Udacity Deep Reinforcement Learning Nanodegree

Project 1: Navigation

This report details the implementation of a reinforcement learning algorithm to solve the navigation problem in Unity's Banana Collector environment.

Overview

1. Establish a baseline agent
2. Adapt reinforcement learning algorithm to this problem
3. Test various training hyperparameters find fastest-training parameters

Baseline Agent

The baseline agent provides a point of comparison for our trained agents. Here, we use a **random action policy** that completely ignores all state inputs as our baseline agent. The baseline agent was run 100 times, and received a reward of $\boxed{-0.2 \pm 1.2}$. Here (as in the rest of this report), rewards are reported as [mean $\pm$ standard deviation of the mean] over the number of trials.

Reinforcement Learning Algorithm

The Deep Q-Learning algorithm is as follows:

- Initialize replay memory D with capacity N
- Initialize action-value function $\hat{q}$ with random weights $\boldsymbol{w}$
- Initialize target action-value weights $\boldsymbol{w}^{-} \leftarrow \boldsymbol{w}$
- For the episode $e \leftarrow 1$ to M:
    - Initialize input frame x_1
    - Prepare initial state: $s \leftarrow \phi(< x_1 >)$
    - For time step $t \leftarrow 1$ to T:
        - Sample:
            - Choose action A from state S using the epsilon-greedy policy
            - Take action A, observe reward R, and the next input frame x_{t+1}
            - Prepare next state $S' \leftarrow \phi(< x_{t-2}, x_{t-1}, x_t x_{t+1} >)$
            - Store experience tuple (S, A, R, S') in replay memory D
            - $S \leftarrow S'$
        - Learn:
            - Obtain a random minibatch of tuples (s_j, a_j, r_j, s_{j+1}) from replay memory D
            - Set target $y_j = r_j + \gamma \max_{\mathrm{a}} \hat{q}\left(s_{j+1}, a, \boldsymbol{w}^{-}\right)$

- Update: $\Delta \boldsymbol{w} = \alpha \left(y_j - \hat{q}(s_j, a_j, \boldsymbol{w}) \right) \nabla_w \hat{q}(s_j, a_j, \boldsymbol{w})$
- Every C steps, reset: $\boldsymbol{w}^- \leftarrow \boldsymbol{w}$

In deep-Q learning, the **policy** – the mapping between states and actions is stored in a neural network, with an input layer of size 37 (corresponding to the size of the state vector), followed by two layers of size 64, and finally, and output layer of size 4.

Here, we actually do not use a deterministic policy directly from the neural net, but rather use an **epsilon-greedy** policy, which is a type of stochastic policy, mapping states to action *probabilities*. Our policy π is defined as

$$\pi(a|s) \leftarrow \begin{cases} 1 - \epsilon + \epsilon/|A(s)| & \text{if } a \text{ maximizes } Q(s,a) \\ \dfrac{\epsilon}{|A(s)|} & \text{else} \end{cases}$$

where *Q* is the action-value function, *s* is the state, $a \in A$ is an action *a* in the full set of actions *A*, and ϵ is a parameter from 0 to 1. When $\epsilon = 1$, the agent always chooses a random action, and when $\epsilon = 0$, it always chooses the action that maximizes the action-value function; values between 0 and 1 change the probability linearly. Since we randomly initialize the policy, it will likely perform poorly at the beginning, and we would be better off using a random selection (taking off-policy actions) to explore possible actions. Later on, we'd expect that the policy will have developed so that it would perform better than random selection. Hence, we begin with a high value of ϵ to begin with, and end with a low value.

Since there are many possible hyperparameters to tweak for the training process, we will focus on two specific parameters to better understand their influence on training. We will focus on observing the effects of the ending ϵ value – the lowest value ϵ can take, and the ϵ-decay rate – the rate at which epsilon decreases from 1 (at the beginning of training).

We will train with a maximum of 500 episodes, with the training ending early if the objective is met.

