Reinforcement learning in Minecraft

TDDE19 Project - Group 5

Viktor Barr, Martin Brolin, Nils Broman, Olle Stenvall, Agaton Sjöberg

January 10, 2021

1 Introduction

Neural networks have shown prominent results in machine learning models and the idea behind the neurons was inspired by the human brain. As the field of artificial intelligence (AI) attempts to move towards more general models that can perform arbitrary tasks, similar human-like concepts have arisen. One example of such an idea is reinforcement learning (RL), where an agent is faced with an environment and a list of actions to choose from each time step. As the agent takes an action, it receives a reward. One of the major challenges with RL is how the reward should be configured, when should the agent receive a reward, and how large or small should it be? In this report, the authors have experimented with a subset of the state of the art models being used in RL today.

The environment used to explore these RL models in this report was Minecraft. Minecraft is one of the best selling videogames of all time with over 200 million copies sold over different platforms [20]. It is an open-world game with endless possibilities and therefore it suits an RL agent well. Microsoft, the owners of Minecraft, has created the Malmo environment in order to let users of the environment explore and learn RL while receiving visually pleasing results of the agent acting in the Minecraft world. Malmo makes it easy for the user to define the map in which the agent will act, how the agent should receive reward and easy retrieval of the states. This makes the Malmo environment a suitable place to train RL agents in.

1.1 The problem

Initially, the idea was to train an agent to find and mine diamond, a rare resource in Minecraft that requires the agent to perform multiple subtasks necessary to reach the end goal of diamonds. As the authors realized that this task requires large amounts of time and extensive computing power, smaller subtasks were set up and strived upon completing instead. These problems will be explained further later in this report but generally, the tasks consisted of subtasks required to acquire wood, a common resource in Minecraft. The subtask of retrieving wood is finding it, and attacking the blocks so they can be picked up by the agent. This requires the agent to identify the trees and woodblocks, navigate to them, look at the woodblocks, and then attack the blocks in order to destroy them.

1.2 The solution

Tabular Q-learning, where each individual state maps to the best action to perform, is one of the simplest RL models available. This approach works well in environments where the action space and state space are small. In Minecraft, the action space is finite, however, the state of the environment is not. Therefore, more complex Q-learning models are used to solve the problem in this report, such as Deep Q-learning and models related to this idea.

1.2.1 Hypothesis

The hypothesis of this report was that a Deep Q-learning Network (DQN) at least should be able to complete the subtasks required to gather wood, and if possible the entire task including all of the subtasks can be learned by one single model. However, if it was shown impossible given the authors' time, computational power, and knowledge constraints, each subtask could be solved by individual models that can be used together in order to solve the problem, namely acquire wood.

1.2.2 Related work

Google DeepMind's Deep Q-learning agent learned to master the game Atari using a DQN agent. They did so using frames as input and trained the agent on 2600 games in order to reach optimal performance

in the game [12]. Furthermore, Thomas S has shown that it is possible to complete tasks using DQN in the game Doom [16]. There are similar results of DQN as the ones mentioned above which made the authors certain that a DQN model was a suitable solution to the problem of the report.

2 Theory

This section presents the reader with the necessary theory related to the project.

2.1 Malmo Enviroment

The Malmo Environment [6] was developed by Microsoft in order to provide a platform for researchers to experiment with different AI techniques. Malmo is built up by two components, a minecraft mod [4] and code to help the agents act on the environment. Malmo offers a tutorial and both API and XML Documentation [7]. Whereas the XML documentation covers how to set up a mission in an XML-file format and the API provides functions to handle input and output from the agent.

2.1.1 MineRL

MineRL [8] is a competition based on the Malmo Environment. The goal is to train a model to gather diamonds in Minecraft using state-of-the-art RL techniques. MineRL provides a data set with real players playing Minecraft with over 60 million frames and an NVIDIA P100 GPU for top contestants. However, the training time is limited to four days and thus far, no team has successfully managed to train a model to gather diamonds, highlighting the challenge with sparse rewards in RL.

2.2 Q-learning

Q-learning is a machine learning algorithm that learns the expected future reward for taking actions in specific states [17]. The algorithm does not use a model, instead, as the agent explores the environment, the quality of an action in a state is stored in a Q-table and the values in the Q-table can then be used to find a policy that maximizes the rewards. Q-learning using a Q-table is often called *tabular Q-learning*. The policy is learned by first initializing the Q-values to some arbitrary value. Then, at every time step t the agent takes an action a_t receives some reward r_t and moves to the next state s_{t+1}. The Q-value for taking that action in the state is then calculated with the formula:

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\Big) \tag{1}$$

Where α is the learning rate and γ is the discount factor, and are chosen by the programmer.

The Q-learning algorithm, combined with exploring different states with some strategy, can teach the agent to navigate simple maps by updating the Q-table until it converges to an optimal policy. Since the Q-table needs to be stored in memory, the algorithm does not scale well memory-wise. The Q-table is of size $actions * states$ and when the number of states increases, the Q-table simply becomes too big.

