Deep Reinforcement Learning: Application of Deep Q-Networks and Deterministic Deep Policy Gradient to the Bipedal Walker Problem

Thomas Tarler
ttarler@gmail.com
(720) 496-9222
Regis University

Reinforcement Learning is a type of machine learning which focuses on training an arbitrary agent to complete a complex task, such a a robot picking up a box. Deep reinforcement learning applies neural networks and deep learning to the reinforcement learning process. In this practicum, the author attempts to complete the Bipedal-Walker tasks using two algorithms: Deep-Q-Networks and Deep Deterministic Policy Gradients. The paper is practically split into two portions: 1) Theory and 2) Application. In Theory, we examine the mathematics behind Q-Learning and Reinforcement Learning, before diving into the specific mathematics of the algorithms we are employing. We then use a deep-learning cluster to train several agents to explore the behavior and performance of these models in practice.

Due to the complexity of the models, the author's agents were never able to successfully complete a run. This may be to be expected as some of the algorithms are using up to four different neural networks. However we do yield interesting diagnostics data that is utilized to determine which model may be more successful in the long run. We finalize this paper with a simple explanation of potential business applications.

Contents

List of Figures

Currently, many applications of data science rely on supervised and unsupervised learning models. These are tasks associated with labeling or predicting a value for a data set or attempting to ascertain an underlying structure to the data. The difficulty with models such as these is applying the results. For example, a model might suggest that a transaction has an 82% probability of being fraudulent but it is up to the bank to decide what to do with that information. Reinforcement learning, on the other hand, focuses explicitly on actions or policies that an agent or actor can undertake in order to maximize some underlying profit function with respect to the environment. Mathematically, we describe this as

$$\mathcal{G} = \{S_i, A_i, \Pi(S, A) \mapsto \mathbb{R}\}$$

where $\mathcal{G}$ is the game, S_i is the state of agents and the environment, A_i is the action set of the agent, and Π is the profit function. The goal of reinforcement learning is to develop a set of rules, policies, or action-decisions that maximize the profit for a given agent. When principles of deep learning (i.e. neural networks) are applied to this framework, we refer to it as deep reinforcement learning.

The author intends to use existing reinforcement learning frameworks and explore optimizing them against a game. The primary intention of this project is to compare the efficacy of deep reinforcement learning compared to techniques that do not utilize reinforcement learning. The intention is to explore and understand when deep-learning is more applicable to the problem, which has a faster training time, which performs better after a set number of training cycles, and other evaluation items.

For this project, we will solve the OpenAI Bipedal Walker environment. This is a simulated physics environment where the goal is for the robot (the eponymous bipedal walker) to walk to the end of the environment. The game is initiated with no existing strategy or policy for the walker to complete the game, and it is deep reinforcement learning that yields the ability to complete the course. We chose this environment due to its intrinsic complexity and ability to compare multiple approaches. First, we will then examine the mathematics of Deep Reinforcement learning, with an overview of Augmented Random Search, Deep Q-Networks and Deterministic Deep Policy gradient. Finally we will demonstrates completed successful runs using various different models. We will then offer a detailed explanation of the Bipedal Walker game and associated components. We will conclude this project with a quick overview of our training results.

Due to the complex computational needs of this exercise, all major iterations have been completed on the Regis University Deep Computing cluster, which does not support the .ipynb format. Wherever possible we will include the scripts that we ran at the end of this document or otherwise linked but will not display code and results.

Theory

As opposed to Supervised Learning and Unsupervised learning, reinforcement learning does not deal with the traditional mathematical constructs of a data set with either some label, value, or underlying structure associated with it. Nor do we evaluate models produced in the traditional manner; there is no binary classification or topological metric that we can use to satisfy business requirements. Rather, reinforcement learning is concerned with some task that an agent may or may not complete using one or more actions. The agent is graded using an award function, and is typically competing against other agents or the environment. As such the central mathematical construct for reinforcement learning is game theory. For this theory section of the paper, the author will walk through what is reinforcement learning and then give details around the various algorithms used to solve a reinforcement learning task.

Reinforcement Learning

There are two entities involved within reinforcement learning: the agent or actor, and the environment The agent, $\mathcal{A}$ has a state and a set of actions associated with it. The is opposed to the environment which only has its state and future encoded states associated with it Beyond there agent and environment framework: there are four central components, the policy, the reward signal, the value function, and the model of the environment. It is important to step through all of these components mathematically before delving into the specific algorithms and problems we are trying to solve.

Within each agent, we have an associated set of states, s at each time step. That is, $S = \{s_1, s_2, \ldots, s_{t-1}, s_t\}$ represents all possible states associated with an agent. We have implicitly defined with time t represents a set of steps, however states are not necessarily linear in time. A policy, p then maps an action, a_i and a state associated with an actor to a new state. That is, $p : s_i \times a_i \mapsto s_{j \neq i}$. The goal of reinforcement learning is to find the optimal policy, p_i for a given actor and state.

The reward signal is vital for determining how good a policy and/or state is. We define the reward signal, r to be the function that takes an agents state and maps it to a number. That is, $r : A_i \times S(A_i)_t \mapsto \mathbb{R}$ where $S(A_i)_t$ signifies the state associated with a given actor at time t.

The valuation function defines very specifically how a given reward is valued over time, or until the end of the game. This enables the agent to consider their total possible award for the rest of the game and ensures that long-term strategic thinking is employed. At its basic, we define it as

$$V(A_i) = \sum_{t_a}^{t} R(A_i, S(A_i)_t$$

but in practice often includes a discount function to factor in present and future value.

An agent will seek to maximize this value function for the remainder of the game. If the value function is relatively low, then agent may take a *riskier* set of policies to maximize their success or failure.

Finally, we need the environment or the model of an environment. The environmental model represents the state of the environment for a given time t and enables future planning for possible actions in an environment and enables an agent to calculate our their future earnings and actions given potential changes to the state of the environment. The environment can also encode the actions of other, either adversarial or cooperative agents in order to facilitate the actions of others upon an agent. This is where the OpenAI gym comes in handy, in that it simulates an arbitrary environment - in this case a two-dimensional physics world.

Deep Q-Networks

Deep Q-Networks are a variation on Q-Learning and so we will explain Q-Learning first. Q-learning is explicitly *model-free* meaning that no agent encodes a model of the environment they are working on, Instead each state is considered to be a step in a Markov Chain, so that the state can always be calculated using the last observation. Q-learning attempts to determine an optimal policy Q for a given agent and action-state combination. Note that each action-state must be discrete and enumerated, meaning we can associated indices $i, j \in \mathbb{N}$ with every action state, and the size of Q-Learning is $i \cdot j$. We first need to define the valuation function as

$$Q'(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \Big(r_t + \gamma \cdot \max_{\alpha} Q(s_{t+1}, a_t) - Q(s_t, a_t) \Big) \tag{1}$$

where Q' is the updated policy, $\gamma \in \mathbb{R} \cap [0, 1]$ is the *discount* rate (or how much the model discount future values to the prevent), $\alpha \in \mathbb{R} \cap [0, 1]$ is the *learning rate* (how much the model learns from past actions) and r_t is the present reward at time t. Necessarily, setting γ low means the model does not discount future actions as much and disregards actions that may improve its current state for a longer-term reward. Conversely, setting γ high means future actions are heavily discounted and the model becomes greedy. When we set α to a low value, the model with not learn as much from its future actions and will bias itself towards using past, successful actions. When α is high, the model will learn more. A final variable that is vital is $\epsilon \in \mathbb{R} \cap [0, 1]$ or the experimentation rate. If ϵ is high, the model will chose to undertake a completely random action more frequently rather than some policy it has already learned to be successful.

Equation 1 is the cost function traditionally used in both Q-Learning and Deep-Variants. It is known as the Bellman equation.

