Reinforcement Model for Games/Simulations

Travis James Smith

**Problem Statement and High-Level Overview:**

The purpose of this project is to explore the application of Deep Q-Learning Networks (DQN) in the training of agents to play games and perform simulated tasks. This is a well-defined task that’s common in the literature, but optimized models are still somewhat difficult to implement and train.

We’ll be implementing two models — the first is a more basic DQN, without some of the optimization techniques required to learn more difficult tasks. The goal of this model is to play a game called “Lunar Lander,” in which the agent has to guide a simulated lunar landing module to a designated landing spot. More information about the task can be found here: <https://gym.openai.com/envs/LunarLander-v2/>

For our second experiment, we’ll attempt to train an agent to play the classic Atari game “Breakout.” The model needed for this task is more intricate, for a number of reasons, and requires optimizations not present in the lunar-lander model. We’ll use a Dueling DQN with Prioritized Experience Replay. This model *might not* converge at an optimal solution due to the added mental overhead and training time, but we’ll try! For more information about “Breakout,” see here: <https://gym.openai.com/envs/Breakout-v0/>

**Description of Technology**:

Coding reinforcement learning algorithms by hand is fairly straightforward. They can generally be written in just a few hundred lines of code. When the algorithm is mean to learn from a complex environment, however, or when the algorithm uses neural networks to approximate value functions, then additional libraries are immensely helpful.

For this project, the deep learning components were built using the Tensorflow implementation of Keras. Keras provides a pair of fluid APIs for creating neural networks — for the Lunar Lander problem we used Keras’s Sequential API, and for the Breakout problem we attempted to build the network using Keras’s functional API. Dueling DQNs possess a unique feature that makes the functional API particularly well-suited to their construction.

For more information about Tensorflow and Keras, see here: <https://www.tensorflow.org/guide/keras>

While we tasked ourselves with building the agents (including the networks and the replay buffers), for our environments we used OpenAI’s Gym. Gym is a Python package that provides a wide variety of standard benchmark environments for RL agent training. We chose to use the Lunar Lander and Breakout environments, since they’re both relatively straightforward (and we *thought* training would be relatively short for both).

Unfortunately, Gym is *not* supported on Windows. This posed something of a problem, and it took a great deal of time and error to have it work as intended. It’s **strongly** recommended that anyone attempting to run these scripts do so on a Linux or macOS environment! If you have an NVIDIA GPU, Linux is your best bet.

For more information about OpenAI and Gym, see here: <https://gym.openai.com/>

**Hardware:**

The machine I used to train these models had the following specs:

Intel Core i9-9900k @3.60GHz (16 CPUs), NVIDIA GeForce RTX 2080Ti, 32GB RAM

This machine was perfectly capable of training both models, but training the Lunar Lander model took several hours and the Breakout model several days. If your machine is equivalently spec’d and you don’t mind having it tied up for some time, feel free to run the scripts on your machine — for everyone else, I recommend maybe looking for a cloud environment. The training times on these made iteration difficult.

