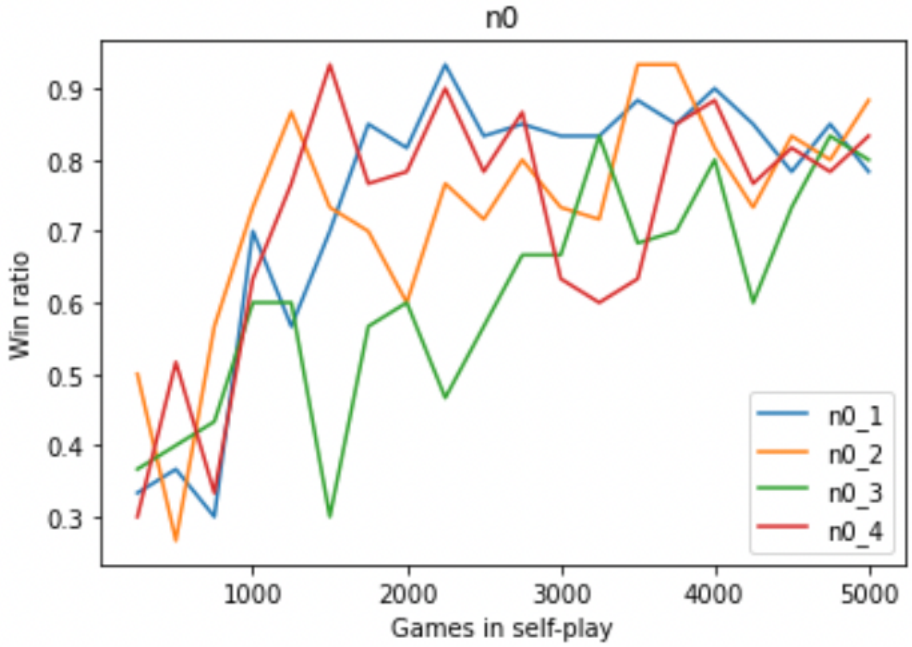


Fig. 25 Winning rate of n0 (models with no SLAP, 0 Num_ResBlock), stage 2

In Fig. 25, the best among non-SLAP models was n0_2, by both highest and final winning rates. Its highest winning rate was 93.3%, equivalent to winning 28 games out of 30. its 95% confidence interval would be 93.3% +/- 8.9%, i.e. (84.4%, 100%)

Both models s0_4 and n0_2 achieved similar winning rates, judged by confidence intervals. If winning rate of two thirds (66.6%) is used as benchmark for this three-tier evaluation, both took 1000 games to achieve or surpass this. However, n0_2 (non-SLAP) took 1250 games only to first achieve winning rate of 86.6%, while s0_4 (SLAP) took 3000 games.

In Fig. 26, SLAP models required 10.8% more time for each move in self-play. In separate version optimizing the speed (see methods in Ch. 3.5.3), the gap could reduce to around 5%.

	1	2	3	4	Average (sec/move)
s0	0.732	0.717	0.785	0.809	0.761
n0	0.681	0.664	0.696	0.706	0.687

Fig. 26 Computation time of self-play for best models, stage 2

Playing against each other

A script was created by allowing two AI agents to play against each other for 100 games, with the same setting as if each was playing against an evaluator in the evaluation. However, it was observed that exactly same moves were repeated each game, as the neural network in evaluation mode was deterministic, and to align with setting against the common evaluator and to allow playing greedily, there was no noise to AI's MCTS. So it was not used for evaluation.

Learning rate multiplier was used to adaptively decrease learning rate when validation loss increased too much. See learning rate multiplier in Fig. 27 and Fig. 28.

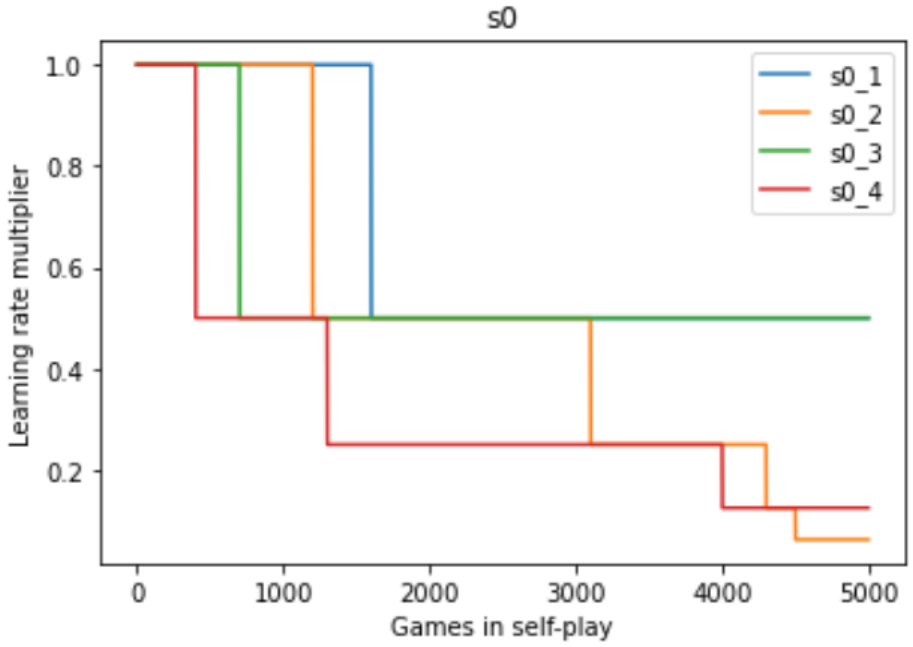


Fig. 27 Learning rate multiplier for SLAP models, stage 2

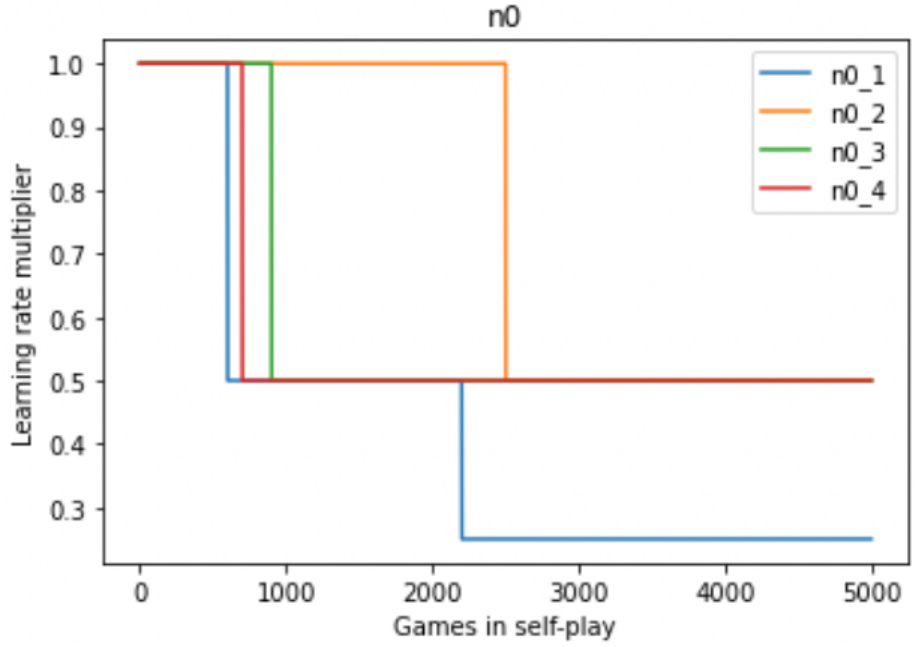


Fig. 28 Learning rate multiplier for non-SLAP models, stage 2

In Fig. 27 and Fig. 28, SLAP models tended to decrease learning rate more frequently than non-SLAP models, implying more frequent significant increase of validation loss.

See Fig. 29 and Fig. 30 for charts of training loss and validation loss, the latter of which were averaged over 25 losses (i.e. states of recent 25 games) to match roughly the batch size of training for comparison.

