Multi-agent reinforcement and imitation learning in Minecraft

Marc Ducret

August 26, 2018

1 General direction

1.1 Simulating a society of learning agents

This project aims at simulating a society made of learning agents. Those agents would not obey to a centralized authority but instead organize themselves in a decentralized fashion. Human societies are complex structures emerging from individual behaviors and interactions. We would like to simulate such a society at the individual level and observe the emerging results at the collectivity level.

To achieve that we plan on learning from human play as a starting point and then leaving agents evolve with reinforcement learning. Evolving agents could be deployed in a society made of only autonomous agents or in some agent-human mixture to ensure they retain the ability to interact with humans.

1.2 Stakes for Machine Learning

While there has recently been a lot of research in single agent reinforcement learning or even simple multi-agent settings, those studies have not focused on whether structures among agents can emerge – can clans, leaders, social norms, trusts and many other critical aspects of human society appear from interactions of adaptive agents? Designing learning approaches that can evolve into and adapt to such societal phenomena seems key for autonomous agents to effectively interact with humans and operate without centralized control or incentives.

1.3 Understanding human societies

This approach could lead to a way of simulating human societies with few modeling assumptions. Indeed, if most of the behavior is learned from imitation the resulting simulation will only make weak assumptions about how humans behave. This suggests that those simulations could give genuine insights about real human societies.

Human societies have been studied in many branches of sciences such as economics, organization behavior, social science, network science, etc. However, to remain tractable mathematically, individual behaviors are often assumed to be similar to each other and the collective behaviors are often studies when the systems of such individuals reach equilibrium. It remains a large scientific challenge how to study evolving dynamical systems where individuals can be drastically different each other as each one of them has gained different behavior by adapting to environments and other agents' behaviors.

2 Project timespan and collaborators

Given its ambitious nature, the project is long term and this internship only lays the foundations. Brainstorming and general decisions involved other collaborators during various meetings. However, given their preoccupation with other projects, they will only start to actively contribute after this internship.

2.1 Resources relevant to this document

Getting an in depth understanding of this document might require taking a look at multiple resources. First of all, obviously, the cited papers, books and repositories are relevant. Then a few other resources can be very useful:

- The GitHub repository of my framework. It contains all of the Java code that constitutes the Minecraft side of the framework, the Python interface and some Python agent examples. The setup and how to run examples should be well detailed. A specific folder contains the animated figures referenced in this document.
- The Minecraft Wiki can be a good source of information concerning Minecraft's mechanics, even from a rather technical perspective.

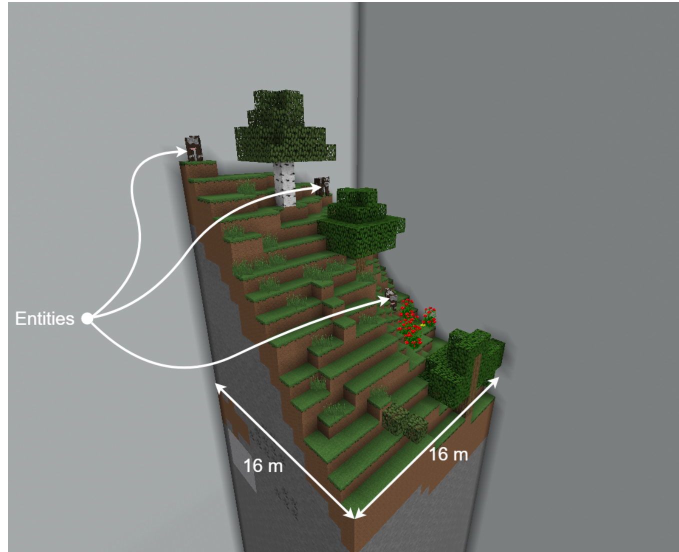

Figure 1: An isolated chunk with multiple blocks and 3 entities (cows)

2.2 Contributions made during the internship

My contributions are of three different natures that correspond to the three following sections. I implemented a learning framework in Minecraft that enables to use it for reinforcement and imitation learning with an eventually large amount of agents. Then, I conducted various preliminary experiments with this framework using different input spaces, environments, learning methods, network architectures... Finally, I explored how environments could lead to *societal* behaviors emerging and which of those behaviors could be interesting.

3 A learning framework inside Minecraft

3.1 Minecraft's architecture

To put choices made and difficulties faced into context, here is a quick introduction to Minecraft's architecture.

3.1.1 Simulation

Without taking networking, player inputs and rendering into account, a game engine can be seen as a simulation. The goal is therefore to predict the evolution in time and space of some objects according to some rules.

Objects are represented in the following manners:

- *Blocks* are simple and rarely change. They are scalable in terms of memory, simulation and rendering. It is crucial as they constitute the terrain. Usually they would occupy a full one meter cube and be opaque but there are exceptions.
- *Entities* are complex and evolve at each step. Storage, simulation and rendering are more costly but living creatures or any moving object would not fit under the *blocks*' limitations.
- *Block Entities* are an hybrid approach to *blocks* and *entities*. They allow a *block* to be extended by arbitrary memory storage and more sophisticated rendering like *entities*. However they are still expected to remain static. A chest for example is a *block* that requires special memory to remember which items are stored and special rendering to animate its opening.

Time would be simulated with steps of constant length (50 ms). Those updates are called *ticks*. A *client* can render multiple *frames* (what would be rendered to a player's screen) in between two ticks. At each of those *frames* the positions and rotations of moving objects would be interpolated. However this is not really important here as it doesn't affect the simulation.

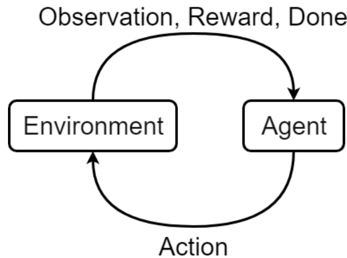

Figure 2: Interactions between an RL environment and an RL agent.

Representation in space is an important part of the engine as the world is pseudo infinite horizontally (more or less only limited by integer and floating point limitations). Therefore, the whole world cannot be loaded at once. Space is divided in *chunks* of 16 by 16 *blocks* or meters horizontally and up to the vertical limit (usually 256 *blocks*) as seen in Figure 1. A good approximation is to consider that only chunks within a certain distance of at least one player are loaded.

