The Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI-20)

Deep Reinforcement Learning for General Game Playing

Adrian Goldwaser, Michael Thielscher
Department of Computer Science, University of New South Wales
adrian.goldwaser@gmail.com, mit@unsw.edu.au

Abstract

General Game Playing agents are required to play games they have never seen before simply by looking at a formal description of the rules of the game at runtime. Previous successful agents have been based on search with generic heuristics, with almost no work done into using machine learning. Recent advances in deep reinforcement learning have shown it to be successful in some two-player zero-sum board games such as Chess and Go. This work applies deep reinforcement learning to General Game Playing, extending the AlphaZero algorithm and finds that it can provide competitive results.

Introduction

Since the start of AI research, games have been used as a testbed for progress as they can accurately capture many of the ideas of thought, planning and reasoning in an easily evaluable way (Shannon 1950). Recent advances have resulted in AIs being able to beat humans in more and more games (Hsu 2002; Silver et al. 2016). While each of these showed incredible improvement in different areas, they all required a tremendous amount of work and each is very specific so can only do the exact game it was programmed to do.

To address this, the idea of General Game Playing (GGP) looks at AI agents which are programmed without any explicit knowledge of a particular game; instead they must be general enough to play any game given a formal specification of the game at runtime (Genesereth and Björnsson 2013). This means that they cannot use specific knowledge or heuristics that do not apply to other games (Genesereth and Thielscher 2014).

The field of reinforcement learning (RL) looks at agents which can learn to maximise some notion of reward by taking actions and noting their effect. Recently, AlphaZero (Silver et al. 2018) has shown promising limited generality in the context of perfect information games, with the same algorithm reaching state-of-the-art in Chess, Shogi and Go. However, it was still limited to zero-sum, two-player, player-symmetric games and had a handcrafted neural network architecture for each. This agent learnt entirely from scratch using self-play deep RL with almost no knowledge of the game, giving it the potential to be extended and applied to the more general setting of GGP.

There has previously been very little work applying RL to the setting of GGP. A framework, RL-GGP (Benaclocher-Ayuso 2012), allowed testing of different RL algorithms, but little was published evaluating the performance of each for GGP. There has also been work into using TD learning to improve playout policies (Finnsson and Björnsson 2010). Finally there has been some initial work looking at applying Q-learning to GGP which found that it does converge but quite slowly (Wang, Emmerich, and Plaat 2018), and even when extending the Q-learning with Monte Carlo Search did not outperform UCT. However this showed that there is potential for RL based GGP agents. Our work pushes this direction forwards further, filling this gap by extending then applying deep RL to GGP and outperforming UCT.

The remainder of this paper is organised as follows. The next section covers previous approaches to GGP as well as an overview of the limitations of AlphaZero. In the following section, the design of the Generalised AlphaZero extension for GGP is explained. Finally, experimental results of this agent are shown which clearly show that deep reinforcement learning within a GGP environment can perform noticeably better than a UCT benchmark agent in a number of games.

Background

General Game Playing

The issue with using individual games as a testbed for AI is that they encourage programs which perform very well only in an extremely narrow domain. GGP addresses this by looking at programs which are only given the rules of the game at runtime. This means that they must be written without any explicit knowledge of any particular game, encouraging strategies which are applicable in different domains and general algorithms that result in agents which can plan and learn rather than simply using game-specific heuristics that humans have worked out. The lack of handcrafted heuristics means that performance should reflect the skill of the algorithm at that game, not the skill of the programmer (Genesereth and Thielscher 2014).

In GGP, players are given a formal specification of a game in a language such as Game Description Language (GDL) (Love et al. 2008), which is a logic program explaining the dynamics and reward system in the game, players then need to play the game without any extra input from humans. The GDL description of a game can be converted into a propositional network (Schkufza, Love, and Genesereth 2008; Cox et al. 2009), which has a number of nodes which perform the logical operations as well as input, output and transition nodes to model the dynamics.

Approaches

The main evaluation of GGP players occurs in the International General Game Playing Competition (IGGPC) and is a good way to compare different approaches in GGP (Genesereth 2016). From 2005–2006, the winning agents used generic heuristic extraction aided minimax and included ClunePlayer (Clune 2007) and FluxPlayer (Schiffel and Thielscher 2007). This changed in 2007 when CadiaPlayer (Finnsson and Björnsson 2008) and Ary (Méhat and Cazenave 2011) entered using the upper confidence bound on trees (UCT) algorithm. Since then all winners have used a variation of UCT, with the exception of the most recent winner, WoodStock, which used a method based on constraint satisfaction programming (CSP) (Koriche et al. 2016).