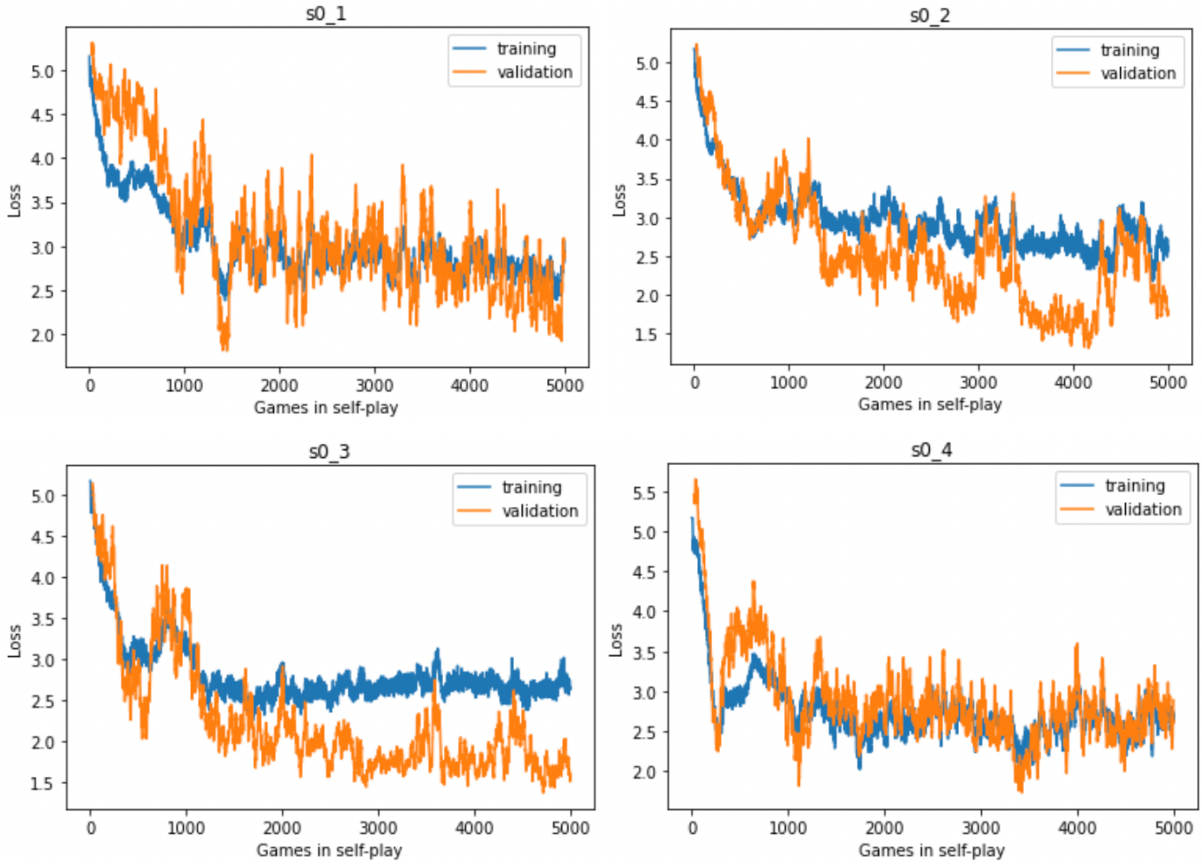


Fig. 29 Training and validation loss of SLAP models, stage 2

In Fig. 29, despite initial over-fitting, the validation loss curve of best SLAP model s0_4 eventually aligned with the training loss curve after roughly 1000 games, showing good generalization ability. Similar pattern was observed for s0_1. However, it was peculiar to observe that s0_2 and s0_3 had validation losses frequently below training losses after roughly 1000 games. This implied it was easier to predict from validation data than training data, which was unusual, but could happen in deep reinforcement learning. The AI agent kept on learning and would behave differently over the course of training, from which the game data would have different distribution over time. And why did it happen after 1000 games but not earlier? In stage 2 testing, data buffer size was set to increase substantially after 1000 games, so training data would start to become even more heterogeneous. There was a trade-off – increasing the data buffer size could have more data to regularize learning, but it could also invite more heterogeneous data. Also notice that during first 1000 games for s0_2 and s0_3, the validation loss initially just fluctuated around training loss and there was no obvious overfitting. After 1000 games, the validation curves of 4 models all shifted downwards relatively more than the training loss curves.

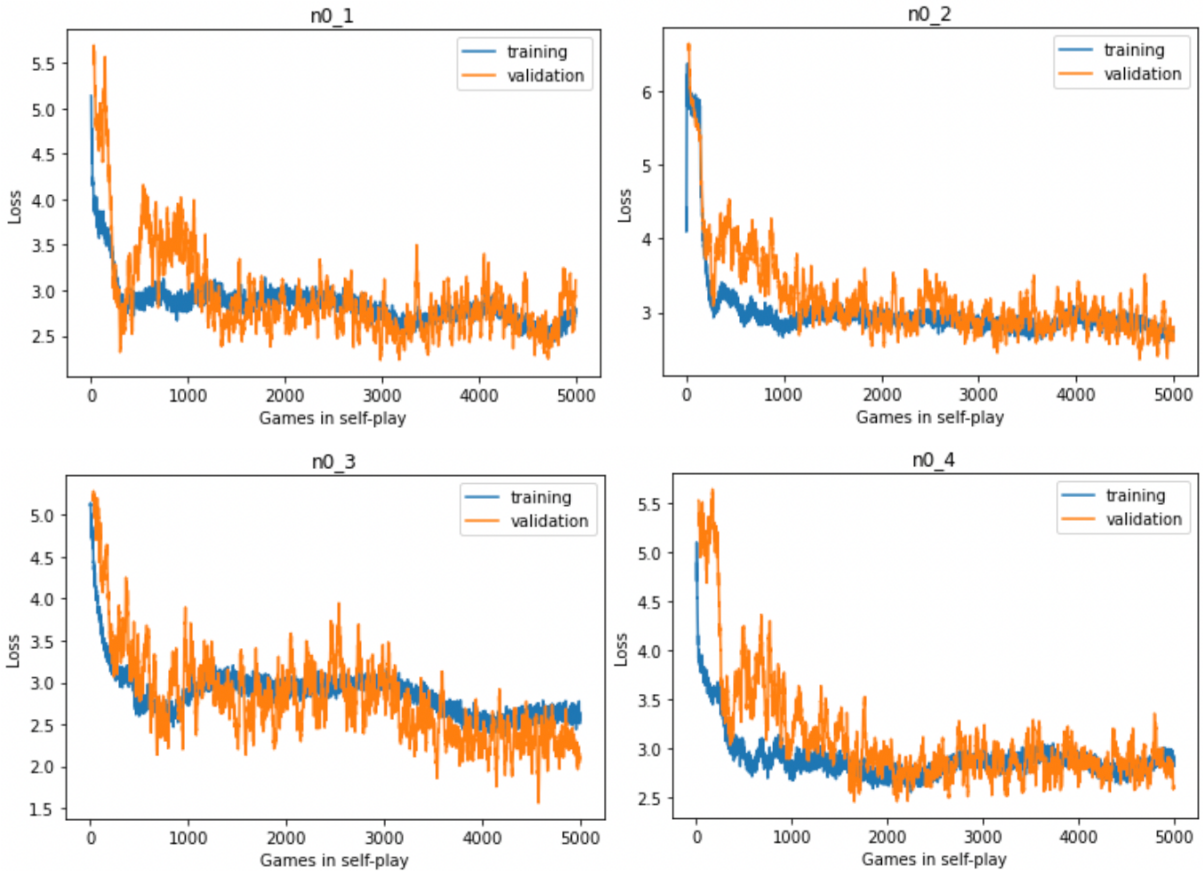


Fig. 30 Training and validation loss of non-SLAP models, stage 2

In Fig. 30, good generalization ability was shown as the validation loss curve of all non-SLAP models eventually aligned with training loss curve after roughly 1000 games, except n0_3, which usually had validation loss smaller than training data. This peculiar phenomenon was similar to that for s0_2 and s0_3 (but to a lesser extent), which were explained in previous paragraph.

4.3.3 Testing Buffer Size

To test the impact of data buffer size, best models of SLAP (s0_4) and non-SLAP (n0_2) were repeated but with data buffer size of only 1,250 and 10,000 respectively throughout whole reinforcement learning, denoted as s0_4_few and n0_2_few respectively.

Similar to stage 2, above models were trained by 5000 games. Their winning rates were shown in Fig. 31 and Fig. 32 respectively in comparison by data buffer size.

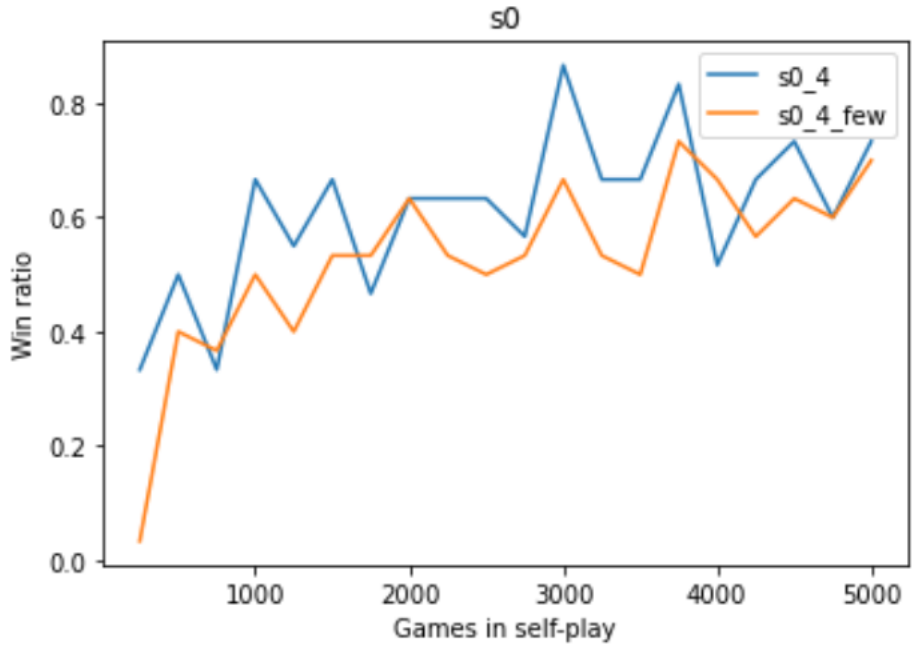


Fig. 31 SLAP model winning rate with different data buffer size

In Fig. 31, with fewer data in buffer, the highest winning rate achieved for SLAP model was only 73.3%, worsen than before and beyond the confidence interval. Also, the new winning ratio curve was almost always below the original curve.

