Exploring a Learning Architecture for General Game Playing

Alvaro Gunawan[1], Ji Ruan[1], Michael Thielscher[2], and Ajit Narayanan[1]

[1] Auckland University of Technology, Auckland, New Zealand
alvarogunawan@aut.ac.nz
jiruan@aut.ac.nz **
ajitnarayanan@aut.ac.nz
[2] University of New South Wales, Sydney, Australia
mit@unsw.edu.au

Abstract. General Game Playing (GGP) is a platform for developing general Artificial Intelligence algorithms to play a large variety of games that are unknown to players in advance. This paper describes and analyses GGPZero, a learning architecture for GGP, inspired by the success of AlphaGo and AlphaZero. GGPZero takes as input a previously unknown game description and constructs a deep neural network to be trained using self-play together with Monte-Carlo Tree Search. The general architecture of GGPZero is similar to that of Goldwaser and Thielscher (2020) [4] with the main differences in the choice of the GGP reasoner and the neural network construction; furthermore, we explore additional experimental evaluation strategies. Our main contributions are: confirming the feasibility of deep reinforcement for GGP, analysing the impact of the type and depth of the underlying neural network, and investigating simulation vs. time limitations on training.

Keywords: General Game Playing · Machine Learning · Reinforcement Learning · Neural Networks

1 Introduction

The recent accomplishments of DeepMind's AlphaGo [9] have reignited interest in game-playing Artificial Intelligence, showing the effectiveness of Monte-Carlo Tree Search (MCTS) with deep neural networks and learning through self-play. Further work on AlphaGo Zero [11] showed that this method was effective even without any human expert knowledge, and in fact able to defeat their previous AlphaGo agent despite learning from effectively zero knowledge apart from the games rules. This method of learning to play a new game was extended beyond Go to AlphaZero [10] for learning to play Chess and Shogi with effectively the same architecture.

In the same vein as AlphaZero, General Game Playing (GGP) proposes a challenge: developing agents that are able to play any game, given a general set of rules [6]. GGP uses a logical language known as Game Description Language (GDL) to represent rules of arbitrary games. These game descriptions are a combination of a knowledge base containing static rules and a state containing dynamic facts. This raises the natural question whether the same methods

** Corresponding Author

that AlphaZero uses to learn to play Go, Chess and Shogi can be applied to general games described in GDL. This was recently addressed with Generalised AlphaZero [4], a system that applies deep reinforcement learning to GGP. While the overarching approach outlined by AlphaZero provides a general architecture for a self-play learning agent, there are a few key limitations that had to be overcome for this purpose:

- AlphaZero assumes the games are two-player, turn-taking and zero-sum.
- Neural network architectures are hand-crafted for each game, encoding features such as board geometry.
- As a result of the specialised neural networks, the agents have some form of implicit domain knowledge for each game.

In the domain of GGP, games are neither required to be two-player, turn-taking nor zero-sum so that a GGP agent must be able to account for games with any number of players, simultaneous action games and non-zero-sum games. With regard to the second point, the agent must also be able to generate a neural network for any given game description, without requiring any additional modification from a human. Finally, the system should be general for any game with no specialised domain knowledge at all. In this paper, we complement the recent work on Generalised AlphaZero [4] as a system to overcome these limitations in an AlphaZero-style self-play reinforcement learning agent designed to learn games in the GGP setting. Our main contributions lie in the further exploration of this learning architecture:

- We confirm the feasibility of deep reinforcement for GGP under different settings, including the use of a different GGP reasoner.
- We analyse the impact of type and depth of the underlying neural network.
- We investigate simulation vs. time limitations on training.

The rest of this paper is organized as follows. Section 2 summarises related works in applying learning to GGP including Generalised AlphaZero. Section 3 describes the architecture of our self-play reinforcement learning agent, GGPZero. In the extensive section 4 we present the results of our experiments, showing the agent's capabilities in a variety of games as well as an in-depth examination in the agent's architecture, followed by a concluding discussion in section 5.

2 Related Work

Research in applying reinforcement learning based approaches to GGP began with the framework RL-GGP [1] that integrates a GGP server and a standard framework for reinforcement learning RL-Glue. No implementation of a full agent is provided, making it difficult to assess performance. QM-learning [14] combines Q-learning with MCTS, and integrates it within GGP. The performance of the agent is evaluated on three small board games. However, this method is still effectively only an on-line search method, using Q-learning to optimise MCTS. The recent GGP agent [3] implements an AlphaZero-style learning component that exhibits strong performance in several games, but requiring hand-crafted convolutional neural networks for these games. Generalised AlphaZero [4] implements a Deep Reinforcement Learning framework for GGP that shows strong performance in several games but weak performance in a cooperative game.