3.1.2 Networking

As any multiplayer game, Minecraft requires clever networking to deal with latency and desynchronization. First of all, there is one *server* per world and one *client* per player. The approach is a relaxed authoritarian server as most things are dictated by the server and clients simply follow, anticipate and correct. However, each client has authority on its player's movements and interactions (the server would still perform some checks to prevent getting an advantage by modifying the client). Doing so prevents players from feeling the latency when performing actions.

A consequence of such an approach is that there is no strong synchronization as at any time step, each client and the server will all observe slightly different states. The magnitude of those differences is hard to estimate and depend mainly on the different latencies between clients and the server. Another consequence is that the server only knows of the intermediate results of players' actions (for instance, the evolution of position instead of the keyboard successive states). This can be opposed to *deterministic lockstep* techniques where clients and the server would all wait for all clients inputs to simulate each time step in a deterministic fashion.

3.1.3 Modding

Mojang [1] does not provide Minecraft's source code or a modding API. Therefore, the community maintains tools to do so. Those I used are the most common: Minecraft Forge Mod Loader and Mod Coder Pack. *MCP* enables users to decompile Minecraft's client and server into readable and error-less code. *Forge* provides a module system that enables writing mods without modifying any source file and therefore combining mods that would otherwise modify the same files.

Writing mods in this manner is rather convenient but has limitations: the less usual source files do not have readable variable names or comments and implementing even simple features can sometimes require complicated tricks if those go against intended usage of Minecraft or Forge.

3.2 Reinforcement Learning framework

3.2.1 Setting

In a Reinforcement Learning setting, an environment and an agent interacts iteratively as described in Figure 2. At each step, the agent is provided with an *observation* vector, a *reward* scalar and a *done* boolean indicating that the environment terminated. The *observation* vector does not have to represent the whole environment state. In return if the environment is not *done*, the agent provides an *action* vector.

Given a *discount factor* $\gamma < 1$, a time step t and the future rewards $r(t+i)$ for $i \geq 0$, the *future discounted reward* is:

$$R_\gamma(t) = \sum_i \gamma^i r(t+i)$$

[1] The company behind Minecraft

Figure 3: 250 agents with a random policy on the same machine

Usually, the agent aims at choosing *actions* to maximize its *expected future discounted reward* $\mathbb{E}(R_\gamma(t))$. More precisely, it optimizes a *policy* π which is a mapping from the *observation space* to the *action space* (eventually with some memory) such that when interacting with the environment according to this *policy*, its *expected future discounted reward* is maximal.

For multi-agent settings, a simple approach is to consider that the previous setting is the perspective of each agent and other agents are 'hidden' within the environment. More sophisticated approaches would specifically identify other agents and their actions and/or observations to take into account the fact that they too are learning.

3.2.2 Environments in Minecraft

In my framework, environments are implemented within the server. To define a new type of environment, one must provide a list of *sensors*, a list of *actuators*, a *step* function and a *reset* function. *Sensors* and respectively *actuators* are objects that define parts of the *observation space* respectively *action space*.

A single world can contain multiple *environments* simultaneously. To instantiate an environment, the type, location in the world and type-dependent parameters must be provided. If the no particular location is desired, the environment can be automatically allocated an area avoiding other allocated environments.

Multiple *actors* can share the same environments. Those *actors* can be *humans* or *agents*. However, different action and observation space do not have any effect on humans.

3.2.3 Agents in Minecraft

The goal here is to have agents in Minecraft controlled by a Python program. The two main constraints are that agents should have similar possibilities and limitations as humans and the approach should be scalable to a large number of agents.

Previous approaches like Project Malmö [7] implement such agents within Minecraft's client. It can therefore easily provide agents with almost the same interface as humans. However, from the perspective of Figure 2 it means that network lag is part of the environment. Moreover, since each agent requires its client which is heavy in memory, computation and rendering, scalability to even 50 agents requires multiple machines. Simulating at a higher frequency to learn faster would also make network lag more noticeable.

My solution to this was to implements agents directly within the server. The main drawbacks are that agents cannot use Minecraft rendering as input space and having the interface aligned with what human have is more complicated. Apart from that, we don't have to deal with network lag and agents are much more scalable as they all share the same Minecraft instance, the server. Figure 3 would not be possible with Malmö on a single machine as it would require at least 500 GB of RAM.

As described in Figure 4, at each time step of an environment, Python agents are sent their current observation and reward. Then the server waits for them to respond with an action. This synchronous approach makes execution speed irrelevant to an environment behavior. I was there-

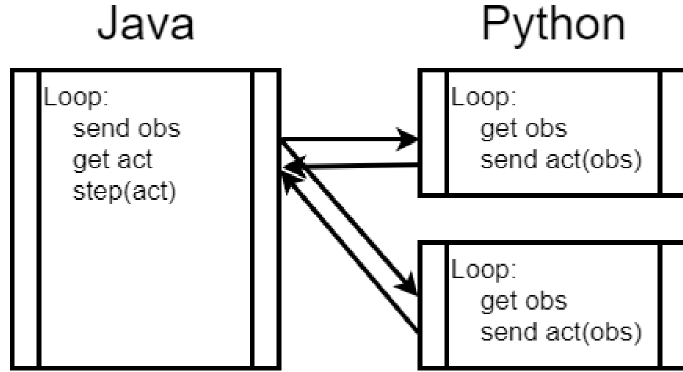

Figure 4: Communication between the server and Python agents

```
dummy = MinecraftEnv(type) # Allocating a new environment
dummy.init_spaces()
dummy.close()
envs = [MinecraftEnv(dummy.env_type, dummy.env_id) for _ in range(N)]
```

Figure 5: Allocation of N agents in the same environment of type *type*

fore able to implement an option in Minecraft's server to simulate as fast at it cans while removing costly features we don't use.

3.2.4 Python interface

From the Python perspective, environments are seen through OpenAI's Gym interface [2] in order to be rather easily compatible with existing Reinforcement Learning algorithms implementations.