2.3 Deep Q-learning

In Deep Q-learning, a neural network is used to approximate the Q-function and was termed by DeepMind in 2013 when they used this method to create agents that learn to play Atari games with pixels from the game-frames as input [11]. Unlike in tabular Q-learning, where every state and action needs to be explored by the agent, in Deep Q-learning, the so-called Deep Q-Network (DQN) can learn a generalized policy function and approximate Q-values from states that the agent encounters for the first time. In a game like Minecraft, it is not feasible to explore every state and action combination so, therefore, using Deep Q-learning algorithms is a much better choice than tabular Q-learning.

The DQN is trained by letting an input state forward propagate through the network, calculate the loss, and update the weights with stochastic gradient descent and backpropagation.

The loss is calculated by subtracting the output estimated Q-value for the action the agent takes from the *target Q-value* for that action:

$$loss = E\big[r_{t+1} + \gamma \max_{a'} q_*(s', a')\big] - E\big[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}\big] \tag{2}$$

Where s' and a' are the state and action in the following time step. To calculate the max term the next state forward propagates through the network and get the max output.

2.3.1 Target Network

A *target network* is a copy of the DQN (which can be called the policy network) but its weights are frozen until the weights from the policy network are copied every n:th time step. The target network is used when calculating the max term in the loss function: the next state s' is passed through the target network and the state s through the policy network during training. DQN with a target network is also known as Double DQN (DDQN).

Using a target network has shown to increase stability when training the DQN. [10]

2.3.2 Experience Replay

Experience Replay is a technique where the agent's *experiences* at every time-step $e_t = (s_t, a_t, r_t, s_{t+1})$ are stored into a *Experience Replay Buffer* (ERB). To train the network a batch of experiences is sampled from the replay memory and fed into the network. This has shown to reduce the number of action executions required to learn a good policy in comparison to only learn from the most previous experience. [5].

2.3.3 Prioritized Experience Replay

An example of an extension to the implementation of the Experience Replay is to be selective when sampling a batch of experiences, is a Prioritized Experience Replay Buffer (PERB). Since the environment had sparse rewards, some memories were more important than others for learning. One measurement of importance for an experience is the error the experience produces between the target and policy networks. The priority value p_i for an experience e_i is the mentioned error plus an offset to get non-zero values (which can occur when the policy and target network outputs the same values for an experience). The probability that the experience is chosen is given by $Pr(i) = \frac{p_i}{\sum_n p}$) where n is the amount of experiences. Since the training data now favors the prioritized experiences it risks being overfitted. To avoid this skewed representation of data, a weighting factor w is introduced to the learning rate of each experience. This weighting factor is given by $w_i = \frac{1}{nPr(i)}$.

2.3.4 Algorithm

The DQN algorithm can be summarized into these steps:

1. Initialize replay memory size.
2. Randomly initialize network weights.
3. For each episode:

1. Initalize starting position.
2. For each time step:
    1. Choose action via some exploration strategy (see chapter 2.7).
    2. Do action in game.
    3. Get reward and next state.
    4. Store experience in replay memory.
    5. Sample batch from replay memory.
    6. Send states from experiences through DQN.
    7. Calculate loss.
    8. Optimize network to minimize loss.

2.4 Transfer Learning

Transfer learning is a way of repurposing an existing trained network to a different task[14]. There are many ways to do this, one way is just changing the fully connected layer. In a classification task, this can be done in such a way that the number of classes increases or decreases that you want to predict. Another way of using transfer learning is to only reuse a few of the first layers of a convolutional neural network (CNN). The first few layers usually extract information about the image, such as image features in a CNN used for classification. This can therefore be used as a spatial feature map extraction to the rest of your untrained network[2].

The motivation behind using transfer learning is that training a deep CNN is very costly and requires a large amount of data and good computational resources. Since the first few layers of a CNN is usually quite similar in most CNNs, they can be reused for different tasks, which in turn should make training faster and less computationally expensive[18]. There are a lot of resources available for reusing a pre-trained network, for example, through both the PyTorch library as well as the TensorFlow library you have access to pre-trained networks such as AlexNet, VGG, InceptionV3, GoogLeNet, and many more[19][1].

The image below 1 describes how transfer learning can be used in theory.

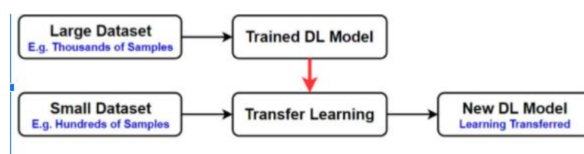

Figure 1: The idea behind tranfer learning [3].

