ENVIRONMENTAL CURRICULUM LEARNING FOR EFFICIENTLY ACHIEVING SUPERHUMAN PLAY IN GAMES

BY

RAY SUN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Adviser:

Professor Jian Peng

ABSTRACT

Reinforcement learning has made large strides in training agents to play games, including complex ones such as arcade game Pommerman and real-time strategy game StarCraft II. To allow agents to grasp the many concepts in these games, curriculum learning has been used to teach agents multiple skills over time. We present Environmental Curriculum Learning, a new technique for creating a curriculum of environment versions for an agent to learn in sequence. By adding helpful features to the state and action spaces, and then removing these helpers over the course of training, agents can focus on the fundamentals of a game one at a time. Our experiments in Pommerman illustrate the design principles of ECL, and our experiments in StarCraft II show that ECL produces agents with far better final performance than without it, when using the same training algorithm. Our StarCraft II ECL agent exceeds previous score records in a StarCraft II minigame, including human records, while taking far less training time to do so than previous approaches.

To Mom, Dad, and my sister, for their love and support.

ACKNOWLEDGMENTS

Many thanks to my undergraduate research partners, Michael McGuire and David Long, for their ideas and code contributions. Thanks to Professor Peng for his advising and support.

This work utilizes resources supported by the National Science Foundation's Major Research Instrumentation program, grant #1725729, as well as the University of Illinois at Urbana-Champaign.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 BACKGROUND 2
2.1 Reinforcement Learning 2
2.2 Pommerman 3
2.3 StarCraft II 4

CHAPTER 3 RELATED WORK 7
3.1 Curriculum Learning 7
3.2 Pommerman 7
3.3 StarCraft II 7

CHAPTER 4 METHODS 9
4.1 Environmental Curriculum Learning 9
4.2 Our Application of ECL 11
4.3 Code Architecture 12

CHAPTER 5 EXPERIMENTAL SETTINGS 16
5.1 Pommerman 16
5.2 StarCraft II 21

CHAPTER 6 RESULTS 27
6.1 Pommerman 27
6.2 StarCraft II 31

CHAPTER 7 DISCUSSION 37
7.1 Implications 37
7.2 Limitations 37
7.3 Future Work 38
7.4 Conclusion 38

REFERENCES 39

CHAPTER 1: INTRODUCTION

One of the fastest-moving branches of machine learning today is reinforcement learning. Recently, several breakthroughs have been made in creating models to play games of increasing difficulty, from the ancient but deep board game of Go to the team-based video game Dota 2 [1, 2]. One of the greatest challenges yet is the real-time strategy game StarCraft II, which has incredibly complex state and action spaces. Many different approaches have been tried to address this challenge, but almost all of them require training for millions of episodes. In this work, we present Environmental Curriculum Training (ECL), a new method of curriculum learning. Making the environment easier for the agent by adding information in the state interface or simplifying the action interface, we can create a series of environment versions as a training curriculum, from the easiest version to the final, original problem. Our experiments in two games, Pommerman and StarCraft II, demonstrate that curriculum learning leads to enhanced final performance, even using training algorithms that make no progress at all on their own. Moreover, we show that environmental curriculum training is sample efficient, reaching previous record scores with far fewer training episodes.

Chapter 2 introduces background concepts for understanding this paper. Chapter 3 discusses previous work related to this area, and how this work differs. Chapter 4 describes our general methodology of environmental curriculum learning. Chapter 5 includes the details of our experimental setup in Pommerman and StarCraft II. Chapter 6 contains the results of our experiments in Pommerman and StarCraft II. We conclude in Chapter 7 by discussing the implications of our results.

CHAPTER 2: BACKGROUND

This thesis presents a new technique in reinforcement learning, one of the most active fields of machine learning research. The experiments that support the efficacy of this technique were performed in two environments: the games Pommerman and StarCraft II. In this chapter, the background information necessary for understanding this work is introduced: basic concepts in RL, and the rules and features of Pommerman and StarCraft II.

2.1 REINFORCEMENT LEARNING

Reinforcement learning (RL) is a branch of machine learning. In reinforcement learning, we are concerned with training an *agent* that interacts with an *environment* to maximize *reward*. An *agent* is an actor that interacts with the environment, and can be formulated as taking the current *state* as input and returning a chosen *action* for that state. An *environment* can be formulated as its counterpart function, giving a reward value and the next state after the given state and agent's chosen action. Generally, an environment generates multiple states in a sequence, each part of a *timestep*, as the agent chooses an action based on the state of each timestep. Each of these sequences is called an *episode*, starting with a *starting state* that does not depend on any previous state and ending with a *terminal state*.

The agent's goal in an environment is defined to be maximizing its total *discounted* reward for each episode. Rewards gained in the future are *discounted* when evaluated in the current timestep, by being multiplied by a *discount factor* for every step removed that they are. That is, for a given discount factor γ, current timestep i, and a sequence of future rewards $r_i, r_{i+1}, \ldots, r_n$, the agent's objective is to choose the action at timestep i that maximizes

$$J = r_i + \gamma r_{i+1} + \gamma^2 r_{i+2} + \ldots + \gamma^{n-i} r_n = \sum_{j=i}^{n} \gamma^{j-i} r_j \tag{2.1}$$

To create agents that maximize reward, reinforcement learning studies algorithms for how to *train* agents, updating them based on past episodes and improving their reward over time. Usually, these agents include *neural networks*, layers of computations and weights for those computations. The agent improves by updating these weights: they are *optimized* by computing a *loss* function based on observed episodes and adjusting the weights in a process called *stochastic gradient descent*.

The neural networks in this work use 4 kinds of layers: convolutional, fully connected, batch normalization, and ReLU. A convolutional layer processes a 2D input by multiplying

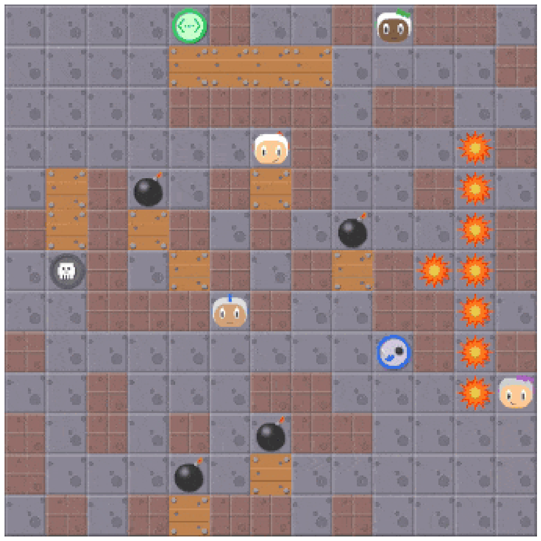


Figure 2.1: A Pommerman match between 4 players. Four bombs are currently planted, and two have already exploded in a chain reaction and left flames behind on the right side. There is a green blast range power-up near the top left, and a blue kick power-up near the bottom right.

it with a smaller square of weights called a kernel, which is slid across the input matrix to make all the multiplications. A fully connected layer yields a given number of outputs, each as a linear function of all the inputs and layer weights. Batch normalization normalizes the means and variances of its inputs when optimizing the network [3]. A ReLU (rectified linear unit) returns $\max\{x_i, 0\}$ for each element of a vector $\mathbf{x}$. Also, we use the *softmax* activation function to normalize mutually exclusive probabilities, which is given by

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{2.2}$$

for a given vector $\mathbf{z}$ of unnormalized probabilities.

2.2 POMMERMAN

Pommerman is an arcade game, specifically created for AI research [4]. It is based on the classic maze-based game series Bomberman. The game is played between multiple players on a square board that players move around on (see Figure 2.1). At each timestep, players can move one step in 1 of the 4 cardinal directions, or they can plant a bomb at their current location (or they can do nothing). Bombs detonate after a fixed timer, destroying light brown wooden boxes (making their squares traversable) and killing any players in the blast

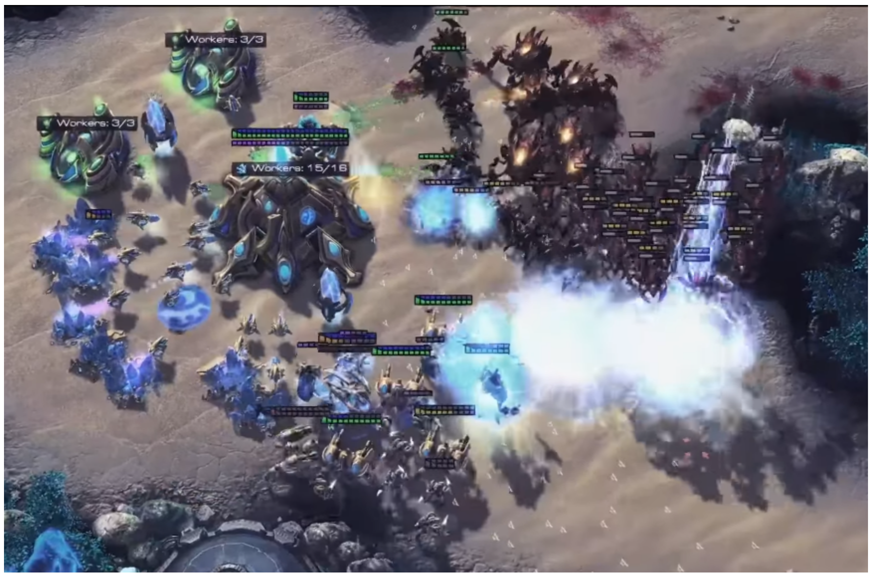


Figure 2.2: A battle between opposing armies during a match of StarCraft II. One player's base on the left side, an important part of his economy, is under attack.

range. Exploding bombs also immediately detonate other bombs in their range (even if they have remaining time on their timers), causing chain reactions of explosions. Bombs leave behind flames for a short time, which kill any players that walk into them. The objective of the game is to be the last surviving player.

When wooden boxes are destroyed by bombs, there is a random chance that a power-up is left in its place. Players can gain power-ups by walking over them, and there are 3 types: more ammo, increased blast range, and kick. By default, players can only plant one bomb at a time, and must wait for it to explode before planting another. The ammo power-up increases the number of bombs a player can plant at a time by 1. By default, bombs planted by a player have a blast range of 1: they explode the tile the bomb is on and each of the 4 neighboring tiles. Each time a player gets a blast range power-up, the range of their bombs extends in each of the 4 cardinal directions by 1. Finally, the kick power-up gives the player the ability to kick bombs: if a player walks into a bomb, the bomb starts moving in the direction kicked, moving by 1 square per timestep until it hits an object or detonates (by default, players cannot walk into bombs).