Upper Confidence Bound on Trees

UCT starts with an, initially empty, game tree and runs a number of simulations. Each one starts at the root of the game tree and proceeds down the tree using a variant on upper confidence bounds (Auer 2003) until a leaf node is reached, as shown in Figure 1a. Here a new leaf node is expanded and a single Monte Carlo Search simulation is performed from that state to give an initial score to the new node as shown in Figure 1b. This initial score is propagated backwards to update all nodes above it in the tree with the new result, as shown in Figure 1c. Running many of these playouts gives progressively better approximations at each node on the tree of how good that state is and hence who is likely to win from there.

Each node stores a node count, $N(s)$, a count of the number of times action a has been taken from node s, $N(s, a)$, and a value for how good the current state is, $V(s)$. The choice of action at a state is then given by:

$$\mathbf{argmax}_{a \in A(s)} \left(V(\delta(s, a)) + C_{\text{uct}} \sqrt{\frac{\ln(N(s))}{N(s, a)}} \right)$$

Where $A(s)$ is the set of legal actions in state s, $\delta(s, a)$ is the transition function — the next state after taking action a from state s and C_{uct} is a constant.

AlphaZero

Recently, Silver et al. (2016) used a neural network-guided Monte Carlo Search Tree (MCTS) agent, AlphaGo, to beat 18-time world champion Go player Lee Sidol. This was generalised to AlphaGo Zero, which beat the original and learnt entirely from self-play reinforcement learning, as opposed to initially learning from expert games (Silver et al. 2017). Finally, this was generalised to AlphaZero which was separately trained on Go, Chess and Shogi and achieved state-of-the-art in each (Silver et al. 2018).

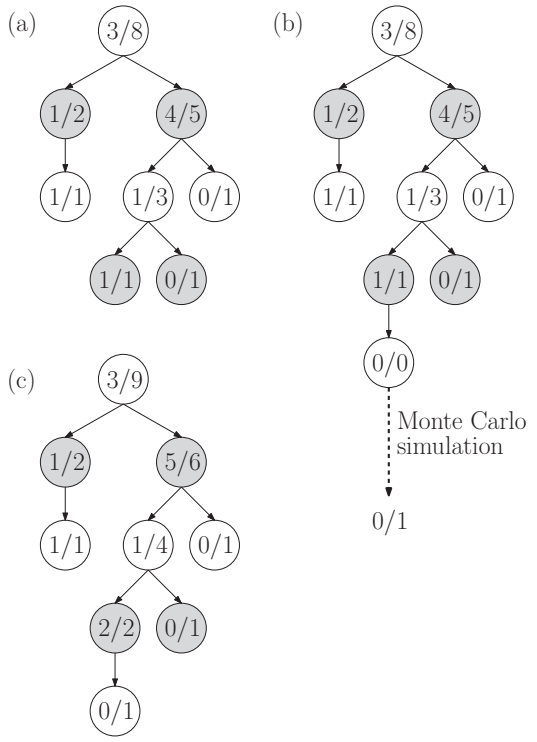

Figure 1: One simulation of an MCTS: (a) the game tree before the simulation (b) the Monte Carlo playout (c) the updated game tree

AlphaZero uses a single neural network which outputs both a probability distribution over actions and an expectation as to who will win. It plays through games with itself, at each state running a number of MCTS simulations to produce a better probability distribution over actions and sampling from that distribution. After a self-play game, the winner is recorded and passed back to all states that were played in that game. These triples of state, new action distribution and winner are then sampled in order to train the neural network (Silver et al. 2018). This process is shown in Figure 2.

The MCTS improvement operator works in a similar way to the standard UCT algorithm. It has an initially empty tree and each simulation expands a single leaf node. Each node stores a number of values, $N(s, a)$ is the visit count from taking action a in state s, $P(s, a)$ is the prior of taking action a from state s using the neural network policy output and $Q(s, a)$ is the average of all final evaluations in the nodes below $\delta(s, a)$ (Silver et al. 2017). Noise drawn from a Dirichlet distribution is combined with the prior at the root node of the MCTS in order to encourage exploration.

