Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Solving Hive Board Game with Deep Reinforcement Learning

Scientific Students' Association Report

Author:

Tamás Bunth

Supervisor:

Dr. Bálint Gyires-Tóth

2019

Contents

Kivonat

Jelen dokumentum célja a Hive elnevezésű stratégiai táblajáték játszására képes mesterséges intelligencia ágens tervezését és implementálását végigkísérni. A Hive egy kétszemélyes táblajáték, melyben a játékosok különböző köveket helyezhetnek le és mozgathatnak a játék szabályainak megfelelően. A játék célja körülkeríteni az ellenfél játékosának méhkirálynőjét. A projekt célja egy olyan mély megerősítéses tanulás alapú algoritmus létrehozása, mely megközelíti a Hive játékosok teljesítményét.

A játék komplexitása hasonló a sakkéhoz. Ugyanakkor, a probléma különlegessége, hogy fix pálya híján a játéktér virtuális, elméletben végtelen méretűre nőhet – és ennek a játékban használt korongok száma szab csak határt. További kihívás, hogy az intelligens ágens tanításához csak korlátozott erőforrás áll rendelkezésre. A problémát az teszi továbbá különlegessé, hogy a dokumentum írásának pillanatában nem létezik olyan ágens a Hive játékhoz, melynek teljesítménye összemérhető lenne a legjobb játékosokéval.

A cél elérése végett szükség van egy olyan szoftveres környezet létrehozására, melyben a tanuló eljárást futtatni és tesztelni lehet. A dolgozat végigkíséri az olvasót az ágens és a környezete tervezésének és megvalósításának folyamatain, illetve betekintést ad a fejlesztést megelőző irodalomkutatás és tervezés lépéseibe.

A fejlesztés során sikerült megvalósítani egy olyan környezetet, mellyel megvalósítható egy megerősítéses tanulás alapú intelligens ágens tanítása. Továbbá sikerült megvalósítani egy olyan ágenst, melynek teljesítménye jobb egy véletlenszerű lépéseket választó ágensnél, és mindezt az ágens szakértői tudás hozzáadása nélkül, csupán a játékszabályok ismerete mellett éri el.

Megvalósítottam továbbá a manapság sztenderdnek tekinthető OpenAI Gym keretrendszer[6] interfészeit, és ezáltal lefuttathatóak az erre a keretrendszerre épülő Stable Baselines[11] algoritmusok is.

Abstract

Hive is an abstract strategy tabletop game for two players, in which the goal is to surround one of the tiles of the opponent. The players can either move one of their tiles or place a new one next to the already placed ones corresponding to the game rules. The main goal of the project is to create a deep reinforcement learning solution for playing Hive.

The complexity of the game is similar to the complexity of chess in several aspects. One difference, however, is the lack of a game board. Hive has a *virtual* game board, which can theoretically be of infinite size – the source of the only limitation is the number of pieces in play. Another challenge is to prepare the agent with the lack of outstanding computational resources. The motivation behind the task is the fact that – at the time of writing this document – no agent for Hive has been implemented which could surpass the skills of a human expert.

In order to create an intelligent AI, investigating the already existing solutions for developing intelligent agents for Hive or other similar problems is of paramount importance. Furthermore, a suitable implementation of the Hive game itself is necessary for the elaboration and learning process of the AI itself. This documentation leads through the steppes of the planning, design and implementation phases of the development, as well as the decision being taken before and during the development.

I have successfully designed and implemented an framework, which can be used as an environment for the intelligent agent. Furthermore, I implemented an agent, which could surpass the efficiency of an AI which steps randomly.

I also realized the interface of OpenAI Gym[6], which is nowadays' most famous standardization effort for Reinforcement Learning environments, and integrated a tool set called Stable Baselines[11].

Chapter 1

Introduction

In the past few years not only computational resources, but our knowledge and tooling of machine learning evolved. This allow us - among others - to create or improve computer-controlled agents for tabletop games, which can play the game on the same level or even better as a human player.

My goal is to investigate the possibilities of creating an AI - a computer controlled player based on artificial intelligence - for the board game Hive.

Hive[1] is a two-player board game, where the object is to surround the opponent's queen bee with tiles. The game consists of hexagonal tiles which can be placed next to each other. Each player has basically two type of actions: he can either place a new tile into the hive - that is how the combination of already placed tiles is named - or move an already placed tile of his own. Although the game technically does not contain any game board, the game is classified as an abstract strategy board game. That is because the rule of the game shares elements with board games like chess. The rules of hive is described more in depth in the pre-planning section. A separate section is allocated for the analysis of the game complexity regarding the number of possible states and actions during the game and the so called branching factor.

In order to create an AI to play Hive, the first task to do is to construct an environment which represents the logic and enforces the rule of the game. There are already some open-source implementations of Hive published on GitHub. I decided to use João Lopes's[2] solution, which serves as a ground base of my work. The description of the base environment as well as the extensions and modifications made on the environment during my work is explained later in the and implementation section.

There are several different approaches when we talk about computer controlled artificial intelligence creation. The most simple approach is to define a policy for each available state.

A in Hive means the current state of board, given by the relative positions of the tiles. A policy is a (state, action) pair, which defines the action the AI will perform in a state.

That means we tell the agent explicitly what to do in every situation. Of course if things were that easy, this theses work would not exists and also Hive would not be such popular game. One of the problems with that approach is that we have insufficient domain knowledge. We can't tell the AI the perfect moves, if we don't know them either. In the pre-planning section I will cover the most popular techniques which could be used to create an AI for Hive.

Some of those techniques use neural networks to approximate the optimal decision at each state. One particular area of machine learning which usually use neural networks is reinforcement learning. I explain how reinforcement learning works and how it can be used to create an AI for hive in the later sections.

Chapter 2

Background

2.1 The Hive game

Hive is a two-player board game. Each player has 11 hexagonal tiles, if they play without extensions. Each stone represents a bug. Different bugs have different moving abilities just as in chess. There are three grasshoppers, two spiders, two beetles, three ants and one queen bee. The goal is to capture the queen bee of the opposite side by surrounding it completely with six tiles. If expansions are included then players might also have a mosquito, a ladybug and a pillbug. Since they are not implemented in the environment I will not cover the expansions in detail.

Figure 2.1: An image of the game Hive, downloaded from the official website

Each player can either put a new tile to the hive or move an already placed bug. The game starts with an empty layout. One of the players may begin with placing a bug of his choice. The next player in the second turn should place a bug of his choice next to that. From the third turn on, the players can place bugs the following way:

- The end location of the piece must not be adjacent to any tiles of the opponent.

- The piece must be placed next to the hive. That means it has to be adjacent with at least one of the pieces on the same side.

The players can place pieces until the end of game or until they run out of tiles. The queen be has to be placed in one of the first four turns of each player. That means if a player has already placed three tiles, but the queen bee is not yet placed, then at the next turn he is forced to put his queen bee down. The queen bee cannot be placed in the first turn.

Movement actions are not allowed until the queen bee of that side is placed. Furthermore, there is an additional rule related to movement often referred as "one hive rule":

A piece cannot be moved, if the movement breaks the hive apart. That means if we represent the board - often referred as the hive - as a graph where the vertices represents the pieces, and there is an edge between two vertices if and only if the represented pieces are adjacent, the graph should be connected even without the piece which is under movement.

2.1.1 Movement rules

Different bugs can move differently. One simple example is the **queen bee**, which can move similarly as the king in chess: it can move to any cells adjacent to it, if the cell is not occupied yet. There is one exception though: the queen bee - and most of the other pieces as well - can only move in a "sliding movement". If a piece is surrounded to the point that it can no longer physically slide out of its position, it cannot move to that tile. This rule is often called "freedom of move rule".

Beetle can move similarly to queen bee. Although, it can also crawl up onto another piece - that means to the top of the hive. From the top of the hive it can move everywhere adjacent. Freedom of move rule is also valid to beetles too. Interesting fact that - even though the game description does not emphasize it - the "freedom of move rule" applies to the beetle as well.

The spider can move exactly three spaces around the hive. These three steps are similar to how queen bee moves. It cannot move to those cells which could be reached with less than three moves. In practice, it is rare that the spider can move to more than two different tiles. It can only happen, if there is a formation similar to a fork.

In figure 2.2 we can see an example. The spider (labeled with "S") with yellow color can move up to 4 places.

The grasshopper can jump through pieces with one movement. It can move in the six direction to a maximum of six cells. It can only move to a direction if there is an adjacent piece that way. If there is, it jumps through it to the first free cell in that direction.

The ants might be the most powerful pieces of all. They can move everywhere around the layout, without violating the one hive rule and the freedom of move rule. It can repeat the move of the queen bee as many times as you want, capable of moving everywhere on the edge of the hive.

In other worlds - in the developer's point of view - it can repeat the movement of the queen $n \in [1; \infty)$ times.

Finally, if a player cannot place a bug and cannot move any of his bugs, then he must pass the turn. The opponent can step multiple times, until the player can move again.

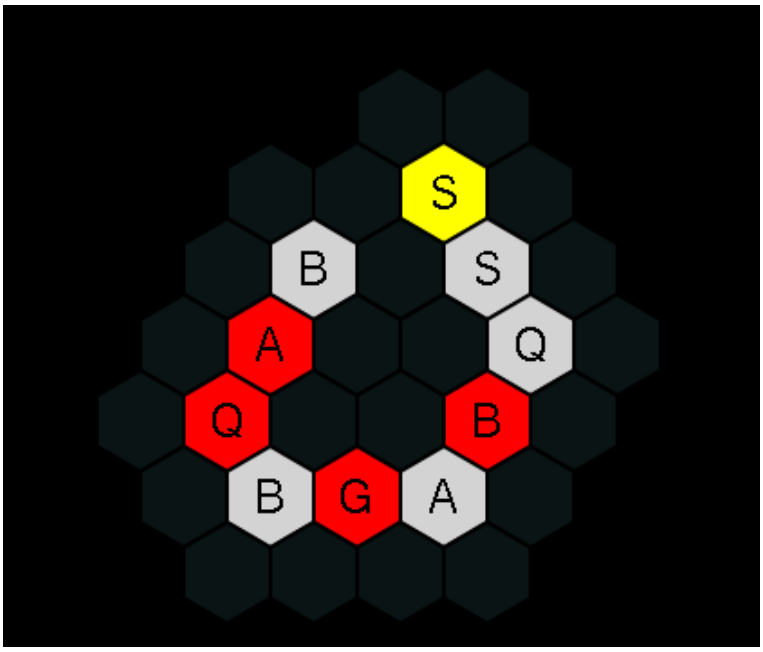

Figure 2.2: An example game state, where a spider (S) piece can move to 4 different tiles

2.2 Complexity of the game

There are common practices for measuring the complexity of chess, since that game has centuries of history behind it. Hive was published in 2001, and it is also not as well known as chess. However, the complexity of Hive can be measured around similar metrics.

There are two main factors which should be taken into account when talking about complexity: the size of the state space and the average branching factor. The state space is the set of all the possible states of the game. The branching factor means the number of possible moves at a given state. The state of the game is one particular state of the board defined with the relative position of the already placed pieces. The state space is the set of all the possible states.

Calculating the size of state space accurately is quite challenging in case of hive, since there is no game board physically. As an upper estimate we can say that each piece can have neighbours at six different sides, or they might not be placed on board yet. It can be adjacent to different kind of bugs. There are 22 pieces without expansions, which leads to $(22 * 21)^7$ possible states. We did not count that the beetle can be on top of the hive though.

There is an algorithm called *Perft*, which is developed to measure the complexity of board game chess. Since the two games are quite similar, the same algorithm can be used to measure the complexity of Hive.

The pseudo code can be found in algorithm 1.

This idea is to represent the possible states in a tree, and count the nodes on each level. That way the branching factor can be calculated between the nodes. The result of executing the above algorithm on Hive shows the following:

- The average branching factor of Hive is much higher than in chess.

Algorithm 1 Perft algorithm

```
1: procedure PERFT(Depth)
2:     move_list ← []
3:     nodes ← 0
4:     if depth = 0 then
5:         return 1
6:     n_moves ← GenerateLegalMoves(move_list)
7:     for move in move_list do
8:         MakeMove(move)
9:         nodes ← nodes + Perft(depth − 1)
10:        UndoMove(move)
11:    return nodes
```

- Since at the beginning there are more possible actions in chess than in Hive, the total number of nodes in a tree level is less in Hive.
- Since the branching factor is higher, and there are no such events as piece removal in Hive, the total number of nodes in Hive is higher after around the eights or ninth movement. There are around 10^11 possible states in depth 9.

The algorithm cannot be used to depth more than 7, because the cost of the algorithm grows exponentially with the depth of the tree.