2.3 STARCRAFT II

StarCraft II is a real-time strategy game for PC developed by Blizzard Entertainment, released for PC in 2010. Players play as one of three races, expanding their base by building buildings and making combat units to destroy the opposing base. The game involves many different skills, including strategy, planning, multi-tasking, and reaction speed. The tasks that a player must accomplish can be abstracted into two areas: macro, creating the correct buildings and improving the economy; and micro, controlling combat units and executing tactics to maximize their efficiency. Compared to other strategy games like chess, StarCraft II has additional challenges like incomplete information: the state of the opponent is unknown, and effort must be devoted to uncovering information. Also unlike traditional strategy games like chess, attention is a scarce resource. For example, the player must control a camera that can only view a small fraction of the battlefield at a time, restricting both their observations and orders. These unique challenges have drawn reinforcement learning researchers to create agents that learn StarCraft II.

2.3.1 StarCraft II Learning Environment

Due to research interest in StarCraft II, the RL research firm DeepMind partnered with StarCraft developer Blizzard Entertainment to create an API for RL research, called the StarCraft II Learning Environment (package name `pysc2`) [5]. The API features a comprehensive state interface, giving agents a view of the game as a stack of images plus some scalars, and a hierarchical action interface, through which agents specify arguments for the many different kinds of actions in the game. The StarCraft II Learning Environment also includes a set of minigames, which focus on specific micromanagement skills and are significantly simpler to learn than the full game. Our StarCraft II experiments in this work are about learning BuildMarines, the most complex of the minigames.

2.3.2 BuildMarines

In BuildMarines, the player plays as Terran (one of the three races in StarCraft II), and the goal is to build as many Marines as possible. Marines are an infantry unit created out of buildings called Barracks. In this minigame, the entire traversable map is small enough to fit within the player's camera, so it can all be viewed at once and the camera cannot be moved. At the start of each episode, the player starts with 1 Command Center and 12 SCVs. SCVs are the workers of the Terran race: they collect resources called minerals, and they build buildings. The Command Center is a central building that makes additional SCVs, and also where SCVs travel to to drop off their collected minerals.

Minerals are collected over time by workers, traveling between mineral patches and the Command Center. The player also starts with 50 minerals. All units and buildings cost minerals: an SCV costs 50 minerals, a Marine costs 50, a Barracks costs 150, and a Supply Depot (see next paragraph) costs 50. The player can make more SCVs from the Command Center to increase the rate of mineral income.

Besides minerals, there is another resource the player must manage: Supply. Supply is a limit on the number of units (total count of SCVs and Marines) the player can have at one time. If the Supply cap is reached, new units cannot be made. The player starts with 15 Supply, and can increase it by building a building called a Supply Depot, which increases the Supply cap by 8. The Supply Depot is also a prerequisite for the Barracks; without at least one Supply Depot already built, a Barracks cannot be made.

Thus, to create a Marine, the player must first build a Supply Depot, then a Barracks, and then make a Marine using the Barracks. The objective of the minigame is to create as many Marines as possible over the course of 15 minutes, after which the episode ends. The agent receives a reward of 1 every time a Marine is created. There is no other reward and no negative reward, even if a Marine dies. Thus, a player can use some Marines to kill other Marines, staying below the Supply cap without reducing score.

CHAPTER 3: RELATED WORK

3.1 CURRICULUM LEARNING

Our work is a development of the idea of curriculum learning, which was introduced by Bengio et al. in 2009 [6]. They found that training on examples in a meaningful order of increasing complexity, introducing more concepts over time, produced better final accuracy in experiments on word prediction. Curriculum learning was first applied in the field of reinforcement learning by Narvekar et al. in 2016, by designing a curriculum of tasks based on observed action trajectories of the learning agent [7]. Shortly afterwards, it found use in first-person shooter games, in combination with Asynchronous Advantage Actor-critic (A3C) [8]. In 2018, Narvekar and Stone developed a method to automatically generate curricula for reinforcement learning, rather than manually crafting them [9]. Our work differs from prior work by introducing a new way of manipulating environment state and action interfaces to create a curricula of environment versions.

3.2 POMMERMAN

As a game designed specifically for research, Pommerman has been used in a variety of experiments in reinforcement learning since its release in 2018 [4]. It has been the subject of an annual competition at NeurIPS for multi-agent research, with some contestants utilizing curriculum learning [10, 11]. Other agents have used imitation learning to learn Pommerman, which is alike to curriculum learning in that it helps training by making the task easier in the beginning [12]. Pommerman has also been used to study the use of shallow Monte Carlo tree search for safe RL, the difficulty of exploration, and the comparative strengths of different statistical forward planning methods [13, 14, 15]. Our experiments in Pommerman are not novel by themselves, but they motivate the design of environmental curriculum learning.

3.3 STARCRAFT II

Recently, StarCraft II has become one of the most popular games in reinforcement learning research. In 2019, DeepMind made a breakthrough by creating AlphaStar, a reinforcement learning agent that was able to beat 2 professional StarCraft II players in the full game (although with minor restrictions on race and map choice) [16]. However, AlphaStar was not trained on the minigames of the StarCraft II Learning Environment, and would likely take many more training episodes to learn them compared to other experiments due to its model complexity. There have been many experiments on the minigames as well. The record score for BuildMarines, the minigame that we experiment on in this work, was set by DeepMind's study of relational reinforcement learning, although it remained below the human high score [17]. Other experiments in the minigames include a study of population-based training of neural networks, although it did not have a good score for the BuildMarines minigame in particular [18]. Curriculum learning has also been applied to StarCraft II, to master the skill of micromanagement of units [19]. However, our work is the first to use curriculum learning on the BuildMarines minigame, and our methodology is novel as well.

CHAPTER 4: METHODS

4.1 ENVIRONMENTAL CURRICULUM LEARNING

Here, we formally describe our technique of environmental curriculum learning (ECL) in general terms. First, we start with an environment that we want our agent to learn. We'll call this the *base* or *original* environment, because we will modify it to create several versions of the environment. We will consider this environment in terms of its *state* and *action interfaces*, rather than the traditional viewpoint of it as a function mapping state-action pairs to next states and rewards. So, the first step of environmental curriculum learning is

1. Identify the state and action interfaces of the base environment. The *state interface* of an environment is the structure of the state input to the agent, i.e. the set of features that make up the state. For example, in StarCraft II, the agent observes multiple layers of images representing the layout of units, health, etc. on the map, as well as scalar values like mineral count and supply. In this case, the state interface is each image layer along with each of the scalar features. The *action interface* of an environment is the set of possible actions that an agent can take.

2. Identify potential state or action *helpers*. Our goal is to initially make the environment easier for our agent to learn and achieve reward, by either providing richer or more structured information through the state, or simplifying the actions required to achieve rewards. Figure 4.1 shows how state and action helpers change the agent's interface with the environment. Types of state helpers include

    - Adding a state feature that combines existing input features in a more structured way. While this doesn't technically give the agent more information, it may lower the agent's burden of interpreting the raw input data. For example, if an agent playing StarCraft II needs to compute how many buildings it has created by looking at images of the map or relying on its memory, it can be assisted by adding a scalar feature to the state that holds the number of buildings constructed.
    - Adding a state feature that adds useful information that is not directly available in the base environment. For example, in StarCraft II, you cannot see parts of the map where you have no units, like your opponent's base (this mechanism is called Fog of War). To assist an agent learning StarCraft II, a state helper can circumvent this restriction by adding data to the state for all of the map, including parts unobserved in the base environment.

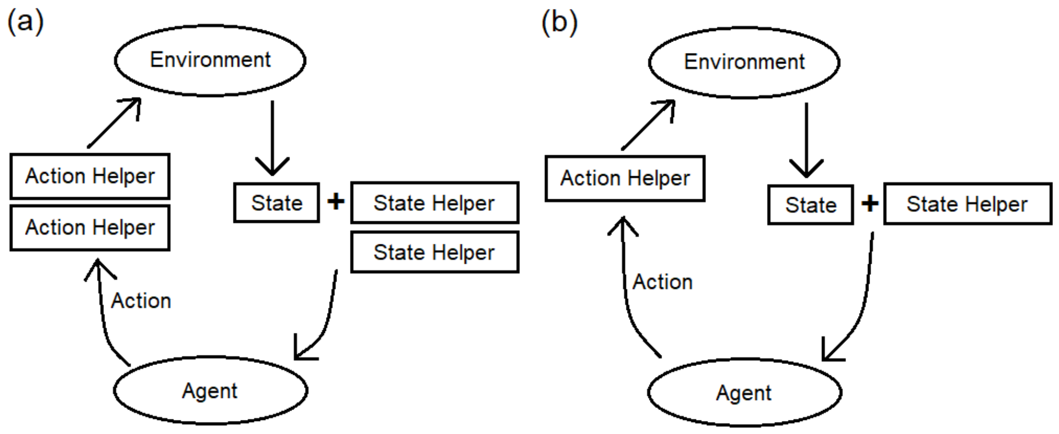


Figure 4.1: On the left, at the start of ECL, information from state helpers is added to the state before the agent receives it, and action helpers modify the agent's action before it is returned to the environment. Over the course of training, helpers are removed one by one (on the right), until all helpers are removed.

Types of action helpers include

- Making an action irrelevant by performing an action automatically for the agent when it is necessary. For example, in StarCraft II, all players must make more workers at the start of the game. To help an agent focus on learning other actions, an action helper can automatically issue orders to the base environment on behalf of the agent to make workers, so that the agent does not need to learn to take this action.
- Combining a necessary sequence of actions into a single action. Sometimes, actions need to occur in sequences or patterns to have a useful result, and it can take an agent a long time to learn the correct sequence before observing any reward. By executing the entire sequence if the agent chooses a new, artificial action, the agent can ignore learning the correct sequence while the helper is active.

3. Rank the helpers in order of increase ease/amount of help given to the agent. Each helper changes the difficulty of the environment by a different amount. Some helpers are bigger "cheats" or are higher impact because they make the agent's task much easier or even trivial, while some helpers are only minor adjustments to the base environment. In general, state helpers that add new information rather than combining existing information should usually be more helpful. Action helpers that perform an action automatically should generally be more helpful than those that combine a sequence of actions into one action. However, the exact comparisons between helpers depends on

the given environment and helper details, so decisions must be made on a case-by-case basis. Helpfulness can be hard to quantify, so deciding on an order may require some human intuition. Fortunately, the precise order is not too important, as long as the helpers are in rough ascending order of helpfulness.

4. Create the modified versions of the environment by adding each helper one by one. First, take the "least helpful" helper (as decided in the previous step), and add it into the base environment to create the first modified version. Then, create the next environment version by adding the next least helpful helper to the previously created environment version. Continue making environment versions, each corresponding to one more helper. In the end, there will be the same number of modified environments as there are helpers, with one environment version including all of the helpers and one version including only the least helpful helper.