Traditionally, Q-Networks have been trained using somewhat of a brute-force approach, where nearly every action and state combination. This means when states or actions gets very large, the efficacy of Q-Learning diminishes. For the agent to be successful, every state must have been experienced and every possible sequence

of actions must also have been undertaken. Deep Q-Networks relax this assumption by allowing very large action-state combination spaces, though they must be discrete in nature. That is, there are still a finite (but very large) number of action-state combinations. This means that the Q-Learner is replaced with a neural network of some sort, $Q : S \mapsto A$ which takes in the state of the environment and outputs a suitable action. Since neural networks can be used to approximate nearly an function f this means that the action of training a neural network can approach a more efficient learner than the brute force approach. Other than using a neural network to determine the best policy, Deep Q-Networks are otherwise homeomorphic to traditional Q-Learning.

Algorithm 1 DQN Algorithm (Mnih et al., 2013)

Initialize Replay memory $\mathcal{D}$ to capacity $\mathcal{N}$
Initialize Action-Value function Q with random weights
for episode = 1,M **do**
    Initialize Sequence $s_i = \{x_i\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    for $t = 1, T$ **do**
        with probability ϵ select a random action a_t
        otherwise select $a_t = \max_a Q^\star(\phi(s_t), a; \theta)$
        Execute action a_t 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 transaction $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random mini batch of transactions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set

$$y_i = \left\{ \begin{array}{ll} r_j & \text{for terminal } \phi_{J+1} \\ r_j + \gamma \max_{a'} Q^\star(\phi_{j+1}, a'; \theta') & \text{for non-terminal} \phi_{J+1} \end{array} \right\}$$

        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
    end for
end for

There are two key distinctions between Q-Learning and a Deep-Q-Network: The experience replay buffer and the actor-network. The Experience replay buffer is necessary because sequential experiences tend to be correlated. That is s_t s_{t+1} and so on. We want these experiences to be independently distributed so that the Actor-network may gain a better understanding of what causes each s_t. Therefore we randomly sample from the replay buffer $\mathcal{D}$ and use a preimage process to de-correlate of $\phi_i = \phi(s_i)$.

The actor-network is a typical neural network, what is critical is how we perform gradient descent in order to facilitate backward propagation. We then take equation

1 and define a loss function

$$L(\theta_i) = \mathbb{E}\left[(y_i - Q(s, a; \theta_i))^2\right] \tag{2}$$

where

$$y_i = \left\{ \begin{array}{ll} r_j & \text{for terminal } \phi_{J+1} \\ r_j + \gamma \max_{a'} Q^{\star}(\phi_{j+1}, a'; \theta') & \text{for non-terminal} \phi_{J+1} \end{array} \right\} \tag{3}$$

We can then take the derivative of 3 and yield

$$\nabla_{\theta_i} = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s, a; \theta_i))\nabla_{\theta_i} Q(s, a; \theta_i\right] \tag{4}$$

Deep Deterministic Policy Networks

The Deep Deterministic Policy Networks (DDPG) are a further refinement to Deep Q-Networks however they relax the assumption that states and actions must be discrete and instead account for *continuous* states and actions. This means that while the input and output dimensions are still categories, the value encoded within them reverts from some boolean value (e.g. Is there an actor in space (n, m)) to continues (e.g. what is the probability that there is an actor in space (n, m).

Algorithm 2 DDPG Algorithm (Lillicrap et al., 2015)

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state s_1
    for t = 1, T **do**
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_1$ according to the current policy and exploration noise
        Execute action a_t and observe reward t_t and observe new state s_{t+1}
        Sample a random mini batch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
        Set $y_I = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the loss $L = \frac{1}{n}\sigma_i(y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta\mu} \mu(s|\theta^\mu)|s_i$$

        Update the target networks:

$$\theta^{Q'} \leftarrow \gamma\theta^Q + (1-\gamma)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \gamma\theta^\mu + (1-\gamma)\theta^{\mu'}$$

    end for
end for

A a very high level, the DDPG take advantage of four different and diverse networks:

1. θ^Q is the *Actor-Critic* network, where an actor or agents actions are criticized by a critic for good output, similar to GAN
2. θ^μ is the deterministic policy network, a neural network that determines best possible action given a time, and serves as long-term memory
3. $\theta^{Q'}$ Target Q-Network, a Time-delayed version θ^Q
4. $\theta^{\mu'}$ Target-policy network, a time delayed version of the policy network

The actor-critic network is similar to a GAN architecture. Since there is no *label* to be applied with this type of learning, the Critic network is meant to tell the actor how good their policy/action is prior to a reward function being calculated. There

are four components to explore within this: Replay Buffer, Actor and Critic Updates, target network updates, and Exploration.

The Replay Buffer is almost identical to the DQN network above, and we will not explore. Similar to above, there is a tuple of (s_i, a_i, r_i, s_{i+1}) that are randomly sampled to be used and create linearly independent states.

Similar to Deep Q-Networks, the Actor network (which focuses on policy) is updated using a discounted future-state value of rewards, though with target value (critic) and target policy networks. Equations 4 and 2 cover these. However the Critic Network is the first significant difference from DQN. The crittic network contains the policy function

$$J(\theta) = \mathbb{E}\left[Q(s,a)|_{s=s_t a=\mu(s_t}\right] \tag{5}$$

Since this is a well-defined function, all we have to do for the loss is to take the dertivate of the Policy function 5 with respect to the policy parameter. That is:

$$\nabla_{\theta\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)} \nabla_{\theta\mu}\mu(s|\theta^\mu)|s_i \tag{6}$$

Target Network updates are then performed where we take a copy of θ^Q and θ^μ and update them with a parameter τ. This allows for a slower *learning* of these networks that takes place on a greater scale than epochs and episodes.

The final important aspect is exploration. In discrete reinforcement learning problems, we can chose a random action a_i from the action space whenever $\epsilon_i < \epsilon$. This is important to learn new actions. However in a continuous action space, we instead want to inject random noise into an action. For example, moving a joint just slightly harder. DDPG networks use the *Urnstein-Uhlenbeck Process* for the noise to ensure that it does not cancel an action. This process is

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N}$$

Bipedal Walker

The Bipedal Walker is a continuous game made available by OpenAI. Like any game, there are two primary components, the environment and the agent, with the two coupled together in a profit or reward function. That is, an **agent** takes an action. This action has a tangible effect on the **environment** and the state of the agent. If the action results in a favorable movement in the environment, then the agent is given a positive award.

We will first examine the agent, or the bipedal walker. The Bipedal Walker or $\mathbb{A}$ has two distinct attributes: the action space $\mathcal{A}$ and the observation space $\mathcal{O}$. In general, our neural network is a function, D such that $D : \mathcal{O} \mapsto \mathcal{A}$.

The observation space for the robot is rather large, that is $|\mathcal{O}| = 24$. That is, there are twenty-four different observations that the agent can make about itself (numbers 0-13) and about the environment using its Lidar (14-23).