To instantiate an agent in a new or existing environment a TCP connection is initiated with the Java server. This enables starting and stopping environments only from Python. Moreover, multiple agents can run within the Python process. That can be especially useful for single agent algorithms collecting experience from multiple independent environments to learn a single policy [11] or multi agent algorithms that share information across agents [10]. Figure 5 shows how a multi agent environment can be instantiated in Python.

3.2.5 Technical problems

In Minecraft's server, player entities are handled in very different manners from other entities since they are expected to be controlled from a client through a network connection. Implementing the agent variant of the player entity therefore required altering the player class in very specific manners.

3.3 Imitation Learning Framework

3.3.1 Setting

Here, we call imitation learning the process of learning a *policy* for an *environment* using a set of *demonstrations* instead of the rewards. A *demonstration* is a whole episode of a policy interacting with the environment. One way of representing a demonstration is the sequence of observation-action pairs indexed by the step number t: $((o_t, a_t))$.

Assuming that all demonstrations are sampled from a single policy $\pi^\star$, the goal is to learn a policy π to approximate $\pi^\star$. One way of approaching the problem is to simply do supervised learning using the demonstrations as data to learn π. However, here are two concerns that could lead to issues:

- The simplest approach is to consider $\pi^\star$ as the sum of a deterministic policy and some gaussian noise. In that case, π would only learn the mean. However, in some situations, two different actions could be equally viable. For instance, if the goal is to find an object that

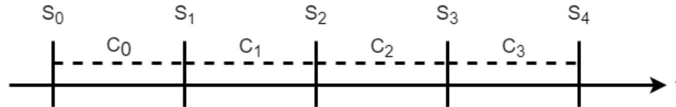

Figure 6: Snapshots and change-sets dispositions in time

does not appear on the screen, the human has two equally viable choices: looking left (-1) or looking right ($+1$). If the human is not biased towards one the options, the learned mean would be 0 even if 0 never appears in the data.

- After learning, when π is used in the environment, errors can accumulate and lead to unexplored space. On the other hand, during training, errors do not accumulate as the sequence of observations does not depend on the predicted actions.

3.3.2 Recording human play

First, I had to implement some way of recording demonstrations. To avoid situations where we would have to redo the recording when going through changes to the observation and action space, we decided that the recordings should be as independent from those as possible. This is possible since human players are not affected by those spaces.

Therefore, the record feature aims at reconstructing a sequence of blocks and entities configurations with as much fidelity as possible.

Within a specified area, at some time step t, let w_t be the configuration of blocks and entities. Instead of storing every w_t, we can store w_0 and then at each time step t, the changes to apply c_t. Then $w_{t+1} = f(w_t, c_t)$ where f is easy to compute. That way, even large areas can be efficiently recorded.

For long recordings, loading a specific w_t could take long if t is large. To prevent this, w_t is periodically saved. Let T be this period, then the following are stored as files (see Figure 6):

- *Snapshots* $S_i = w_{iT}$
- *Change-sets* $C_i = (c_t)_{iT \leq t < (i+1)T}$

This way getting to any time step of the recording requires at most loading one w and applying T times f.

Player entities are treated a bit differently so that their configuration contains information about the environment they are in (*reward* and *episode end*) and about the actions they just performed (placing a block for instance).

3.3.3 Replaying a recording

In order to verify the integrity of recordings, I implemented a graphical interface to replay a recording. It enables users to inspect a recording from a player's perspective or with a free camera. It seems that the best way to do so with as much visual fidelity to Minecraft is to reuse its renderer.

To achieve this, I had to replay the recorded events within a modified Minecraft *World* object. It contains storage for *blocks* and *entities* and a lot of various details such as blocks light levels, chunk loading or procedural terrain generation. However, Minecraft makes a lot of assumptions concerning the worlds to render using its renderer and some of those cannot be verified with such a '*fake*' world. I therefore had to use a lot of dirty tricks to get it working. The worst of those is probably disabling the multi-threaded computation of chunks meshes: there is no Forge hook to do so, therefore I had to modify one of Minecraft's base classes and doing so with the Forge framework require editing the class bytecode at runtime. Thankfully I only had to replace one method with: '*return true;*' which is only two bytecode instructions.

One natural way of testing the recording and replaying is to compare them with client side screen recording. This experiment can be seen in Animated Figure 1 The increasing delay probably results from slightly different speed of doing *ticks*. A few details are not recorded such as particles or the door's state.

Then I implemented a tool to make an observation-action dataset out of a recording. Doing so is fairly straightforward, the recording is replayed within a *'fake'* world as before and a given

Figure 7: PARKOUR environment: the agent must go from the blue block to the green block without falling.

environment type is used to compute observations and actions for each time step and player entity. The only difficulty is computing actions which can be different for each action type. For instance, computing movement actions requires computing the derivative of a player's successive positions and rotations.

3.3.4 Loading an environment in a specific state from a recording

Approaches such as Scheduled Sampling [1] or Backplay [13] requires the ability for an environment to load a specific state and continue simulation from that point.

Implementing this required a few adjustments for the replay to use the real Minecraft world and use the players entities of agents already within the environment. It was also necessary for episodes resets and loading to be controlled from Python. The solution I went for is another TCP connection through which Python provides one command per step among *continue*, *reset* and *load(t)* after being informed of weather the environment reached a terminal state.

4 Experiments

4.1 Basic RL

The simplest approach to Deep Reinforcement Learning is probably Deep-Q-Learning. It aims at learning a Q-function $Q^\star : \mathcal{O} \times \mathcal{A} \rightarrow \mathbb{R}$ where $\mathcal{O}$ is the *observation* continuous space and $\mathcal{A}$ is the *action* discrete space, such that $\pi^\star : o \mapsto argmax_a Q^\star(o, a)$ is optimal. To do so it updates Q after each step towards the *discounted future rewards* after choosing action a given observation o. Exploration can be handled by, at each step, with probability ϵ (that can change during training) choosing a random action and choosing the best action according to Q otherwise.

