The Cost of Reinforcement Learning for Game Engines: The AZ-Hive Case-study

Danilo de Goede
d.degoede@uva.nl
University of Amsterdam
The Netherlands

Duncan Kampert
fjf@hotmail.nl
SURF
The Netherlands

Ana Lucia Varbanescu
a.l.varbanescu@uva.nl
University of Amsterdam
The Netherlands

Abstract

Although utilising computers to play board games has been a topic of research for many decades, the recent rapid developments in the field of reinforcement learning - like AlphaZero (and variants) - brought unprecedented progress in games such as chess and Go. However, the efficiency of this process remains unknown.

In this work, we analyse the cost and efficiency of the AlphaZero approach when building a new game engine. Thus, we present our experience building AZ-Hive, an AlphaZero-based playing engine for the game of Hive. Using only the rules of the game and a quality of play assessment, AZ-Hive learns to play the game from scratch.

Getting AZ-Hive up and running requires encoding the game in AlphaZero, i.e., capturing the board, the game state, the rules and the assessment of play-quality. And different encodings lead to significantly different AZ-Hive engines, with very different performance results. Thus, we propose a design space for configuring AZ-Hive, and demonstrate the costs and benefits of different configurations in this space. We find that different configurations lead to a *less* or *more* competitive playing-engine, but the training and evaluation for different such engines is prohibitively expensive. Moreover, no systematic, efficient exploration or pruning of the space is possible. In turn, an exhaustive exploration can easily take tens of training-years.

CCS Concepts

• **Computer systems organization** → **Multicore architectures**; • **Software and its engineering** → *Software performance*; • **Computing methodologies** → *Parallel programming languages*; *Massively parallel algorithms*; **Artificial intelligence**.

Keywords

energy efficiency; reinforcement learning; AlphaZero: Hive; game-playing engines; computational cost; design space exploration.

ACM Reference Format:
Danilo de Goede, Duncan Kampert, and Ana Lucia Varbanescu. 2022. The Cost of Reinforcement Learning for Game Engines: The AZ-Hive Case-study. In *Proceedings of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22), April 9–13, 2022, Bejing, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3489525.3511685

1 Introduction

Utilising computers to play board games has been a topic of research for many decades. Although chess is the most famous example of such a game [15], other games like Go, Atari, or Shogi have also seen increasingly stronger play over the years.

Traditional play engines rely on a combination of game-specific heuristics and alpha-beta pruning [11]. For instance, Deep Blue, programmed with this approach, managed to defeat the world chess champion Garry Kasparov in 1997 [12]. These approaches aim to replicate the decisions of human expert-players, and require expert datasets, which are often unreliable, expensive, or simply unavailable [19]. Moreover, some games are of such high complexity that the creation of heuristics good enough to beat top players is very difficult [2].

To overcome these challenges, machine learning became an interesting alternative [17], and the rise of reinforcement learning has changed the way we design playing engines for turn-based strategy games [4]. AlphaZero, Leela Chess Zero and MuZero have shown that it is possible to achieve superhuman performance in the games of Atari, chess, and Go, by *tabula rasa* reinforcement learning from games of self-play [13, 18].

Fundamentally, the advantage of the self-taught playing engines over traditional, heuristic-based approaches, amounts to learning heuristics by *practising*; consequently, it seems a playing engine for any game can be built that way. Unfortunately, the *efficiency* of this approach remains largely unknown, as the procedure comes at a significant price in terms of compute resources.

In this work, we investigate the benefits and challenges of the AlphaZero approach through a case-study. Specifically, we analyse the construction of AZ-Hive, a self-taught playing engine for Hive, a two-player tile-based strategy game that is unsolved: all existing engines are far from the level of expert human players [9].

Our exploration is based on a generalized AlphaZero framework [19]. To enable AZ-Hive to learn to play using this approach, we need to provide it with (1) the rules of the game, for it to explore different *correct* game actions, and (2) a testing procedure, for it to self-assess whether the actions being explored are beneficial. However, for a game as complex as Hive, how we *encode* these two components in the AlphaZero framework has profound consequences on the design, implementation, and performance of both the training phase and the resulting engine. To systematically analyse different encoding options, we define a *multi-dimensional design space*, where each design represents a possible encoding. We further explore this space to determine the best design for

ICPE '22, April 9–13, 2022, Bejing, China
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9143-6/22/04…$15.00
https://doi.org/10.1145/3489525.3511685

AlphaZero-Hive, i.e., the design that creates the best possible Hive playing engine. Finally, we estimate the cost of this exploration in terms of computational resources and energy.

Our results, based on the exploration of 10 different designs, indicate that AZ-Hive does indeed learn to play the game of Hive, but the computational cost for this process is significant. Moreover, we have not yet trained a super-human AZ-Hive engine, and, without rules to better explore the design space, the search for the best AZ-Hive can take 20+ compute-years.

In summary, this work makes the following contributions:

- We define a design space to capture the different aspects Hive encoding in AlphaZero; each design represents a different configuration, and is equivalent to a different AZ-Hive version.
- We demonstrate design space exploration to select the best configuration.
- We analyse the performance of different AZ-Hive versions in terms of playing capability, thus assessing the success of our design-space exploration.
- We quantify the energy cost of the full space exploration.

2 Background

2.1 Traditional game engines

Traditionally, game engines are built upon minimax [15], which performs a search through the game-tree until depth N. Many improvements exist for this algorithm, which either reduce the search-space (e.g., $\alpha - \beta$ pruning), or provide better guidance for the search (e.g., killer heuristic). The downside of using minimax is that, ultimately, the strength of the algorithm is bound by the accuracy of the board evaluation: the better the board evaluation, the quicker the convergence towards better board-states. This board evaluation is typically based on extensive expert player knowledge, and it requires both computer science skills and game expertise to be built into the game engine.

