Reinforcement Learning for Bipedal Robots using Deep Deterministic Policy Gradients

by Jordan Phillips
Student ID: B623995

Loughborough University

19COP324/5: Advanced Computer Science (ACS)

Supervisor: Dr Qinggang Meng
Submitted 28th August 2020

Abstract

Reinforcement learning has been around since before the 1980's. When R. Sutton and A. Barto publish their paper, *Reinforcement Learning: An Introduction*, reinforcement learning became a more wider reaching topic within the computer science field (R. Sutton A. Barto, 1981).

With this previous study, came deep reinforcement learning. Deep reinforcement learning is an ever-evolving field, with the area consistently growing over the years. This paper aims to progress the area forwards in a subsection of off-policy Q-learning called Deep Deterministic Policy Gradient (DDPG). The final result being a DDPG agent using the CoppeliaSim environment to model a bipedal robot and assist with action selection for the robot. DDPG is designed for continuous action spaces and alleviates heavy calculations and computation that would otherwise take place as each action is taken which are not viable in a constantly changing, continuous environment.

Acknowledgements
Many thanks to Christos Kouppas and Dr. Qinggang Meng for help and support throughout this project.

Contents

1. Introduction

1.1 Background and Problems from Literature

Reinforcement learning is the process where an agent seeks to maximize/minimize some long term reward, R, based on actions $a \in A$, taken within states, $s \in S$ as based on a reward function, *r(s,a)*.

Principally, deep reinforcement learning can be split into two main sections see Figure 1.1 on page 1, model based and model free learning, the difference being a model of the environment and a lack of such a representation respectively.

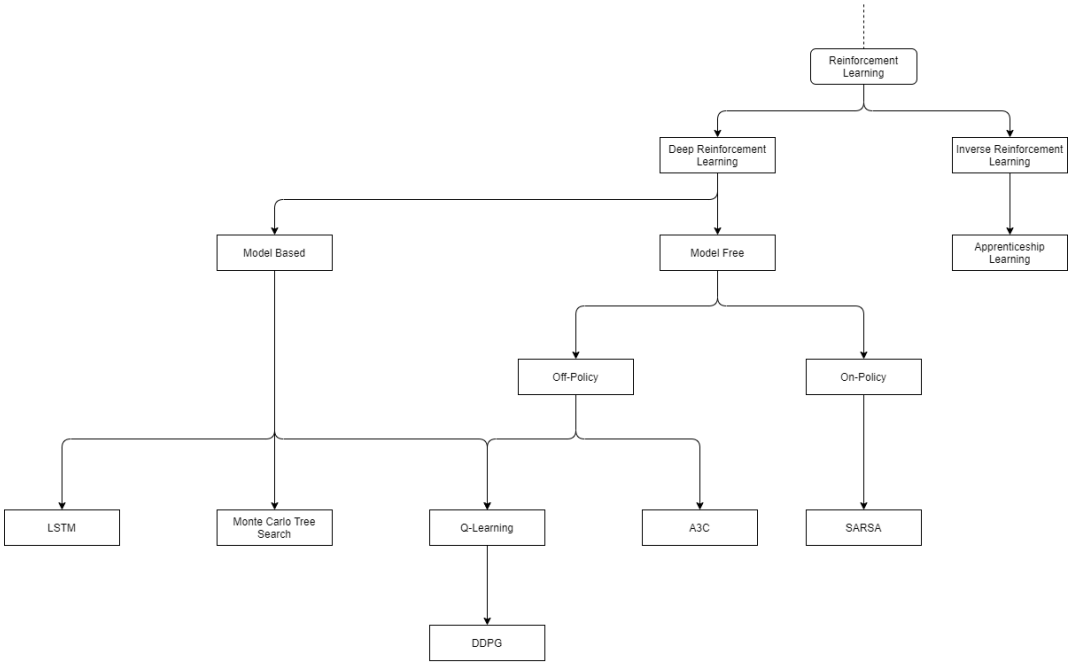

Figure 1.1: Focused expansion tree in the reinforcement learning branch of machine learning.

Figure 1.1 shows the main algorithms within the reinforcement learning area, the algorithms being leaf nodes of the graph. The leaves may expand further into subsections or variants of the algorithms may split off. Note that even though the diagram shows no connection between A3C and DDPG, DDPG uses the actor-critic model, outlined later in this section on page 2.

Model based learning is quite computationally expensive since the algorithms within this branch need a model of the entire environment to be able to map states and actions to accumulate some reward. An example of model based learning would be the 2015 DeepMind *Nature* journal paper, (Mnih et al., 2015), which used deep reinforcement Q-learning (DQN) to play the Atari 2600 library of games. The games are played in an 80x80 pixel

grid environment, compressed in the actual storage and memory batch retrieval stage of the learning process. The environment state is stored with the action that has been taken as a result of such a state. Memory storage of such magnitude means that memories have to be forgotten and dropped as to make room for new states. Processing the entire environment is made easier through preprocessing of the environment. However in larger, more complex, environments this approach does not suffice and the time taken to compute the next action may be too late for the robot or agent to react and therefore the action taken does not match the current state of the environment. Model free learning sees the most benefit in environments that cannot be realistically represented accurately with a small amount of resources within the most current computing technology.

Model free learning can be split into two further sections, Off-policy learning and On-policy learning. For this paper, we will focus on DDPG which is a subsection of Off-policy Q-learning. This branch is illustrated in Figure 1.1 on page 1. Off-policy Q-learning involves learning an optimal policy independently of the agents action, namely the bipedal robot. DDPG uses the Bellman equation to learn a Q-function and then the Q-function learns the policy (Achiam and Morales, 2020). DDPG is designed for a continuous environment space and the aim for this paper, a network that will be designed to cope with such an environment for a bipedal robot to be used in allowing the robot to walk. Since the state-action space is continuous, the DDPG algorithm is ideal to be able to handle an extraordinary amount of data (Achiam and Morales, 2020).

The problem with a continuous environment space is that an algorithm would have to run every time the robot wishes to take an action, this is trivial for a discrete environment since a max Q-value for an action, $a \in A$, can easily be found. DDPG allows the best value to be approximated and saves expensive computation (Achiam and Morales, 2020).

In 2014, (Silver et al., 2014) produced a paper exploring deterministic policy gradients and providing a framework for the actor-critic model and showing that the model greatly outperforms the stochastic counterpart (Silver et al., 2014). The implications of this paper provide the baseline for most of the papers later discussed in this section as well as this paper. The paper by (Silver et al., 2014) intentionally leaves gaps so that branches off the subject may be explored further.

