Deep Reinfocrcement Learning: Application of DQN and DDG to the Bipedal Walker Problem

Thomas Tarler

ttarler@gmail.com

(720) 496-9222

Regis University

Deep Reinfocrcement Learning: Application of DQN and DDG to the Bipedal Walker Problem

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 Reinforcment 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 bheavior 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

Deep Reinfocrcement Learning: Application of DQN and DDG to the Bipedal Walker Problem **2**

Theory **5**

Reinforcement Learning 5

Deep Q-Networks 7

Deep Deterministic Policy Networks 10

Bipedal Walker **14**

Results **17**

Methodology 17

DQN 17

DDPG 18

Concluding Remarks **25**

References **26**

Currently, many applications of data science rely on supervised and unsupervised learning models. These are tasks associated with labelling or predicting a value for a dataset 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 Searchs, 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 dataset with either some label, value, or underlying structure associated with it. Nor do we evaluate models produced in the tradiitonal 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

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

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 heavility 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 minibatch 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 buffe 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

Inititilaize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$

Initialize replay buffer R

for epsisode = 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 minibatch 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 criticic by minimizing hte 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-Critiic* 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 pror 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 continuos 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 atributes: 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 obsevations 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.

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. The rewards over time was pulled back as a .csv file and then rendering using python graphs.

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 minorly tweaked to get them operating and returning results.

DQN

Below is the neural network used for the DQN run. It was using the following parameters:

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 2500)              62500
_________________________________________________________________
dense_2 (Dense)              (None, 2000)              5002000
_________________________________________________________________
dropout_1 (Dropout)          (None, 2000)              0
```

```
_________________________________________________________________
dense_3 (Dense)              (None, 1500)              3001500
_________________________________________________________________
dense_4 (Dense)              (None, 4)                 6004
=================================================================
Total params: 8,072,004
Trainable params: 8,072,004
Non-trainable params: 0
_________________________________________________________________
None
----------------------------------
Environment Observation_space:  Box(24,)
Environment Action_space:  Box(4,)
----------------------------------
```

Unfortunately, due to a corrupted model we did the final trains with the below specification.

Overall, since rewards were never greater than 300, we did not have a lot of success here.

DDPG

```
 \label{ActorNN}
```

```
__________________________________________________________________________________________________
Layer (type)                 Output Shape              Param #     Connected to
==================================================================================================
input_3 (InputLayer)         (None, 10)                0
__________________________________________________________________________________________________
input_4 (InputLayer)         (None, 14)                0
```

```
__________________________________________________________________________________________
dense_1 (Dense)                 (None, 256)          2816        input_3[0][0]
__________________________________________________________________________________________
dense_2 (Dense)                 (None, 256)          3840        input_4[0][0]
__________________________________________________________________________________________
dropout_1 (Dropout)             (None, 256)          0           dense_1[0][0]
__________________________________________________________________________________________
gaussian_noise_1 (GaussianNoise (None, 256)          0           dense_2[0][0]
__________________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 512)          0           dropout_1[0][0]
                                                                 gaussian_noise_1[0][0]
__________________________________________________________________________________________
dense_3 (Dense)                 (None, 128)          65664       concatenate_1[0][0]
__________________________________________________________________________________________
dense_4 (Dense)                 (None, 4)            516         dense_3[0][0]
==========================================================================================
Total params: 72,836
Trainable params: 72,836
Non-trainable params: 0
__________________________________________________________________________________________
Actor None
Model: "model_2"
__________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==========================================================================================
input_7 (InputLayer)            (None, 10)           0
__________________________________________________________________________________________
```

```
input_8 (InputLayer)            (None, 14)           0

__________________________________________________________________________________________
dense_5 (Dense)                 (None, 256)          2816        input_7[0][0]

__________________________________________________________________________________________
dense_6 (Dense)                 (None, 256)          3840        input_8[0][0]

__________________________________________________________________________________________
dropout_2 (Dropout)             (None, 256)          0           dense_5[0][0]

__________________________________________________________________________________________
gaussian_noise_2 (GaussianNoise (None, 256)          0           dense_6[0][0]

