Deep Reinforcement Learning Dodgeball

Kevin Tan
tankevin@stanford.edu

Andy L. Khuu
andykhuu@stanford.edu

December 14, 2019

Abstract

Deep reinforcement learning has produced both techniques and neural network architectures that are able to learn intelligent agent policies in complex environments. In this project, we designed a physically-realistic simulation environment using the Unity game engine and Unity ML-Agents toolkit to train an agent with a double DQN model and prioritized experience replay buffer. Our method produces promising results in environments with small state and action spaces but fails to generalize well in more complex environments, which we attribute to a lack of training resources.

1 Introduction

Recent years have seen the emergence of deep reinforcement learning agents that are able to play complex video games like Dota 2 and Starcraft II at superhuman levels [5][6]. These achievements have made waves in both the gaming and artificial intelligence communities but are largely inconsequential for most people. One way to bridge this gap is by developing agents with the express intention of deploying them into actual hardware (i.e. robots) that ultimately engage with people and other physical objects in their environment. OpenAI's Dactyl project demonstrated that the key to doing this successfully is to develop simulation environments that mimic the real world as closely as possible [10]. Concretely, this means that environments should include realistic data streams, accurate physics, and a convincing amount of noise.

In this project, we were interested in designing such an environment. In addition, we wanted to apply some classic techniques in the field of deep reinforcement learning to our environment, collect empirical evidence of their performance, and experiment with recently published modifications on these core techniques. We should mention that a peripheral motive of the project was to train agents with learned behaviors that lend themselves very naturally to human interpretation, which is a natural consequent of our realistic simulation design.

2 Literature Review

Designing realistic simulation environments and rapidly iterating on them in response to bugs and algorithmic innovations requires both a physics engine and an intuitive interface. One way researchers have approached this is by taking a well-known physics engine like Mujoco and writing a set of Python wrapper functions that facilitate environment development and generation. This is the method that OpenAI used to develop the robotics environments that are part of their popular Gym package [3]. While it works, it isn't the most efficient approach because Mujoco is proprietary software (which means that isn't very accessible or easy to set up) and it doesn't come equipped with common abstractions seen in environments.

An arguably more effective alternative is to use use a game engine, as most already come with all the core components that comprise environments: agents, opponents, controls, and logic. An added benefit of using game engines for environment development is that they usually include polished user interfaces to programmatically alter the environment and peruse debugging logs. This second method led to the creation of the Unity ML-Agents toolkit based on the Unity game engine [1]. This toolkit makes it straightforward to creating environment objects, instantiating agents, dictating their properties, attaching data streams to them, and specifying rewards for their actions.

After trying both approaches, we settled on latter (Unity game engine and ML-Agents toolkit) to build our environments.

As for our learning algorithm, we chose Double DQN over Standard DQN and Q-Learning with Function Approximation because it has been shown both mathematically—under very special circumstances—and empirically—on six different Atari games from the Arcade Learning Environment—to make better value estimates in the computation of the bootstrapped target used during parameter updates. In this context, "better" means "less biased" or "closer to the true value." This difference is of utmost importance because overly optimistic (i.e. positively biased) value estimates for a state s wrongly favor actions that cause the agent to end up in s and, in practice, leads to lower quality learned policies. A more detailed exposition on this topic with concrete theorems can be found in the work of Hasselt et. al [2].

The implications of our choice of learning algorithm is that, for our implementation, instead of using the standard bootstrapped target estimate found in Q-Learning with Function Approximation

$$Y_t^{\mathrm{Q}} \equiv R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a; \boldsymbol{\theta}_t\right)$$

or the one found in Standard DQN

$$Y_t^{\mathrm{DQN}} \equiv R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a; \boldsymbol{\theta}_t^{-}\right)$$

we use a slightly modified version which decouples the action selection with the value estimation

$$Y_t^{\text{DoubleQ}} \equiv R_{t+1} + \gamma Q\left(S_{t+1}, \underset{a}{\operatorname{argmax}} Q\left(S_{t+1}, a; \boldsymbol{\theta}_t\right); \boldsymbol{\theta}_t'\right).$$

In particular, instead of only having one set of parameters for both action selection and value estimation as in Y_t^{Q}, we have two sets of parameters for the two different components of each update step. The difference between Y_t^{DQN} and Y_t^{DoubleQ} is slightly more subtle: both use two sets of parameters, but Standard DQN only uses the second set in target value estimation whereas Double DQN uses both sets. These two sets of parameters are alternatively optimized, meaning they are swapped every preset number of episodes or epochs.