5. Train the agent on each environment version in succession, starting from the easiest environment and ending with the base environment. The easiest environment is the environment version containing all the helpers, so the order of training is the reverse order of creation (if the environments were created from closest to the base environment to farthest). The amount of training for each environment before moving on to the next environment can be decided in two ways. More simply, the agent can be trained on each environment for a fixed number of episodes per environment, giving the same training duration for each. A more flexible approach is training the agent on each environment until the performance of the agent (as measured by average score per episode) plateaus. When the performance of an agent is stable for a long time, that usually indicates that the agent has learned as much as possible from the current environment, and is ready to absorb more difficulty in the next one. After training in this environment sequence, the agent will finally train on the base environment, and learn the target task as it was originally formulated.

4.2 OUR APPLICATION OF ECL

To give an example of how environmental curriculum learning can be applied, we will give an overview of its application to StarCraft II. In doing so, we will also show how we derived the idea of environmental curriculum learning in the first place. Our experiments in Pommerman were performed before we started in StarCraft II, and our Pommerman results motivated our methodology of environmental curriculum learning.

Our novel technique of environmental curriculum learning is based on the idea of curriculum learning in the past. Curriculum training has been used in the past in games between multiple players, when an agent is trained against successively harder opponents. Our experiments in Pommerman follow this sense of curriculum training. By warming up against a random agent (easy to beat), we expect that our agent will perform better in the end against a harder opponent (with a scripted strategy) compared to an agent that is trained against the hard opponent from the beginning. Based on our observations of the Pommerman results, we developed a new kind of curriculum learning to be used in StarCraft II.

In StarCraft II, we were concerned about creating agents to master a minigame called BuildMarines. This minigame had no opponent; the player's goal is to maximize their score of Marines produced within 15 minutes. Since there is no opponent, curriculum learning cannot be applied by increasing the difficulty of an opponent. Instead, we adjust the difficulty level of the task by modifying the environment directly. After all, from a reinforcement learning perspective, a scripted opponent is just one part of the environment, and changing any other aspect of the environment in the course of training is no different from changing the scripted opponent. To start training our agent at an easy difficulty, we scripted some of the necessary actions for accomplishing the goal, e.g. building workers. That is, we added action helpers to perform some actions automatically. (We did not use any state helpers in our experiments.) As a result, the agent only had to learn part of the set of actions necessary to reach the goal. Over the course of training, we disabled these helper scripts, increasing the difficulty of the task. By automating some actions at first and then removing such automation, we applied curriculum learning in a new way. Our hypothesis is that this kind of curriculum learning through environment modification will have similar benefits to previous applications of curriculum learning.

Details about the curriculum of opponents used in Pommerman and the curriculum of environments used in StarCraft II will be described in the next chapter.

4.3 CODE ARCHITECTURE

The technique of environmental curriculum learning introduced in this work and our experiments are enabled by a very modular code architecture. We developed a very general framework that is capable of running any reinforcement learning experiment, using any training algorithm and network and any environment. The framework consists of a collection of abstract classes, which are implemented by a corresponding set of concrete classes for each experiment. Any general training logic that is common among RL experiments is implemented in the abstract classes, reducing duplicate code. Without this framework,

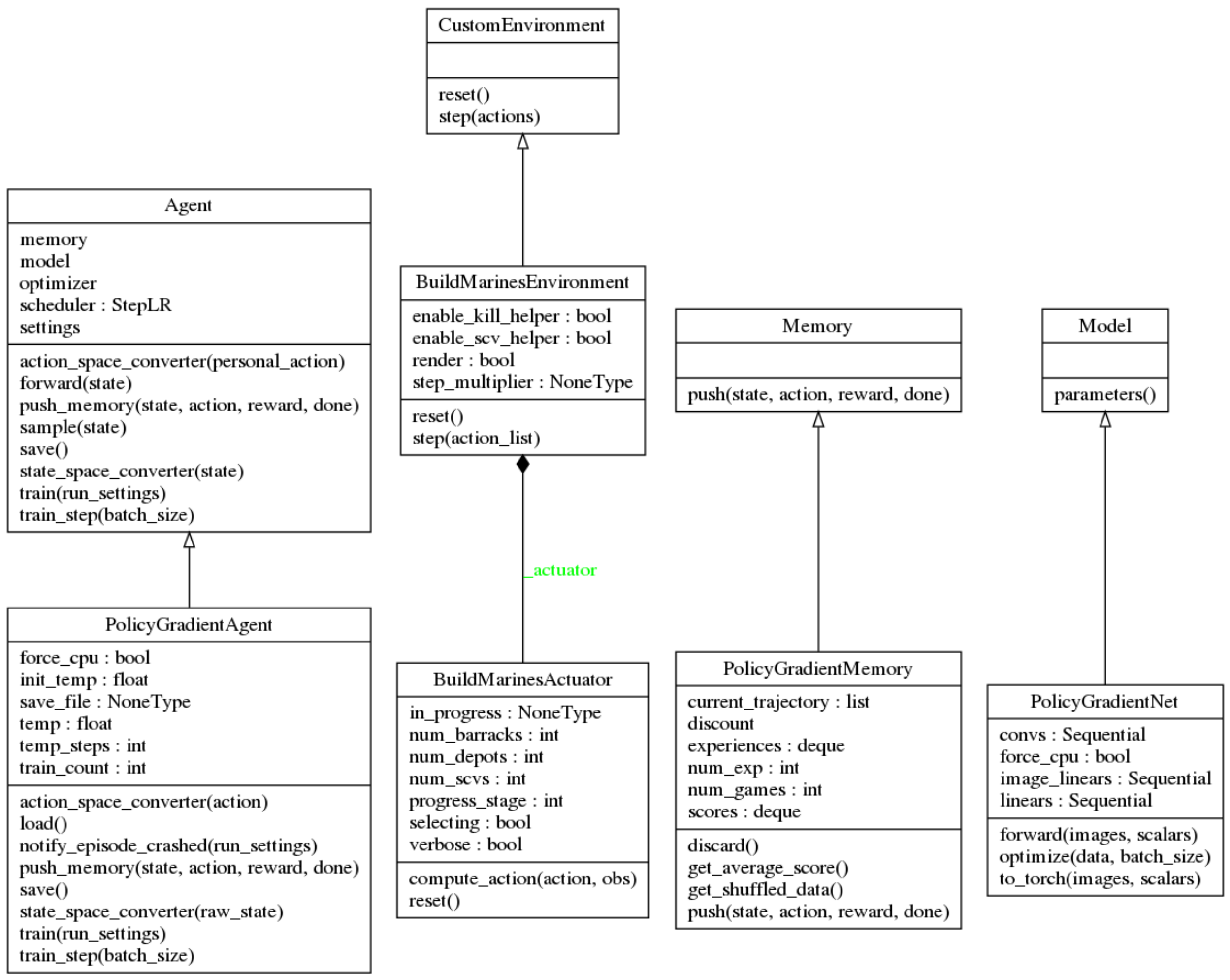


Figure 4.2: A UML class diagram of the general abstract classes and the concrete classes for the StarCraft II experiments. `Agent`, `CustomEnvironment`, `Memory`, and `Model` are abstract classes used in both Pommerman and StarCraft II experiments. They are implemented by `PolicyGradientAgent`, `BuildMarinesEnvironment`, `PolicyGradientMemory`, and `PolicyGradientNet` for the StarCraft II experiments. `BuildMarinesActuator` is a helper class used by `BuildMarinesEnvironment` to compute the transformation between our custom environment and the original PySC2 environment.

switching between slightly different modifications of the original environment would be too time-consuming to make environmental curriculum learning practical. Since the modularity of our code is so essential to our technique, we will discuss the details here.

4.3.1 `Agent`

The core of our architecture is the abstract `Agent` class, which represents the agent being trained and how it interacts with the environment. The core functions are `sample()`/

`forward()` and `train()` (see Figure 4.2). `sample()` and `forward()` are used when playing in the environment (either for data collection or testing), and returns the agent's chosen action for a given state (`forward()` may return a probability distribution across actions, while `sample()` will always return a single action). `train()` invokes the main training routine for the agent, running gradient descent on collected experiences or any other type of parameter adjustment. Two other important functions are `state_space_converter()` and `action_space_converter()`. These functions are automatically called by the main loop in `Experiment` when passing states from the environment to the agent and actions from the agent to the environment. Any `Agent` class can optionally implement these functions to insert a transformation between the environment and the agent. This is useful when training multiple agents using basically the same environment, but each agent uses a slightly different state or action interface.

4.3.2 `CustomEnvironment`

A concrete subclass of `CustomEnvironment` implements the mechanics of a simulation environment for agent learning. `CustomEnvironment` follows the interface of OpenAI Gym [20] by exposing `reset()` and `step()` functions to compute each step of the environment on demand (see Figure 4.2). These functions are called by `Experiment` in the main training loop. In many cases, such as Pommerman and StarCraft II, a library for an environment by a third party is already available. However, these environments from different sources have different interfaces, so they must be unified with wrappers that are subclasses of `CustomEnvironment`. In addition, `CustomEnvironment` wrappers can also modify the behavior or state/action spaces of external environments to better fit the needs of certain experiments, as we do here. Thanks to the `CustomEnvironment` component of our framework, the implementation details of various environments are hidden from the rest of the framework, and multiple different environments can be plugged in easily.

4.3.3 `Experiment` and Other Classes

`Experiment` contains the main training loop in its `train()` function (see Figure 4.3), which glues the other instantiated classes together by calling `step()` on the given instance of `CustomEnvironment` and `sample()` and `train()` on the given `Agent` to run data collection and training. The entire Python process is started by creating an `Experiment` and invoking `train()`.

Besides `Agent` and `CustomEnvironment`, there are other abstract classes that are used by

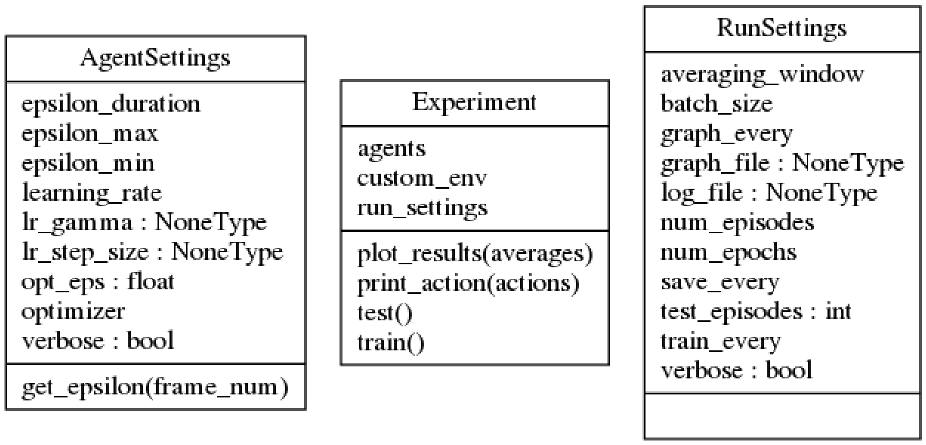


Figure 4.3: A UML class diagram of the container classes of the experimental framework. `AgentSettings` and `RunSettings` hold all adjustable hyperparameters, and are used by `Experiment` in its main training loop.

