Going Bananas Report

Implementation

As discussed in the README.md file for this project, this project solved the Unity ML-Agents Banana.exe/app environment using a deep reinforcement learning model. The method chosen to solve this problem was a DQN (Deep Q Network) as described in the "Human-level control through deep reinforcement learning" by Mnih, et. al. A soft update and custom epsilon decay functions were also used to help learning progress.

Stable Deep Learning

In order to stabilize the learning performance of a neural network for reinforcement learning problems, DQN creates a buffer of prior experience tuples (state, action, reward, next_state, done) to be used during training. These prior experience examples are sampled randomly when learning updates are performed.

DQN uses a unique loss function to stabilize learning

$$L_i(\theta_i) = E_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma \max_{a'}(Q(s',a';\theta_i^-)) - Q(s,a;\theta_i)\right)^2\right]$$

Where D is the dataset containing the experience tuples buffer, $U(D)$ is a uniform sampling distribution from D, r is the reward for the experience example selected, gamma is the discount rate, θ_i represents the weights of the "local" network and timestep i, and θ_i^- represents the weights from the target network at timestep i. This target network gets updated more smoothly/slowly and is therefore a more stable estimate of action-state values. The prime symbols (e.g. s' and a') represent value for the next time step (e.g. s' means next state and a' means next action).

What makes this loss function unique is that it uses 2 different neural networks to estimate the loss (local and target networks). The target network gets updated slowly according to the hyperparameter tau as explained in the table below, but the local network aggressively "learns" with each update. At a high level, this loss function is taking the squared error between a conservative/stable estimate of the value of state s and action a (the actual reward received plus the discounted value from the more stable target model) and the more aggressive guess of that value prior to taking action a.

Hyperparameters

The DQN algorithm requires the user to define several hyper-parameters for training to be successful. A summary of these hyperparameters are their selected values are listed in the table below.

Hyperparameter	Summary Description	Value Used
Buffer Size	How many (state, action, reward, next_state, done) to store in memory at a time	100,000
Batch Size	How many examples from the buffer to use for training when performing a learning update	64
Gamma(γ)	Discount factor for rewards. Gamma=1 means no discount, Gamma=0 means only consider immediate rewards	0.99

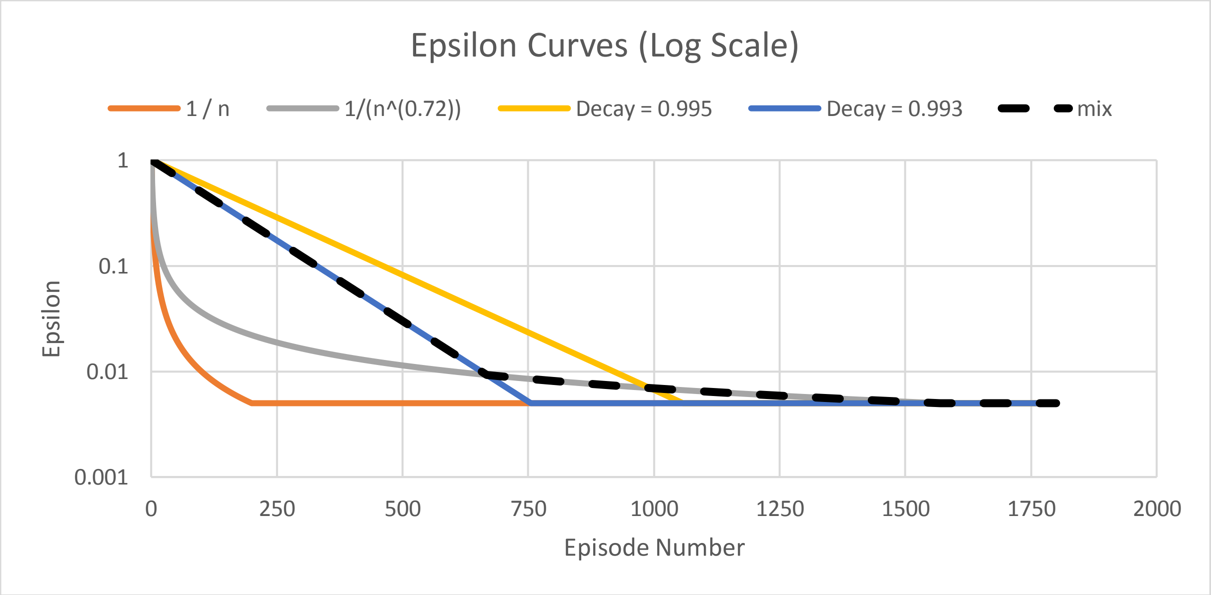
Tau(τ)	When performing model update, the weighting that gets applied to locally modified model weights compared to the more stable target model weights (θ_target = τ*θ_local + (1 - τ)*θ_target)	0.001
Update Every	Determines how often (in episodes) the target model will perform a learning update from information learned in the local model.	4
Weight Decay	Parameter that can be used to perform regularization of the neural network weights.	0.0

Epsilon Greedy

The learning algorithm I trained uses an epsilon-greedy method for selecting new actions while being trained. The epsilon represents the probability of choosing an action at random instead of following what is currently expected to be the "best" action in the given state. From experience, I have found that finding the right decay function(s) for epsilon can greatly increase the speed of training. Three techniques are common for calculating epsilon

1. Using a 1/n decay (or similar) rate where n is the number of episodes completed in training so far
2. Starting epsilon at a high value (usually 1.0) and then multiplying it by a decimal value just below 1.0 (e.g. 0.995) after each episode.
3. Setting it to a static value (e.g. 0.1), or creating a minimum value for the methods above

I experimented with various combinations of the above and ended up using the maximum value from all three as shown in the "mix" black dashed line in the following chart. This method allows the algorithm to use multiplicative decay (#2 above multiplying by 0.993) from episode 1 to ~690, then from ~690 to ~1500 it used the epsilon from #1 above, but with 1/(n^(0.72)). And finally, epsilon settles to its minimum value of 0.005, or ½ of a percent chance of acting randomly, after ~1500 episodes.



