CS7642 - Project 2: Lunar Lander

Git SHA1: ba68eb75993bb52dd0d2c2916ff66f8a32aad3af

Ceslee Montgomery
cmontgomery38@gatech.edu

Abstract

For this project, this implementation of Mnih et al.'s DQN algorithm solved the Lunar Lander environment in 320 episodes in under 30 minutes. More impressively, the agent was intentionally implemented with a simple 3-layer neural network (unlike Mnih et al.'s fancier CNNs) updated every step in the environment, a small batch size and buffer and without fixed Q targets. After this agent was successfully implemented, experiments were undertaken to evaluate the role of each of these hyperparameters.

1 Introduction

1.1 The Lunar Lander Environment

The goal of this project is to implement a Reinforcement Learning (RL) agent capable of solving Open AI's Lunar Lander v2 environment.[1] . In the Lunar Lander environment, the objective is to land the lander on the moon. The state the agent is able to observe is an 8-tuple consisting of its position, velocity, and whether its legs are touching the ground. The four actuators the agent has available are *do nothing*, *fire left orientation engine*, *fire main engine*, and *fire right orientation engine*. The reward function contains many aspects, but essentially incentivizes the agent the closer it is to landing. The reward function penalizes crashes as well as minor faults such as a small penalty of -0.3 points for firing the main engine. The epsiode ends when the lander comes to rest, whether successfully, or otherwise, crashing, or falls off screen. Ultimately, the problem is considered solved when the agent achieves an average score of 200 over 100 consecutive runs.

1.2 Deep Q-Learning

For this project, I chose the *Deep Q-Learning (DQN) algorithm* introduced by Mnih et al.. DQN is a widely popular variant of the Q-learning RL learning algorithm that shows impressive state-of-the-art performance on several Atari games. The authors were the first to demonstrate how to leverage deep learning for state-action value function approximation capable of learning high-quality feature representations from raw pixels. The key to this innovation was the use of an *experience replay* mechanism. Supervised methods, including neural networks expect data to be *i.i.d*. However, in RL, the states encountered in an episode are highly correlated. Further, with RL, the data distribution changes as the algorithm learns new behaviors, leading to non-stationary distributions. Thus, *experience replay* is necessary to break the correlated state history by randomly sampling previous state transitions from a *n*-sized batch. Another feature that helped stabilize the deep learning model was the use of *fixed Q targets*. This effectively means bootstrapping the Q-network towards frozen Q-learning targets. However, Mnih et al.'s results showed (1) games could be solved without it and (2) a relatively minor bump in performance was attributable to fixed Q targets. Therefore, these experiments are motivated to see what the simplest DQN implementation might be that's capable of solving Lunar Lander. In fact, Mnih et al. state that simpler implementations such as this (using

[1] https://gym.openai.com/envs/LunarLander-v2/

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

experience replay and simpler neural networks) existed prior, but were not applied towards raw pixels. As solving Lunar Lander doesn't require learning from raw pixels, some could argue that DQN is overkill. Thus, this simplicity-driven approach, undoubtedly, would contribute towards a more fine-grained understanding of the contribution of each aspect of the algorithm.

2 Experiments

2.1 Experiment 1 - Successful Baseline DQN Implementation of Lunar Lander

The goal of the first experiment was to produce a working implementation of DQN capable of solving Lunar Lander, by achieving an average score of 200 points over 100 consecutive runs. To accomplish this I tested early versions of the algorithm against the Cart Pole environment[2] which is a simpler environment with continuous state spaces (these continuous states are part of what makes a more complex value function interesting). From there, I made updates to the neural network architecture and hyperparameters to solve Lunar Lander.

2.1.1 Implementation

The DQN algorithm was implemented closesly to [Mnih et al.], but as there was no raw pixel input no pre-processing was needed.

[Mnih et al.]'s DQN is implemented as follows:

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

The final successful implementation of the above algorithm was trained over a maximum of 5000 episodes, capping the maximum environment steps at 1000. The gradient descent update step occurred every environment step.

