Attempting Q-Learning

A Journey of Lost Sleep

Alex Harper
Mercer University Student
Macon , Ga

***Abstract*—Detailing the attempts of a student to figure out and internalize the workings of Q learning to accomplish some prepackaged goals.**

I. Introduction

For the machine learning class, the students were told to choose a topic that was different than what had been covered thus far. This student has chosen to attempt implementing reinforcement learning. While the results are not as good as what DeepMind was able to accomplish [4], the agent was able to learn to be at least semi-effective at several tasks. The tasks that were used was the simple CartPole example [10] and one of the Atari games provided by ALE (Atari Learning Enviroment). The network used only fully-connected layers and ReLU activation.

II. Prior Examples

The most notable example of previous work was done some researchers at DeepMind where they trained an agent to play many Atari games.[4] This is what inspired the choice of project by the student who was excited about unsupervised style of training. It was his hope to be able to move past the Atari games do some more complicated examples such as the walking simulations that were made.[11]

III. Deep Q Learning

Q Learning is a style of reinforcement learning that trains an agent to know what actions are more worth-while than others. Deep-Q Learning is typically in reference to neural nets of sufficient depth when the Q Learning method is used to train. The point of this training is to make approximations of how valuable each action is. The value of the action is supposed to represent how much reward is expected at the current state. Typical implementations for neural nets is to have the input be the current state and there be an output for each action that represents the anticipated value.

Fig. 1.

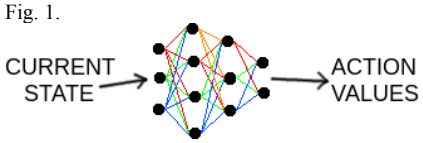

Illustration 1: Example NN with 3 input state values and 2 output actions

The way in which the training happens is that there is a memory of states, what action the agent decided to do at the time, and what reward it got (if any) by taking that action. To train it to anticipate the correct value, the agent is asked again what value the chosen action is worth at that state. That estimated value is compared to the maximum estimated value of the next state, no matter what action that would be. It is then trained to expect at the first state what the value of the second state is plus the reward it got.

To provide an example of how the training works, imagine the cartpole problem (inverted pendulum). There are 2 actions of move left or right. The score is how much time the pole stays upright. If the pole is leaning left, the agent might anticipate that moving left will be worth more future time than moving right. So when it trains, the agent guesses that moving left is worth 20 score and right is 5 score. The next state we see that it says moving left again is worth 24 score and right 9 score and it got a reward of 1 for making through that cycle. To reinforce, we train that the first state should really be worth 24+1 for moving left, and we don't train anything for the right action.

Q Learning is known to be a little unstable because of the way in which is does the learning. The issue is that if your agent is asking itself for the next estimated value and you just trained it to say something different, then you essentially are moving the goal posts as you train. The main way to get around this problem is called Double-Q Learning. What happens is that instead of the agent asking itself for the appropriate estimated value of the next state, a non-changing "teacher" agent gives the estimate. The "teacher" is just a copy of the agent that was made in the past and gets updated from time to time as the training progresses. This makes training a bit slower but the estimated values are more stable.

IV. First Attempts (Cart Pole)

To get started, the example problem of the cart pole was used as shown in the documentation for OpenAI gym.[10] This was supposed to be a simple problem that was fast and easy to train. After a few days of work, it was not working and no real clue as to why. For the most part, the papers and websites talking about how to implement it said the same things and had thought to be fully implemented in the training code. After messing around with the depth and width of the layers, adjusting different parameters such as the discount rate and memory size, it was found that a single and very important

condition was missed. The idea of being "done" was not being honored where it should train the expected future value to be 0 because no further steps will be taken and so no more score can be received.

After getting the training working to some degree with the cart pole problem, it was time to move forward with more advanced and complicated systems. What was implemented so far was a basic and straight forward Q Learning system. The code got cleaned some and a lot of debug code removed that wasn't helping.

V. Atari Start

Testing with the Atari game "Space invaders" made it clear that the current system has many faults for it's ability to learn. In general, when the hyper parameters were just in the butter zone, the ship would learn to fire while sitting still. After working around with the it, it was noticed that the optimizer was SGD instead of Adam. That change made it much wider of a butter zone for the hyper parameters and the ship sometimes would decide to head right while firing.

