Overview

This notebook walks through two variants of a RL agent that uses a deep Q-network with different tricks to solve the Unity banana-collection environment. The version demonstrated are as follows:

1. Base learner: a very basic Q-learner with a single-layer network
2. Advanced learner: a Double Deep Q-learer that includes several optimization tricks (notably Replay Prioritization and Multi-Step Learning)

Each learner is tested with a small Q-network (1x64 hidden layers) and a larger Q-network(4x64 hidden layers).

While there is no strict definition of solving the banana-gathering task, for the purposes of this experiment, we will consider any agent that has averaged an average reward of 13 or higher for at least 100 episodes to have solved the environment. (For reference, the highest acheivable 100-episode average seems to be a little over 16.)

Under this definition, with nearly-optimal hyperparameters, the advanced learner can consistently solve this environment in 250 episodes.

Note: the required Banana.app package is configured for mac only.

1. base learner

The base learner is a very basic implementation of Q-learning. The few "tricks" it is equipped with are:

memory replay: the learner stores state-action-reward tuples and randomly samples them to learn

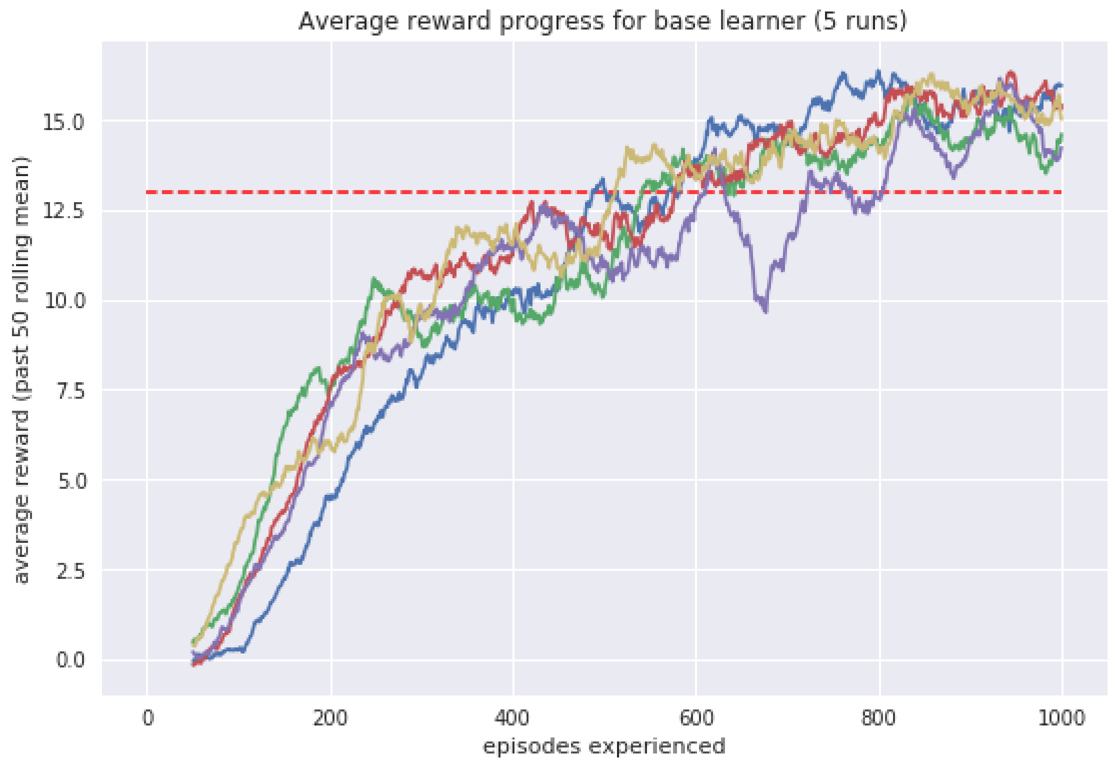
separate actor and evaluator networks (double DQN): the learner has a separate network for choosing actions (the actor network) and estimating the value of the next state (the evaluator network for calculating the target value). However, the networks are not trained separately – the evaluator's weights are soft-updated from the weights of the actor network – so this is a very simple form of a DDQN.