For continuous action spaces, $argmax_a Q(o, a)$ cannot easily be computed. Therefore two functions are learned, the value function $V : \mathcal{O} \rightarrow \mathbb{R}$ and the policy $\pi : \mathcal{O} \rightarrow \mathcal{A}$. Such a model is often called '*actor-critic*'. Most of the time, π would be stochastic instead of deterministic. In that case, the conditional distribution $\pi(a|o)$ would be learned, eventually making assumptions such as the distribution of actions being *gaussian* given an observation. Those functions would be optimized using *policy gradient* methods. More details can be found in [17] (except they consider full observability so they would talk about *states* instead of *observations*).

In my experiments, I mostly used *Proximal Policy Optimization* (PPO) [14] and *Advantage Actor Critic* (A2C) which is OpenAI's synchronous version of *Asynchronous Advantage Actor Critic* (A3C) [11]. Both of them would be policy gradient methods. I used implementations from [8] and [4].

One of the environments that I used to test the framework is the PARKOUR environment as seen in Figure 7. A path of white blocks is randomly generated from a blue block to a green block. The agent must go from the blue to the green without falling. Actions are the horizontal

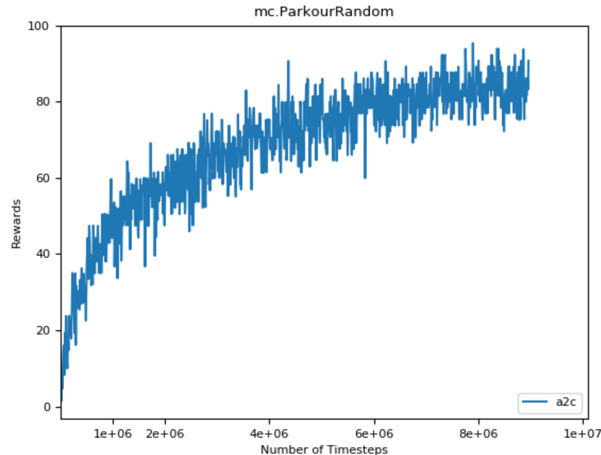

Figure 8: Reward as a function of training time in the Parkour environment using A2C

velocity (2 dimensions) and observations are made of the position relative to the blue block and an encoding of the surrounding blocks in a 7 by 7 square centered on the agent (empty is -1, green is 10 and others are 1). The rewards are: when finishing on the green block, 100, when falling, 0 to 10 depending on the distance and -0.001 otherwise. Figure 8 shows the mean reward per episode while training. The final policy can be observed in Animated Figure 2.

4.2 Observation Spaces

Designing the observation space is a crucial part of solving a task with machine learning. We will focus on the general structure which would describe what symmetries can be used within the neural network to reduce the number of parameters.

4.2.1 Blocks

In most environments, agents should be aware of the blocks surrounding them. If this is represented by some vector in $\mathbb{R}^n$ and fed to a fully connected layer, what is learned about the state of a block at a position relative to the agent cannot be easily transferred to other positions.

A solution, is to represent those blocks in some multi-dimensional array in $\mathbb{R}^{xyzc}$ where (x, y, z) is the size of the box and c is the dimension of the encoding of one block. This can be fed to some convolutional neural network which makes learning position invariant or covariant behaviors a lot easier.

4.2.2 Entities

As for blocks, perceiving multiple entities requires some invariance properties on the neural network. However, another issue is that the number of entities in sight can vary. Both problems can be addressed with attention mechanisms similar to [3].

The principle is to compute a score for each entity and then fed the rest of the network only the best scoring entity's representation. If each entity is represented by a vector in $\mathbb{R}^c$, E is the set of such representations observed and $s : \mathbb{R}^c \to \mathbb{R}$ is the learnable scoring function, the output of the attention layer is:

$$a(E) = \frac{\sum_{x \in E} e^{s(x)} x}{\sum_{x \in E} e^{s(x)}}$$

If multiple entities can be relevant to decisions making at the same time, multiple such attention layers can be concatenated with different scoring functions.

To demonstrate when this could be useful, I designed the CowArena environment: Figure 9. Each cow is encoded as a 3-dimensional vector made of the 2D position of the cow relative to the agent and weather the cow is dead or alive. The action space is the 2D velocity of the agent. The

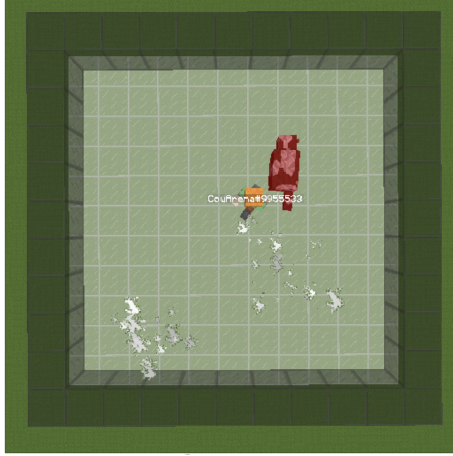

Figure 9: CowArena environment: the agent must touch all cows to kill them.

	Without attention	With attention	With attention, learned with 1-2 cows
1-1 cows	24	24	24
1-2 cows	26	27	27
1-20 cows	36	59	50

Figure 10: Mean reward per episode of different approaches in CowArena environment.

agent is rewarded 5 for each cow killed, 20 for killing all cows (terminating the episode) and -0.1 per time step. The environments terminates after at most 100 steps.

Figure 10 suggests that using such an attention layer allows learning in a simpler setting and generalizing to a more complex one. In each episode, the amount of cows is random within the setting's bounds (1-1, 1-3 or 1-20). Since the observation size must be constant without the attention layer, cows that did not spawn are considered dead. In each case, training lasts for 200 000 time steps and uses PPO. The rightmost column trains a single policy on the 1-2 cows setting and tests it on every setting. Also, learning in the 1-20 cows setting after initializing from the 1-2 cows attention policy leads to 68 mean reward which beats learning straight in the 1-20 cows setting.

The policy learned with an attention layer in the 1-2 cows setting can be seen in Animated Figure 3 (same as training setting) and Animated Figure 4 (transfer to 1-20 cows setting).