2.3 Reinforcement learning

One of the techniques used to create AI's for games like Hive is reinforcement learning[27]. It is an area of machine learning. Consequently, it is a subset of artificial intelligence, where the algorithm builds up a mathematical model based on sample data, called as "training data", in order to make estimations on the optimal output.

In addition, reinforcement learning defines some terminologies used when modelling the system:

- The agent is the part of the system which can learn, and which interacts with the environment respectively.
- The environment is a state-full unit, where actions can be executed. The actions will be evaluated by the environment and it can be rewarded afterwards. The action may move the environment to a new state.
- Action is a decision made by the agent and executed on the environment. For example moving or placing a piece in Hive.
- State: Regarding the current state of the environment, other actions may be valid, and actions may be rewarded differently.
- Reward: The environment produces it after each action. The amount of reward may depend on the state of the environment.
- Policy: The decision-making function of the agent. It maps states to actions.

- Model: The environment from the agent's point of view. Hive has a fully observable environment, which means that every information of the state can be handed out to the agent. In that case the model clearly defines the state. A state can be modelled several ways though. For example, a state of Hive - which holds the location of the pieces - can be modelled with a two-dimensional array, where an element represents a cell on board, or it can be modelled as a graph, where the vertices are the bugs, and an edge is present between bugs, if they are adjacent. A weight on edges may represent the direction of the adjacency. Some reinforcement learning algorithms do not need any model.

Reinforcement learning is a subset of online learning[5]. In case of online learning, instead of having a full batch of training data we train the neural network only with one data at a time.

There are two subcategories of reinforcement learning in which people usually categorize the known techniques: model-based and model-free reinforcement learning. The difference between them is mainly that the model-based version stores information about the state transitions. It stores which state will be the next given an action.

The first, least smart solution for an AI would be the following: We store every possible path of the game as a tree. The vertices are the states and the edges are the actions. If we are able to store every possible state action pairs, we can see every possible outcomes and therefore we can choose the optimal solution.

Sadly, the above method does not work, since the number of possible states usually - and in Hive too - grows exponentially in the tree. Instead, we introduce a reward function and a value in connection with the state-action pairs that represents the value of the state and/or the action and we use a neural neural network to calculate that.

In case of the Hive AI I implemented a probability vector is assigned over an action called policy vector in a given state. The states are also evaluated and a scalar value is assigned to them. Theses values source from a neural network.

The main essence of reinforcement learning is to find a balance between exploration and exploitation. When exploring, the algorithm decides to chose an action in the current state which is either not yet used, or it was used but it is not the best action according to prior experiences. Exploitation means to go for the best action according to the current knowledge of the model. At first, the AI has no experience in the game yet, so it must explore. In the end, it usually exploits more than explores.

The environment of the reinforcement Learning paradigm is usually formulated as a Markov decision process, as many RL algorithms for this context utilize dynamic programming techniques. The difference between dynamic programming methods and Reinforcement Learning is that the latter does not assume knowledge of an exact mathematical model, and usually RL is used on larger Markov Decision processes, where the exact methods become infeasible. Just as in our case.

2.4 Neural networks

In the section above I described how reinforcement learning can be used to create an AI for Hive. In order to achieve that I use a neural network to predict the optimal decision of the player in a specific state. A neural network[23] is a densely connected graph in

which the edges have weights. These weights are called parameters and they are updated periodically during the learning phrase.

An artificial neural network is based on a collection of connected units or nodes called neurons. Each connection between these nodes has a weight.

There is two phases in the life-cycle of a neural network. First, we have to make it learn, where the weight of the edges are updated to solving a problem. In the second phase, we use the network on a same kind of input to predict something.

In the learning phase, the weight between the neurons are paramterized in order to be able to solve or estimate the optimal solution of a specific problem. In the second phase, data flows through the network, and – depending on the parametrized edges – it changes it value of the data.

A neural network is a mathematical model built for approximating a complex problem. Given a huge amount of inputs and expected outputs - usually referred as training data - the optimal algorithm is approximated with periodic miner changes in the mathematical mode. Neural networks are usually used if the optimal algorithm is either too complex or not known. The reason of the complexity can arise from the complexity of the input or the output, or the computational complexity of the problem.

The original goal of the neural networks was to solve problems the same way as a human brain would. Although, over time, attention moved to performing specific tasks, which resulted in a deviation from biology.

The history of neural networks go back to 1943, when Warren McCulloch and Walter Pitts [18] opened the subject by creating a computational model for neural networks. This work was inspired by the way neurons work in the human brain. After that, in 1958, Rosenblatt created the first artificial perceptron[29].

The neural network used in this implementation f_Θ is parametrised by Θ that takes the board state as an input (s), and outputs the continuous value of the board state $v_\Theta \in [-1, 1]$, where the value is interpreted as the reward for the current player to obtain that state. The neural network also outputs $\vec{p}_\Theta(s)$ policy vector, which is a stochastic policy that is handy during self-play.

2.5 Deep Learning concepts

In the following section I want to describe those functions, techniques and settings of the neural networks, which I used in my work.

2.5.1 Convolutional neural network

Convolutional neural network (*CNN*)[16] is a class of deep neural networks, which is usually applied on those areas, where the more-dimensional orientation and neighbours of the inputs holds significant information. For example analyzing visual imagery is that kind of task. It can be used in connection with the Hive AI as well, since the state representation is two-dimensional, and elements next to each other in the adjacency matrix - which is described in later sections - hold similar information.

In a convolutional layer, neurons receive input from only a restricted subarea of the previous layer. Typically the subarea is of a square shape (e.g., size 5 by 5). The input area of a neuron is called its receptive field. With these method it achieves to reduce the number

of neurons needed which results in less computational time compared to a simple fully connected layer[14].

2.5.2 Pooling

In order to reduce the dimensions of the data set I use a pooling layer[24] after the convolutional layer. The pooling layer looks at a region - a subrectangle of the input data - and aggregates it with a particular method. In my work I use max pooling, which means that the output regard that region is the maximum value in that.

2.5.3 Adam optimizer

During the learning phase of a neural network, the deep learning libraries use an optimizer to decide the method of fine-setting the learning rate and the momentum of the gradient descent method. The learning rate is used for setting the degree of impact of the calculated gradient on the actual weights in the network, whereas momentum is a regularization method mostly to prevent the vanishing gradient problem[12] and the exploding gradients.

The Adam[15] optimization is an optimization method available in the Keras deep learning library, which is an extension to stochastic gradient descent that has recently seen broader adoption for deep learning applications in computer vision and natural language processing. It makes the neural network approximating to the optimal solution faster while it is still computationally efficient.

The Adam optimization adapts the parameter learning rates based on the average first moment (the mean) and also makes use of the average of the second moments of the gradients (the uncentered variance).

2.5.4 Categorical Crossentropy

Categorical crossentropy is a loss function that is used for single label categorization. This is when only one category is applicable for each data point. In other words, an example can belong to one class only.

This can be used to determine the policy vectors in a state of the Hive board. Later sections provide more information how categorical crossentropy is used as the loss function of the neural network.

$L(y, \hat{y}) = -\sum_{j=0}^{M} \sum_{i=0}^{N} y_{i,j} \log(\hat{y}_{i,j})$

2.5.5 Softmax

Softmax is a commonly used quite simple non-linear activation function, defined by the formula:

$softmax_i(a) = \frac{\exp a_i}{\sum \exp a_i}$

It can be used to transform the output to be a probability. The result will be a probability in the sense that:

- Every element of the output will be positive and
- The sum of the elements is 1.

2.5.6 Rectified Linear Unit (ReLU)

Rectified linear unit is a commonly used non-linearity when it comes to convolutional networks[10] [17]. The function is quite simple:

$f(x) = max(0, x)$, where x is the input to the neuron. It is non-linear, since there is a breakpoint at the origo. The ReLU is the most popular activation function since 2017. The advantage is that fewer vanishing gradient problems appear compared to sigmodial activation functions, and it can also be calculated efficiently: only a comparison and addition and a multiplication is needed. A disadvantage of this activation function is that it is unbounded, which can lead to exploding gradients. Moreover, it is non-differentiable in zero. However, choosing the derivative at zero arbitrary to be 0 or 1 solves that problem.

2.6 Monte Carlo Tree Search

The Monte Carlo Tree Search[7] – often abbreviated as MCTS – is a heuristic search algorithm, which helps prioritizing possible action paths using a tree search.

Each node represents a state of the game and each Edge is an action. There are two possible decisions during traversing a Vertex: exploration and exploitation. Exploration means in that we want to try out a new node even though there are nodes in the given state which are already explored. Exploitation means that given values of nodes - which came from an exploration processed before - the best alternative is chosen in a given state. Each node should store at least three numbers: the number of times the node was visited, the number of times the game has been won from this state and also a predicted value of the state.

The operation consists of the following parts:

- Selection: given the current state of the MCTS we want to select leafs which should be expanded. The most popular method to decide which Leaf should be expanded is the so-called upper confidence bound which works the following way:

  $a_t = \arg\max_{a \in \mathcal{A}} Q(a) + \sqrt{2 \log t / N_t(a)}$

  Every step we estimate Q values using sample-average method and then we add the bonus term that only depends on the number of time steps t and the number of times we picked that action, $N_t(a)$.

- Expansion: after selecting a leaf to expand we calculate the possible actions from that date. The three should be expanded with one or more nodes.

- Simulation: in this stage we want to measure the predictable value of a given node. This is the stage where reinforcement learning has role by predicting the value of the given state using it's reward function. The reward function of the learning algorithm should be related to the outcome of the game. Winning the game should mean positive reward, losing the game should be rewarded with a negative number. The simulation part is not necessarily implemented using reinforcement learning, other significantly simpler methods can be used as well. For example a possible simulation method would be to run a complete the game with random actions performed in each state until the game is over. The benefits of using reinforcement learning instead of that method are that it takes much less time or calculation resources.

- Backpropagation: the value returned from the simulation stage should be propagated back in the tree, which means that the values off the parental nodes -should be updated accordingly. That means that the parental mode Should value more if its children represent a state in which the game can be won easily.

This algorithm becomes handy when there are too many possible state-action pairs.

2.6.1 Monte Carlo Tree Search for Policy Improvement

In this AlphaGo implementation I use Monte Carlo Tree Search (MCTS) to improve upon the policy learned by the neural network. In order to balance the exploitation with the exploration the MCTS explores the game state - represented by the nodes of the tree - through different actions, which are represented by the edges of the tree. There can be a directed edge between two different game states when there is a valid action that can cause a state transition between the two states.

For each edge we maintain a Q value assigned to a pair of $(state, action)$ which is the expected reward for taking that action in that state. The tree also stores the number of times an action is taken in a state. Both values can be used to distinguish advantageous states and/or actions from disadvantageous ones. A vector is also stored for each state, which is called *Policy Vector,* $P(s, \cdot) = \overrightarrow{p}_\Theta(s)$. It stores the prior probability of taking an action. The values in our case come from the neural network.

The MCTS can be used to find a policy from a given state s. First, we create an initial tree with only a root node s. In each iteration, we calculate action a, which maximizes the upper confidence bound. If the state after taking action a from state s (s') is already present in the MCTS, than we continue our simulation. If it does not exist, new nodes are being created and initialized with a policy vector obtained from the neural network.

After that we need to estimate the value of the new node. In order to do that, we check first whether the game has been entered to a final state. If so, than we use the result as a value (e.g. 1 meas a victory, -1 means lose, 0 is draw). If not, than we have to use the neural network to predict a value alongside the policy vector.

Now, we can propagate the value back up the MCTS tree, updating all Q values seen during the simulation.

After a few simulations performed on the MCTS, we can use the already existing tree to make predictions on actions to take.

In order to control the degree of exploration and exploitation during building up the tree, an another parameter τ, which is called *temperature* is introduced. Setting τ high means the MCTS is more likely to explore, and setting τ low reduces the probability of exploration. I change τ dynamically during the training process, lowering it in each iteration in order to encourage exploration in "early game", but enforce exploitation in "late game".

Pseudo code of the Monte Carlo Tree Search is provided in Algorithm: 2

2.7 AlphaGo Zero

During my work I implemented a subset of the AlphaGo Zero Algorithm[25], which relies mostly on reinforcement learning and uses Monte Carlo tree search. The goal here was

Algorithm 2 Monte Carlo Tree Search

procedure MCTS(s, Θ)
  if s *is terminal* **then**
    return *game_result*
  if $s \notin Tree$ **then**
    $Tree \leftarrow Tree \cup s$
    $Q(s, \cdot) \leftarrow 0$
    $N(s, \cdot) \leftarrow 0$
    $P(s, \cdot) \leftarrow \overrightarrow{p}_\Theta(s)$
    return $v_\Theta(s)$
  else
    $a \leftarrow argmax_{a' \in A} U(s, a')$
    $s' \leftarrow getNextState(s, a)$
    $v \leftarrow MCTS(s')$
    $Q(s, a) \leftarrow \frac{N(s,a) * Q(s,a) + v}{N(s,a)+1}$
    $N(s, a) \leftarrow N(s, a) + 1$
    return v

to create an artificial intelligence, which is able to compete with human in a challenging domain.