To work around the expertise required to build an exceptionally-strong board evaluation, Monte-Carlo Tree Search (MCTS) proposes a promising alternative: instead of a-priori assessing a game state using a well-crafted heuristic-based evaluation function, MCTS performs *playouts*, effectively playing out the game (randomly) until a terminal state (win/loss/draw) occurs. It will, for every possible move, keep track of the playout outcomes, and use them to provide an approximation of the strength of each move. In theory, when giving MCTS infinite amount of time, it will converge towards the optimal moves. The downside of MCTS is that for complex games, with high *branching factor* (i.e., many possible moves from a given state), it will take a lot of playouts (and, therefore, compute time) for this strength approximation to become meaningful.

2.2 The AlphaZero approach

Although traditional game engines have been successful in a wide variety of games, they heavily rely on the creation of heuristics. Despite decades of work to create such heuristics, there remain games for which the best traditional engines are only able to play at the level of human amateurs [16]. AlphaZero provides an alternative approach: the engine only knows the rules of the game, uses neither heuristics, nor game-specific knowledge, and learns how to play the game by playing against itself, and keeping track of the (in)successful moves and their outcome. In the following paragraphs we introduce the two main components of the algorithm, and further show how they are combined into a self-play reinforcement learning algorithm.

Neural network The first component of a playing engine is a deep neural network f_θ with parameters θ. The neural network takes a board state s as an input and produces a policy vector $\vec{p}$ and a value $v \in [-1, 1]$. The policy vector $\vec{p}$ contains probabilities over the possible moves such that the a-th component of $\vec{p}$ corresponds to the probability of selecting move a; v describes the probability of the current player winning the game from board state s. Intuitively, $\vec{p}$ describes which moves are considered to be good, and v describes who is winning from the current board state. The parameters θ of the neural network are initialised randomly and then updated using the self-play reinforcement learning algorithm that will be discussed in Section 2.2.2. The architecture of the neural network will be described in Section 4.3.5.

2.2.1 MCTS for policy improvement MCTS works on a search tree whose nodes correspond to game states and whose edges correspond to actions. An edge in the search tree corresponds to an action a, taken from board state s, and maintains (1) the prior probability $P(s, a)$ of choosing action a from board state s according to the policy vector $\vec{p}$ output by the neural network f_θ, (2) the visit count $N(s, a)$, i.e., the number of times we took action a from board state s, and (3) the action value $Q(s, a)$ that stores the reward for taking action a in state s.

MCTS iteratively expands the game tree by selecting a move a that maximizes the upper confidence bound $U(s, a)$ [19]. Next, the value v produced by the neural network f_θ is propagated back, updating $N(s, a)$ and $Q(s, a)$ accordingly. If the newly added node corresponds to a terminal state of the game, we propagate the actual rewards along the path (e.g., 1 for a win, 0 for a draw, and -1 for a loss). After several simulations, MCTS outputs a vector $\vec{\pi}$ of search probabilities, such that the a-th component of $\vec{\pi}$ is proportional to the visit count of the edge (s, a) [19]. The vector $\vec{\pi}$ generally selects stronger moves than the policy vector $\vec{p}$ produced by the neural network; intuitively, this allows the network to improve.

2.2.2 Self-play reinforcement learning The main idea of the self-play reinforcement learning algorithm is to use the neural network f_θ in combination with MCTS to create an iterative policy improvement procedure. Each iteration has three stages: (1)*generate training data* by letting the current iteration of the engine play many games against itself, with every move played according to $\vec{\pi}$, (2)*train the network* by adjusting the parameters θ of the neural network f_θ to maximise the similarity between $\vec{p}$ and $\vec{\pi}$, and to minimise the error between the predicted and the actual winner [19], and (3)*test the new network*[1] by letting it play a number of games against its previous version to determine (and keep) the better version.

Using this iterative algorithm for many iterations, the neural network incrementally learns which game states are advantageous for a given player, and which actions lead to such states. The computational cost of this iterative algorithm is the accumulated cost of the games played against itself, the network training cost for a

[1]Technically, a generic AlphaZero framework might skip this step; however, in our design, we chose to use it.

large number of epochs, and, possibly, the cost of the playoff games for validation, all repeated for many iterations.

3 Related Work

AlphaZero for games Google's DeepMind first major contribution to solving games through reinforcement learning is *AlphaGo*. It is the first computer program to defeat a human professional player in the game of Go [16], a long-standing milestone for artificial intelligence. After the success of AlphaGo, DeepMind introduced *AlphaGo Zero*, which is based solely on reinforcement learning [19]. AlphaGo Zero is not provided with any data, guidance or domain knowledge besides the rules of the game. In the same year, DeepMind showed that AlphaGo Zero's approach can be generalized to other games such as chess and Shogi [17]. This algorithm was able to achieve superhuman performance in all of these games within a day of training on a single machine with 4 TPUs. Since AlphaGo Zero's publication, learning through self-play has become an influential and competitive method to build game engines, especially due to its perceived accessibility. In this work we demonstrate that, although getting started with AlphaZero is straightforward, being successful in designing a competitive game-engine comes at a significant cost.

The computational and energy cost of AI Our work aligns well with the message of [14], in that we argue for quantification (and raising awareness) of the cost of AI workloads. To this end, our paper provides a much more concrete and detailed example of one AI domain - designing game engines based on AlphaZero - whose costs can easily go out of hand when no game knowledge is included. While many efforts exist to quantify the performance of AI (see a collection here, for example, MLPerf[2]), a lot less research has been invested in assessing the cost and energy efficiency of existing workloads for ML in general, and DNN or reinforcement learning in particular. For example, work by Sze et al. in [20, 21] look into the methodology for assessing and quantifying energy efficiency, while providing examples of realistic cases and guidelines. Our work, instead, focuses on the quantification of the (energy) cost a specific workload, from design to result, while contrasting it against its perceived ease-of-use. There exists also quite some work on quantifying the energy efficiency of specific accelerators [24], or designing and implementing energy efficient processors for DNN workloads [10, 23]. Such work is complementary to ours: indeed, our cost analysis does depend on the hardware in use, and improving its efficiency would diminish the cost. However, we also argue that the *principle* of reinforcement learning in the context of AZ-Hive can be improved to further contribute to improved energy efficiency.