`Experiment` as placeholders for objects and data that exist in any experiment (see Figures 4.2 and 4.3). The `Memory` class stores the experiences that an agent observes during play, and each subclass of `Agent` can implement its own `Memory` class to store different data that is needed. The `Model` class represents the neural network used by an agent. Finally, `AgentSettings` and `RunSettings` store adjustable hyperparameters, such as learning rate, choice of optimizer algorithm, batch size, and training period.

CHAPTER 5: EXPERIMENTAL SETTINGS

5.1 POMMERMAN

5.1.1 Environment

In Pommerman, we trained an agent to play in a 1v1 format on a 8×8 board with a fixed starting state (see Figure 5.1). The light brown wooden squares are destructible to bombs laid by the players, and sometimes reveal power-ups when destroyed. The appearance and location of power-ups is randomized in each episode. A player wins when the opposing player dies by being near a bomb that explodes or walking into flames that are left for a short time after a bomb explodes. The agent receives a reward of 1 at the end of an episode if it wins and -1 if it loses. The episode can end in a draw if both players die on the same frame, in which case the agent receives 0 reward.

Opponent

The opponent we trained against is a scripted opponent, whose strategy does not change between episodes. In Pommerman, we applied an existing method of curriculum learning by having a curriculum of two opponents. The first opponent was a random agent who took each of the 6 actions with equal probability (moving in the 4 directions, laying a bomb, or doing nothing). The second, more difficult opponent followed a script written by the authors of the Pommerman environment (called SimpleAgent in the code). This scripted agent plants bombs near wooden boxes or the opponent, runs away from existing bombs, and moves towards power-ups that are close by. However, this agent does not try to aggressively corner or attack the opponent for reliable wins. (We also tried training against an agent that does nothing as the easy opponent, but found that episodes ran to their maximum length and it was too difficult for the learning agent to find out how to kill the idle opponent and win.)

State Input

At each timestep in a game, our agent observes 14 two-dimensional 8×8 images representing the board state and 6 scalar values. Ten of the 14 images represent the locations of empty squares, wooden boxes, solid walls, bombs, flames, extra bomb power-ups, increased

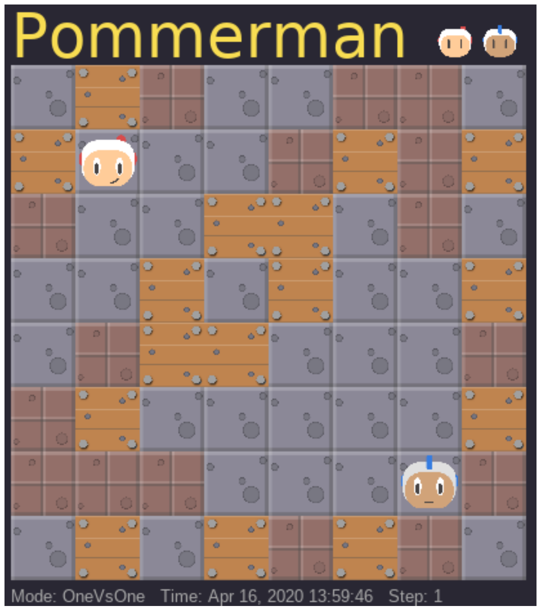


Figure 5.1: The Pommerman map layout and starting positions used in our experiments. The learning agent controls the bomber starting in the top left.

bomb range power-ups, kick power-ups, the player, and the opponent. Each of these location images contain only the binary values 0 and 1, indicating the absence or presence of the corresponding object. The other 4 images represent the blast range of existing bombs on the field, the number of steps until their explosion, whether the bombs are moving (due to being kicked), and how long current flames will last.

The 6 scalar values represent 3 values for each of the two players: their ammo (how many more bombs the player can plant), the blast range of new bombs from that player, and whether the player can kick. The scalar values are each represented as a 8×8 constant matrix so that they can be processed along with the images in the initial convolutional layers of the network.

5.1.2 Training Algorithm

Our Pommerman learning agent used Monte Carlo tree search and an actor-critic network to choose its actions. During play, Monte Carlo tree search used the output of the actor-critic network, as well as previous tree searches of prior timesteps, to choose an action. To train and improve the agent, stochastic gradient descent was performed on the actor-critic network to adjust its parameters.

Monte Carlo Tree Search

In Monte Carlo tree search (MCTS), random Monte Carlo rollouts are made to build a tree of nodes and evaluate them, where each node is a game state [1]. The distribution of nodes in these rollouts and their evaluations are determined by the actor-critic network and previous rollouts.

Let $(\mathbf{p}, v) = f_\theta(s)$ represent the output of the network f_θ for a game state s, consisting of action probabilities $\mathbf{p}$ and scalar value v. The Monte Carlo tree consists of nodes of game states s and edges (s, a) between them, where s is the parent node and a is the action to transition to the child game state. For each edge (s, a), the following are stored: a prior probability $P(s, a)$ set to the policy output $\mathbf{p}_a$, a visit count $N(s, a)$, and a Q value $Q(s, a)$. Each rollout begins at a given root state and chooses actions with the highest upper confidence bound $Q(s, a) + U(s, a)$, where

$$U(s,a) = c \cdot P(s,a)\sqrt{\frac{\sum_{b \in A(s)} N(s,b)}{1 + N(s,a)}} \tag{5.1}$$

where c is a hyperparameter controlling exploration (higher $\rightarrow$ more exploration) and $b \in A(s)$ are all the possible actions from s. In our implementation, $c = 1.5$. The rollout continues choosing actions maximizing this upper confidence bound until it reaches a leaf node l which has not been explored before. The network is used to initialize $P(l, \cdot), V(l) = f_\theta(l)$. For every edge traversed in the rollout, $N(s, a)$ is incremented by 1 and $Q(s, a)$ is updated to be the mean value $V(l)$ reached across rollouts with this edge:

$$Q(s,a) = \frac{\sum_{l \mid l \text{ reached after } (s,a)} V(l)}{N(s,a)} \tag{5.2}$$

The Monte Carlo tree is used to take an action with probability distribution π proportional to the exponentiated visit counts:

$$\pi_a \propto N(s,a)^{1/\tau} \tag{5.3}$$

where τ is the temperature parameter (higher $\rightarrow$ more exploration). We set $\tau = 1$ for the first 30 steps of each game and $\tau = 0$ afterwards (when $\tau = 0$, the formula above is ignored and the greedy action $\arg\max_a N(s, a)$ is always taken).

Actor Critic Optimization

In each training iteration, games against the random or scripted opponent are generated using MCTS as described above. These experiences are stored in an experience replay buffer, which is used in each training iteration in randomly shuffled order. The network parameters are updated with the gradients for the critic's loss, which is the mean squared error between the predicted future reward for each game state and the observed discounted reward (in our case, the reward represents the winner as +1 or -1). After the critic is updated, the network parameters are updated with the gradients for the actor's loss, which is the cross entropy loss between the current action probabilities and the best action according to the critic (that is, the action leading to the game state with the highest expected future reward). We used the Adam optimizer for all SGD updates [21].

5.1.3 Network Architecture

The actor-critic network used by our Pommerman agent is a deep residual network, using the idea of residual connections from ResNet [22]. The first layer of the network is a convolutional layer with kernel size 3×3 and padding 1, transforming the 20 input channels into 32 channels. This is followed by a batch normalization layer and a ReLU. Then, the network contains 4 residual blocks in sequence, where each block consists of

1. Convolution of 32 output channels with kernel size 3×3 and stride 1
2. Batch normalization
3. ReLU
4. Convolution of 32 output channels with kernel size 3×3 and stride 1
5. Batch normalization
6. Skip connection adding the input of this block
7. ReLU

After the 4 residual blocks, there is another convolution of kernel size 3×3 and padding 1 that transforms the 32 channels into 4 channels, followed by another batch normalization and ReLU. Then the 4 channels of 8×8 images are flattened into a single 256-vector. This vector goes through a 256-to-256 fully connected layer, batch normalization, and a ReLU before the network separates into actor and critic branches.

For both actor and critic outputs, the 256-vector goes through another 256-to-256 fully connected layer with batch normalization and ReLU activation, but the actor and critic maintain separate weights for this linear layer. For the actor, there is a final 256-to-6 fully connected layer that outputs unnormalized probabilities for the 6 possible actions. During play, softmax is applied to normalize these probabilities before initializing MCTS nodes. For the critic, there is a final 256-to-1 fully connected layer and a tanh activation to output a predicted reward between -1 and 1.

5.1.4 Hardware

Our Pommerman experiments were run on the HAL computing cluster managed by the National Center for Supercomputing Applications. We used a single node of the cluster for our experiments, which had 2 20-core IBM POWER9 CPUs @ 2.4 GHz and 4 NVIDIA V100 GPUs (each with 5120 cores and 16 GB of HBM2 memory). However, our training code was not multithreaded, so it only used one CPU core and one GPU. The node had 256 GB of DDR4 RAM.

5.1.5 Hyperparameters

Episodes against random opponent	10,000
Episodes against scripted opponent	3,350
Training iteration period	4096 game steps
Batch size	32
Discount factor	0.99
Learning rate	0.0001
Denominator addend for numerical stability in Adam	10^{-8}
Experience replay memory size	32,000 game steps
MCTS rollouts per game step	32
Temperature	1.0
Initial game steps using temperature	30
MCTS upper confidence bound coefficient (c)	1.5
Window size for averaging scores in results and graphs	200 episodes

Table 5.1: Hyperparameters for Pommerman experiments

5.2 STARCRAFT II

5.2.1 Environment

We performed our experiments in StarCraft II on the BuildMarines minigame in the StarCraft II Learning Environment. To apply environmental curriculum learning to the minigame, we made modifications to both the features of the state input and the action interface (although the modification of the state was not a helper because we removed information rather than adding information). Then we created a curriculum of 3 environments that used 2 action helpers.

State Input

The original state given by the StarCraft II Learning Environment consists of 20 image layers and over 38 scalar values. We simplified this input by filtering most of the features out to reduce input noise and computational load, since we had limited resources. We kept 2 image layers and 8 scalars from the original state as our simplified state. The 2 image layers represent the type of unit (including buildings) at each position on the screen, and the amount of hit points (health) of any unit at each position on the screen. We chose a resolution of 84×84 for the image layers, a common small screen size used in other StarCraft II research. In the BuildMarines minigame, the entire map is small enough to fit within the player's observable camera, so these image layers represent the information of the whole map at all times.

