Learning to Navigate with Deep Reinforcement Learning

Adam Holmes

May 2022

1 Introduction

This project was solved in the context of a value-based module, so I focused on Deep Q-Network (DQN) algorithms. Before implementing any improvements to the vanilla DQN algorithm, I first set out to optimize the hyperparameters, both to see how far I could get with a vanilla DQN, and to have a fast baseline so that I could try many iterations as fast as possible. The example given in the problem specification solved the environment in about 1800 episodes, so I set the arbitrary goal of solving it in 300 episodes.

2 Vanilla DQN

In my optimization of vanilla DQN, I used the standard implementation, except that I used periodic hard updates to the Q_{target} as in the Atari DQN paper, rather than the soft updates specified in the course. It seems to me that soft updates, which update the parameters linearly, should only be used for a function approximation that is linear in its parameters, such as a single-layer neural network applied to pretrained functions of the inputs. Since I instead used a multi-layer neural network in this project, hard updates made more sense.

2.1 Q-network

As in the Atari DQN paper, I used a Q-network that took the state as input, and returned the set of Q-values corresponding to each of the available actions as output. The reason for this is so that one can evaluate $\max_a Q(s, a)$ without evaluating the Q network more than once.

The one area where I didn't optimize hyperparameters was in the architecture of the Q-network. In my limited experience in reinforcement learning, a relatively small neural network of 2-3 dense layers (possibly following 2-3 convolutional layers if the input is an image) is often flexible enough for relatively simple tasks such as this one, and the interesting part is in the learning algorithm, not the neural network architecture. Thus, I used a simple 3-layer neural network, where each hidden layer had 64 units and ReLU activation (with no regularization), for all experiments in this project. It was flexible and robust enough to do a good job solving this environment.

2.2 Hyperparameters

I varied the future reward discount γ, the ϵ schedule for ϵ-greedy action selection, the learning rate of the Adam optimizer, the size of the experience replay buffer, the number of time steps between each replay buffer sample (and Q_{local} update), the batch size per Q_{local} update, and the number of time steps between hard updates to Q_{target}.

I even tried varying γ during training, with the intuition that it should be lower on early time steps, to reduce the reliance on the bootstrapped (i.e., uncertain) values of Q. I also experimented with varying the batch size during training, with the intuition that cheap, noisy updates could allow for faster training and more exploration (as in avoiding local minima in the parameter space optimization, not exploring the environment) early on. However, in the end I found that it was best to keep both γ and the batch size fixed during training.

After extensive experimentation on the DQN, the best set of hyperparameters I could find are:

$$\begin{aligned}
\gamma &= 0.95, \\
\epsilon_{\text{init}} &= 0.4, \\
\epsilon_{\text{decay}} &= 0.995, \\
\epsilon_{\text{min}} &= 0.01, \\
\text{learning rate} &= 0.0005, \\
\text{experience replay buffer size} &= 3e4, \\
\text{time steps between replay samples} &= 4, \\
\text{batch size} &= 64, \\
\text{time steps between } Q_{\text{target}} \text{ updates} &= 2500.
\end{aligned}$$

2.3 Results

After optimizing the hyperparameters and using hard updates rather than soft, my vanilla DQN implementation ended up being able to solve the environment in about 278 ± 17 episodes (taken from 5 runs with different random number seeds), much faster than the 1800 episodes in the problem specification.

3 Double DQN

I next improved on the vanilla DQN by implementing a double DQN.

3.1 Hyperparameters

I again attempted to optimize the hyperparameters. I found that for the most part, the hyperparameters that worked well in vanilla DQN also worked well here, with two exceptions.

First, I found that the double DQN required less random exploration to work well: I set the initial value of ϵ to 0.05, compared to the 0.4 for vanilla DQN. I suspect that the reasons for this are some combination of the following:

1. averaging the two Q values reduces their fluctuations, so the agent starts off with less 'idiosyncrasies' that would otherwise get in the way of exploration, and

2. the learning process already includes some randomness, since only one of the two Q-networks is randomly selected to be updated at a time

Second, I found that double DQN worked better with more frequent replay samples: I took a new replay sample every time step, instead of every 4 time steps for vanilla DQN. I suspect that this is because double DQN removes the need for periodic hard updates to Q_{target}, so the more updates the better. In contrast, the vanilla DQN needs to weigh the benefits of more frequent updates against the fact that the training is biased due to the infrequent updates to the target network.

3.2 Action selection

In my first experiments with double DQN, the agent selected actions according to the mean of the two Q-networks, as was done in the original double DQN paper. That is,

$$a = \text{argmax}_{a'} \left[\frac{Q_1(s,a') + Q_2(s,a')}{2} \right].$$

However, this is not the only method of selecting an action using two Q-networks. The alternative I tried was to select each action using one of the two Q-networks (randomly selected).