Continuing on with changes, the exploration factors was messed with. Initially it would start from an exploration rate and decrease it until reaching zero where training was considered done. The number of training times was experimented with, favoring longer runs. Quickly that idea platued with no major gains to be found. After that was the idea of resetting the exploration factor after reaching it's minimum. This was to have it find new ways to do things after it had refined itself. This lead to better results of the ship but only marginally; it would choose a single direction and most of the time move that way, all while firing.

After mixed results of improvement with only the random choice changes, a review of the memory was done. In theme with the simple Q Learning implemented, the memory was also simple, being an array that would push out old states as new ones were pushed in. After finding that increasing the memory didn't help past a point, an attempt at a different style was made. A memory of the top replays since the last reset was implement, where it would only keep track of the top number of runs through the game according to the reward received. Video This did move around but was uncertain about if it was any better than a random agent.

Next was a series of blunders that took a week to work through. It was things from missed hard-coded numbers to incorrect code causing memory leaks. It made many results that were run just invalid and many times the said results were very slow to get. It culminated in essentially rewriting the code base and in my change log was ended with a curt "too many small things to list" note.

VI. Dialing In The Structure

After getting everything put together correctly, the memory that was used for training was still what was being looked at. The best effort memory, while it seemed to somewhat work, was not the solution. Next was combining both the normal and the best effort queues, that way there is more temporal memory and also a focus on the best moves it had made. As can be expected at this point, it was lack-luster and so I moved on to other things.

At this point, I was noticing while staring at the output of the training network that it was bouncing around a lot. It was bad enough that no graph was needed to see it. This arguably shows progress of it training well enough to do better than a random agent. To combat this instability, the Double-Q idea was implemented. This took a couple days to figure out how to structure things to do the copy in a way that is easily compatible with my style of programming. Upon getting the system able to make a copy of the learning neural net and doing so upon every reset of the random selection rate, it improved enough to have some sort of strategy. Video

Last round of working on the memory system was to change to the priority queue. The idea is that it is better to train on events that "surprise" the agent more. A "surprise" example would be that it estimates that the event will be worth 100 but instead is worth 10 or 500; the difference in what it expects now versus the value it expects from the state after. Implementing the new system made a huge improvement in how well it works. While the code per episode did take longer, it took fewer episodes of training to get better results, resulting in less time overall. Video

Just to check the sanity of the project, the code was retooled to do the cart pole problem again but using the same parameters as the space invaders game. Video. Without even tweaking the parameters, it is able to do a fantastic job.

VII. Last Bumps To Working Well-ish

After doing some bug hunting and fixing some minor issues, the next stage was adding in batch norm. Previously batch norm was not used as it was not something I knew how to use at the time, but after some looking, it was really simple. It was found that it allowed for more interesting and varied behavior of the ship. Video Video

Next was trying to add dropout. This has held off until now because it was not known where it should be placed. It was decided that it should only be in effect once the training was happening, not when running to get more experience. It was also decided that the "teacher" for getting the expected values should not be compromised and should not have any dropout. As such, only the "student" network that was to be changed should have dropout applied when asked about what value it currently thinks things should have.

Due to implementation details, the dropout broke the ability to learn. The issue was that the agent was asked once to get it's current thoughts on the current state which had dropout, and then later for the optimize step had the dropout happen again. Since it had dropout at two different times, they were different in state, and so was not consistent enough to learn with. It had to be changed for the loss function to handle part of the calculation so that the same dropout occurred for asking what it currently thought and for back-propogation.

A side note of something interesting tried. The batch-norm layer was tried in different locations in the neurons, trying to see what results could be had from different places that it gets

inserted. None of them worked well, if at all, except for the normal style of between neurons.

The last thing to add was a "wide" layer.[2] This layer is supposed to be shallow to allow simple concepts to easily make it up without taking valuable space on the deep side of the network. It didn't hurt so it has been left in.

VIII. Graphs

So far it has been probably slightly boring with a giant wall of text. The problem is that the problems that were did not need graphs to see for the most part, and trying to optimize, well, I kindof forgot. But here are some graphs from the later nets, where I will discuss what I think is going on but not sure what it means.

Fig. 2.

