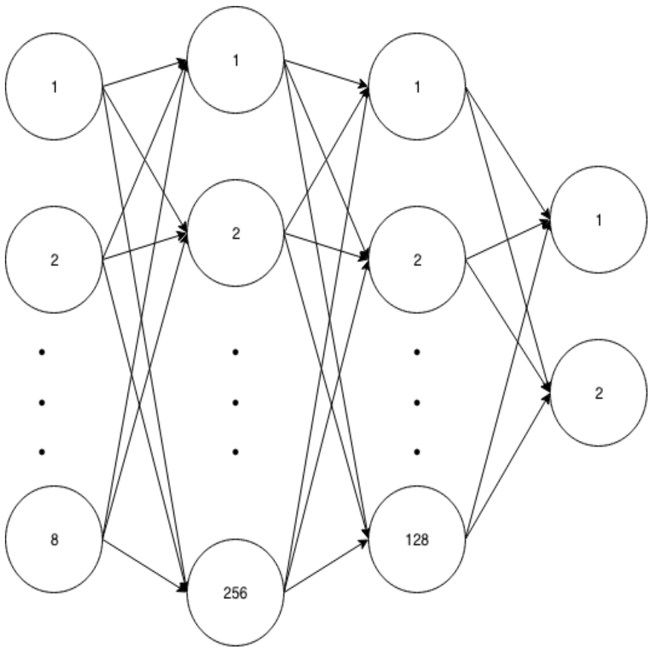
Mathias Babin
10/21/2018

P3 – Continuous Control Report

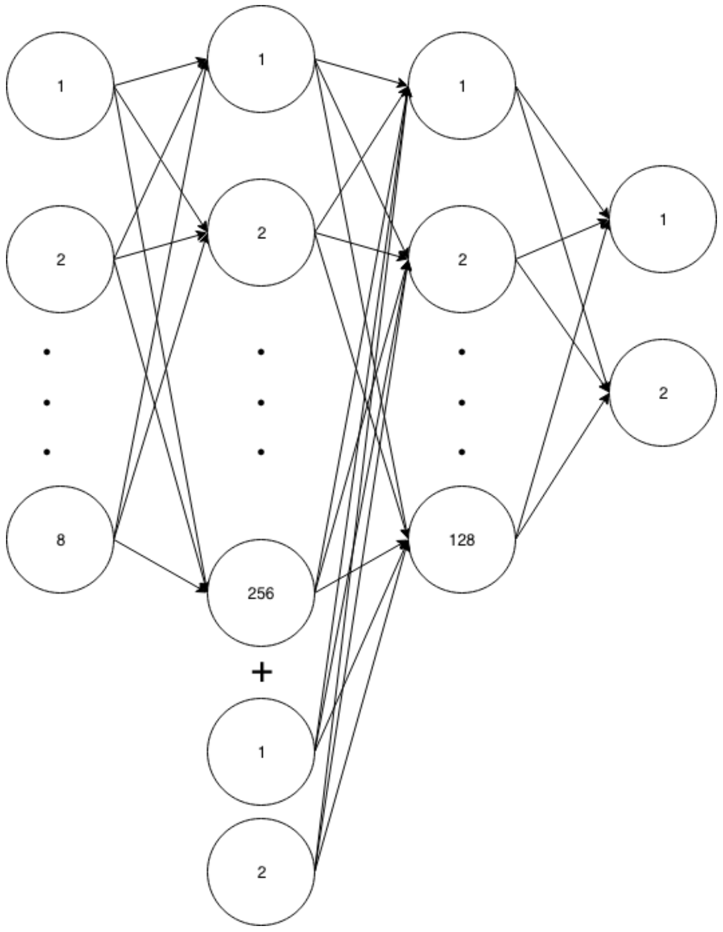
1) Learning Algorithm

For this project I implemented the DDPG algorithm, with improvements made via Parameter Noise. The Neural Network(NN) architecture for this project includes 8 input neurons to reflect the environment's state space, 256 and 128 hidden units, and 2 outputs for the actor networks and 1 output for the critic networks. It should be noted that the second hidden layer for the critic networks include 2 additional units to account for the two outputs produced by the Actor networks. The learning rate for all NNs was 0.00015.