3.3 Results

After 5 runs each, Double DQN solved the environment in 92 ± 19 episodes using $\text{mean}(Q_1, Q_2)$, and in 82 ± 24 epsisodes using $\text{RandomChoice}(Q_1, Q_2)$. Thus, both Double DQN algorithms were much faster than the vanilla DQN, but there was not enough evidence to prove that either of the two action selection algorithms was faster.

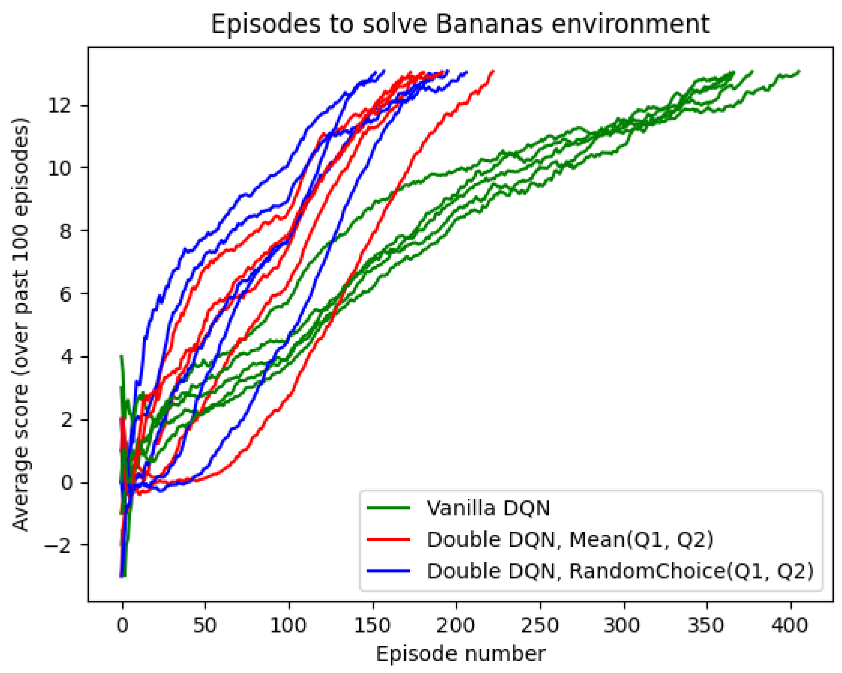

Figure 1: Results from running each algorithm 5 times with different random number seeds. Double DQN vastly outperforms vanilla DQN. While the RandomChoice(Q_1, Q_2) action selection appeared to solve the environment faster on average than mean(Q_1, Q_2), the difference was not statistically significant.

4 Summary

While most of the fun of deep reinforcement learning comes from dreaming up and implementing new algorithms, substantial gains can be obtained by simply optimizing the hyperparameters. I found that, without implementing any new algorithms (beyond the small tweak of switching from soft updates to periodic hard updates to Q_{target}), I was still able to reduce the number of estimates required to solve the environment from about 1800 to about 300.

I also implemented double DQN, and when I re-optimized the hyperparameters found that many of the same values still worked well. However, two of the hyperparameters, namely the initial ϵ value controlling exploration, and the frequency of taking new replay samples, had to be re-optimized and yielded a large improvement when they were. I found that my double DQN solved the environment much faster than the vanilla DQN, in about 90 episodes.

I tried a tweak to the double DQN's method of action selection – using a random one of the two Q-values rather than their mean. However, this did not have a statistically significant effect on the number of episodes required to solve the environment.

5 Future research

There are many other well-known optimizations to the DQN that could be implemented next, including prioritized experience replay, dueling DQN, and distributional Q-learning.

A few new ideas I had that could be worth pursuing are:

- A twist on Generalized Advantage Estimation (GAE): use a simple average of different estimates of the Q-value, e.g.:

$$\begin{aligned} Q_1(s,a) &= R(s,a) + \gamma \max_{a'} Q(s',a'), \\ Q_2(s,a) &= R(s,a) + \gamma \left[R(s',a') + \gamma \max_{a''} Q(s'',a'') \right], \\ &\dots \\ Q(s,a) &= \frac{1}{N} \sum_{i}^{N} Q_i(s,a) \end{aligned}$$

  The idea is to learn from several future Q-values with equal weight, to reduce the fluctuations in the target Q-values you learn from.

- Double Q-learning using two halves of the same network: Define a single network Q. Define Q_1 by applying dropout (with $p = 0.5$) to the last dense layer of Q. Define $Q_2 = 2Q - Q_1$, so that Q_1 and Q_2 apply dropout to disjoint sets of connections in the last dense layer of Q. Then, perform Double Q-learning as usual, where the definitions of Q_1 and Q_2 change with each training batch. In the end, only one Q network is trained.