4.2.3 Vision

A general approach to Minecraft would require some combination of both previous approaches. However doing so seems to have two main drawbacks. First, general interactions in Minecraft require a precise control of the agent's orientation[2]. It seems that learning such precise control would be a lot easier if observations are relative to orientation. Entities' positions could be expressed relative to the agent's local coordinates but it does not seem that there is a nice way of doing so for blocks. Then, such input spaces are very different from what a player would see. Objects hidden by another would still be considered and the agent's orientation is not relevant to what is seen or not.

Using vision seems to solve both of those problems. However, detecting objects using Minecraft's renderer is a challenge in itself and since agents run on the server they haven't even access to rendering. My solution to that is to create custom rendering based on simple *ray tracing* that would create images in a *machine friendly* encoding. Ray tracing, as illustrated by Figure 11, simply computes the intersections of multiple rays with the world. Each ray is associated with a pixel and the values assigned to the pixels are based on the point hit by the ray. All rays start from the viewer's eyes and have directions slightly different from the viewer's orientation, depending on the pixel. Objects' bounds are defined as their collision boxes which are available on the server.

[2]The location of interactions is decided based on where a player looks by computing where a ray starting for the eyes intersects the world.

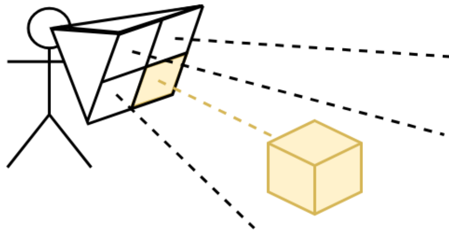

Figure 11: Ray tracing to create an image made of 4 pixels.

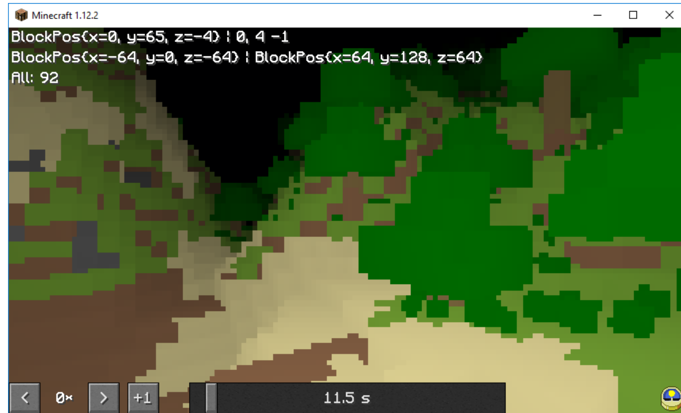

Figure 12: A typical Minecraft landscape viewed with this raytracing.

Creating an input space with this general approach requires specifying a resolution, a field of view and an encoding. The encoding can have an undefined amount of *channels*[3] and associates one value to each of those based the object that was hit by the ray. Figure 12 is an example of an *RGB* encoding where each block is assigned one color and colors are darkened based on the distance of the hit.

I tried to measure the performance cost of such ray tracing. The parameters significant to performance are the resolution and the maximum depth of rays. Figure 13 shows that with a resolution of 32×32 and rays of at most $60\,m$, computing all rays should take about $4\,ms$. This seems acceptable as resolutions of vision in RL are usually not bigger than that. However it could become a problem to scale to a huge number of agents. The computation of one ray's intersection with the world is already implemented in Minecraft and is fairly efficient because of the cubic structure of the world.

4.3 Imitation

4.3.1 Team game environment

A natural step towards agents in a human like society is experimenting with a team game environment with collaboration and competition. This environment should be interesting for human players and also not too hard for agents to learn. To be a nice test case for the framework, it should also use Minecraft's building mechanics.

Based on those requirements, I designed the PATTERN environment seen in Figure 14. Players find themselves assigned a colored team (*yellow* or *blue* if there is two teams, but as in Figure 3,

[3]The number of values per pixel

Situation	Computation time per ray
In front of a block	$0\,\mu s$ (insignificant)
Typical	$4\,\mu s$
Looking at the sky	$6\,\mu s$

Figure 13: Time required to compute the hit of a ray with a maximal distance of $60\,m$

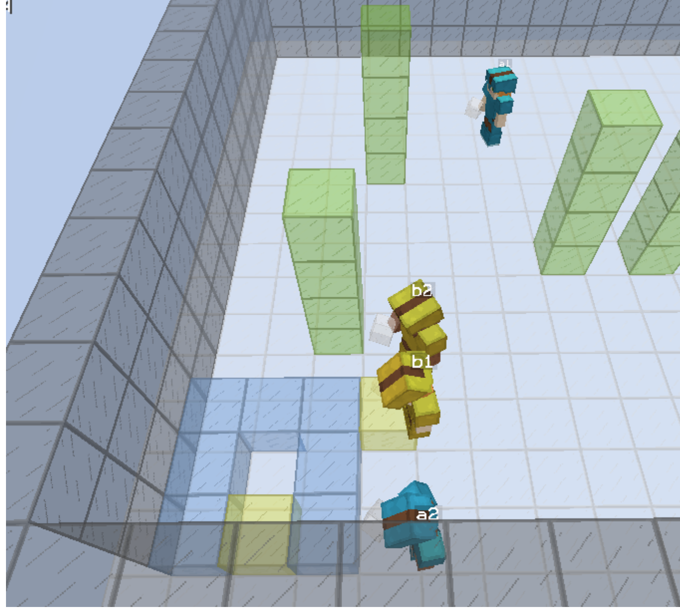

Figure 14: PATTERN environment. The yellow team just stopped the blue team from successfully building the 3×3 pattern.

there can be many more). A team wins when the *arena* contains a predefined *pattern*[4] made out of this team's color. To do so players collect blocks by breaking the green *trees*. For each broken tree, the player that broke it receives some quantity of his color blocks and another tree grows at a random free position. Blocks can only be placed at the surface of the ground and only *trees* can be broken. If no team has won after some time limit, the episode is considered a *draw*.

During our human experiments, we found the following configuration to be rather well balanced and fun at least for inexperienced players:

- 15×15 blocks arena not counting walls.
- 2 teams of 2 players
- 4 trees at all times, breaking one gives 2 blocks
- 3×3 square with a hole in the middle as pattern
- 2 minutes time limit (2400 steps)

We speculate that it might be too easy for an experienced player to ensure a draw by blocking all attempts by the opposing team but we have not yet observed such behavior.