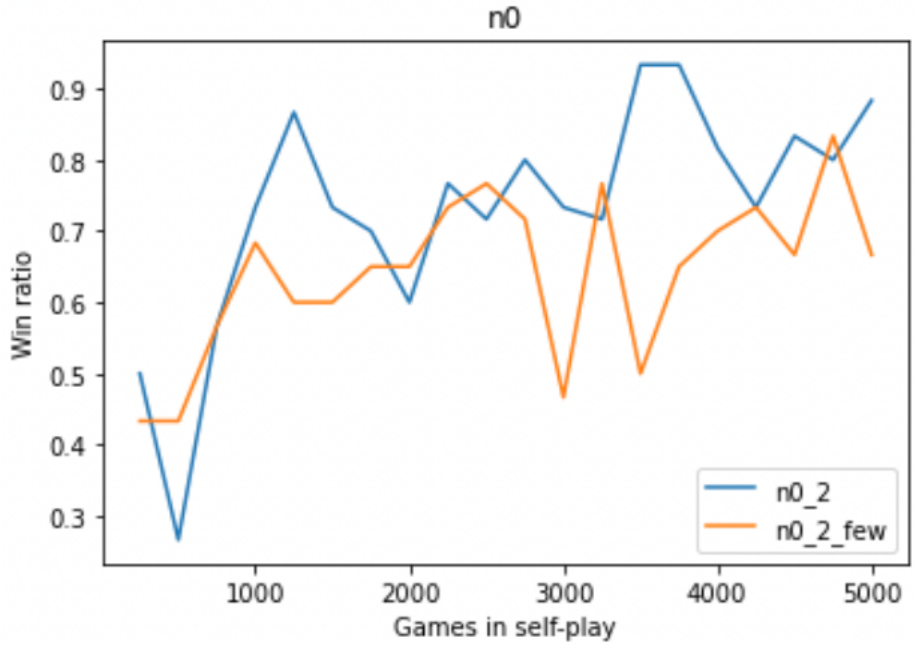


Fig. 32 Non-SLAP model winning rate with different data buffer size

In Fig. 32, with fewer data in buffer, the highest winning rate achieved for non-SLAP model was only 83.3%, worsen than before and beyond the confidence interval. Also, the new winning ratio curve was almost always below the original curve.

As it was designed to adaptively decrease learning rate by changing learning rate multiplier when validation loss increased too much (beyond 3 sigma, measured every 100 games), it would be interesting to see if there is a change to learning rate multiplier pattern, in Fig. 33 and Fig. 34:

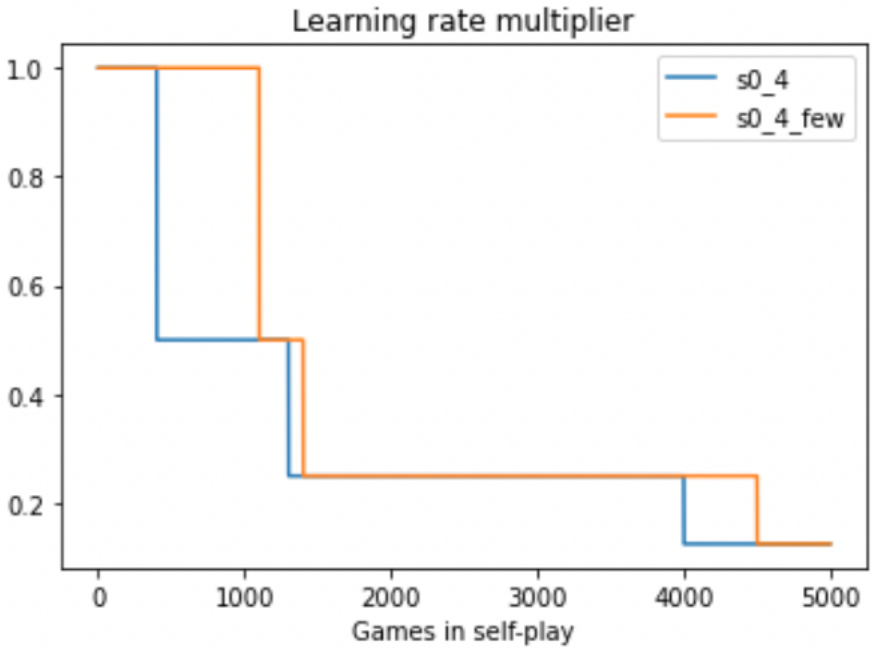


Fig. 33 SLAP model learning rate multiplier with different data buffer size

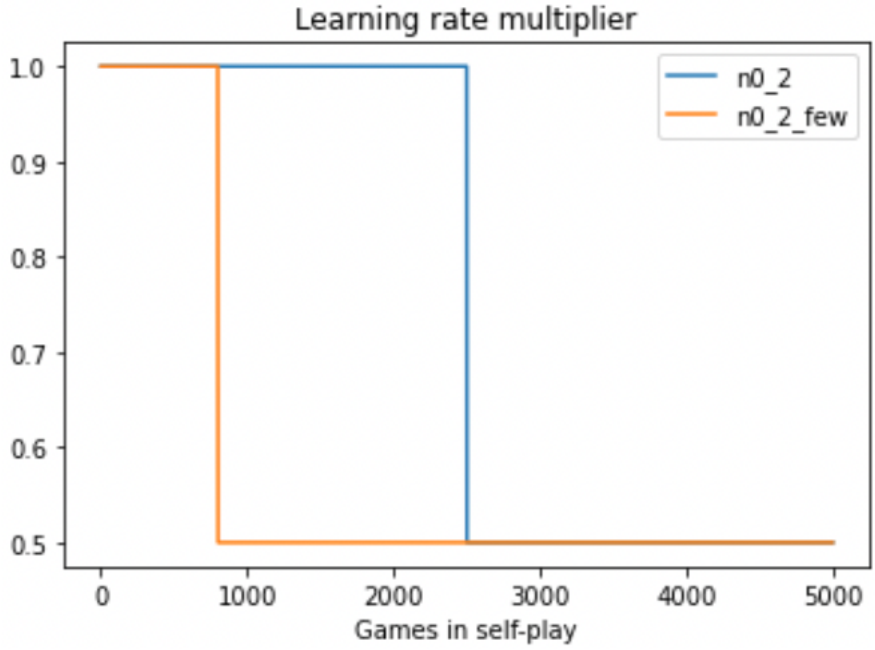


Fig. 34 Non-SLAP model learning rate multiplier with different data buffer size

In Fig. 33 & 34, there was no change to frequency of decreasing learning rate multiplier upon decreasing data buffer size for SLAP and non-SLAP models, though timing would differ.
In Fig. 35, there were more fluctuations to both training and validation losses for small data buffer.

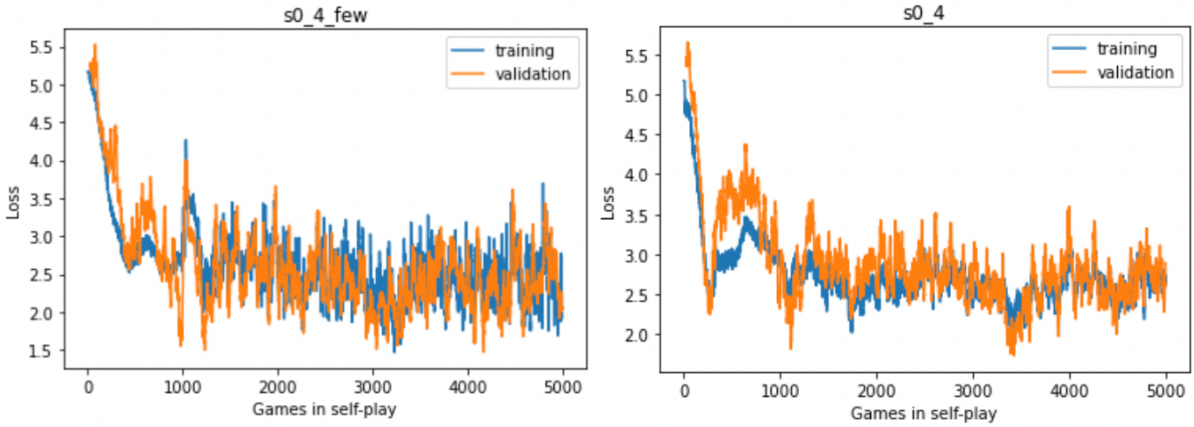


Fig. 35 Training and validation losses of SLAP models with different data buffer size

In Fig. 36, there were more fluctuations to both training and validation losses for small data buffer. Also, obvious over-fitting occurred with fewer data in buffer.