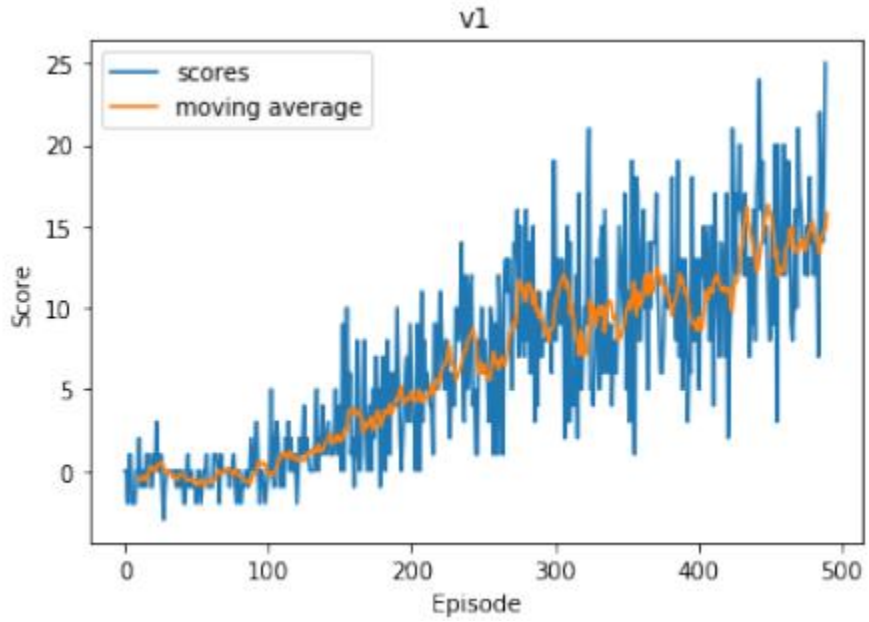
Testing Hyperparamters

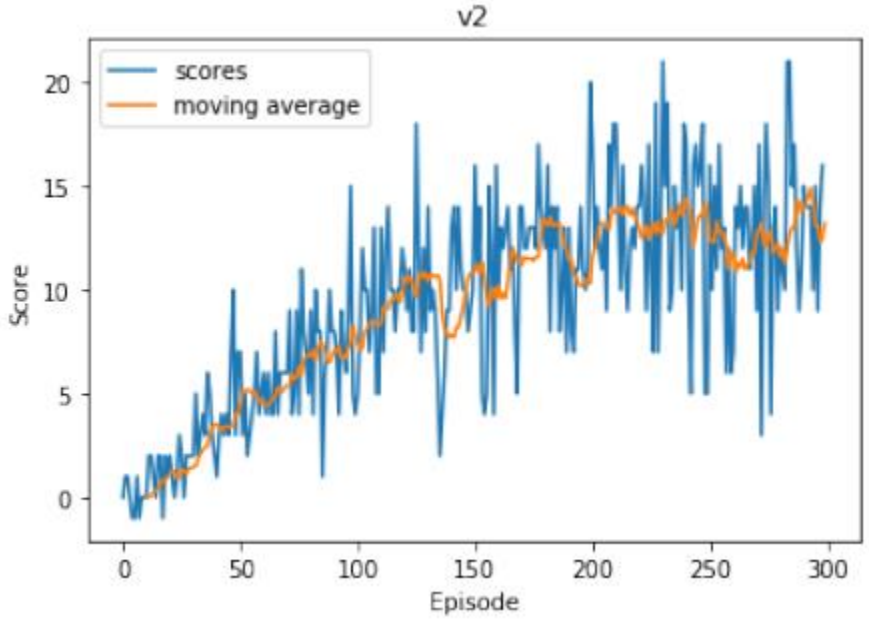
Here we report the scores as means and standard deviations of the last 100 episodes.