Another core element of our learning algorithm, the experience replay buffer, is motivated by a common issue in deep reinforcement learning: extremely high sample complexity. In general, agents need to be trained for millions of episodes before they accumulated enough experience to learn intelligent behavior. For instance, it took OpenAI's Hide and Seek agents around 2.69 million episodes before the first intelligent behavior, chasing, emerged [8]. A common remedy for high sample complexity is the use of an experience replay, where experiences—(s, a, r, s') tuples—are stored in some sort of buffer that is uniformly sampled from for later learning [4].

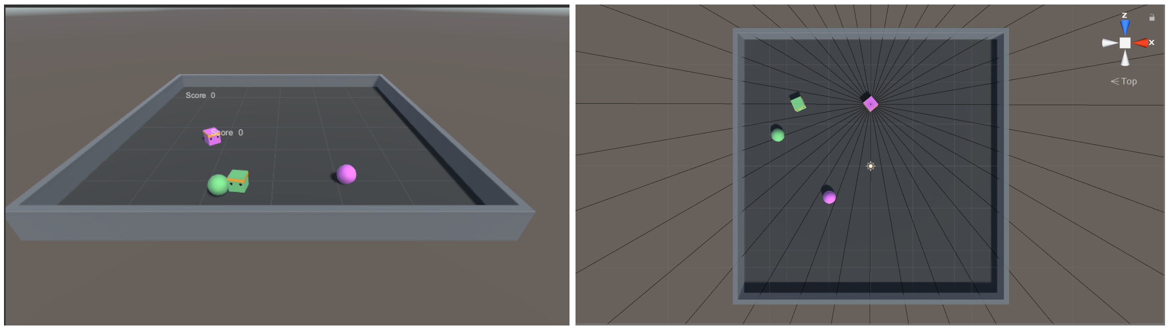
Experience replay has two main benefits: rare experiences are learned from multiple times, and the highly correlated nature of successive experiences in broken. The first benefit is especially useful in environments with sparse rewards and cognitively complex goals (end states that require long sequences of correct actions). The second benefit is conducive to training agents learns a robust understanding of the world dynamics. In the literature, there are a number of variations on this core idea including Reverse Experience Replay [12] and Prioritized Experience Replay [9].

3 Environment

Our dodgeball environment consists of an agent (purple), agent ball (purple), opponent (green), and opponent ball (green) in a square arena. The agent and opponent objectives are the equal and opposite: have its ball touch the other player. When a ball and player of different colors collide, the episode immediately ends and final rewards are assigned.

3.1 States

In this project, we experimented with both complete and partial environment observability. In the case of complete observability, we fed a vector containing the positions and velocities of the agent, agent ball, opponent, and opponent ball into our agent's policy network. In the case of partial observability, we fed a vector of simulated LiDaR rangefinder measurements into our agent's policy network. These measurements were encoded as one-hot vectors that not only include indicator variables representing which object was detected but a distance reading of how far that detected object is. With the latter approach relied on the Unity ML-Agent RayPerception library, which imbued our agent a 360-degree view of its surroundings.



3.2 Actions

We experimented with a number of different action spaces. The largest action space had size seven and is detailed in the table below. All other action spaces were a subset of the original action space that we deemed sufficient for the agent to complete the task at hand. Something that we wanted to do but were ultimately unable to was *action masking* where, in some states, certain actions would be masked out and unavailable to the agent. This would be beneficial in some circumstances (e.g. when the agent is in the corner or along the wall of the arena) because some actions would be ineffectual in these states and thus be a waste of a simulation timestep. Internally, the environment advances by converting the desired action into forces and torques applied to the agent's physics engine representation, then stepping through the simulation by one frame.

