Udacity Deep Reinforcement Learning

Navigation

Max Baugh

November 28, 2021

Abstract

1 Introduction

This project solves the Banana environment provided by Unity. In this environment, the agent moves around a 2D space littered with blue and yellow bananas, with the goal being to obtain as many yellow bananas (each worth +1) while avoiding the blue bananas (each worth -1). The state space (input to the agent) contains 37 elements representing the agent's velocity as well as measurements along several different forward-facing beams which can be thought of like LIDAR, although the exact description is deliberately vague.

2 Q Networks

At the core of the agents tested in this project are the Q networks. This is not an especially complex task, as such the Q networks are not overly complex. The first one, now called "OldQNetwork" in the notebook, consists of a fully-connected network with only one hidden layer and only 64 neurons per layer.
The second, "BaseQNetwork", only has two hidden layers and, like its predecessor, all layers have only 64 neurons and are fully-connected.
The third and final network, "DuelingQNetwork", is more complex: this makes use of the dueling network architecture first described in 2016 by researchers from Google Deepmind (https://arxiv.org/abs/1511.06581). This architecture has the same head & stem as the BaseQNetwork, but at the final layer it splits in two, with one branch attempting to compute the state-value function and the other the action-advantage function; the output of the two branches is then combined to predict the action-values for that particular state.

3 Experience Replay

Experience Replay is a method to significantly increase the sample efficiency of an agent by revisiting previous states & actions. This is done by storing experience tuples as (state, action, reward, next_state, done) in a buffer and randomly drawing from that buffer at regular intervals. Due to memory limitations, a deque was used with a finite size of 100,000 tuples.
The basic Replay Buffer does well enough, but intuitively it seems obvious that not all experiences are equally informative, and preferentially sampling from the more informative ones should lead to faster learning. This is the basis for the Prioritized Experience Replay introduced by the Google Deepmind team at ICLR 2016 (https://arxiv.org/abs/1511.05952). How does one determine which are most informative? The most natural method is to use the TD-error, declaring experiences in which the agent was "most wrong" to be "most informative" At an implementation level, this means the experience tuples are now (state, action, reward, next_state, done, TD_error).
There are two main ways of implementing the prioritization: magnitude-based and rank-based. For this project magnitude-based prioritization was used, because early experiments found my implementation of rank-based prioritization to be too slow, although in the end that same problem

proved to be insurmountable for magnitude-based prioritization as well. For both methods, the TD error is used to determine the probability of being chosen during the learning step.
Magnitude: the absolute value of the TD error is taken, then a small safety parameter epsilon is added, to prevent experiences that by sheer dumb luck had zero TD error from being ignored. These terms are them exponentiated, raised to the power α, to blend with the uniform distribution; $\alpha = 0$ corresponds to a purely uniform distribution, $\alpha = 1.0$ corresponds to the purely prioritized distribution. These terms are them summed and each individual term is divided by the sum to get a final probability.

$$P(i) = \frac{(|\delta_i| + \epsilon)^\alpha}{\sum_j (|\delta_j| + \epsilon)^\alpha} \tag{1}$$

Rank: the absolute value is taken, the experiences are ranked according to the magnitude of their TD errors, priorities are set inversely according to rank, and where we again blend with the uniform distribution by exponentiating with α

$$P(i) = \frac{\left(\frac{1}{rank(i)}\right)^\alpha}{\sum_j \left(\frac{1}{rank(j)}\right)^\alpha} \tag{2}$$