Num	Observation	Min	Max	Mean
0	hull_angle	0	2π	0.5
1	hull_angularVelocity	-∞	+∞	-
2	vel_x	-1	+1	-
3	vel_y	-1	+1	-
4	hip_joint_1_angle	-∞	+∞	-
5	hip_joint_1_speed	-∞	+∞	-
6	knee_joint_1_angle	-∞	+∞	-
7	knee_joint_1_speed	-∞	+∞	-
8	leg_1_ground_contact_flag	0	1	-
9	hip_joint_2_angle	-∞	+∞	-
10	hip_joint_2_speed	-∞	+∞	-
11	knee_joint_2_angle	-∞	+∞	-
12	knee_joint_2_speed	-∞	+∞	-
13	leg_2_ground_contact_flag	0	1	-
14-23	10 lidar readings	-∞	+∞	-

In the Bipedal Walker, the size of the action space is A or $|\mathcal{A}| = 4$ with each action corresponding to some joint (two knees, two hips). For each $i \in \mathbb{Z} \cap 4$, there is a torque associated with it. Hence we have

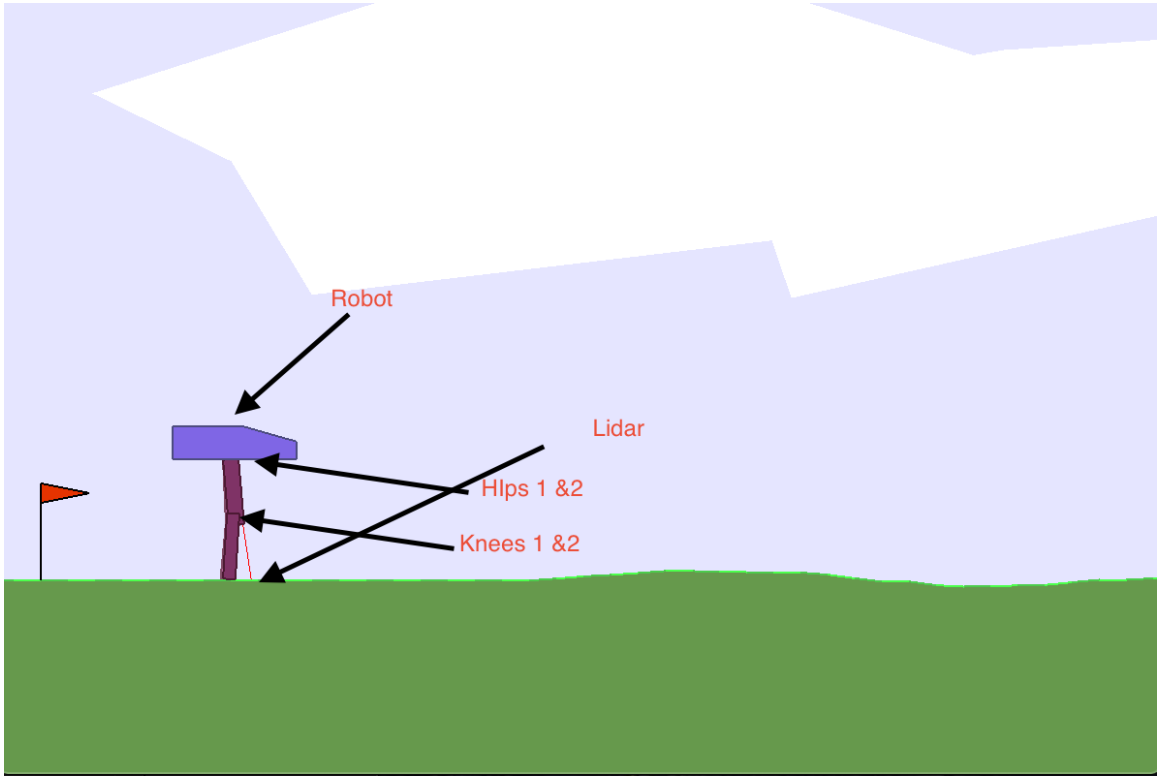
Num	Name	Min	Max
0	Hip 1	-1	+1
1	Knee 1	-1	+1
2	Hip 2	-1	+1
3	Knee 2	-1	+1

The goal of the bipedal walker is to train the robot to complete the course. The reward function $R(t)$ is defined as the following:

$$R(t) = \left\{ \begin{array}{ll} 300 & \text{Walker completes course} \\ -100 & \text{Walker falls down} \\ -x & \text{Seconds where torque is applie to the motor} \end{array} \right\}$$

This creates an easy to see victory condition, that is when $R > 300$. The specific structure of the cost function is such that the best performing Robot walkers will not only complete the course in a minimum of time, but will also have the smallest amount of energy used for walking. The starting position of the simulation is with the robot at the far left of the course with mostly straight legs. The failure condition is when the hull-contact sensor touches ground (the robot has fallen) or the stage has been complete. Please reference **??** for a visual diagram of the environment.

Figure 1. Bipedal Walker Environment



Results

Methodology

For both the DQN and the DDPG, training was done on a remote Regis cluster with three GPU's and job management. These clusters were built without a GUI interface. To begin with, results were run with the episodes and epochs setting set to 10000 each with model checkpoints as applicable.

Code was implemented from the [insert] repositories. Code was tweaked to run on the Regis University cluster, with parameters changed around environment rendering, save-states and V1 vs. V2 implementations. The neural networks were minority tweaked to get them operating and returning results.

DQN

The following hyperparameters were used for DQN networks.

- $\gamma = 0.98$
- $\alpha_{agent} = 0.02$
- $\epsilon_{\text{initial}} = 1.0$[1]

Additionally, we had a simple sequential model for the DQN, as shown in Figure **??**.