4 Designing AZ-Hive

In this section we introduce the game of Hive, its implementation, and introduce the different dimensions of encoding the game into the AlphaZero approach.

4.1 The game of Hive

Rules Hive is a two-player *tabletop abstract strategy game* designed by John Yianni and published in 2001 by Gen42 games [25]. The goal of the game is to surround the opponent's *Queen Bee* with other pieces; the first player to do so wins.

[2]Available here: https://mlcommons.org/en/

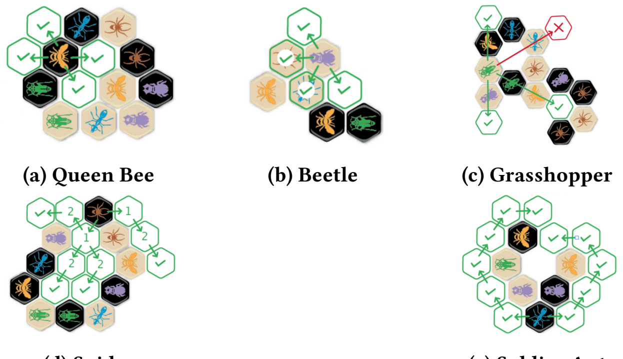

(a) Queen Bee **(b) Beetle** **(c) Grasshopper**
(d) Spider **(e) Soldier Ant**

Figure 1: The rules for moving different tiles in Hive [1].

Both players start with 11 hexagonal tiles of *different types*: queen bee×1, beetle×2, grasshopper×3, spider×2, and soldier ant×3. Players take turns to place or move tiles. The tiles in play form a playing surface, called the *Hive*, and the rules prohibit any actions or moves that disconnect the Hive. The type of tile determines how it can move through the Hive (see Figure 1).

If a player has no possible moves, the turn passes to the other player. The game ends when one queen is completely surrounded by tiles of any colour. If both queens get surrounded in the same move, the game ends in a draw. Both players may also agree on a draw when they are both forced to make the same moves repeatedly.

Game complexity There are two metrics commonly used for game complexity: the *size of the state space* and the *branching factor*. Finding the theoretical state space size for Hive is challenging due to the dynamic nature of the states; however, to attempt an illustrative estimation, there are $\frac{22!}{(2!\cdot 2!\cdot 3!\cdot 3!)^2} \approx 5.4 \cdot 10^{16}$ possible ways to arrange the 22 tiles of Hive in a straight line. And this is only a fraction of the huge space of possibilities. The Hive branching factor increases as the game progresses. After less than 20 moves, the average branching factor is already above 50 (when random moves are played) [9]. For comparison, the average branching factor of chess is 35 [3].

4.2 AZ-Hive implementation

Our implementation of a playing engine for the game of Hive, AZ-Hive, is based on the AlphaZero approach. To enable AZ-Hive to learn to play, we need to provide it with the rules of the game, for it to explore different *correct* game actions. To this end, we choose for an existing high-performance, C-based Hive implementation, called *BeeKeeper* (BK) [9]. In this work, we preserve the back-end of Beekeeper (i.e., its implementation of the mechanics of the game, like making moves, testing for correctness, testing for a win, etc.), but adapt its front-end to enable different encodings of the board state, the moves, and some of the rules, which eventually define different AZ-Hive versions.

To implement our AlphaZero-based game engine for Hive, we use the AlphaZero General framework[3], an implementation of the work by Silver et al. [19]. This framework provides the required *software training infrastructure* for the full self-play learning procedure, effectively providing a Python-based skeleton for the algorithm presented in 2.2.

[3]Available at: https://github.com/suragnair/alpha-zero-general

Because the training infrastructure and the neural network are both written in Python, our game implementation has to communicate between C and Python to integrate the BeeKeeper's logic of the game into the training infrastructure. Consequently, we compile BeeKeeper as a shared library file and let our game implementation interact with it using *ctypes* [4], which allows us to call functions from dynamic link libraries (DLLs).

4.3 The design space

For a game as complex as Hive, how we *encode* the game-specific components in the AlphaZero framework has profound consequences on the design, implementation, and performance of both the training phase and the resulting engine.

In general, the neural networks' input data is an array of numerical data of some finite dimension (encoding the training data), while its output is another numerical (encoding some form of probability that helps decisions). In the case of AlphaZero, the input data effectively consists of the board state, while the output data ultimately represents a score for the moves/actions to be taken. *Encoding the game* thus means finding a representation of the board state and moves that fits this numerical model well. The size and data types of these arrays further play a role in the architecture of the network. Finally, all these decisions influence the training effectiveness (i.e., how good of an engine we can train) and efficiency (i.e., how fast the training completes).

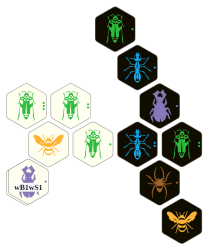
To systematically analyse different encoding options, we define a *multi-dimensional design space*, whose dimensions correspond to different design choices. Every combination of design choices corresponds to a single *configuration*, i.e., a single point within the *design space*. In turn, each of these configurations that is ultimately implemented and trained into a fully-fledged playing engine becomes an AZ-Hive version. This section introduces the dimensions of the design space and motivates a number of promising design choices for each dimension.