epsilon decay: the learner starts with a high epsilon, and decays gradually to a low epsilon

small network

In [17]:

```
Average episodes to solve environment: 549
```

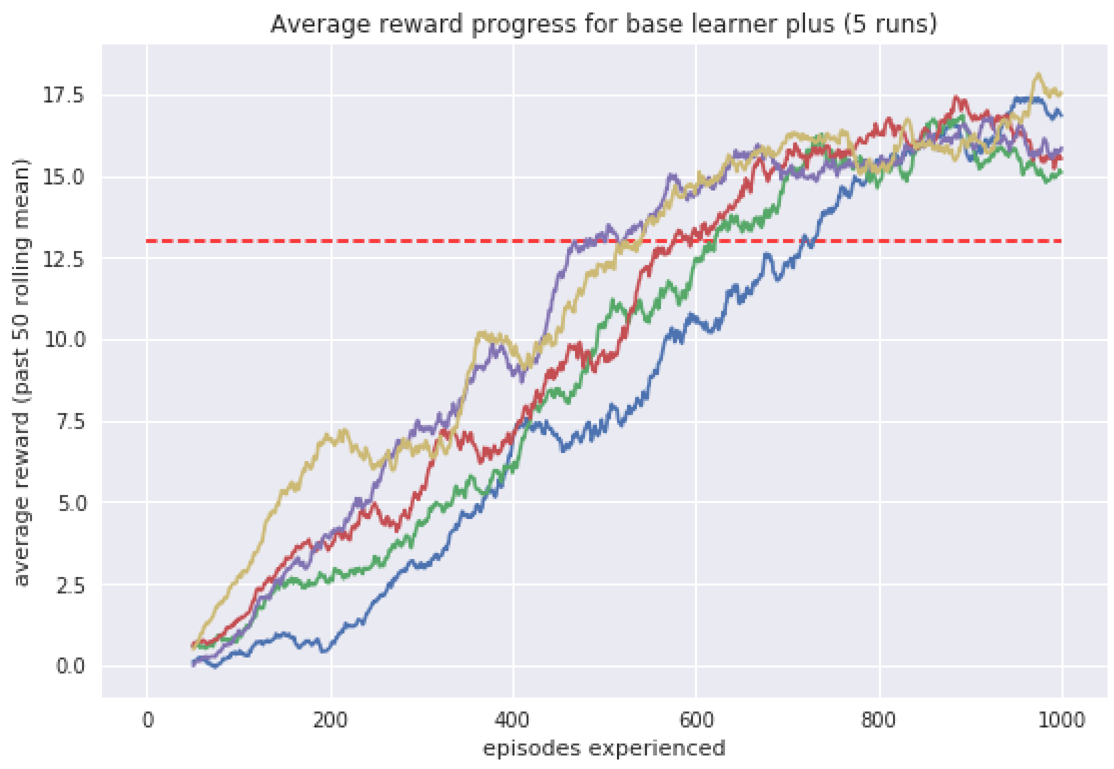


large network

With a bigger network (four hidden layers instead of one), the base learner actually performs slightly worse.

```
In [18]:
```

```
trial 0 begun...
Warning: insufficient stored steps; skipping training
trial 0 completed
trial 1 begun...
Warning: insufficient stored steps; skipping training
trial 1 completed
trial 2 begun...
Warning: insufficient stored steps; skipping training
trial 2 completed
trial 3 begun...
Warning: insufficient stored steps; skipping training
trial 3 completed
trial 4 begun...
Warning: insufficient stored steps; skipping training
trial 4 completed
Average episodes to solve environment: 572
```



```
CPU times: user 35min 23s, sys: 3min 27s, total: 38min 50s
```

```
Wall time: 1h 7s
```

3. Advanced learner

This Q-learning agent is equipped with several additional tricks for faster learning. In addition to the epsilon decay and memory replay used in the base learner, it implements the following:

double DQN (citation): as in the base learner, the agent has two separate networks – one for choosing actions, and another for evaluating them. However, in this version, the models are trained separately on different random samples from the memory replay, which ensures they won't have any interdependencies that could lead to harmful feedback loops.

prioritized replay (citation): to learn, the agent selects randomly from its stored experiences. However, the experiences are weighted by model error, so that the agent is more likely to pick experiences that it has more to learn from. To ensure that new (never-learned-on) experiences get a priority bump, the first-time error is multiplied by 2.