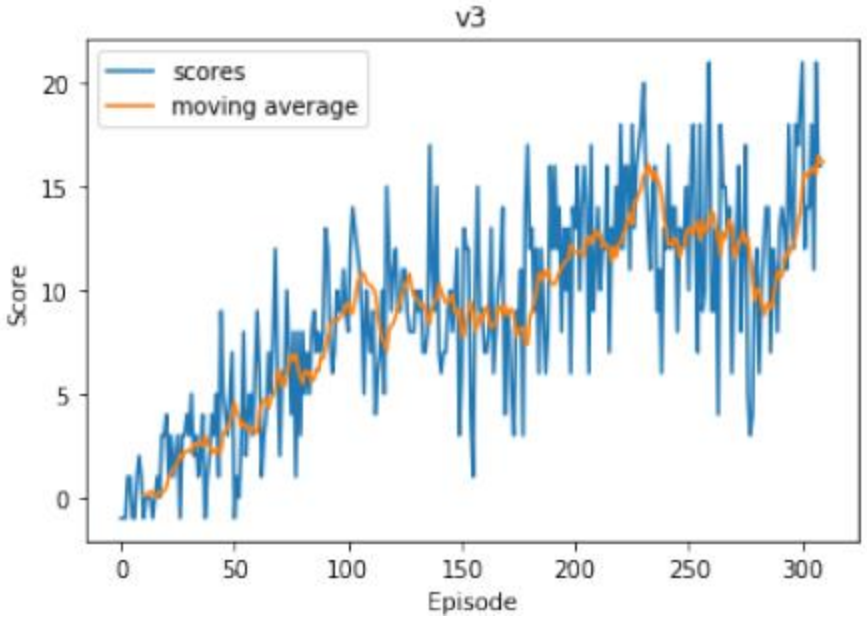
Test Version	**Ending ϵ**	**ϵ Decay Rate**	**Episodes Until Solved**	**Scores**
v1	0.02	0.99	490	13.08 ± 4.66
v2	0.05	0.99	299	13.02 ± 3.85
v3	0.08	0.99	309	13.01 ± 3.88
v7	0.11	0.99	477	13.04 ± 3.74
v8	0.13	0.99	243	13.01 ± 3.08
v4*	0.05	0.90	102	13.01 ± 4.37
v5	0.05	0.80	52	13.17 ± 5.10
v6	0.05	0.70	22	13.00 ± 4.85

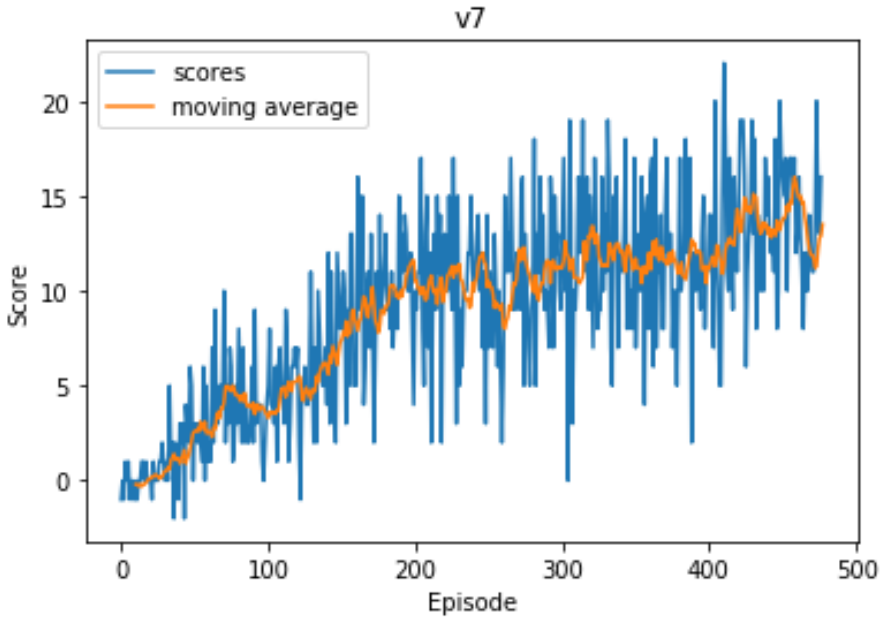
* v4 is included for the Udacity project submission as a representative “best agent,” even though it required more episodes to solve than v5 and v6, since it includes more than 100 episodes, per the solution requirement.