With rank-based priority, the experience with the largest TD_error is always twice as likely to be picked as the one with second largest TD_error, 3x more likely than the 3rd largest, etc., regardless of the magnitude of the differences. This has a regularizing effect, and also prevents experiences which, by dumb luck, had zero TD_error from being dropped completely. Rank-based prioritization is arguably more robust than magnitude-based prioritization, but because it requires sorting the experiences it is must be implemented much more carefully to be fast enough for reasonable use; due to overall time-constraints with this project I was unable to do a proper comparison between the two prioritization strategies.
An important detail that comes from the deep math: using prioritization introduces a bias into the updates because it samples experiences according to a different distribution, which changes the solution that would be converged to. This can be corrected for by using importance sampling (IS) weights:

$$w_i = \frac{(N * P(i))^{-\beta})}{max_j(w_j)} \tag{3}$$

$w_i = (1/N * 1/P(i))^\beta / max_i w_i$ where N is the size of the replay buffer, $max_i w_i$ is the largest IS weight, and β is another hyperparameter! The division by the largest one is to ensure the weight is never greater than 1.0 & therefore always moves downward. The authors of the original paper linearly anneal β from some initial value to 1.0, the point at which it is fully compensating for the bias introduced by non-uniform sampling, though we do not do that here.

4 Experiments

All agents tested here use the DDQN architecture, meaning the primary network is used to pick the action taken at each step but while the target network is used to evaluate the effect of that action.
All agents were run with the same set of hyperparameters, which is possible because of the similar architectures (prioritized replay has a couple extras). Several terms in Table1 require explanation.
α is a parameter used in prioritized experience replay to “blend" the prioritized distribution with the uniform distribution. This is a generalization of prioritized replay that includes both uniform and prioritized replay as special cases at the limiting values of 0 and 1, respectively; values of α less than 1.0 can be thought of as “not quite trusting" the prioritization.
β is another hyperparameter used only for prioritized experience replay and like α it varies between 0 and 1, with a value of 1 “fully compensating" for the non-uniform sampling that prioritized replay does and smaller values compensating less. At this point I do not understand it well enough to experiment with it, so β was just set to 1.0 for all experiments.
γ is the discount factor, how much less a future reward is worth vs a present reward of the same

Hyperparameter	Value
α	0.5
β	1.0
Batch Size	64
Buffer Size	100,000
γ	0.99
ϵ	1.0
Learning Rate	0.0005
seed	37
τ	0.001
Update Every	4
Window Size	100

Table 1: Hyperparameter settings

Model	Threshold Episode	Maximum Score	Peak Episode	Total Wall Time (seconds)
Old Agent	359	16.83	720	1249.031
Base Agent	331	17.18	973	1318.864
Dueling Agent	338	17.45	724	1397.513

Table 2: Agent Comparison

magnitude.
ϵ is the initial exploration parameter for "epsilon-greedy" action selection, set to 1.0 because initially the agent doesn't know anything and has no reason to try any one action over any other. During the course of training the exploration parameter decays harmonically as $\frac{1}{\text{episodenumber}}$, which worked well in earlier experiments.
τ is the soft update parameter, declaring how much of the primary network gets transferred to the target network when the latter is updated. As with several other Greek hyperparameters in this project, τ varies between 0 and 1, with 1 meaning a "full update", completely replacing the target network's parameters with the primary network's, and 0 corresponding to no update at all. Update Every is how often to perform the soft update of the target network in the DDQN architecture.
Window Size is the number of episodes to average over when doing the final performance evaluation.

Several different agents were constructed using combinations of the Q networks and replay buffers. In ascending order of complexity, we have: OldAgent, BaseAgent, DuelingAgent, PriorityAgent, and DuelingPriorityAgent. It should be noted that PriorityAgent and DuelingAgent are of comparable complexity, theoretically prioritized replay and dueling architecture are entirely orthogonal strategies.
OldAgent uses the simple "OldQNetwork", which is a fully-connected network with only one hidden layer and 64 neurons per layer, as well as uniform experience replay.
BaseAgent & PriorityAgent both use the "BaseQNetwork", which is almost the same as the OldQNetwork but with one additional hidden layer of 64 neurons. As expected from the names, BaseAgent uses uniform experience replay while PriorityAgent uses magnitude-based prioritized experience replay.
DuelingAgent & DuelingPriorityAgen use the DuelingQNetwork as their core, which is very similar to the BaseQNetwork but where the output layer is broken into two branches, one which determines the state-value function & the other which handles the action-advantage function.