DDPG originates from Google DeepMind and is one of the first deep reinforcement learning algorithms for continuous environments (Guillaume et al., 2019). In the conference paper titled *Continuous Control with Deep Reinforcement Learning*, (Lillicrap et al., 2015), they propose the new DDPG algorithm as a continuation of DQN. The aim of the paper is to overcome the shortcomings of DQN such as the fact it deals in discrete spaces with low dimensions. The DDPG network uses four NN; a Q-network, DPG network,

a target and target policy network, (Yoon, 2019). The Q-network and policy network outlined in this paper is similar to Advantage Actor-Critic (A2C). In A2C, the critic's action advantage value comes from how good the action is in the current situation and how valuable it could be. An advantage function is used to stabilise the policies of the model (Karagiannakos, 2018). However, unlike A2C, the actor in DDPG maps the states directly to actions. It was found that the DDPG algorithm performed well with simple tasks in the 2015 DDPG paper (Lillicrap et al., 2015) however for harder tasks, the Q-value estimates were worse after training than the true returns during testing. It was noted that even when the values worsened, the policies that the agent had learned were still good policies (Lillicrap et al., 2015). A precursor to the 2015 DeepMind DDPG paper and some of the inspiration behind the ideology is the DPG paper by (Mnih et al., 2015), mentioned earlier in this section. This paper, (Lillicrap et al., 2015), gives a good contribution towards DPG as it branches out a new field off of DPG in the form of DDPG. DDPG gives way for more complex environments and action spaces such as physics tasks in a continuous domain. The outcome of the paper is laying the foundations for the community to build upon the ideas outlined within.

DDPG has two primary networks, an actor and a critic and based upon actor-critic frameworks. The actor network takes an observation and returns an action. For the critic network, there are two inputs, the same observation given to the actor and the current action, which is the output of the actor network (*See Figure 1.2 on page 4.*) The critic then outputs the Q-value or reward for that action in that observation. The Q-value contains the current reward, plus a reward for the next action called the 'discounted Q-next' value. DDPG contains a replay buffer that holds the information within a set, $<$*observation*, *action*, *reward*, *next_ observation*$>$.

Let the Actor be defined as:

$$\pi_{\mu}(a_{\mathrm{t}}|s_{\mathrm{t}}) \tag{1.1}$$

and then the critic be defined as:

$$Q(s_{\mathrm{t}}, a_{\mathrm{t}}|\theta) \tag{1.2}$$

where μ is the parameters of the actor and θ is the parameters of the critic. Element a, is a chosen action of all actions, A for some state/observation, s.

In DDPG, the actor is more likely to choose a more 'reasonable' action whilst being trained, it is less likely to overfit due to target and learning networks (Watts, 2019).

The target network is another actor-critic pairing in a similar fashion to the training network. The target network will seek to maximise the Q-value output of the network. A diagram, outlining this can be shown below in Figure 1.2 on page 4. In the target network, the input to the actor is the next observation and the critic outputs the next Q-value assuming t for the training network and $t+1$ for the target network such that $Q(s_{\mathrm{t}}, a_{\mathrm{t}}|\theta)$ and $Q(s_{\mathrm{t+1}}, a_{\mathrm{t+1}}|\theta)$ respectively.

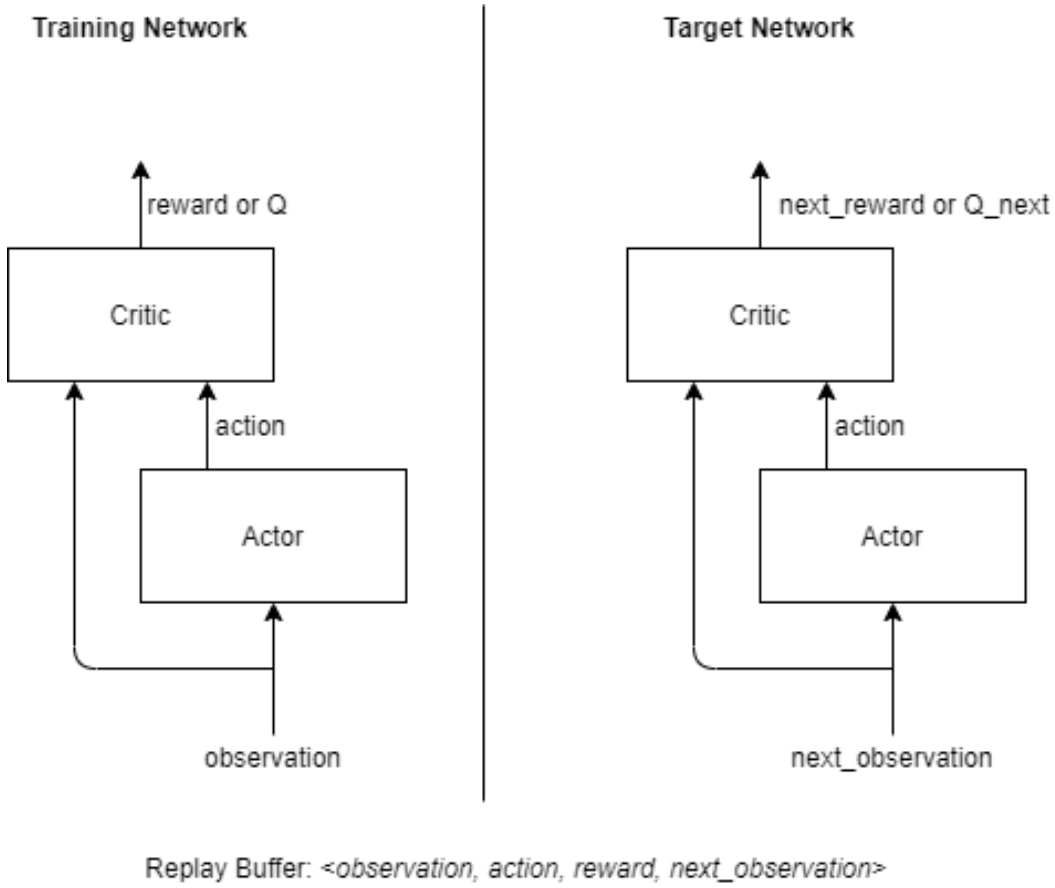

Figure 1.2: A diagram to show the inputs and outputs of the training and target networks for DDPG.

The learning network is far more unstable than the target network due to exploration or artificial noise in the training network. The target network is trained using the refined parameters learned from training, so it will be less volatile than the training network (Watts, 2019). The target network is frozen for a period of time that allow the parameters to be updated less often to avoid this volatile nature (Weng, 2018). The updates on the parameters are referred to as soft updates, whereby the parameters of the actor and critic network both receive slow updates as mentioned using the below equation:

Parameter updating for actor and critic, (Weng, 2018):

$$\tau << 1 : \theta \leftarrow \tau\theta + (1 - \tau)\theta' \tag{1.3}$$

The target is defined:

$$r + \gamma(1 - d) \max_{a'} Q_\phi(s', a') \tag{1.4}$$

where the Q-function is directed to this target when trying to minimize means squared Bellman error (MSBE) loss (Achiam and Morales, 2020). Since the target depends on parameters that are being trained, ϕ, the parameters of the target network are parameters that are close to the original where $\phi_{target} \neq \phi$ (Achiam and Morales, 2020).

DDPG has an exploration policy, ξ', to add better exploration to the agent whilst it is learning and training. The exploration can be defined as the policy combined with some noise $\mathcal{N}$:

$$\xi'(s) = \xi_\theta(s) + \mathcal{N} \tag{1.5}$$

In the paper (Lillicrap et al., 2015) and reaffirmed in the GitHub post (Weng, 2018), the DDPG model excels in robotics due to the normalization of physical units of low dimensional features. In robotics, the precise measurements of robot positioning and other physical attributes of a real world agent are never quite the same each time the agent is run. Using batch normalization the model can approximate the dimensions and normalize the model across multiple agents or robots.