4.3.2 Encodings and architecture

In the PATTERN environment, observations results from the ray tracing vision approach. A $24 \times 12 \times 7$ image is created. Each of the 7 channels has values in $[-1, +1]$ or $[0, +1]$. These are illustrated in Figure 15 at a higher resolution and with *blue* being -1 and *red* being $+1$. A human friendly *RGB* reconstruction can be computed using Team, Sky and Tree channels for color and Depth and Face normal for brightness.

Actions are represented by 6-dimensional vector:

- 2 dimensions for *(forward, strafe)* velocity (in $[-1, +1]$)
- 2 dimensions for *(pitch, yaw)* angular velocity (in $[-1, +1]$)
- 2 dimensions for respectively breaking and placing blocks (as a probability in $[0, 1]$)

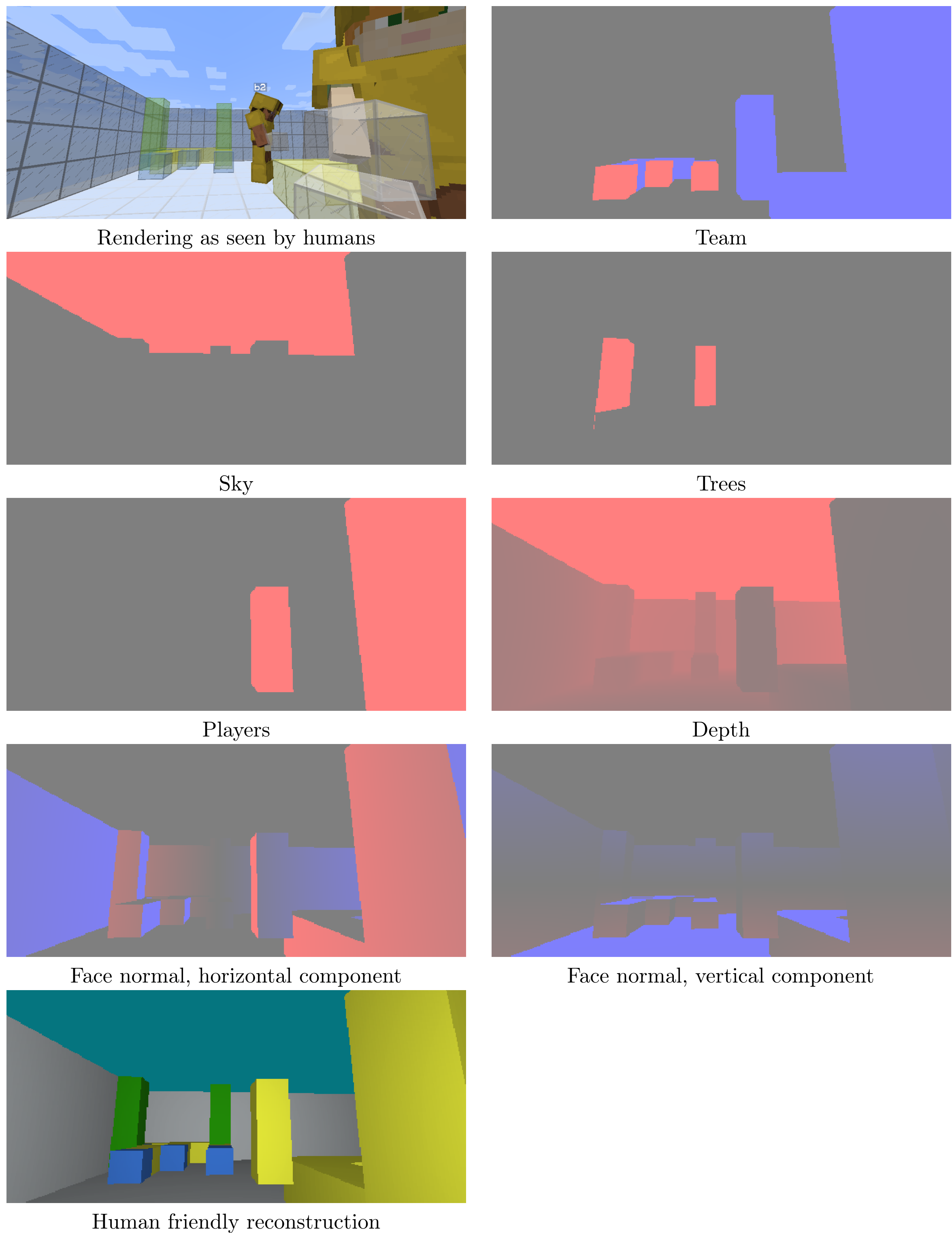

Figure 15: Observation vision encoding with 7 channels for PATTERN environment at a high resolution.

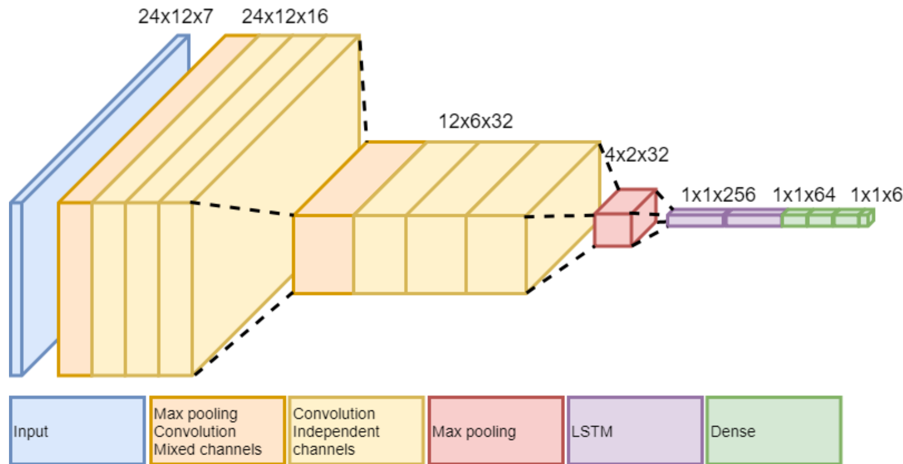

Figure 16: Network architecture for the PATTERN environment. Each box represents one of the successive arrays of values and are colored according to the operations used to compute it based on the previous layer.