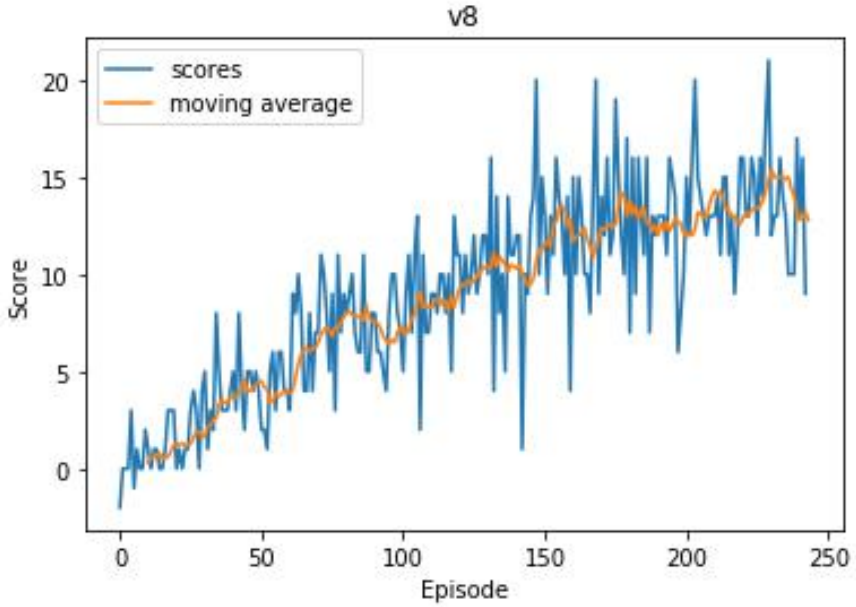
Training Plots

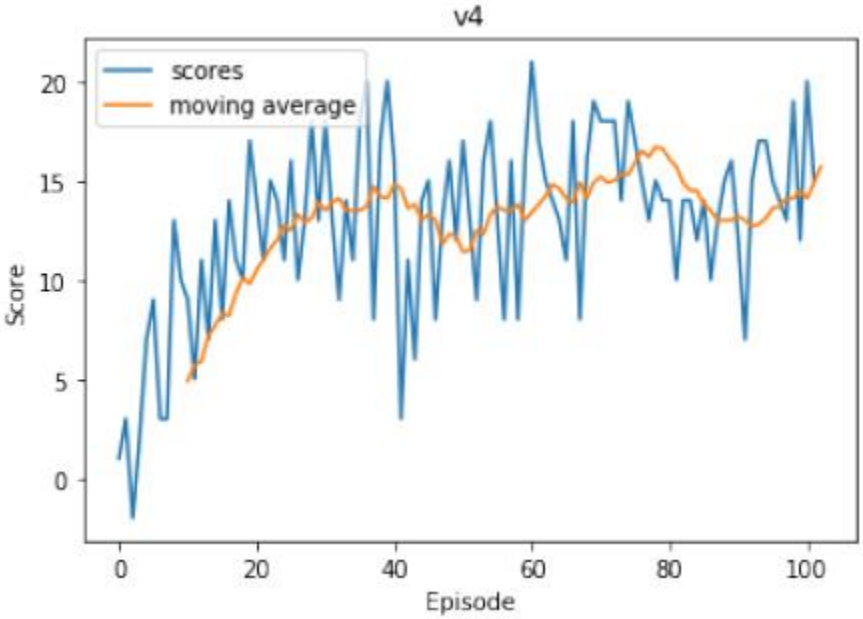


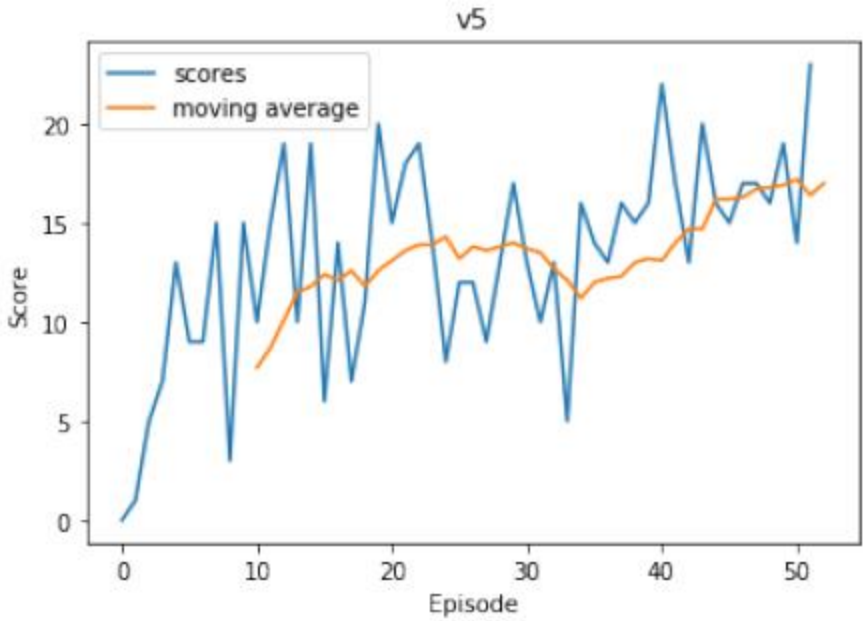


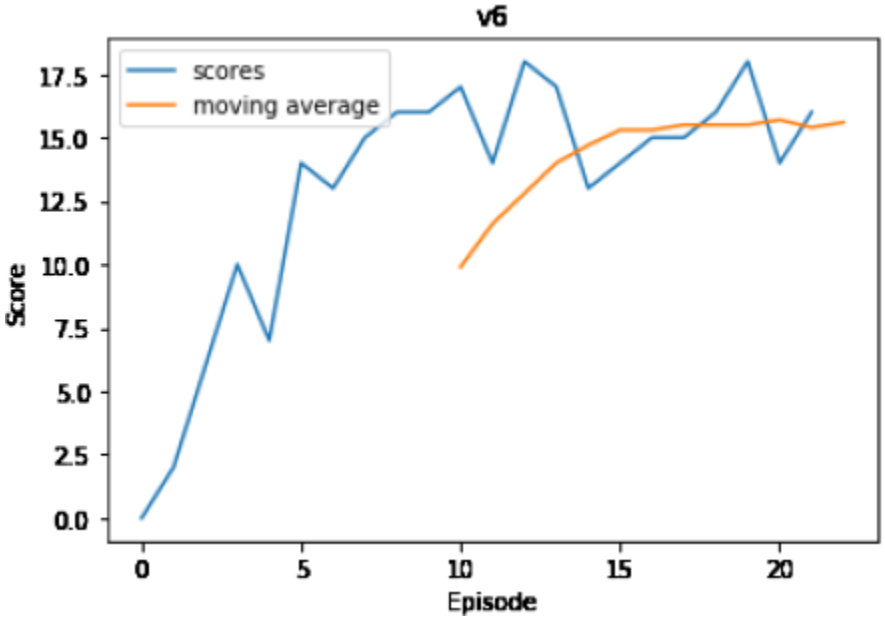

v3
scores
moving average
Score
Episode
0
5
10
15
20
0
50
100
150
200
250
300


v7
scores
moving average
Score
Episode
0
5
10
15
20
0
100
200
300
400
500


v8
scores
moving average
Score
Episode
0
5
10
15
20
0
50
100
150
200
250


v4
scores
moving average
Score
Episode
0
5
10
15
20
0
20
40
60
80
100





Discussion

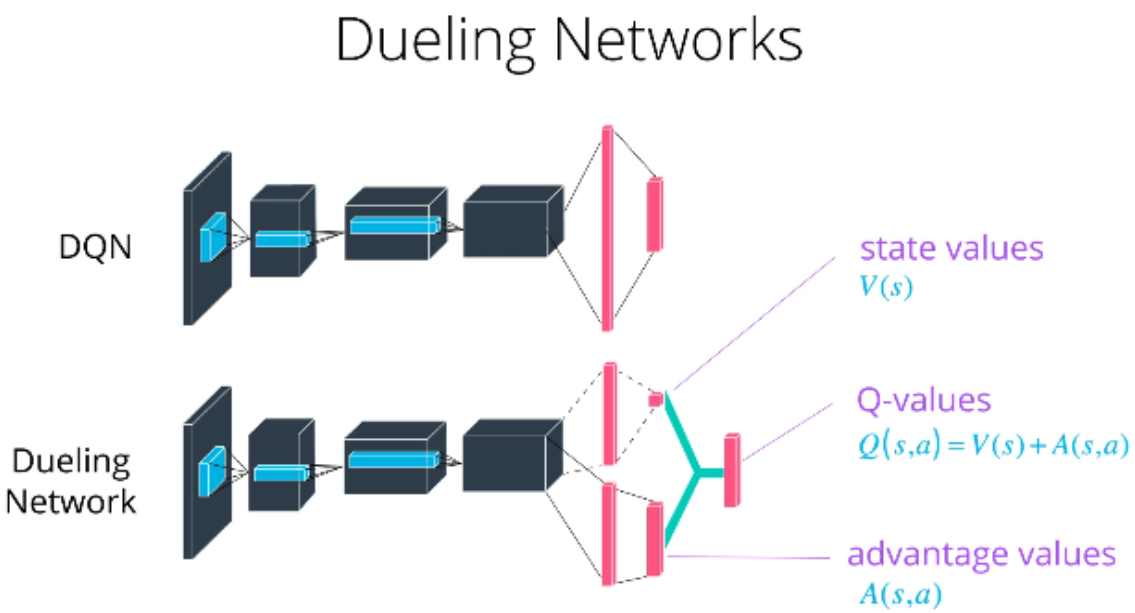
From these results, it is unclear what the effect of changing the ending ϵ value does. The difficulty in interpreting this likely comes from the randomness inherent in the training, and especially in stopping condition for the training. Perhaps a better way to do this analysis would have been to stop the training when the average reward hit a minimum threshold *and* the standard deviation crossed below a given threshold, indicating that the training had reached a sort of stable plateau.

