Battling Robots with Deep Q-Networks

Lenny Khazan

Abstract

Since their introduction in 2013, Deep Q-Networks have been touted as a shining example of the power and flexibility of modern reinforcement learning methods. They have been shown to achieve super-human performance across the domain of Atari games, all without specific per-game hyperparameter tuning or optimization. In this report, we apply Deep Q-Networks to the novel environment of Robocode to assess their ability to learn behavior in a more complicated, multi-agent competitive setting. We find that learning to play competitively against even a basic baseline agent requires a number of important optimizations, without which the network fails to converge to a correct strategy due to the sparse reward signal extracted from the environment.

1. Introduction

Reinforcement learning is a powerful tool for tackling problems of learning behavior. Unlike traditional supervised learning, which excels at picking out patterns and generalizing over (often large) datasets, reinforcement learning operates on a different kind of problem: given an agent that lives in an environment, how does one come up with a decision-making strategy to maximize potential future reward? This class of problems covers a diverse range of use cases that span everywhere from video games to continuous control problems. In this report, we apply a known reinforcement learning technique, Deep Q-Networks, to a novel task, Robocode, and evaluate its performance. We will argue that this environment serves as a rich framework to highlight many of the classical problems of reinforcement learning.

Section 2 provides a brief overview of reinforcement learning, Deep Q-Networks, and the Robocode environment. Section 3 explains the methods and architecture used in this report. Section 4 presents our results, and Section 5 concludes.

2. Background

2.1. Reinforcement Learning

As mentioned in the introduction, reinforcement learning deals with the problem of learning behavior in an environment. More concretely, we define an environment $\mathcal{E}$, which operates in discrete time steps (for our purposes, we consider only environments that terminate after a finite number of steps). At each time step t, the environment $\mathcal{E}$ is in some state $x_t \in \mathbb{R}^d$. An agent operating in this environment observes this state at each time step and chooses an action $a_t \in \mathcal{A}$ to perform based on some (usually learned) policy π (while it is also common to consider continuous action spaces, we only consider discrete action spaces here). The environment executes the chosen action, gives a scalar reward $r_t \in \mathbb{R}$ to the agent, and provides the next state x_{t+1}. This loop continues until a terminal state x_T is reached, which marks the end of an episode. The objective of a reinforcement learning agent ultimately is to learn an action-selection policy to maximize $R_T = \sum_{t=0}^{T} \gamma^t r_t$ (or, more formally, its expectation), the discounted total reward, for some discount factor $0 < \gamma \leq 1$.

As an example of a reinforcement learning environment, consider the classic Atari game of Breakout. We can define the state to be the current configuration of the ball, bricks, and paddle, with three possible actions: move left, move right, or do not move. The environment provides a positive reward signal when successfully hitting a brick, a negative reward signal if we lose a ball, and a 0 reward otherwise. The episode terminates when we lose all of our balls, or when we destroy every brick.

This example highlights a number of important challenges that arise in reinforcement learning. Firstly, the environment dynamics are unknown, and can even be random: we do not know a priori that moving left will cause our state to change in a certain way (although a successful agent would likely need to learn such relationships, either implicitly or explicitly, to perform well on any sufficiently advanced task). Second, associating the reward with a given action is non-trivial: if we move the paddle left to hit the ball in time step t, we may hit the brick and receive a reward at time step $t+10$, and it is the job of our agent to determine which action (or series of actions) was responsible for that reward. Finally, an agent does not start with any knowledge of the state space, which means that it must make an important trade-off between exploring new possible states and actions, and choosing to greedily follow a path that it has already seen success in. These problems and more make reinforcement learning a challenging and diverse part of machine learning.

2.2. Deep Q-Networks

The wide array of problems that reinforcement learning encompasses has given rise to a rich taxonomy of algorithms and ideas. In this report, we consider the subset of off-policy, model-free reinforcement learning algorithms. In contrast to on-policy algorithms, off-policy reinforcement learning allows our agent to update its policy π based off of experience collected from some alternative policy π'. This provides a significant advantage in environments where collecting experience is the bottleneck in the training process (such as Robocode), because experience collected from an older policy can be reused to update a newer policy in a process called experience replay. This makes the training process more sample efficient.