A proposed version of DDPG trying to alleviate issues with inefficient exploration and unstable training comes in the form of Self-Adaptive Double Bootstrapped DDPG or SOUP in short. SOUP is estimated to achieve 45% faster learning than standard vanilla DDPG on OpenAI Gym's MuJoCo (Zheng et al., 2018). In their paper titled, *Self-Adaptive Double Bootstrapped DDPG*, Zheng et al. discuss how DDPG can struggle in complex environments, which may lead to unsuccessful convergence. This can be shown by a paper, *Reproducibility of Benchmarked Deep Reinforcement Learning Tasks for Continuous Control*, where the importance of adjusting and fine-tuning hyperparamters are difficult and can radically change the performance outcome of the algorithm (Islam et al., 2017). The Bootstrap (Efron and Tibshirani, 1994) part of SOUP works by random sample replacements which, in the paper by (Zheng et al., 2018), are outlined to be helpful for variance reduction. The paper itself provides good solutions for the problems associated with the vanilla DDPG algorithm and may help to explain some findings for the results

section later in the report.

An alternative approach to solve the same environment as a DDPG algorithm comes from (Wu et al., 2019) in the form of actor-duelling-critic (ADC). The paper proposes this new algorithm to alleviate an issue with DDPG, convergence in noisy environments. The ADC algorithm uses the familiar actor-critic framework but modifies the way in which the Q-value is estimated. To modify this estimation an advantage function was used, the value taken from the advantage function is independent from the state and environment noise (Wu et al., 2019). Results from this study show that ADC was faster to converge in noisy environments than that of vanilla DDPG, claiming progress for the training efficiency of the model. The ADC struggled in complex environments and was only conducted with a simulation which may not be truly representative of the actual performance of the robot and/or environment that the agent will exist in.

Similar to this project, is a paper titled *Implementation of Deep Deterministic Policy Gradients for Controlling Dynamic Bipedal Walking* by (Liu et al., 2019). The application of the study is to use in biomechanics and aiding people who have a walking impediment. Liu et al. developed a DDPG network for use in the Gazebo physics simulator, this simulator software is similar to V-REP/CoppeliaSim. In a journal article by (Pitonakova et al., 2018), it is explained that Gazebo does not have as many features as V-REP for example, V-REP supports more physics engines than Gazebo which offers just the one with the ability to program others (Pitonakova et al., 2018). In the article it mentions that the meshes of robots cannot be changed within the simulator and this can cause problems during iterative development of robotic systems. The article goes on to explain the poor user experience of the Gazebo system with a bad user interface (UI) and long lists of objects and models that make finding elements in the list difficult. The aim of the paper by (Liu et al., 2019) is to predict the ideal placement of the foot of a bipedal robot given external forces applied to the model. In the study, the inputs to the DDPG network were simplified although the required inside knowledge of which information can be omitted. While the implied knowledge could be seen as a drawback, the agent could learn faster and the network is able to work more efficiently due to less input data. The paper noted that the removal of knowledge that is deemed to not be important could have an unseen correlation to learning efficiency and so removing the extra knowledge could be detrimental to the efficiency of the model (Liu et al., 2019). This paper provides a good application for the DDPG network that is effective for its purpose however the study does not use real-time learning (Liu et al., 2019) and this in of itself is a flaw due to complexity that it adds to the network structure and design of the model as a whole.

1.2 Scope of the Project

The goals for this project, briefly mentioned in the introduction, are to create a DDPG neural network to best suit a continuous environment for a bipedal robot. Furthermore, to gain a greater understanding of a continuous environment for an agent, it will provide more insight to that area.

Achieving improved skills in both deep learning and neural network design, working towards deadlines and professionalism within writing, presenting and researching will also be a consequence of this project.

The project is using a simulation of a bipedal robot, opposed to a real life, physical representation. A simulation allows for more control over less variables, although the simulation does allow for complexity within various forces on the robot. Having a simulated model keeps the cost of the project lower and eliminates error with physical wear-and-tear of the robot.

1.3 Current Methods and Limitations

The project is an extension of work on-going by Christos Kouppas, Loughborough University. Working alongside their PhD project to produce comprehension of more learning methods for a bipedal robot. A limitation and strength of the study as a result is that the deadlines of the project are based upon the deadlines of the PhD project and that the project will need to be in-line with the level of detail expected of a PhD student.

An unforeseen limitation is the COVID-19 coronavirus pandemic which has slowed progress of the study due to local hardware not being as powerful or as well equipped for the task. Installing new software while the VPN and server authentication is set up will slow progress further as compatibility of the software and versions of libraries is a problem for any project using multiple dependencies on different versions of software.

1.4 Objectives

1. Put together a literature review and plan for approaching the project.
2. Become familiar with the CoppeliaSim simulation environment.
3. Use CoppeliaSim for modelling the robot.
4. Design an appropriate test for the DDPG network.

5. Create a DDPG network for training the robot.
6. Analyse the outcome of the DDPG algorithm.
7. Compare and discuss other possible approaches to the DDPG algorithm.
8. Compare the DDPG algorithm to other algorithms available.
9. Provide conclusions and discussion to the outcome of the project and draw up findings.

2. Methodology

2.1 Technical Approach

To approach the project, an understanding of the simulation environment will be required, having not used CoppeliaSim or any robotic simulator before will require some time to be put into getting familiar with the software. Furthermore, a network will be run for a period of time that will train on data for the environment as so to instruct the robot's decisions. Adapting Keras layers and designing a neural network for use in the simulation will also be required. Developing an algorithm capable of implementing DDPG will be the cornerstone of the study and comparing and contrasting the results of this paper to result gained by Christos Kouppas using DQN.

2.2 Software

The simulation software that will be used is CoppeliaSim, as mentioned above. Previously V-Rep, CoppeliaSim has been chosen as it has the ability of importing custom models. For example the bipedal SARAH robot, (Kouppas et al., 2019). The feet of the robot have a human like shape that is important for stability and this, in turn, is good for comparison to human behaviours. CoppeliaSim is a versatile platform, catering to many languages such as Python and C/C++. Having coding experience with Python and C will help the project to develop faster in the initial stages.

The language being used on the project will be the Python programming language. Python is a very popular language and is one of the most versatile and most used languages in the field. Python is very modular in nature, providing ease of installing libraries and code repositories allows the language to develop quickly and adapt to the need of the user. Whilst Python can be slow for certain applications (Shaw, 2019), the online support and community that surround the Python language are paramount.

Keras, a deep learning framework, is being used for creating the neural network. It is a library designed for fast NN designs and is the official front-end for TensorFlow. Keras is very modular and is very user friendly which makes it a good tool for developing networks, (Chollet, 2015). Chollet, F. plays a large role in the reinforcement learning and deep learning fields. Having written Deep Learning with Python in 2017, Chollet has a great understanding of the field. His creation, Keras, has been explicitly designed for the modern approach of networks and is used by many large projects for machine learning at Google, Apple, CERN and NASA, (Chollet, 2020).

2.3 Risk Assessment

The risk assessment will use a RAG (Red, Amber Green) risk level identifier, where RED = high risk, AMBER = medium risk and GREEN = low risk, (Veromann, 2019).