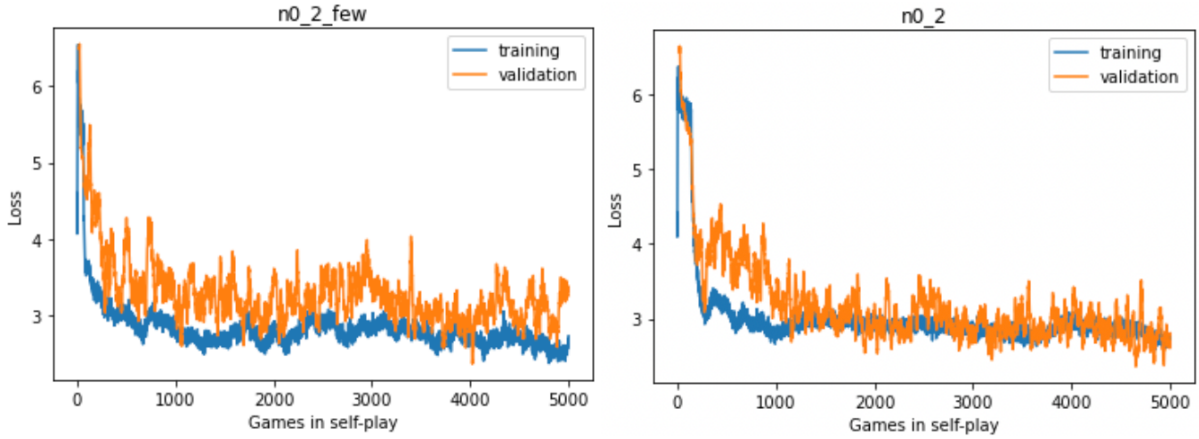


Fig. 36 Training and validation losses of non-SLAP models with different data buffer size

So, it harmed reinforcement learning when data buffer was too small and it was good decision to test with larger data buffer at stage 2.

4.3.4 Testing SLAP-CC

SLAP-CC was tested by same configurations as best baseline model n0_2 from Ch. 4.3.2, but adding information from SLAP-CC and scaled position indices as extra input feature planes. The new model, denoted as nc0, ran for 5000 games. See methods in Ch. 3.1.3 and Ch. 3.7.

In Fig. 37, the highest winning rate achieved for SLAP-CC model was 96.7%, slightly higher than that of baseline model but within its confidence interval. The winning rate curves were moving up and down around similar level relatively.

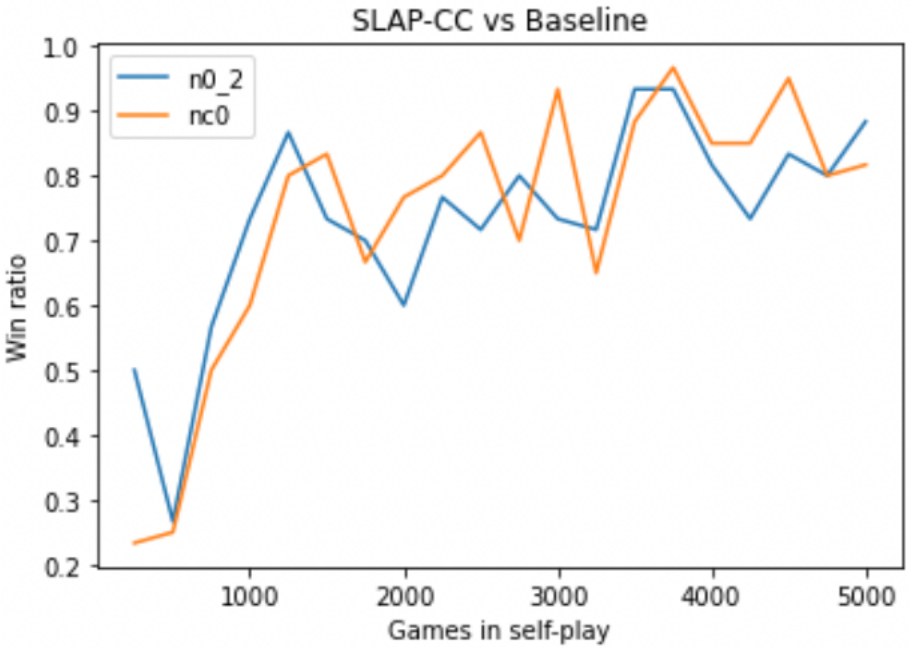


Fig. 37 Winning rate of SLAP-CC vs non-SLAP models

In Fig. 38, SLAP-CC did not decrease learning rate multiplier throughout training.

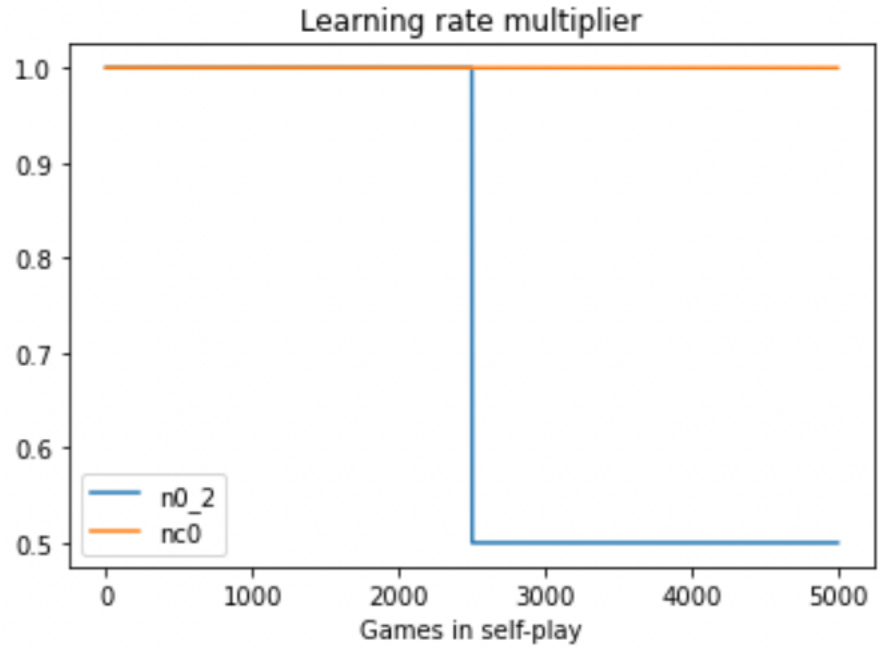


Fig. 38 Learning rate multiplier of SLAP-CC vs non-SLAP models

In Fig. 39, SLAP-CC model had similar training and validation loss curve.

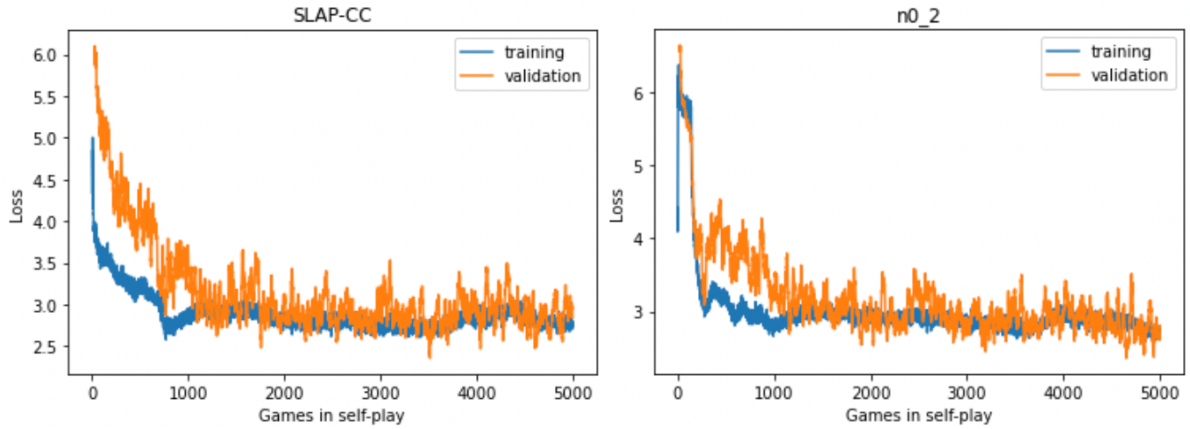


Fig. 39 Training and validation losses of SLAP-CC vs non-SLAP models

Overall, the results of SLAP-CC model were similar to its baseline counterpart model.

4.4 AI vs Human

See how AI agents perform against human (myself), displayed in simple graphics below.

4.4.1 Model s0_4 (SLAP) vs myself

Snapshots of the game against s0_4 are shown in Fig. 40a, Fig. 40b and Fig. 40c.