A model-free algorithm learns an action-value function $Q(s, a)$, the expected total discounted reward after taking action a from state s. A policy can be derived from this function by greedily selecting the action to maximize $Q(s, a)$ from the given state. Unlike a model-based algorithm, a model-free algorithm does not explicitly learn the state transition dynamics of the system, but instead learns to approximate the expected reward directly from states and actions.

Deep Q-Networks are an approach to off-policy, model-free learning that has proven to be extremely successful in the domain of Atari environments. To train a Deep Q-Network, we collect state transitions of the form (x_t, a_t, r_t, x_{t+1}) and store them in an experience replay memory. We maintain a neural network (the model used in this report is explained in Section 3) which takes as input the state, and outputs the expected Q-value for each action in $\mathcal{A}$. We sample minibatches of experience from our experience replay memory and update the network with the following loss function:

$$(y - Q(s_t, a_t))^2$$

$$y = r_t + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a')$$

(Note that parameters for y are held fixed in this update.) Unlike on-policy methods, sampling minibatches of experience from different episodes allows each update to contain uncorrelated experience from different runs. This solves a common problem of correlated updates forcing the network to learn a policy that significantly overfits to the current episode, without learning the general state space and environment dynamics. A full writeup of the Deep Q-Network algorithm and the learning process is given in (Mnih et al., 2013).

2.3. Robocode

Robocode is a simulated virtual robot battle environment that is commonly used to teach programming concepts. Virtual robots, typically programmed in Java, are placed into a battlefield with one or more other robots. Robots can operate a radar to scan for other robots, move around the battlefield, and fire bullets. Each robot starts with 100 energy points that decrease when hit by a bullet.

Robocode comes packaged with 25 pre-built robots that demonstrate common basic strategies. In addition, Robocode maintains a large-scale competition where user-submitted robots are submitted and fight against each other to determine a community-wide leaderboard. This collection of thousands of hand-engineered agents is a unique benefit to this environment: while a common baseline in reinforcement learning is the performance of a human agent, it is not nearly as easy to compare agents against a human-engineered (but programmed) agent. This large and varied dataset of agents provides a unique testing ground for reinforcement learning.

Robocode also provides an incredibly versatile platform to tackle different kinds of problems. In addition to the default competitive mode, Robocode includes a collaborative team mode, where agents can work together (optionally with a communication channel) to compete against other teams. This provides a testing ground for learning collaborative policies, either where both agents are following the same learned policy, or learning to collaborate with a separate pre-programmed agent. Finally, Robocode lends itself well to varying levels of complexity and different algorithms, including discrete and continuous action spaces, multi-agent settings, and self-play.

3. Methods

In this report, we explore the challenges with training a basic reinforcement learning agent in Robocode. All experiences were collected in a 1-vs-1 battle against `SittingDuck`, a pre-built agent that does not move or fire any bullets. Despite the seemingly simple nature of the task, we find that several key implementation details are necessary in order for learning to converge to a successful policy. Because the spawn points are randomized at each episode, the network must learn to turn towards the opposing robot, get within an appropriate distance (too close and there is a collision penalty, too far and it becomes hard to aim), and then fire at the appropriate rate.

One of the challenges with training a reinforcement learning agent in Robocode is that the agent runs in a heavily sandboxed process that is not guaranteed to be shared across games. Thus, training and experience persistence must happen outside of the robot process. Our training setup makes use of an out-of-process Python server that listens for experience samples from the agent, stores them in-memory, samples batches and performs learning updates, serializes the network to a file periodically, and broadcasts the weights over an HTTP server to agents. The agent opens a UDP connection with the server upon starting a game and sends transitions (as defined in Section 2.2) to the server, updating its weights upon the start of the game and again every 100 timesteps. A diagram of the training setup is shown in Figure 1.