2.5 Actor Critic

Another method that was also tested during the project was the implementation of the actor-critic algorithm. The algorithm that was chosen was Advantageous Actor-Critic (A2C), which is a variation of the Asynchronous Advandtagoeous Actor-Critic (A3C), first introduced by Minh V. et al. in the paper "Asynchronous Methods for Deep Reinforcement Learning"[9]. The idea behind Actor-Critic algorithms is to have two neural networks trained simultaneously, where all the layers in the network look the same except the last layer. In the last layer for the actor, the actor outputs probabilities for all actions that the agent can take given the state it is in. The critic however only outputs a single value from the last layer, this value is used to calculate the loss for taking specific action in the given state. Figure 2 below shows the idea of the actor-critic method.

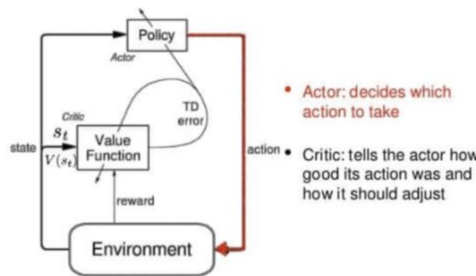

Figure 2: Figure from Sutton & Barto, 1998.

2.6 Exploration

In reinforcement learning exploration strategies are needed as an untrained agent likely would not explore optimal states and actions.

2.6.1 Stochastic Exploration

Greedy Epsilon Exploration (GEE) was the projects first and main exploration strategy. With GEE the agent would for each time step have a probability ϵ to take a random action instead of the action output from the DQN. The probability value for ϵ decayed as the agent learned.

2.6.2 Curiosity-driven exploration

GEE had limitations as most explored states were geographically close to the agents spawn point. A way to try to prevent this problem was to implement a Curiosity-driven exploration. Intrinsic Curiosity Module (ICM) was introduced, which was shown in the article "Curiosity-driven Exploration by Self-supervised Prediction" by Pathak et. al. to explore a similar environment more efficient than GEE[13]. Instead of stochastically choosing a random action, the ICM gave extra rewards to the DQN for being in states that were unfamiliar or *curious* for the agent. This way it teached the DQN to not only try to get rewards from the environment but also to get *intrinsic* rewards from exploring. The intrinsic reward r_i was calculated by a module consisting of three networks as seen in Figure 3. *Feature-network* inputted a state which consisted of one or more images and output a vector of features via CNN-layers, *Inverse-network* inputted the current and next time step's feature vectors and predicted what action was made between the states and *Forward-network* inputted the current time step's actions and feature vector and output a predicted next time step feature vector. The intrinsic reward r_i was a difference between the real and the predicted next time step's feature vector. In short, this means that the ICM's ability to recognize the relation between the current and next time step's state given an action showed how familiar this pair of state and action was and thereafter the agent would be rewarded. If the familiarity was high, the agent would be less rewarded as it probably already had explored the state-action pair and vice versa.

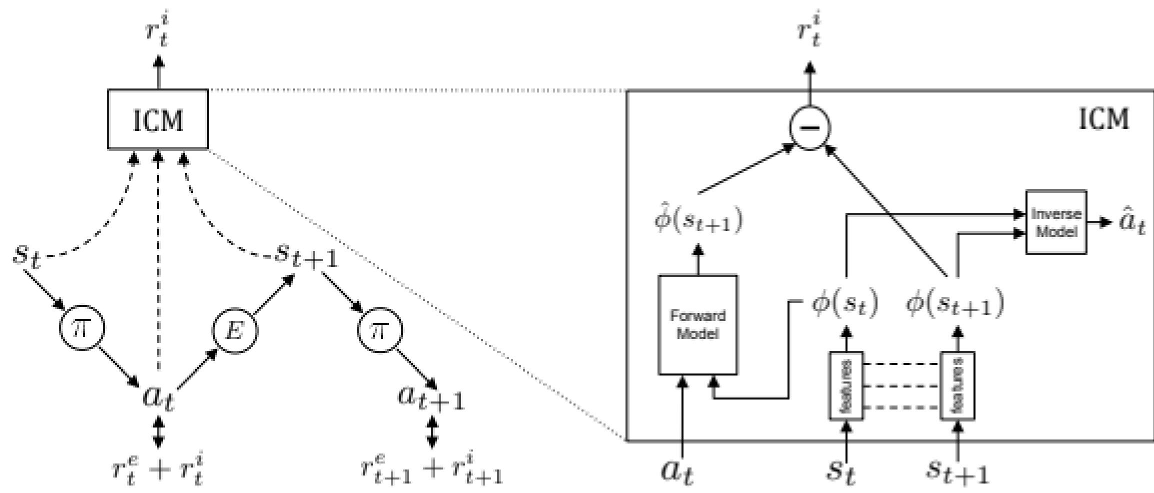

Figure 3: Figure from Pathack et. al. 2017

3 Method

This section will present the way in which the study was conducted. Generally, the study was done in an incremental manner. Chapter 3.1 describes the first step of the study, chapter 3.2 describes how manually created maps were time-inefficient and bad for general models. The work was divided into two branches, one where all models handle a grid-like input and the other handling frame input. These two chapters, 3.3 and 3.4, present the individual work on the two different types of states in a chronological order.