AlphaGo was the first program that defeated a world champion in the game of Go. This predecessor of the AlphaGo Zero algorithm was trained by supervised learning trained by a database of moves consists of actions taken by human experts. This algorithm also used reinforcement learning with self-play. The next generation, the AlphaGo Zero algorithm is based only on reinforcement learning, thus it can be prepared without human data, guidance, or domain knowledge beyond game rules. It uses a neural network which improves the strength of a tree search, resulting in better decisions in each iteration.

Running AlphaGo Zero against its predecessor, AlphaGo, the new generational algorithm won 100 times from 100 matches.

The new method uses a deep neural network f_Θ with parameters Θ. The network takes the board representation of Go s and outputs $(\vec{p}, v)$. $\vec{p}$ is a vector probabilities representing the probability of choosing action a given state s: $p_a = Pr(a|s)$, and v is the value of the state, which is a scalar value representing the estimated outcome of the game for the current player – a bigger value should represent a positive outcome.

The neural network in AlphaGo Zero is trained from games of self-play with the concept of reinforcement learning. In each state s, an MCTS search is executed, guided by neural network f_Θ. The MCTS search outputs probabilities π which is usually a stronger – better – probability vector for selecting action a than the one returned by the neural network. Therefore, the MCTS may be viewed as a policy improvement operator[13][27].

The MCTS uses the neural network f_Θ to guide its simulations. Each edge (s, a) in the tree stores a probability $P(s, a)$, a visit count $N(s, a)$ and an action value $Q(s, a)$. The simulation of the Monte Carlo Tree Search selects edges of the tree according to an upper confidence bound $Q(s, a) + U(s, a)$ (see section 2.6. If a leaf node is encountered during the search phase, than it is expanded and evaluated by the neural network to generate $p(s, a)$. After that, each edge is traversed in the simulation is updated by incrementing $N(s, a)$ and updating its action value $Q(s, a) = 1/N(s, a) \sum_{s'|s,a \to s'} V(s')$, where s' is the resulting state when performing action a in state s.

They applied to above algorithm to train AlphaGo Zero. The training started from completely random behavior and was under execution for three days. Over this time, it made 4.9 million of self-play, using 1,600 simulations for each MCTS, which corresponds of approximately 0.4s thinking per move.

Chapter 3

Software Environment

3.1 OpenAI Gym

Apart from the AlphaGo Zero implementation, I also wanted to elaborate baseline solutions so that I would have some results to compare the AlphaGo Zero implementation with.

OpenAI Gym[6] is a standardization of environments which are designed to be environments of reinforcement learning methods.

As such, the OpenAI Gym standard defines a class interface called *Env*. *Env* has one of the most important methods: *step()*, which takes action a – an arbitrary, yet distinguishable Python object, and returns a tuple of (obs, r, d, i), where *obs* is a Python object representing the next state, r is the reward for taking action a in the current state of Env, d is a boolean whether it is the end of the current episode or not (i.e the end of the game), and i stands for some arbitrary debug information passed in a dictionary.

The above function $step()$ is the only way to change the state of the environment, except $reset()$ which resets the state.

Apart from the above methods, you can also query information about the environment, e.g. the size of the action- and the state space, a human readable rendering of the current state and a lot more.

The goal of this project is to standardize the interface of different environments so that various reinforcement learning methods can be used on the same environment.

3.2 Stable Baselines

Stable Baselines[11] is a set of Reinforcement Learning (RL) algorithms written above OpenAI Gym. You can choose from several RL algorithms including variations of the Actor Critic methods [20] as well as a simple Deep Q Network implementation[19].

In case of this project, Stable Baselines can be used to try different RL methods without the need of implementing them.

3.3 Choosing the language and the tool set

Before designing or implementing the desired functionality, the first question that comes up is choosing the proper programming language and the tools that suits our needs. In my case, the following qualities were considered:

- **Performance**: The implementation should be fast enough, so the learning process would not take too long.
- **Tooling**: I need maintained and easily available libraries and tools to avoid reinventing the wheel.
- **Productivity**: I should be able to progress forward in reasonable time.

I decided to use Python as the main programming language. Even though, one might choose a low level language like C++ in order to achieve better performance, for my task it does not worth it. Python is a much better choice mainly because the available libraries. There are several Deep Learning frameworks maintained like Keras[8], PyTorch[21] and Tensorflow[4]. Moreover, it has a great way to handle mathematical operation on array with the help of Numpy.

Taking productivity into regard, Python performs quite good as well.

3.4 Hive implementations

Before I could make an AI for solving Hive, first I had to create an environment where the AI can learn and where it can be tested. Instead of implementing the whole game, using and extending an available open-source implementation is sufficient. I choose João Lopes's solution on GitHub[2], because of the following reasons:

- It is written in Python. Although it was originally Python 2.7, after upgrading to Python 3, it can be used alongside any Python machine learning framework.
- The source code is simple enough to understand easily. Also, it is easily extendible, and it does not contain unnecessary parts. For example a graphical interface would only just increase the complexity of the source base, while it is not usable for an computer controlled AI. This implementation has no GUI, only a command line output, which is far enough for testing.
- The implementation already has some tests which ensures the functionality of the rule validation. That is a key point before creating an AI. Failures during move validation could result in a significantly different solution. For example, the lack of "one Hive rule" validation would result in significantly more possible states in the environment.

The source does not contain any computer controlled AI, which means I have to implement that myself. The program allow us to play Hive as human, using the command line. Therefore, I also have to extend the capabilities of the game logic, in order to provide those information about the environment, which turns out to be useful for an AI. These are the following functionalities:

- The environment should be capable of providing all the possible actions in a given state. Given that the AI can explore the actions space.
- The environment should provide a unique identifier for each state. That way the agent will know which state he is in. Identical states should be filtered out. For example, if we rotate the game board with 90 degrees, the result state is still valid, and is identical with the original board. If there were any bugs, which movement would depend on the actual direction, e.g. it could not move north, that the rotated board would not be identical, but currently there are no expansions where a bug like that exists.
- It might be also useful, if the environment could provide a state description, from where domain logic could be used. For example, the task of the AI could be simplified with saying that the less bugs surround the queen bee, the better. Those kind of domain knowledge may lead to bad decisions on the other hand. A state description could be for example an adjacency matrix, which describes the neighbours of each piece.

The base source code consists of the following modules:

- Board Module: This module is responsible for providing a virtual board. The board is stored in memory as a two dimensional array. The first bug placement will define the (0,0) position. Figure 3.1 shows how the isometric space is mapped to a two dimensional array. The module also provides some minor validation logic related to the board. For example it can be queried if two pieces are in the same line. This is useful when validating the move of the grasshoppers.

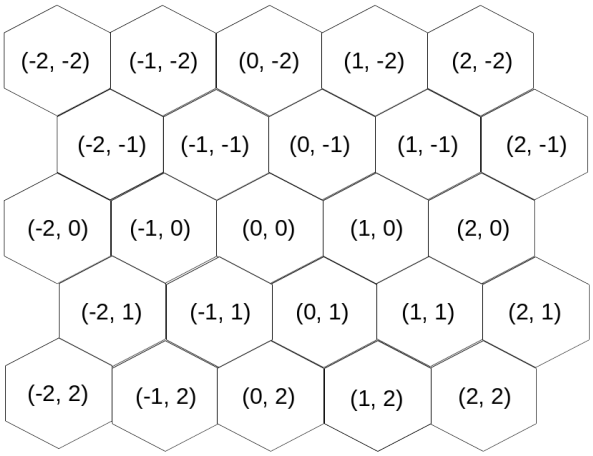

Figure 3.1: Representation of the hexagonal layout in two dimensional space

- Hive Module: This module contains the logic of the game. It consists of the piece placing validation, general rules like one hive rule, and other special rules. It also stores the state of the game. The location of the pieces are tracked and stored. Unfortunately, it also contained the logic of the move validation for each bug type. I moved that logic to a separate module later.

- View Module: It contains a simple class which can be used to show the current state of the environment on the command line. It can draw a pretty ascii figure of hexagons, which helps the human player to see and understand the actual layout.
- Main Module: The is the entry point of the program. It contains the logic of creating and configuring the game. Also, it interacts with the human players.

3.5 Testing

In order to manage test suites efficiently I decided to use the nose Python test utility[3]. Nose can be used to collect tests defined in the project and run it at once, or manage which tests should be run and collect the results.

Chapter 4

System Design

4.1 Inner representation of the board

I worked with two different board representations during the project. My first intention was to use the structure of Jclopes' solution the way it is described in section 3.4.

Although, the original solution was not flexible enough to serve as a software environment for the intelligent agent. It was much harder to interpret log files, because the indexes used for identifying a hexagon was less intuitive. Moreover, it was harder to validate the movement of pieces, especially the movement of the grasshopper.

So, I came up with another solution. The advantages and the description of the new representation is described in the following sections in detail.

4.1.1 The previous representation

The previous representation of the board which is used for validation of moves can be found in class *Board*. This object keeps track of the played pieces in a dynamically extendible two-dimensional array. The hexagonal map is mapped to this matrix using the following method:

- In the first turn, the two-dimensional array is created and the first bug is placed to the $(0, 0)$ position.
- After the first bug is placed, the next bugs should be placed next to the piece already placed. The matrix will be expanded the way I described in section 3.4.
- When a piece is moved from its previous position, the engine won't remove the empty tiles from the array. It is more efficient to just leave them there.

This representation could not be used to feed the neural network of the AI, because theoretically an infinite amount of identical states can be described with different arrays. For example if I push the whole board to one of the six directions, the resulting state would be identical. A possible solution to that would be to normalize the indexes after each turn. For example the top-left piece would be at position $(0, 0)$, no matter what. The problem with that approach is that the in case the board is expanded to the top-left corner, all of the pieces indexed under a different index as before. In the neural network's

point of view it would mean that every piece has a new place now, so the previously learnt correlations do not fit any more.

Still, this representation is quite useful for validating the bug placements and movements.

4.1.2 The novel representation

The above structure has the following disadvantages:

- Having a dynamically extendable two-dimensional array is impractical. Different boards are really hard to compare, and the solution is overly complex. In order to appends tiles to the board, new rows or columns have to be inserted to the array.
- The indexing structure showed in section 3.4 is error prone. It is hard to decide if two tiles are in the same line - which means stepping into one of the six directions n times we may reach one tile from the other. This property is needed for example during the validation of a grasshopper movement.

In order to avoid the problems above, I decided to store the state of the game in a simple dictionary instead. The key of the dictionary is a cell position, and the value is a list of bugs, which are placed on it. The cell positions are represented with an immutable object type with two indexes. In python, an immutable type can be defined for example by inheriting from a so called *namedtuple* type. Named tuples have the same nature as simple tuples, but they can be referred and distinguished with a name.

After creating the immutable hexagon type, which I named *Hex*, I paired a bunch of methods to the new class, which are relevant to the cells. For example the method *neighbours* can be called to get a list of the adjacent hexagons.

Furthermore, instead of the original indexing structure, I introduced a new representation, which is visualised in figure 4.1.

As you can see, there are coordinate pairs in the representation, which are not valid, e.g: (1, 0). More precisely, a hexagon tuple is valid if and only if the sum of the indexes is even.

Depending on the direction, the indexes change with the amount shown in table 4.1.

Direction	**Δ Indexes**
Northwest	(-1, -1)
West	(-2, 0)
Southwest	(-1, 1)
Southeast	(1, 1)
East	(2, 0)
Northeast	(1, -1)

Table 4.1: Unit value of each direction.

This structure has the following advantages:

- There is a linear correlation between the indexes of the hexagons in the same line. Because of that, the validation whether two hexes are in the same line becomes easy:

if the following equation applies to any direction's unit value (nb), then they are in the same line, otherwise not:

$(x1 - x2, y1 - y2) \equiv (0, 0) \mod (nbx, nby)$

- The indexes are proportional to the actual distance of the hexagons in a two-dimensional coordinate system. It comes handy when implementing a graphical user interface of the game.