__________________________________________________________________________________________
concatenate_2 (Concatenate)     (None, 512)          0           dropout_2[0][0]
                                                                 gaussian_noise_2[0][0]

__________________________________________________________________________________________
dense_7 (Dense)                 (None, 128)          65664       concatenate_2[0][0]

__________________________________________________________________________________________
dense_8 (Dense)                 (None, 4)            516         dense_7[0][0]
==========================================================================================
Total params: 72,836
Trainable params: 72,836
Non-trainable params: 0

__________________________________________________________________________________________
Actor None
ddpg_batch.py:228: UserWarning: Update your `Model` call to the Keras 2 API: `Model(inpu
  model  = Model(input=[lidar_input,state_input,action_input], output=output)
Model: "model_3"

__________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
```

```
==================================================================================================
input_9 (InputLayer)            (None, 10)           0
--------------------------------------------------------------------------------------------------
input_10 (InputLayer)           (None, 14)           0
--------------------------------------------------------------------------------------------------
dense_9 (Dense)                 (None, 256)          2816        input_9[0][0]
--------------------------------------------------------------------------------------------------
dense_10 (Dense)                (None, 256)          3840        input_10[0][0]
--------------------------------------------------------------------------------------------------
input_11 (InputLayer)           (None, 4)            0
--------------------------------------------------------------------------------------------------
dropout_3 (Dropout)             (None, 256)          0           dense_9[0][0]
--------------------------------------------------------------------------------------------------
dropout_4 (Dropout)             (None, 256)          0           dense_10[0][0]
--------------------------------------------------------------------------------------------------
dense_11 (Dense)                (None, 256)          1280        input_11[0][0]
--------------------------------------------------------------------------------------------------
concatenate_3 (Concatenate)     (None, 512)          0           dropout_3[0][0]
                                                                 dropout_4[0][0]
--------------------------------------------------------------------------------------------------
dropout_5 (Dropout)             (None, 256)          0           dense_11[0][0]
--------------------------------------------------------------------------------------------------
concatenate_4 (Concatenate)     (None, 768)          0           concatenate_3[0][0]
                                                                 dropout_5[0][0]
--------------------------------------------------------------------------------------------------
dense_12 (Dense)                (None, 128)          98432       concatenate_4[0][0]
--------------------------------------------------------------------------------------------------
```

```
dropout_6 (Dropout)             (None, 128)          0           dense_12[0][0]
__________________________________________________________________________________________
dense_13 (Dense)                (None, 1)            129         dropout_6[0][0]
==========================================================================================
Total params: 106,497
Trainable params: 106,497
Non-trainable params: 0
```

The actor networks are relatively complex. Generally speaking, there are two neural network input spaces: the hull releadings and the lidar readings. Each of these networks is trained separate and uses a Convulitional Neural Network on the lindar inputs before meging them. Then a seperate LSTM network (if enabled). is used to combine the inputs.

Seperately, the target policy and the critic networks are generally sequential models.

```
Critic None
Model: "model_4"
__________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==========================================================================================
input_12 (InputLayer)           (None, 10)           0
__________________________________________________________________________________________
input_13 (InputLayer)           (None, 14)           0
__________________________________________________________________________________________
dense_14 (Dense)                (None, 256)          2816        input_12[0][0]
__________________________________________________________________________________________
dense_15 (Dense)                (None, 256)          3840        input_13[0][0]
__________________________________________________________________________________________
input_14 (InputLayer)           (None, 4)            0
__________________________________________________________________________________________
```