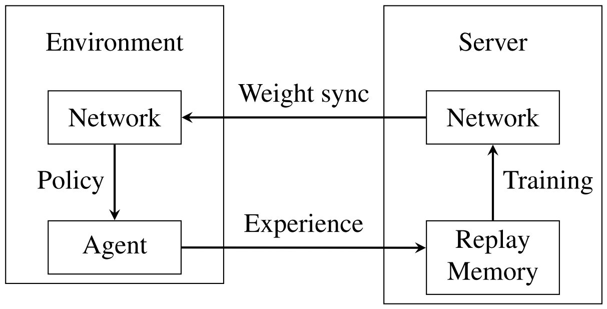

Figure 1. A diagram depicting the training setup used to overcome restrictions of the Robocode sandbox. Experience was collected in the environment (which was run multiple times in parallel), and sent via UDP to the server, which stored the experiences, ran the training loop, and broadcasted new weights back to the environment.

The state space consisted of the following values:

1. **Heading.** The direction the agent is currently facing, in the range $[0, 2\pi]$.
2. **Energy.** The amount of energy the agent has, in the range $[0, 100]$.
3. **Gun heat.** The Robocode environment requires a cooldown period after firing a bullet before you can fire again. Earlier attempts did not include this in the state, which introduced a significant source of variability in the environment: attempting to fire would not actually execute the action, which prevented the network from converging. Adding this to the state resolved this inconsistency.
4. **X and Y position.** Earlier attempts did not include this in the state, but Robocode imposes a small penalty for running into walls. Without these state variables, it is impossible for the network to learn this behavior.
5. **Opponent bearing.** The direction that the other robot is, relative to the agent's heading.
6. **Opponent energy.** The energy of the opposing robot.
7. **Opponent distance.** The distance to the opposing robot.

Note that Robocode only provides information about opponent bearing, energy, and distance when the robot is scanned with the radar. While the radar is freely controllable in the general case, we chose to simplify the environment to continuously spin the radar. Because the state only updates when we scan the robot, we consider 10 in-game ticks to correspond to 1 environment time step (so we only evaluate the network and take an action once every 10 ticks). While this decreases the quantity of transitions we can collect, it ensures that we do not take duplicate actions, which caused inconsistencies in earlier attempts.

The action space consisted of the following actions: turn left, turn right, move forward, move backward, fire, and do nothing. While the Robocode environment allows for move, turn, and fire events to specify a quantity (length of move/turn, strength of bullet), these values were fixed to simplify the action space. This is another example of a simplification that can be relaxed to attempt to learn a more general agent.

Because Robocode does not provide a reward signal directly, it is necessary to construct one artificially. One standard approach to doing this is to provide a ± 1 reward only on terminal states, depending on if the agent won or lost the battle. This reward signal is extremely sparse, and was found to not provide enough of a signal for the agent to learn a reasonable policy. Instead, the reward at each time step was given by

$$r_t = E_t - E_{t-1} + F_{t-1} - F_t,$$

where E_t is our agent's energy at time step t and F_t is the opponent's energy at time step t. This allows the agent to receive consistent positive reward as it makes progress. All experiments were performed with $\gamma = 0.99$ to give a slight priority to more immediate rewards, or else the action-value of doing nothing would be the same as the action-value of the best possible action.

In keeping with (Mnih et al., 2013), an ϵ-greedy policy was implemented in the agent. When choosing an action, the greedy (max-Q) action would be selected with probability $1 - \epsilon$. With probability ϵ, an action would be selected uniformly at random. In our experiments, ϵ was annealed from 1 to 0.1 over the first 40,000 training updates, and then fixed at 0.1. This allows the network to explore the state space thoroughly early on in training, and then focus more on exploiting the policy that it learned to further refine the strategy.