5 Results

In Table2 we see the results, where the "Threshold Episode" is the first one in which the 100-episode moving average crosses 13.0 and "Total Wall Time" is the time required to do the full run of 1001 episodes; to get "Wall Time to Threshold", multiply the total wall time by threshold episode divided by 1001 (at least for the uniform prioritized replay agents).

Unfortunately, despite many hours of efforts, the Prioritized Experience Replay agents were simply too slow to actually run on this environment. PriorityAgent was run several times, but it

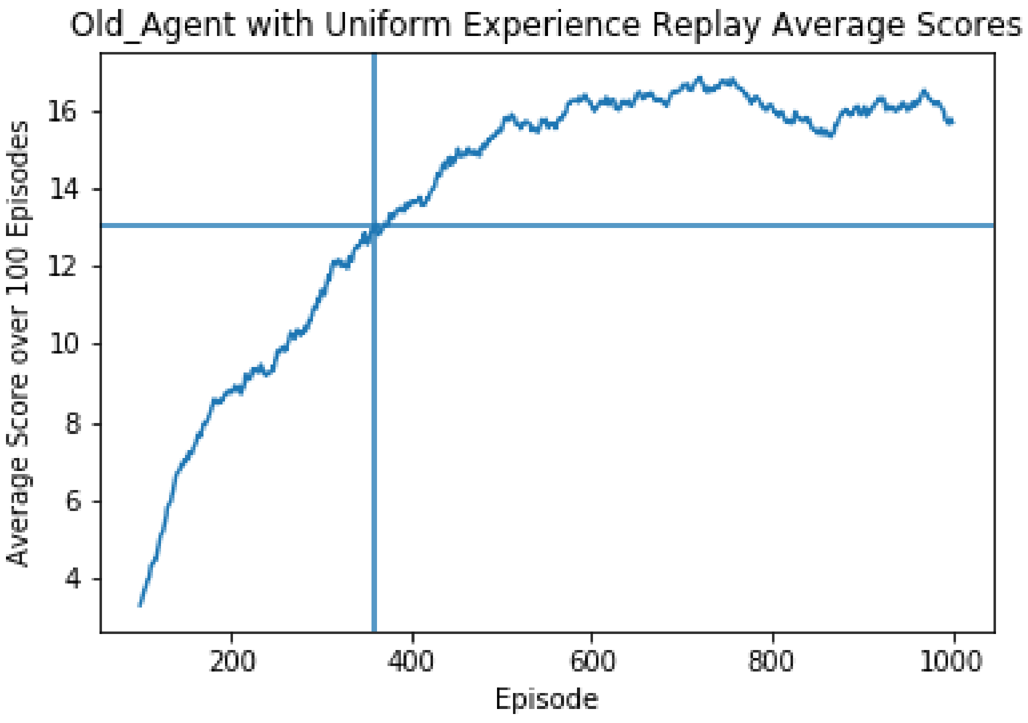

Figure 1: Old Agent Performance

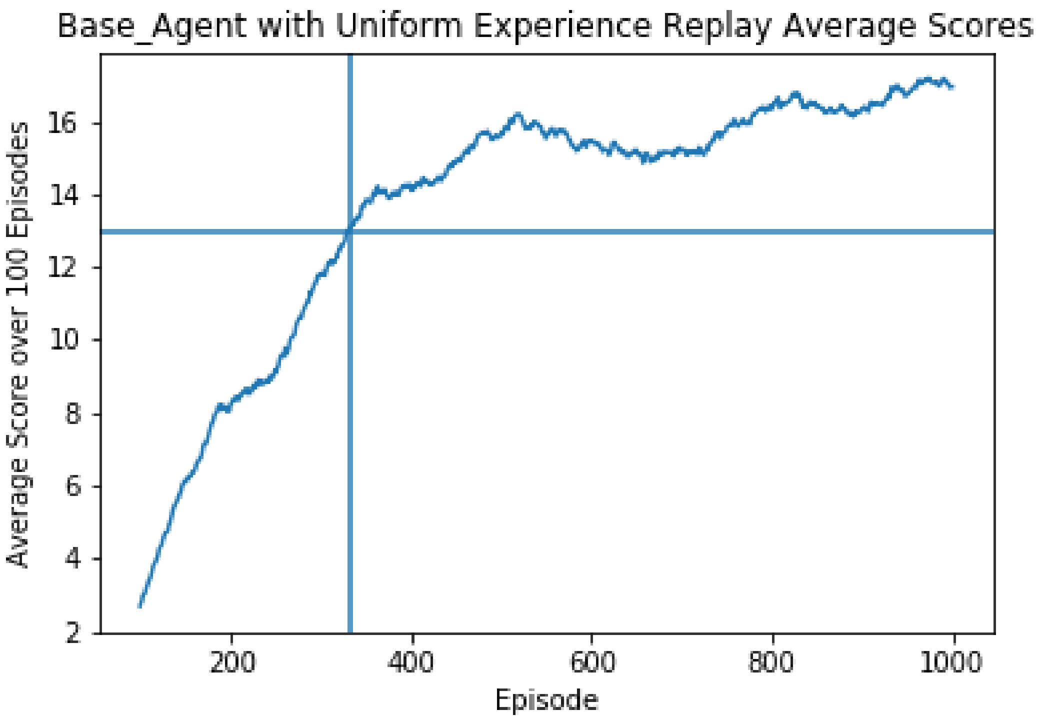

Figure 2: Base Agent Performance

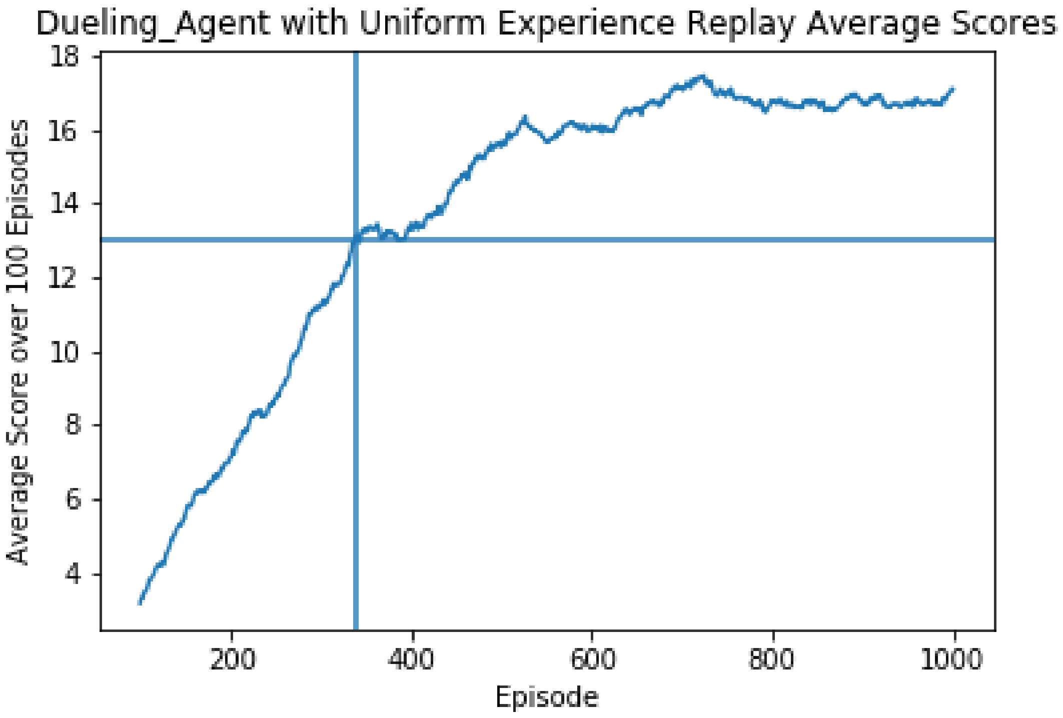

Figure 3: Dueling Agent Performance

never made it more than a couple hundred episodes before something would happen to the Udacity Workspace & it would crash. Though I cannot prove it because the numbers were never recorded, it did *look like* PriorityAgent was learning a bit faster than the ones with uniform experience replay, which is to be expected.

6 Conclusions

Simple agents with a DDQN architecture are quite capable of solving simple banana-collecting tasks. The “benchmark" DQN agent (equivalent to Old Agent) provided in the initial project description reached a 100 episode average score above 13.0 after a full 1800 episodes, whereas the DDQN architecture, which splits the action selection & evaluation in the TD error calculation between the primary and target networks, was able to get these simple architectures up above 13.0 in under 400 episodes, and up to about 17 within 1001 episodes, although all three agents did seem to level off. The dueling architecture is, in principle, superior