Code Base

The actual implementation of this solution can be found in the following 3 file on GitHub at https://github.com/jedisom/dqn_bananas.

1. model.py; defines the structure of the neural networks that are used by the learning agent
2. dqn_agent.py; creates a python Class that defines how the learning agent acts when asked to provide an action, learn from a time step, etc.
3. Navigation.ipynb; provides the higher level learning architecture that loops through episodes and updates the epsilon value.

Neural Network Architecture

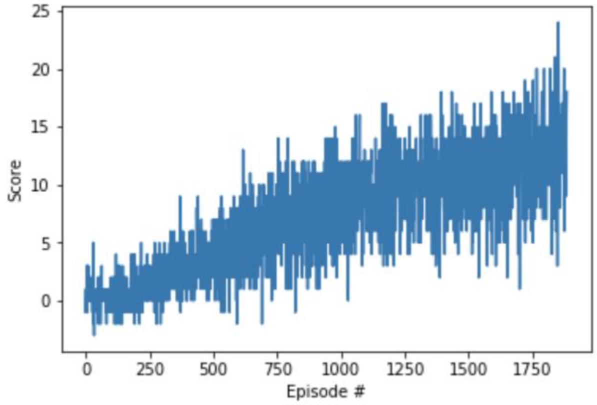
The Q network architecture must have 37 inputs (state space per Unity program) and 4 outputs (values for each available action in the environment). The selection of layers in between the input and output layers is part of the exploration required to find a solution to this problem.

The QNetwork class defined in model.py allows the user the option to create either 2 fully connected hidden layers or 3 fully-connected hidden layers. In the final solution to this problem the neural networks had the following structure.

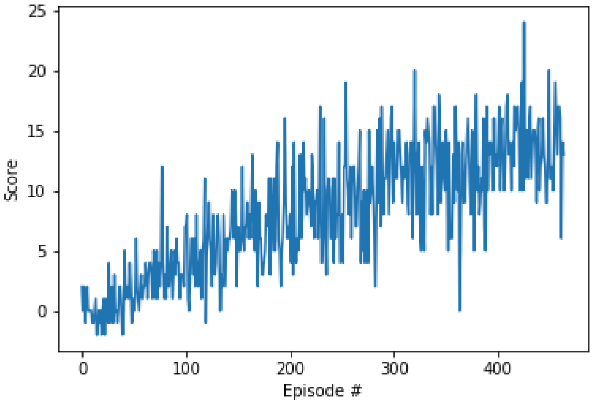
- 37 inputs from state space
- Fully-connected RELU/linear hidden layer with 150 units
- Fully-connected RELU/linear hidden layer with 75 units
- Fully-connected RELU/linear hidden layer with 25 units
- Full-connected RELU/linear output layer with 4 outputs

Results

Below is a plot of the rewards per episode experience by the agent during the final solution training, as well as the benchmark performance plot provided as part of the Udacity project description.



Benchmark Performance Plot



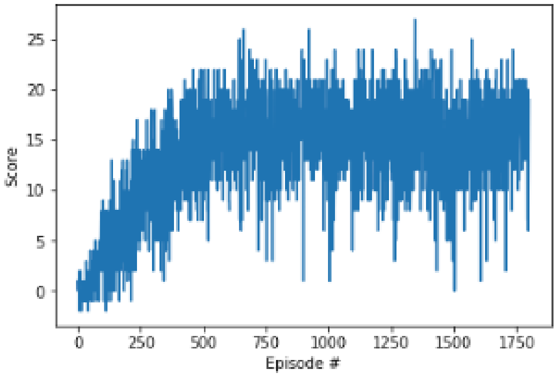
Final Solution Performance Plot

Both plots reach the benchmark performance of an average score of 13.0+ over the last 100 episodes, but the solution I found was able to gain a 13.04-point average in only 464 episodes, compared to 1800 episodes. This means that my solution was able to learn the environment in ~¼ the time (~25.7%).

Next Steps

There are lots of things that could still be implemented to improve the performance of the learning algorithm. Here's a prioritized list of improvements that could be made

1. Simply changing the agent to implement Double Q learning instead of just DQN. This fixes the bias created by taking $\underset{a}{\operatorname{argmax}} Q(s,a)$ to decide the best action to take. See "Deep Reinforcement Learning with Double Q-learning" from the team at Google DeepMind. This is simple algorithm change, that can have some nice performance benefits
2. Optimize performance using the hyperparameter τ. This parameter greatly affects how quickly the target model matches what is being observed and learned in the local model. I did not try many values for τ to solve this problem. Perhaps tuning τ and epsilon together might yield even faster solution convergence.
3. Implement a prioritized experience replay buffer. Right now, the DQN algorithm just samples prior experiences using a uniformly random distribution. Per "Prioritized Experience Replay" by Schaul et.al prior experiences can be sampled based non-uniformly based how big the TD error term is. Based on the results in that paper, this speeds up the learning a lot because
4. I also tried to let my algorithm run the full 1800 episodes to see how well it could perform with more training time. I got the following performance curve.



Interestingly, as I observed the training, it peaked around a 16-point average for 100 episodes. It then started to decrease slightly and bounce around randomly. I think that this may be due to overfitting of the neural network. I can explore using small values for the weight decay hyperparameter since that will introduce regularization and might prevent this overfitting.