3.1 Environment

Since the data required for the agent to learn came from Minecraft and Malmo, the initial step of the study was to implement all of the necessary functionalities required to receive the training data. This data is retrieved each iteration and consisted of the state, action, next state, reward, and whether or not a episode is finished. Apart from making sure that the data required was available, worlds in which the agents were going to train in was created using Malmo's map creator. After the code regarding the environment was implemented, the reinforcement learning models could be developed.

3.2 Pathfinding model

To create a general model the decision was made to create a random map generator. This was done so that we could be sure our model didn't just learn a specific pattern for a select number of maps.

The first map generator that was created can generate random paths from a random point A to another random point B in a 2D-grid. Figure 4 below illustrates what a random path could look like.

Figure 4: Above the white square is the starting position, the yellow is the goal position and the gray squares is the path between the start and the goal.

In the Malmo environment the gray and white square represents cobblestone, the orange squares represent lava and the goal square is a lapis-block. The agent was awarded for reaching the goal position and was punished for stepping in the lava.

When starting a training session for the agent, you can input how many maps you want the agent to train on and be evaluated on. It was also possible to input the width of the grid N, the height of the grid M, and the minimum Manhattan distance between the start and the goal.

1. Create a grid of size $N * M$, where each cell is given a random integer between 1 and 100.
2. Pick two random positions in the grid where $0 <= x <= (N - 1), 0 <= y <= (M - 1)$. Where the starting position is denoted as **a** and the end position as **b**.

3. Calculate the Manhattan distance between the **a** and **b**.
4. If the value is smaller than the minimum distance imputed by the user, go back to step 2, else go to step 5.
5. Find the shortest path between point **a** and point **b**, using Djikstra's algorithm.
6. Create a grid of lava of the size $N * M$, fill in position **a** with a cobblestone-block, position **b** with a lapis-block, and the rest of the path with cobblestone-blocks as well.

3.3 Grid state

The agents started out having only a two-dimensional grid as state input. For example, a 3x3 grid of values, where each value represented a specific block type in the environment, with the agent positioned in the middle of the grid, as seen in Figure 5. The different block types were then transformed into a one-hot-vector and passed to our model.

Initially, the DQN algorithm (2.3.4) was used for our agent. When the agent could consistently complete simple environments, the algorithm was developed further to also complete more complex environments. Both DDQN and DDQN with PERB were implemented and when DDQN with PERB showed promise the other algorithms were discarded.

When the DDQN with PERB agent completed an environment, the environment was made more complex. As the environments became increasingly more complex, the state, action, and neural network structure were tuned to suit the new environment. For example, in Figure 6 the agent will not be able to complete the environment with only a 3x3 grid as state input, due to not knowing whether to go right or left. Therefore, the state was expanded to also include the previous action in order for the agent to know where it came from. In this example, the neural network structure was a single layer of 64 neurons, however, in other cases, the neural network was expanded. Further, the grid state input was expanded to a third dimension for environments with obstacles as well as trees. The biggest input was provided to a hill jumping agent which had an input of size 9x9x9. Another agent, a tree cutting agent, had an input of size 5x5x4 with additional input for previous action and pitch.

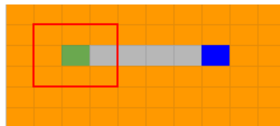

Figure 5: The green square is the agents current position, the blue square is the goal position and the gray squares is the path between the start and the goal. The red line indicates the input states received by the agent.

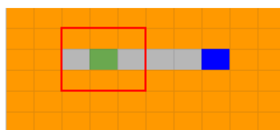

Figure 6: In this position, the agent does not know to go left or right based on the grid state only. Therefore, the previous action was added to the state.

3.3.1 A2C

The initial idea was to use A2C with the frames agent, but before testing the implementation on the frames agent the method was evaluated in the grid state agent. The agent was trained and evaluated on

both random maps created by the map generator and four easier maps which were four straight paths left, right, up and down.

3.4 Frame state

To create an agent that was as generalizable as possible the agent's state (DQN-input) was changed to pixels from the frames that a Minecraft player would see. Inspired by DeepMinds Ataris deep learning model, 3-5 consecutive frames from Minecraft were used to give the agent some temporal information but increased the input size.

3.4.1 Navigating to a block

The first challenge was to get the agent to navigate to a certain block in a simple flat platform map. The idea was that the grid state agent would not be able to find a goal when it was surrounded by one block type, so it would be a suitable challenge for the frame agent. The map was set up so there was a block which the agent received a reward of +100 when reaching, and a -1 reward for every action taken.