The neural network architecture is described by Figure 16. The convolutional part is mainly inspired from VGG [16]. Every convolution has a 3×3 kernel, uses reflection padding and is followed by *tanh* nonlinearity. Although there is still room from improvement, I tuned this architecture empirically. One of the surprises is that *tanh* seems to yield better results than *ReLU*. I suspect this comes from our encoding differing a lot from natural images (on which *ReLU* seems to usually do better).

It seems that memory is necessary to mimic human behaviors in this environment as it is only *partially observable* and humans tend to commit to decisions. Moving from one point to another requires many steps, therefore some *long term* memory might be necessary. We choose to use *Long-Short Term Memory* layers [6] for this purpose. Those are extensively used in *Natural Language Processing*. It might be worth exploring other alternatives [12] in the future.

4.3.3 Testing with a simple policy

Before using extremely noisy human data, I used data from a simple policy I designed. This policy simply aimed at breaking trees. This process enabled me to fine tune the architecture and correct mistakes in my implementation. Also, I observed some overfitting by computing the imitation loss on some validation data. Adding weight decay successfully dealt with that.

The final results of this experiment is a policy (Animated Figure 6) that a human would not easily distinguish from the original policy (Animated Figure 5).

4.3.4 Human data

Imitation from humans is a lot more difficult for the following reasons:

- Policies have a lot of noise.
- Reactions are delayed.
- Aside from the noise, policies are still random.
- Having enough data requires learning from different humans with different policies.
- Even a single human's policy evolves as he learns more about the game.

We held multiple recording sessions where we asked lab-mates to play. This added up to 80 episodes with 4 players each, about 1 hour of recordings and 250.000 observation-action pairs. Some small sample of this data can be seen in Animated Figure 7.

After fine tuning I was able to get a policy that seems *locally* human illustrated by Animated Figure 8. While short term actions are similar to what human subjects did, the policy lacks

[4] A specific disposition of blocks in space such as a square or a cross of some specific size

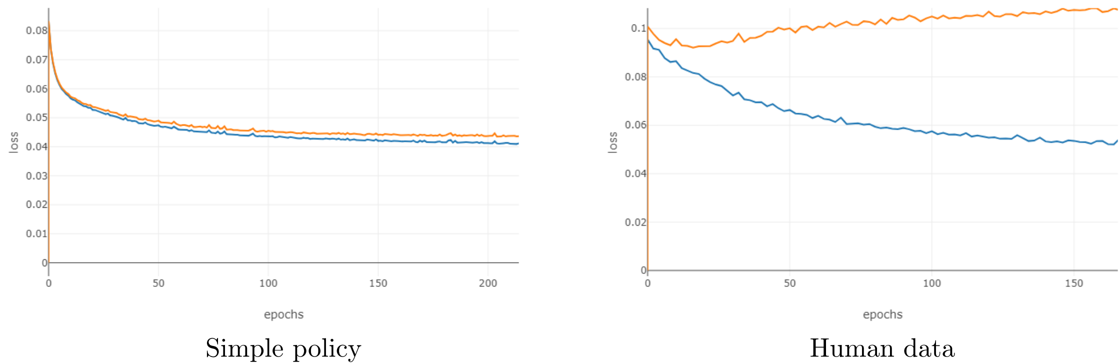

Figure 17: Training (blue) and validation (orange) losses on imitation while imitating the simple policy or from human data.

medium and long term decision making and doesn't seem to really take into account the blocks already placed.

One surprise is that learning from human data lead to a lot of overfitting as shown by Figure 17 but mitigating it with more weight decay or early stopping resulted in significantly less human-looking policies. It therefore seems that our validation loss is not aligned with our qualitative objective of a human-looking policy. It might highlight a lack of data, issues with our architecture or training approach. It also raises the question of finding a better evaluation technique.

4.4 Improving upon imitation

After imitation, one natural follow-up is to use the policy as initialization for some Reinforcement Learning algorithm. However, this policy seems to not be good enough and almost never leads to a completed pattern and therefore never gets reward (random exploration is fairly hard in this environment and rewards are very sparse).

To make rewards easier to grab, one idea is to start agents in the state that is close to winning. Once they learned enough from here they can start further from winning, until using the initial state. Essentially, this is Backplay [13] which consists in starting the environment in an intermediate step of some human trajectory. This step is chosen closer and closer to the initial state as training continues.

I implemented this procedure on top of PPO to optimize the policy obtained by imitation. One detail was how to initialize the LSTM memory. It seems that the most natural way of doing that is to use the human trajectory observations until that point.

I did not have time to make it successfully enhance the policy. However, it confirms that my framework can support such algorithms. Loading a state from a trajectory took around 50 *ms* in my testings, which is reasonable.

5 Ideas going forward

5.1 Interesting societal environments

5.1.1 Societal structure

Human societies are complex systems and we should focus only on some of the most fundamental and interesting phenomena. Here, we will consider a society as a collection of individuals where some of the following behaviors are observed:

- Some *common* interests are emerging from individual interests.
- There is some *trust* among its members.
- Concepts of *property*, *trade* and *currency*.
- Work is allocated based on *skills* instead of *needs*.

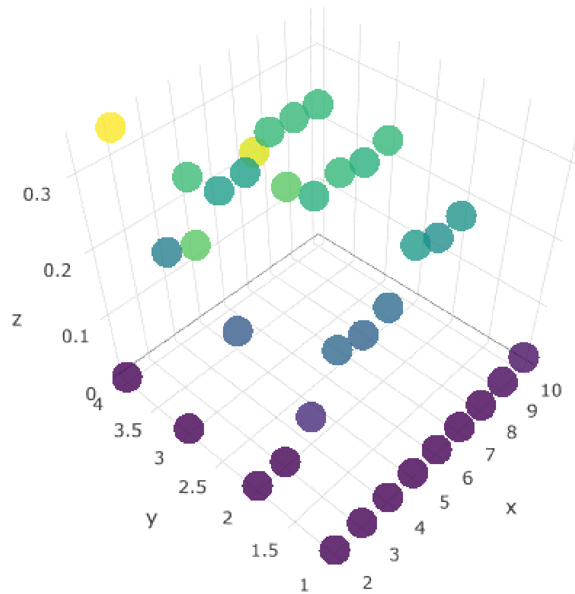

Figure 18: z is the ratio of successful cooperation in the Trust environment with $R = y$ and $N = x$ independent learning agents. Points are colored according to their z value.