2.3.1 Coronavirus

A risk of this project primarily is the COVID-19 Coronavirus. With the UK in a lock-down and social distancing in place, the risk should be preventable. However, some people are in the vulnerable category of people, the risk will be RED. The impact that the virus has on the project will be large as resources in shared areas will either not be utilized or be heavily restricted due to risk of infection. The simulation software, CoppeliaSim, is available on the computers in the lab, these computers are currently not accessible. Remote access or something of that nature will be required for the project to implement the algorithm. Server access to the laboratory server also requires first a physical connection to authenticate a new user and so the lab technicians will have to authenticate a new user for the project to enable usage of laboratory resources.

2.3.2 Virtual Access

Another risk of this project is that the remote access to laboratory and library resources will be restricted to online copies which can restrict the amount of copies or users active for a particular resource. The possibility of not having access to some resources that may have been beneficial to the final project will mean that the risk for this is Green.

2.3.3 Bugs and Errors

A risk with any programming project is bugs and errors, not only the errors created by self malpractice, errors and bugs may also be a result of the libraries and code that will be imported for use. Keeping track of current versions of the software and the bugs that have been identified and outlined will help to minimize this risk. The risk of bugs for this project are Green and do not pose much threat to the project unless they persist without resolution for a long period. Bugs may be identified by the community of Keras, CoppeliaSim or Python and therefore solutions to them can be found earlier due to the aforementioned size of the community for these areas.

3. Plan and Progression

3.1 Tasks and Timeline

The tasks for this project are broken up into a Trello timeline that has been created to outline weekly goals and tasks that should arise and keep track of what has yet to be completed. Submission deadlines and key dates are highlighted within the Trello board so as to keep track of those dates. Having used Trello successfully in previous projects both alone and with groups, the benefit of Trello is known and it provides a useful tool to compare to the Gantt chart as well as talking points for the project supervisor.

3.2 Gantt Chart

The Gantt chart shown below, Figure 3.1, reflects the week to week timeline of the entire project with room for additional tasks to be set or even removed.

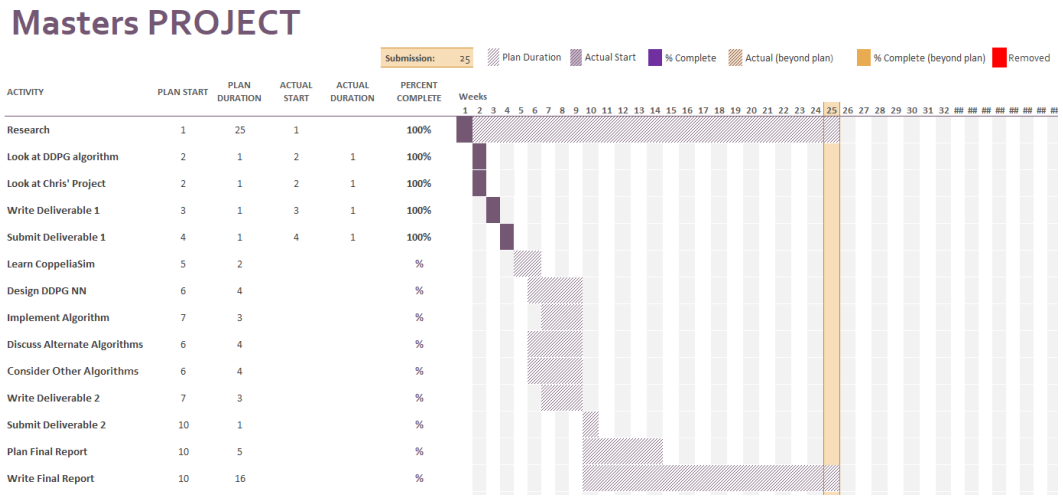

Figure 3.1: Initial Gantt chart to track progress of the project.

The second Gantt chart, on page 12, (Figure 3.2) shows the progress of the project at the time of week 10 during a progress check for the project. As can be seen from the chart, certain aspects of the project have yet to be started at the week 10 checkpoint. The implementation of the DDPG algorithm and the consideration of other algorithms for attempting the same task as DDPG. The implementation of the DDPG algorithm has been halted partly due to the COVID-19 virus and partly due to setting up the necessary infrastructure for the CoppeliaSim environment and the keras-rl (Achiam and Morales, 2020) library to run.

Bugs and incompatibility errors have slowed the project also however only after the initial DDPG algorithm is implemented, analysed and discussed in detail will the consideration of other similar algorithms to DDPG begin. Due to how the project is planned out, there is lots of time dedicated to writing the final report, this allows the timeline to shift and change as needed. Figure 3.2 shows a revised look at the Gantt chart with updates on the progress of sections in the timeline, with no progress on the two mentioned sections, the Gantt chart reflects the true progress as of week 10.

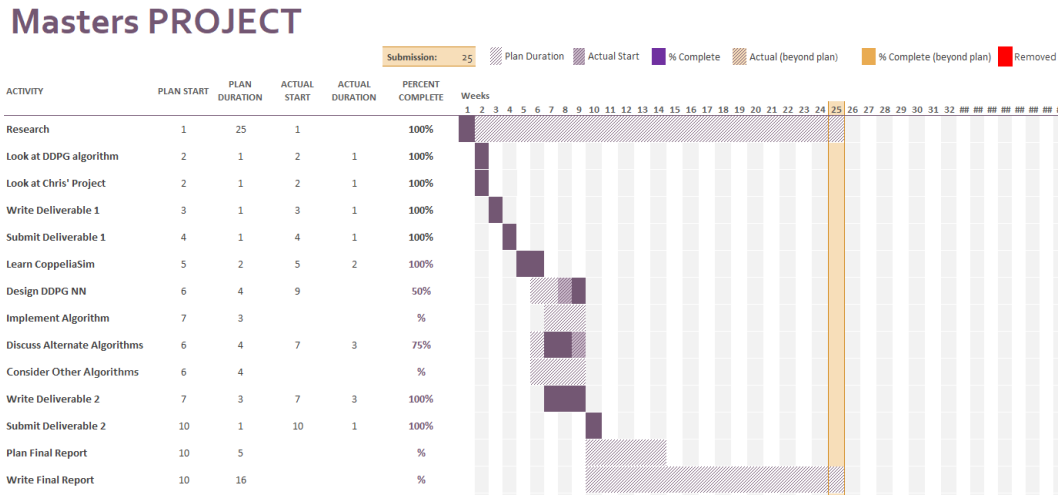

Figure 3.2: Gantt chart after 10 weeks to see how the progress of the project compares to the prediction and timeline of the project seen in Figure 3.1.

3.3 Trello Board

The project also makes use of a Trello board as mentioned, the Trello board allows a more dynamic plan of the project and keeps track of the sub tasks within a week. By breaking up the weeks into smaller tasks it is easier to keep track of what has been done and is yet to be done that week.

Trello has a card system that allows the user to move tasks around and lets the user set deadlines. Using the deadlines and card movements, it is easy to manage time and resources more effectively. An example, tasks that are taking more time can be moved a week or however long to allow for previously unforeseen delays to be taken into account.