The Q-Network consisted of two hidden layers of size 32, with an input size of 8 (corresponding to the dimensionality of the state space) and an output of 6 (corresponding to the size of the action space). Mini-batches of size 32 were sampled once for every transition that is received from an agent. Four environments were run in parallel to reduce the bottleneck of experience collection and further increase the diversity of the experience replay memory. A replay memory of size 5,000 was maintained, and a transition would be discarded when the memory was at capacity and it was the oldest recorded transition (but see the discussion on experience persistence below). Training was performed for 4,000 episodes, which corresponded to approximately 1.3 million training updates. The Adam optimizer was used for training with a learning rate of .001. Tensorboard was used for tracking and visualizing agent and network performance.

Throughout the course of trying to learn a correct policy for the task, the following optimizations were necessary (listed approximately in order of decreasing effect size):

1. **Experience Persistence.** One of the difficulties of the Robocode environment is that the reward signal is sparse, even with the reward scheme outlined earlier. This is because the robot must randomly shoot in the direction of the opposing robot in order to ever see any positive reward. Shooting without hitting a target leads to a negative reward. As a result, the agent would only see a few positive-reward transitions early on in training when ϵ was near 1. As ϵ was annealed down, these transitions became rarer, and the network was never able to learn what caused the positive rewards. To solve this, a technique which we will refer to as *experience persistence* was implemented. With experience persistence, the experience replay memory will discard positive-reward transitions only with probability $p < 1$ when they are due to be discarded (in this report, we used $p = 0.1$). If a transition is not discarded, we attempt to discard the next-oldest transition, and so on until we discard a transition. Nonpositive-reward transitions were always discarded when they were due to be discarded. This allowed the training loop to perform more updates with positive-reward transitions, which enabled the network to associate actions with these rewards more readily. This optimization is similar to the one proposed in (Isele & Cosgun, 2018). This change is discussed in more detail in Section 4.

2. **State Scaling.** The state space included different variables provided by Robocode, but they all take on different ranges. Normalizing these variables to be of the same order of magnitude as the expected Q-values prevented a large spike in loss and gradient norm that would often cause instability early in training.

3. **Decreasing model size.** The network was originally implemented with two hidden layers of size 128, and training was found to be slow and inconsistent. Given the relatively simply relationship between the input and the expected Q-values, we hypothesized that the inconsistent results were caused by a form of overfitting to the specific region of the state space that was explored. Decreasing the size of the hidden layers enabled the network to generalize better to different parts of the state space, so fewer training samples were needed to learn a good policy.

4. **Mid-episode network updates.** Because an episode can extend for potentially thousands of timesteps, a small number of episodes can lead to many training steps (in our experiments, the ratio was approximately 300 updates per episode). Implementing a change to periodically sync the agent's weights with the server's weights mid-episode ensured that the experience collected was more relevant and enabled the network to converge faster.

Table 1. Agent performance against `SittingDuck`, `Walls`, and `Corners`, computed as a total over 50 rounds.

AGENT	TOTAL SCORE
DQN	**7815 (97%)**
`SittingDuck`	240 (3%)
DQN	8 (0%)
`Walls`	**8858 (100%)**
DQN	400 (4%)
`Corners`	**8843 (96%)**

4. Results

Following the optimizations described in Section 3, the network successfully learned a correct policy over 4,000 episodes (Figures 2, 3). Following an ϵ-greedy strategy (with $\epsilon = 0.1$ during testing), the network attains a 100% win rate against `SittingDuck`. As expected, the model does not generalize well to other (especially aggressive) opponents, as it has not been trained against agents that fire bullets (Table 1).

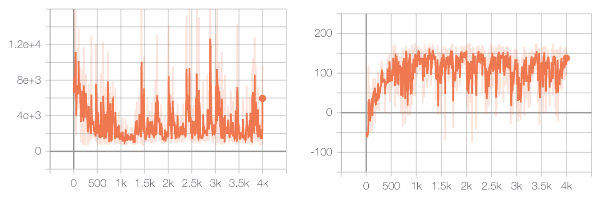

Figure 2. Training performance of the network over 4,000 episodes, depicting the number of timesteps per episode (left) and the total reward for each episode (right).