AI: O Human: X

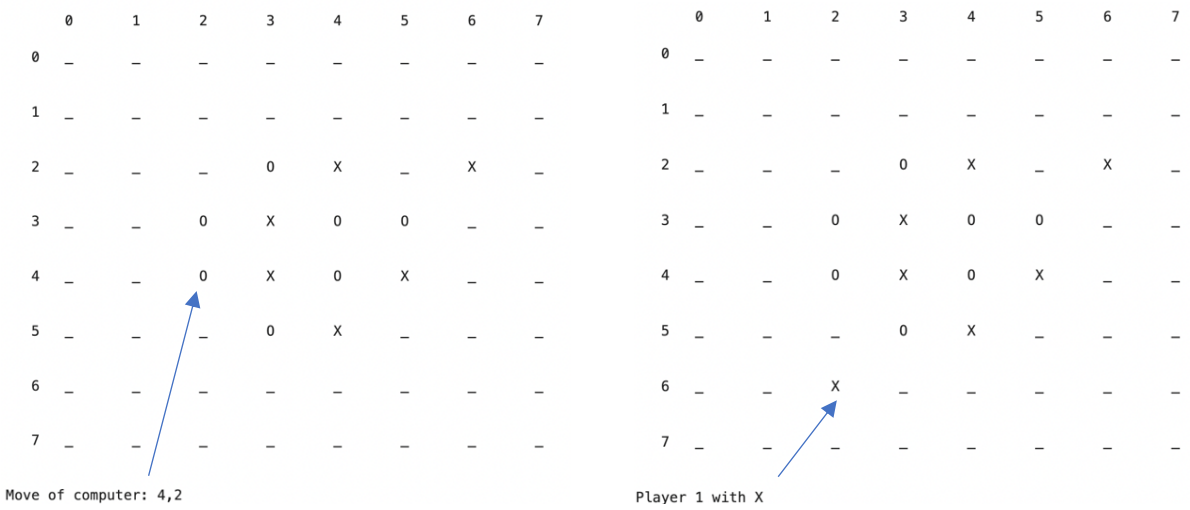


Fig. 40a Hidden combo attack by s0_4

In Fig. 40a, AI's move seemed to be weak at first sight. But I had to defend at (6,2) rather than attack at (6,3), otherwise AI could win by: O-(6,2) X-(7,1), O-(5,2), i.e. "straight four" with both ends open, then either (2,2) or (7,2) forcing a win.

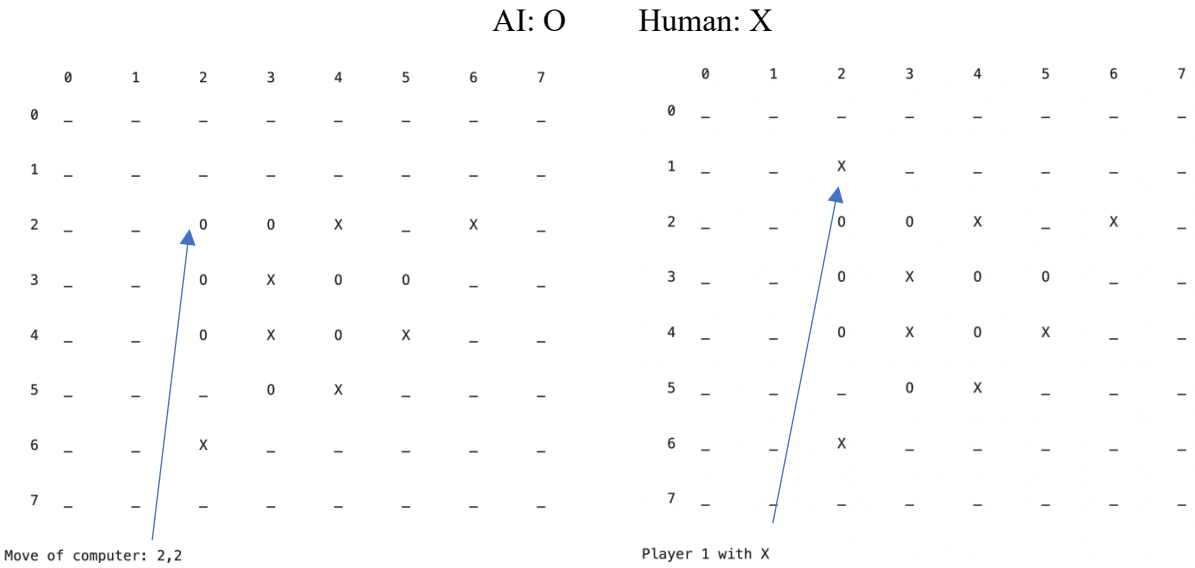
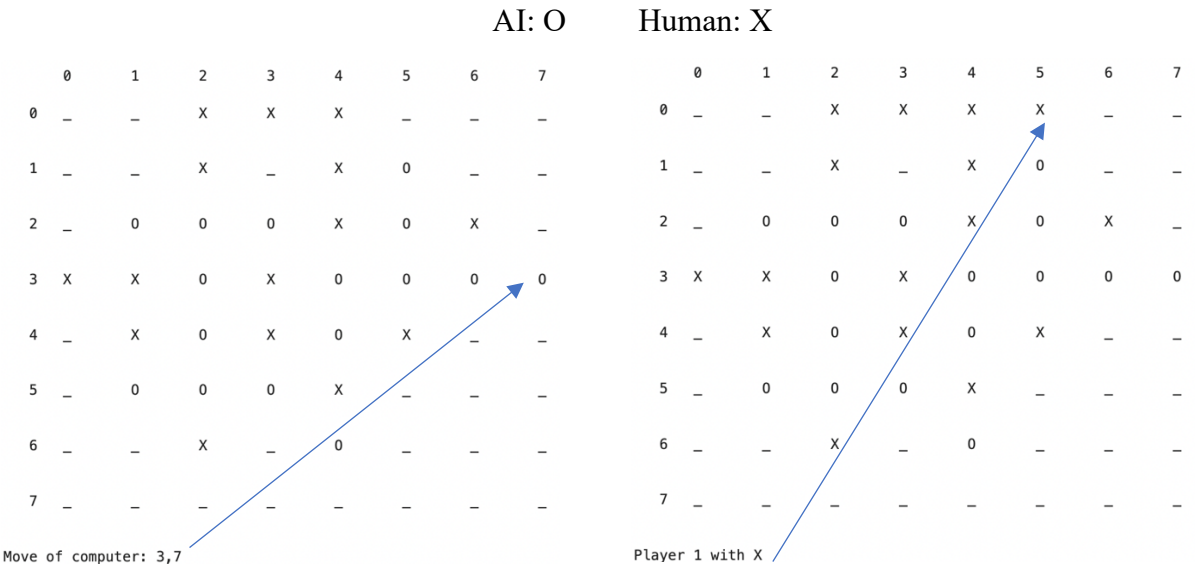


Fig. 40b Direct attack by s0_4

In Fig. 40b, AI at (2,2) threatened to get “straight four” at (1,2) with both ends open.



In Fig. 40c Failure to defend on the edge by s0_4

In Fig. 40c, AI did not prevent me from getting “straight four” at (0,5) with both ends open, leading to lose either at (0,1) or (0,6). So this game ended at time t = 34 finally.

I was amazed by AI s0_4’s hidden combo attack showing decent sophistication, while I was disappointed that it could not defend against a simple attack on the edge case. I suspect this is because AI lacked experience training on the edge case, which usually happens when the game is played long enough. From training data, its average game duration was 17.6 time-steps during self-play.

4.4.2 Model nc0 (SLAP-CC) vs myself

Snapshots of the game against nc0 are shown in Fig. 41a and Fig. 41b.

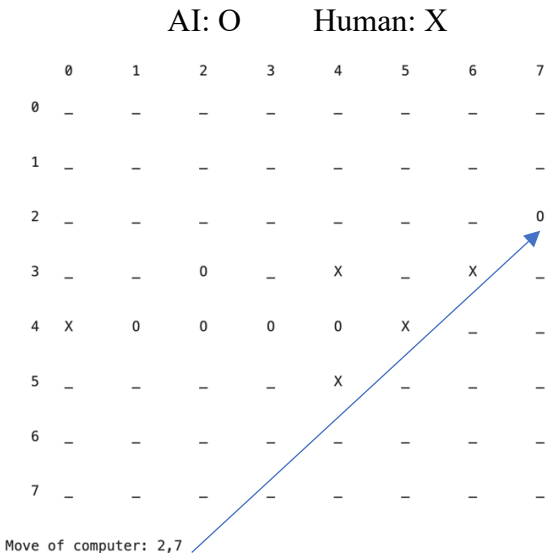


Fig. 41a nc0 prevented opponent's "straight four" attack (with both ends open)

The model nc0 had multiple similar defence occasions like Fig. 41a.