In building this model, it became clear that it was very sensitive to its chosen hyperparameters. This is explored in detail in later experiments in the paper, however, for the chosen seed (=1), standard hyperparameters were held relatively consistent through experiments. These included $\gamma = 0.99$ and $\epsilon = 1.0$, decaying at a rate of $\epsilon_{decay} = 0.995$ to a $\epsilon_{min} = 0.01$ with a learning rate of $\alpha = 5e-4$. These hyperparameters were found to balance performance, exploration, and fast computation. While further tuning could've yielded improvements, other hyperparameters were suspected to have greater influence on poor training results. As well, Mean Squared Error was the chosen loss function for the Q-network for all experiments.

The hyperparameters that were hypothesized to be the bottlenecks for performance in both Cart Pole and later Lunar Lander were related to experience replay: the replay batch size, replay memory buffer size and the Q Network architecture. Interestingly, in a talk, Mnih stated that they didn't

[2] https://gym.openai.com/envs/CartPole-v1/

expect performance to vary much by buffer size, for their work they found 32 to work well. I resisted adjusting this but eventually decreased the batch size as it appeared that increased batch sizes understandably biased the learner towards prior runs. If you think about this like a sample "window", this means it took longer for worse/older estimates to fall out of the window. This, however, is just a theory, so this is explored later in further experiments. The buffer size used during Cart Pole was much smaller, meaning only a small number of states were sampled in the much larger state space presented in Lunar Lander. Eventually, a buffer size of $1e5$ was found to perform well without overburdening available memory. Finally, equally important as the buffer size, the neural network architecture heavily affected both performance and training time. In the end, a simple neural network with three hidden layers, $64X128X64$ with ReLU activation functions was found to balance speed of training and performance.

This experiment runs in $< 20m$ with a Python 3, Numpy, Keras setup. The key victory for making this possible outside of simplifying the model as much as possible (lower batch size, buffer size, simpler NNs, no fixed Q target) was vectorizing the experience replay. This was relatively time consuming to implement without a *single for loop or if statement*, but it was worth it. A related useful strategy was batch model predictions and updates. Again, given that this step happens *every* time step in training the agent it was the computational bottleneck.

2.2 Outcome

The following graphs show that with the above implementation details, the DQN agent was successfully trained in less than 350 episodes (as mentioned as well, in less half an hour)! This was with a rather simple implementation that didn't contain the complexities of truly massive neural networks, more sophisticated architectures such as convolutional (used by Mnih et al.) or recurrent neural networks or fixed Q targets!