Actor Networks:



Critic Networks:



The size of this replay buffer was 100 000, and batches contain 64 experiences. Finally, as suggested in an OpenAI blog post(https://blog.openai.com/better-exploration-with-parameter-noise/), I added a parameter noise to all four NNs via a parameter noise layer.

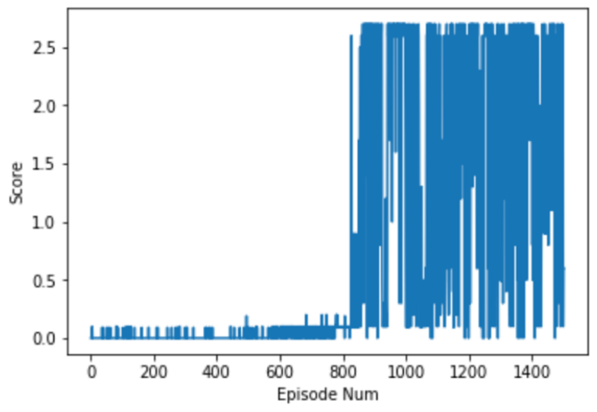
It should also be noted that a soft update rules was used to update the weights of the NNs where Tau was set to 0.01, where updates occurred every timestep.

What made this project unique is the fact that two agents were being trained at the same time. Both agents share not only the same actor and critic networks, but also the same replay buffer. Initially, I ran into a problem where that the agents very quickly learned to stick the net and refused to try new actions that would yield higher rewards. This led to some unsuccessful experiments where I would add noise to their action values. I believe the problem with this approach was that the actor NNs did not know how to train when the same state input led to two very different resulting rewards because the noise forced the agent into taking very different actions. I eventually resolved this learning problem by having one agent initially only take random actions while still adding his experience to the replay buffer for the other agent to learn from. This can be viewed as having one agent only explore, while the other is more focussed on exploitation. Eventually the exploiting agent learns to play fairly well, and at this point we can tell its partner to start using the same policy as it. This resulted in two fairly well-trained agents able to now volley back and forth and very quickly improve the policy.

2) Results

As shown from the average scores listed below, the scores go from very little to no improvements, to suddenly spiking to a phenomenal level within 100 episodes. This is because at an average score of 0.05, the second agent is told to start using the policy being trained by its partner. This means that only one agent is responsible for getting the average score up to 0.05 by itself, but by doing so it proves that it is fairly competent at the task. Once both agents start using this policy, they very rapidly start working to together to raise the average score well passed what is required to consider the environment solved (an average of +0.50 over 100 episodes).

```
Episode: 100    Average Score: 0.01     Score: 0.00
Episode: 200    Average Score: 0.01     Score: 0.00
Episode: 300    Average Score: 0.01     Score: 0.09
Episode: 400    Average Score: 0.01     Score: 0.00
Episode: 500    Average Score: 0.01     Score: 0.00
Episode: 600    Average Score: 0.01     Score: 0.00
Episode: 700    Average Score: 0.04     Score: 0.10
Episode: 800    Average Score: 0.07     Score: 0.10
Episode: 900    Average Score: 1.12     Score: 0.10
Episode: 1000   Average Score: 2.00     Score: 0.70
Episode: 1100   Average Score: 1.40     Score: 2.60
Episode: 1200   Average Score: 1.70     Score: 2.60
Episode: 1300   Average Score: 1.86     Score: 1.90
Episode: 1400   Average Score: 2.08     Score: 0.80
Episode: 1500   Average Score: 1.78     Score: 0.60
```



3) Future Improvements

Normally when I work with the DQN/DDPG algorithm I like to at least implement Priority Experience Replay (PER), but in this instance I ran into a stubborn error with pytorch that would not accept the dimensions of the tensor I was supplying it. I tried a work around where I used the reward from a particular state as the p value, and while this did help the agent learn initially, it eventually lead to the agents only learning from a small subset of sampled positive experiences. This is a problem because the agents become satisfied repeating sub optimal actions and not learning from any mistakes they've made. So without PER, I was forced to use a simple replay buffer that supports just random sampling, which leads to a larger variance in the number of episodes it takes for the agents to learn, and would ideally be replaced by PER. Also my current implementation has the agents sharing both the actor and critic networks. Perhaps better results could be obtained if they maintained their own separate critic networks as this would impact the training of the shared actor network.