- Members conform to some *norms*.

I can see two interesting approaches for such a structure in machine learning. One would be to study what environments and algorithms can lead to the emergence of such a structure. Another would be how to have agents learn how to successfully interact with a preexisting structure of this kind.

5.1.2 Related work

Some of the stated phenomena have already been studied in a context of learning agents. [5] proposes a learning algorithm, LOLA, that anticipates the learning of other agents and can learn collaboration in the iterated prisoner's dilemma. [9] studies sequential social dilemmas and how the difficulty of learning *cooperation* and respectively *defection* is relevant to the modeling of such dilemmas. Finally, [15] shows how *norms* can emerge through interactions of random individuals of a population.

5.1.3 Emergence of trust

I quickly designed a simple iterated prisoner's dilemma style environment: Trust. At each step, each of N agents interacts with every other agent choosing to *cooperate* or *defect*. Cooperating costs $\frac{1}{R}$ reward but yields the other agent 1 reward. The policies are made of one *multilayer perceptron* made to process one interaction with one other agent. It computes what action to choose and a *reputation* scalar based on the opponent previous action and the average of previous recent *reputation* outputs. This reputation acts as some kind of simple memory per other agent, handled by the environment.

In Figure 18, I studied how often both agents cooperate based on the cooperation return R and the number of agents N. When $R = 1$, the game is zero-sum and always defecting would be expected. When $N = 2$, always defecting confirms the results of [5]'s baselines with independent learners. However what is surprising is how when $R > 1$ and $N > 2$ it seems that agents converge to a policy that uses cooperation.

I did not have time to investigate this phenomenon in a deep and rigorous manner, therefore, I don't know how reliably this can be achieved and weather the learned policies are exploitable. Doing so in the future would help figuring out how agents can learn cooperative behavior without

Figure 19: The user interface when opening a chest.

complex opponent modeling. It also aligns to some extent with studying what holds our societies together and prevents too much chaos.

5.2 Some flaws of the framework

5.2.1 Timeouts

Currently, the Minecraft server detects terminations of agents when their associated TCP connection times-out. Meanwhile, python scripts initializations and policy updates can take more than a few seconds and are highly dependent on hardware and hyper-parameters. Therefore timeout durations are hard to adjust and can lead to timeouts during a long training sessions.

At least those durations should be adjustable from the python side at start. Ideally there should be a clean exit procedure so that we can have no timeout limit or a very long one.

All of this also applies to environments' TCP connection to decide when to *reset* and *load*. Moreover requests to join or create environments are treated simultaneously if they occur during a certain time-frame in order for multi-agent environments to be populated with synchronized agents. Those delays create the same kind of issues and should also be dealt with.

5.2.2 Large datasets

The size of observation-action pairs datasets is limited in two ways.

First when computing such a dataset from a recording, I made the assumption that the whole dataset would fit in RAM. Progressively writing the file would remove this constraint. Large datasets might require splitting them in multiple files. Currently, datasets files are not compressed, different compression methods should be tried to see if it can make reading them faster.

Then, my supervised learning implementation assumes that the dataset fits in the GPU's memory. This limits how much data we can learn from. Removing this constraint could be done by dynamically updating the trajectories stored in memory. Specific attention should be given to execution speed as those memory operations could have significant impacts.

5.2.3 Inventories

Another flaw of my framework is that it doesn't support anything related to inventories or containers. This means that agents cannot reorganize their inventory, interact with chests, craft objects...

The main issue is that *Forge* does not provide events to detect when a player interacts with an inventory. Therefore while recordings describe the contents of inventories at every frame, changes are not attributed to a player. Detecting this would require altering a lot of Minecraft's base classes and doing so is difficult while using *Forge*.

Another problem, is that I was not able to figure out a machine friendly encoding of inventory related interactions for agents.

References

[1] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 1171–1179, 2015.

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

[3] Jinyoung Choi, Beom-Jin Lee, and Byoung-Tak Zhang. Multi-focus attention network for efficient deep reinforcement learning. 2017.

[4] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Openai baselines. https://github.com/openai/baselines, 2017.

[5] Jakob Foerster, Richard Y Chen, Maruan Al-Shedivat, Shimon Whiteson, Pieter Abbeel, and Igor Mordatch. Learning with opponent-learning awareness. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 122–130. International Foundation for Autonomous Agents and Multiagent Systems, 2018.

[6] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[7] Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. The malmo platform for artificial intelligence experimentation. In *IJCAI*, pages 4246–4247, 2016.

[8] Ilya Kostrikov. Pytorch implementations of reinforcement learning algorithms. https://github.com/ikostrikov/pytorch-a2c-ppo-acktr, 2018.

[9] Joel Z Leibo, Vinicius Zambaldi, Marc Lanctot, Janusz Marecki, and Thore Graepel. Multi-agent reinforcement learning in sequential social dilemmas. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pages 464–473. International Foundation for Autonomous Agents and Multiagent Systems, 2017.

[10] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems*, pages 6379–6390, 2017.

[11] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[12] Junhyuk Oh, Valliappa Chockalingam, Satinder Singh, and Honglak Lee. Control of memory, active perception, and action in minecraft. *arXiv preprint arXiv:1605.09128*, 2016.

[13] Cinjon Resnick, Roberta Raileanu, Sanyam Kapoor, Alex Peysakhovich, Kyunghyun Cho, and Joan Bruna. Backplay:" man muss immer umkehren". *arXiv preprint arXiv:1807.06919*, 2018.

[14] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[15] Sandip Sen and Stéphane Airiau. Emergence of norms through social learning. In *IJCAI*, volume 1507, page 1512, 2007.

[16] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[17] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*. In progress, second edition, 2018.