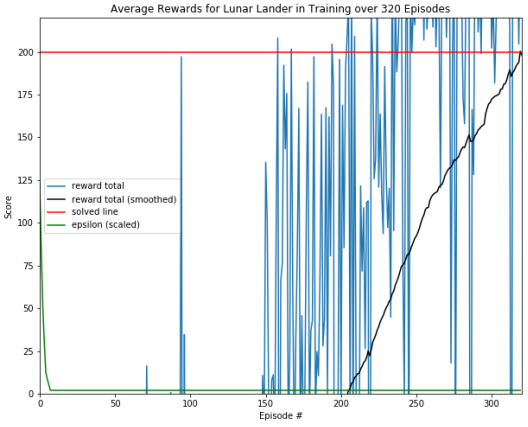


Figure 1: Successful training of DQN Implementation after 320 episodes

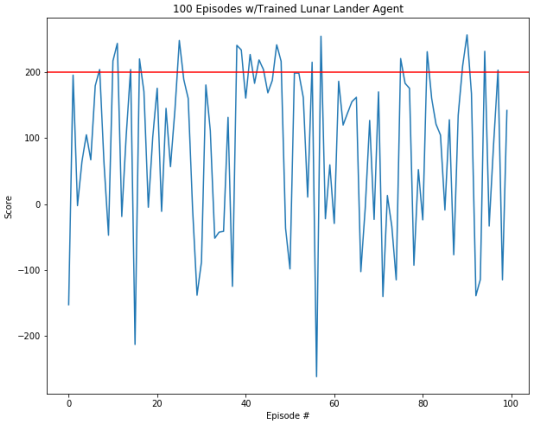


Figure 2: Successfully trained agent tested over 100 episodes

In figure 2, its shown at each episode that the rewards vary wildly, even close to the end of being successfully trained. The figure is cut off at 200 points, but after about 250 episodes its regularly exceeding the 200 score threshold. As is characteristic of TD-based methods, the learning gains of this model are impressive as evidenced by the extremely steep curve as the agent begins to regularly earn positive rewards. Figure 4 shows 100 further episodes of the trained agent (with a greedy policy of no exploration).

2.3 Experiment 2 - Effects of Batch Size, NN Size, and Update Frequency on DQN Performance

As mentioned, I was interested to experiment with the hyperparameters I expected to have the most substantial impact on the DQN algorithm performance given their role in the algorithm. For example, as discussed, because larger batch sizes create more of a computational bottleneck, I was interested to see if I had given up some performance upside by reducing the batch size. Similarly, I was interested to see what the lower limit was on neural network architecture for this environment, so I varied the number and size of hidden layers chosen. Finally, following from the fact that subsequent time steps are highly correlated, I was interested to see if the agent achieved better performance by bypassing updates, and therefore, visiting many more disparate states in the same amount of computation time.

The implementation details are consistent with the previous experiment, except that hyperparameters under test.

2.4 Outcome

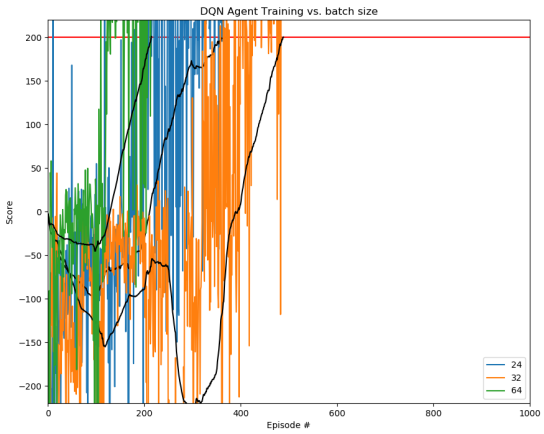


Figure 3: DQN Implementation at various batch sizes: 24, 32, 64

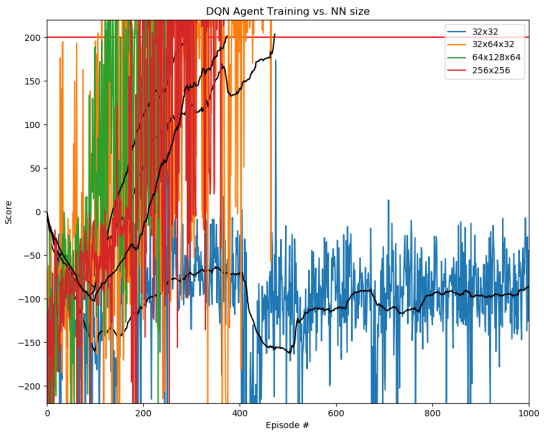
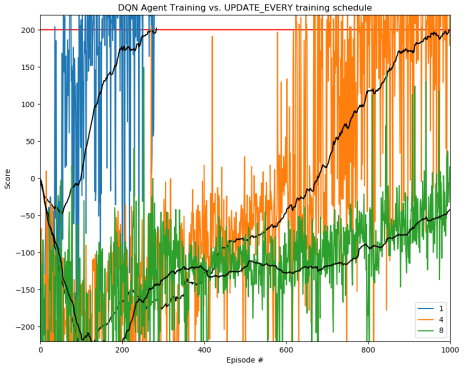