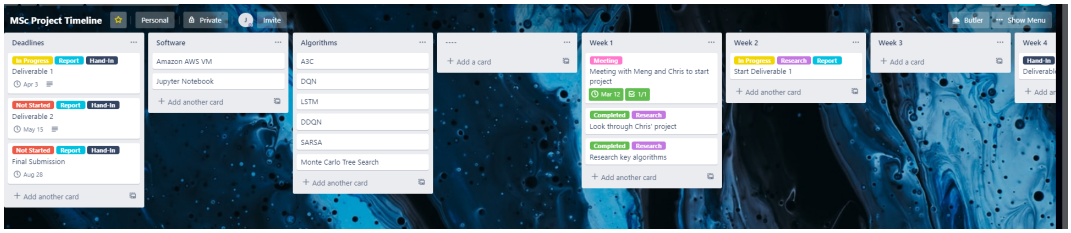

Figure 3.3: Initial Trello board to track progress of the project on a more detailed, dynamic level.

4. Algorithm Development and Test Results

The code has been adapted from Christos Kouppas's code and uses the DDPG code provided by the Keras-rl library by Matthias Plappert et al. for use as the DDPG agent. The code has been modified to deal with the dimensions of the CoppeliaSim environment which has more dimensions than that of the pendulum example the code was made for.

The idea behind DDPG is to best approximate the max action in $max_{\alpha}Q*(s,a)$, using a target network and training network to reduce volatility of the agent discussed in section 1.1 on page 1. A proposed algorithm would take the form below;

Algorithm 1: DDPG algorithm

Randomly initialize the actor $\mu(s|\theta^{\mu})$ and critic $Q(s,a|\theta^{Q})$ with weights θ^{μ} and θ^{Q}
Initialize the 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 $a_t = \mu(s_t|\theta^{\mu}) + \mathcal{N}_t$ according to the current policy and exploration noise;
    Execute action a_t and observe reward r_t and new state s_{t+1};
    Store transition (s_t, a_t, r_t, s_{t+1}) from R;
    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 critic by minimizing the loss:

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^{Q}))^2$$

    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:

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

  end
end

DDPG Algorithm. Source: (Lillicrap et al., 2015) (Weng, 2018)

The algorithm on page 14 is for a general DDPG network, using the actor-critic pairing and target actor-critic pairing shown in Figure 1.2 on page 4. The value of τ should be $\tau \in (0, 1)$. As seen from the algorithm, the agent beings by initializing some parameters and the random process. The random process is designed to get parameters of the network set up for exploration. The model is put into the initial state s_1. The for loop is the training of the model and contains the steps and action selection for the agent as it responds to the environment. An action is selected based upon the state s_t given the parameters of the actor network that were initialized as θ^μ. When the action is chosen, a reward r_t and new state s_{t+1} is observed. The observed parameters are stored in the replay buffer as shown in Algorithm 1. After the actor has executed, the critic is updated and a loss function is minimized. The target networks are then subsequently updated after the actors policy has been updated using the policy gradient calculated from the reward and future discounted rewards.

4.1 Implementation

4.1.1 Network Parameters

To begin training and testing the DDPG network, the correct hyperparameters should be found in order to correctly train and test the model. Finding the correct learning rate for the target and training networks required many tests, the target network should be less volatile than the training network and so the parameters in the target network are stabilized as such. In the target the network, the learning rate should be lower as not to learn incorrectly whilst the training network can be higher as to promote exploration of the network and not settle in a trough created by low learning rates and local minima. The results of the tests can be shown in section 4.1.2.

Initially in the the training network, the actor has a learning rate of 0.0001 and the critic has a learning rate of 0.001. The learning rates are such that the critic is more refined than the actor. This means that the learning rate can be higher since the actor is trying more aggressive solutions it needs to have a lower learning rate to avoid being stuck in local minima caused by higher learning rates.

The actor network will output the parameters of the forces that are applied to the robot, these are the most important parameters as the main objective of the network is to provide the robot with the parameters rather than the network, to make the response time faster for the robots reactions. The actor output shown is per-step the forces that are applied in a given direction to the robot. An example of the action parameters that the robot produces are shown below:

Step 1: [-87. -96. 6.]
Step 2: [42. -31. 16.]
Step 3: [-80. -14. 94.]
Step 4: [-9. 21. -2.]

The parameters of these forces are initially float types but converted to integers when the action is multiplied by 100. The multiplication is necessary for the forces used by the simulation software to function properly.

Using studies in the field of DDPG (Hossny et al., 2020) as well as articles such as MissingLink (2019), it was advised to use the TanH activation function. The TanH activation function, compared to the other activation functions, can handle extreme values better than that of other activation functions and therefore where the values mentioned above are multiplied by 100, large negative and positive values can be more easily accommodated for. In the study, (Hossny et al., 2020), they created a hybrid activation function using TanH and Sigmoid to get better results from testing at "23.15% and 33.80% increase in total episode reward" as stated in their paper. One environment that the paper used was the Pendulum-v0 from OpenAI Gym example that the code for the model used within this study is derived. The paper notes there is not a significant improvement on the pendulum example however but that the model was more stable overall. In the paper by (Hossny et al., 2020), it concludes with the success in a bipedal environment, although the environment it uses is less complex from the environment used in this study the combination of activation functions may solve some issues later in this study.

4.1.2 Initial Test

An initial test to check the performance of the network and the learning rates was done on a step limit of 100,000. The results of the initial tests are shown in Figure 4.1 and Figure 4.2 on page 17.

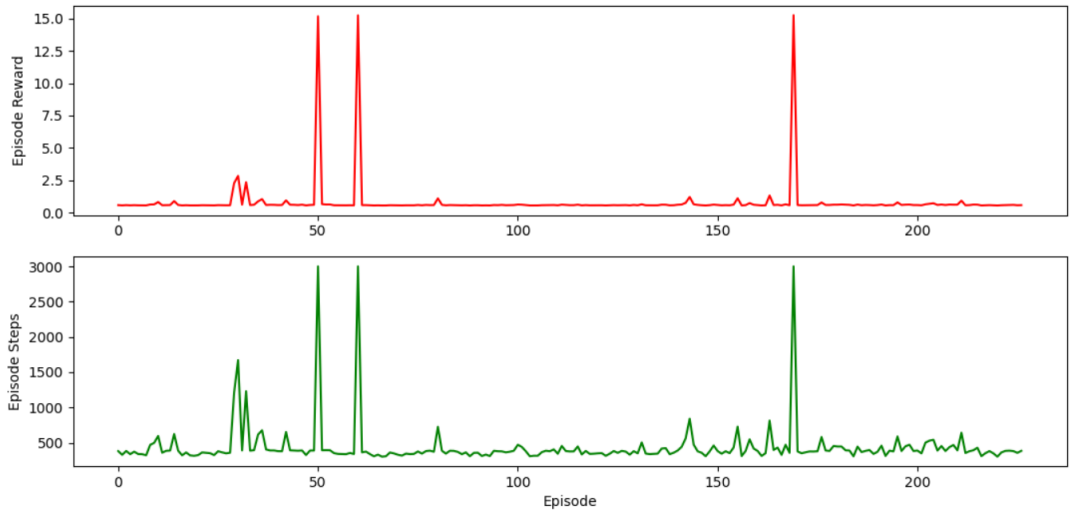

Figure 4.1: Graphs showing the episode reward and episode steps against the episode number.