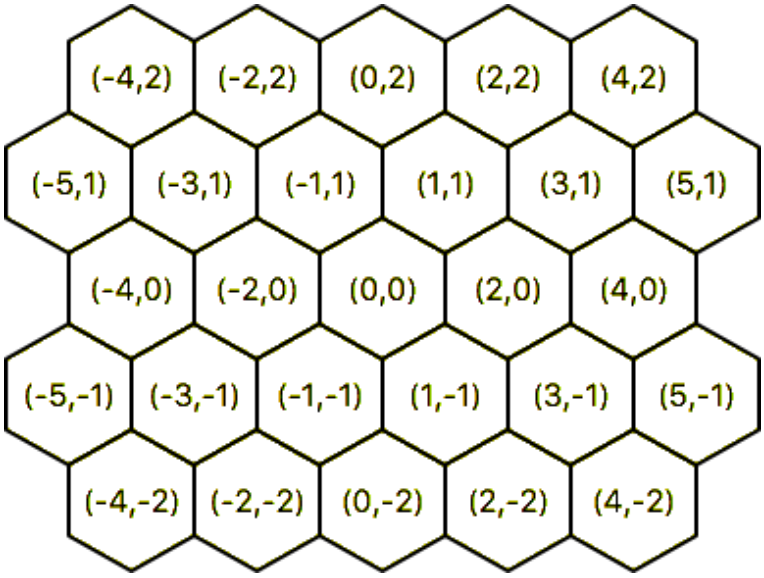

Figure 4.1: New indexing structure

Having only a dictionary for storing pieces on board, queries for piece locations become faster. From now on, I don't have to store the position of the bugs inside the class which represents it, because I can look up quickly in the dictionary. Instead, the python type representing the bugs can also be a stateless immutable type.

4.2 Matrix representation of state and action space

In this work I use a fixed size action and state space, as varying size state and action space would make the problem even more complex. The state space can be implemented using the adjacency which I described in detail in the above sections.

The action space is a bit trickier. The possible actions can be described with the vector which consists of three parts:

- Piece placement: when you want to achieve an action space with fixed size a list of possible actions cannot be used, I need a vector instead. A brutal approximation of the problem can be given the following way. I can place each pieces next to each other, which ends up in 22 x 21 possible actions. Most of these actions will be invalid though. A piece cannot be put next to itself, obviously.
- Piece movement: for some of the pieces a fixed size movement vector is no problem. The Bee, the beetle and the Grasshopper can perform 6 steps at most. The number off possible movements that a spider or an ant can perform is depending on the state of the board in a pretty complex way. Because of that I decided to limit the possible

movement count of the spider to 15 and the ant to 50. That means that I assume these pieces can step to no more than 15, respectively 50 steps. If it is not so, than the algorithm will just ignore the rest of the actions.

- Initial movement: the first 2 turns have to be distinguished from the other piece placements, because it requires another way to identify it. General piece placements are identified using the neighbour where the piece should be put down. In the first two turns there are no such pieces. That means that I need an action bit for each possible initial movements as well. Since I cannot put the Bee Queen down in the first turn there are 10 such movements.

Figure 4.2 shows how the action numbers are mapped to place piecement actions. As you can see, the number in the table are incremented by 6 each time. Those six numbers means the placement of the same piece next to the same adjacent neighbor, but a different direction. If the number is shown in the table, it means a placement to the west side of the adjacent piece. After that the directions go clockwise with the number increasing. This is a arbitrary decision, there is no gain in going counter-clockwise or starting counting from another side.

Neighbor	wA1	wA2	wA3	wB1	wB2	wG1	wG2	wG3	wS1	wS2	wQ1
wA1		10	16	22	28	34	40	46	52	58	64
wA2	70		76	82	88	94	100	106	112	118	124
wA3	130	136		142	148	154	160	166	172	178	184
wB1	190	196	202		208	214	220	226	232	238	244
wB2	250	256	262	268		274	280	286	292	298	304
wG1	310	316	322	328	334		340	346	352	358	364
wG2	370	376	382	388	394	400		406	412	418	424
wG3	430	436	442	448	454	460	466		472	478	484
wS1	490	496	502	508	514	520	526	532		538	544
wS2	550	556	562	568	574	580	586	592	598		604
wQ1	610	616	622	628	634	640	646	652	658	664	

Figure 4.2: Decode piece placement action numbers. For example, the action number 376 means that piece $wA2$ shuld be placed west from piece $wG2$.

4.3 Random search for benchmark

After providing a method for obtaining state information and all the possible moves from the Hive component, I am ready to create my first search algorithm, which is random search. Random search can be surprisingly effective in optimization problems, thus, it can be used as a benchmark.

The method is simple: it takes all the possible actions, then chooses one randomly and executes it on the environment.

This functionality allows us to make some powerful testing, and it also creates a way for complexity measurement. Having two random AI of that kind playing against each other for even a shorter period of time can reveal failures in the Hive implementation. Because of that, I created a test suite where two random search AI's play for 10 seconds.

Interesting and promising detail, that the average number of wins during that 10 seconds is around 1 win. That means that even with stepping randomly the game can be finished in reasonable time.

4.4 Interface for the search tree

The implementation of the search tree uses an interface of the game engine, which provides a stateless game representation. For example a step can be executed with the *getNextState*

method, which has a state and an action input, an returns the next state. The advantage of the stateless representation is that the game can be continued in an arbitrary state. The interface also provides a way to query the canonical representation of a state. The canonical representation is a person-of-view model of the state which is implemented by a side flipping logic. That way each player can see his and the opponents pieces in a canonical way. In case of the white player there is nothing to do, the view is already canonical. In case of the black player I have to switch the color of each pieces already played. Practically - since it is performed on an adjacency matrix - it means the columns of the matrix have to be set in a different order. Luckily it does not need much calculation time.

The interface also provides a hashable representation of the state. This is currently implemented via converting the adjacency matrix into a string by concatenating every elements. After running a performance analysis on the learning process it turns out to be one of the most expensive methods. Because of that I am planning to change the implementation to a more effective one. The hashable representation is needed by the search tree which stores the states in a hash map.

An other important feature of the interface - which makes the search tree much more effective - is the filtering of identical states. In order to avoid storing the same state as different ones in the tree, I have to identify the logically identical ones. Since the board is hexagonal, I can rotate the whole board with 60 degree 6 times. If one of them is already in the tree, then those states should be handled as identical states. Luckily the rotation on the adjacency matrix can be performed with not much computational power. Since the cells store the direction of the neighbourhood between two pieces, all I have to do is performing an addition on each element consequently. The Hive board is invariant to rotation and reflection.

As I mentioned earlier, the result of a step on a given state can be calculated calling the *getNextState* method. The action given as input is a number, which precisely identifies which piece should be placed where. It is possible due to the fixed size action space. The number can be interpreted as an index which refers to an action in the action space.

The whole action space can also be queried with the *getValidMoves* method. This method returns a binary vector. There is a 1 in each positions with a valid action and 0 if that move cannot be executed in the give state of the board.

Finally, the interface provides information about the end of the game. If the game is ended, it can also tell which player won the game. This is a mandatory feature of the interface since the current player can win or even lose the game in his turn. It is possible in Hive to make a suicide move and surround your own bee. The game ends up in draw if a move surrounds both his own and the opponents bee piece in.

4.5 Graphical User Interface

The following section is going to describe the structure of the graphical interface (GUI) implementation.

In order to implement a GUI, I had to choose a framework, with which polygons and simple structures can be created effortlessly. I picked the Qt project[26], because it is platform independent and easy to use.

The GUI uses the same representation as the move validation part of the code, which is a simple dictionary, where the keys are hexagon positions, and the values are lists of bugs.

The GUI has the following responsibilities:

- Show the state of the game on the window. For each occupied hexagons the bug on the top is shown. As an improvement in the future, it should also show the bugs under the top one somehow.
- Show those hexagons as well which are not occupied, but the neighbors of one or more occupied cells. This is important, because the user can perform actions on those hexagons - he can move or place a bug there.
- Read and perform user action on the GUI. You can see the detailed list of user actions below.

The following user actions - use cases - are implemented:

- Selecting a piece: It can be performed with a left click on the appropriate hexagon. Only the hexagons with the current player's bug on top can be selected.
- Moving a piece: It can be done only if a hexagon was selected beforehand. Left clicking on the target cell, the piece on top of the selected hexagon will be moved if the movement is valid.
- Placing a piece: Right click on a border cell - cell which has at least one occupied neighbour - should open a drop-down menu, where the desired piece can be selected. The menu item should appear only if there is a piece of that kind which is not jet played. When clicking on one of the menu items, the placement is performed. The placement should not be performed when it violates the rule of the game - e.g. when there is an adjacent piece of the opponent's color.
- Drag camera: The user can move around the map to see other parts if the board is too huge by pressing and holding the left mouse button and drag the mouse. If the mouse button is being hold, no selection should be performed.

The map is repainted whenever a user action is performed or when an action from the AI is taken. Because of the camera dragging feature, an (x, y) coordinate-pair has to be introduced. This represents the *center of view (COV)*. Each hexagon should be shifted with COV when painting out the hexagons. Each player has a separate color - I picked white and red, because they are easily distinguished from each other and from the black background.

The different type of pieces should be also distinguished. To achieve that I picked an easy and quick solution: the beginning letter of the piece type is painted in the center of the hexagon (see figure 4.3). The letters are ordered to the pieces according to table 4.2.

Piece	**Letter in GUI**
Ant	A
Beetle	B
Grasshopper	G
Spider	S
Queen	Q

Table 4.2: Distinguish different type of pieces in GUI

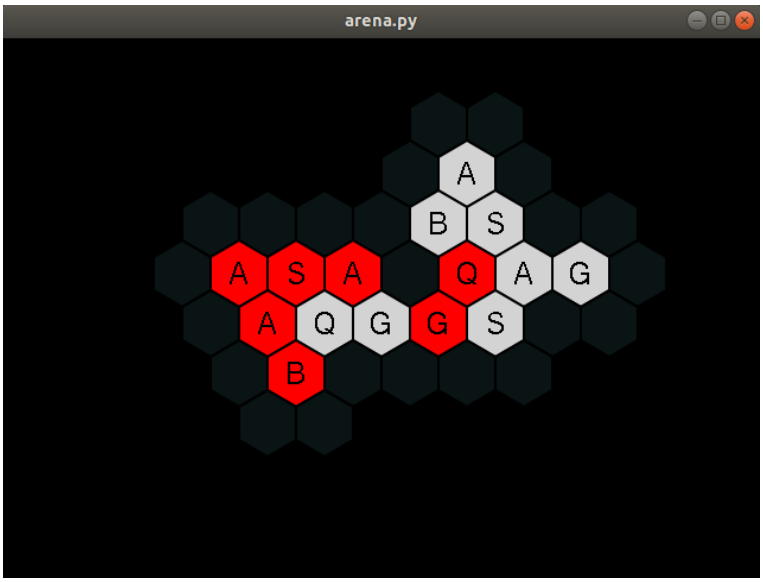

Figure 4.3: A screenshot of a hive match played on the graphical interface

4.6 Move validation based on Hive rules

The move validation logic can be found in the *Hive Validation Module* and in the files which contain logic about individual bugs. The following rules are considered while playing:

- One-Hive rule. This is one of the most important rules during the game play. The rule restricts those movements which would result in having the board in two pieces. Those moves are restricted too which leads to breaking the hive during the move. So if I remove a piece temporary and put it somewhere else, I have to make sure that the intermediate state is correct as well. The implementation of this rule is quite simple: before moving the piece to its new position the validator removes it temporarily. After that it checks if the hive is still in one piece.

  In order to check if the hive did not break, the validator collects the surroundings of the removed piece. After that it tries to reach all of them from one of them. If it is successful, than the hive did not break and the function returns true. False otherwise.

- The queen bee cannot be put in the first turn. Checking it is trivial.

- The queen bee should be put down until the fifth turn of a player. It is checked by counting the already placed pieces of the same color. It the queen bee is already set there is nothing to do.

- The player should move only his own colors. It is easy to implement, but missing this rule would have big consequences,

- Player can place a piece only to those positions which has no surroundings of the opposite side. The one-hive rule should be forced here too.

  In order to achieve it, I have to check the surrounding tiles of the candidate cell. No surprising concepts here.

- A piece cannot move to the same place it was standing. Even though saying this rule explicitly might be a bit redundant, since the individual move rules of the pieces also restrict it, it is wise to handle it separately for more robustness.

Apart from the rules described above, there are specific movement rules for each pieces types. There are two different approaches when it comes to validating the movements, which might be more efficient to implement separately: I can ask for a candidate movement if it is valid or not, or I can ask for all the valid movements in current state. The second approach could be implemented using the first one on every possible tile, but that would not be efficient, so I created a separate algorithm for each bug. The movement rules are the following:

- Moving the bee piece: In order to decide if the bee can move to place X the following conditions should be met:
  - X should be adjacent to the position of the bee (P).
  - X should be free - there are no bugs on it.
  - A piece exists which is adjacent to X and P.
  - There is a free cell which is adjacent to X and P.