The tree is followed down again based on an upper confidence bound $Q(s, a) + U(s, a)$, where $U(s, a) =$

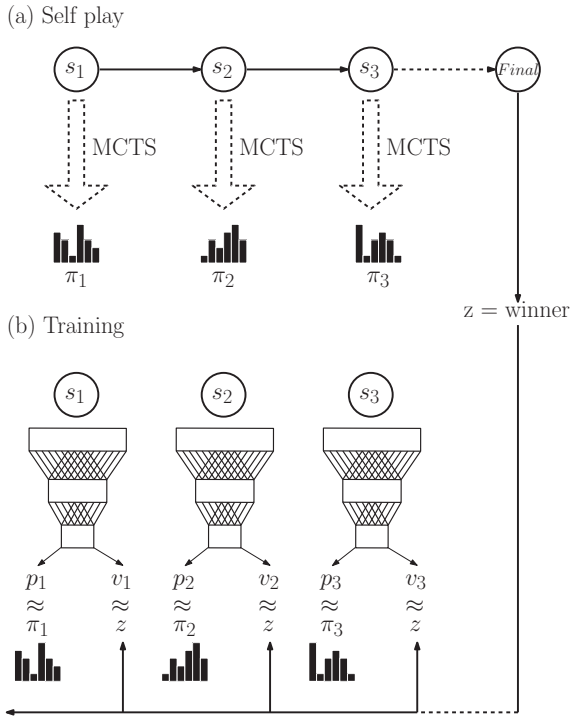

Figure 2: The self-play algorithm and training algorithm: (a) the self-play process for a game (b) the new training data generated for each state visited in the game, image based on Silver et al. (2017)

$C_{\text{puct}} \frac{P(s,a)}{1+N(s,a)}$ and C_{puct} is a constant. Once a leaf node is reached, an option is expanded and that node is evaluated using the neural network. The evaluation result is then propagated back up to all parent nodes in the tree.

Method

Learning Agent

There are a number of important limiting assumptions that AlphaZero makes about games that it plays, namely it assumes the following:

1. the game is zero-sum
2. the game is symmetric between players
3. the game is for two players
4. the game is turn based
5. the game has a board and a neural network architecture handcrafted for the features of that board
6. the rules of the game are known

The assumption number 6 is also assumed in GGP, extensions to cope with the first 5 are outlined below.

Cooperative games This is addressed by replacing the *expected outcome* output of the neural network (1 for a win, 0 for a draw, -1 for a loss) with an *expected reward*, or more specifically only allowing rewards between 0 and 1 — the rewards from GDL between 0 and 100 are rescaled by dividing by 100. After this change, each agent will simply try to maximise their own reward with no care for the others, so cooperative policies can be learned.

Asymmetric games To deal with asymmetry between players, one method would be to keep a separate neural network for each player which is trained separately, however, it can be noted that these will all be trained to extract very similar features from the game state, giving rise to the option of combining the early layers of all the players' neural networks, as shown in Figure 3. A similar unification of earlier layers was implemented in AlphaGo Zero when the value and policy networks were combined (Silver et al. 2017) and in machine translation with each language having a separate head (Gu et al. 2018) and provides a regularising effect on the network.

Multi-player and simultaneous games This method of having a head for each player has the added benefit of easily generalising to multiple players and allowing simultaneous play without having to make a pessimistic assumption of the other players knowing your move — simply have all players predict a move at once and use the combined move as the move choice to get to the next state.

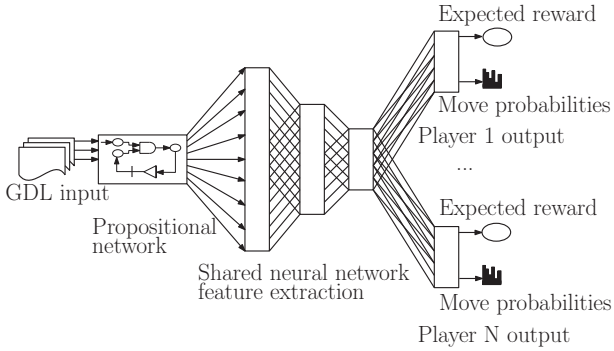

Figure 3: The architecture of the neural network.