4.3.1 Game state representation In contrast to games that are played on a board with a *fixed* grid structure, coming up with a Hive board representation that fits the "array-of-numbers" model, and thus can be fed into the neural network, is not trivial because (a) the tiles are hexagonal, and (b) it is not played on a physical board. We address problems by skewing the axes of a 2-dimensional array of size 26×26 [5], such that the six neighbours of a hexagonal tile with coordinates (x, y) have coordinates $(x-1, y)$, $(x+1, y)$, $(x, y-1)$, $(x, y+1)$, $(x-1, y-1)$, and $(x+1, y+1)$. Such an array is large enough to capture any possible state of the game of Hive, whose "widest" hive spans 22 tiles placed in a straight line.

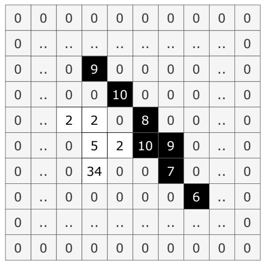
With this representation, we can encode the individual tiles as a number between 0 (emtpy) and 10. This representation can be fed into the neural network directly, but it does not deal with a *beetle*'s capability to be placed on top of other tiles. To further address this problem, we concatenate the individual digits of the tile encoding. To illustrate this, Figure 2b shows the board representation of the Hive structure shown in Figure 2a.

[4]https://docs.python.org/3/library/ctypes.html

[5]We added 2 layers of padding on each side of the board for computational efficiency.



(a) A Hive game state.



0	0	0	0	0	0	0	0	0	0
0	..	..	..	..	..	..	..	..	0
0	..	0	9	0	0	0	0	..	0
0	..	0	0	10	0	0	0	..	0
0	..	2	2	0	8	0	0	..	0
0	..	0	5	2	10	9	0	..	0
0	..	0	34	0	0	7	0	..	0
0	..	0	0	0	0	0	6	..	0
0	..	..	..	..	..	..	..	..	0
0	0	0	0	0	0	0	0	0	0

(b) Encoding of state

Figure 2: A game state with stacked tiles and the corresponding board representation.

4.3.2 Action encoding Recall from Section 2.2 that both the neural network and MCTS output their own policy vectors, encoding a probability distribution of possible moves. Therefore, an action has to be encoded as a non-negative integer, such that the *n*-th component of the policy vector corresponds to action *n*.

We propose three encodings:

(1) *Action-list encoding:* One naive approach is to build a list of possible actions, and encode each action according to its index within this list. One problem with this approach, however, is that the interpretation of the action encoding depends on the state of the board. That is, a given action encoding may correspond to two completely different moves for different board states.
(2) *Absolute coordinate encoding:* To eliminate the dependency between the board state and the action encoding, we propose *absolute coordinate encoding*, which defines an action as a set of 11 planes of size 26×26. Each plane corresponds to one tile that can be moved, and the location within that plane corresponds to the destination of the move. Although this encoding provides an action with a consistent definition and a geometric interpretation, it increases the size of the action space significantly. In particular, the size of the action space is $26^2 \cdot 11 = 7436$, because there are 26^2 possible destinations and each player owns 11 tiles. This large action space increases the complexity of the neural network, as it has to optimize a policy vector $\vec{p}$ of 7436 components.
(3) *Tile-relative encoding:* To address the size problem of absolute encoding, we define an action based on the tile being moved and the *location* (i.e., the neighbour and its "side") it is placed in. Thus, an action is a set of 22 planes of size 11×7. Each plane corresponds to a tile the moved tile may be moved next to. The row within each plane then describes the tile that is being moved, and the column describes on which of the possible sides the neighbour the tile is moved. [6] Hence, the tile-relative encoding has an action space of size $22 \cdot 11 \cdot 7 = 1694$ - still huge, but significantly smaller than the space of absolute coordinate encoding.

4.3.3 Board representation Representing the board such that we enable the neural network to recognize and learn patterns that generalize is crucial for the strength of AZ-Hive. We proposed in total *five* board representations.

Intial The naive representation (Section 4.3.1) stores the board as a 26×26 array of tiles, where each stack of tiles is encoded as

[6]Note that we require 7 columns due to the *beetle*'s capability to move on top of a tile.

a concatenation of positive integers between 0 and 10. Although this board representation uniquely identifies a board state and is therefore suitable for the logic of the game, it may be sub-optimal for the neural network to understand the meaning behind the concatenation of tile numbers.

Symmetric : The symmetric board representation only encodes tiles that are visible from the top view of the board, thus disregarding tiles that are underneath a *Beetle*, which are not able to move. This representation is symmetric in the sense that it encodes a tile such that the board representation corresponding to the inverted tile can be obtained by negating the original tile encoding. Because there are 5 types of tiles, the domain of the symmetric board representation is $[-5, 5]$, where the integer 0 is assigned to the empty tile, positive integers are assigned to tiles of the player who is to move, and the remaining integers are assigned to tiles of the opponent.

Simple : Similar to the symmetric board representation, the simple board representation does not encode the information of tiles hidden below a *Beetle*. We further reduce complexity by abstracting tile type: we treat the *Beetle*, *Grasshopper*, *Spider*, and *Soldier Ant* the same. Intuitively, this board representation should make the win condition easy to identify for the neural network, as the types of the tiles surrounding the *Queen Bee* do not influence whether a player wins the game.

Binary planes : It is also possible that the neural network can more easily recognise patterns when the tile types are integrated into the board representation in a discretised manner. Instead of representing a tile type as a certain integer, we can encode them by allocating binary planes for each tile type. We end up with a board representation with 3 dimensions: the first two specify the locations of the tiles within the board, and the last describes the tile type. The binary planes board representation uses ten planes, one per player per tile type. A single binary plane is filled with zeros and contains a one at coordinates at which a tile of the type for which the plane is allocated is present.