The above-mentioned restrictions of AlphaZero were overcome by extending the policy-value network with move probabilities and expected rewards for all players in an n-person game. Furthermore, a general propositional reasoner to process arbitrary GDL game rules was used as input to the neural network, thus overcoming the reliance on a board and a handcrafted neural network. In this paper we use a similar architecture to that of [4] but with the main differences in the choice of the GGP reasoner, the neural network construction and the experimental evaluation. In particular, we allow a manual setting of the neural network depth to allow for further investigations into the agent's behaviour, which was only done in a limited manner in [4].

3 GGP Learning Agent Architecture

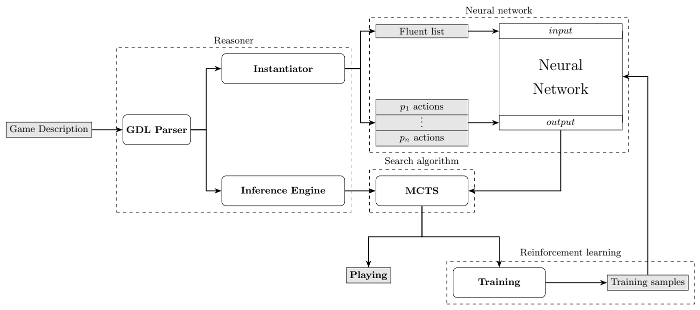


Fig. 1: Architecture of GGPZero agent.

The overall architecture of our system GGPZero consists of four main components, each made up of smaller subcomponents. This first component is the reasoner, which takes the GDL description and processes it to generate the neural network and conduct inference during play. The second component is the neural network, which is used as a policy and utility estimator for the search algorithm. The third component is the search algorithm, using both neural network estimation and MCTS for policy generation. Finally, the self-play reinforcement learning component is used to train the neural network with training samples generated by the search algorithm.

While GGPZero's architecture is similar to that of AlphaZero, there are two key differences. Firstly, as the GGP platform requires the agent to infer the rules of a game from a GDL description without human aid, the agent requires a reasoner to do this inference. Secondly, we use the reasoner's instantiated fluent and action lists to automatically generate the neural network, as we cannot rely on hand-crafted neural network architectures for the more general setting of GGP. Additionally, the search algorithm and reinforcement training have been designed with the GGP setting in mind, as the games may have an arbitrary number of players and are not strictly zero-sum. The rest of this section will provide more detail on each individual component of the GGPZero architecture.

Reasoner The GDL parser takes as input a game described in GDL and converts it into appropriate representations for the instantiator and the inference engine.

The instantiator generates the sets of possible *fluents* (dynamic predicates that are true for a given state) and possible actions, using the method described in [13,12]. The inference engine is used by MCTS to infer facts about the game such as legal moves, subsequent states, terminal conditions and utility values. There are two main inference methods: Prolog-based and Propositional network (Propnet) based [8]. A Prolog-based method converts the GDL rules to an equivalent Prolog representation and uses an off-the-shelf Prolog reasoner for inference. In our paper, we use the Prolog-based inference engine as described in [13,12] for its greater generality, while a Propnet-based inference method is used in Generalised AlphaZero[4]. While this is more efficient than a Prolog-based method, Propnets are not applicable to some larger games as the grounding process can be intractable. Additionally, to improve the efficiency of the agent, we save the results of queries. This allows repeated visits to states during the MCTS to be more efficient, especially as the neural network will tend to prefer certain optimal states and actions after sufficient training.

Neural Network The neural network takes a fluent vector as input. The vector has length equal to the total number of all fluents possible in the game, and each index corresponds to a single fluent. For an input vector $V = (a_1, a_2, \ldots, a_n)$ and a state consisting of fluents in the set $S \subseteq \{a_1, ..., a_n\}$, the value of element $a_i = 1$ if $a_i \in S$, otherwise $a_i = 0$. The neural network outputs for each player a normalised policy vector for player's actions and the estimated goal value.