  The above algorithm can be used to decide if a candidate action is valid or not. In order to get all the possible actions, I can simply run the above algorithm on all the surrounding tiles.
- Beetle piece: The beetle can certainly move to those places where the bee can. Additionally, it can move on top of every bugs surrounding it. Obtaining all the possible actions can be performed the same way as the queen bee.
- The spider: Validating the movement of the spider is a bit tricky: I perform three queen bee moves (sub-movements). After the first two moves the reachable tiles should be stored. It if forbidden to step on these tiles in the next sub-movements. The available tiles of the third move are the result. The validation as well as the query of all the possible movements work that way.
- Grasshopper: In order to validate the movement the following conditions should be met:
  - The candidate cell (X and the position of the grasshopper (G) should be in one line on the hexagonal map. It can be tested using the *get_line_dir* function of the Board object.
  - Check if X and G are adjacent. If so, than its an invalid movement.
  - Otherwise I have to check is there are any free cells between them. If yes, it's invalid, else it's good.

  Obtaining all the possible actions works a bit differently. I check the surrounding tiles of G. If it is free, than invalid. Otherwise the movement is certainly valid, so I can simply put a 1 there. The interpreter will deal with determining the exact position of the move.
- Moving the ant: I have to perform bee moves repeatedly until there are no more tiles to go.

Algorithm 3 Check whether moving to *target* from *pos* is available

```
1: procedure AVAILABLE_MOVES(pos, target)
2:     if check_blocked(pos) then
3:         return False
4:     Remove piece temporarly
5:     to_explore ← {pos}
6:     visited ← {pos}
7:     result ← False
8:     while to_explore not empty do
9:         found ← {}
10:        for cinto_explore do
11:            to_explore ← to_explore ∪ bee_moves(c)
12:        found ← found \ visited
13:        if target ∈ found then
14:            result ← True
15:            break
16:        visited ← visited ∪ found
17:        to_explore ← found
18:    Put back piece
19:    return result
```

As an example, pseudo code 3 shows how the available target cells of the and piece is resolved:

In order to test whether the ant piece can move to the target location, we should check the state in the current position first. If there is a beetle siting on the ant piece, than we should return *False* no matter what. After that, let's just remove the ant temporarly. Let's do a step according to the rules of the queen piece. Store the available tiles – if not stored already – and explore the neighbor tiles recursively (the algorithm itself is not recursive, but the logic is easier to explain that way).

If the target tile is an element of the available cells, than return true. We can exit the loop in that case. Otherwise, we should continue to expand the available cells until the target cell is an element of the result, or there are no more tiles to add to the *to_explore* set.

4.7 The neural network

The neural network is implemented using the *Keras*[9] Deep Learning library. The network itself is defined in *Hive Net Module*. It can be used through an interface which provides the following methods:

- The *train* method can be used during the learning period. This function trains the neural network with examples obtained from self-play. It requires a list of training examples of a triple (board, pi, v), where board is the state representation, pi is the policy vector for the given board, v is its value. The board is in canonical form.
- The *predict* function can be used to make a prediction with the current weights of the neural network for a given board. It returns a policy vector for each action and a predicted value of the state.

- *save_checkpoint* and *load_checkpoint* methods are responsible for saving and loading the weights. Once the network is trained I can save it for later use.

In order to be able to use the network through the above described interface there is a wrapper class called NNet which implements the interface. It converts the inputs to a format which is suitable for training in Keras. The neural network needs numpy arrays of a particular shape.

Chapter 5

Implementation

Before I can use reinforcement learning to solve Hive, I have to extend the functionality of the game implementation, so one can attach an AI to it. Also I need to obtain information from the environment which at that point were not available. I started the work with a mayor refactoring.

5.1 Refactor and maintain

First, I made an upgraded the source of João Lopes's solution[2] from python 2.7 to python 3. I used the 2to3 tool[22] for that. After the conversion one of the grasshopper tests failed. After some investigation I found out that it was because of an integer division which was not upgraded to python 3. While in python 2.7 the '/' division operator can be used both to float and integer division - depending on the operands - in case of python 3 that kind of division is always interpreted as float point division. One has to say integer division explicitly with the '//' operator.

After that I started to refactor the architecture of the program a bit. I introduced the following modules:

- Environment Module: This module consists of a wrapper class for hive. It publishes only those methods of the hive module, which is needed for the controller to manage the game. It also publishes information for the AI's.
- Piece Module: This module contains an abstract class for pieces. Every kind of piece should provide information about how they can move, and what are the possible moves according to its specific rule. This class is overriden by a class representing the queen, beetles, grasshoppers, ants and spiders respectively.
- Arena Module: This module will be responsible to manage AI's and human players. It can reset the environment on demand.

After creating the above mentioned new modules, I could ship some of the functionality from the Hive class to other modules. This results in a easier maintainable code. Now I could extend the functionality of the Hive class to provide more information about the current state.

I also wanted to separate the logic which provides information to the learning modules from the stateful Hive game engine. The validation logic could also be separated from the original Hive class. The result is the following files:

- Hive Representation Module: A functional module without any classes which is responsible for converting state and action representations. For example, the neural network of the proposed solution requires a binary vector as action space and a two-dimensional representation of the state, which is an adjacency matrix in my case.
- Hive Validation Module: This files contains the function which should be called in order to decide if the action being taken is valid or not. The module works with the inner representation, so it needs a Hive object as parameter in order to perform the validation.
- Hive Module: The rest of the logic comes here. It uses a Board object to keep track of the game state. It also knows which player's turn is the next, it can be used to load or save state.

5.2 Graphical User Interface

First thing to do was to create a class responsible for all the graphics related functionality. I call it *GameWidget*. As a subclass of QtWidget, it can be shown in a new window with simply calling the *show()* method. Without any further development, the window would be a blank window.

In the constructor of the new Qt widget contains a bunch of property initialization. It initializes the background of the window, the default font style and size to use. The hexagon, which represents the *Point of View* (PoV) is also instantiated to the $(0, 0)$ hexagon.

Next, in order to make the interface interactive, I have to catch and handle mouse event. The following methods of *QtWidget* have to be overriden:

- **mousePressEvent**: This callback is triggered when a mouse button is pressed. The mouse button types can be distinguished inside the function body, using the *event* parameter of the callback.

  If it is a right mouse button, than I have to display the menu of available pieces. The information whether the hexagon, which was being clicked, has my piece on top can be queried from the game representation. The menu should appear only when the hexagon mentioned above has the current player's piece on top.

  In case of a left click, it can be three different type of behaviors. It can either mean the cancellation of the current hexagon selection. It happens if the user clicked on a blank space. It can also be and action execution, if and only if there is a hexagon already selected and the target hexagon – the hexagon under cursor – is a border of the hive. The third possible action is the selection of a piece.
- **mouseMoveEvent**: If the cursor moves when the left mouse button is held, than the *PoV* has to be changed. This can be easily done by modifying the corresponding parameter.
- **mouseReleaseEvent**: This method is useful when I want to decide whether the left mouse button is held or not.

If the user accomplished something, that changes the state of the game or the selection of the hexagons, the whole GUI has to be repainted. It can be done with calling the *repaint*() method.

5.2.1 Future improvements

Even though the GUI is fully functional, There are a bunch of future opportunities to make it more convenient:

- When dragging the camera the widget is repainted each time the cursor moves. It takes a lot of computational resources. It would be better if the repaint would occur more rarely.
- After selecting a hexagon it would be nice to have the available possible target hexagons highlighted.
- Textual explanation should appear when the user tries to make an invalid movement. It should explain that the move is invalid. It would be nice if there would be an explanation as well, e.g. "Invalid movement. One-hive rule is violated".
- Victory is not yet handled in the GUI. In case of victory, at least a message should appear that the game is over.

5.3 Introducing Reinforcement Learning Capablilty

One thing that the computer controlled players will possibly need is a state representation. There are two basic requirements related to the state representation:

- The representation should be fixed size. This is important because most of the standardized RL algorithms - for example the algorithms of the Stable Baselines framework[6] - can work only with fix sized inputs.
- The representation should be related to the reward function as much as possible. This requirement helps finding the optimal solution for RL algorithms, because then the neural network can learn easier the correlation between the actual state and the upcoming rewards of the next actions.

Taking all these into consideration I came up with the following possible state representations:

- A two dimensional array with fixed size. Each element of that array holds a location, where one ore more pieces can be located. Since physically there are no actual board in case of Hive, I have to store than a two dimensional array of the maximal size of the virtual board. Worst case I can have all the 22 pieces - without extension - in one line, which leads to a table of size (22, 22).

  That is fine, but in that case I have to reassign the centre position sometimes. Let's consider the following situation: I have a piece at the (0, 0) position. After that I put all the pieces left from that. That way our pieces will be located from (-21, 0) to (0, 0). If I had put the pieces to the right side of the centre piece, then I would have got an interval of (0, 0) to (21, 0). That means I either reassign the centre position or I need a array of size (43, 43).

  The problem with the reassignment of the centre piece is that from the AI's point of view is seems that all of the piece moved at once. That may or may not be a problem.

- Graph representation. I can represent the actual state with a graph, where vertices holds the pieces, and edges between two pieces mean that they are adjacent. The label - or weight - of the edges represents the direction, in which the pieces are adjacent.

After careful considerations I choose to use the graph representation. I store the graph in memory as and adjacency matrix. The adjacency matrix is a way to store graphs in memory, where the graph is mapped to a two-dimensional matrix. The two axes represents the nodes of the graph. There is an edge between two nodes if the cell is 1. There are no edges otherwise.

One reason to use adjacency matrix is that it is more related to the reward function. As a first approach the player gets reward +1, if he wins, 0 otherwise. I can be sure that this case the reward function models my requirements correctly. In that case, I get a +1 reward if and only if the agent chooses an action which results in surrounding the opponents queen bee completely. The number of the pieces adjacent to the queen be can be read from the adjacency matrix easily. If the row of the queen be contains no 0 columns - where zero means that there are no adjacent pieces in that direction - then the game is over.

However, I intend to keep the original internal representation of the state as it is - which is a dynamically expandable two dimensional array with fixed centre. This representation is useful for action validation.

Another thing which is needed in order to successfully create a useful environment for AI's is the collection of all the possible actions in a given state. Since human players do not depend on those information, this feature was not yet implemented. Algorithm 4 shows the general idea.

Algorithm 4 Collecting possible actions

```
1:  procedure GET ALL ACTIONS(hive)
2:      result ← []
3:      played_pieces ← hive.played
4:      for piece inplayed_pieces do
5:          s ← surroundings of piece
6:          for cell in s do
7:              if check_adjacent_friendly(cell) then
8:                  result ← (unplayed pieces, cell)
9:      for piece in played_pieces do
10:         target_cells ← get_possible_moves(piece)
11:         for cell in target_cells do
12:             if not validate_general_rules(cell) then
13:                 remove(cell, target_cells)
14:         result ← (piece, target_cells)
```

Some basic information of the pieces like their location can be obtained from the Hive module. I would like to categorize the possible actions into two category:

- Piece placement: The hard task here is to obtain all the locations where pieces can be played. After that - aside from the queen rules - I can place there a piece of any kind. So I iterate over all the already played pieces and look at their neighbours. If those cells have no neighbours of the opposite side, then I can place a piece there.

- Piece movement: I iterate over each already played piece. Those pieces are represented as objects, and they can be queried for all they possible actions. That's what the function *get_possible_moves()* do. After that I make a bunch of validations on those candidate cells. I have to validate the queen specific rules, the one hive rule, and so on.

  The concatenation of the above defined two result sets will cover all our possible moves in a given state. The above pseudo code does not deal with some details, e.g. it does not handle beetles on the top of the hive.

5.4 Load and save states

The learning algorithm I implemented requires the possibility of loading a state of any kind and continue playing from there. However, the initial engine did not support that.

The input of the state loading is a state represented by an adjacency matrix. The current player should be given as well, because that cannot be determined by state sometimes. The number of already performed actions cannot be determined from this input, but it is not a problem, until it does not affect the validation of the subsequent possible moves.

There are some cases, where the turn number affects the game, but the turn number can be guessed in all that cases. For example, the bee queen cannot be placed in the first two turn. The turn number is equals the number of already placed pieces plus one in that case. Another related rules is that the bee queen should be placed at last in the seventh and eighth turn. That can be translated to a rule which is independent from the turn number. As shown in figure 5.1, the needed information can be derived from the number of already played pieces.