Hybrid : The hybrid board representation is a mixture between the binary planes board representation and the other representations: it contains two binary planes that are used to encode the positions of both *Queen Bees*, and two additional planes describe the positions of the remaining tiles for each player using an integer encoding between 1 and 4. Similar to the simple board representation, the purpose of this board representation is to make the win condition easy to identify for the neural network by allocating a separate plane for the *Queen Bees*.

4.3.4 *Optimizing the training procedure*

Invariance under rotation and reflection The rules of the game of Hive are invariant under reflection and rotation by 60 degrees. We can exploit this property to increase the amount of training data by a factor of twelve because we can rotate the board 6 times and for each rotated version of the board we can reflect the board. Being able to exploit these invariances allows us to significantly speed up the training procedure.

Game rule modifications As Hive is a game of high complexity, we also attempted to adapt the game rules to reduce this complexity. For example, we modified the rules of the game such that a player wins when 4 tiles surround the *Queen Bee* of the enemy, thus decreasing the average turn at which the game ends, and speeding up the training procedure. This modification also mitigates the problem of a high branching factor by reducing the number of long games. In addition, we introduced a turn limit of 30 moves after which the player with the least tiles surrounding its own *Queen Bee* wins the game. If both players have the same number of tiles surrounding their *Queen Bee*, black wins the game. Finally, a draw is no longer forced when the same board state is reached three times. With these game rule modifications, draws are no longer possible, which increases the efficiency of the training data generation.

4.3.5 Neural network architecture The Hive playing engine uses a convolutional neural network (CNN) that was implemented in PyTorch. [7] The CNN takes one of the board representations introduced in Section 4.3.3 as an input. The board representation is first fed through *either 4, 6 or 8 convolutional blocks* depending on the architecture that is desired. Each convolutional block applies the following operations:

(1) A convolution using 256 filters of size 3×3 with stride 1;
(2) Batch normalization, which allows us to use higher learning rates and be less careful about parameter initialization [8];
(3) A rectifier nonlinearity (ReLU) activation function.

The output of the convolutional blocks is then passed into two separate *heads*, one head outputs the policy $\vec{p}$ and the other head outputs the value v. The policy head consists of two fully connected layers and then performs a softmax function. The output of the policy head is a vector with a size that corresponds to the action encoding that was used. The value head also consists of two fully connected layers, which are followed by a Hyperbolic tangent activation function that outputs a scalar in the range [-1, 1].

4.3.6 Hyperparameters There are many hyperparameters that influence the learning process of the Hive playing engine, such as the number of self-play games in each training iteration (*numEpisodes*) or the number of moves to be simulated to expand the search tree of MCTS (*numMCTSSims*) - see [6] for a full list. To reduce the dimension of the design space we search in, we do not base the hyperparameters on extensive empirical analysis. Instead, we used the hyperparameters suggested by Wang et al. as a starting point and balanced them to ensure that a single iteration of training takes no more than 20 minutes [22].

5 Empirical evaluation

To determine the best design for AZ-Hive, this section looks into different configurations in the design space and empirically evaluates its performance. We run all our experiments on a cluster node featuring an Intel Xeon Bronze 3104 CPU and an NVIDIA GeForce GTX 1080 Ti. This empirical evaluation includes the correctness of the training infrastructure, the playing strength, and the playing speed of the engine.

Three aspects make up a successful implementation, namely (a) whether the training infrastructure is working correctly, (b) how strong the Hive playing engine is, and (c) how much time the engine spends on deciding upon a move. We briefly discuss them in the following paragraphs.

[7] https://pytorch.org/

5.1 Correctness of training infrastructure

To test whether the network actually learns, we investigate the progression of the policy loss and value loss of the neural network. To do so, we train five neural networks based on the tile-relative encoding and hybrid board representation for ten iterations, and measure the losses after every epoch of training. The results, presented in Figure 3, indicate that both losses decrease quickly and then converge to some value over time, indicating progress. We also observe that the loss spikes at the first epoch of every iteration due to the newly-added, generated training data.

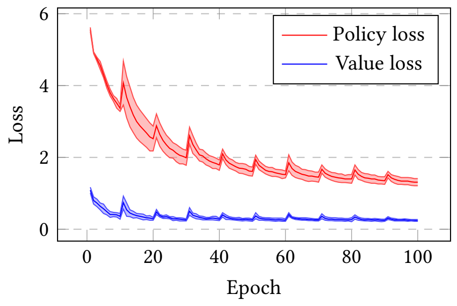


Figure 3: Policy and value loss over 10 training iterations.

5.2 Playing strength

To evaluate playing strength, we use *the win rate against the random agent* and, when the performance analysis involves more than two engines simultaneously, we generalise this metric by introducing Elo ratings.

Win-rate of different models: To test whether the win rate improves (consistently) during training, we train several AZ-Hive configurations that use a 4-layer CNN with the same set of hyperparameters, capping the training time to 4h per configuration.

During the training procedure, we track how many models are produced, and how many of those models are accepted. The iteration of a model increases when this model is accepted. For every accepted model, we play 50 games against the random agent and measure win rate. We repeat the same experiment 5 times per configuration, report the average. When a subset of the 5 experiment instances of a single configuration reaches a certain model iteration, we average the measurements of this subset to benchmark that model iteration.

The first set of experiments uses *the absolute coordinate encoding*. Figure 4 shows the win-rate evolution per configuration during the training procedure when pitting against a random agent. Over time, the win rate of each configuration improves, but the fluctuations in win rate indicate that this improvement is not consistent. This inconsistency can be explained by the fact that (a) the win rate is taken from only 50 games, and (b) it is not guaranteed that the neural network gets better during a single iteration. We also observe that, for this specific action encoding, the *original* board representation achieves the highest win rate after 4 hours of training.