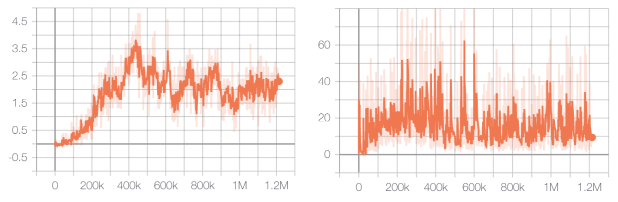

Figure 3. Minibatch statistics during training, depicting the per-minibatch average reward (left) and loss (right). The effect of experience persistence is visible in the first several hundred thousands training updates as the average per-minibatch reward grows quickly.

Experience persistence was found to be critical for the learning process. Prior to implementing this optimization, the network was unable to learn a correct policy: the network would usually choose to do nothing, resulting in 0 reward, which was still an improvement over the random policy which would lose reward by shooting at arbitrary times.

Experience persistence can be seen as a crude and simplified approximation of Prioritized Experience Replay (Schaul et al., 2015). Prioritized Experience Replay will probabilistically sample transitions from the replay memory, weighted by the error between the predicted reward and the actual observed reward. In the Robocode environment, the positive reward samples will have caused the most error early in training, increasing the frequency with which they are sampled. In the experience persistence case, this same effect is achieved, as all samples are chosen uniformly at random during training, but the large-error samples are present in memory for more training updates, and therefore in expectation are chosen more. Unlike prioritized experience replay, which requires significantly more computation and more advanced data structures, experience persistence requires no additional computation and is simple to implement.

It was also observed that the training process was extremely sensitive to the environment dynamics and reward structure. An earlier implementation unintentionally neglected to provide a reward when the last bit of damage was done to the opponent (leading to a victory), and so the true expected Q-values of all actions would be 0 as soon as the opponent health got sufficiently low. Rather than generalizing the policy that shooting when facing the opponent always provides a positive reward, the network correctly learned this unintentional environment feature, and would consistently fail to fire the final battle-ending bullet. In a supervised learning context, behavior like this may be classified as overfitting – clearly the more general solution is to smooth over these kinds of inconsistencies – but this behavior is actually expected and beneficial in a reinforcement learning setting. If the environment truly did have such an unnatural feature, then the true optimal behavior is to not shoot at this point, as it provides no additional benefit (and in fact imposes a risk of missing the shot and losing energy). The ability of the network to correctly learn and act on this behavior demonstrates the power and sensitivity of these general-purpose reinforcement learning algorithms. Patching the bug in the implementation allowed the network to correctly learn the end-of-game environment dynamics.

Similarly, it was observed that the network is sensitive to exploitation of the reward structure. In earlier attempts to encourage the network towards firing more bullets, the reward structure was adjusted to give a stronger weight to opponent damage than agent damage (so losing 1 energy point to do 1 energy point of damage would receive a positive reward). While this did encourage the agent to be more aggressive in its tactics, it quickly discovered a loophole in this reward structure: while Robocode applies collision damage equally to all robots involved, weighting opponent damage more highly than the agent's damage allowed it to receive a consistent positive reward by running into the opposing robot, even though this is not a winning strategy. This highlights the importance of designing artificial reward functions with care to avoid this kind of exploitation.

5. Conclusion

We have shown that even simple reinforcement learning tasks can be tricky to train, particularly under conditions of sparse reward. However, a number of simple optimizations can improve training performance significantly, even to the point of learning a strategy that is essentially optimal. While this report only achieves positive results on the most rudimentary baseline available, applying these optimizations to train a model on more advanced agents is a logical next step that may shed more light on the kinds of difficulties that arise in the Robocode environment.

References

Isele, D. and Cosgun, A. Selective experience replay for lifelong learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.