The 8 scalar features from the original state that we used are the number of minerals the player has (currency mined by workers and spent on buildings and units), the current Supply used, the current Supply cap (as given by the current number of Supply Depot buildings), the amount of Supply currently used by Marines, the amount of Supply used by workers, the number of Marines on the field, the current timestep number, and the number of workers queued to be created at the Command Center. Note that the amount of Supply used by Marines and the number of Marines are not necessarily the same, because Marines in production take up Supply but do not exist yet.

Action Interface

In the BuildMarines minigame, all actions that are irrelevant to the goal of making marines are removed from the action space. For example, the agent cannot choose to make buildings besides Supply Depots or Barracks, or build any unit from a Barracks besides a Marine. We further customized this action space by making the necessary steps towards building marines explicit. That is, the action interface for our agent consists of 6 actions:

- `NO_OP`: Do nothing
- `MAKE_SCV`: Order the Command Center to construct an SCV worker
- `MAKE_MARINE`: Order the Barracks to construct 1 Marine (all Barracks are selected as a group so the Marine is added to the queue of the next free Barracks)
- `BUILD_DEPOT`: Order a currently mining SCV worker to build a Supply Depot building
- `BUILD_BARRACKS`: Order a currently mining SCV worker to build a Barracks building
- `KILL_MARINE`: Order all Marines to attack a randomly chosen Marine (to free up Supply while not reducing score)

Environmental Curriculum

To make the minigame easier for our learning agent at first, we created 2 action helpers. Both action helpers involved taking a certain action at scheduled times automatically, so that the agent did not need to learn those action timings itself. The first action helper automatically inserts the `MAKE_SCV` action multiple times at the start of the episode, because in the beginning the player needs more workers to generate more minerals for the rest of the episode. This helper keeps making SCVs until a total of 22 SCVs is reached, because 22 SCVs is the maximum number that can still mine minerals efficiently (when there are too many, SCVs block each other from mining). Note that even if this target number is not optimal, the agent will learn to make SCVs on its own in the environment without helpers, so the agent can converge to the optimal number.

The second action helper periodically inserts the `KILL_MARINES` action once Marines have been created and are sitting on the field. At each timestep, this helper checks if there are multiple marines and if the player has less than 50 minerals. If both of these conditions are true, it inserts a `KILL_MARINES` action. The first condition is necessary because otherwise the `KILL_MARINES` action does nothing. The second condition ensures that there is not another useful action that the agent might want to take. All the other meaningful actions, which create units or buildings, require at least 50 minerals. So if the player has less than 50 minerals, a `KILL_MARINES` action does not waste an action. This helper makes the minigame easier because killing Marines reduces the Supply used without reducing the score (number of Marines made), so the player should kill Marines when possible to avoid having to construct extra Supply Depots.

We decided that the second action helper is more helpful than the first action helper, since the timing of taking the `KILL_MARINES` action is more dependent on context (i.e. when you have less than 50 minerals and no other actions to take), while the timing of the `MAKE_SCV` action is fixed and the same for every episode (at the start). Thus, it is easier to learn the correct timing for creating workers compared to learning the timing for killing Marines. Then the helper that automatically kills Marines makes a greater impact than the helper that automatically makes SCVs in terms of reducing the difficulty. Following our methodology, the most helpful helper should be removed first in the sequence of training. Thus, our curriculum of environment versions is an environment with both helpers, an environment with only the SCV helper, and an environment without either.

Action Interval

The StarCraft II Learning Environment allows an action to be taken every frame, where a frame is 1/16 of a second. However, if we trained at this interval, episodes would be very long in terms of timesteps. As a result, the agent would take a long time to finish an episode, generating less data and making training slower. Since optimal play does not require an action every 1/16 of a second, we sped up our training by using a step multipler of 16. As a result, our agent takes an action every second instead of every 1/16 of a second, and the frames in between are computed by the environment without any new actions (which is equivalent to playing with no step multiplier but taking 1 real action and 15 no-ops for every 16 frames).

5.2.2 Training Algorithm

Our StarCraft II agent uses the REINFORCE algorithm to update its policy network, one of the oldest and simplest policy gradient algorithms. During play, the policy network outputs probabilities for the 6 actions at each timestep, and these probabilities are sampled to choose the action taken. The states seen, actions taken, and rewards received are stored in an experience replay buffer, which is used in each training iteration in randomly shuffled order.

The loss function used in the REINFORCE algorithm is cross entropy loss, given by

$$L = -\sum_{i=1}^{b} v_i \cdot \log(\pi(s_i, a_i)) \tag{5.4}$$

where s_i, a_i, v_i are the state observed, action taken, and total discounted future reward received, $\pi(s_i, a_i)$ is the network's output probability for choosing a_i at s_i, and b is the batch size [23]. After computing this loss for a batch, the network weights are updated using stochastic gradient descent and the Adam optimizer.

We found that as training went on, improvements required more fine-tuned adjustments and hence smaller step sizes. To address this, we used a learning rate schedule, reducing learning rate over the course of training. We started with a learning rate of 0.0001, and halved the learning rate every 100 training iterations. When moving on to the next environment version, the learning rate was reset to the initial 0.0001, so that the network could take larger step sizes again to accommodate the new environment.

We also found that at the start of training, the agent tended to repeat useless actions rather than exploring randomly. To encourage exploration, we used temperature to smooth out the probabilities between actions. Then the sampled action probabilities become

$$\boldsymbol{\pi} = \boldsymbol{\sigma}(\boldsymbol{\pi^o}/T) \tag{5.5}$$

where $\boldsymbol{\pi^o}$ is the normalized output probabilities from the network, $\boldsymbol{\sigma}$ is the softmax function, and T is the temperature. For each environment version, we started with a temperature of 1.0 and annealed it linearly to 0 over the first 16 training iterations. When the temperature is 0, the network output probabilities $\boldsymbol{\pi^o}$ are used directly (as if there is temperature is not used).

5.2.3 Network Architecture

Like our Pommerman agent, our StarCraft II network is a residual network. However, the residual blocks in this network use fully connected layers instead of convolutional layers. The network begins with two convolutional layers of kernel size 3×3 and padding 1, each followed by batch normalization and a ReLU. The first convolution goes from the 2 input image channels to 32 channels, while the second convolution goes from 32 channels to 1 channel. That channel is then flattened into a vector of length $84 \times 84 = 7056$. The vector goes through 2 fully connected layers, each followed by batch normalization and a ReLU. The first fully connected layer goes from 7056 nodes to 294 nodes, and the second layer from

294 to 8.

This 8-vector is then concatenated with the 8 scalar input features, making a vector of length 16. These 16 values go through a fully connected layer to become 32 nodes, with batch normalization and a ReLU. Then comes 2 residual blocks in sequence, where each residual block consists of

1. Fully connected layer from 32 nodes to 32 nodes
2. Batch normalization
3. ReLU
4. Fully connected layer from 32 nodes to 32 nodes
5. Batch normalization
6. Skip connection adding the input of this block
7. ReLU

Finally, there is a fully connected layer taking the 32 nodes and outputting 6 values, representing unnormalized probabilities for each of the 6 possible actions. When choosing an action during play, softmax is applied to normalize these probabilities before sampling.

5.2.4 Hardware

Our StarCraft II experiments were run on a desktop computer with an Intel Core i7-4790K CPU @ 4.0 GHz and a NVIDIA GeForce GTX TITAN Black GPU. It had 4 cores, 8 threads, and 16 GB of DDR3 RAM.

5.2.5 Hyperparameters

Screen size	84×84
Target number of SCVs for SCV helper	22
Environment step multiplier	16
Episodes with both helpers	30,000
Episodes with only SCV helper	24,000
Episodes without helpers	22,000
Training iteration period	16,000 game steps
Batch size	32
Discount factor	1.0
Initial learning rate	0.0001
Learning rate multiplier	0.5
Learning rate update period	100 training iterations
Denominator addend for numerical stability in Adam	10^{-8}
Experience replay memory size	64,000 game steps
Initial temperature	1.0
Training iterations using temperature	16
Window size for averaging scores in results and graphs	100 episodes

Table 5.2: Hyperparameters for StarCraft II experiments

CHAPTER 6: RESULTS

6.1 POMMERMAN

6.1.1 Stage 1: Random Opponent

In the first stage of our Pommerman experiments, we trained our agent against the random agent for 10,000 episodes. The agent saw the most improvement in the first 1,000 episodes of training (see Figure 6.1). The agent's average score stabilized after 2,000 episodes, staying between 0.5 and 1.0 for the rest of training (which translate to winrates between 75% and 100%).

6.1.2 Stage 2: Scripted Opponent

In the second stage of our Pommerman experiments, we trained our agent against the scripted agent, initializing the network with the final weights from stage 1. This stage ended up being 2 separate training runs, because we made a hyperparameter change in the middle. In the first run, we did not use temperature to encourage exploration at the start of each episode (so temperature was 0, and the action with the highest visit count in the MCTS tree was always taken). After training against the scripted agent for 1,750 episodes, we observed that the agent experienced strong fluctuations in winrate (see Figure 6.2). We especially noted that there were two steep dives in the average score over time. We hypothesized that during these dives, the agent found a poor starting sequence of actions and stuck to that sequence for many episodes. To resolve this issue, we set a temperature of 1.0 for the first 30 timesteps of each episode, so that the agent made a variety of starting actions and did not get stuck. Our use of temperature resulted in more stable results in our second run against the scripted opponent (see Figure 6.3).

In the first run (Figure 6.2), the agent loses winrate in the first 250 episodes, down to an average score of -0.75 (12.5% winrate), before recovering. The agent learns how to play against the new scripted opponent, and maintains an average score of about 0.4 (70% winrate) for 750 episodes. Then the agent seems to get stuck in an ineffective strategy, rapidly dropping to almost 0% winrate, before recovering again to -0.2 (40% winrate). At this point, we paused training to apply temperature, before resuming in the second run.