multi-step Q-learning (citation): instead of the traditional Q-learning reward formula:

```
total_reward = reward + gamma\*Q_evaluator(next_state)
```

total_reward is calculated using the discounted actual rewards of the next *n* timesteps:

```
reward_0 + (gamma**1)*reward_1 + (gamma**2)**reward_2 ... + (gamma*
*n) * Q_evaluator(nth_state)
```

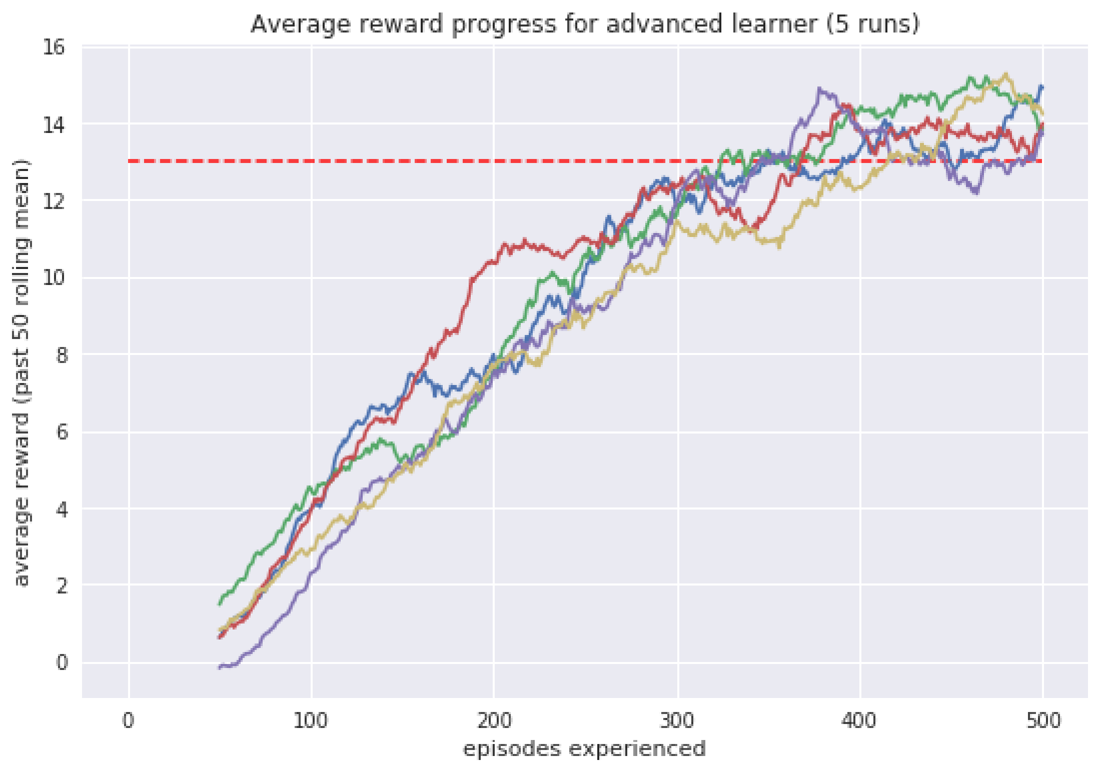
NOTE: as before, these hyperparameters have been chosen after dozens of tests in other notebooks. (For example, while raising n_multistep to 8 resulted in a faster solution, raising all the way to 20 causes the learner to peak at an average reward at 6.) Repeated trials with these optimal hyperparameters place the runs-required-to-solve in the 175-250 range

small network

```
In [5]:
```

```
Running on cpu

trial 0 begun...
Warning: insufficient stored steps; skipping training
trial 0 completed
trial 1 begun...
Warning: insufficient stored steps; skipping training
trial 1 completed
trial 2 begun...
Warning: insufficient stored steps; skipping training
trial 2 completed
trial 3 begun...
Warning: insufficient stored steps; skipping training
trial 3 completed
trial 4 begun...
Warning: insufficient stored steps; skipping training
trial 4 completed
Average episodes to solve environment: 354
```