The neural network architecture is automatically generated based on the game description and a network depth parameter. The input and hidden layers are generated as fully connected layers with ReLU activation with width equal to the number of fluents, and depth equal to the network depth parameter. For the output layers, a number of output heads are generated equal to the number of players. Each output head is then further split into an estimated policy and an estimated goal value. The number of nodes in the policy output layer is equal to the number of actions possible for the given player and the estimated goal value is a single output node.

Unlike the large, convolutional neural network used in AlphaZero, the neural network used in GGPZero uses fully connected layers and is comparatively small. One advantage of this approach is that games described in GDL are not guaranteed to be easily represented with a board-like structure, such as a matrix in the case of the CNN in AlphaZero. Additionally, we use smaller networks for prioritizing of rapid experimentation and to identify the limits of this approach that may then lead to the need for more complex deep learning approaches.

Search algorithm The search algorithm outputs the finalised policy vector to be used in play and self-play reinforcement training. It uses a *simultaneous action* MCTS [5] that is able to accommodate multi-player, general sum games with simultaneous actions. Rather than using random playouts during the expansion phase, the search algorithm uses the neural network's estimator to evaluate the value of a newly expanded state. Additionally, the policy generated by the neural network is considered during the selection phase. As a result, a modified form of upper confidence bound on trees $U(s,a)$ is used as the selection strategy:

$$U(s,a) = Q(s,a) + C_{uct} \cdot P(s,a) \cdot \frac{\sqrt{N(s)}}{1 + N(s,a)}$$

where $Q(s,a)$ is the estimated goal value, $P(s,a)$ is the estimated policy, $N(s)$ is a node count of state s, $N(s,a)$ is the action count of a at s, and C_{uct} is a weight constant for the exploration factor which can be set as a parameter.

Reinforcement Learning The agent is tasked with learning an optimal policy for a game by training the neural network to improve its approximation of the function $f^{\star}(s) = (\mathbf{V}, \boldsymbol{\Psi})$, where $\mathbf{V}$ is the estimated utility and $\boldsymbol{\Psi}$ is the estimated policy. The neural network with weights $\boldsymbol{\theta}$ is trained on the policy estimated through $MCTS(s; \boldsymbol{\theta})$. In essence, the agent uses the neural network guided MCTS to continuously improve the neural network only through iterations of self-play. A single iteration of learning has 3 distinct phases:

- **Self-play**: generate training examples using MCTS with the current best network weights $\boldsymbol{\theta}_{best}$.
- **Training**: train the current best neural network with new examples generated during self-play.
- **Evaluation**: compare the performance of the newly trained network $\boldsymbol{\theta}_{new}$ against the current best network $\boldsymbol{\theta}_{best}$. If $\boldsymbol{\theta}_{new}$ outperforms the current $\boldsymbol{\theta}_{best}$, then $\boldsymbol{\theta}_{best}$ is updated by $\boldsymbol{\theta}_{new}$.

To generate training examples for the neural network, the agent plays episodes of self-play games against itself. The policy generated by the MCTS is used as the training target for the policy vector, while the actual goal value at the terminal state of a game play is used as the training target for the estimated goal value. The neural network is trained by minimising the sum of the following, across all players for a training example: (1) **mean squared error** between the neural network estimated goal values and the self-play terminal goal values; (b) **cross entropy** of the neural network estimated policy and the MCTS generated policy.

4 Experimental Studies and Results

We first present an evaluation of the GGPZero agent on five games: Tic-Tac-Toe, Connect-Four, Breakthrough, Pacman3p and Babel, followed by further experiments to evaluate various key aspects of the architecture. The selection of games is similar to [4] with a mixture of complexity levels and game types. We have three benchmark agents for these evaluations: a random agent (RANDOM) that samples uniformly from legal actions, a neural-network based agent (NN), which uses the network of GGPZero without MCTS search, and a MCTS based agent (MCTS). Where applicable, these benchmark agents are given the same parameters as GGPZero.[3]

[3] All components are implemented in Python, with Prolog components using the SWI-Prolog interpreter with PySWIP as a bridge. We base our implementation of the MCTS on the code provided in [7].

All experiments are conducted on a machine with an Intel Xeon E5-1630v4 3.7 10M 2400 4C CPU, 128GB RAM and NVIDIA GeForce GTX1080 8GB.