Non-board games The final restriction to deal with is the reliance on a board and a handcrafted neural network for each game. In order to remove this limitation, we used the propositional network as the input to the neural network. The propositional network consists of a number of nodes representing the logic program described by the GDL encoding of the rules. This consists of a set of boolean nodes and includes nodes to input player moves, output move legality and model both turn-based dynamics and general logical operations. Together these nodes form a graph which represents the rules and dynamics of the game – given a set of input moves and the current state it will show the next state and next legal moves. The structure of this graph is constant and thus is useful as input into a neural network, in order to do so, we take all nodes from the propositional network and convert them to integers before feeding them into the network. We use all nodes rather than only the minimal set to represent the state as the rest of them contain calculations

Algorithm 1: Network initialisation

```
Algorithm: init_network(n_inputs, n_outputs,
  min_size=50)
size = num_inputs
while size ≥ min_size do
    Add ReLU fully connected layer with size nodes
    size = size / 2
Add ReLU fully connected layer with min_size nodes
Store current layer as state
for each role do
    Initialise current layer to build on to state
    head_size = min_size
    while head_size ≤ n_outputs[role] do
        Add ReLU fully connected layer with head_size
          nodes
        head_size = head_size × 2
    Store current layer as head
    Add softmax fully connected layer with
      n_outputs[role] nodes for policy head
    Add sigmoid fully connected layer to head with 1
      node for value head
```

Algorithm 2: High-level training loop

```
Algorithm: train(agent, model)
while Time left to train do
    Re-initialise game state
    while Game not finished do
        Perform MCTS with agent
        Record new action probabilities, π, and new
          expected value, q, for each player
        Sample moves from new action probabilities
    Record final result z for each player
    Add triples (state, π, rz + (1 − r)q) to replay buffer
    Train model on 10 mini-batches from replay buffer
```

from the rules that reduce the amount of computation that has to be done by the neural network which was important due to the small size of the neural networks used. The increase in number of inputs was not significant enough to cause problems and all propositional networks of the games tested had between 1000 and 5000 nodes.

This input was then passed through a number of automatically generated fully connected layers, which were generated as follows. Initially all inputs passed through to a series of fully connected hidden layers, with each one half the size of the previous, finishing with a layer of size 50. Then each player's head comes off that layer as a common start, doubling the size of the layer with more fully connected layers until it reaches the number of legal actions. At this point there is also a fully connected layer which goes to a single output for expected return prediction. All hidden layers use `ReLU` activation, the expected reward uses `sigmoid` and policy output uses `softmax`. This construction is shown in Algorithm 1. This diverges significantly from the convolutional residual blocks used in AlphaGo Zero (Silver et al. 2017) and AlphaZero (Silver et al. 2018). It has far smaller value and policy heads and contains different heads for each player. These changes were made for two main reasons. Most importantly was the fact that convolutional layers require knowledge about ordering which is not given in GDL. Though there is some work with extracting this information (Schiffel and Thielscher 2007; Kuhlmann, Dresner, and Stone 2006), these automatic methods may fail to detect an ordering and do not apply to, for example, non-board games which do not have a geometric structure. Additionally the large networks used for Go, Chess and Shogi all require large amounts of computational resources and a long time to train, making them difficult to use for GGP when training time is so limited.

While this generalised version would work with the original training setup from Silver et al. (2018), A single thread version is proposed in Algorithm 2 which simply runs a sample game, adds recorded data to the replay buffer, then trains on 10 mini-batches from the replay buffer and repeats. This method better fit the hardware being used for testing. The choice of training on 10 mini-batches (of size 128) was made to make it so that it will train on around 5% of the data in the replay buffer (of max size 20,000) at each stage, this means that each state will be used for training around 20 times before it leaves the replay buffer. This strikes a good balance between not training on each sample enough, which would end up requiring significantly more training data as it would forget information, and training on each sample too much causing it to overfit.

Another generalisation which was added was in what value the expected reward was trained to approximate. AlphaZero simply used the final result of the self-play game (referred to as z) as after many games it should average to a good estimate. Following Prasad (2018), information from the new expected reward at the root node after finishing the MCTS simulations was also considered — this is referred to as q below. These were combined using parameter r with the reward being trained to approximate $r \times z + (1 - r) \times q$. This parameter was varied in the range $r \in \{0, 0.5, 1\}$ for Connect-4 and the best value (0.5) was used for all further tests.

Finally, where AlphaZero used a single value α as a parameter for the Dirichlet noise added to the root of the MCTS search and scaled inversely with the average number of legal moves (Silver et al. 2018), the GGP generalisation explicitly scaled the noise inversely with the number of legal moves at that state. This is both more tailored to the state causing improved exploration and also simplifies the implementation as there is no need for initial randomised playouts to calculate the average number of legal moves.