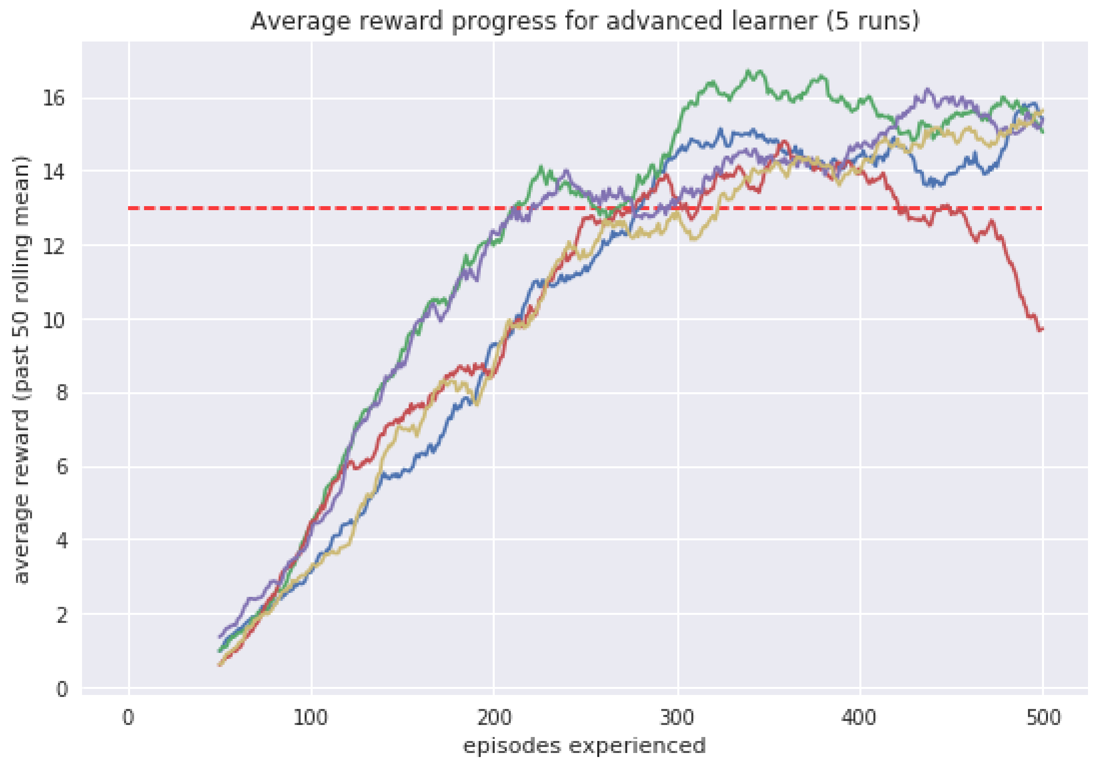
```
CPU times: user 1h 15min 54s, sys: 2min 15s, total: 1h 18min 10s
Wall time: 1h 28min 24s
```

large network

NOTE: the trial run below whose score heavily drops off at the end is fairly anomalous; it's the only case I observed out of about 15 total runs with these parameters (though I did see this result a little more commonly with different, suboptimal parameters).

```
In [6]:
```