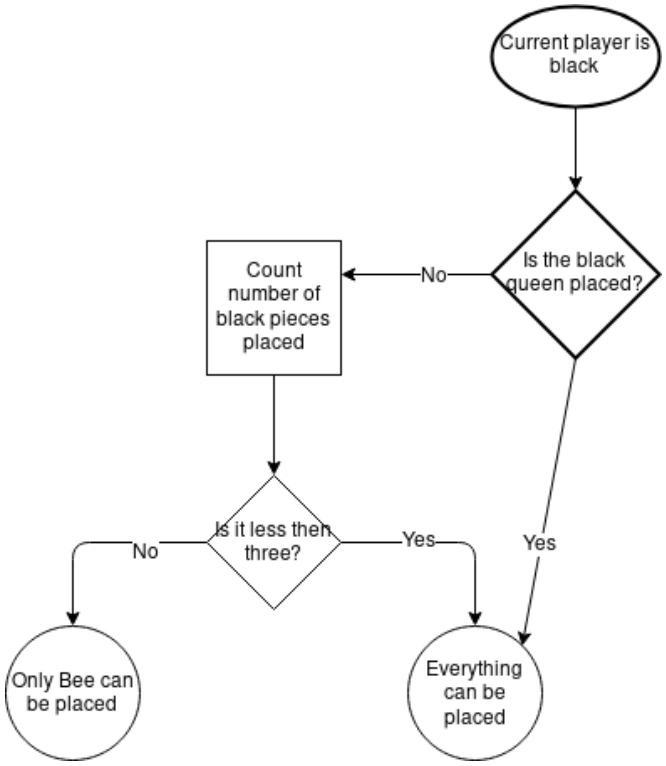

Figure 5.1: Deciding whether the bee piece should be placed on the next turn

The logic which performs the loading of the state uses the adjacency matrix to create a two-dimensional array, where the pieces are set. The state will be stored in that representation, which allows the engine to validate the next moves. The loading is executed similarly as an actual game-play, the pieces are being placed one-by-one. However, during that phase no step validation is performed. The validation would fail, since the actual state is the result of bug placements and movements as well, which ends up in a state that could not be achieved by only bug placements in a normal game-play.

Saving the game means practically a conversion from the inner representation to an adjacency matrix. This is achieved the following way:

- Initialize a matrix width height 22 and width 21. All the elements are initialized to 0. In the end-result 0 will mean that the bug which is represented by the current column is not yet placed. That means a matrix with a column which contains a zero value, but not every cells are zero is invalid.

  Apart from the zero value and the directions, there are other special values as well. Numbers from 1 to 6 represents the directions. 7 means the piece *under* the current item - which means that the current item is a beetle -, 8 means that there is a bug over the current item - so it is inactive and no actions can be performed on that. 9 means that the two pieces are not adjacent, and the current piece is already set.

  Introducing the number 9 is required, because otherwise the model would not be able to distinguish the initial state from the state in turn one. Both would be represented with a matrix filled with null values, since there are no adjacent pieces in both cases.

- I loop over each played piece - the list of played pieces is available in the inner representation. For each piece the related column should be reset in the adjacency matrix. The adjacent tiles of a tile can easily be queried from the Board object, which holds the position of the played pieces in a dynamically extendible two-dimensional array.

5.4.1 Loading state

In order to load a state, I need an adjacency matrix and the information which player's turn is the next. First I check if there is any numbers in the matrix besides zeros. If not, than it represents the initial state and there is nothing to do.

After that I pick the first non-null column from the matrix and I put down that piece. The next step is a breadth-first search on the graph represented by the adjacency matrix starting from the piece which is already placed. While traversing the graph I put down the pieces one-by-one.

5.4.2 Import from and export to json file

I also implemented a solution to store the state of game to the persistent storage. One of the best ways to do that is to use the JavaScript Object Notation (json). It is a semi-structured file format which has the advantageous nature that it can be easily parsed and de-parsed with any programming language and it is easily readably for human as well.

For the sake of a complete backup of the game state we have to store two things:

- The current player. Json can only store a few types of data, but it causes no problem, because the current player can be stored as a simple string.

- The pieces and their positions. This is stored in a dictionary, where each key is a string holding the two indexes of a hexagon. The value is a list of strings which represents the pieces. The pieces are represented in the usual way: a 3 digit string holding a color, a kind, and a number (e.g. wQ1 means the white Queen, there is only one of this kind). The position of the hexagon is converted to string with the built-in *___repr___()* function of the ordinary tuple type in Python. Because the hexagons are stored as a named tuple, it has to be converted into an ordinary tuple before calling the *str()* function.

This import/export feature was a great help during debugging of the software. Using this tool it is much easier to create a regression or unit test after a bug fix: I just have to save the state of the game, load it from the test case and perform the very same action which caused the faulty behaviour. Those tests are stored in a different file called *regression_test.py*.

A possible future improvement in this area would be to use a json schema file to validate the format of the import file.

5.5 Implementing Monte Carlo Tree Search

The logic related to the search tree can be found in Section 2.6 and is organized in a class. A tree object has two methods which relevant to the learning algorithm. The *getActionProb* method can be used to run a number of simulations on the tree starting from a state which is given as parameter. This function uses the *search* method several times to extend and improve the tree.

The other relevant method is *search* which allow us perform one iteration of the tree search. It recursively calls itself until a leaf node is found. The action chosen at each node is the one that the maximum upper confidence bound has chosen. Once a leaf node is found, the neural network is called to return an initial policy P and a value V for the state. This value is propagated up the search path. In case the leaf node is a terminal state, the outcome is propagated up the search path.

The search tree uses the alphaGo interface of the game engine, which can be found in the *Game Module*. It uses this interface to make moves in name of both of the players, and the neural network predicts the value of each state for the canonical representation for each player. This means practically that the colors of the pieces on board are inverted in each step. The visited states are stored in a hash map, so that identical states can be detected.

5.5.1 Components

The constructor of the $MCTS$ class takes $(e, n, args)$ as parameters, where e is the representation of the environment, n is the neural network implementation – including the current weights – and $args$ is a dictionary of custom configurations. The constructor has nothing else to do than store the initialization parameters.

The function getActionProb performs n simulations – where n is determined in dictionary $args$ – of MCTS starting from s, which is passed as a parameter.

The function $search()$ takes state s as parameter, and performs one iteration of MCTS. It is recursively called until a leaf node. The nodes of the search tree are represented

as a string representation of the adjacency matrix, because the string python object is hashable. If the leaf node is found, the method asks environment e for all the valid movements $\vec{m} = valid\_actions(s)$, $\vec{m} \in \{0,1\}^*$ and calls neural network n for $\vec{\pi}$ policy vector. The policy vector is masked by the validation vector: $\vec{\pi_v} = \vec{\pi} \odot \vec{m}$. Now all we have to do is to store $\vec{\pi_v}$ to the corresponding node of the tree and return the value returned by the neural network multiplied by -1. This multiplication is needed, because if the value v is the result of the current player, than the opposite is the reward for the other player. The multiplication with -1 deals with the fact, that the next player regards the reward differently.

5.6 Modules for the learning phase

5.6.1 Coach module

The *Coach Module* module contains the functionality which manages the learning process. It uses the Monte Carlo search tree module to make predictions using a neural network. It performs multiple iterations of self-play. After each iteration the *learn* method retrains the neural network using the training examples obtained during the game. After that I run a match with the updated neural network against the previous most successful version. The modified neural network is saved if it beats the previous winner.

Executing one episode of self-play given a neural network means the following:

- The game starts with player one - which is the white player in case of Hive.
- As the game is played, each turn is added as a training example to a list, and the search tree is updated.
- The game is played until somebody wins or it's draw. After the game ends, the outcome of the game is used to assign values to each training example. The end-result is a list of tuples of size three. Each tuple consists of a state, a policy vector which is predicted by the neural network and is masked with the possible actions in that state, and a value, which comes from the result of the game. The current implementation returns 1, if the player eventually won the game, else -1. Draw games could be distinguished as well. That is a matter of principle if I want to teach the AI to win, or to avoid losing. The reward function should be modified accordingly.

The module described above contains the core essence of the learning process. The *Coach* object can be initialized easily, the constructor needs only the environment, a neural network to work with. I can also pass a bunch of parameters to it, with which the learning can be configured. The parameters are the following:

- *numIters* The number of iterations of learning. Each iteration has its own monte carlo search tree, but the neural network is common. The neural network is updated in each iteration using training examples obtained from several episodes.
- *numEps* The number of self-play episodes. It is the number of matches that should be played until the end in one iteration of training. The episodes in one iteration use the same search tree. An episode starts with the initial board and ends with a state where there is a winner or the result is draw.

- *numMCTSSims* The number of simulations performed at once after poking the search tree to expand himself.
- *cpuct* The parameter of the upper confidence bound. The lesser this value is, the lesser the probability is that the search tree wants to explore a new or a less valuable path.
- *maxlenOfQueue* The maximum length of the queue which holds the training examples. This is useful when one has limited memory capacity.
- *tempThreshold* A threshold of game numbers. Before *tempThreshold* episodes of game I choose an action randomly from the search tree, after the threshold the best action is taken.[28]

There are also some somewhat less interesting parameters like file name where the neural network can be saved to and can be load from. These parameters and the creation of the Coach object is located in the *Learning Module*.

The constructor of the *Coach* class takes e, n, a as parameter, where e is the representation of the environment, n is the neural network to use, and a is a set of additional parameters, which configures the MCTS and the neural network.

The constructor initializes the network as well as the competitor network. The latter is initialized as a copy of the original neural network – which neural network is initialized to generate its output randomly. After each episode of self-play, the neural network is updated, but the competitor network is delayed with one generation. This way we can challenge the neural network with the updated parameters with the predecessor network and see whether the new model performs better.

The *Coach* class has two main methods: $executeEpisode()$ and $learn()$. executeEpisode executes one episode of self-play. The game is started with player 1 and played until the end of game. Each turn is stored as a training example inside *Coach* in a simple list. After the game ends, the outcome of the game is used to assign values to each example in the training examples. The method returns a list of training examples in the form of (s, π, v), where s is a state, π is the policy vector assigned to state s, and v is $+1$ if the player eventually won the game, -1 if defeated, 0 otherwise (draw).

The pseudocode of *executeEpisode* can be found at algorithm 5. This algorithm is only a simplified version of the method, because it does not deal with symmetrical states.

Algorithm 5 Execute one episode

```
1:  procedure executeEpisode(env)
2:      trainExamples ← []
3:      s ← env.getInitBoard()
4:      currentPlayer ← 1
5:      episodeStep ← 0
6:      while r ≠ 0 do
7:          episodeStep ← episodeStep + 1
8:          cs ← eng.canonical(s, currentPlayer)
9:          π ← mcts.getActionProb(cs)
10:         r ← env.getGameEnded(cs, currentPlayer)
11:         trainExamples ← trainExamples ∪ {(cs, π, r)}
12:     return trainExamples
```

As you can see, the method always work in the first player's point of view. This makes sense, because the role and the game rules for the two players are identical, so the state of one player could be used when playing with the other player. The only difference between the two player is that conventionally the white player makes the first step, but it is irrelevant for us.

So in order to examine the state always in a canonical manner, we need to use always one player's point of view. This is what happens in the method *canonical*(). This method of the environment will invert the colors in case it is the other player's turn.

The algorithm iterates until the game has ended. In each iteration it uses the MCTS to generate a policy vector π for state s and updates the training examples. It uses π to choose and action in state s according to the probabilities and executes that. If the next state is terminal, than the method returns the training examples collected.

The *learn*() method is the starting point of the algorithm. It performs *ni* iterations with *ne* number of episodes of self-play in each iteration. After every iteration, it retrains the neural network with examples in the train examples – which has a maximum length set by a configuration. It then pits the new neural network against the old one and accepts it only if it wins $w \geq updateThreshold$ fraction of games. *updateThreshold* is a configuration parameter which specifies how better the freshly parametrized neural network has to perform compared to the old one.

After each iteration the weights of the neural network is stored into a file. Because of that, the *Coach* module also needed two methods for saving and loading the weights of the neural networks. There are two additional methods for storing not only the weights, but the weights alongside the whole structure. This is called only at the end of the training phase. The stored neural network can be reused then.

I did not append a pseudocode to this paper about the *learn*() method, because it has several different responsibilities and it contains too many implementation specific details.

5.6.2 Arena module

The *Arena* class defines an object, which can be used to pit different players against each other. The constructor takes the following parameters:

- **player1**: An object that represents the player, who starts the game.
- **player2**: An object that represents the player, who is the second in turn.
- **Environment**: This object represents the game. This parameter is optional. If *None*, than the Arena object creates and initializes a game from the beginning.