**Installation**:

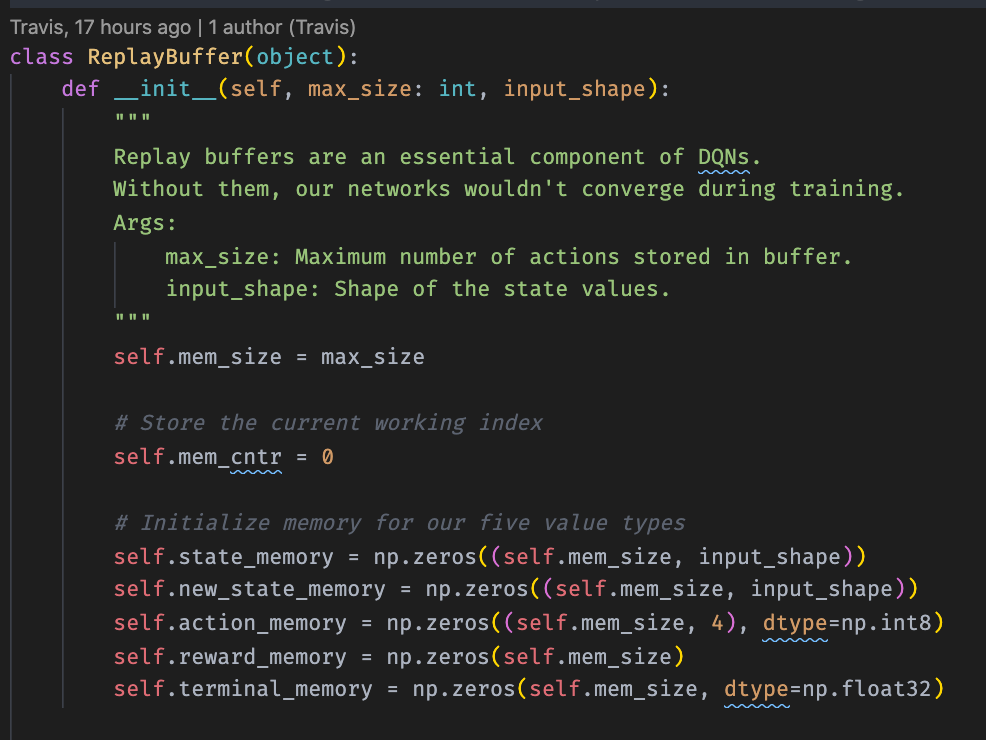
To install and run the scripts in this project, the following should work:

1. Create a conda environment and install the project requirements with:  
    `pip install -r requirements.txt`
2. Run either `lunar\_lander/train\_test.py` script with the `TRAIN\_MODEL` flag set to true  
   ——or——  
   Run the `atari/train.py` script.

This will begin training one of the two models. Model logs and checkpoints should be saved automatically as training progresses.

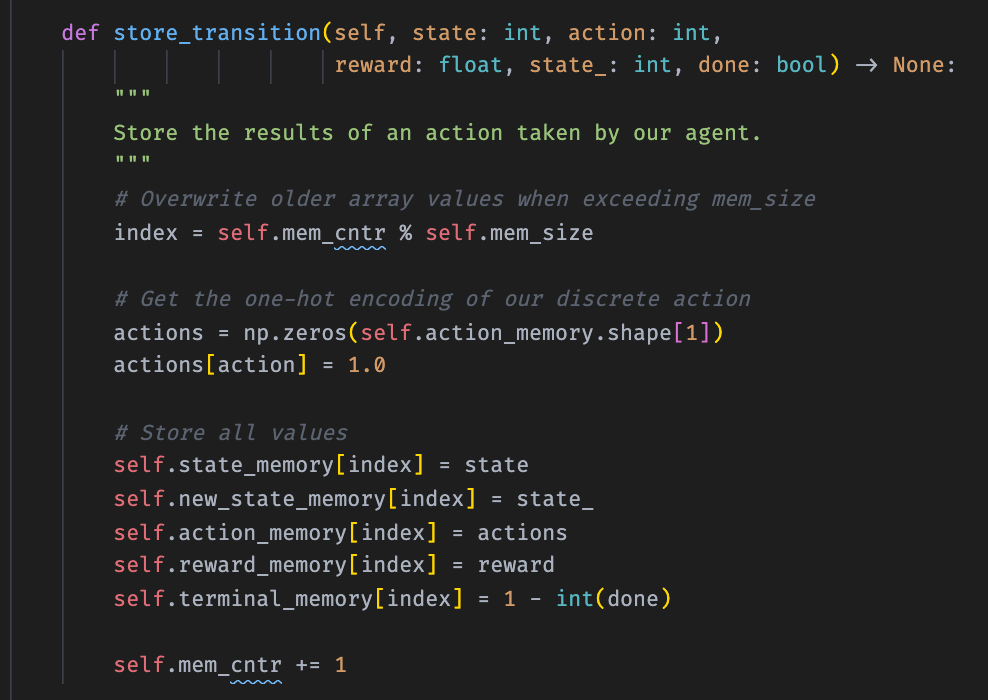
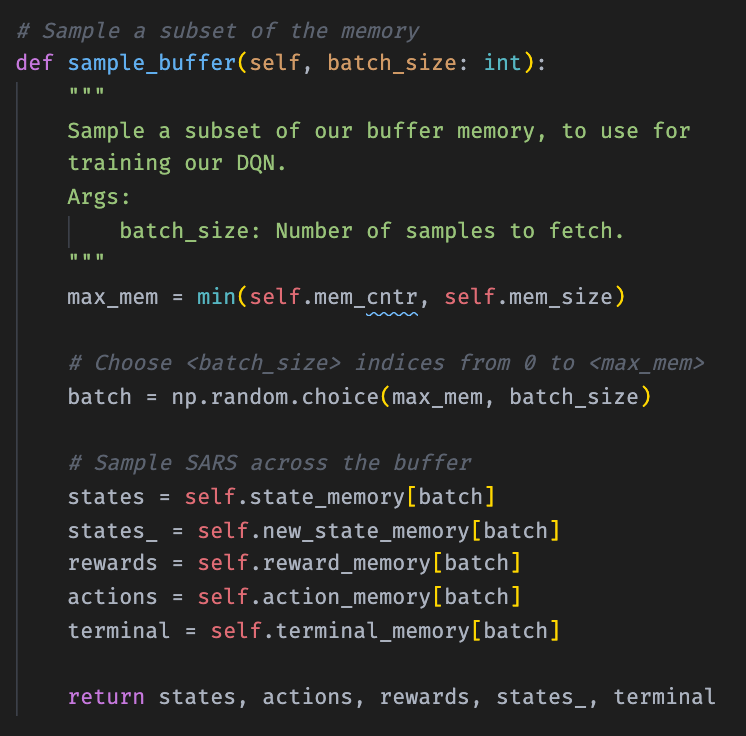
**Code:**