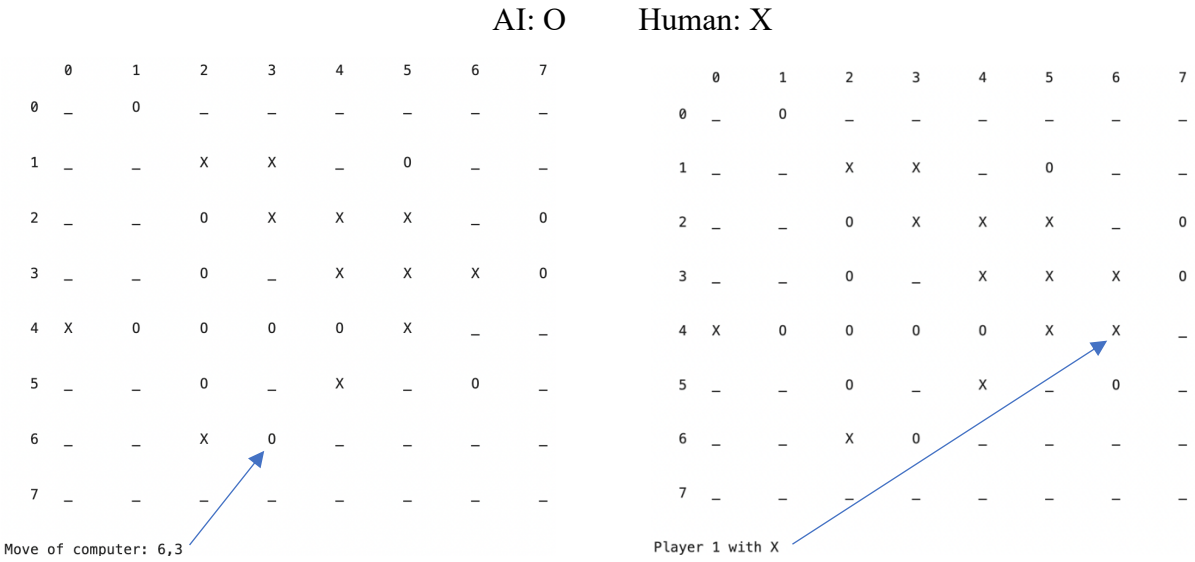


Fig. 41b nc0 failed to prevent 'straight four' attack

In Fig. 41b, nc0 chose (6,3) to stop opponent's weak extension at (6,3) and strengthened its diagonal connection, but forgot the most threatening move (4,6) from opponent.

In short, nc0 was able to detect and prevent attacks in multiple occasions, but it seemed to have problems to prioritize multiple threats in the same occasion.

4.4.3 Model n0_2 (baseline) vs myself

Snapshots of the game against s0_4 are shown in Fig. 40a, Fig. 40b and Fig. 40c.

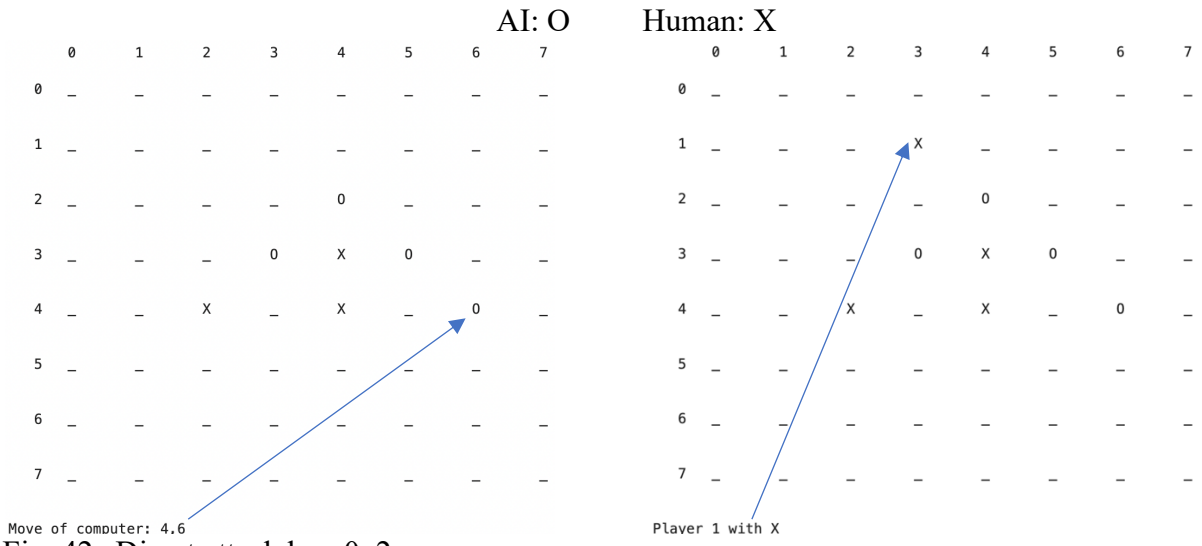


Fig. 42a Direct attack by n0_2

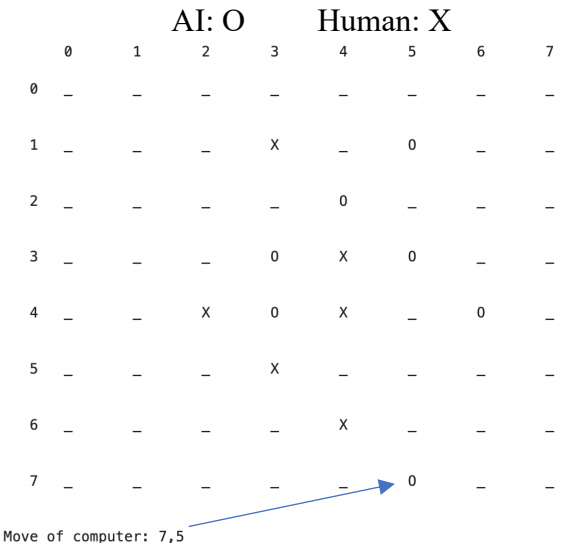


Fig. 42b n0_2 prevented opponent's "straight four" attack (with both ends open)

Fig. 42a and Fig. 42b are common situations as explained in the caption.

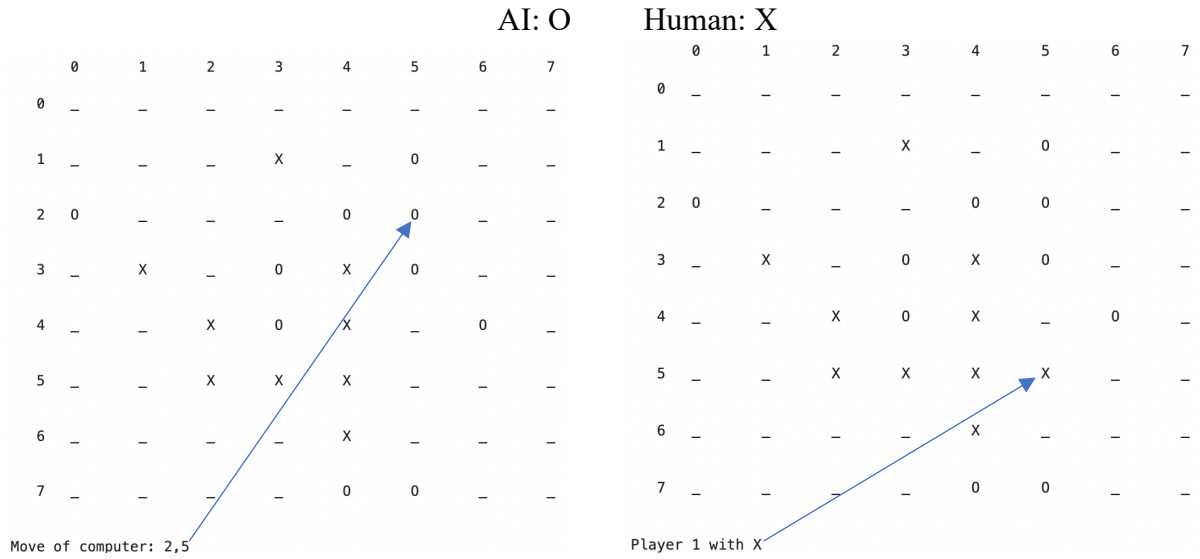


Fig. 42c n0_2's wrong prioritization of attack and defence

In Fig. 42c, n0_2 failed to detect it was more urgent to defend rather than attack. In fact, it even had a choice to both defend and attack – (5,5) instead of (2,2), with a hidden next move of (4,5) threatening to win at two positions (2,5) or (6,5).

In short, n0_2 could handle simple attacks usually but failed to prioritize attack and defence correctly.

Overall, each agent would start from the centre showing basic intelligence, and also played intelligently most of the time, but could fail to prioritize when to attack or defend, and there was a potential for failure on the edge case. Among these I was most amazed by s0_4 (SLAP)'s hidden combo attack, but I was also most disappointed by its failure to prevent simple attack on the edge case. Without this simple mistake, I would evaluate s0_4 as the strongest. Given that each agent failed to defend a simple attack (for different reasons) at the end and lost, (despite multiple successful defences), I would regard them as similar in terms of strength.