Firstly the agent's action space consisted of the moves: move south, north, east, west. The actions restricted the agent to face one direction. The agent's action space was later changed to the moves: move forward, turn right, turn left. Both action spaces were discrete, i.e the agent moved one block when executing a move command and turned 90 degrees when executing a turn move. The agent explored the environment using GEE with an epsilon starting at 1 and decaying to nearly 0 by the time of the last training episode.

Many different input preprocessing settings were used for the network input but that which showed the most promising results where 3 grayscale frames resized to 45x45. The idea behind this was to keep the input space small but give the DQN the ability to understand which way the agent is traveling (by having 3 frames) and enough pixels to see points of interest.

To learn features from the pixels the final network used three convolutional layers with 16, 32, and 64 output channels in that order. All convolutional layers had kernels of size 5x5, stride 2, and used ReLU as activation functions and the outputs of the layers were batch normalized. The output of the third layer was flattened and passed through a fully connected layer. The output of DQNs is the Q-values for every possible action given the state, so whether or not the agent uses turn commands affects the number of possible actions. Therefore, when designing networks for the different tasks the output size of the last fully connected layer of the network was changed to the correct number of actions.

The agent learned quite quickly to solve single maps but could not learn multiple maps with the same model. To try to solve this another exploration strategy was implemented and is described below.

3.4.2 Implementing curiosity

ICM was a substitute for GEE. This method was only used on the frame state agents as ICM relied on CNNs to semantically analyse one or multiple images. The architecture of was for based on the article "Curiosity-driven Exploration by Self-supervised Prediction" though had some adjustments to better fit Malmo and the computational resources the team had access to[13].

3.4.3 Transfer Learning

In the project, transfer learning was tested by trying to use the first few layers of the pre-trained network AlexNet as a feature extractor for the frames agent. The agent was trained in the same environment as described in section 3.4.1.

3.4.4 Tree finding

In the last weeks of the project, a new navigation challenge was setup. The idea was that it would be easier for the agent to identify and move to larger objects, so the goal of the challenge was to move into a one-block distance to a cluster of trees.

For this challenge, another random map generator was created that generated a random forest of trees in an open flat world. The forest consisted of four trees close to each other. The image 7 below illustrates what a random tree map could look like, both in a grid as well from the agent's point of view. Similarly to the pathfinding map, the programmer could specify the width, height, and minimum distance to the trees from the starting position.

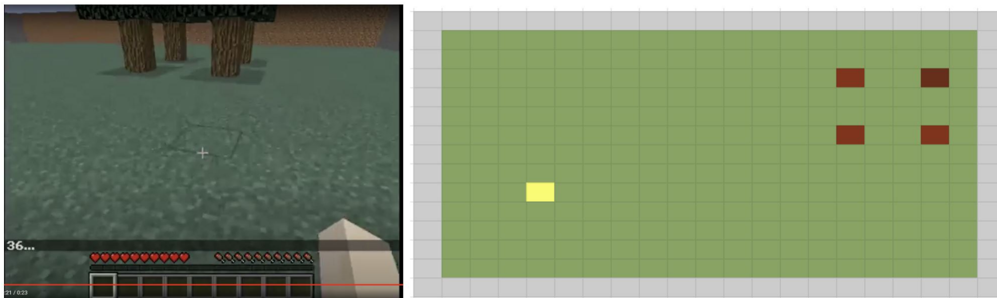

Figure 7: To the left, you can see what the map looked like for the agent during training, and to the right, the brown squares represent the trees, the yellow square the starting positions, the green square grass-blocks, and the surrounding gray-blocks are walls surrounding the bot.

The network, reward and exploration that showed the most promising results were the same as described in section 3.4.1.

3.4.5 Moving to a continuous action space

In the challenge of moving to the trees, a problem arose where when the trees spawned close to 45 degrees relative to the agents facing direction, the agent would be stuck trying to turn to face the trees since only 90-degree turns were possible. This was solved by moving to a continuous action space where the turning actions made the agent turn in smaller angles.

4 Results

In this section, the result of the project will be presented. Including the best model for each type of state and the environments in which the agent was successful.

4.1 Grid state

Using grid state, DQN, DDQN, and DDQN with PER were all successful in completing a subset of the grid state environments. The main difference between the models was the number of iterations that it took for the model to converge. DQN required the most number of iterations, followed by DDQN and the DDQN with PER converged the fastest. In Figure 8 the number of iterations are presented for all models in two different environments.