Because iterating on the Breakout agent was very difficult, I’ll focus here primarily on the Lunar Lander model. The project was inspired by a video demonstration of a OpenAI agent found here: <https://www.youtube.com/watch?v=5fHngyN8Qhw&t=1411s>

This model didn’t run for me at all without code modifications. *Note: If you’re reading this and don’t know much about deep Q-learning models, the following might not make sense. Please see the PowerPoint in the same directory of this project for a quick overview*.

That said, I’ll assume the reader knows what a replay buffer is and start from there.

Our replay buffer stores all of the past `mem\_size` results of the actions taken by the agent. This is necessary because training requires us to randomly sample batches from the memory (training sequentially works very poorly). Additionally, we’d like a counter telling us the present index of the buffer, so we can write to the buffer chronologically (even though we’re reading randomly.

Additionally, we need methods to store our actions and to read batches of them:

Next is our neural network construction code. It’s really very straightforward, especially compared to some of the more interesting network varieties we’ve encountered.

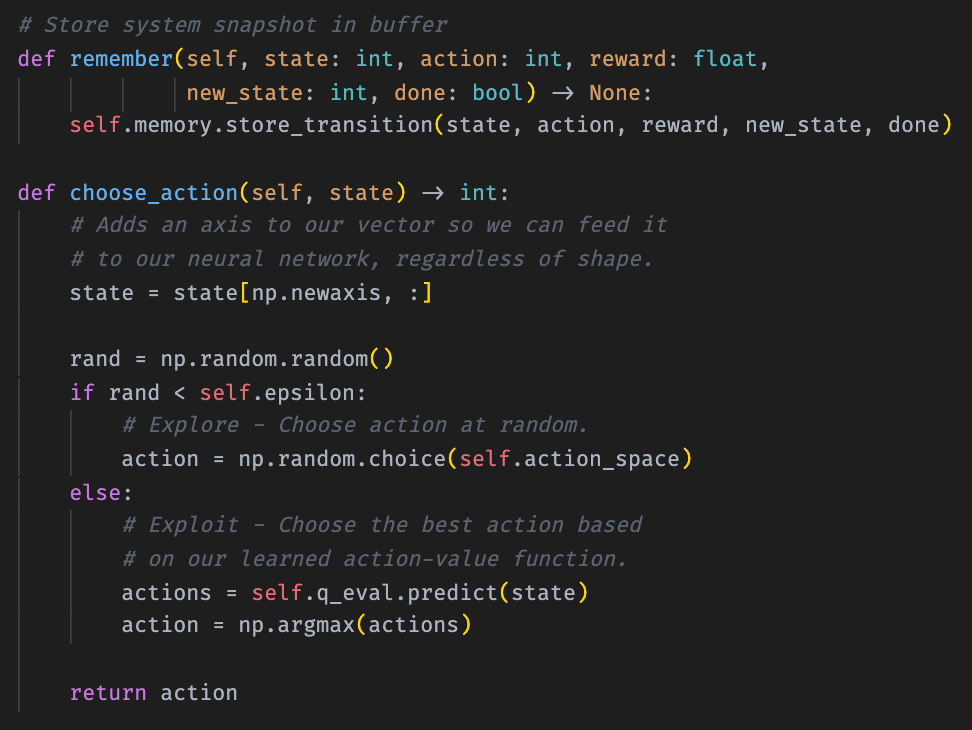
A stack of two dense layers seems to be pretty common in the deep Q-learning literature, especially around simple problems like Lunar Lander. I wasn’t able to test very many layer sizes, but ultimately one layer of size 256 feeding into a second layer of size 128 worked very admirably. The Lunar Lander environment affords an agent four possible actions at any given time (from any given state), so the two hidden layers are then connected to a final output layer with size 4.