to the standard neural network, but this is not obviously true for this task, although the dueling agent did achieve the highest maximum it is not the highest by very much. It may be the case that this environment is not complex enough to reveal the dueling architecture's superior ability, or that the experiment was not run long enough to see it win out, but either way we cannot conclude from this experiment that dueling is a big improvement.
Prioritized experience replay was attempted, but it just ran too slowly to really test. Given the performance gains found by the original authors of the prioritized experience replay paper, it seems reasonable to expect it would work here as well, but we cannot conclude that from this experiment at this time.

7 Future Opportunities

There is ample opportunity to continue expanding this project. By far the best improvement that could be made would be optimizing the code for prioritized experience replay. It *may* be more effective on a per-episode basis, but it is orders of magnitude slower according to the clock on the

wall.
An interesting modification would be to try solving the environment from raw pixels, but that would require some additional details about the input (specifically: how to get the raw pixels in the first place). The Q networks in that case would not be fully-connected as was done here but rather convolutional networks, likely with only two or three hidden layers as was done here before a fully-connected output.
Hyperparameter Optimization can have major impacts on model performance and was not done **at all** in this project due to time constraints. Some hyperparameters are work more effectively if they are allowed to vary during the course of training. Specifically, the authors of the original prioritized experience replay paper anneal β from 0.5 to 1.0 linearly over the course of the training & found that this lead to superior results, but in these experiments β was held fixed. Similarly, it is well-known that learning rate decay can improve ultimate performance of deep learning models, but although PyTorch has learning rate schedulers the learning rate in these experiments was held fixed to keep things manageable overall. Other hyperparameters, like α, γ, and τ, likely don't need to vary during training, but it is highly unlikely that the values chosen for them are truly optimal, though it may be the case that their specific values don't have much overall impact.
Another, minor investigation would be to swap out epsilon-greedy action selection for softmax-greedy selection. In epsilon-greedy, the agent takes the greedy action with probability $1 - \epsilon$, and with probability ϵ it chooses an action uniformly at random (note that this includes the most potent action, so the action with the highest Q-value has an overall selection probability of $1 - \epsilon + \frac{\epsilon}{\mathcal{A}}$). In softmax-greedy action selection, the greedy action is again taken with probability $1 - \epsilon$, but with probability ϵ the agent chooses an action according to a softmax distribution, so that actions with higher values are more likely to be chosen than those with lower values. For the same value of ϵ, softmax-greedy is more conservative, choosing the highest Q-value action a greater proportion of the time, which means a less aggressive ϵ decay strategy is probably better suited to it. It would not be difficult to implement softmax-greedy with any of the agents tested here, I simply ran out of time for this project.