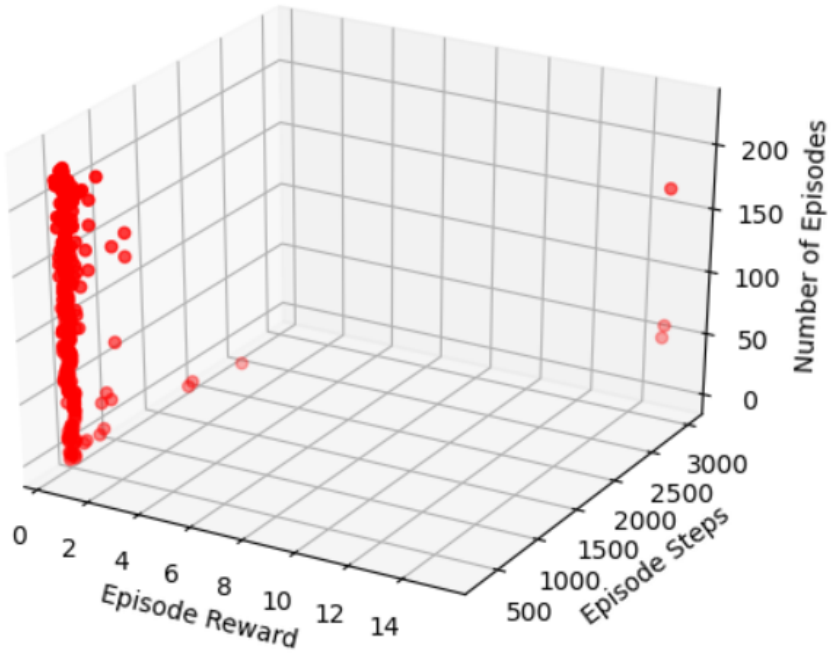

Figure 4.2: A graph to show the episode reward, episode steps and episode number in a 3 dimensional scatter graph.

As seen in the graphs above, 4.1 and 4.2, the results are not conclusive as would be expected from the test result. However, in the later episodes there is a higher frequency

of more steps taken by the robot so the network is not learning properly but it is suspected that the learning rates are not set correctly in order to learn from the model.

4.1.3 Second Preliminary Test

After the initial test a further test was run using different learning rates. The learning rates for the next test were trying to remove some of the lack of exploration and exploitation issues found within the initial test. The learning rates for this set of tests were 0.0005 for the actor network and for the critic and target networks the learning rate is 0.005. The slightly higher learning rates will make the actor more likely to exploit memory rather than explore more often than it should. A trait associated with DDPG is that it is slow to converge and this is illustrated by Figure 4.3 on page 18.

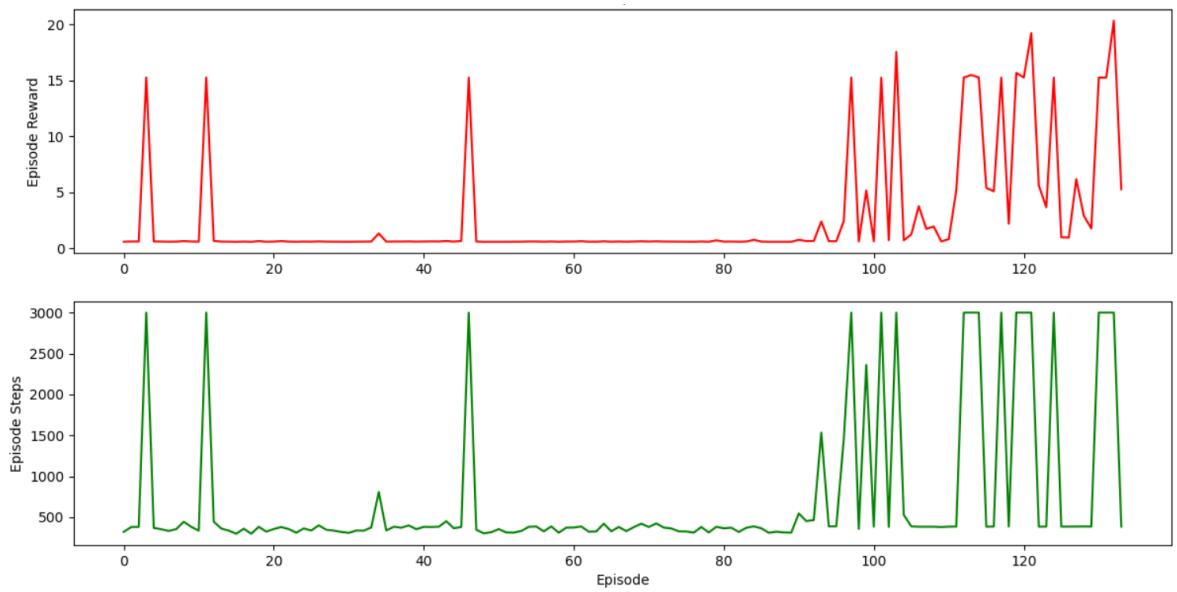

Figure 4.3: Graphs showing the episode reward and episode steps against the episode number.

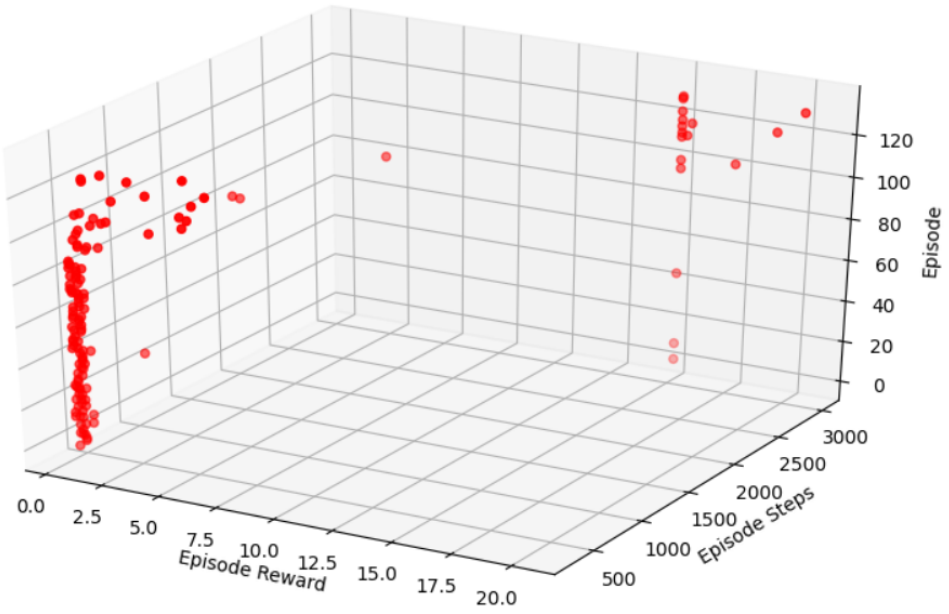

Figure 4.4: A graph to show the episode reward, episode steps and episode number in a 3 dimensional scatter graph.