In the second run, using temperature, the agent's winrate is more stable (Figure 6.3). After initial variance (the first 200 episodes show extreme winrates since the score is averaged over

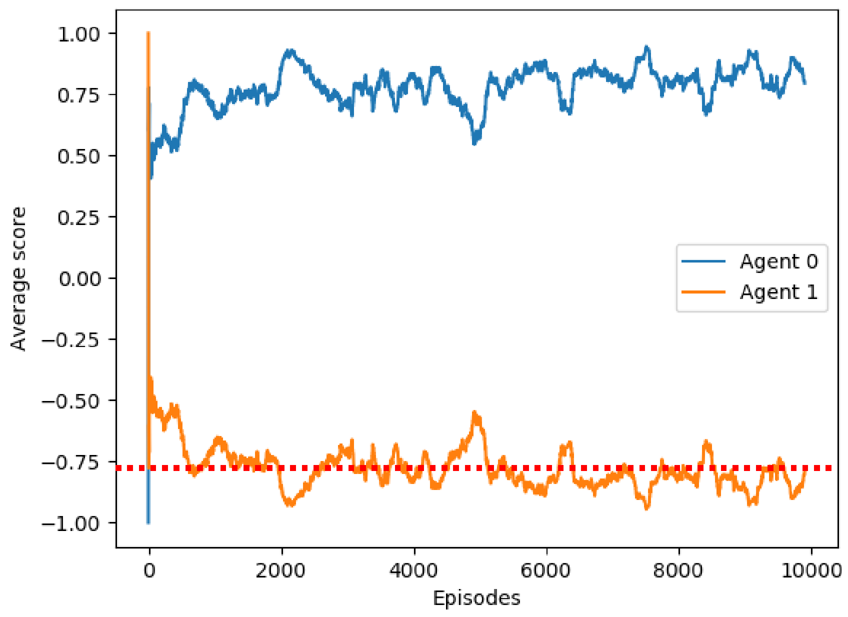


Figure 6.1: The average scores of both agents over number of training episodes in stage 1 (against the random opponent). The blue line is the learning agent's score and the orange line is the random agent's (the exact inverse). The red dashed line represents the high score of the baseline agent without curriculum learning. Averages are taken over a moving window of 200 episodes.

fewer episodes), the agent stabilizes around 0.3 (65% winrate). Over the course of 1,600 episodes, the winrate dips twice to about -0.15 (42.5% winrate), before recovering each time. However, these dips are not as extreme as in the first run without temperature. The agent peaks above 0.5 (75% winrate) before the end of the run.

6.1.3 Converged Strategy

Here, we will describe the general behavior of our curriculum learning agent when facing the scripted agent, after training for 13,350 episodes across 2 opponents. Our agent uses a mostly reactive strategy, waiting for the opponent to approach before aggressively counterattacking. In the beginning, it plants some bombs to destroy wooden boxes to check for power-ups, but otherwise it waits for the opponent to destroy the other wooden boxes separating the two players. The opponent is scripted to plant a bomb next to the agent if possible. After the opponent plants this bomb, the ECL agent springs into action, avoiding the bomb and countering by moving towards the retreating opponent and planting its own bombs. It can often create chain explosions by planting bombs near existing ones (even the initial bomb placed by the opponent), which are too complex for the scripted agent to avoid

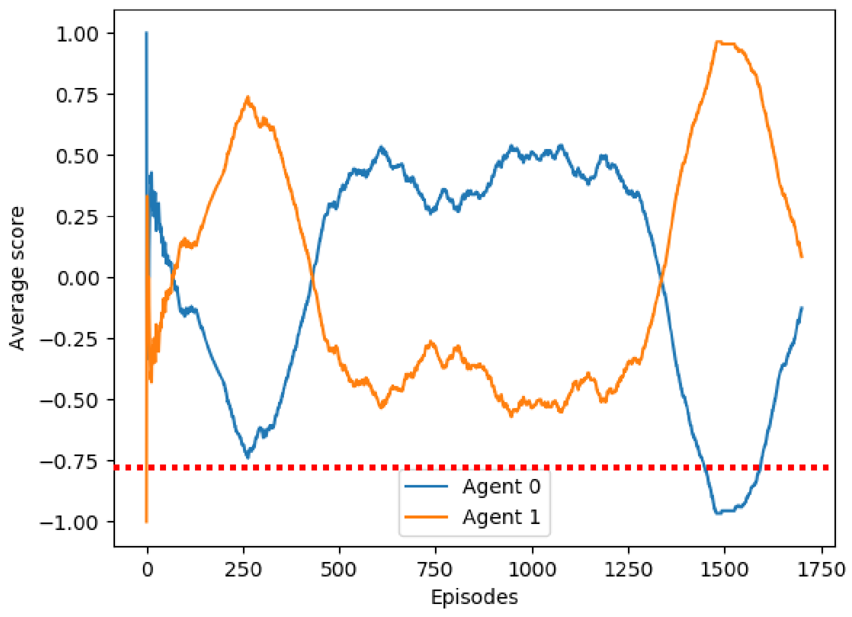


Figure 6.2: The average scores of both agents over number of training episodes in the first half of stage 2 (against the scripted opponent). The blue line is the learning agent's score and the orange line is the scripted agent's (the exact inverse). The red dashed line represents the high score of the baseline agent without curriculum learning. Averages are taken over a moving window of 200 episodes.

reliably. The agent picks up any power-ups that it can reach without crossing paths with the opponent. With the kick power-up, it gains a considerable advantage, as it kicks bombs towards the opponent when counter-attacking and increases the chance of a kill. This strategy is good enough to win against the scripted agent 75% of the time. The players sometimes tie when both die to the same chain of explosions (the agent seems to prioritize killing over surviving), and occasionally the ECL agent loses when it gets trapped in a corner by the opponent.

6.1.4 Winrate Comparison

To evaluate the effectiveness of curriculum learning, we trained an agent against the scripted agent from scratch for comparison. This agent did not use the curriculum, and initialized the network weights arbitrarily, like at the start of stage 1. We can see that this agent made no real progress after 4,000 training episodes (see Figure 6.4). The highest average score it achieved was -0.78, which translates to a winrate of 11%. Our agent that used curriculum learning achieved a maximum average score of 0.5, or a winrate of 75%, which is far better. The agent using curriculum learning was able to consistently beat the

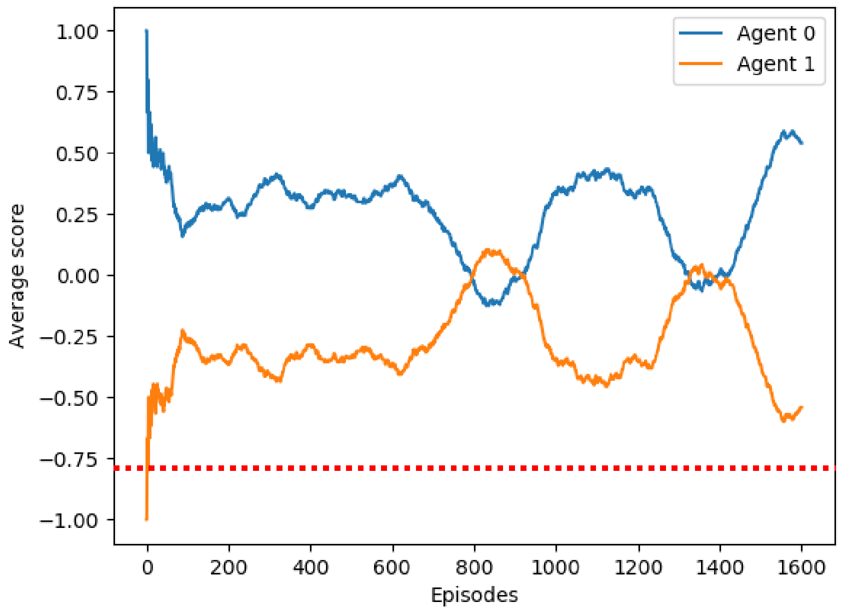


Figure 6.3: The average scores of both agents over number of training episodes in the second half of stage 2 (against the scripted opponent). The blue line is the learning agent's score and the orange line is the random agent's (the exact inverse). The red dashed line represents the high score of the baseline agent without curriculum learning. Averages are taken over a moving window of 200 episodes.

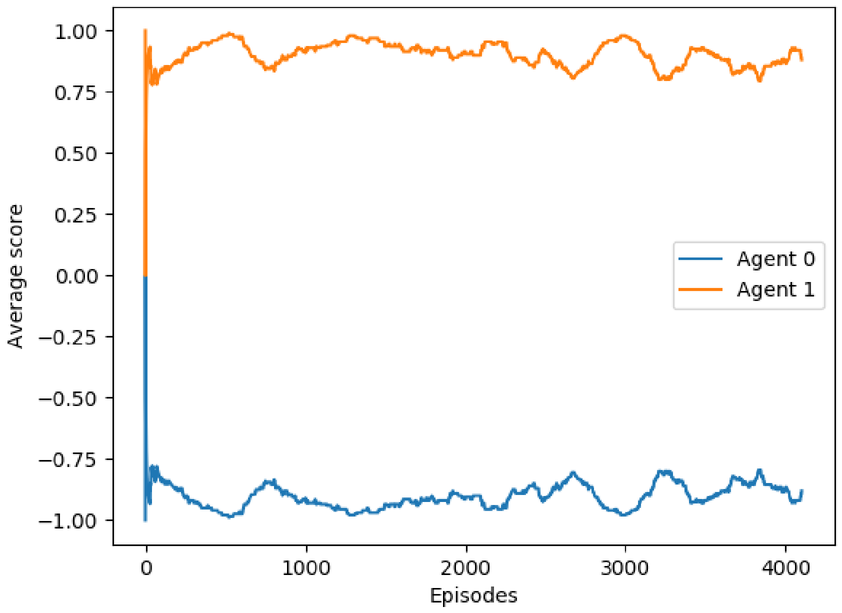


Figure 6.4: The average scores of both agents over number of training episodes when against the scripted opponent, without any prior training against the random agent. The blue line is the learning agent's score and the orange line is the scripted agent's (the exact inverse). The average is taken over a moving window of 200 episodes.

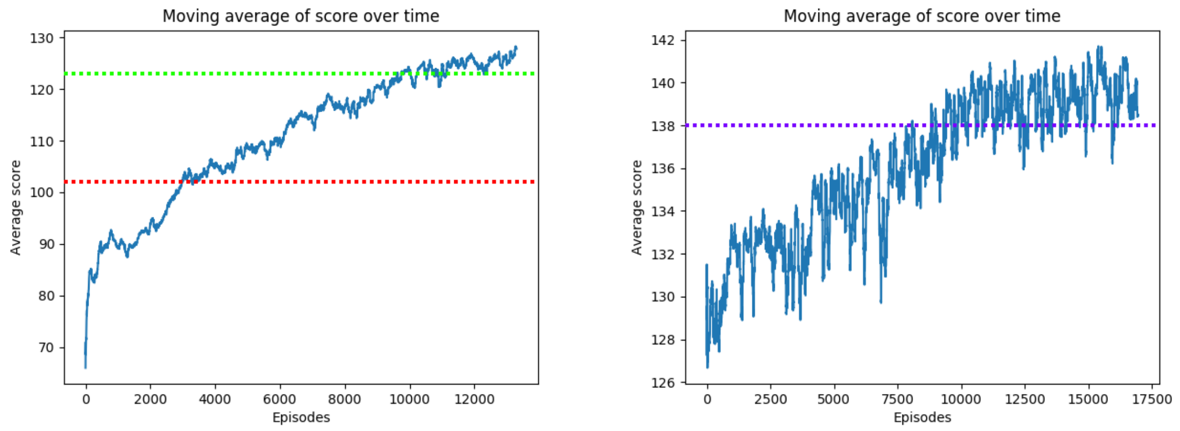


Figure 6.5: The average score over the course of stage 1, with both helpers. The figure on the left displays the first run, and the figure on the right displays the second run after it. The moving average is taken over the last 100 episodes. The red dashed line represents the high score of our baseline agent without ECL, the green dashed line represents the previous highest RL agent score, and the purple dashed line represents the human high score.