```
dropout_7 (Dropout)             (None, 256)          0           dense_14[0][0]
__________________________________________________________________________________________
dropout_8 (Dropout)             (None, 256)          0           dense_15[0][0]
__________________________________________________________________________________________
dense_16 (Dense)                (None, 256)          1280        input_14[0][0]
__________________________________________________________________________________________
concatenate_5 (Concatenate)     (None, 512)          0           dropout_7[0][0]
                                                                 dropout_8[0][0]
__________________________________________________________________________________________
dropout_9 (Dropout)             (None, 256)          0           dense_16[0][0]
__________________________________________________________________________________________
concatenate_6 (Concatenate)     (None, 768)          0           concatenate_5[0][0]
                                                                 dropout_9[0][0]
__________________________________________________________________________________________
dense_17 (Dense)                (None, 128)          98432       concatenate_6[0][0]
__________________________________________________________________________________________
dropout_10 (Dropout)            (None, 128)          0           dense_17[0][0]
__________________________________________________________________________________________
dense_18 (Dense)                (None, 1)            129         dropout_10[0][0]
==========================================================================================
Total params: 106,497
Trainable params: 106,497
Non-trainable params: 0
__________________________________________________________________________________________
Critic None
```

Due to the large number of networks used in DDPG, we were unable to achieve a success using this method. Indeed, of greatest interest is the fact that rewards tends to

plummet after a few training runs. The author suspects that this is due to the neural network running through large amounts of random torque movements to the robot walker causing it to fall down and effectively have a seizure. While the author anedotally saw occurrences of $R(t) \approx -5$ no succesful runs were completed using this methodology.

In the future, the author would completely rewrite this script to use the Tensorflow 2.0 to save models. Once checkpointing begins, the author would suspect that various improvements could be done.

Concluding Remarks

References

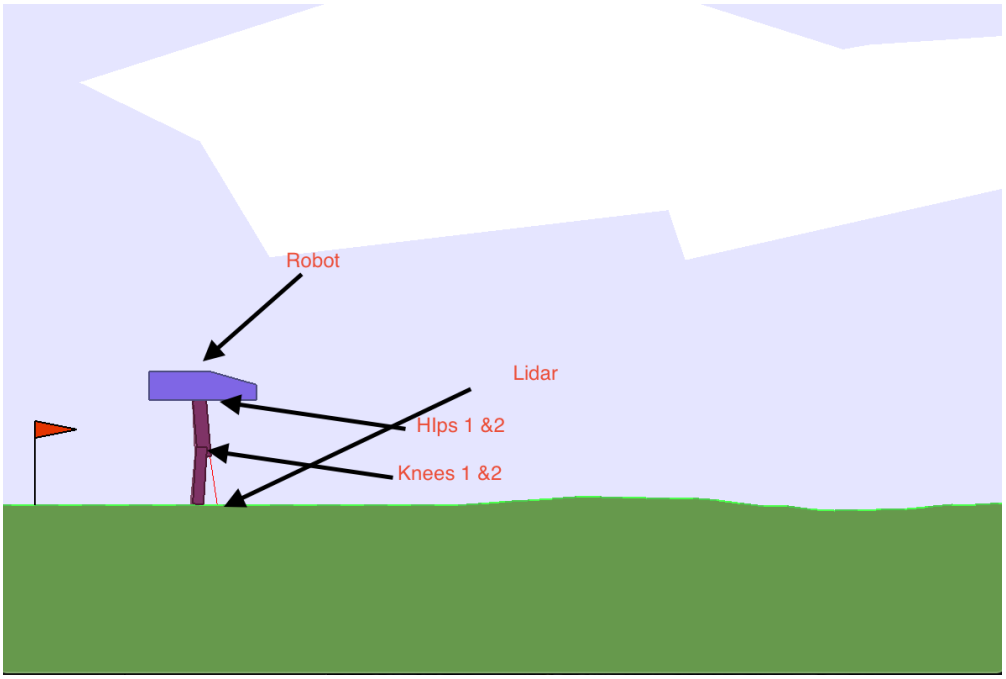
Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... Wierstra, D. (2015). *Continuous control with deep reinforcement learning.*

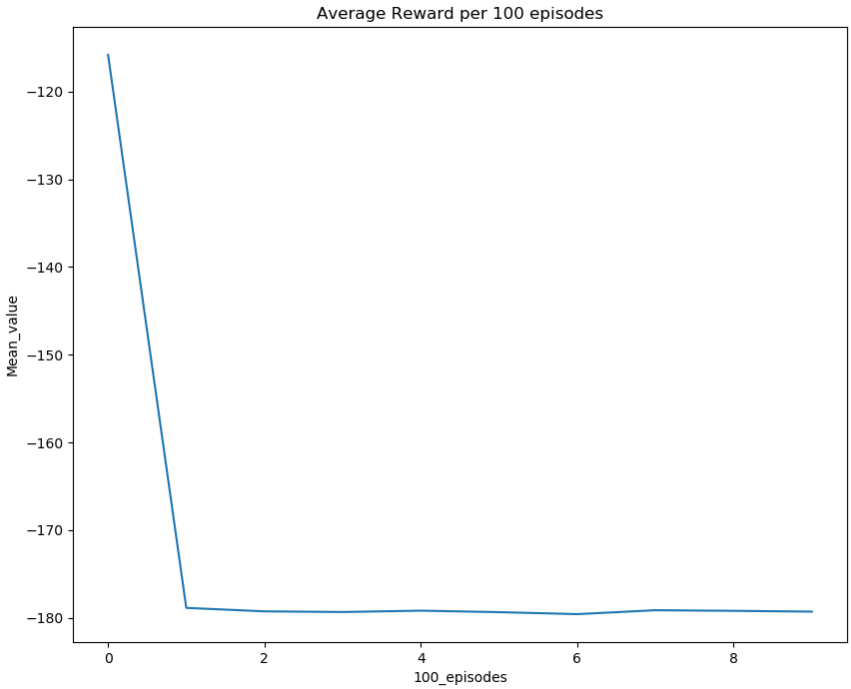
Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, *abs/1312.5602*. Retrieved from `http://arxiv.org/abs/1312.5602`

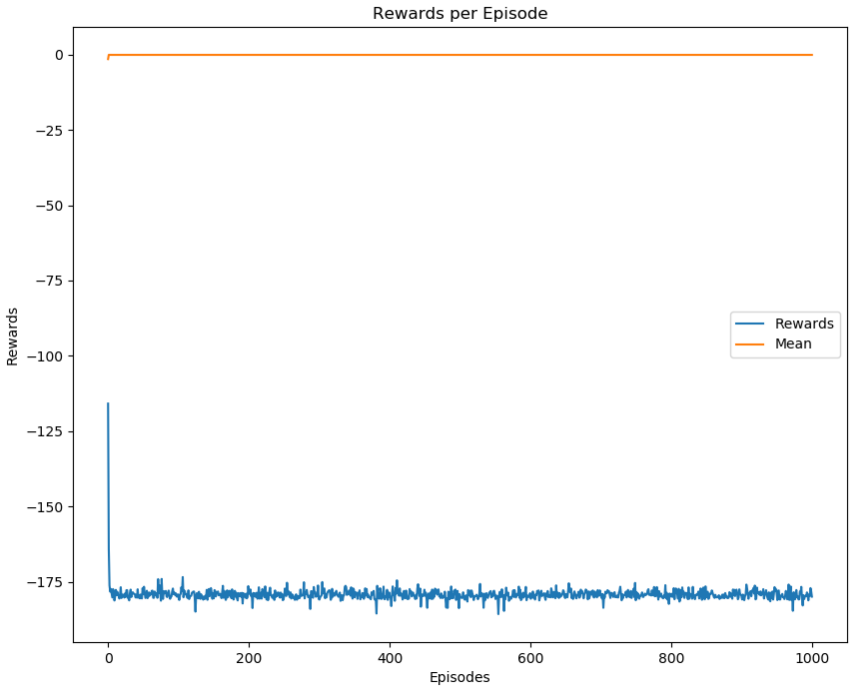
Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. Retrieved from `https://doi.org/10.1038/nature14236` doi: 10.1038/nature14236

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

Figure 1. Bipedal Walker Environment




Average Reward per 100 episodes
Mean_value
−120
−130
−140
−150
−160
−170
−180
0
2
4
6
8
100_episodes


Rewards per Episode
Rewards
Mean
Rewards
Episodes
0
−25
−50
−75
−100
−125
−150
−175
0
200
400
600
800
1000