4.1 Game Evaluation

Performance on two-player zero-sum games The following table shows the performance of GGPZero (with 20 layers and 1000 training iterations) against three benchmark agents in three games with increasing complexity. The results are shown in the format of (Win/Lose/Draw):

	Opponent Agent (Win/Lose/Draw)		
Game	RANDOM	NN	MCTS
Tic-Tac-Toe	91 / 0 / 9	93 / 0 / 7	11 / 27 / 62
Connect-Four	97 / 3 / 0	71 / 29 / 0	56 / 43 / 1
Breakthrough	100 / 0 / 0	73 / 27 / 0	86 / 14 / 0

Table 1: Results of GGPZero on two-player zero-sum games. GGPZero and MCTS are given the same simulation limit of 50.

GGPZero performs well in these games, and these games follow a similar format to the games that AlphaZero were trained and played on. However, unlike AlphaZero, the training of these agents is completely end-to-end, not requiring hand-crafted features for any particular game. Interestingly, GGPZero performs better on the more complex games, e.g., winning only 11% against the MCTS agent in Tic-Tac-Toe, while winning 86% in Breakthrough. This is likely due to the fact that the less complex games have state spaces small enough for the other agents to still be competitive in.

Performance on non-two-player general-sum games Games such as Pacman3p and Babel highlight the challenge presented by GGP, as they are asymmetric, multi-player and general sum games.We show that GGPZero is able to learn to play these games, albeit with limited performance compared to the two-player, zero-sum case. The following table shows the goal value of the agents playing 20 games. E.g., the first row shows that role pacman is played by MCTS obtaining score 1.96, and two ghosts are played by RANDOM, obtaining score 8, while the third row shows that GGPZero gets score 2.34 in a similar setting. We can see that GGPZero (with 5 layers and 300 training iterations) outperforms MCTS.

Agents	*pacman*	*two ghosts*	*pacman* score	*two ghosts* score
MCTS vs RANDOM	MCTS	RANDOM	1.96	8
	RANDOM	MCTS	1.12	10
GGPZero vs RANDOM	GGPZero	RANDOM	2.34	7
	RANDOM	GGPZero	0.66	10
GGPZero vs MCTS	GGPZero	MCTS	1.53	10
	MCTS	GGPZero	0.79	10

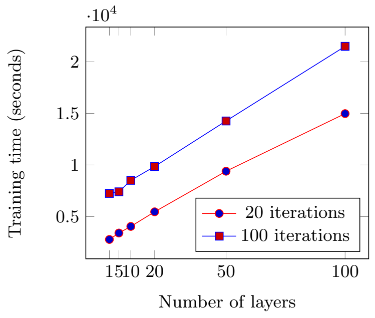
Table 2: Total goal value of various agents playing 20 games of Pacman3p

The next experiment is on the game of Babel, a three-player, simultaneous, cooperative tower-building game. 100 games are played for each agent acting three roles in the game. RANDOM gets an average goal value of 0.122, MCTS gets 0.777, GGPZero (with 20 layers and 100 training iterations) gets 0.443, and GGPZero (with 20 layers and 500 training iterations) gets 0.638. We can see GGPZero performs better than RANDOM but worse than MCTS. Similar results are obtained in [4]. This shows a limitation of the current architecture in dealing with this type of games.

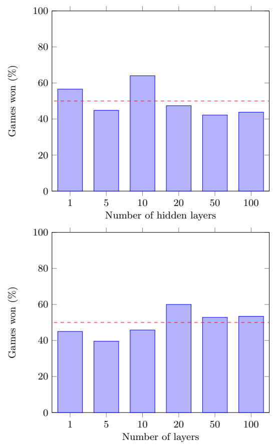
4.2 Varying Neural Network Depth

Systematic experiments were used to investigate the effects of the Neural Network depth on agent performance by varying the number of hidden layers and the training iterations. To do so we set up agents with 1, 5, 10, 20, 50 and 100 layers and each agent plays 100 games against every other agent. We also select 20 iterations and 100 iterations as the amount of trainings are given to these agents. These experiments were carried out with the game Connect-Four.

Fig. 2a shows that increasing the number of hidden layers causes the total training time to increase linearly, and that training 100 iterations takes proportionally more time under different layers. Fig. 2b shows the winning rate by each agent over 500 games (100 games against each other agent). With 20 iterations of training, the agent with 10 layers has the best winning rate, while with 100 iterations of training, the agent with 20 layers has the best winning rate. Overall, this shows that smaller networks generally perform better with less training, while larger networks perform better with more training. This is likely due to the fact that larger networks suffer from underfitting with insufficient training while smaller networks suffer from overfitting. The former can be seen with the 100 layer agent after 20 iterations: the policy generated by the neural network is typically a flat distribution across all actions.