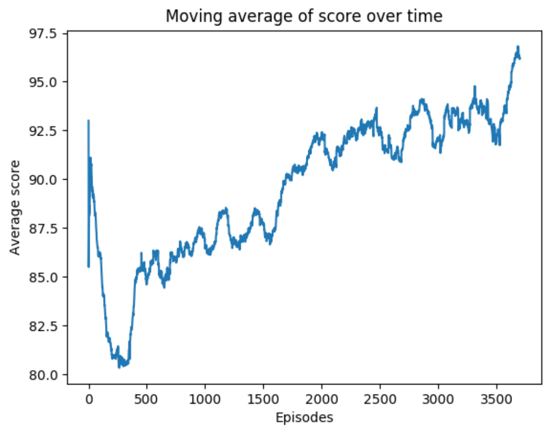
scripted agent, which was not possible without curriculum learning.

6.2 STARCRAFT II

6.2.1 Stage 1: Both Helpers

In the first stage of our StarCraft II experiments, we trained our agent using both SCV-creation and Marine-killing action helpers for 30,000 episodes (see Figure 6.5). This stage was broken up into 2 training runs in sequence, because our system was interrupted and we had to resume in a second run. The second run is a direct continuation of the first: the network weights were initialized to the last checkpoint of the first run, and no hyperparameters were changed.

In the first run, the agent steadily climbed in average score, reaching above 125 after 13,000 episodes. The sharpest improvement was in the beginning, from below 70 to 90 in the first 2,000 episodes (although the first few data points are subject to increased variance due to small averaging window). In the second run, the agent improved from 128 to above 140 over 17,000 episodes. We ended stage 1 at this point since the agent had already exceeded the previous record for this minigame (Table 6.2), so it was clear that the agent had gained a solid grasp on the game and was ready to move on without a helper.



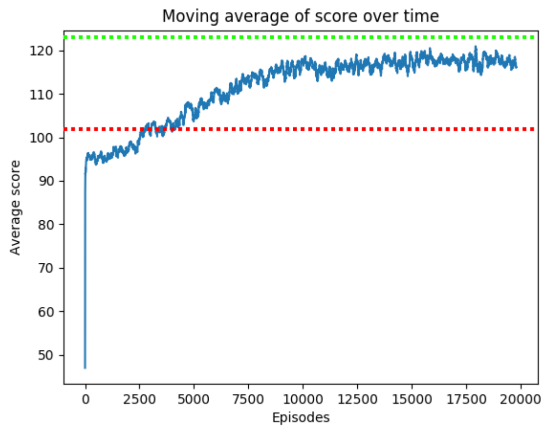


Figure 6.6: The average score over the course of stage 2, with only the SCV helper. The figure on the left displays the first run, and the figure on the right displays the second run after it. The moving average is taken over the last 100 episodes. The red dashed line represents the high score of our baseline agent without ECL and the green dashed line represents the previous highest RL agent score.

6.2.2 Stage 2: Only SCV Helper

In the second stage, the agent continued training with the SCV-making helper and without the Marine-killing helper, starting with the final network weights of stage 1. Our system was interrupted again during training, so this stage was also performed over two runs like stage 1 (again, nothing changed between runs). In this stage, the agent's average score rose steadily from 85 to 115 over the first 13,000 episodes, besides a dip in the first 500 episodes (see Figure 6.6). (The second run appears to start from a low score due to the variance of the small averaging window.) The score then plateaued, oscillating around 115, for about 11,000 episodes. At this point, with apparently no more gains to be made at this stage, we moved onto stage 3.

6.2.3 Stage 3: No Helpers

We conclude our environmental curriculum learning in stage 3, training without any action helpers. Due to the increased difficulty, the agent takes some time to get back to its high score from stage 2 (see Figure 6.7). It improved rapidly over the first 5,000 episodes, from below 70 to over 100. Score growth slowed in the next 5,000 episodes, until the agent peaked at 125. After the peak, the average score reduced slightly to between 110 and 120, where it remained for 12,000 episodes. Thus, the environmental curriculum agent achieved a high score of 125 after 3 stages of 76,000 episodes total.

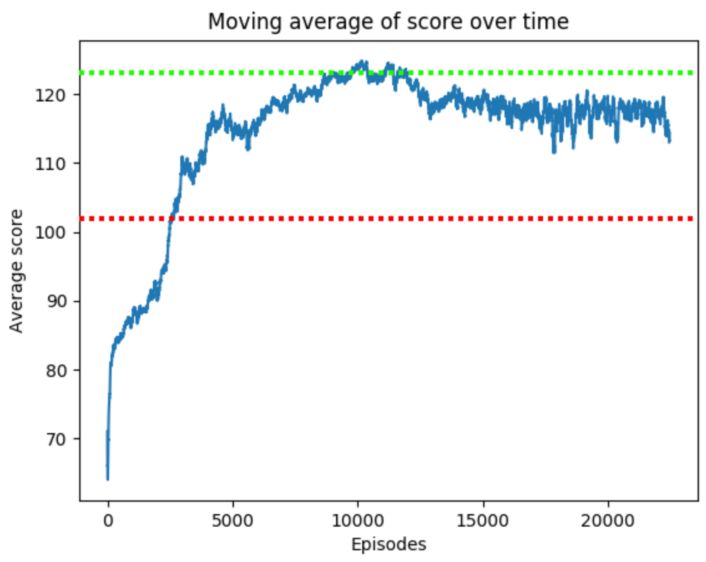


Figure 6.7: The average score over the course of stage 3, without helpers. The moving average is taken over the last 100 episodes. The red dashed line represents the high score of our baseline agent without ECL and the green dashed line represents the previous highest RL agent score.

6.2.4 Converged Build Order

In StarCraft II, a build order describes a player's starting strategy by listing what units and buildings they make in the opening of a game. We can describe our agent's final converged strategy for the BuildMarines minigame through its build order. The timing of each item in the build order is usually denoted by the current Supply of the player, but here we will use timestamps so it is more generally understandable. Each episode, the agent builds units and buildings in the following order:

Time	Unit	Time	Unit	Time	Unit	Time	Unit
0:01	SCV	1:00	Barracks	1:51	SCV	2:25	Barracks
0:09	SCV	1:09	Barracks	1:56	SCV	2:29	Marine
0:16	SCV	1:26	Barracks	2:00	Marine	2:44	Barracks
0:19	SCV	1:32	Barracks	2:09	Marine	2:50	Barracks
0:30	Supply Depot	1:42	SCV	2:10	SCV		
0:42	Supply Depot	1:49	SCV	2:12	Marine		
0:50	Supply Depot	1:50	SCV	2:16	Marine		

Table 6.1: The agent's final build order by in-game timestamp.

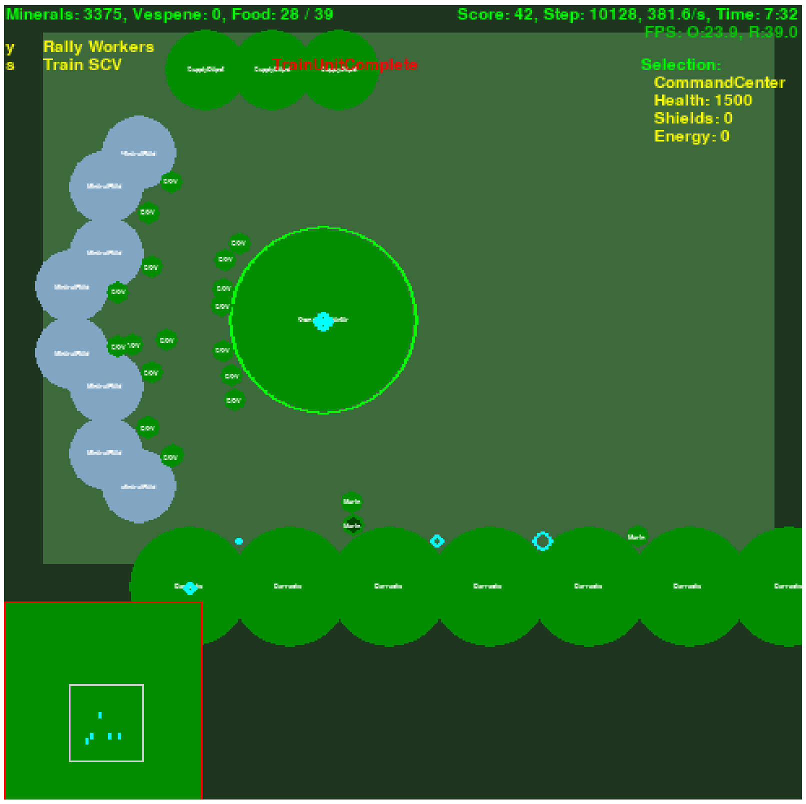


Figure 6.8: The buildings that the final agent builds each episode (3 Supply Depots at the top and 7 Barracks at the bottom), as viewed through the agent's `unit_type` input image layer.

At 2:50, the agent has built 3 Supply Depots and 7 Barracks (see Figure 6.8). From this point on, the agent repeatedly makes Marines and kills them, ordering a new Marine as soon as 50 minerals are available and issuing attack orders while waiting for minerals. The final score of each episode varies slightly between 110 and 120, due to the randomness of SCV starting locations, Marine spawn locations, SCV and Marine pathing, the random choices of which SCV to select to build a building and which Marine to attack, and slight differences in action timing.

Notably, the agent creates 3 Supply Depots before making its first Barracks, which is highly unusual in normal StarCraft II matches. The agent has optimized for making the most Marines in 15 minutes, rather than doing what would make sense in a normal game (as humans are biased to do). Also, the final agent only makes SCVs to a total of 16 before building its first buildings (making the rest afterwards), differing from the SCV-making action helper, which made all SCVs at the start. Thus, action helpers in ECL do not constrain the agent's final strategy, and the agent is able to optimize beyond them.