5.6.3 Player module

The classes which represent players have to implement the *Player* interface, which was designed by me. This interface has the following methods:

- **step**: The *step* method calculates the next desired step of the player given a state. An *Environment* object is passed to the *step* method, and the state of the game can be queried through the *Environment* object. The method does not perform any step on the engine, it only returns the desired step. It returns a tuple of (p, t),

where p is the object representation of the piece, which should be moved, and t is a hexagon, which is the target of the movement.

One can also call *step* if there are no available movements for the AI. In that case the method should return the string *pass*.

- **feedback**: This method returns nothing. It is a callback function which is called after the action returned by *step* is actually executed on the engine. This is useful for logging the behavior of the AI or to communicate with the player in case it is a human player. The method has parameter s, which is a boolean that indicates whether the execution of the action succeeded or not.

At the time of writing this paper, the following implementations of *Player* exist:

- **HumanPlayer**: This implementation asks for decision on the standard input during execution of the *step* method.
- **RandomPlayer**: The AI chooses a random action from the available ones.
- **AplhaPlayer**: The AI uses the neural network and MCTs resulting from the learning phase of the AplhaGo Zero algorithm.
- **BaselinePlayer**: The AI uses the result of the training of an arbitrary Stable Baselines method.

The HumanPlayer class waits for an action from the standard input. The following text inputs are valid:

- The string "*pass*". It means that I want to pass my turn. The player can only pass if he has no available movements.
- Three characters: the first character represents the color of the piece, the second represents the type of the piece, and the third is the number. E.g. *wA2*. It is a valid action in only one case: if it is the very first action of the game.
- 8 characters: the first 3 characters are the identification of $p1$, the next two characters determine the direction d, and the last 3 defines $p3$. d is called "*point of contract*" in the code. There are six different point of contracts:
  - "/*": Put $p1$ southeast from $p2$
  - "*\": Put $p1$ southwest from $p2$
  - "*": Put $p1$ northeast from $p2$
  - "*/": Put $p1$ northwest from $p2$
  - "|*": Put $p1$ east from $p2$
  - "*|": Put $p1$ west from $p2$

Luckily, it is not necessary to distinguish a piece placement action from a piece movement, because only one of them can be a valid action at the same time. For example, the following command: "$wA2 * |wQ1$" can either mean a bug placement next to the white queen, or a movement. It can be decided whether it is a movement or a placement by trying to find the piece on the board. If it is already placed, than it is a move attempt.

The AI based on AlphaGp Zero on the other hand requires an already prepared neural network to be passed to its constructor. It than initializes an MCTS, which can be used to produce an improved policy vector. The AI will choose the action with the maximum probability in π returned by the MCTS object. The problem that I encountered here was, that the MCTS module can only work with the white player. But what if I want to use the AI on the opposite side?

In that case, all we have to do is to invert the state of the game. Every white piece should turn black and every black piece should be considered as a white one. Now I can call the *getActionProb* method of the MCTS to generate the π values. The resulting action is still not perfect, because the action can be interpreted only in the inverted state. The next task is to invert back the action to the original state.

Furthermore, in order to ensure that the action is valid, we have to mask the resulting π vector with the possible movements vector the same way as we did it in the MCTS algorithm. The $feedback$ method of the *AlphaPlayer* can simply be a blank method, or we can use it to generate some information to the log file.

The BaselinePlayer is a quite simple player. It calls the *predict* function of the OpenAI Gym's Env interface, which generates the desired action. If it is not a valid action according to the Hive rules, than we call *predcit* until it is finally valid. Finally, the resulting action has to be decoded and returned.

As you may have already noticed, there is no player for the human to play on the graphical interface. This has technical reasons. There has to be a different start point for the GUI than a regular console application, so the user can start the game without pulling every dependency needed for PyQt. Because of this, I did not integrate the GUI tightly to the rest of the application.

5.7 Structure of the network

The neural network is parametrized by the two-dimensional representation of the game state. There is no need to pass a boolean whether white or black player is the current player, because we train the network always with the white player.

The neural network outputs a value $v_\Theta \in [-1, 1]$ from the perspective of the white player and a policy vector p, which represents the stochastic policy vector used during self-play.

The neural network uses the following formula as its loss function:

$l = \sum_t (v_\Theta(s_t) - t_t)^2 + \vec{\pi}_t \log(\vec{p}_\Theta(s_t))$

Where t refers to the input data, v_Θ is the value generated by the network, s_t is the state, z_t is the actual value of the state - which is the outcome of the game from the perspective of the white player, $\vec{\pi}_t$ is the improved estimate of the policy after performing MCTS starting from s_t.

The layers of the neural network is the following:

- The first two layers are two-dimensional convolutional layers with paddings to output the same size.
- The next two layers are 2D convolutional layers too, but without padding.
- After that the output of the forth convolution is made flat.

- Than there are two dense layers with dropout included after both of them.
- There are two outputs of the network. Both come from a dense layer after all the above layers. One for creating the policy vector, one for predicting the value of the state.

For the convolutional layers rectified linear unit (ReLU) is used as non-linearity. The policy vector is made using a *softmax* non-linearity and the value is generated with a tangent hyperbolic activation function.

The model uses categorical crossentropy as loss function and Adam as optimizer.

5.8 Incremental research and development

It is a common practice during the design and implementation of a Reinforcement Learning model to restrict some of the functionalities of the model in order to simplify the problem and incrementally make the problem more and more complex afterwards.

My first attempt to train the AI based ono AplhaGo Zero on the Hive environment was with the following parameters (only those parameters are listed, which affects the time of the learn phase):

- Number of iterations: 1000. It means the training process should build up 1000 different models, 1000 different search trees, and the models should always pit against the previous ones.
- Number of episodes: 500. An episode is a self-play until the end of game working on the same search tree. Altogether is means $1000 * 500 = 500000$ episodes.
- Number of MCTS simulations: 10. There are 10 simulations in each iteration of the search tree.
- *ArenaCompare*: 40. After successfully collecting the training examples from the search tree, the neural network is trained, and we have a new model. After that the new model is tried out by making it pitting against the previous model 40 times. The more *ArenaCompare* is, the better is the probability that the current model is *truly* better than the previous one.

I tried to parametrize the neural network with the above parameters. It turned out that it would take several years to run it on a low to middle-end computer.

The main reason the AI learns that slow – compared to for example the game called Othello – is that a large number of turns might elapse until the game is over and we can finally pass a reward to the neural network.

Theoretically, the number of turn elapsed until the end of game can be infinite. Let's consider for example a scenario, where the two opponents move their beetle piece back-an-forth all the time.

Another aspect that makes my task challenging is the size of the action space. In order to avoid this problem I could restrict the movements of the pieces, that way I would get a problem similar, but easier to Hive. The two restrictions that were the most powerful are the following:

- The pieces can only be set in a specific order for both opponents. With this restriction the size of the action vector is reduced to less than the half of the original size. A big disadvantage is that the resulting game is considerably different. It is a big challenge during the game to decide which piece to put on board next time. I decided not to make this restriction.

- Reduce the movement of the ant pieces. In the developer's point of view the ant is the most challenging piece to deal with, because its high degree of freedom when it comes to moving a piece. An upper estimation of the maximum number of tiles where the ant can move is: $5 * (n - 1)$, where n is the number of pieces in the game (22 without extensions). The ant can move next to $n - 1$ pieces (every piece except itself) and it can choose 5 sides of the piece at most. One of the sides must be occupied for every piece.

  Without the extension, it would mean $5 * 21 = 105$ different tiles. Instead of doing so, we can say that in practise that an ant piece can never be put to more than 50 tiles. I decided to use this restriction, because it does not affect the game in practice.

Another way to minimize the time needs of the software is to change the victory conditions. There are two possible ways to efficiently do so:

- Introduce a **step limit**. When the total elapsed number of turns are equal to the step limit, we make an announcement of the results. The player with more tiles around the queen piece loses, the other player wins. If the number of adjacent pieces are the same for both of the opponents, it is a draw.

  It turns out to be a great way to simplify the problem, because it successfully solve the problem of the AI's playing forever.

- Change the victory conditions. The original rule is that whoever has a queen piece with all the 6 neighbors occupied loses the game. It can be generalized to $n \in [1; \infty)$ occupied neighbors instead of 6.

  The case $n = 1$ means that whoever decides to put down the queen piece first, loses. This is certainly not an interesting game for us, but it is a good starting point when testing the AI. In this case the AI does not have to learn how to make or deal with an offensive movement, thus it is a much easier game.

  However, in case of $n = 2$ the best strategy is to place your own queen in the third turn of yours. It shouldn't be placed in the second turn, because the third movement in that case can be only suicidal. The forth bug to place should be an ant, which can defeat the opponent in the fifth's turn.

  The $n = 3$ seems to be not trivial already.

I tried out both the above methods to simplify the game. Both ways turn out to be efficient, but I prefer using a step limit, because it makes sure there won't be any matches with unreasonable number of turns.

5.9 Stable Baselines integration

The obove algorithm based on the original AlphaGo Zero implementation[25] is a complex way to achieve a working intelligent agent. In order to prove that this complexity is

mandatory, I also integrated the game environment to the Stable Baselines3.2 framework. As a first step, my goal was to implement the interfaces provided by the OpenAI Gym3.1 project.

5.9.1 Implement the OpenAI Gym environment

The following interfaces had to be implemented:

- **Env**: The environment which represents the board game. Has a *step*() methods with which actions can be elaborated. It can also be reset. Let the name of the implementation be **HiveEnv**.
- **Space**: The Space interface is responsible for representing an action or observation space. For the observation space – which is the state of the game – I could use a builtin class called *Box*. *Box* can be used to represent the state of the game with a multidimensional array with optional low and high bounds.

  However, the action space must be implemented, because none of the builtin types are suitable for representing the action space. The resulting class is called **HiveActionSpace**.

The *HiveEnv* class sets an object property called *reward_range* by initialization. This range represents the bounds of possible rewards during the training of the model. This is necessary to set, because some of the algorithm in the Stable Baselines implementations might use it. I set it to $[-1, 1]$, because the reward will be -1 in case the algorithm loses the game, and 1 if it wins.

The observation and action space also has to be initialized in the constructor of the environment. The action space is initialized to be a *HiveActionSpace*, and the other has to be a *Box*. The *Box* can be initialized with bounds. The lowest value that can occur is 0, the highest is 9. I use here the very same adjacency matrix as by the AlphaGo Zero implementation. The dimensions of the box can be set in the constructor as well. In my case it is $(22, 21)$, i.e. the dimensions of the adjacency matrix. The type of the values in this multidimensional array is set to *numpy.uint*8 which is a type used in the *numpy* module that represent a simple unsigned byte.

The first and most intuitive method to implement is *reset*(). There is nothing else to do than create a new Hive object – I use the same *Environment* object, as in the AlphaGo Zero implementation – and reset the action- and the observation space.

An method, which is easy to implement is the *render*() method which can be used to display the current state in a human readable way. I could use the graphical interface as well, but I decided to stick with the ASCII-art board display, because it is easier to store in a log file.

One of the most important methods is the *step*() method, which requires an action as a parameter. The action can be any arbitrary object. In my case, it makes sense to use the integer representation of the action space. That means the passed object is a simple integer. The *step*() method should return a tuple of form (obs, r, d, i) (see 3.1).

The body of the *step*() method does the following steps:

- Perform action a, which was received as parameter. In order to do that, the integer parameter has to be decoded.

- If the action did not succeed, return with $(obs, -1, 0,)$ where *obs* is the original state. Here the attempt of performing an invalid action is punished with the "reward" of -1. The discussion whether it is a good idea or not can be found later.
- Check if the game is finished. In that case return $+1$ as reward if game is won, 0 if draw, -1 otherwise.
- In case the game is unfinished, let the opponent step. In our case – as a first attempt – the opponent is the random player (this player performs a random action each time).
- If the game is finished as the opponent stepped, return the values accordingly. If not, than we have to check whether the RL agent has available moves or it has to pass.
- If there are no available moves for the agent, than we should continue stepping until the game is finished or there are available moves for the agent.

The above description is represented by the decision diagram 5.2. One of the interesting thing here is the handling of failure. Sadly, the Stable Baselines framework does not support masking valid actions in the action space. Because of that, it can often happen that the neural network makes a prediction which is illegal – against the rule descriptions – in the current state. In that case, I decided to punish the Reinforcement Learning algorithm with a "reward" of -1, and the state remains the origin.

Another interesting aspect is the handling of "passing" situations. When a player has no available moves, i.e. all of its bugs are blocked by other bugs, the player has to *pass*. It means the player who still has movements should make moves until the other player has something to move. In practice, it can be handled in the AI's point of view, that the series of moves performed until the other player is freed counts as one single move. This way, whenever it is the turn of the AI, we can be sure that the AI is not blocked from movement.