[1] ϵ varies throughout based on a decay rate of 0.995

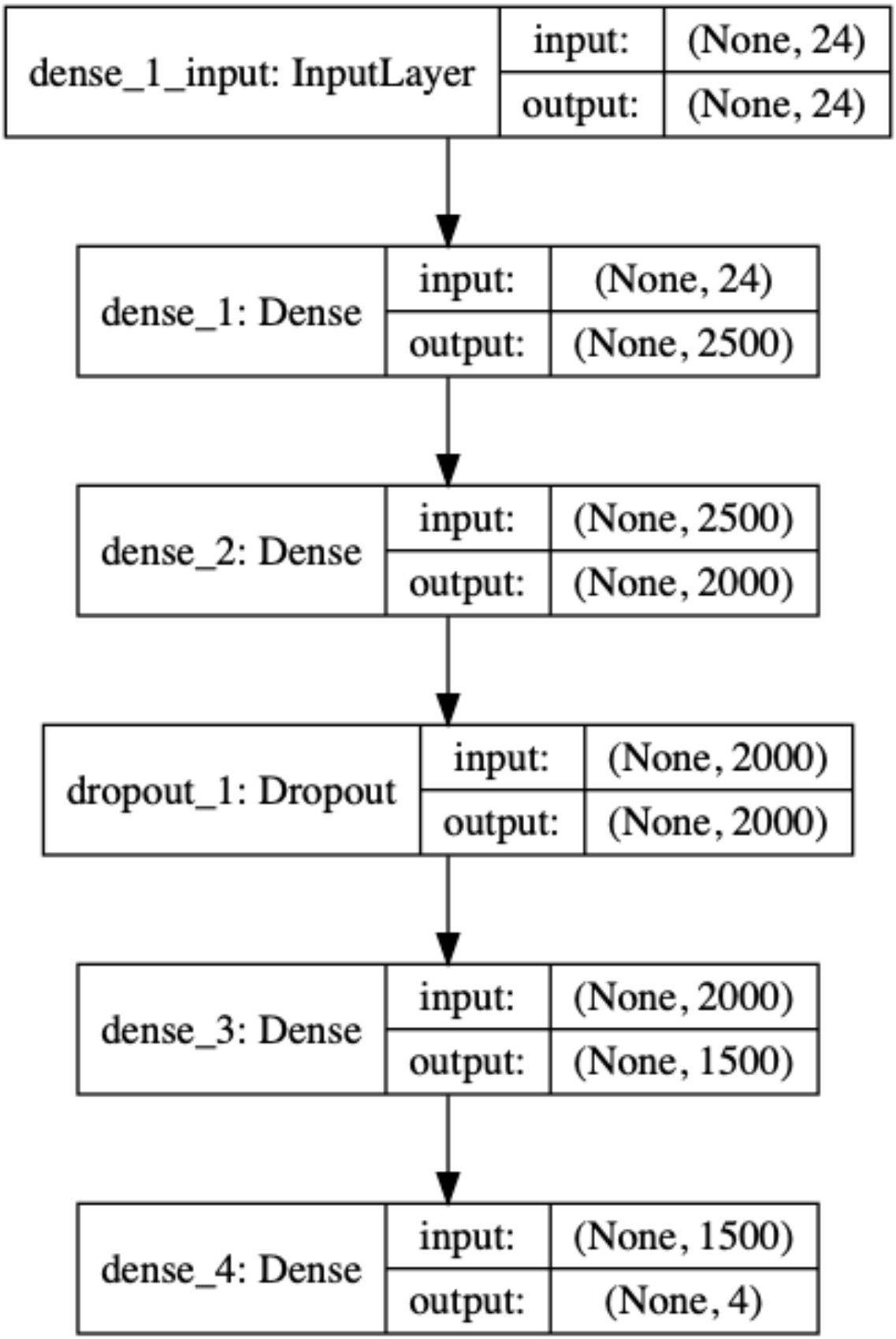


Figure 2. Neural Architecture of the DQN Model used for training

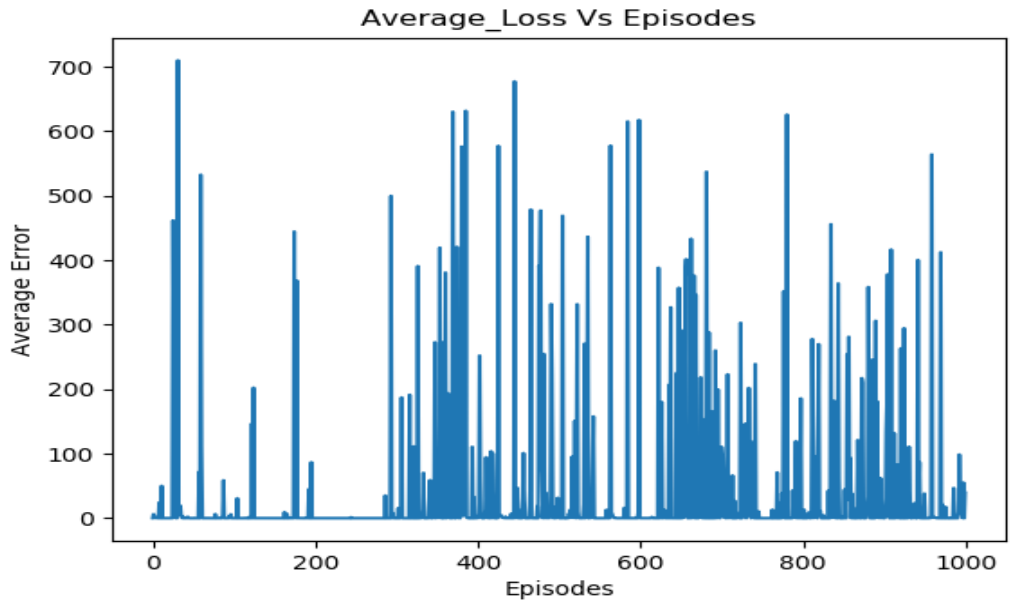


Figure 3. Loss and Error when DQN is run for 1000 Episodes

1000 Episodes.

Figure 4. Reward when DQN is run for 1000 Episodes

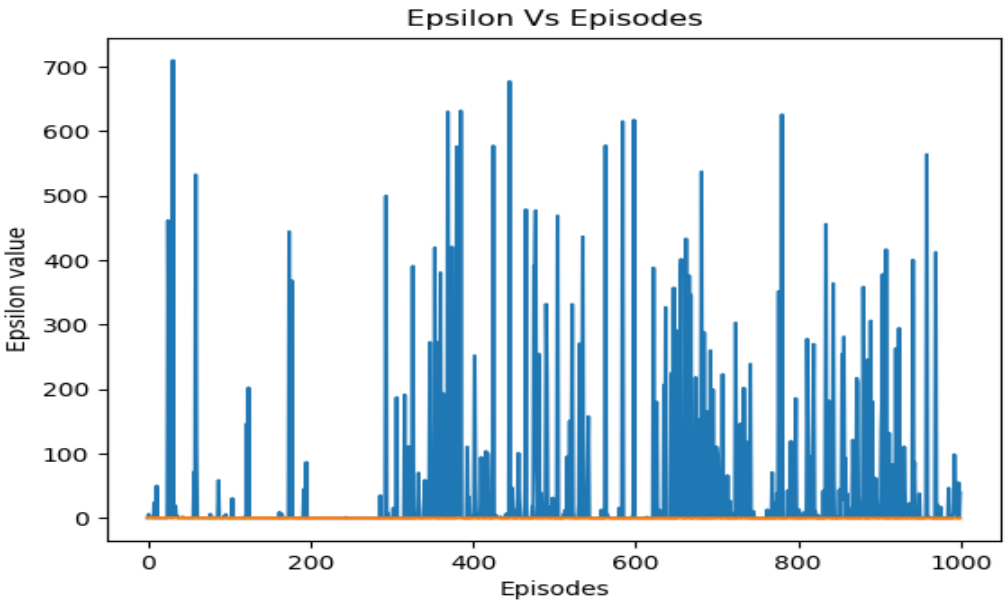


Figure 5. Epislon when DQN is run for 1000 Episodes

5000 Episodes.

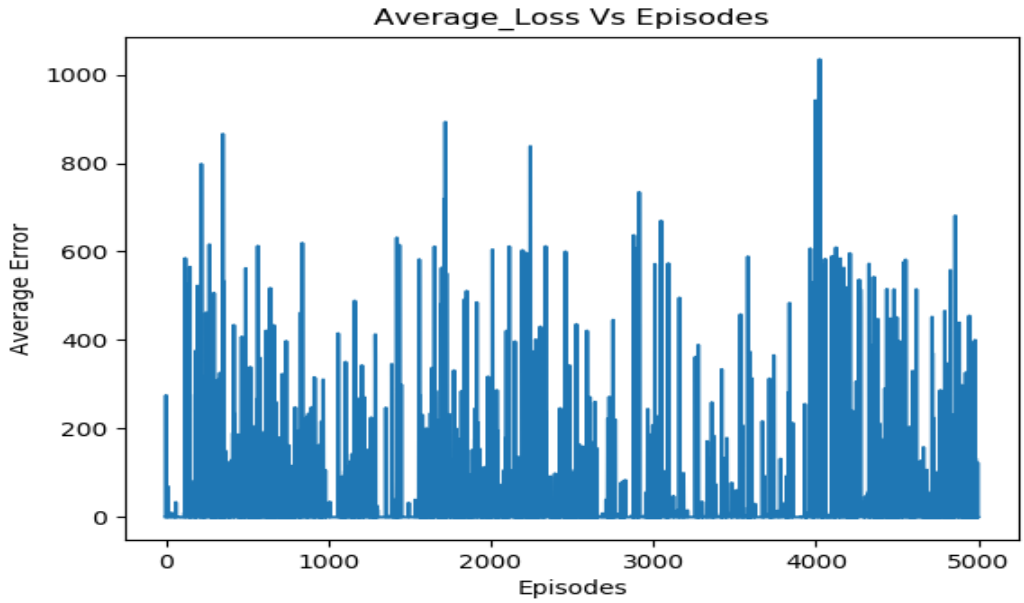


Figure 6. Loss and Error when DQN is run for 5000 Episodes

Figure 7. Reward when DQN is run for 5000 Episodes

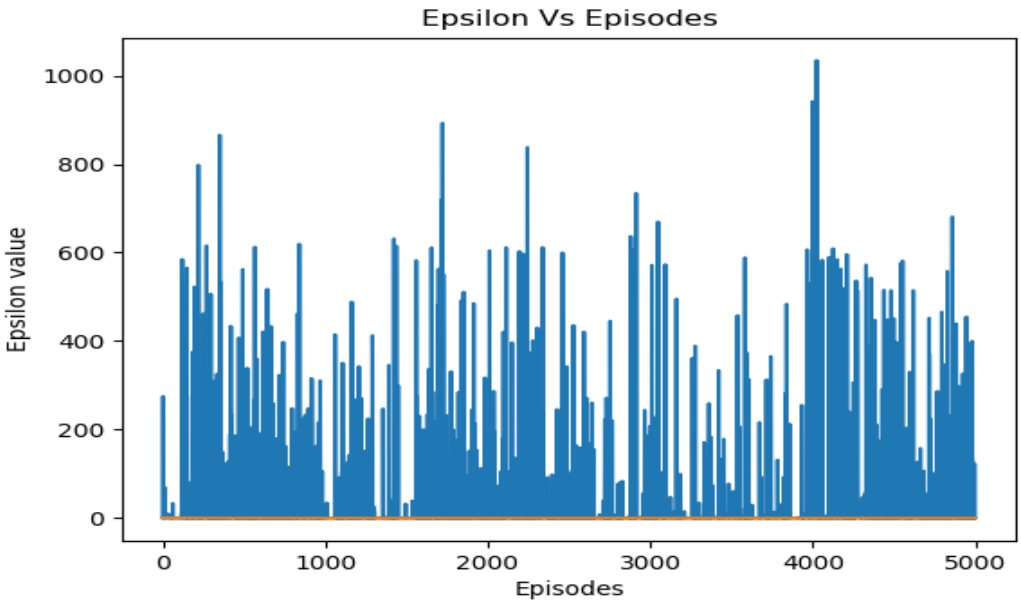


Figure 8. Epsilon when DQN is run for 5000 Episodes

DDPG

The following hyperparameters we used for the DDPG networks.

- $\gamma = 0.85$
- $\alpha_{agent} = 0.1$
- $\alpha_{critic} = 0.02$
- $\hat{\alpha} = 0.1$
- $\tau = 0.01$
- $\epsilon_{\text{initial}} = 1.0$[2]

The four networks used in the DDPG are listed below:

[2] ϵ varies throughout, similar to DQN, and is governed by the Ornstein Uhlenbeck Process