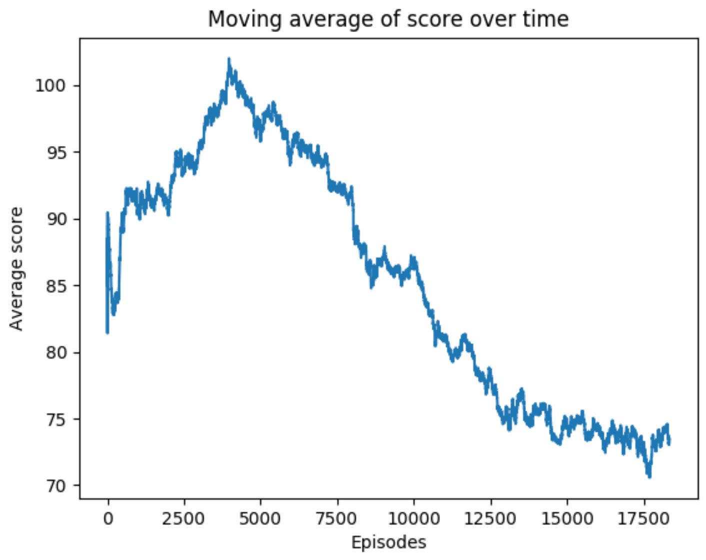


Figure 6.9: The average score of the baseline REINFORCE agent trained from scratch without environmental curriculum learning. The moving average is taken over the last 100 episodes.

6.2.5 Score Comparison

Similar to our Pommerman experiments, we trained a baseline agent from scratch without using any helpers, to evaluate the benefit of using environmental curriculum learning. This baseline agent performed significantly worse than our ECL agent after 3 stages (see Figure 6.9). This baseline agent was able to learn to a certain extent, rising and peaking at an average score slightly above 100. However, it was not able to improve beyond that, eventually falling below an average score of 75. Thus, environmental curriculum learning enabled the REINFORCE agent to achieve a much higher level of performance.

Table 6.2 shows how our results compare to prior work on the BuildMarines minigame. Before this work, no reinforcement learning agent had ever surpassed the human record of 138. Our ECL agent surpassed this record by reaching 141 in stage 1, albeit with both action helpers still active. Our agent's final high score of 125 is comparable to the previous reinforcement learning record of 123, set by DeepMind's experiments on relational deep reinforcement learning [17]. Our result is distinguished by a remarkable difference in sample efficiency: the DeepMind relational agent was trained over 10 billion game steps, while our ECL agent only experienced 44 million game steps (total across all 3 stages). Thus, environmental curriculum learning achieves the same performance 227x more efficiently, using just the basic REINFORCE update algorithm.

Agent	Score	Training Duration (steps)
DeepMind Human Player [5]	138	-
StarCraft Grandmaster [5]	133	-
Random Policy [5]	< 1	-
FullyConv LSTM [5]	6	600,000,000
PBT-A3C [18]	0	1,000,000,000
DeepMind Relational Agent [17]	123	10,000,000,000
DeepMind Control Agent [17]	120	10,000,000,000
ECL Agent Stage 1	141	17,000,000
ECL Agent Stage 2	121	31,000,000
ECL Agent Stage 3	**125**	**44,000,000**
Baseline REINFORCE Agent	102	12,000,000

Table 6.2: Highest average scores and number of game steps trained in the BuildMarines minigame for various agents. The training duration for the ECL agents are cumulative, e.g. stage 2 includes steps from stage 1.

CHAPTER 7: DISCUSSION

7.1 IMPLICATIONS

The efficiency of environmental curriculum learning and its effectiveness even with simple algorithms make it especially useful in two contexts: when limited samples can be taken, or when only simple algorithms can be used. The former situation is studied by an active area of research called sparse reinforcement learning, which focuses on the idea of sample efficiency [24, 25]. Environmental curriculum learning could be useful in this area. In another context, sometimes there are limitations on the complexity of training algorithms that can be used. For example, in low power settings like embedded devices, any machine learning code must be lightweight as possible, and deep complex networks cannot be supported. If RL training needed to be run in such a situation, environmental curriculum learning could be potentially be used to boost the performance of the simple training algorithms available.

7.2 LIMITATIONS

There are some issues that limit the applicability of environmental curriculum learning. Our application of environmental curriculum learning requires fine-tuned control over the environment, or at least some way to insert state and action helpers between the agent and the environment. This is feasible in simulator-based training, where the environment is all run in software, but may not be possible for real-world environments. However, adding helpers could still be possible in some of these situations. For example, a robot could initially learn by choosing from human-crafted sequences of actions, before taking over low level of control later in the training. Another limitation of environmental curriculum learning is that it often requires human knowledge or intuition to choose and craft helpers. Such knowledge may not exist in all context, and sometimes intuition can be wrong. Even in our own experiments, we found that our SCV-making helper did not create SCVs with optimal timing. Slight misjudgments in helper design do not seem to hold back the final agent, but helpers that are significantly worse may be useless.

Finally, the experiments used to evaluate environmental curriculum learning were only done in one minigame of StarCraft II. To conclusively establish the efficacy of ECL, experiments in other environments must be done as well.

7.3 FUTURE WORK

To further explore the possibilities of environmental curriculum learning, more experiments need to be done. First and foremost, the potential of ECL will be much better known if it can be tested in a variety of environments, perhaps even non-simulated environments. Besides other environments, environmental curriculum learning can also be tested with other training algorithms besides REINFORCE. Its effects might be magnified with more sophisticated algorithms, or it might be maximally beneficial with only simple ones. These are some of the many directions to continue studying environmental curriculum learning.

7.4 CONCLUSION

In this work, we presented a new approach to curriculum learning called environmental curriculum learning, which involves modifying the environment to create stages of easier versions. Our experiments in the BuildMarines minigame of the StarCraft II Learning Environment demonstrated that ECL was able to achieve previous records with over 2 orders of magnitude fewer training episodes. At one point during training, it even exceeded the human record, which has never been done before by a learning agent. Our experiments in Pommerman, which inspired environmental curriculum learning, confirmed previous research that curriculum learning with a series of opponents outperforms training against only the hardest opponent. Overall, curriculum learning is a very promising technique for simple and efficient reinforcement learning.

REFERENCES

[1] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. R. Baker, M. Lai, A. Bolton, Y. Chen, T. P. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–359, 2017.

[2] OpenAI, "Openai five," https://blog.openai.com/openai-five/, 2018.

[3] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *ArXiv*, vol. abs/1502.03167, 2015.

[4] C. Resnick, W. Eldridge, D. Ha, D. Britz, J. Foerster, J. Togelius, K. Cho, and J. Bruna, "Pommerman: A multi-agent playground," *CoRR*, vol. abs/1809.07124, 2018. [Online]. Available: http://arxiv.org/abs/1809.07124

[5] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. P. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing, "Starcraft ii: A new challenge for reinforcement learning," *ArXiv*, vol. abs/1708.04782, 2017.

[6] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *ICML '09*, 2009.

[7] S. Narvekar, J. Sinapov, M. Leonetti, and P. Stone, "Source task creation for curriculum learning," in *AAMAS*, 2016.

[8] Y. Wu and Y. Tian, "Training agent for first-person shooter game with actor-critic curriculum learning," in *ICLR*, 2017.

[9] S. Narvekar and P. Stone, "Learning curriculum policies for reinforcement learning," *ArXiv*, vol. abs/1812.00285, 2019.

[10] C. Resnick, C. Gao, G. Márton, T. Osogami, L. Pang, and T. Takahashi, "Pommerman & neurips 2018," in *NeurIPS*, 2020.

[11] C. Gao, P. Hernandez-Leal, B. Kartal, and M. E. Taylor, "Skynet: A top deep rl agent in the inaugural pommerman team competition," *ArXiv*, vol. abs/1905.01360, 2019.

[12] H. Meisheri, O. Shelke, R. Verma, and H. Khadilkar, "Accelerating training in pommerman with imitation and reinforcement learning," *ArXiv*, vol. abs/1911.04947, 2019.

[13] B. Kartal, P. Hernandez-Leal, C. Gao, and M. E. Taylor, "Safer deep rl with shallow mcts: A case study in pommerman," *ArXiv*, vol. abs/1904.05759, 2019.

[14] C. Gao, B. Kartal, P. Hernandez-Leal, and M. E. Taylor, "On hard exploration for reinforcement learning: a case study in pommerman," in *AIIDE*, 2019.

[15] D. P. Liebana, R. D. Gaina, O. Drageset, E. Ilhan, M. Balla, and S. M. Lucas, “Analysis of statistical forward planning methods in pommerman,” in *AIIDE*, 2019.

[16] O. Vinyals, I. Babuschkin, J. Chung, M. Mathieu, M. Jaderberg, W. Czarnecki, A. Dudzik, A. Huang, P. Georgiev, R. Powell, T. Ewalds, D. Horgan, M. Kroiss, I. Danihelka, J. Agapiou, J. Oh, V. Dalibard, D. Choi, L. Sifre, Y. Sulsky, S. Vezhnevets, J. Molloy, T. Cai, D. Budden, T. Paine, C. Gulcehre, Z. Wang, T. Pfaff, T. Pohlen, D. Yogatama, J. Cohen, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, C. Apps, K. Kavukcuoglu, D. Hassabis, and D. Silver, “AlphaStar: Mastering the Real-Time Strategy Game StarCraft II,” https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/, 2019.

[17] V. F. Zambaldi, D. Raposo, A. Santoro, V. Bapst, Y. Li, I. Babuschkin, K. Tuyls, D. P. Reichert, T. P. Lillicrap, E. Lockhart, M. Shanahan, V. Langston, R. Pascanu, M. M. Botvinick, O. Vinyals, and P. W. Battaglia, “Relational deep reinforcement learning,” *ArXiv*, vol. abs/1806.01830, 2018.

[18] M. Jaderberg, V. Dalibard, S. Osindero, W. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, C. Fernando, and K. Kavukcuoglu, “Population based training of neural networks,” *ArXiv*, vol. abs/1711.09846, 2017.

[19] K. Shao, Y. Zhu, and D. Zhao, “Starcraft micromanagement with reinforcement learning and curriculum transfer learning,” *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 3, pp. 73–84, 2019.

[20] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “OpenAI Gym,” *arXiv e-prints*, p. arXiv:1606.01540, June 2016.

[21] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *CoRR*, vol. abs/1412.6980, 2015.

[22] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[23] R. J. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning,” *Machine Learning*, vol. 8, pp. 229–256, 1992.

[24] Z. Qin, W. Li, and F. Janoos, “Sparse reinforcement learning via convex optimization,” in *ICML*, 2014.

[25] Y. Yu, “Towards sample efficient reinforcement learning,” in *IJCAI*, 2018.