However, it is quite clear that decreasing the ϵ decay value has significant implications for training. The fastest training time was just 22 episodes, which used an ϵ decay of 0.7. This makes sense because the lower the decay value, the faster the agent moves away from using a random policy to using its own learned policy. Having too low of a decay value might cause a poor policy to be learned and reinforced, but that does not appear to happened here.

Ideas for Future Work

Prioritized Experience Replay – Instead of just using the replay buffer by selecting random episodes, sample in a way that weights samples with higher temporal-difference errors more. This tends to decrease training time because we'd expect to learn more from cases where we encountered more error.

Dueling DQNs – Use a network architecture that splits at the end before recombing, with one end of the split estimating Q values, and the other estimating advantage values. The advantage values tell us how much the actions that an agent takes at each state matter in the eventual Q value, and is expected to change much more than the state values would.



Convolutional DQN – In the current implementation, state information – including visual information –is given to the agent in a single vector. We know, however, that for visual information, neighboring pixels tend to have related information. A convolutional network would allow us to exploit this spatial

information in the pixel values to simplify the input vector before it passes through the rest of the network.

Autoencoders – Another way to simplify the input would be to train a neural autoencoder to compress the information in the input vector. The autoencoder decreases the dimensionality of the input while attempting to retain as much information about the input as possible. Using this would require pre-training an entirely different network for compression, but may result in faster learning times as the information passing through the DQN would already be simplified.