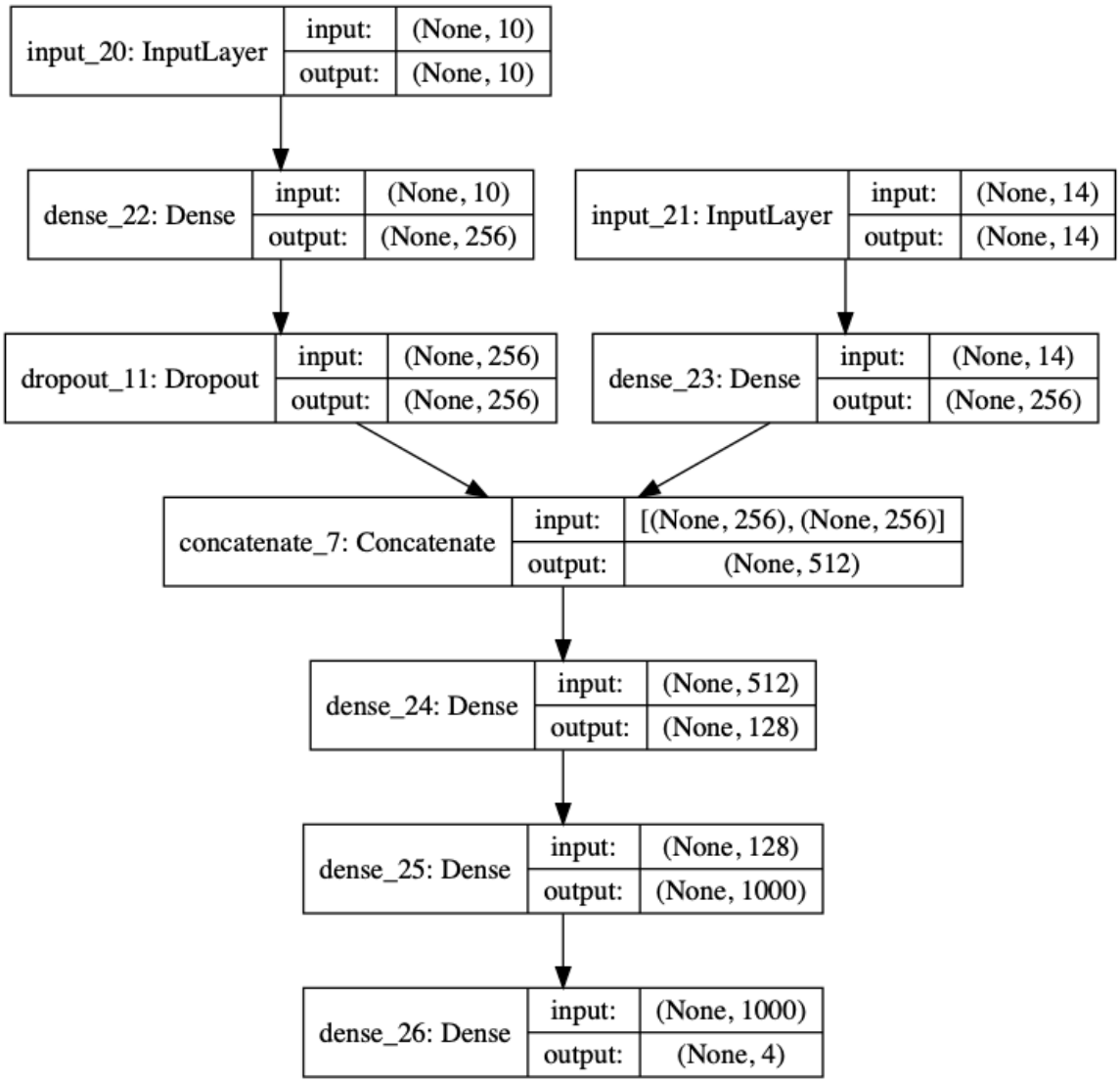


Figure 9. DDPG Actor Local Network

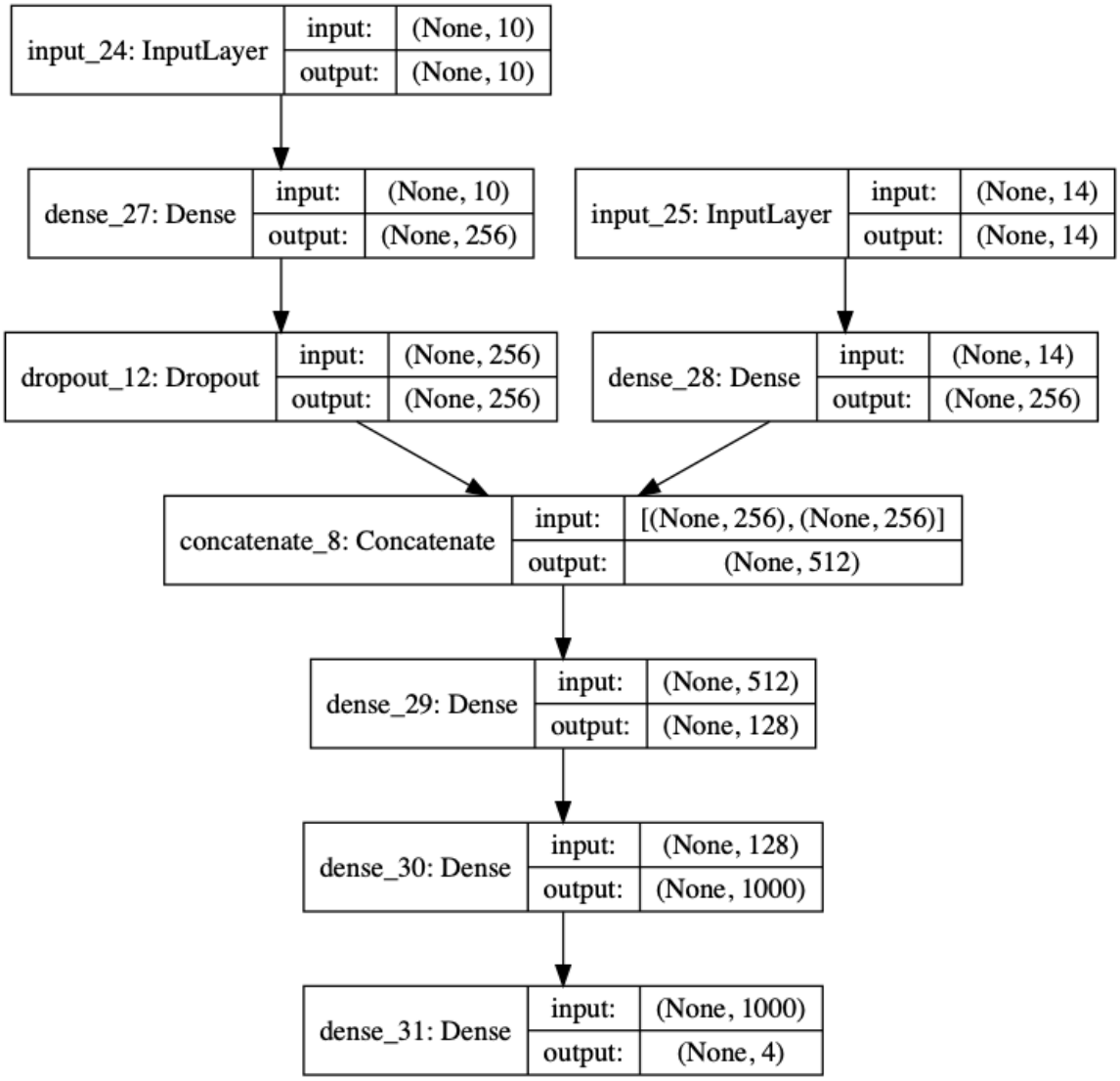


Figure 10. DDPG Actor Target Network

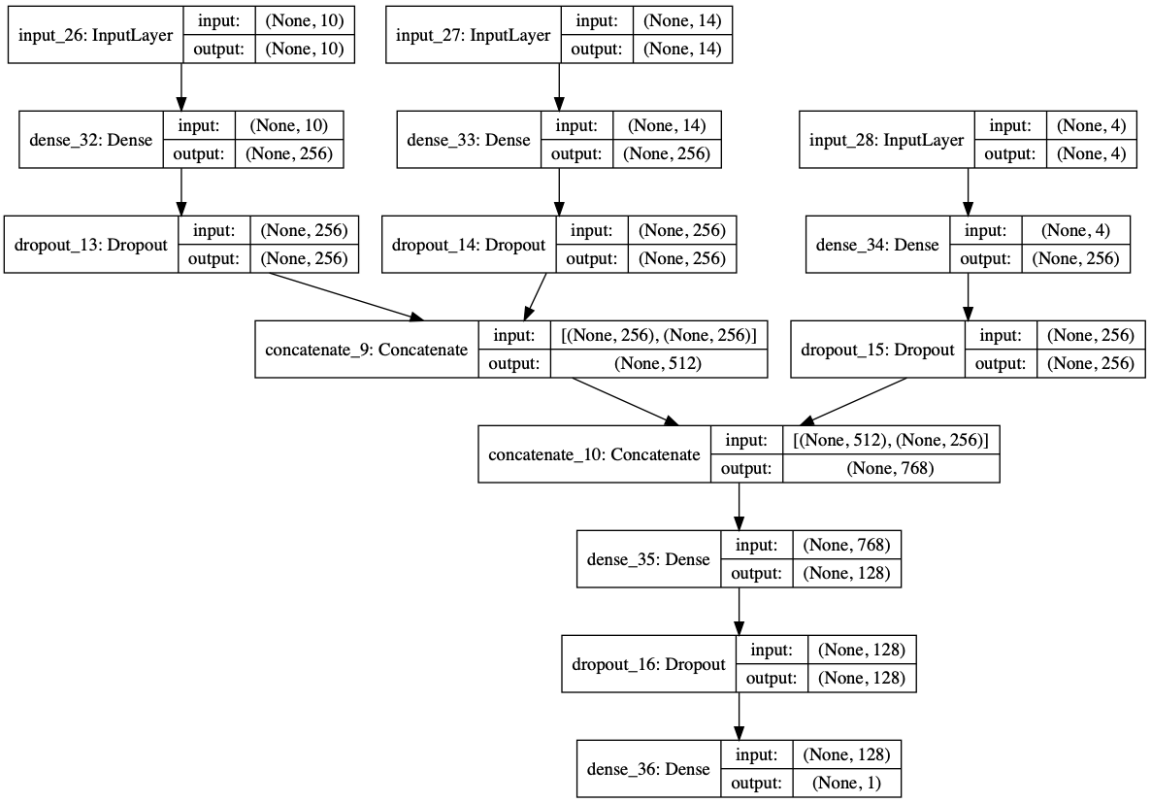


Figure 11. DDPG Critic Local Network

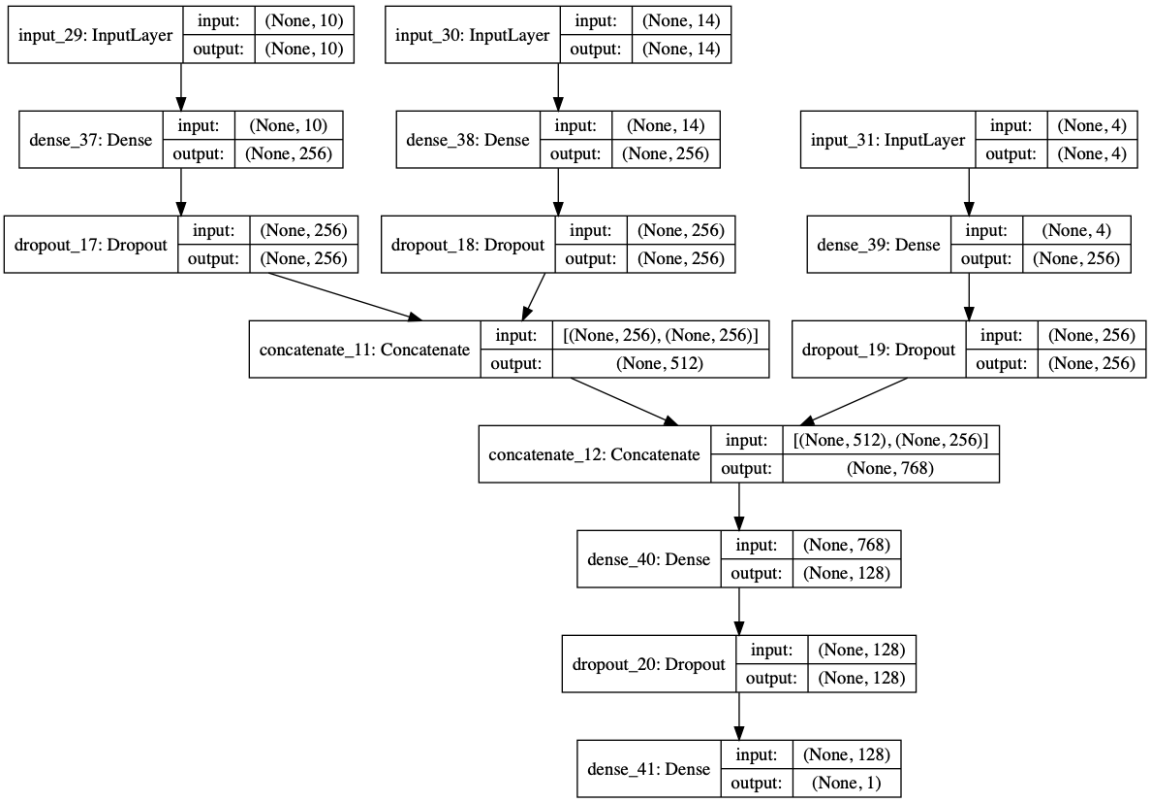


Figure 12. DDPG Critic Target Network

1000 Episodes.

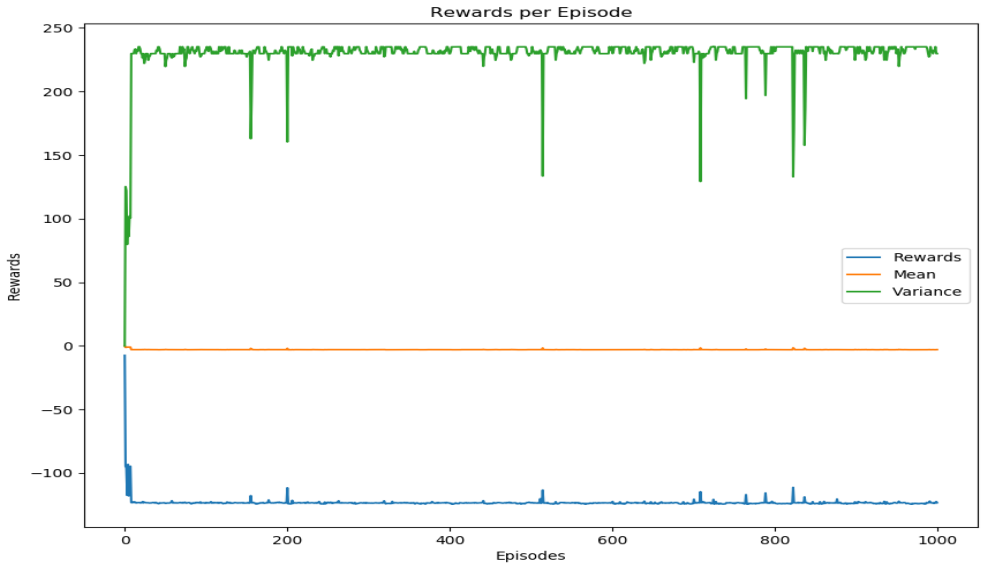


Figure 13. Reward when DDPG is run for 1000 Episodes

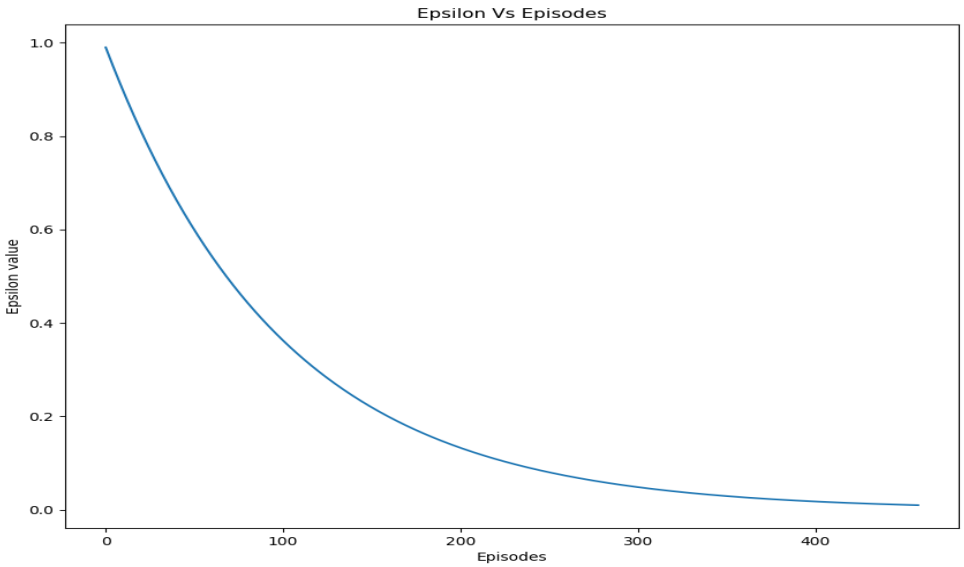


Figure 14. Epislon when DDPG is run for 1000 Episodes

5000 Episodes.

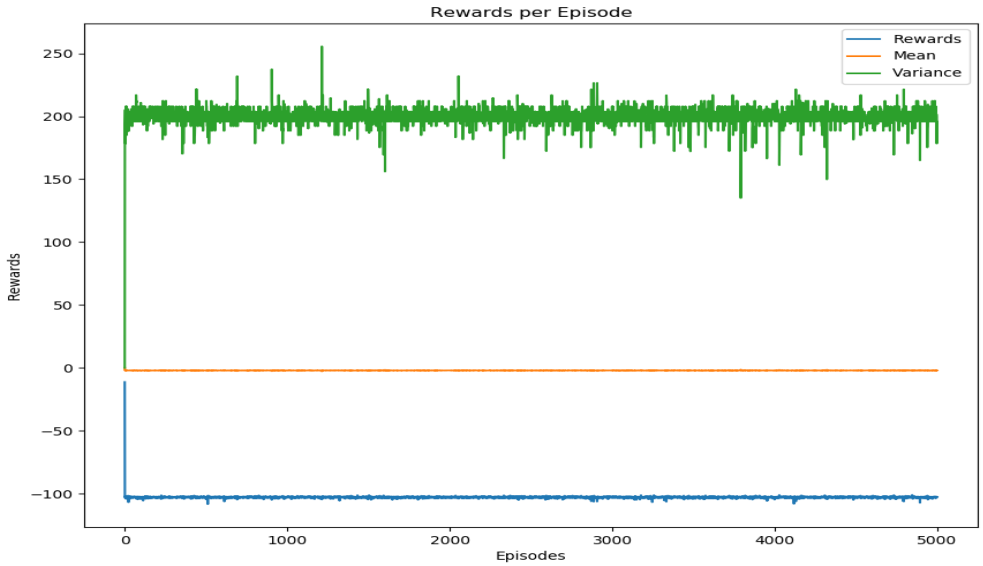


Figure 15. Reward when DDPG is run for 5000 Episodes

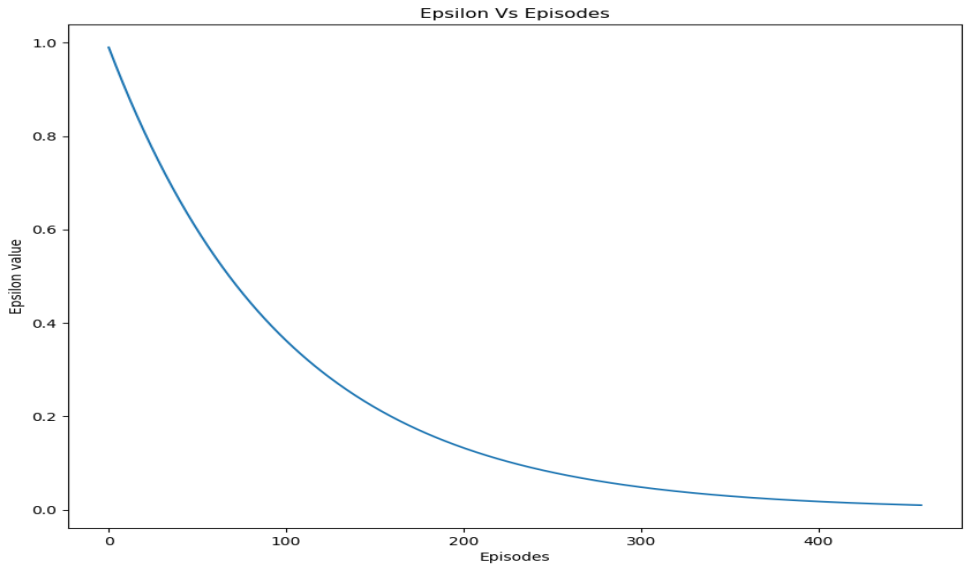


Figure 16. Epislon when DDPG is run for 5000 Episodes

Comparison of DQN vs. DDPG

The DDPG, while taking a lot more computational resources due to four different neural networks being utilized, does show steadier and more consistent results. While neither robotic walker completed the course, the DDPG walker had less *very large* negative rewards (comparing figures **??** vs **??**). This indicates that the LSTM used for action replay, as well as the critic network enabled a stabilization of the DDPG network that the DQN was otherwise unable to access. This can also be partially explained by the DQN using a more random ϵ rather than the smoothly-decaying ϵ. We can see an extremely smooth graduation of the ϵ as seen in figure **??** versus figure **??**. This implies that with sufficient computational power the DDPG would more smoothly get to a trained walker, as opposed to the DQN which has to rely much more on random exploration in order to determine an optimal strategy.

Where the two diverge more absolutely is the training time each took. While it is roughly difficult to compare the two due to training occurring on a shared cluster, we saw the following approximate results:

# Episodes	DQN	DDPG
1000	$\tilde{2}$0 minutes	$\tilde{6}$0 minutes
5000	$\tilde{4}$0 minutes	$\tilde{5}$ Hours

The DDPG and DQN difference is not a linear difference, at least at the early stages but rather exponential. This is due to the DDPG using four different networks, the actor (agent) and critic (policy) networks, but also slowly decaying target networks for each. The DDPG also utilized a LSTM for the replay buffer, further increasing complexity as opposed to the DQN.