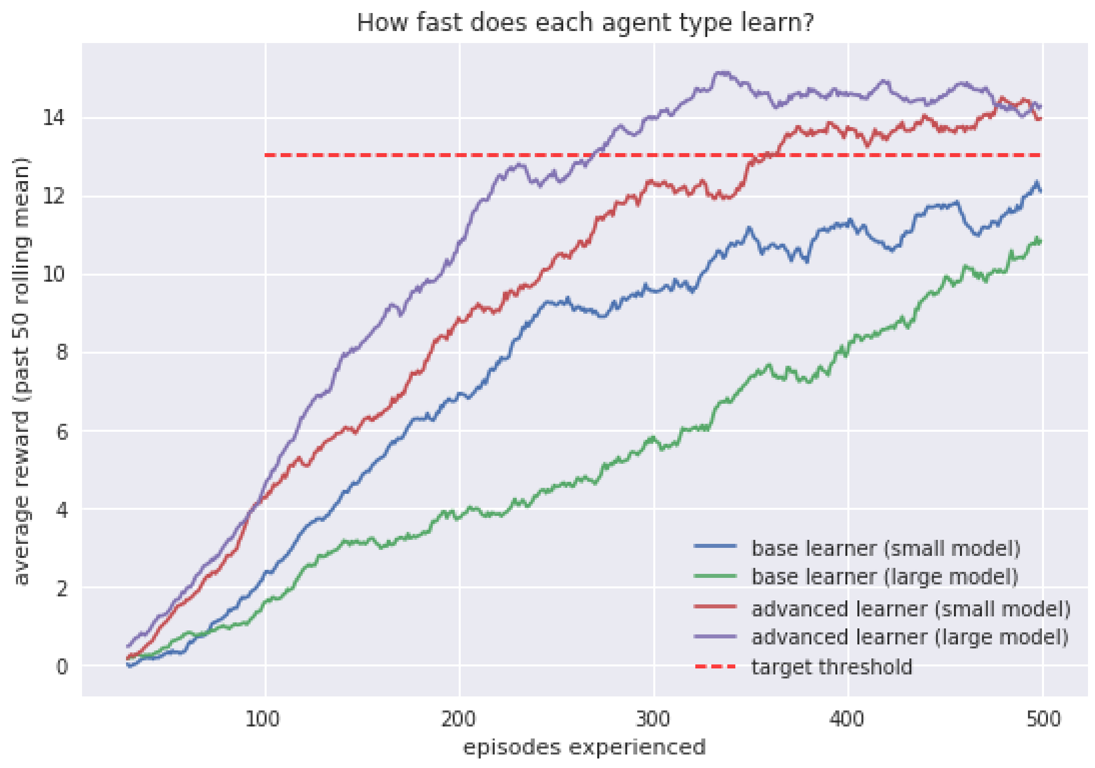
```
trial 0 begun...
Warning: insufficient stored steps; skipping training
trial 0 completed
trial 1 begun...
Warning: insufficient stored steps; skipping training
trial 1 completed
trial 2 begun...
Warning: insufficient stored steps; skipping training
trial 2 completed
trial 3 begun...
Warning: insufficient stored steps; skipping training
trial 3 completed
trial 4 begun...
Warning: insufficient stored steps; skipping training
trial 4 completed
Average episodes to solve environment: 245
```



```
CPU times: user 1h 18min 30s, sys: 2min 1s, total: 1h 20min 32s
Wall time: 1h 30min 48s
```

Conclusions

```
In [18]:
```



In this experiment, I've demonstrated that my advanced learner can solve Unity's banana-collection environment in about 250 steps. As the plot above shows, the combination of prioritized replay, double DQN, and multi-step learning cut the learning time by approximately half, which is a large improvement. This demonstrates the effectiveness of these techniques in combination.

Additionally, a larger network (4x64 hidden layers) provided a significant boost to the advanced learner, but was actually detrimental to the base learner.

The primary caveat to these results is that they do require extensive hyperparameter tuning. Each trick added to the agent introduces more hyperparameters, each of which multiplicatively increases the number of iterations required to optimize hyperparameters. So while adding more of these tricks was hugely helpful for the learning speed, it drastically increased the meta-learning speed. This tradeoff will not always be worth it in real-world use cases.

Future Work

There are several other Q-learning tricks that I have not employed in the advanced learner. Most notably, the Distributional DQN technique pioneered by Bellemare, Dabney, and Munos in 2017 has been shown to add further learning speed to combinations of other optimizing techniques like those used here (Hessel et al., 2017). Implementing a distributional value estimation strategy could allow our advanced agent to learn even faster.

Sources

Hessel et al., 2017. "Rainbow: Combining Improvements in Deep Reinforcement Learning". https://arxiv.org/pdf/1710.02298.pdf (https://arxiv.org/pdf/1710.02298.pdf)

Hasselt, Guez, & Silver, 2015. "Deep Reinforcement Learning with Double Q-learning". https://arxiv.org/pdf/1509.06461.pdf (https://arxiv.org/pdf/1509.06461.pdf)

Schaul et al., 2016. "Prioritized Experience Replay". https://arxiv.org/pdf/1511.05952.pdf (https://arxiv.org/pdf/1511.05952.pdf)

Bellemare, Dabney, & Munos, 2017. "A Distributional Perspective on Reinforcement Learning". https://arxiv.org/pdf/1707.06887.pdf (https://arxiv.org/pdf/1707.06887.pdf)