Numeric Representation	Action Description
0	Do nothing
1	Move right
2	Move left
3	Move forward
4	Move backward
5	Rotate counterclockwise
6	Rotate clockwise

3.3 Rewards

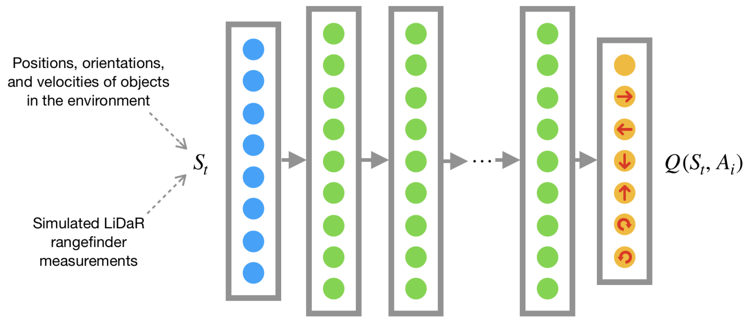
We experimented with numerous variations on our reward system, but the all generally followed the same structure, detailed in the table below (omitting specific values). We hoped the large positive reward would teach the agent to use its ball to hit the opponent, the large negative reward would teach it to dodge the opponent ball, and the small negative reward at every timestep to encourage the agent to expediently complete the task.

Event	Reward
Agent gets hit by opponent ball	Large negative
Opponent gets hit by agent ball	Large positive
Episode length increases by one	Small negative

4 Agent

There are two main ways of using a deep neural network to power an agent's decision making. The first is to define it as a mapping $S \times A \rightarrow \mathbb{R}$, that is, as a mapping from a state and action to the corresponding Q-value. The second is known as a *refined DQN* and is define as a mapping $S \rightarrow \mathbb{R}^n$ where n is the number of different actions available to the agent at a given state. The former has the benefit of being very close in formulation to the original Q-learning algorithm, but it requires n passes for every decision that is made which quickly adds up.

While this inefficiency can be partially made up for with *frame-skipping* where a decision is made every k simulation frames to reduce to total number of computations that require the neural network, we opted for the second architecture because it lends itself nicely to a clean and efficient implementation.


Positions, orientations, and velocities of objects in the environment
Simulated LiDaR rangefinder measurements
S_t
$Q(S_t, A_i)$

Simulated experiences would be stored in an experience replay buffer from which mini-batches are sampled and learned from. The learned update is the canonical Bellman update, which is a form of temporal difference (TD) learning. In particular, each gradient descent step was of the following form:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \left(Y_t - Q\left(S_t, A_t; \boldsymbol{\theta}_t\right)\right) \nabla_{\boldsymbol{\theta}_t} Q\left(S_t, A_t; \boldsymbol{\theta}_t\right)$$

where α is a scalar step size and the target Y is a bootstrapped estimate of the optimal Q-value. As mentioned in the literary review section of this paper, there are a number of different estimates for Y: Y_t^{Q}, Y_t^{DQN}, and Y_t^{DoubleQ}. Because the last target value has been shown to produce the most realistic and least biased estimates of the true state value, we opted for it. We used PyTorch to implement our networks and perform stochastic gradient descent updates.

5 Training Metrics

During the training process, we used Tensorboard to track the following three training metrics; the first two tended to be extremely noisy while the last was more stable. These metrics are derived from the work of Mnih et. al [7].

1. **Average Total Reward**: the average total reward for k training episodes.
2. **Average Episode Length**: the average episode length for k training episodes.
3. **Average State Value**: sample a random set of states before training and periodically evaluate the average value of those states—according to the DQN—during training.

6 Optimizations

In this section, we discuss a series of optimizations that we implemented for our agent in hopes of it learning faster and more effectively. These include algorithmic or infrastructural innovations that we employed on top of various other optimizations including decreasing the size of our arena, diminishing the number of redundant actions available to the agent, compiling our game in headless server mode, and spinning up concurrent simulation environments.

6.1 Curriculum Learning

Curriculum learning is the idea that cognitively complex tasks are best broken up into smaller sub-tasks that are individually easier to achieved. The idea is to progressively increase the difficult of the task presented to the agent and transferring network weights across the different stages of task learning.[11] In this project, we broke down our overall objective of getting the agent to successfully play dodgeball into five different sub-tasks with the intention of incrementally teaching the agent new things about how to succeed.

Lesson	Environment Description	Targeted Skill
1	Agent and Agent Ball	Identification of Agent ball.
2	Agent, Agent Ball, and Opponent Ball	Differentiation between Agent and Opponent balls.
3	Agent, Agent Ball, and Opponent	Utilization of Agent ball to hit Opponent.
4	Agent, Agent Ball, and Opponent	Anticipation of predictably moving Opponent.
5	Agent, Agent Ball, Opponent, and Opponent Ball	Develop strategy against hard-coded Opponent.