5. Discussion

5.1 Objective 1 and Relevant Results

Objective 1 of this research was to implement SLAP function for the case of Gomoku, without using any domain-specific knowledge except the required symmetric properties.

In Fig. 12 and Fig. 13 of Ch. 4.1, the results respectively showed that the built SLAP and SLAP-CC functions could output the same variant regardless of which symmetry (of specified type) variant was input, fulfilling the functionality of SLAP introduced in Ch. 1.1.1. SLAP function was to exploit reflection and rotation symmetry of Gomoku, while SLAP-CC was to handle translation variants as Gomoku could be considered 'partially' translation invariant. No Gomoku-specific knowledge was applied inside the SLAP and SLAP-CC function, except using specific type of symmetry as mentioned. Therefore objective 1 was accomplished.

5.2 Objective 2 and Relevant Results

Objective 2 of this research was to test whether SLAP would benefit neural network learning, measured by validation losses and/or training time for convergence.

Synthetic Gomoku states were created to test impact of SLAP on neural network learning, isolating from reinforcement learning dynamics. The baseline control, 'non-SLAP models', used 8 times the training sample size by data augmentation.

5.2.1 Generality to different hyperparameters

In Ch. 4.2.1 Preliminary Stage Testing, it tested 2400 combinations of hyperparameters and formed 96 sub-groups by choosing the best learning rate and dropout rate, i.e. 48 pairs of models with SLAP vs without SLAP (i.e. data augmentation baseline). Among these, models with SLAP outperformed its counterparts in 77% and underperformed in 23%, measured by validation loss with 0.001 precision. As this stage only took 1000 iterations by batches for network learning, the losses could not be taken as final convergence limit, rather they should be taken as an indication of likelihood to converge well and fast, used for selection of models for more intensive training at the next stage in Ch. 4.2.2. As the same grid search setting was applied to both SLAP and non-SLAP models, and I considered 2400 combinations of

hyperparameters as sufficiently extensive search, this result indicated SLAP's likelihood to learn well or fast in more situations, which brought benefits to neural network learning. Another interesting finding was that each pair of SLAP and non-SLAP counterparts could share the same best learning rate and dropout rate if we relaxed the validation loss comparison precision to within +/- 0.01.

5.2.2 Convergence speed

In Ch. 4.2.2 Stage 2 Testing, it showed that both SLAP and non-SLAP models could converge to similar value by neural network learning without over-fitting. As SLAP only involved one-off data pre-processing and there was no other cost to neural network training, the convergence speed could be compared simply by number of iterations required for convergence. Fig. 19 showed that SLAP improved the convergence speed by 83.2%. As it was based on average by repeating experiments 4 times each creating new dataset of synthetic states again for training and testing, it should be convincing that SLAP could speed up convergence of neural network learning. As explained, no domain specific knowledge was applied inside SLAP function, so I argue this would apply to other domains that could exploit symmetry, especially reflection and rotation symmetry.

5.2.3 Limitation by sample size

Ch. 4.2.3 showed the limitation of SLAP – it was more vulnerable than its baseline counterparts to decrease of original training sample size. The reason would be obvious – baseline had 8 times the training sample size by data augmentation, while they had exactly same network architecture and hyperparameters. The minimum training sample size required for SLAP to converge was around 5000, but this figure would be problem and architecture dependant.

5.2.4 Groupoid and 'partial' invariance

Ch. 4.2.4 showed that models with SLAP-CC could converge to similar values as their baseline counterparts. However, the convergence speed worsened by 30.7%. This showed that more work would be required to deal with groupoids or to exploit 'partial' invariance.

5.2.5 Summary

In short, it was proved that SLAP could improve convergence speed of neural network learning despite using only one eighth the training sample size (given that not too small). This should apply to domains that are invariant to symmetry, especially reflection and rotation symmetry.

The testing purpose of objective 2 was accomplished, with quite satisfactory results.

5.3 Objective 3 and Relevant Results

Objective 3 was to test whether SLAP would benefit reinforcement learning in playing Gomoku, measured by winning ratios and/or training time for attaining certain winning ratios.

The baseline algorithm followed the algorithm of AlphaGo Zero and Alpha Zero (see methods in Ch. 3.4), with data augmentation applied to training samples by reflection and rotation, using 8 times the data buffer size of SLAP counterparts to store latest games in self-play for neural network training. The same neural network structure was used for comparison, but SLAP models might tune for different hyperparameters due to different dynamics with reinforcement learning.

5.3.1 Winning Ratio

Ch. 4.3.2 showed that SLAP and non-SLAP could achieve winning rate of 86.6% and 93.3% respectively by reinforcement learning. The difference was insignificant as determined by 95% confidence interval, given that there were 30 trials (games) in each evaluation. So both SLAP and non-SLAP models achieved similar winning rate. On the other hand, SLAP-CC achieved winning rate of 96.7%, slightly higher than its baseline counterpart. But it was also regarded as similar winning rate as it was within the confidence interval of baseline. Also, the winning rate curve looked similar as that of the baseline model. The evaluation by human in Ch. 4.4 also showed that the strength of these models were similar.

5.3.2 Computation and Training Speed

On computation time, SLAP affected self-play most as SLAP was applied to every move in self-play and only applied to every other move in evaluation (as evaluator agent didn't use SLAP), and not applied during training iterations of neural network except for pre-processing

training samples. And roughly 90% of time was spent on self-play, so let's focus on self-play. Fig. 26 showed that SLAP models required 10.8% more time than non-SLAP for every move in self-play. It was mainly because MCTS used 400 simulations for every move and this would require SLAP to process data every time for network inference. Note that in separate version optimizing the speed (see Ch. 3.5.3), the gap could reduce to around 5%. Also, the network architecture was not as deep as popular ones such as ResNet-50 or ResNet-101, which could be 10 times deeper, implying that the proportion of overhead cost of SLAP to network inference would drop by 10 times and become further insignificant. Therefore, it would be justified to use the number of games required as a simple measure to compare training time. If winning rate of two thirds (66.6%) is used as benchmark for the three-tier evaluation, both SLAP and non-SLAP took 1000 games to achieve or surpass this. However, non-SLAP took 1250 games only to first achieve winning rate of 86.6%, while SLAP took 3000 games. There was no evidence that SLAP could speed up reinforcement learning. On the other hand, SLAP-CC had similar winning rate curve (against number of games) as its baseline counterpart model, as shown in Fig. 37.

5.3.3 Others

A bug about winner status was detected in a diagonal corner case for reinforcement learning. Testing after stage 1 were carried out again but stage 1 results were kept as it was expected that the corner case bug would not affect the early learners much. Results were similar, producing the same best models for SLAP and non-SLAP, confirming that the corner case bug did not affect the pattern much. This also helped confirmed the reliability of results.

5.3.4 Summary

In short, it was proved that SLAP could similar winning rate by reinforcement learning despite using only one eighth the training sample size. SLAP-CC also could achieve similar winning rate, but with same training sample size as baseline model. Based on experiment results, neither SLAP nor SLAP-CC appeared to speed up reinforcement learning.

The testing purpose of objective 3 was accomplished, though the results were not as expected.

5.4 Answer to Research Question

The major question of this research: can transformation variants be exploited directly by SLAP to further improve and combine with CNN for machine learning?

Recall that the motivation of SLAP was to concentrate experience to speed up learning by standardizing symmetry variants. It was proved that SLAP could improve convergence speed of neural network (CNN used in this research) learning despite using only one eighth the training sample size. As no Gomoku-specific knowledge was used in building SLAP function, this should apply to other domains that are invariant to symmetry, especially reflection and rotation symmetry as these were tested. Note that SLAP is model-independent, so I also argue that above these results should apply to models beyond CNN. But for groupoid variants, more work would be needed to exploit for machine learning.

Unfortunately, SLAP could only achieve similar performance as baseline during reinforcement learning in the experiments. Given that SLAP could speed up neural network learning, why couldn't it benefit deep reinforcement learning to increase learning speed? This might suggest deficiency in the experiments and more work to do, to be discussed in the next chapter.

6. Evaluation, Reflections and Conclusions

6.1 Evaluation of the Project

The choice of objectives was a bit stretching but still within my capabilities. It was proved to be a good decision to add objective 2 as opposed to initial plan, as this could decouple neural network learning from the dynamics of reinforcement learning for more focused investigation, leading to more meaningful results being drawn. It was also a good planning decision to de-prioritize groupoid investigation in the initial plan, as it turned out to be difficult even for just performance at par. The literature review was adequate given the scope of this research.

The methods employed successfully drew convincing results that the novel function SLAP could improve convergence speed of neural network learning. But further experiments could not show similar improvement in reinforcement learning, which likely suggested deficiency of the experiments, to be discussed in Ch. 6.2.

Overall, it was rewarding by undertaking this project. For neural network learning, my novel method SLAP was shown to outperform data augmentation, a standard industry method to exploit symmetry invariance.

6.2 Reflections

6.2.1 Reinforcement Learning Experiment

One major area for reflections is the potential deficiency of the deep reinforcement learning experiments. How could something improve neural network learning but could not improve deep reinforcement learning that involved neural network learning?