Next, we repeat these experiments with the *tile-relative encoding*. Figure 5 shows that tile-relative encoding provides similar performance for the symmetric and binary planes board representations. However, the hybrid board representation performs better, while the original and simple board representation perform significantly

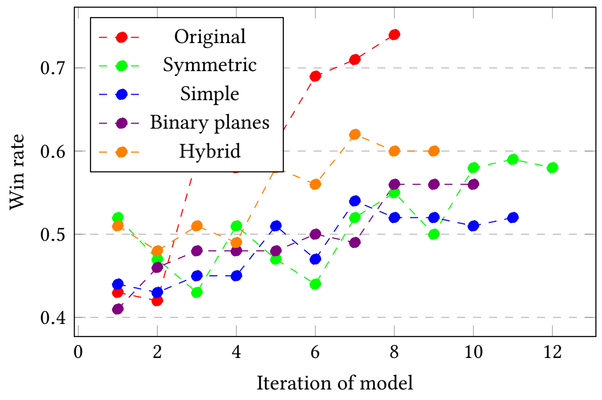


Figure 4: Win rates of different configurations based on the absolute coordinate encoding when playing against a random agent.

worse. In fact, no notable win rate improvement is observable for the original and simple board representations.

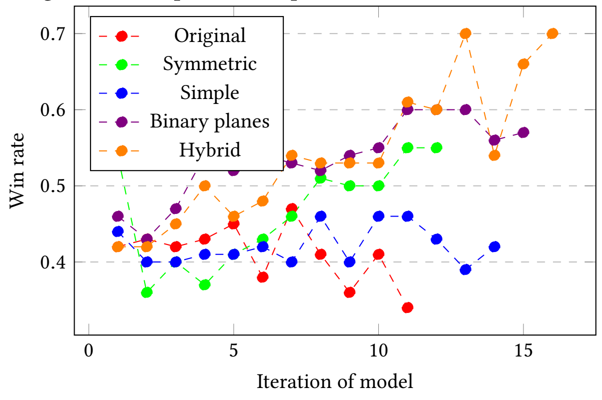


Figure 5: Win rates for different configurations based on the tile-relative encoding when playing against a random agent.

Furthermore, we investigated the performance of a number of neural network architectures. Although we expect that increasing the number of layers provides the neural network with more versatility to tune its weights per layer, it could increase the cost of the optimization procedure. Additionally, we evaluate the performance of each neural network architecture both with and without exploitation of invariance, as we expect including it increases the performance of the engine (as the procedure provides more training data to the neural network). Our results (omitted due to space limitations) show that increasing the number of layers of the neural network does not necessarily increase the performance of our engine. However, the quality of a single iteration is much higher when invariance is exploited. Optimizing the hyperparameters such that they are more suitable to handle more training data could allow the invariance exploitation to become more practical.

Elo rating To get a clearer understanding of how the self-play based engine performs against engines that employ traditional approaches, we set up a tournament involving *six engines*. First, we use *Self-play RL* AZ-Hive engine, which combines tile-relative action encoding with hybrid board representation, and exploits invariance to generate training data; we "enter" two instances in the tournament, trained over 4h and 24h, respectively. Additionally, we use an *Untrained self-play RL* AZ-Hive engine, which corresponds to the self-play based engine before it is trained. To represent non AlphaZero approaches, we use a *Random* engine that corresponds to the random agent that served as a baseline implementation in our previous empirical analyses, and add two engines from BK [9]: *Minimax* and *Classic MCTS*.

The engines are pitted against each other in a round-robin fashion. Each engine plays a total of 400 matches (100 matches per opponent). We maintain an Elo rating for each engine throughout the tournament. If engine A has a rating $R(A)$ and engine B has a rating $R(B)$, we can estimate the probability that engine A beats engine B using the logistic function [7, 19]: $P(\text{A defeats B}) = \frac{1}{1+\exp(c_{\text{elo}}\cdot(R(B)-R(A)))}$, where $c_{\text{elo}} = \frac{1}{400}$ is the standard constant. We estimate the ratings $R(\cdot)$ of each engine using Bayesian logistic regression using the BayesElo program [5]. We initialised the Elo rating of each engine to 1000 and set the first mover's advantage to 0 to negate its impact on the Elo gain throughout the tournament.

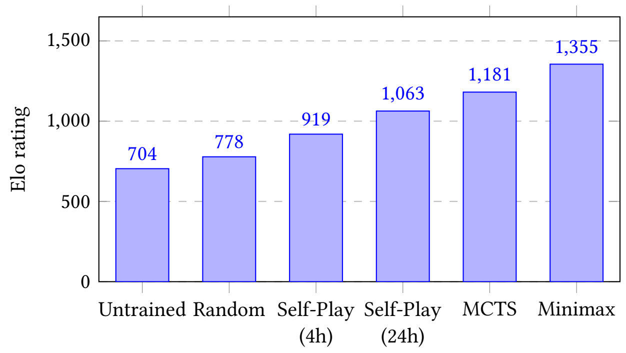


Figure 6: The Elo ratings of different Hive engines.

The final Elo ratings of the engines after the round-robin tournament are summarised in Figure 6. We observe that both traditional approaches outperform our self-play based engine. Nonetheless, in terms of performance, the self-play based engine comes much closer to the traditional approaches than the untrained and random engine. In fact, the self-play based engine gained 359 Elo rating in 24 hours of training.

5.3 Playing speed

We estimate an engine's playing speed in terms of the average Thinking Time per Move (TTM). Because TTM mostly depends on the number of MCTS simulations per move, we measure TTM for a number of MCTS simulations $M \in \{100t \mid t \in \mathbb{Z}^+,\ t \leq 25\}$. Our results indicate that TTM grows linearly with the number of MCTS simulations - as expected, given that the cost of every simulation is independent of the size of the search tree.