In practice, we found that while these lessons appear to constitute a logical progression and hint that an agent might be able to more efficiently learn the overall tasks, the agent was not actually able to transfer much knowledge across the different lessons. One reason for this is because of the heterogeneity of rewards. After training the agent to perform reasonably well on lessons 1 and 2, moving it past lesson 3 was next to impossible because all the agent wanted to do was navigate to its own ball. This is ultimately because the agent's prior experiences caused it to dogmatically expect a positive reward when it hit its own ball, which became a difficult preconception to change.

Another reason why we believe our agent's progress plateaued at lesson 3 is because the task complexity disproportionately increased, and we were still throwing roughly equal amounts of training resources. In particular, at lesson 3, the episodes no longer end when the agent reaches its own ball. Instead, the episodes end when the agent's ball hits the opponent. This is the first lesson in which the effects of the physics engine came in and the agent ball was no longer stationary but rather a moving, inertial object.

6.2 Prioritized Experienced Replay

Due to the sparsity of rewards in our environment coupled with the poor performance we were observing from our network, we surmised that we were facing a situation similar to the "Blind Cliffwalk" scenario.[9]. To counteract the sparsity of incentives in our environment we implemented a prioritized replay buffer which stored "memories" of previous actions in a cache which would be sampled and replayed as updates to our agent, therefore enabling rare beneficial experiences to be utilized more than once. As a result of hardware limitations, our memory buffer was limited in size which we remedied by removing experiences as capacity was reached. To further add upon this idea of replaying certain memories, we added prioritization to the replay of memories based on the magnitude of their temporal-difference(TD) error.

$$\delta_t = R_{t+1}\gamma V(S_{t+1} - V(S_t)$$

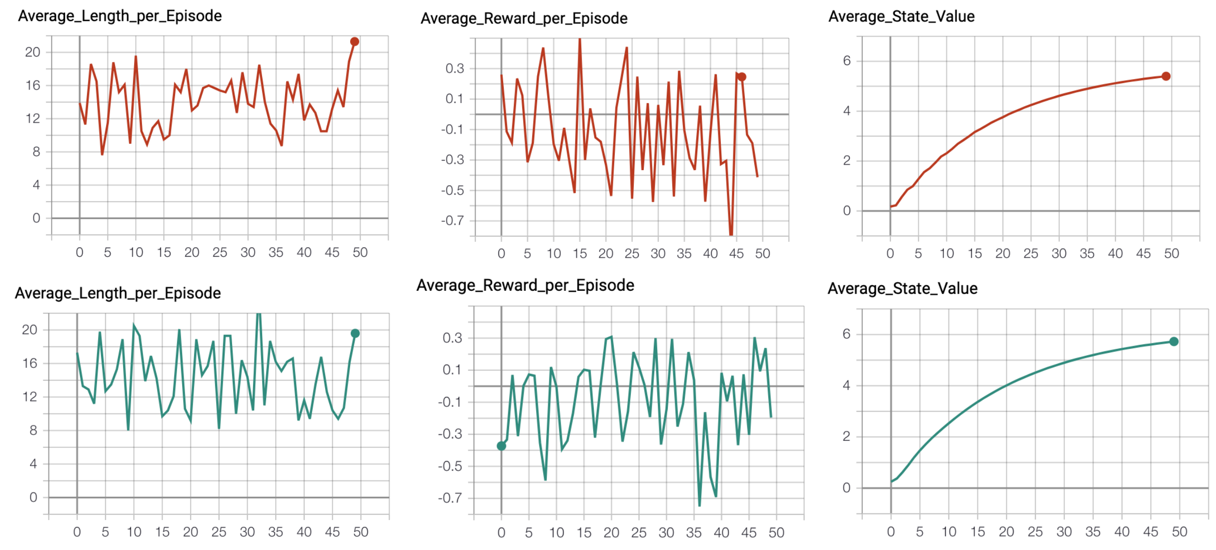
This prioritization enabled us to further hone in on positive rewarding actions, such as a hitting the opponent and having them be of focus during every update.

6.3 Target Network

One of the major additions that we implemented into our DQN was this idea of a second target network. We believed that the complexity of our environment would have proved to cause the weights of our network to spiral out of control extremely quick in a multitude of episodes, therefore giving rise to this need of a second target network whose weights would only be updated in an infrequent manner[7] such that training could continue to proceed smoothly without fear of quickly reaching a point of no return.