(a)



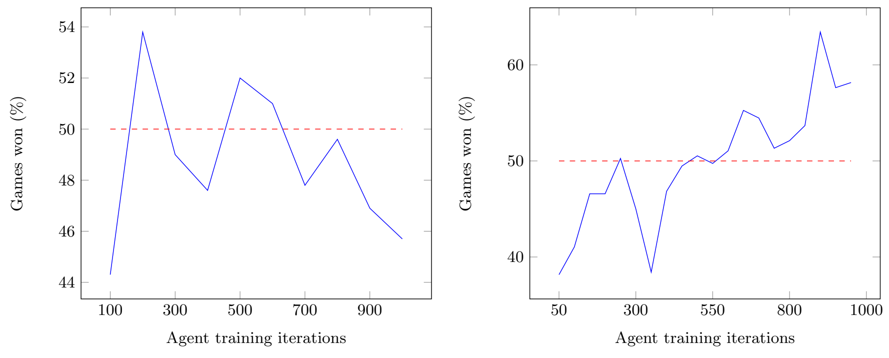
(b) Top: 20 iterations. Bottom: 100 iterations.

Fig. 2: (a): Training time for GGPZero with varying number of hidden layers. (b): Winning rate of GGPZero with varying number of hidden layers.

4.3 Convolutional Neural Network Experiments

While the agent with a fully-connected neural network showed good performance when playing against the benchmarks, we would like to investigate the stability of the training method with regard to winning outcomes. We tested the agents with different training iterations, under the assumption that an agent with more iterations will have more success than an agent with fewer iterations. Fig. 3 shows that the agent with a convolutional neural network increases its win rate to above 60% with increasing iterations, whereas a fully connected network reaches optimal win rate of just under 54% relatively early in learning with more fluctuations later on. This is likely due to the fully connected network overfitting its parameters.

For zero-sum games such as Connect-Four, we would ideally prefer an agent that increases in performance with additional training. As Connect-Four is solved, we know that there is an optimal strategy for the game. While GGPZero will most likely be unable to learn this optimal strategy with the limited training time that it is provided, we would like to see the agent continuously improve, eventually converging to the optimal strategy towards the limit.



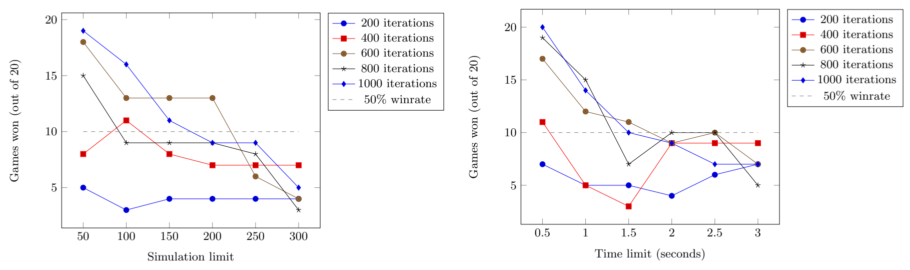
(a) Fully connected network **(b)** Convolutional network

Fig. 3: Winning rates by agents with different training iterations in Game Connect-Four.

4.4 Effects on Simulation-limited and Time-limited MCTS

In this section, we compare the effects of simulation and time limitation on the GGPZero and MCTS during the game play. For the following experiments, we use the game Breakthrough as it has a large state space and branching factor. For GGPZero, we look at 5 versions with training iterations from 200 to 1000.

Simulation Limited In the prior experiments and test games, we use a *simulation limited* variant of MCTS. When selecting an action to play, the agents (both



(a) Simulation-limited **(b)** Time-limited

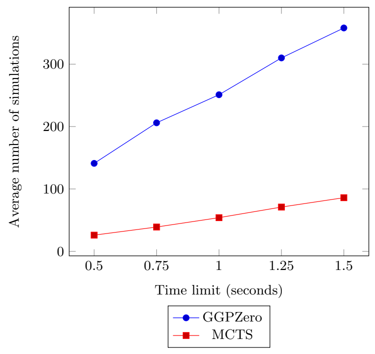
Fig. 4: GGPZero winning rate against MCTS with increasing time and simulation limits with varying amounts of training iterations on Breakthrough games.