For future iterations of this project, with a much larger and distributed computing cluster, the author would likely go with a DDPG network. However, to improve performance, the author would create two hyperparameters, a γ and γ' for the local and target networks, respectively. During early stages of training γ should be extremely low in order to ensure the agent can operate. However, as time goes on, we would increase γ so that $\lim_{t\to\infty}\gamma = 0.99$ with the γ' parameter scaling similarly. This is to ensure that in the long-run and the agent approaches a successful completion, they become more cautious and maximize reward output.

Concluding Remarks

While we were ultimately unsuccessful in getting the agent to complete the course, this research project was ultimately fruitful in understanding reinforcement learning, modifying code to successful execute reinforcement learning jobs, and yielding meaningful business value from future applications.

First, a brief reflection on the epistemology of reinforcement learning and data science. Reinforcement learning theory does not formally fit into the epistemology of data science but is ultimately a successful tool for application of data science. While data science is fundamentally an empirical science since it uses existing data sets to design a model to predict future values or labels, data science has historically used heuristics and rational approaches. For example, while many data scientists can design a relatively useful credit card transaction fraud model, it takes deep expertise in the credit card industry to yield a well-turned model that fully takes advantage of most features. Indeed, deep learning is an attempt to make data science even more empirical by building feature engineering into an automatic function. Reinforcement learning, in contrast relies entirely on an *environment*, or a model of an environment in order to yield knowledge making it an entirely rational endeavor. Deep applications to reinforcement learning yield an even more rational discipline since data scientists are no longer focusing on γ discounting, game theory, and other intersections of mathematics and economics and rather on providing the most optimal environment. Ultimately, reinforcement learning attempts to create *actions* which is a truly unique knowledge outcome across all studies of human knowledge and wisdom. The author predicts this will ultimately yield a bifurcation between data scientists and nascent applications for reinforcement learning, where the skillets are similar but the theory of knowledge differs completely.