From Figure 4.3 and Figure 4.4, there is improvement to the model as shown by the increase of higher scoring episodes with a maximum reward value of 19.725 compared to the initial test which yielded a maximum reward of 15.524. The model is now achieving higher step counts and reward from episode 90 onward. There is a close correlation between the episode reward and the episode steps as a criteria for the reward of the performance of the episode is the number of steps taken. Figure 4.3 shows that even when the robot achieves the maximum number of steps for an episode the reward is not always the same this is in part due to the other factors like cost and discount factor that play into account for the reward function.

4.1.4 Third Preliminary Test

The final test was a longer running simulation to see if the results from the second test, that the model was improving remained true for the duration of the simulation. In this test the parameters of the network remained the same as the prior test, however two agents were run simultaneously to compare how the model learns.

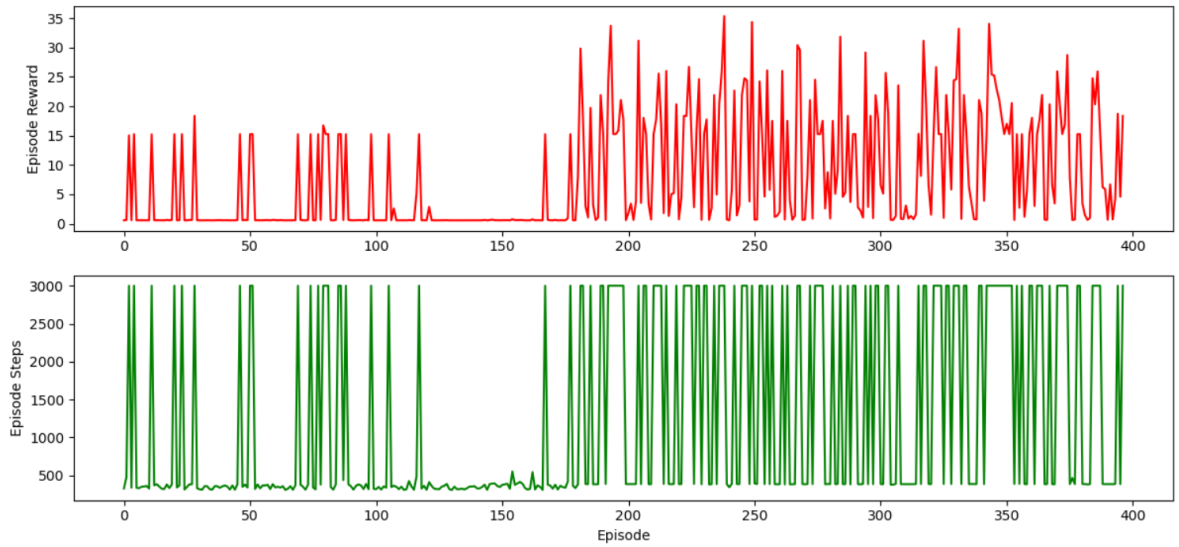

Figure 4.5: A graph to show the model training for a maximum of 500,000 steps.

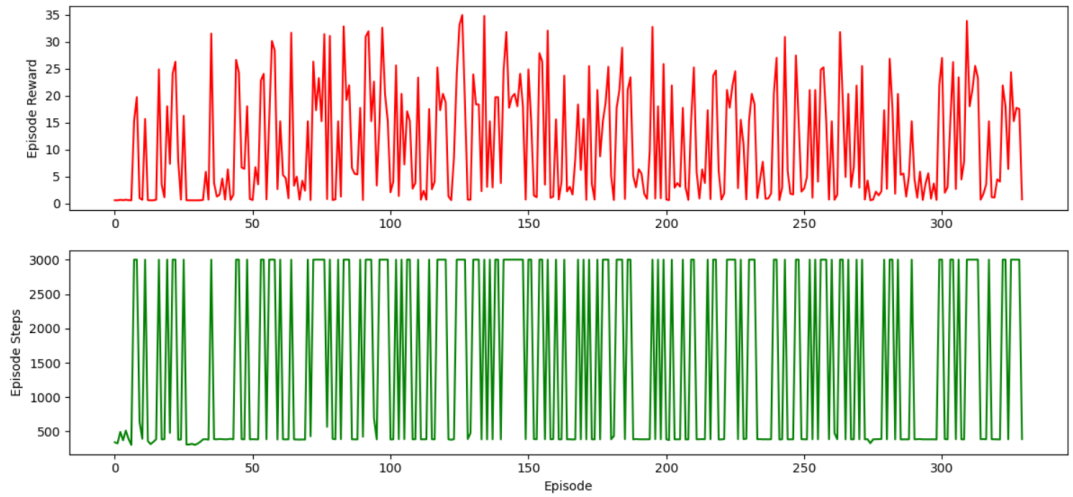

Figure 4.6: A graph to show the model training for a maximum of 500,000 steps.

In the third test, one agent performed well (Figure 4.5) and another agent (Figure 4.6) did not seem to learn at all and was almost immediately performing as well as the other agents, gaining high scores and many low scores throughout the training. The results from this third test made clear that some parameters where still not correct and also a new type of graph should be constructed to see how the reward compares to the steps of the robot.

Table 4.1 on page 21 shows that the agent can perform well in one direction of forces applied to it but it can not translate the force applied in the other direction in order to stabilize the robot. All occurrences of the 386 steps show that the robot was unable to stabilize and this resulted in terminating the simulation. Episode 9 shows a value of

Episode	Reward	Steps
1	25.238	3000
2	25.246	3000
3	18.042	3000
4	0.956	386
5	25.236	3000
6	0.956	386
7	25.236	3000
8	0.956	386
9	4.647	386
10	0.947	386

Table 4.1: Table to show the results of testing with parameters of Preliminary Test 3.

4.647 which is higher than the other low values except it is the only occurrence and does not represent the overall failure of the second direction of force.

The structure of the network might be too large, for all the preliminary tests, the size of the actor network is (16, 256, 256, 256, 256, 64, 64, 3, 3) and the critic network is (16, 19, 256, 256, 256, 256, 64, 64, 1, 1). The width of the network and the amount of connections a node has at each layer may cause the network to overfit the data.

To resolve the issue the network could be thinned and nodes removed to reduce the node connections at each layer. Another solution is that a dropout layer could be introduced to reduce the overfitting as well. In the next test a dropout layer was introduced to reduce the amount of overfitting in the model, the model was also decreased to a smaller width. The actor network was changed to (16, 16, 16, 32, 32, 32, 32, 32, 3, 3) and the critic network (16, 19, 16, 16, 32, 32, 32, 32, 32, 1, 1). The layer highlighted in red is the additional dropout layer which not only can reduce overfitting but can thin the network during the training stage.

4.1.5 Final Preliminary Test

The fourth test using the dropout layer and the smaller network was run for 50,000 steps to see how the model performs and immediately the network was seeing higher success with the testing after the training had been completed. The test was run five times and the results are shown in Table 4.2.

Episode	Reward	Steps
1	18.751	3000
2	25.373	3000
3	4.840	386
4	25.364	3000
5	18.759	3000

Table 4.2: Table to show the results of testing with a dropout layer and a smaller network size.