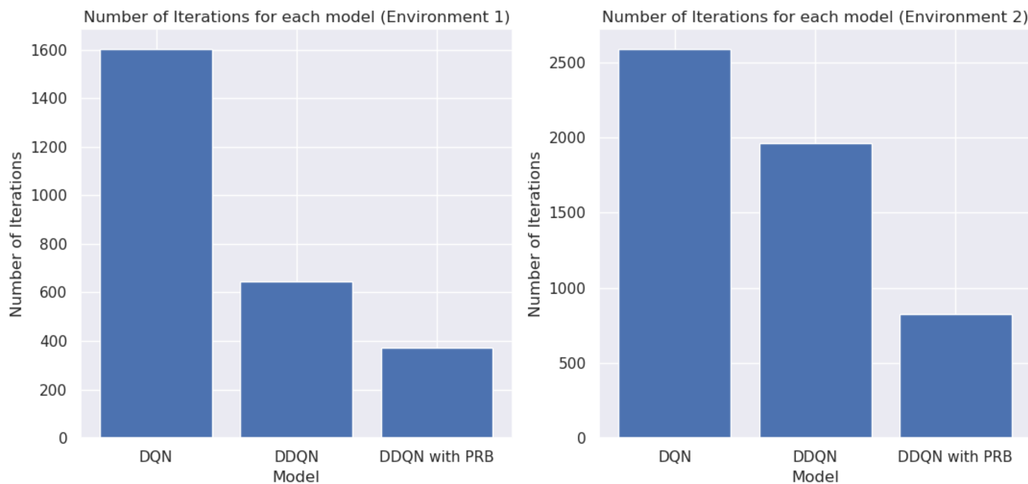

Figure 8: Number of iterations required for each model to complete the environments. Environment 1 is slightly easier than environment 2

4.1.1 Environments completed

The grid state models managed to complete multiple different sub-tasks required in order to gather wood in Minecraft. In this section, all the different environments and models with their respective state-action pairs will be presented. All of the results come from the most prominent model, the DDQN with PERB as can be seen in Table 1.

4.1.2 A2C

The A2C agent was implemented after the PERB agent and was not able to learn simple tasks such as pathfinding efficiently. It only seemed to be learning a few patterns and the agent did not work well with the random map generator. Due to the poor performance and the fact that the PERB agent was working for the grid state, this approach was therefore abolished at an early stage.

Environment	State	Action	Reward	I
Navigate towards a goal block. Only straight path of two blocks.	4 blocks, up, right, down left	Move west, south, north, east.	+Goal block. -Action	83
Navigate towards a goal block. Path like a maze.	4 blocks, up, right, down left and previous action.	Move west, south, north, east.	+Goal block. -Action	226
Navigate towards a goal block. Only straight path of two blocks.	4 blocks, up, right, down left and previous action.	Move forward, turn left, turn right.	+Goal block. -Action	372
Navigate towards a goal block. Path like a maze.	4 blocks, up, right, down left and previous action.	Move forward, turn left, turn right.	+Goal block. -Action	823
Navigate towards a goal block. Hills in environment.	9x9x9 blocks surrounding the agent.	Move forward, turn left, turn right, jump forward.	+Goal block. -Action	5631
Floor with a tree. The goal is to navigate to the tree and chop it down.	5x5x4 blocks surrounding the agent, previous action and pitch direction.	Move forward, turn left, turn right, attack, change pitch.	+Get wood. -Action	2340

Table 1: Completed environments and their corresponding state action pair, reward and number of iterations (I) required for convergence of the DDQN with PRB model.

4.2 Frame state

Several agents with states as frames were tested. This section will describe the different approaches that were tested, what results were yielded and what problems that occurred.

4.2.1 DDQN

The earliest implementations of an agent with frames as states was a DDQN and used ERB to obtain its training batches. This agent was for the most run on a single map where the task was to move to a coordinate which contained a blue block via the actions *move forward, right, backward* and *left.* The map was flat, the border blocks were yellow, and the rest of the floor was gray. The agent got a high negative reward for being on the yellow blocks, a low negative reward for being on the gray blocks and a high positive reward for being on the blue block. The space was discrete meaning for example that the action *move forward* moved the agent exactly one unit forward. The agent explored its environment using GEE. This model had some success, at certain points in the learning it could successfully and consistently complete the task. The path was however not optimal and therefore was theorized to not have *deep* knowledge. An example of this behaviour is shown in Figure 9 where the red line describes the agents movement. The agent was also trained with multiple maps with different coordinates on the blue block, varying the maps between iterations. This approach performed better than choosing random actions, but was less successful than only using one map.

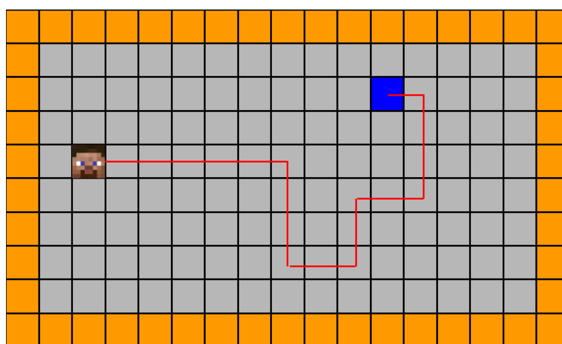

Figure 9: An action sequence the agent performed consistently during a certain point during training, successfully completing its task.

All frame agents was based on this implementation, using DDQN.