The author most regrets the lack of detailed coding for this practicum though still finds the experience rewarding. Ultimately, the base algorithms for DQN(Mnih et al., 2015) and DDPG(Lillicrap et al., 2015) are open-sourced and implementations will not vary terribly much. As a result, the author largely took existing source code[3] and modified it to the needs of this project. This included the following items: 1. Helper code to save videos, if possible; 2. Tensorflow 1.0 vs. Tensorflow 2.0 compatibility, and; 3. Modification of neural networks architecture.. Unfortunately item 1 become largely inapplicable due to XServer configurations and inability to complete the course and item 2 was enough to get everything running. Ultimately this left item 3 as the most significant place where modifications could happen. Significant work was done on the DQN neural network with additional density layers and regularization, though this work was ultimately removed due to a corrupted save file (though the source still has these functions). Much more substantial work was done on the DDPG, but ultimately the model would not compile in the Tensorflow space. The author wishes to see this project more as an operational and research project, and will likely create

[3] https://github.com/Kyziridis/BipedalWalker-v2.git

fresh code over time.

The business applications and value of reinforcement learning are potentially awesome. This is particularly true because business can be broken down into sub-components of reinforcement learning. A business will always have a state, either a balance sheet or a capture of the market, that can be encoded into reinforcement learning. Actions, or operations on their systems and process represents policy networks that we can train. Ultimately, since most profit functions from a business are necessarily convex (by standard definitions in economics) any business can use reinforcement learning to seek optimal market share in an arbitrary collection of other plays (the game), a market (the environment) and returns to their shareholders (the reward functions). Indeed, at the author's current company we are already deep into the process of creating such a framework and have several research-partner institutions. The limiting factor is computational time. As the author experienced it takes a material investment in machines and power to yield a successful reinforcement learning model.