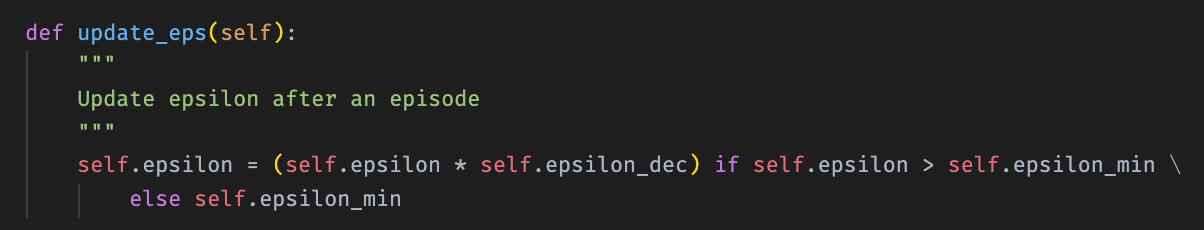
The network structure of the Breakout model is admittedly much more interesting — because it takes images as inputs, it uses convolutional network layers. Additionally, it has a specialized reinforcement learning feature that improves performance based on a unique loss calculation. I had next-to-no time to experiment with it, though, as it took forever to train, so I’m not confident showing it here (for now).

Now let’s look at the agent, which ties everything together. The agent takes a learning rate, a discount factor, a training batch size, and then information about *epsilon* (the explore-exploit hyperparameter).

you can see here that the agent initialization code also initializes the replay buffer and creates our primary neural network. Additionally, we establish a Tensorboard logger for our metrics.

Our agent needs methods to store action history and to choose an action, taking into account the current state of the explore-exploit parameter.

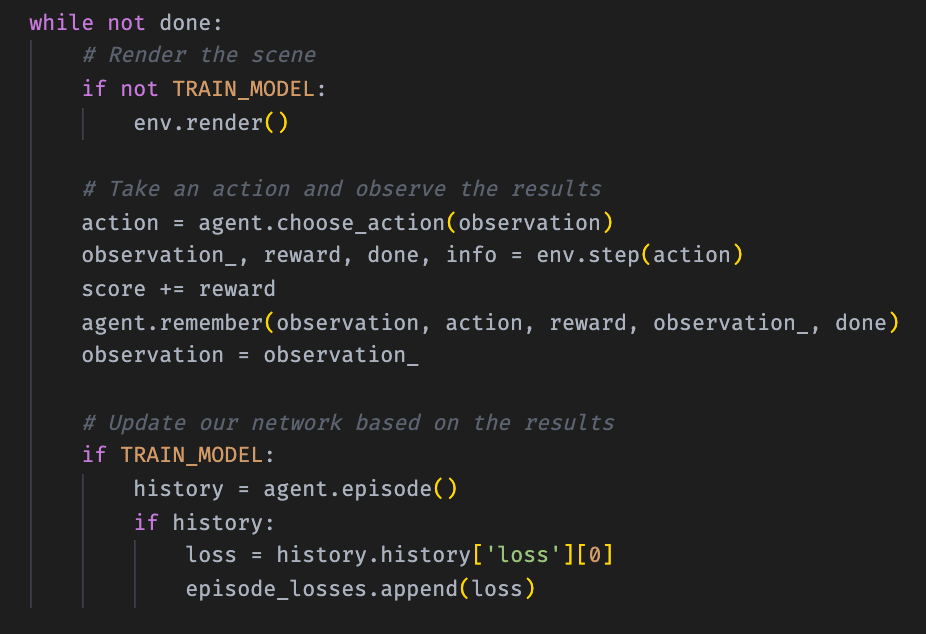
The code for these can be seen below:

Also of importance is the method by which the agent gradually reduces the exploration rate (epsilon annealing). Our agent does this multiplicatively, by taking an initial epsilon value (1.0 in our tests), a rate by which it diminishes, and a minimum value. After each epoch, the agent multiplies the current epsilon by the epsilon decrement value and stores the result as the new epsilon. Slowly over the course of training, the agent explores less and less, until it bottoms out at the minimum epsilon value. The code for this is short, seen below:

Finally, the heart of the agent learning process is the episode method. While our main training loop chooses actions, this method is where we update the value function approximator.

The method is pretty thoroughly annotated, but take special note of how the loss is calculated (using the Bellman equation and the separate Q target network copy).

First we sample a batch of past actions from our replay buffer, then we make predictions about those actions based on the current network. We then update rewards for actions based on their future rewards (discounted) and update out network with those new rewards as the targets.

For more difficult tasks, an entire second neural network is often needed, but Lunar Lander is pretty simple.

Last but not least is our training loop. That script is mostly logging and saving, but the agent takes actions in this part here.

This is pretty transparent, but note the order — it takes an action, records all information about that action into the buffer, and then performs a training episode on similar memories stored in the buffer, *at random*.

Note that it trains after each action. This is an online process, in contrast with many supervised learning techniques, and it can be slow as a result.

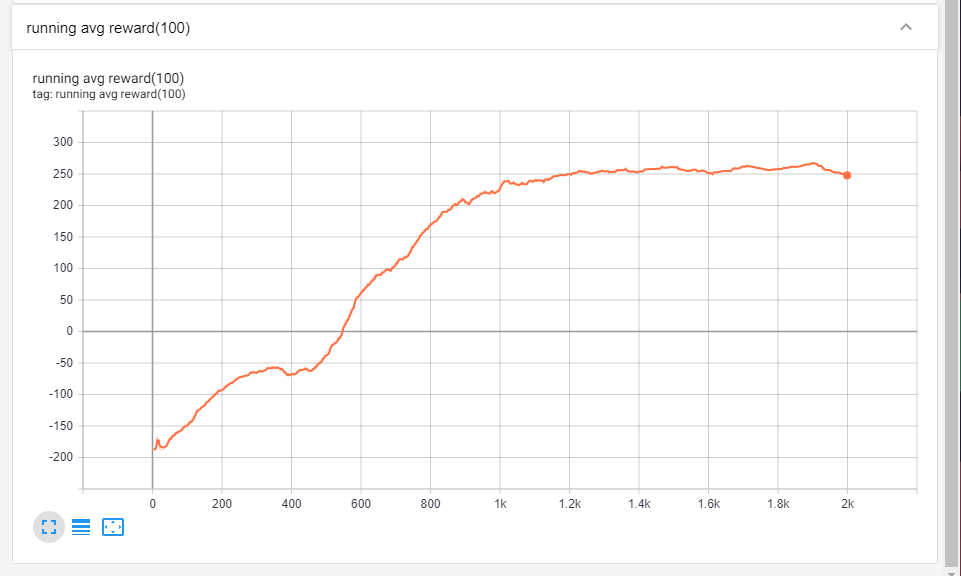
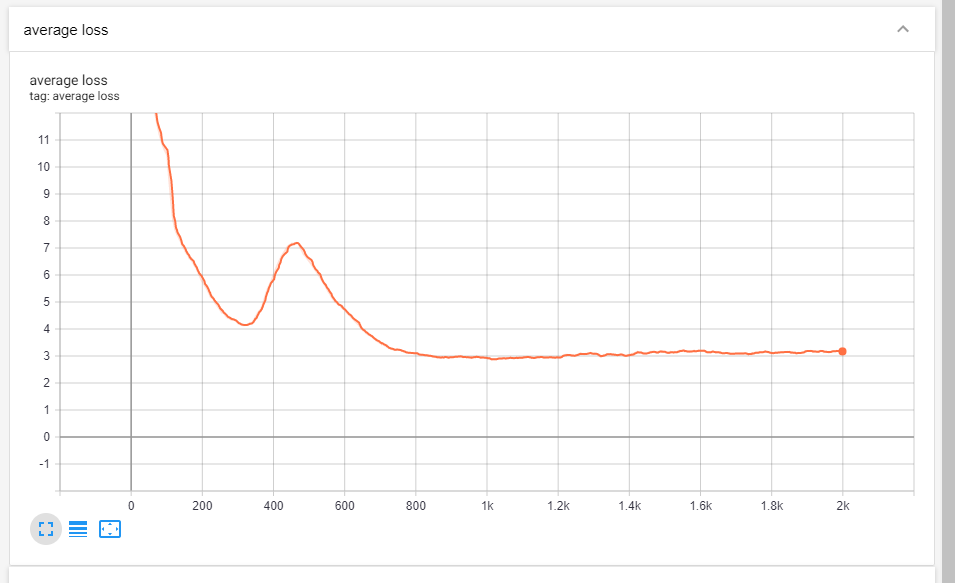
That’s it for the code — it’s really not very much. I lost the logs for most of the training iterations due to a git fiasco, but here are the logs for the training run.