We estimate human move speeds from basic tournament rules and game archives: 10 minutes [25] thinking time per player, with an average game taking 20 moves per player, indicate a desirable TTM of 15 seconds. TTM for AZ-Hive exceeds 15 only for more than 1800 MCTS simulations, an unnecessarily large number; thus, all our AZ-Hive versions are sufficiently fast to play against a human.

6 Cost analysis

As illustrated by the results in Section 5, finding the strongest AZ-Hive engine requires a systematic design-space exploration. In this section, we evaluate the cost of this exploration in terms of computational resources and energy.

6.1 Exploration cost

In Section 4.3, we described several choices that can be made within the design space. An exploration of the design space is necessary to determine the best performing configurations for the game[8]. However, because there are no rules in AlphaZero's methodology to discern between good and bad designs for a game like Hive, exploring the design space requires trying all possible configurations, which, in turn, is extremely compute-intensive. We estimate the cost of this exploration by assessing the size of the design space and the cost to assess every instance (Section 6.2).

Our current design space has 4 dimensions, with different options each: action encoding (4), state representation (5), NN architecture (3), and game rules (2). Thus, our current design space features 120 configurations. For each configuration, we used a fixed set of hyperparameters (Section 4.3.6). In addition, we based our findings on 5 training episodes per configuration. In an ideal scenario, we would base our findings on significantly more training episodes, and analyse 3 different values for the hyperparameters *numEpisodes* (50, 100, 200), *numMCTSims* (25, 50, 100), and *cpuct* (0.5, 0.8, 1).

Let us denote *numEpisodes* with E, and *numMCTSSims* with M. Based on the training episodes we performed, a typical iteration takes approximately $5\cdot\frac{E}{100}\cdot\frac{M}{25}+10+3\cdot\frac{M}{25}$ minutes (using the same Intel Xeon Bronze CPU and a 1080Ti GPU node from Section 5) with invariance exploitation and the modified rules. Using training episodes of 20 iterations leads to a training time, for a single neural network design for the original game, of $20\cdot(5\cdot\frac{E}{100}\cdot\frac{M}{25}+10+3\cdot\frac{M}{25})$ minutes. Based on these estimates, the total computation time needed to explore our *120 designs* with *27 different hyperparameter combinations* based on *100 training runs* is approximately:

$$100\cdot 120\cdot \sum_{E\in\{50,100,200\}}\ \sum_{M\in\{25,50,100\}} 3\cdot 20\cdot(5\cdot\frac{E}{100}\cdot\frac{M}{25}+10+3\cdot\frac{M}{25}) \quad (1)$$

Filling in the different values for E and M in Equation (1), we estimate the computational cost of exploring our design space to be $\approx 2.0\cdot 10^8$ minutes, which is about *381 node-years*. In energy, this is equivalent to *602.78MWh* (see Section 6.2 for our calculations), which is enough energy for roughly 90 European citizens for an entire year [9]. This approximation illustrates the cost (and difficulty) of applying AlphaZero to a complex game such as Hive.

6.2 Energy cost per instance

This section presents a detailed analysis of the cost of exploring the AZ-Hive design space.

To estimate the overall energy cost of exploring the AZ-Hive design space, we first estimate the energy consumption of training a single instance. To do so, we measure the CPU and GPU energy consumption for a typical AZ-Hive configuration (i.e., hybrid board representation and tile-relative action encoding) during its first 10 iterations. We use *Likwid Powermeter* [10] and *NVIDIA System*

[8]In this context, *best performing* refers to playing strength. However, the exploration principle is metric-agnostic.
[9]https://data.worldbank.org/indicator/EG.USE.ELEC.KH.PC?locations=EU
[10]https://github.com/RRZE-HPC/likwid/wiki/Likwid-Powermeter

Management Interface (NVIDIA SMI) [11] to measure the energy consumption for the CPU and GPU, respectively.

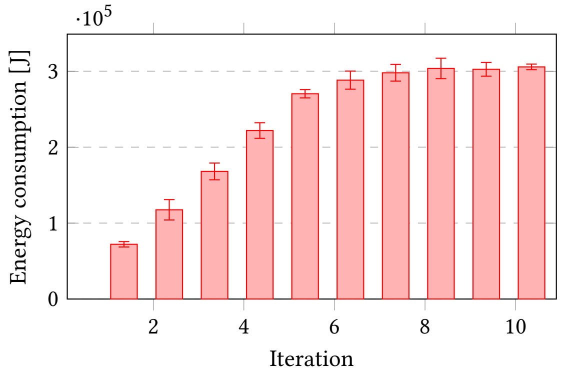


Figure 7: GPU energy consumption [J] during the first 10 episodes when training a typical self-play based configuration.

Figure 7 presents the GPU energy measurement results. We observe that the energy consumption linearly increases in the initial 5 iterations - this is due to the linear increase in the number of training examples used during the neural network training phase (Section 2.2.2), and then stabilises at roughly $3 \cdot 10^5$ Joule.