7 Results and Discussions

In our most simple curriculum lesson, our agent was able to achieve respectable results in that it learned to identify where it's own ball was and ultimately move towards it. Similarly, our agent was able to differentiate between it's own ball and the opponent's ball and selectively navigate towards it's own ball while simultaneously staying away from the opponent's ball. However, our agent's progress hit a notable block starting with our third curriculum lesson; despite training the agent for many thousands of episodes, it was still unable to learn a policy that allowed it to use its own ball to hit a stationary opponent.



Here are the results for two training runs on our basic environment using different network architectures, which garnered similar results. The average length and reward per episode were noisy metrics that did not closely reflect the agent's actual learning progress, stemming from the fact that the agent employs a semi-random epsilon-greedy strategy during training. The average state value (after some smoothing) was a more stable metric of learning progress; these findings are similar to those in the work of Mnih et. al [7].

Ultimately, we believe the game that we designed was too complex for the types of neural architectures and agent setups that we were considering (in conjunction with the amount of compute we were throwing at it). From playing the game ourselves, we noticed that it was incredibly hard to win; the objective of hitting the opponent agent was highly nontrivial. The main reason for this is because it requires a great deal of foresight. This need for foresight manifests itself in many ways:

1. **Anticipation of opponent movement**: The agent needs to predict where the opponent is going to be at future timesteps, otherwise there isn't even a concrete objective for what its control actions are supposed to do.
2. **Understanding of collision dynamics**: In our game, it was exceedingly hard to hit the opponent with the agent ball directly without first bouncing the ball against the arena walls. This is due to the fact that the opponent (in the later curriculum lessons) moved quickly enough that it was impossible to get into the perfect position for a direct hit. Thus, in order for our agent to succeed, it needed to have a model of how the ball would ricochet off the walls.
3. **Awareness of moving projectiles**: Similar to the need for the agent to understand how its own ball would interact with the walls to hit the opponent, the agent also needs to know to keep track of where the opponent ball is as well as its nominal trajectory.

8 Future Work

Recognizing that our environment requires a degree of planning, awareness, and foresight, we could experiment with policy networks that that directly incorporate all of these things. For instance, to increase our agent's capability of planning and foresight, we could add some recurrent connections to hold state. Also, for heightened awareness, we could augment our vector data streams with a simulated camera feed that is fed into a CNN for processing.

9 Conclusion

In this paper, we implemented a refined DQN and evaluated it's performance across simple environments built in Unity as well as a hand built complex Dodgeball environment. Our method demonstrated strong results in an environment with simple state and action spaces, but was unable to replicate this success in our Dodgeball environment—whose state and actions spaces were significantly more complex—despite many optimizations to improve sample complexity. Noting that environments similar to ours, in particular, OpenAI's Hide and Seek environment[8], required 2.69 million episodes before any intelligent behavior emerged, we partially attribute this result to a lack of computation power and training resources; another possible reason is the simplicity of our policy network architecture, which may not be expressive/deep enough to simultaneously interpret raw sensory information and suggest optimal actions.

10 Code

`https://github.com/kevtan/drl-dodgeball`

References

[1] Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A General Platform for Intelligent Agents *arXiv preprint arXiv:1809.02627*, 2018.

[2] Hado van Hasselt, Arthur Guez, and David Silver. Deep Reinforcement Learning with Double Q-Learning. *arXiv preprint arXiv:1509.06461*, 2015.

[3] Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, Vikash Kumar, and Wojciech Zaremba Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research *arXiv preprint arXiv:1802.09464*, 2018.

[4] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, Volume 8, Issue 3-4, pages 293–321, 1992.

[5] OpenAI. OpenAI Five. blog post https://openai.com/blog/openai-five, 2018.

[6] The AlphaStar Team. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. blog post https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii, 2019.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with Deep Reinforcement Learning. *ArXiv e-prints, December*, 2013.

[8] Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent tool use from multi-agent autocurricula. *arXiv preprint arXiv:1909.07528*, 2019.

[9] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *International Conference on Learning Representations*, 2016.

[10] Andrychowicz,Baker,Chociej, et al. Learning Dexterous In-Hand Manipulation. *ArXiv e-prints, August*, 2018.

[11] Hacohen, Guy, and Daphna Weinshall. On The Power of Curriculum Learning in Training Deep Networks. *ArXiv e-prints, Apr*, 2019.
ArXiv e-prints, August, 2018.

[12] Rotinov, Egor.. Reverse Experience Replay *ArXiv e-prints, Oct*, 2019.