You can see that the average loss decreased pretty rapidly at first, but that the model seemed to have gotten stuck in a bit of a local minimum. It found it sway out, but beyond that the loss didn’t really decrease much more.

Average total reward shows an inverse story, as we would expect.

**Summary — Pros/Cons:**

Reinforcement model training is distinct from supervised learning in a variety of important ways, and the episodic nature of the process makes it quite slow. The structures and techniques used to train the function-approximating neural networks are similar to those used in supervised learning tasks, but there’s a large body of literature (centered around concepts from control theory) that informs how those deep learning techniques can be effectively applied.

I knew a bit about RL going into this project, but I *do* feel that a lot of my time and effort was spent trying to more deeply understand what the models were attempting to do, rather than e.g. experimenting with model hyperparameters. This is to say that reinforcement learning includes a layer of conceptual overhead not present in many other deep learning tasks.

Additionally, while I was able to experiment and make significant code changes to the Lunar Lander model, training the Atari Breakout model took much too long to meaningfully experiment with it (within the window of this project). Even with a relatively powerful consumer GPU, training the Breakout model took around three days and, because model checkpoints required saving the replay buffer, many gigabytes of storage. I attempted to reduce the memory load by removing out-of-date checkpoints when possible, but this process took some trial and error (difficult when combined with the long training times).

On a positive note, there’s no data management component to reinforcement learning, at least not when training agents to play small games (like with OpenAI Gym). But I had difficulties with a number of libraries (namely Gym), and visualizing the agents requires time investment when the packages meant to do so aren’t cooperating (Gym!).

All in all, this project was great fun to work on and I feel that I learned at lot. I’d like to continue working on the Breakout model and to try training agents for other simulated tasks in the future (probably on cloud machines!).

**References:**

*Theoretical Background:*

Richard S. Sutton and Andrew G. Barto, “[Reinforcement Learning: An Introduction.](http://incompleteideas.net/book/the-book-2nd.html)”

*Implementation Guidance:*

Lilian Weng, “[Implementing Deep Reinforcement Learning Models with…](https://lilianweng.github.io/lil-log/2018/05/05/implementing-deep-reinforcement-learning-models.html)”

Ray Heberer, “[Why Going from Implementing Q-learning to Deep Q-learning…](https://towardsdatascience.com/why-going-from-implementing-q-learning-to-deep-q-learning-can-be-difficult-36e7ea1648af)”

*Machine Learning with Phil*, “[Deep Q Learning is Simple with Keras | Tutorial.](https://www.youtube.com/watch?v=5fHngyN8Qhw&t=1411s)”

Sebastian Theiler, “[Building a Powerful DQN in TensorFlow 2.0.](https://medium.com/analytics-vidhya/building-a-powerful-dqn-in-tensorflow-2-0-explanation-tutorial-d48ea8f3177a)”

**Youtube URLs:**