Having both the players unable to move is physically impossible, so there is no need to handle that case.

5.9.2 Use Stable Baselines on environment

In order to make use of the Gym environment, we can use the Stable Baselines framework. It is quite easy to use. The following steps have to be performed:

- Initialize a $HiveEnv$ object.
- Put it in a $DummyVecEnv$. The Stable Baselines interface uses an environment vector, which is their standard way to support more than one environments. $DummyVecEnv$ wraps the single environment so it can be used by the Stable Baselines interface.
- Instantiate a Reinforcement Learning model of your choice. First, I tried out the Actor-Critic model called "$A2C$", because it is one of the simplest and I have already heard of it. A policy also has to be passed to that model, which desides e.g. the structure of the neural network behind the algorithm. I chose a builtin policy called "$MlpPolicy$", which means that the algorithm will use simple multilayer perceptrons

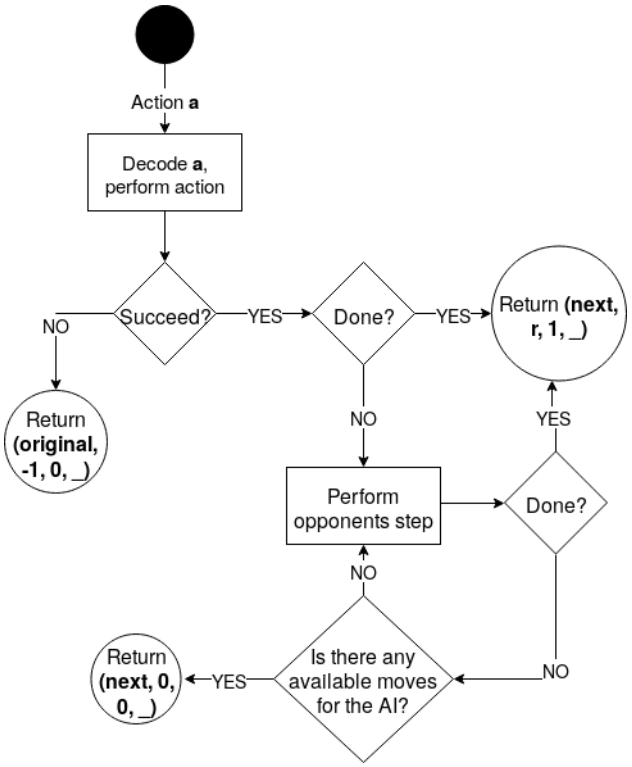

Figure 5.2: Decision graph of the *step*() method.

(two layers with 64 nodes in each, to be specific). Again, the main reason for this decision is simplicity.

- Call *learn*() on model. The number of total timesteps can be passed as a parameter. The more timesteps the learning consists of, the better the result will be – at least if the validation loss is decreasing. The *learn*() method will take care of the whole Reinforcement Learning process related to the Actor-Critic method for us.
- Reset the environment. This is required, because the training process might leave the state of the environment in a not initial state.
- Try out what we got. The model can be asked for a decision with the *predict*() method, which asks for the current state as parameter, and returns the desired action.

Chapter 6

Testing, Evaluation and Results

6.1 Testing functionality

I made several tests during my work which are separated into various unit test modules. I use the *nosetests* Python utility to manage the test execution. The following test modules can be executed:

- *Board Test Module* tests the Board class which is responsible for the inner game representation.
- *Hive Test Module* tests the Hive object and the validation functions. It uses a pre-created board and performs steps on that with both players.
- *Representation Test Module* can be used to verify the conversion logic and the functions which operate with the state and action space representations used by the learning algorithm.
- *Random AI Test Module* is a tricky test suite which tests both the logic of the Hive class and the representations. It tests them with executing a game with random moves from both opponents for a given time. The suite expects that no exception should happen during the execution. It is a good way to test whether there are any invalid moves listed in methods like *getValidMoves*.

Testing the learning process is a much more difficult problem, because that is time consuming process, and also the expected result is not easily testable. However, some of the functionality of the Monte Carlo tree search could be tested.

6.1.1 Testing Board and movements

The *TextHexBoard* class, which can be found in *Board Test Module* is responsible for testing the functionality of the Board class. It test the dynamical resizing of the board, the functionality of *get_surrounding*, which can be used to get the adjacent tiles of a cell, and the *get_boundaries* function, which returns the size of the board with the internally used indexing.

The *HiveTest* class holds the tests for simple bug movements and placements. The class tests the functionality of the *Hive* class and the *hive_representation* module. In the setup phase, it creates a board shown in figure 6.1. Every pieces is represented with three

characters: the color, the kind and the number of the piece. For example "*wS2*" refers to the second white spider.

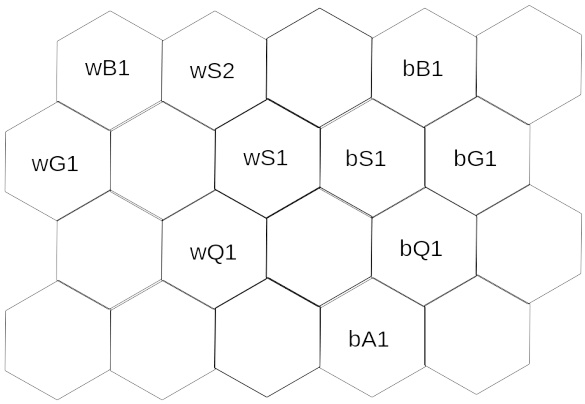

Figure 6.1: Initial phase of the *HiveTest* test suite

This layout has several advantages when it comes to testing the Hive engine:

- One hive rule can be tested easily with e.g. trying to move white spider number two. It should not move since it would tore the board apart.
- All the pieces appear in the test. The grasshoppers appear in an even and in an odd row index too. This is important, because the two dimensional mapping of the hexagonal board handles even and odd rows a bit differently.
- Piece placements can be easily tested too.

I also have a separate file for regression-testing. In case of a buggy behavior, the system automatically saves the last state of the game alongside the action that coused the failure in a json file. This json file can be loaded in the test suite and the same action can be performed there to make sure that this failure will never occur anymore. The system always store only the state of the last failure. This state has to be copied manually to the test suite.

6.1.2 Testing the representation

After testing the functionality of the Hive engine, it is also wise to test the inputs and outputs of the *Environment* module as well, which converts the representation to be suitable to teach the neural network.

The test read its input from a json file. The json file contains an adjacency matrix, which represents the same state that is used in *HiveTest*. After that, if everything went well, it

queries the binary representation of the action space, and it tries to execute each of them. It also asserts the structure of the action vector with an expected value which is stored in the json file.

With executing each actions in the binary action vector, the conversion from an action number to a (piece, target cell) pair is tested. A missing feature from the test is that it does not test the failure of the action, which are not allowed. In the future, it should try to execute those action numbers from the action vector too, which are disabled. The expected result in that case is a *HiveException*.

6.2 Evaluation and Results

The overall performance of a proposed solution can be measured by pitting it against other (baseline) methods or to have it play against a human player.

In this project, at the time of writing there is only one baseline method implemented, which is the random search algorithm. The random search algorithm takes actions randomly from the set of available actions and executes them.

6.2.1 Evaluating Stable Baselines algorithms

Deciding the effectivity of a particular algorithm after the learning process can be done the following way: make a match of n separate games against an arbitrary opponent and count how many time it won. Because the learning process used the random search algorithm – which takes an action randomly from the available ones – it makes sense to test the result with the same AI.

At the moment of writing this paper, I did not find any configurations and algorithms of the Stable Baselines tool set that could perform better than the random search algorithm. The reasons are the following:

- The Stable Baselines toolset does not support having an action space, where the actions might be illegal in some states. For example, a player cannot move pieces if the queen piece is not yet set, i.e. a player cannot move a piece which is not yet set.

  The most common way to deal with this problem is to generate negative reward in case of an illegal movement – just like I did. The problem with that is, that illegal movements are much more common than winning or losing a game, thus the algorithm will optimize for avoiding illegal movements instead of finding the winning steps for a long-term reward.

- The game is overly complex to be trained on a simple personal computer as mine. It is also an ongoing task to use GPU for learning and to test whether it would speed up the learning phase.

- The environment created to train the Stable Baselines models use the random search as an opponent, because the toolkit doesn't support multi agent environments. This means that even if I managed to train the AI to play properly against the random search, it would not mean automatically that the same AI would be sufficient to play against human players.

6.2.2 Results

Having the victory conditions modified to $n = 2$ (see section 5.8), I decided to reduce the time-consuming hyper-parameters until the learning process is less than 5 minutes. The new configuration can be seen in table 6.1.

Hyper-parameter name	**Value**
numIters	10
numEps	7
tempThreshold	15
updateThreshold	0.5
maxlenOfQueue	200000
numMCTSSims	2
arenaCompare	40
cpuct	0.8

Table 6.1: Configuration of time-consuming parameters to reduce time needs to 5 minutes

With the above configuration, after 5000 matches against the random search algorithm (see section: 4.3), I got the following results:

- Proposed solution won: 3553 times
- random search algorithm won: 1447 times
- Draw games: 0 times

In order to make the game as simple as possible, I decided to count all the draw results as a win of the random search algorithm. That way I did not have to handle draw games.

As it can be seen, the proposed solution won significantly more games than the Random player, and we needed only 5 minutes of self-play to achieve that.

Before these results, I had several attempt to teach the AI. The main reason of failure was because of the high reduction of the *cpuct* parameter. With the reduction of this parameter I managed to achieve a far faster learning process. A possible drawback is that the AI might end up in a local maximum of rewards due the lack of proper exploration of the state space.

Chapter 7

Summary

Implementing an intelligent agent for Hive[1] is challenging due to various reasons.

One difficulty was to represent the state of the board game in a way it can be used to pass to either an implementation similar to the AplhaGo Zero[25] algorithm or the Stable Baselines[11] tool set. My solution was to design matrix based representation of the state depending on the adjacency of the game tiles.

Once I accomplished to implement the environment capable to be used by Reinforcement Learning[27] algorithms I implemented the random search algorithm, which could be use as a baseline solution.

I also implemented an algorithm based on Monte Carlo Tree Search[7] and Reinforcement Learning that can surpass the random search algorithm without any domain knowledge of the game. The implementation was inspired by the AlphaGo Zero algorithm[25]. I managed to successfully prepare the neural network behind the algorithm, store the parametrized network and reuse the agent against an arbitrary opponent.

Furthermore, I integrated my environment to OpenAI Gym[6] and used Stable Baselines[11] to execute Reinforcement Learning algorithms like the A2C algorithm[20].

In order to help the implementation of the agent, I created various utility functionality for the game, like saving and loading game state, a Graphical User Interface for the game, a console interface for the human player and various tests for each module.

Chapter 8

Bibliography

[1] Official webside of hive. https://gen42.com/games/hive.

[2] Original hive implementation from joão lopes. https://github.com/jclopes/hive.

[3] Nose 1.3.7. https://github.com/nose-devs/nose, 2015.

[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[5] Terry Anderson. *The theory and practice of online learning.* Athabasca University Press, 2008.

[6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[7] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[8] François Chollet. keras. https://github.com/fchollet/keras, 2015.

[9] Antonio Gulli and Sujit Pal. *Deep Learning with Keras.* Packt Publishing Ltd, 2017.

[10] Richard H. R. Hahnloser, Rahul Sarpeshkar, Misha A. Mahowald, Rodney J. Douglas, and H. Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789):947–951, 2000. ISSN 1476-4687. DOI: 10.1038/35016072. URL https://doi.org/10.1038/35016072.

[11] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol,

Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

[12] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[13] Ronald A Howard. Dynamic programming and markov processes. 1960.

[14] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[15] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[16] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[17] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553): 436, 2015.

[18] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943. ISSN 1522-9602. DOI: 10.1007/BF02478259. URL https://doi.org/10.1007/BF02478259.

[19] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[20] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[21] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[22] Lennart Regebro. *Porting to Python 3: An in-depth guide*. CreateSpace, 2011.

[23] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[24] Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *International conference on artificial neural networks*, pages 92–101. Springer, 2010.

[25] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[26] Mark Summerfield. *Rapid GUI programming with Python and Qt: the definitive guide to PyQt programming*. Pearson Education, 2007.

[27] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[28] Hui Wang, Michael Emmerich, Mike Preuss, and Aske Plaat. Hyper-parameter sweep on alphazero general. *arXiv preprint arXiv:1903.08129*, 2019.

[29] P.J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Harvard University, 1975. URL `https://books.google.hu/books?id=z81XmgEACAAJ`.