GGPZero and MCTS) perform a limited number of simulations, expanding the search tree. After completing these simulations, the action policy is then generated. We use this variant of MCTS to reduce the impact of the inference engine's efficiency on the performance of the agent, allowing both GGPZero and MCTS agent to search a similar amount of the game states. Increasing the simulation limit allows the agents to search more extensively, potentially leading to better performance at the cost of slower action selection. This is especially evident with the MCTS agent, as the playout phase requires extensive use of the inference engine. GGPZero is able to circumvent this computationally taxing operation by using the neural network.

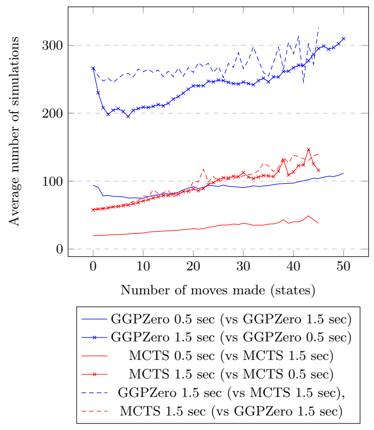
Fig. 4a shows the GGPZero's winning rate against MCTS when simulation limit varies from 50 to 300. For GGPZero with 1000 training iterations, it clearly outperforms the MCTS when the simulation limit is low, but loses out when the limit increases. A similar trend can be seen for GGPZero with 600 and 800 iterations. This result is not entirely unexpected as the strategy generated by the MCTS agent begins to approach the Nash equilibrium when the number of simulations performed increases. While for GGPZero with lower training iterations, the winning rate stays relatively flat and is generally worse than the MCTS. This suggests that when GGPZero's training is insufficient, the MCTS component inside GGPZero is not as effective as the pure MCTS agent due to the limitation of the neural network.

Time Limited In the usual GGP format, actions must be selected under a limited time budget ranging between 10 to 90 seconds, depending on the complexity of the game. As a result, most GGP agents favour using a time-limited variant of MCTS, performing as many simulations as possible in a limited time frame before generating their action policies. In the next set of experiments, we set time limits from 0.5 seconds to 3 seconds to examine the effects of time limit. Fig. 4b presents GGPZero's winning rate against MCTS under different time limit. For the GGPZero with 1000 training iterations, it significantly outperforms the MCTS agent when the time limit is short, but performs worse when the time limit increases. Similar trend is shown in GGPZero with 600 or 800 iterations, but less so in GGPZero with 200 or 400 iterations. These results are consistent with the simulation-limited case in Fig. 4a.

We further examined the simulation numbers under different time limits and during different stage of the game play. Fig. 5a shows that as the time limit increases, the average number of simulations of the MCTS agent does not increase as rapidly as that of the GGPZero. Despite that, the MCTS agent still performs better when given a longer time limit. Note that GGPZero with different training iterations performs similar numbers of simulations within the same time limit. This is due to the fact that the neural network can be seen as an $O(1)$ operation regardless of the amount of training; there is no trade-off with regard to efficiency when using a more trained agent.



(a) Average simulations per time limit



(b) Average simulations over the course of a game

Fig. 5: Average number of simulations completed by agents

In the time limited scenario, another aspect to consider is the varying amount of time required to perform search at different stage of the game. Although in Fig. 5a we see that the MCTS agent has significantly less *average simulations* throughout the course of a game, we see in Fig. 5b that as the game progresses, the average number of simulations tends to increase for all agents. This is likely due to the search more likely visiting terminal states as it progresses through the game, causing simulations to finish earlier. In the case of the MCTS agent, this early stopping by finding terminal states presents a significant reduction in simulation time, as it does not require the search to enter the playout phase.

To conclude, in both the simulation-limited and time-limited settings, we see that GGPZero with enough training significantly outperforms the standard MCTS agent when given a strict simulation or time limit. However, this difference in performance diminishes and eventually is reversed once these limits are large enough. From these observations, we can view the trained neural network as a form of *compression*, replacing the computationally expensive playout phase with a neural network. The quality of this compression is in part affected by the amount of training provided to the agent but also by the architecture of the neural network itself. As we saw in subsection 4.3, the convolutional neural network had better and more stable performance than the fully connected neural network when given sufficient training. The final comment is that the neural network with limited training can also be a limiting factor for the MCTS component in the GGPZero, as when giving a longer time limit, the pure MCTS agent can always outperform the GGPZero as in our experiments.