Using this figure, we can approximate the energy consumption of the GPU E_t^{GPU} (in Joules) during iteration t as:

$$E_t^{\text{GPU}} = \begin{cases} (0.55t + 0.2) \cdot 10^5 & \text{if } t < 5 \\ 3.0 \cdot 10^5 & \text{if } t \geq 5 \end{cases} \tag{2}$$

The total energy consumption (in Joules) of the GPU when training a typical configuration for 20 iterations is then:

$$\int_1^{20} E_t^{\text{GPU}} dt = 10^5 \cdot \left(\int_1^5 0.55t + 0.2dt + \int_5^{20} 3.0dt \right) \approx 5.2 \cdot 10^6$$

Similarly, we found an analytical approximation for the energy consumption of the CPU E_t^{CPU} (in Joules) during iteration t to be:

$$E_t^{\text{CPU}} = \begin{cases} (0.9t + 3.3) \cdot 10^4 & \text{if } t < 5 \\ 7.8 \cdot 10^4 & \text{if } t \geq 5 \end{cases} \tag{3}$$

The total energy consumption (in Joules) of the CPU when training a typical configuration for 20 iterations is then:

$$\int_1^{20} E_t^{\text{CPU}} dt = 10^4 \cdot \left(\int_1^5 0.9t + 3.3dt + \int_5^{20} 7.8dt \right) \approx 1.4 \cdot 10^6$$

Consequently, the total energy consumption of training a typical configuration for 20 iterations (CPU + GPU) is $6.6 \cdot 10^6$ Joules.

7 Conclusion

In this work, we assess the energy cost required for using AlphaZero, a reinforcement learning approach, to build a playing engine for the game of Hive. We illustrate the challenges of encoding a new game in the AlphaZero framework, and how it leads to many different engine implementations, with various playing strengths. To systematically explore all these options, we propose a design space, with multiple dimensions. We further explore this space and show different configurations and their performance.

Finally, we assess the cost of training all these different versions of the game. Given that none of them has reached super-human playing strength, we calculate a comprehensive search for the best possible AZ-Hive will take tens of node-years to train.

[11] https://developer.nvidia.com/nvidia-system-management-interface

References

[1] [n. d.]. Hive: The Game. Gen42. https://www.gen42.com/games/hive Accessed: 2020-04-15.
[2] Bruno Bouzy and Tristan Cazenave. 2001. Computer Go: An AI oriented survey. *Artificial Intelligence* 132, 1 (oct 2001), 39–103. https://doi.org/10.1016/s0004-3702(01)00127-8
[3] J. Burmeister and J. Wiles. 1995. The challenge of Go as a domain for AI research: a comparison between Go and chess. In *Proceedings of Third Australian and New Zealand Conference on Intelligent Information Systems. ANZIIS-95.* IEEE. https://doi.org/10.1109/anziis.1995.705737
[4] Tristan Cazenave, Yen-Chi Chen, Guan-Wei Chen, et al. 2021. Polygames: Improved zero learning. *ICGA Journal* 42, 4 (jan 2021). https://doi.org/10.3233/icg-200157
[5] R. Coulom. 2010. BayesElo. https://www.remi-coulom.fr/Bayesian-Elo/ Accessed: 2020-06-13.
[6] Danilo de Goede. 2021. *Enhancing a Hive Playing Engine with Reinforcement Learning.* B.S. thesis.
[7] Arpad E Elo. 1978. *The rating of chessplayers, past and present.* Arco Pub.
[8] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning.* PMLR, 448–456.
[9] Duncan Kampert. 2021. Creating and optimizing a game-playing program for Hive. (July 2021).
[10] Sangyeob Kim, Juhyoung Lee, Sanghoon Kang, et al. 2021. PNPU: An Energy-Efficient Deep-Neural-Network Learning Processor With Stochastic Coarse–Fine Level Weight Pruning and Adaptive Input/Output/Weight Zero Skipping. *IEEE Solid-State Circuits Letters* 4 (2021), 22–25.
[11] C. Piech. 2013. Deep Blue. Stanford University. https://stanford.edu/~cpiech/cs221/apps/deepBlue.html Accessed: 2020-04-01.
[12] Aske Plaat. 2020. Self-Play. In *Learning to Play.* Springer International Publishing, 195–232. https://doi.org/10.1007/978-3-030-59238-7_7
[13] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, et al. 2020. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature* 588, 7839 (dec 2020), 604–609. https://doi.org/10.1038/s41586-020-03051-4
[14] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. 2019. Green AI. *CoRR* abs/1907.10597 (2019). arXiv:1907.10597 http://arxiv.org/abs/1907.10597
[15] Claude E. Shannon. 1950. XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41, 314 (1950), 256–275. https://doi.org/10.1080/14786445008521796
[16] David Silver, Aja Huang, Chris J. Maddison, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (jan 2016), 484–489. https://doi.org/10.1038/nature16961
[17] David Silver, Thomas Hubert, Julian Schrittwieser, et al. 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. arXiv:1712.01815 [cs.AI]
[18] David Silver, Thomas Hubert, Julian Schrittwieser, et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (dec 2018), 1140–1144. https://doi.org/10.1126/science.aar6404
[19] David Silver, Julian Schrittwieser, Karen Simonyan, et al. 2017. *Nature* 550, 7676 (oct 2017), 354–359. https://doi.org/10.1038/nature24270
[20] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* (2017).
[21] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. 2020. *Efficient Processing of Deep Neural Networks.* Morgan and Claypool Publishers.
[22] Hui Wang, Michael Emmerich, Mike Preuss, and Aske Plaat. 2019. Hyper-parameter sweep on alphazero general. *arXiv preprint arXiv:1903.08129* (2019).
[23] Yang Wang, Yubin Qin, Leibo Liu, Shaojun Wei, and Shouyi Yin. 2021. HPPU: An Energy-Efficient Sparse DNN Training Processor with Hybrid Weight Pruning. In *2021 IEEE AICAS.* 1–4. https://doi.org/10.1109/AICAS51828.2021.9458410
[24] Yuxin Wang, Qiang Wang, Shaohuai Shi, et al. 2020. Benchmarking the Performance and Energy Efficiency of AI Accelerators for AI Training. In *2020 IEEE/ACM CCGRID.* 744–751. https://doi.org/10.1109/CCGrid49817.2020.00-15
[25] Wikipedia contributors. 2021. Hive (game) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Hive_(game)&oldid=1007191566