Figure 4: DQN Implementation at various NN hidden layer sizes

Interestingly, the results in Figure 4 show that an agent can be successfully trained faster with a batch size of 24 than 32. However, surprisingly, a batch size of 64 trains in even fewer episodes. If my theory was correct, the time to solving the environment would be roughly linear in batch size. However, clearly that's not the case. The strongest theory that I have for this is that this is either somehow related to the neural network architecture, as the hidden layers chosen were 64 x 128 x 64, or the performance of the algorithm is highly variable over multiple runs. An environment and Numpy seed was set, but no such seed was set in Keras. Meaningfully, Keras initializes NN model weights randomly uniformly, so some variability in results is to be expected. Upon further research, to alleviate this problem would've required non-trivial setup that might affect the computation time of the algorithm (removal of parallelization, etc). Therefore, I declined to investigate this further. Next, I investigated the effects of NN hidden layer size, while holding the activation function constant (ReLU). It is worth stating this is my first time implementing neural networks, so my next experiment begins with a naive question about how the neural network architecture is affecting the agent. Initially, I evaluated the original agent $64X128X64$ against agents with hidden layers of size $32X32$ and $32X64X32$. As shown in figure 4, reducing the neural network by half had only a marginal effect on performance, however removing the middle hidden layer completely degraded experience. However, I couldn't conclude that the problem was having 3 layers, as this didn't make sense to me as an explanation. I suspected that there exists a trade-off between nodes and number of hidden layers. In order to evaluate this, I trained another agent with 2 layers, but the same number of nodes (=65,536) as the $32X64X32$ agent. As I suspected, it falls roughly in line with the performance of the agent

with the extra hidden layer. Therefore, my takeaway is that smaller networks, balancing quantity of layers and nodes could likely still work in this environment and potentially be more computationally efficient. However, because the agent runs relatively quickly, I'm less concerned about improving run times. Finally, I ran an experiment to evaluate the effect of the update schedule on algorithm performance. Figure 2.4 shows how increasing the update schedule to every 4 time steps does still converge to a solution.



Extending the batch size to every 8 steps, however, cripples the agents learning. My hypothesis is that it will likely still converge given sufficient training episodes. But there seems to be no computation time or performance benefits of doing so.

3 Discussion

Without question the two most difficult parts of this assignment were ramping up on neural networks and how to implement them in Keras (which layer/node sizes, activation functions, loss functions, etc.) and implementing the experience replay in a computationally efficient manner. To begin, I used typical activation and loss functions and focused energy on optimizing the size of the neural network. To reduce the search space for the right size, i started with the size from the Mnih et al. paper and reduced the network over time. This was necessary as bigger networks make training iterations much slower. Original training times ran over an hour (often falling over and diverging). Secondly, two of the biggest hurdles to vectorization were: (1) reshaping input data for batch model prediction (the errors weren't the most helpful) (2) batch updating Q-targets. I wasn't sure this was possible at first, but with Numpy you can supply a row and column position vectors and an update vector and update a vector all at one time. Given more time I am interested in many extensions including: a more fine-grained analysis of update schedules ([1,2,3,4,5]), performance of fancier implementations such as fixed Q-targets, a more memory efficient/larger memory buffer, and lower-level implementation using PyTorch/GPUs for speedups. From a practical perspective, two more improvements would've made implementation more efficacious: due to MuJoCo license issues, I had to downgrade to Python 3.4 to use a specific conda package installing Box2D, to get the Lunar Lander environment working. After several hours spent on this, I settled for slightly more stale software and given more time would be interested to see how Python 3.7+ higher compatible packages would impact the results. Finally, given the variability highlighted in the hyper-parameter experiments, for publishing rigorous results it would be imperative to perform multiple runs over multiple seeds to ensure robust general performance (versus a run exploiting a lucky local optima). Taking a simple approach, as this work has done, makes multiple runs over multiple hyperparameters in multiple experiments tractable.

References

Volodymyr Mnih. Deep RL bootcamp lecture 3: Deep q-networks. URL `https://www.youtube.com/watch?v=fevMOp5TDQs`.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. page 9.