4.2.2 ICM

To better explore the environment, implementations were made which replaced GEE with ICM. Some implementations of ICM showed promising values, the intrinsic reward r_i was lower at the agents spawn point and higher in coordinates that was rarely explored. This meant that the agent was rewarded for being in unfamiliar states which was a wanted behaviour from the ICM. This did however not show any positive results from the DDQN and with this implementation the agent was less successful in completing the task the implementation without ICM.

4.2.3 Transfer learning

The result did not show any promise and this approach was abolished early on. After training the agent for 1000 iterations the agent still only went to the left completely disregarding the state. It was not able to produce any result that showed any type of improvement compared to other agents.

4.2.4 PERB

These implementations used GEE and used PERB instead of the regular Experience Replay Buffer. This did not show any positive results. Some implementations used the *tree finding* map, a continuous environment meaning that an action did not move the agent exactly one unit. The actions in continuous space was *move forward, turn left* and *right*. Agents with this implementation performed worse than choosing completely random actions.

5 Discussion

This section presents the discussions of the conducted study.

5.1 Project Malmo

The Malmo environment that was used made it possible to read the state as well as send commands to the agents. It was also easy to create new maps and set different rewards. However, initially, there were some issues with setting up the environment for 3 of the 5 project members who had Windows computers. This was due to that the pre-built Malmo environment was missing a file for the Windows version. Because of this, it took about a week of debugging and reading numerous forum posts to find the solution of how to get it started on Windows. There was no issue for the 2 members who had Linux and macOS as their operating systems.

Getting started with creating the agents was quite straight forward, project Malmo provided a tutorial that went over the basics of how to create agents and interact with the environment, both in the discrete and continuous space.

Another issue had with Malmo was that we had to have a Minecraft client running during training, meaning the graphical UI had to be rendered with each frame. Since it was rendered at each frame, it required us to sleep the agent after taking an action, this was to make sure the environment had changed before fetching a new observation. Both these factors led to that training took a long time. Training the frame agents for 2000 iterations took about 4-8 hours depending on the configuration of the bot. The training time for the grid state agent was faster but still slow in comparison to training a model using supervised learning on for example a classification task. Training for 2000 iterations with the grid state agents took about 2 hours. This led us to believe that one of the reasons that our frame agents were not able to learn the tasks they were given was due to limited computational resources and time. In comparison to the open AI baseline for DQNs trained on Atari games [15], the agents reward converged the agent had seen about 50-150 million frames depending on the model.

5.2 Main method

The main method and workflow were to work as iteratively as possible. This meant that before letting the agent explore a real procedurally generated Minecraft world, small training rooms were created. Since we never reached the point where the agent was ready to train in a real Minecraft world, this was a good decision. This made it possible to isolate problems with the agent. For example, if the agent could not learn to move away from a lava block that instantly gave the agent -100 in reward, it would not be able to learn a policy in a setting where rewards are more sparse.

The main problem with the method was implementing new features before adequate testing. Because of the big number of various hyperparameters and the limitation of computing power it is difficult to see if a model is truly working correctly, and what is causing problems. It was only in the later stages of the project where the models were adequately tested and results from different hyperparameter configurations documented. This should have been done from the beginning.

5.3 Result

The authors of the study initially expected better results than the ones actually achieved. This chapter will discuss why this was the case.

5.3.1 Grid state

In this project, using a grid as a state for the RL models resulted in the best agents. However, there are downsides to this approach as well. The following chapters will discuss the pros and cons of using a grid as the state.

5.3.1.1 Models

The simple DQN model was easy to implement and a good baseline model. However, even in simple environments, it required many iterations for convergence. The DDQN and DDQN with PERB techniques showed promise in speeding up the convergence. As was shown in the result, the difference in the number of iterations between the worst model DQN, and the best, DDQN with PERB, was a magnitude of around 4, see chapter 4.1. Given that the DDQN with PERB model was implemented at late stages of the project, it is plausible that the agent could have completed more complex environments given that the most impacting limitations were training time and computational power.

5.3.1.2 Limitations

In theory, the grid state works well on simple tasks. However, it does not scale well with increasingly complex environments. For example, each new type of block added will cause the input to grow by the grid dimensions. Furthermore, since the state is a plane or a cube, increasing the distance the agent can process, the state is expanded at a rapid rate. A larger state heavily impacts the time to train the agent. This is indicated in Table 1, for example in the first environment, the model converges in 83 iterations and as the state grows to a 9x9x9 cube, the number of iterations increases to 5631.

On the other hand, for simple environments, the grid state is quick and easy to set up and you get pleasing results faster than using frame state. This helped the authors to get started and generate some results and inspiration to develop the algorithms further.