5 Conclusion and Future Work

We described GGPZero, an AlphaZero-style self-play reinforcement learning agent and implementation, designed to learn games in the GGP setting. While the architecture was inspired by the recent Generalised AlphaZero [4], the main differences are in the choice of GGP reasoner and the neural network construction; furthermore we carried out systematic experimental evaluation of various aspects of the use of deep reinforcement learning for GGP. Our main contribution is in the exploration of this learning architecture: confirming its feasibility, the impact of neural network depth, types of network, and simulation vs. time limitation on training. The experiments with network architectures have shown that the agent scales well in terms of both efficiency and effectiveness. More powerful hardware will allow us to take advantage of deeper neural networks. These more complex network architectures have been shown to be more effective for games such as Connect-Four, given sufficient training.

To further improve the overall performance of the agent, the current implementation could be improved by parallelising several components. Firstly, training could be parallelised to allow for multiple games to be played concurrently, allowing for either faster training times or more extensive training within the same timescale. Secondly, the MCTS itself could be parallelised [2] — however, the selection of which parallelisation method to use must be investigated within the context of the agent's architecture.

Another possible improvement to the agent is to use larger and deeper network sizes for all games. As the experiments show, deeper networks with more training tended to have better performance than shallower networks. The current trends in machine learning seem to indicate that larger networks are a reasonable approach for improving overall performance, given sufficient training is available.

The Babel results show that the agent is capable of learning and playing games that are not zero-sum, turn-taking or two-player. This means that GGPZero is able to generalise beyond the games that AlphaZero was able to play. However, we also see that a pure MCTS agent still outperforms GGPZero in this game. Further investigation is required into the nature of the training method itself, as it may have limitations when learning to play more general games.

While the current method for generating neural network architectures is general, it still requires a separate neural network for each game. This method is still effective in playing games, but can be argued to not be truly general, as each network is specialised for each game. A more general neural network architecture that can be used for any game would be ideal, but two key challenges still stand: (1) what kind of general game state representation can be used for the input to the neural network and (2) what is the appropriate learned and generalized representation for transfer to another architecture?

References

1. Benacloch-Ayuso, J.L.: RL-GGP. http://users.dsic.upv.es/~flip/RLGGP (2012)
2. Chaslot, G.M.B., Winands, M.H., van den Herik, H.J.: Parallel monte-carlo tree search. In: International Conference on Computers and Games. pp. 60–71. Springer (2008)
3. Emslie, R.: Galvanise zero. https://github.com/richemslie/galvanise_zero (2017)
4. Goldwaser, A., Thielscher, M.: Deep reinforcement learning for general game playing. In: AAAI. pp. 1701–1708 (2020)
5. Lanctot, M., Lisỳ, V., Winands, M.H.: Monte carlo tree search in simultaneous move games with applications to Goofspiel. In: Workshop on Computer Games. pp. 28–43. Springer (2013)
6. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General game playing: Game description language specification (2008)
7. Nair, S.: A simple alpha(go) zero tutorial. https://web.stanford.edu/~surag/posts/alphazero.html (2017)
8. Schkufza, E., Love, N., Genesereth, M.: Propositional automata and cell automata: Representational frameworks for discrete dynamic systems. In: Australasian Joint Conference on Artificial Intelligence. pp. 56–66. Springer (2008)
9. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al.: Mastering the game of Go with deep neural networks and tree search. Nature **529**(7587), 484 (2016)
10. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al.: Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv preprint arXiv:1712.01815 (2017)
11. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al.: Mastering the game of Go without human knowledge. Nature **550**(7676), 354 (2017)
12. Vittaut, J.N., Méhat, J.: Efficient grounding of game descriptions with tabling. In: Workshop on Computer Games. pp. 105–118. Springer (2014)
13. Vittaut, J.N., Méhat, J.: Fast instantiation of GGP game descriptions using prolog with tabling. In: ECAI. vol. 14, pp. 1121–1122 (2014)
14. Wang, H., Emmerich, M., Plaat, A.: Monte carlo q-learning for general game playing. CoRR **abs/1802.05944** (2018), http://arxiv.org/abs/1802.05944