Memory optimisations

One issue that appeared on large games was memory usage. Storing the entire state at each node in an MCTS proved to be too memory intensive for larger games, which inspired the following approach. The main observation is that typically a constant amount of the state changes at each step, this amount is very small compared to the size of the state

so a way is needed of storing only the changes. This was accomplished using a persistent array to store the state which allowed using $O(\log n)$ extra memory per change.

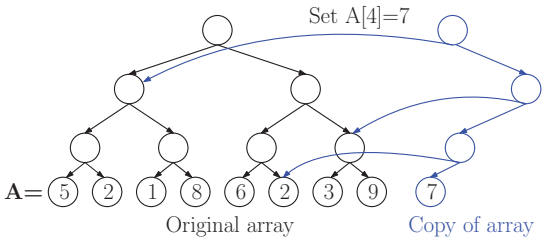

Figure 4: A persistent array with a single modification

The persistent array can be implemented in the standard way by building a binary tree on top of the state array after which updates can be accomplished through simply creating a new root node and new nodes down the path to the element that changed, but pointing to original nodes for subtrees which have not changed. This procedure is shown in Figure 4.

Experimental Evaluation

Evaluation Methodology

The following method was used to evaluate the agents. A set of 50 games were simulated of Generalised AlphaZero against a UCT agent with uniform rollouts. Both agents had a time limit of 2 seconds per move and each had their first 2 moves randomised. Additionally tests for Breakthrough were also run with a constant simulations limit.

UCT was chosen as a comparison as it forms a good benchmark due to variants of it being state-of-the-art for many years (Finnsson and Björnsson 2008; Genesereth and Björnsson 2013). However, both Generalised AlphaZero and UCT are mostly deterministic and fairly stable where they are not, with non-determinism only when two actions have the exact same count and during rollouts for UCT. This makes it difficult to run multiple tests to compare the two as all tests will result in the same game playouts. By randomising the first 2 moves, this allowed a number of independent tests to be run to give better results.

One important artefact of this randomisation method is due to it testing a wide variety of states and initial moves. This shows the breadth of the state space which the neural network has learnt but also means that it could win most games against a UCT agent but still lose when removing all randomisation and running a full game from the initial state.

All hyperparameters were tuned on Connect-4 with a 6×8 board, then evaluated on the following games from past GGP competitions (Genesereth and Björnsson 2013):

- Connect-4 (6×7 board) — 2 player, zero-sum, player-symmetric, turn based
- Breakthrough (6×6 board) — 2 player, zero-sum, player-symmetric, turn based
- Babel — 3 player, cooperative, player-symmetric, simultaneous

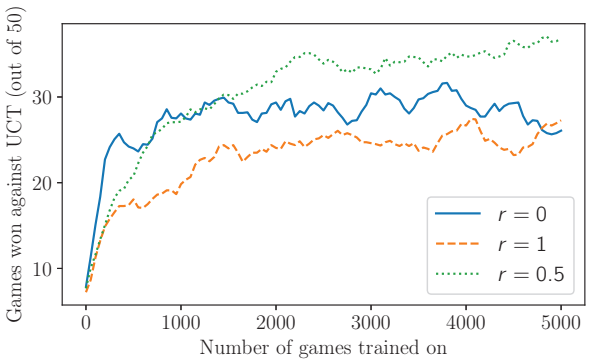

Figure 5: Three runs of training on Connect-4 (6×7 board), comparing $r \in \{0, 0.5, 1\}$, a moving average of kernel size 9 has been applied

- Pacman 3p (6×6 board) — 3 player, cooperative/zero-sum, player-asymmetric, mixed turn-based/simultaneous

Results and Discussion

As can be seen from Figure 5, a choice of $r = 0.5$ is the best option of the three considered. This makes sense intuitively as when using $r = 1$, it relies only on the final result meaning even if it is in a good state, a single bad action might cause a low final reward. On the other extreme with $r = 0$ it doesn't care enough about the long term so even though it becomes a good player quicker, the lack of grounding in final results causes more forgetting during training as seen in Figure 5. The simple but effective balance between long-term and short-term rewards obtained with $r = 0.5$ reflects the results of Prasad (2018) who additionally found even better performance with a linear dropoff between the two, however, this is less applicable to GGP as it requires knowledge of how long it will take to train and hence how quickly it should vary from 0 to 1. Moving average smoothing has been applied to Figure 5 in order to more clearly see the trends. Due to this averaging, error bars have been omitted, both for clarity and as they are less meaningful after smoothing.