5.3.2 Frame state

The frame state agents never worked in a satisfactory way. Other projects had succeeded in implementing a frame state agent that could do simple tasks so the reason to why our implementation could not is not known. One aspect which we could never fully explore was if the training time was sufficient. As we only had our own PCs for training with relatively low specs on both CPU and GPU, training was slow. 10000 iterations took over 24 hours of training and even with that long training time no *deep* knowledge was detected. The ICM paper trained their policy with a pretrained ICM with DOOM as the environment and reach convergence with over millions of iterations[13], however, they had another definition of an iteration; one-time step with numerous agents instead of our definition which was in average ca 40 time steps with one agent. Therefore we still do not know if we had the right implementation with just too little training time or vice versa.

5.3.2.1 DDQN

At times when the agent did solve the task in multiple consecutive iterations, it did seemingly not use any semantic information to complete its task. Figure 9 is an example of this, for the last few actions the agent could not even see the blue target block as the agents visual input was to the right of the agent, meaning that the agent had no visual indication of where the goal block was. This was likely due to early overfitting where the agent had learned the pattern from GEE and fitting each frame state to a specific action.

5.3.2.2 ICM

The implementation of ICM was a big part of this project and the values seemed as if they worked as mentioned in Section 4.2.2. However it was unclear whether it could be combined with and Experience Replay Buffer as the rewards for exploring would be stored for a long time, slowing the rate of which

the agent learned what areas to explore. The ICM paper used A3C where the batch was created the latest time step instead of creating it from ERB[13]. That way the agent would have a direct connection between intrinsic reward r_i for the time step t. In our implementation, the agent could learn from a batch of old experiences when a state was more unexplored and had a higher intrinsic reward.

References

[1] Torch Contibutors. *TORCHVISION.MODELS*. [Online; accessed 06-Jan-2021]. 2019. URL: https://pytorch.org/docs/stable/torchvision/models.html.

[2] Per-Erik Forssén. *TSBB17 Visual Object Recognition and Detection, Lecture 2: Feature Descriptors*. [Online; accessed 06-Jan-2021]. 2020. URL: https://gitlab.liu.se/cvl/tsbb17/-/blob/master/lecture02/Lecture2.pdf.

[3] Ahmed Gad. *Part 1: Image Classification using Features Extracted by Transfer Learning in Keras*. [Online; accessed 06-Jan-2021]. 2019. URL: https://www.alibabacloud.com/blog/part-1-image-classification-using-features-extracted-by-transfer-learning-in-keras_595289.

[4] Gamepedia. *Mods*. 2020. URL: https://minecraft.gamepedia.com/Mods.

[5] Long-Ji Lin. "Reinforcement Learning for Robots Using Neural Networks". PhD thesis. USA, 1992.

[6] Microsoft. *Malmo Enviroment*. https://github.com/Microsoft/malmo. 2019.

[7] Microsoft. *Project Malmo Documentation*. 2020. URL: http://microsoft.github.io/malmo/0.30.0/Documentation/index.html.

[8] MineRL. *MineRL: Towards AI in Minecraft*. 2020. URL: https://minerl.io/.

[9] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *CoRR* abs/1602.01783 (2016). arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783.

[10] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning." In: *Nature* 518.7540 (2015), pp. 529–533. URL: http://dblp.uni-trier.de/db/journals/nature/nature518.html#MnihKSRVBGRFOPB15.

[11] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[12] Silver D. et al Mnih V. Kavukcuoglu K. "Human-level control through deep reinforcement learning". In: *Nature 518* (2015), pp. 529–533. URL: https://doi.org/10.1038/nature14236.

[13] Deepak Pathak et al. *Curiosity-driven Exploration by Self-supervised Prediction*. May 2017. URL: https://arxiv.org/abs/1705.05363.

[14] Sebastin Ruder. *Transfer Learning - Machine Learning's Next Frontier*. [Online; accessed 06-Jan-2021]. 2017. URL: https://ruder.io/transfer-learning/.

[15] Szymon Sidor and John Schulman. *OpenAI Baselines: DQN - Benchmarks*. [Online; accessed 09-Jan-2021]. 2017. URL: https://openai.com/blog/openai-baselines-dqn/.

[16] Thomas Simonini. *An introduction to Deep Q-Learning: let's play Doom*. 2018. URL: https://www.freecodecamp.org/news/an-introduction-to-deep-q-learning-lets-play-doom-54d02d8017d8/.

[17] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[18] Chuanqi Tan et al. "A Survey on Deep Transfer Learning". In: *Artificial Neural Networks and Machine Learning – ICANN 2018*. Ed. by Věra Kůrková et al. Cham: Springer International Publishing, 2018, pp. 270–279. ISBN: 978-3-030-01424-7.

[19] TensorFlow. [Online; accessed 06-Jan-2021]. 2020. URL: https://github.com/tensorflow/models/tree/master/official/vision/image_classification.

[20] Tom Warren. *Minecraft still incredibly popular as sales top 200 million and 126 million play monthly*. 2020. URL: %5Curl%7Bhttps://www.theverge.com/2020/5/18/21262045/minecraft-sales-monthly-players-statistics-youtube%7D.