Reinforcement learning is best encoded as a multi-agent game across a complex environment, meaning that a quantum strategy may work best for a solution (Guo, Zhang, & Koehler, 2008; Meyer, 1999). The environment can be thought of an arbitrarily complex quantum system with the reward function established as a measure of the energy state of the environment, or the Hamiltonian. Since $\dim(\mathcal{A}) = 24$ for this environment, and $\dim(\text{Agent}) = \dim(\text{Actor}) \oplus \dim(\text{Critic}) = 106497 + 72836 = 179833$ we get that $\dim(\hat{H}) = \dim(\mathcal{A}) \otimes \dim(\text{Agent}) \approx 4.3 \times 10^6$. Roughly speaking, a 4.3 Mega-Qubit computer could solve the Bipedal walker. While Quantum-Computing and the necessary computing power are likely decades off, the intersection of deep reinforcement learning and quantum-mechanics could create truly unique business the exhibit the first facets of Strong-AI.

References

Chen, X., & Wang, L. (2008). Promotion of cooperation induced by appropriate payoff aspirations in a small-world networked game. *Physical Review E*, *77*(1), 017103.

Fu, F., Liu, L.-H., & Wang, L. (2007). Evolutionary prisoner's dilemma on heterogeneous newman-watts small-world network. *The European Physical Journal B*, *56*(4), 367–372.

Grabbe, J. O. (2005, April). An introduction to quantum game theory. *Arxiv: Quantum Physics*.

Guo, H., Zhang, J., & Koehler, G. J. (2008, December). A survey of quantum games. *Decision Support Systems*, *46*(1), 318–332.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... Wierstra, D. (2015). *Continuous control with deep reinforcement learning.*

Meyer, D. A. (1999, February). Quantum Strategies. *Physical Review Letters*, *82*(5), 1052–1055.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, *abs/1312.5602*. Retrieved from `http://arxiv.org/abs/1312.5602`

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. Retrieved from `https://doi.org/10.1038/nature14236` doi: 10.1038/nature14236

Sutton, R., & Barto, A. (2018). *Reinforcement learning: An introduction* (2nd Edition ed.). The MIT Press.