Figure 6 shows Generalised AlphaZero's performance against UCT for Breakthrough for both a time and simulation limit. Interestingly, with a constant time limit, it quickly reaches winning 49–50 games out of 50 after just 400 games of self-play (just over 10 hours real time). This is significantly better performance than was experienced for Connect-4, which is the type of game that the parameters were tuned on. While this is good evidence of the generalisability of the methods given, it is also affected by the game in this instance. Breakthrough has a much higher branching factor of typically around 16 as opposed to just under 7 for Connect-4. UCT struggles with higher branching factors, but Generalised AlphaZero was originally designed for the massive branching factor of Go so is able to cope with it. Another advantage in Breakthrough is that the games can take a while if played out randomly, so the Monte Carlo rollouts in UCT take much longer than a single pass of a neural network

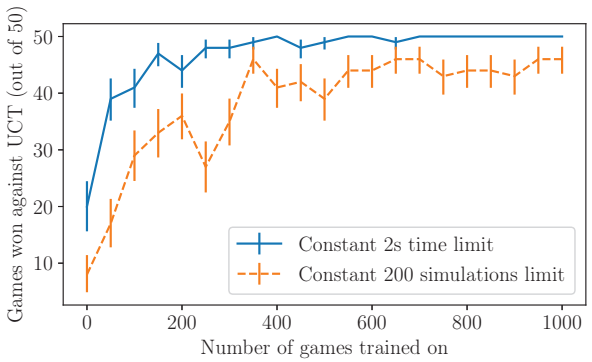

Figure 6: A training run of Breakthrough vs a UCT agent, error bars show the 80% confidence intervals.

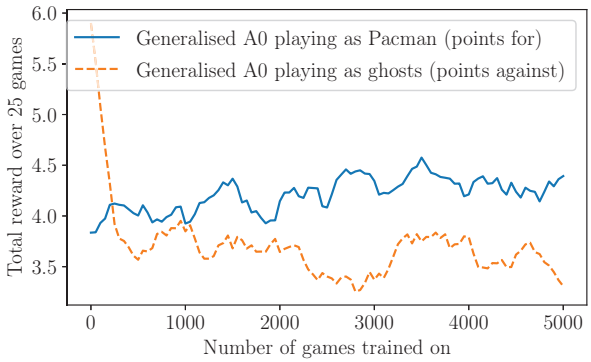

Figure 7: Points won by Generalised AlphaZero while playing as Pacman vs points won against Generalised AlphaZero while playing as the ghosts, a moving average of kernel size 9 has been applied

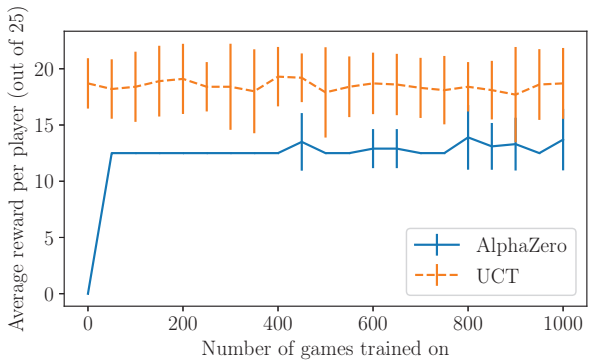

Figure 8: Three instances of Generalised AlphaZero playing Babel vs three instances of UCT playing Babel, error bars show 80% confidence interview

so the UCT agent does far fewer simulations, to determine how much of an effect this was having, an evaluation with a constant number of simulations per move was run. As can be seen from the graph this had only a minor effect on the final results, considering it gives the UCT agent 3-8x more time this is a very strong result.

The training of Breakthrough is easier than Connect-4 to fit into the 10-minute limit in GGP competitions as it beats UCT comfortably in only an hour of training, hence with simple optimisations it could outperform UCT after 10 minutes and with some parallelisation could be winning earlier.