One likely (not necessarily practical or affordable) improvement could be increase in number of games for training in stage 1 testing for more reliable choice of hyperparameters as success at too early stage might not be representative for experience in longer training, but this was really constrained by time and computation resources.

One potential reason could be wrong choice of hyperparameters in both initial stage and second stage. The more frequent decrease of learning rate multiplier adaptive to validation loss increase might suggest sub-optimality of hyperparameters. Note that AlphaGo Zero only dropped learning rate twice over a million training steps (Silver D. et al 2017a). Also, 2400 combinations of hyperparameters were initially tested in the neural network learning experiment, while only 240 combinations of hyperparameters were tested at the first stage of reinforcement learning experiment. But it wasn't done so without reason. The neural network learning experiment was based on synthetic Gomoku states, so the resulting best hyperparameters might be not far from those in Gomoku reinforcement learning, hence narrowing the search space. This was the starting point of choosing the set of hyperparameters to be tested in reinforcement learning. Computation resource would be another practical reason, as reinforcement learning was much more computation intensive than the neural network learning for each trial – the difference could be minutes vs hours, if not days. But it should be noted that reinforcement learning had additional hyperparameters on top of neural network learning, so naturally it already required more extensive search of hyperparameters. And perhaps it could be that synthetic states for supervised learning were too easy to learn compared with the reinforcement learning dynamics, where AI agents were learning from zero. Furthermore, from the experience I learnt that a network that learnt fast might not be always good in reinforcement learning, as it could make learning from self-play less stable, which might imply different hyperparameters would be needed as compared to supervised learning, such as decreasing the learning rate. One practical improvement on hyperparameter search, if I were to repeat the experiments, would be to follow AlphaGo Zero using Bayesian optimization for hyperparameter search (Silver D. et al 2017a), instead of grid search in this project.

Another potential try would be further increase in data buffer size to have more data to regularize or suppress the fast learning of neural network, because SLAP speeded up learning in neural network and this might make reinforcement learning less stable. But this involved a trade-off and could invite more heterogeneous data and backfire. This might also explain the importance for AlphaGo Zero and Alpha Zero to have multiple GPUs running in parallel games, which allowed fast-learning networks to have more training data to regularize learning without sacrificing homogeneity of data (or waiting for more games for homogeneity before update).

6.2.2 Others

Groupoid – I should try different structures as input for groupoid variants, and I should probably not add it as additional input planes projecting to the same feature map, as this might interfere with original input planes and become noise (because they were shifted toward centre). Rather, it should be projected to another feature map(s) or channelled to separate convolutional layers and then transformed back to corresponding original positions.

It might be interesting to combine with symmetric weights (Hu X. S., Zagoruyko S. and Komodakis N. 2018) to see if it further improved results, as they both worked by forcing symmetry.

Had I more time, I should explore whether my underdog noise was a novel method, and do more experiments to see if it could outperform constant noise weight. I should also add option to allow noise in evaluation mode to facilitate AI agents playing against each other to avoid exact repetition of game moves.

6.3 Conclusions

A novel function SLAP was successfully built to standardize symmetry variants. It was shown that SLAP could improve the convergence speed of neural network (CNN in my experiment) learning synthetic states in Gomoku by 83.2%. Since no domain specific features or knowledges were used in SLAP, it should also benefit neural network learning generally for domains that are symmetry invariant, especially for reflection and rotation symmetry. Note that SLAP is model-independent, so I argue that the benefits should apply to models beyond CNN. However, it was not yet proved to be beneficial to reinforcement learning. Neither was it proved to exploit groupoid variants effectively.

6.4 Future Work

Perhaps SLAP may be applied for domains that are not fully symmetry invariant, by breaking down the neural network layers into two parts – first learning as if it were fully symmetry

invariant. Although SLAP is not directly differentiable, one workaround would be similar to that in transforming Gomoku action probabilities. That is, given the transformation information as another input, transform the learned output back to corresponding original position, and then carry out necessary subsequent computations forward. This is inspired by CNN structure – first learning as if it were fully translational invariant, but fully connected to linear layers finally to apply different weights to different positions to break the translational symmetry. Also inspired by this simple phenomenon – a kid usually thinks ‘b’ and ‘d’ are the same thing at the beginning of learning alphabets, indicating learning as if it were symmetry invariant first.

Another related future work might be to work out differentiable implementation by approximation, as mentioned in Ch. 3.1.2.

References

1. Mnih V. et al 2013, 'Playing Atari with Deep Reinforcement Learning', *NIPS Deep Learning Workshop 2013*, 1 Jan 2013.
2. Silver D. et al 2017a, 'Mastering the game of go without human knowledge', *Nature,* 550, pp. 354– 359.
3. Silver D. et al 2017b, 'Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm', *Science*, Vol 362, Issue 6419, pp. 1140-1144.
4. Hinton G. E., Krizhevsky A. and Wang S. D. 2011, 'Transforming Auto-Encoders', *International Conference on Artificial Neural Networks (ICANN)*, 2011.
5. Sabour S., Frosst N. and Hinton G. E. 2017, 'Dynamic Routing Between Capsules', *Neural Information Processing Systems (NIPS),* 2017.
6. Vistoli A. 2011, 'Groupoids: A Local Theory of Symmetry', *Isonomia (Epistemologica)* 2011, 26, pp. 1–12.
7. Peer D., Stabinger S. and Rodriguez-Sanchez A. 2021, 'Limitations of Capsule Networks', *ScienceDirect (Pattern Recognition Letter)*, Vol 144, pp. 68-74.
8. Nair S. 2017, *A Simple Alpha(Go) Zero Tutorial*, Stanford University, Accessed 18 May 2022, http://web.stanford.edu/~surag/posts/alphazero.html
9. Dan S. et al 2021, 'Lightweight multi-dimensional memristive CapsNet', *International Joint Conference on Neural Networks (IJCNN),* 07/2021.
10. Sun K. et al 2021, 'Dense capsule networks with fewer parameters', *Soft computing (Berlin, Germany)* (1432-7643), Vol 25 (Issue 10), pp. 6927.
11. Hu X. S., Zagoruyko S. and Komodakis N. 2018, 'Exploring Weight Symmetry in Deep Neural Networks', *arXiv preprint arXiv:1812.11027*, Dec 2018.
12. Higgins I. et al 2018, 'Towards a Definition of Disentangled Representations', *arXiv preprint arXiv:1812.02230*, Dec 2018.
13. Caselles-Dupré H., Garcia-Ortiz M. and Filliat D. 2019, 'Symmetry-Based Disentangled Representation Learning requires Interaction with Environments', *33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada, 2019.*
14. Higgins I., Racanière S. and Rezende D. 2022, 'Symmetry-Based Representations for Artificial and Biological Intelligence', *arXiv preprint arXiv:2203.09250*, 2022.
15. Anselmi F. et al 2017, 'Symmetry Regularization', *The Centre for Brain, Mind and Machines (CBMM) Memo No. 63*, 26 May 2017.

16. Botvinick M. et al 2018, ‘Reinforcement Learning, Fast and Slow’, *Trends in Cognitive Sciences*, Vol 23, Issue 5, pp. 408-422, 2019.
17. Bergman D. 2019, ‘Symmetry Constrained Machine Learning’, *arXiv preprint arXiv:1811.07051v2*, 2019.
18. Song J. 2017, ‘An implementation of the AlphaZero algorithm for Gomoku (also called Gobang or Five in a Row)’, Accessed Jun-Sep 2022, https://github.com/junxiaosong/AlphaZero_Gomoku
19. Seetharaman P. et al 2020, ‘AutoClip: Adaptive Gradient Clipping for Source Separation Networks’, *2020 IEEE 30th International Workshop on Machine Learning for Signal Processing (MLSP)*, 2020.
20. Jang E., Gu S. and Poole B. 2017, ‘Categorial Reparameterization with Gumbel-softmax’, *Proceedings International Conference on Learning Representations (ICLR)*, 2017.
21. Blondel M. et al 2020, ‘Fast Differentiable Sorting and Ranking’, *International Conference on Machine Learning*, 2020.
22. Koker T. and Betz R. 2021, ‘Fast, differentiable sorting and ranking in Pytorch’, Accessed Sep 2022, https://github.com/teddykoker/torchsort
23. Alonso E. 2022, *Lecture 2: The Problem*, lecture notes, Deep Reinforcement Learning INM707, City, University of London, delivered 9 Feb 2022.

Appendix

Hardware specification

It was initially planned to use single GPU of Hyperion of the university (to be shared with other students) for computing, with a GPU cloud provider Jarvislabs as backup plan. But with disruption due to maintenance and any other reasons, Jarvislabs became the major source of GPU for computing.

Hyperion
gpu04: 4 x A100 80GB cards
RAM 384GB
2 x Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz (48 cores in total)

Jarvislabs
GPU – Nvidia Quadro RTX 5000
Nvidia Ampere A100 – CPU: AMD Processor 7203
RAM – 32GB
CPU - Intel(R) Xeon(R) Silver 4216
vCPUs - 7