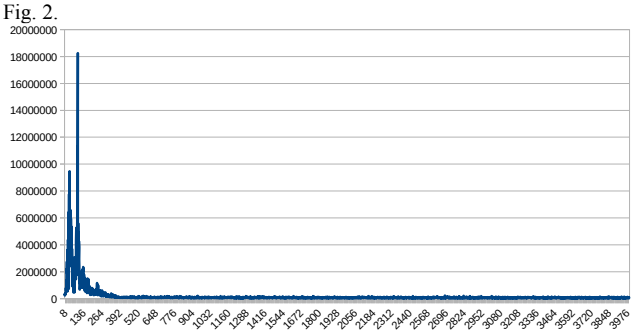

Illustration 2: Raw Loss Values From Training (batch size 100) for Every Episode

As you can see, the graph is really hard to use because of the gargantuan spikes at the beginning. After episode 400, it is fairly calm, which is the time that it starts using the double-q system.

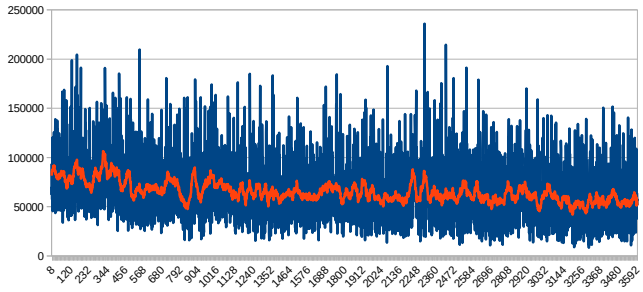

Illustration 3: Loss Values on range 400 - Last

This graph is cutting off the first 400 episodes so that we can see the more stable section. The blue line is the raw values while the red is the average over 25 episodes. At this point, besides being interesting to me about how crazy the spikes are, there was nothing I could draw conclusions about. There are other graphs, but they all equally are as crazy looking, such as the average error graph where the spike are a little smaller but mostly as crazy looking.

Fig. 3.

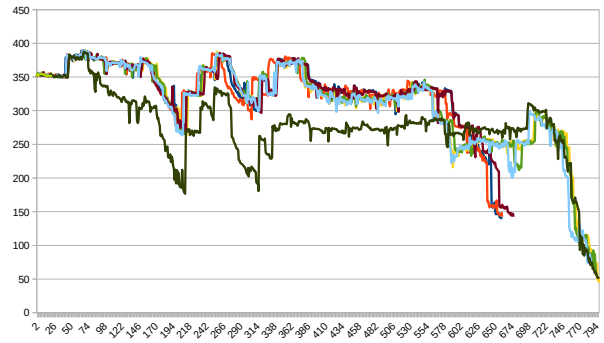

Illustration 4: Estimated Values Per Action

The only interesting left to look at now is the estimated value for each action taken during a run of the game of space invaders. The values estimated spike up right after getting hit, and seem to be flat when using the last life of the ship. Not sure what any of it means, but it has colors.

References

[1] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning," publication.

[2] "TensorFlow Wide & Deep Learning Tutorial." [Online]. Available: https://www.tensorflow.org/tutorials/wide_and_deep.

[3] T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, and T. Schaul, "Deep Q-learning from Demonstrations," rep.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," rep.

[5] A. A. Rusu, S. G. Colmenarejo, and C. Gulcehre, "POLICY DISTILLATION," rep.

[6] D. Silver, "Deep Reinforcement Learning." [Online]. Available: https://deepmind.com/blog/deep-reinforcement-learning/.

[7] "Deep Q-Learning with Keras and Gym." [Online]. Available: https://keon.io/deep-q-learning/.

[8] O. Anschel, N. Baram, and N. Shimkin, "Averaged-DQN: Variance Reduction and Stabilization for Deep Reinforcement Learning," rep.

[9] Y. Li, "D EEP R EINFORCEMENT L EARNING : A N O VERVIEW," rep.

[10] "OpenAI Gym Documentation." [Online]. Available: https://gym.openai.com/docs/.

[11] N. Heess, J. Merel, and Z. Wang, "Producing flexible behaviours in simulated environments." [Online]. Available: https://deepmind.com/blog/producing-flexible-behaviours-simulated-environments/.

[12] "Let's make a DQN: Double Learning and Prioritized Experience Replay." [Online]. Available: https://jaromiru.com/2016/11/07/lets-make-a-dqn-double-learning-and-prioritized-experience-replay/.