Multi-player, asymmetrical games Pacman 3 player is a game where one player controls Pacman and two players work together against the first, each controlling a ghost. For the evaluation, it was played both with Pacman being a UCT agent and both ghosts being Generalised AlphaZero agents and the reverse. On all runs, the ghosts caught Pacman so reward as ghosts has been omitted from the graph shown in Figure 7. The 'points against' series shows how well Pacman as a UCT agent scored against Generalised AlphaZero — the learning of the network as the ghosts, while 'points for' shows how well Generalised AlphaZero performed as Pacman against UCT — the learning of the network as Pacman. It is quite clear from the graph that it learnt very well as the ghosts but far less convincingly as Pacman. There are a few explanations for this, the first is that there is a far simpler strategy for ghosts (namely to simply move towards Pacman) than there is for Pacman who has to strategise to evade the ghosts. This simpler strategy can be more easily learnt by the neural network but also good strategies for Pacman may simply be too complex for the very small network architecture used to be able to encode or need a far larger set of games to train on in order to be able to learn. The final point that may have had a large impact is that it typically has a branching factor of around 2–3 (maximum 4) as Pacman which is the type of game where Generalised AlphaZero doesn't have as much of an advantage over UCT.

Despite all this, Generalised AlphaZero performed comfortably better than UCT, even quite early in training. This shows that even for games outside of its ideal area, using combined cooperative, zero-sum, simultaneous, turn-based and player asymmetric games, Generalised AlphaZero is still able to beat UCT after training. Note that error bars have again been omitted due to smoothing.

Cooperative games Babel is a game where players work together to build a tower. The results of Generalised AlphaZero are shown in Figure 8 alongside the results of a UCT agent, each run consists of either three Generalised AlphaZero agents or three UCT agents — mixed teams were also tested but fell monotonically in between as expected so are omitted for clarity. In this instance, Generalised AlphaZero never outperformed a basic UCT algorithm. The consistency between the three players due to the setup of the UCT agent caused a large advantage here because players tended to perform correlated actions which worked well as a strategy. Despite this, it is clear that Generalised AlphaZero was able to learn and improve still and it is likely that with some extra complexity in the neural network architecture and more training time it would be able to learn a more sophisticated approach.

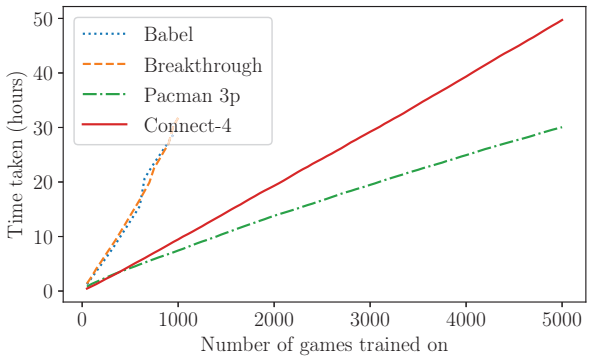

Figure 9: Time taken to train on different numbers of self-play games, for all tested games

For Pacman, it took 30 hours for the entire run, but only 7 hours before learning stabilised. The timings for Babel were similar with again 30 hours for the entire training run and 5 hours until learning stabilised. Again this is on the upper edge, but with optimisation and faster or more parallelised computing, this could fit in the 10-minute *startclock* of GGP competitions. As can be seen in Figure 9, all games except for Connect-4 were run for around 30 hours, Connect-4 was run for longer in order to make sure the results for varying r were meaningful. Tests were run on an Intel Core i5 running at 2.9GHz and used a GeForce GTX 780Ti graphics card for neural network operations. The speed of training appears to have been mostly dependant on the branching factor and the typical number of moves in a game. The branching factor changes how big a tree it has to look through, hence how quickly it can converge to a good path, with Pacman having by far the lowest branching factor and hence fastest training time. Breakthrough has the highest branching factor, however both Connect-4 and Babel have similar branching factors so the main difference in training speed there was due instead to Connect-4 having a strict 42 move limit whereas games of Babel can go for much longer, causing it to be on par with Breakthrough with respect to training time.

Conclusion and Future Work

We have developed an extension to the AlphaZero algorithm (Silver et al. 2018) for a truly general game-playing system that removes limitations based on assumptions that the games being played are:

- zero-sum
- turn-based
- two-player
- player-symmetric
- have a handcrafted neural network.

Our work extends the policy-value network to remove these restrictions and presents a way of automatically scaling it based on the size of the state of the game, allowing it to play any perfect information game.

Our results clearly show that using deep reinforcement learning within a GGP environment performs noticeably better than the UCT benchmark agent in a number of games. This approach also has the potential to improve dramatically with larger networks, more training time and further tuning.

The training times needed for the experiments have been well outside the typical 10 minute *startclock* of GGP competitions, however with speedups through simple optimisations, parallelisation and faster hardware a large amount of training could be squeezed in to at least perform better than a basic UCT agent in a competition format. Moreover, and in our view more importantly, the general notion of a GGP system should not be artificially restricted by the limitations of a specific competition format, and there is great value in a general AI system that improves further as it is given more time to learn new games.

Interesting future work includes rerunning at a larger scale with larger networks and games and more compute power. This would allow learning of more complex strategies and give more reliable results about the applicability of this extension in general. Additionally, the properties of the game, in particular branching factor, complexity and size of the propositional network, could be used to vary, for example, the replay buffer size, the number of simulations in the MCTS during both training and evaluation, the minimum layer size, as well as the parameters r and C_{puct} for the tradeoff between long/short term reward and amount of exploration/exploitation respectively. As well as these, the network structure could include known facts about the game such as player symmetries and convolutions over extracted ordinals in the game description to improve regularisation.

References

Auer, P. 2003. Using Confidence and Bounds for Exploitation-Exploration Trade-offs. *The Journal of Machine Learning Research* 3:397–422.

Benacloch-Ayuso, J. L. 2012. RL-GGP. http://users.dsic.upv.es/~flip/RLGGP/. Accessed: 14/04/2018.

Clune, J. 2007. Heuristic Evaluation Functions for General Game Playing. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, 1134–1139. AAAI Press.

Cox, E.; Schkufza, E.; Madsen, R.; and Genesereth, M. R. 2009. Factoring General Games using Propositional Automata. In *Proceedings of the IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA)*, 13–20.

Finnsson, H., and Björnsson, Y. 2008. Simulation-Based Approach to General Game Playing. In *Proceedings of the 23rd National Conference on Artificial Intelligence*, volume 1, 259–264. AAAI Press.

Finnsson, H., and Björnsson, Y. 2010. Learning Simulation Control in General Game-Playing Agents. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, volume 1, 954–959. AAAI Press.

Genesereth, M., and Björnsson, Y. 2013. The international general game playing competition. *AI Magazine* 34(2):107–111.

Genesereth, M., and Thielscher, M. 2014. *General Game Playing*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool.

Genesereth, M. 2016. International general game playing competition - past winners. http://ggp.stanford.edu/iggpc/winners.php. Accessed: 10/04/2018.

Gu, J.; Hassan, H.; Devlin, J.; and Li, V. O. K. 2018. Universal Neural Machine Translation for Extremely Low Resource Languages. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, volume 1, 344–354.

Hsu, F.-H. 2002. *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press.

Koriche, F.; Lagrue, S.; Piette, E.; and Tabary, S. 2016. General game playing with stochastic CSP. In *Constraints*, volume 21, 95–114.

Kuhlmann, G.; Dresner, K.; and Stone, P. 2006. Automatic Heuristic Construction in a Complete General Game Player. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, 1457–1462. AAAI Press.

Love, N.; Hinrichs, T.; Haley, D.; Schkufza, E.; and Genesereth, M. 2008. General Game Playing: Game Description Language Specification. Technical report, Stanford University.

Méhat, J., and Cazenave, T. 2011. A Parallel General Game Player. *Künstliche Intelligenz* 25(1):43–47.

Prasad, A. 2018. AZFour: Connect Four Powered by the AlphaZero Algorithm. https://medium.com/oracledevs/lessons-from-implementing-alphazero-7e36e9054191. Accessed: 15/09/2018.

Schiffel, S., and Thielscher, M. 2007. Fluxplayer: A Successful General Game Player. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, 1191–1196. AAAI Press.

Schkufza, E.; Love, N.; and Genesereth, M. 2008. Propositional automata and cell automata: Representational frameworks for discrete dynamic systems. In *AI 2008: Advances in Artificial Intelligence*, 56–66. Springer Berlin Heidelberg.

Shannon, C. 1950. Programming a computer for playing chess. *Philosophical Magazine 7* 41(314):256–275.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; Chen, Y.; Lillicrap, T.; Hui, F.; Sifre, L.; van den Driessche, G.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529:484–489.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; Chen, Y.; Lillicrap, T.; Hui, F.; Sifre, L.; van den Driessche, G.; Graepel, T.; and Hassabis, D. 2017. Mastering the game of Go without human knowledge. *Nature* 550:354–359.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science* 362(6419):1140–1144.

Wang, H.; Emmerich, M.; and Plaat, A. 2018. Monte Carlo Q-learning for General Game Playing. *ArXiv e-prints* ArXiv:1802.05944.