4.1.6 Execution of Refined Model

From the preliminary tests, it was concluded that the model needed to be run for a longer period of time to address the issue of slow convergence of the DDPG algorithm. This execution of the model involves running the model for a maximum of one million steps. During the one million steps the agent may learn, through exploration, that the failure cases, occurrences of non-3000 step episodes, can be successful as well as the direction it learns successfully.

In this execution of the model, the parameters have been kept the same in order to note the effect of a longer run time and test the effect this has on the learning of the failure cases. The results of the test are shown in Figure 4.7 on page 23.

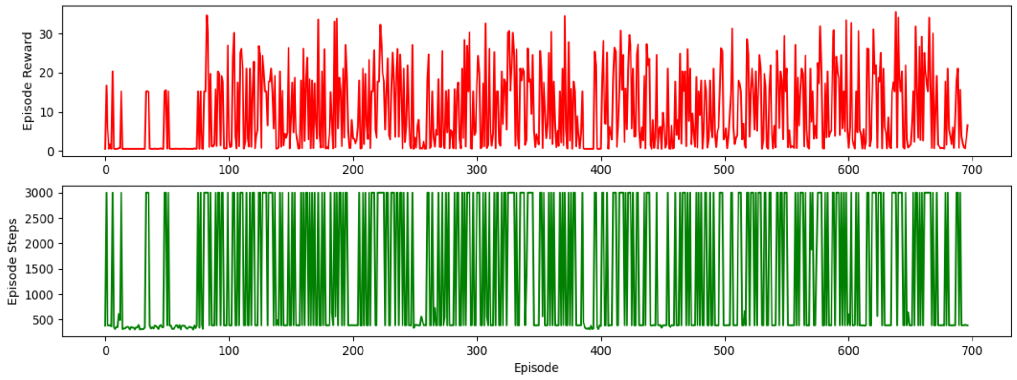

Figure 4.7: A graph to show the model training for a maximum of 1,000,000 steps.

As shown in Figure 4.7, the model starts learning as would be predicted, with low scoring ($\sim$ 15 reward score) 3000 step completions. As the model gets to episode 91, the scoring for 3000 step completions increases as the model performs better during those simulations from avoiding the no reward steps chosen by the actor in the prelude of episode 91. The trend of the graph is a loose wave pattern where the agent performs well for a period and then begins to slump back into lower scoring episodes, one such slump is around episode 200, where the agent had performed badly for a large number of consecutive episodes.

Figure 4.8: A graph to show the model training for a maximum of 1,000,000 steps.

Figure 4.8 on page 23 shows how the rewards are clustered in failure, below a reward score of 15 and success, a score above 15. The failure scores are often at the lowest possible score with some simulations running for a longer period of time (better stabilization thus higher reward) but ultimately do not succeed for 3000 steps. The maximum reward possible in the code is 50, (seen in the Appendix) which no episode had achieved in any test.

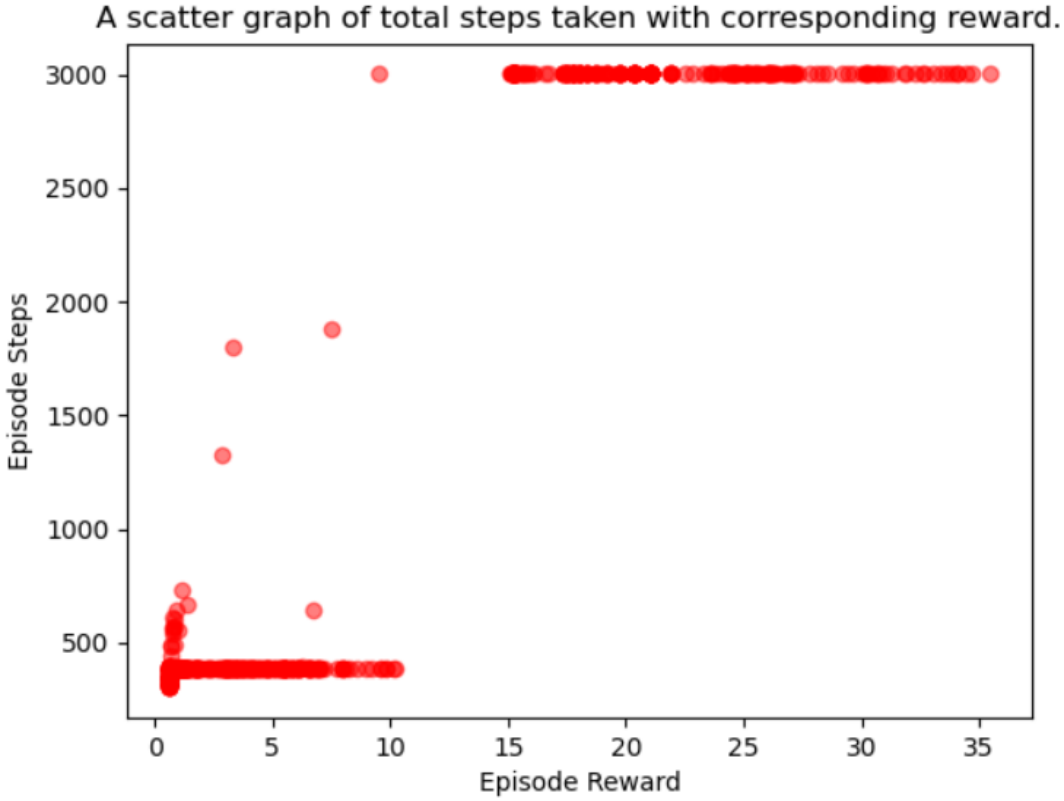

Figure 4.9: A graph to show the model training for a maximum of 1,000,000 steps.

From the scatter graph shown in Figure 4.9, the trend is similar to what has been noted prior with failure at $\sim$ 386 steps. However, in this graph you can see that there is a case where the model has performed a completion at 3000 steps but the reward is less than 15. This anomaly is part of the edge of a hypothesised line which arcs up towards the top line following the vertical part of the bottom cluster. The reason for this hypothesis is that the line would follow through the points at 1312 and 1801 reward score to join up to the top horizontal line, which would follow the reward equation. Since these higher scoring points of the scatter graph are below 3000 steps, they are failure states. The occurrence of the 1801 step, with a reward of 3.299 of the graph was late into the simulation when the model had done 810,854 steps. This could mean that the model had chosen exploration of exploitation of previous simulations and resulted in a new path being found through the network, though with poor exploration a trait of DDPG, it evidently did not happen often or more anomalies would be seen on the graph.

4.1.7 Network Structure

The finalized structure of the Actor network is shown in Figure 4.10 on page 25. The finalized structure of the critic network is shown in Figure 4.11 on page 26. The network structure is based upon the initial structure of the pendulum example found within the Keras-rl library, this is a tried and tested network that proved as a good baseline for the DDPG network to be built upon. The DDPG network has been adapted from the pendulum example and new layers have been added such as the dropout layers in both networks as well as the activation layers have been changed from ReLU. ReLU is not a good layer for this environment as it flattens the response if the values are below a threshold which would not be ideal when working with both positive and negative forces. The ReLU layers were replaced with TanH as mentioned earlier using the study (Hossny et al., 2020) for guidance.

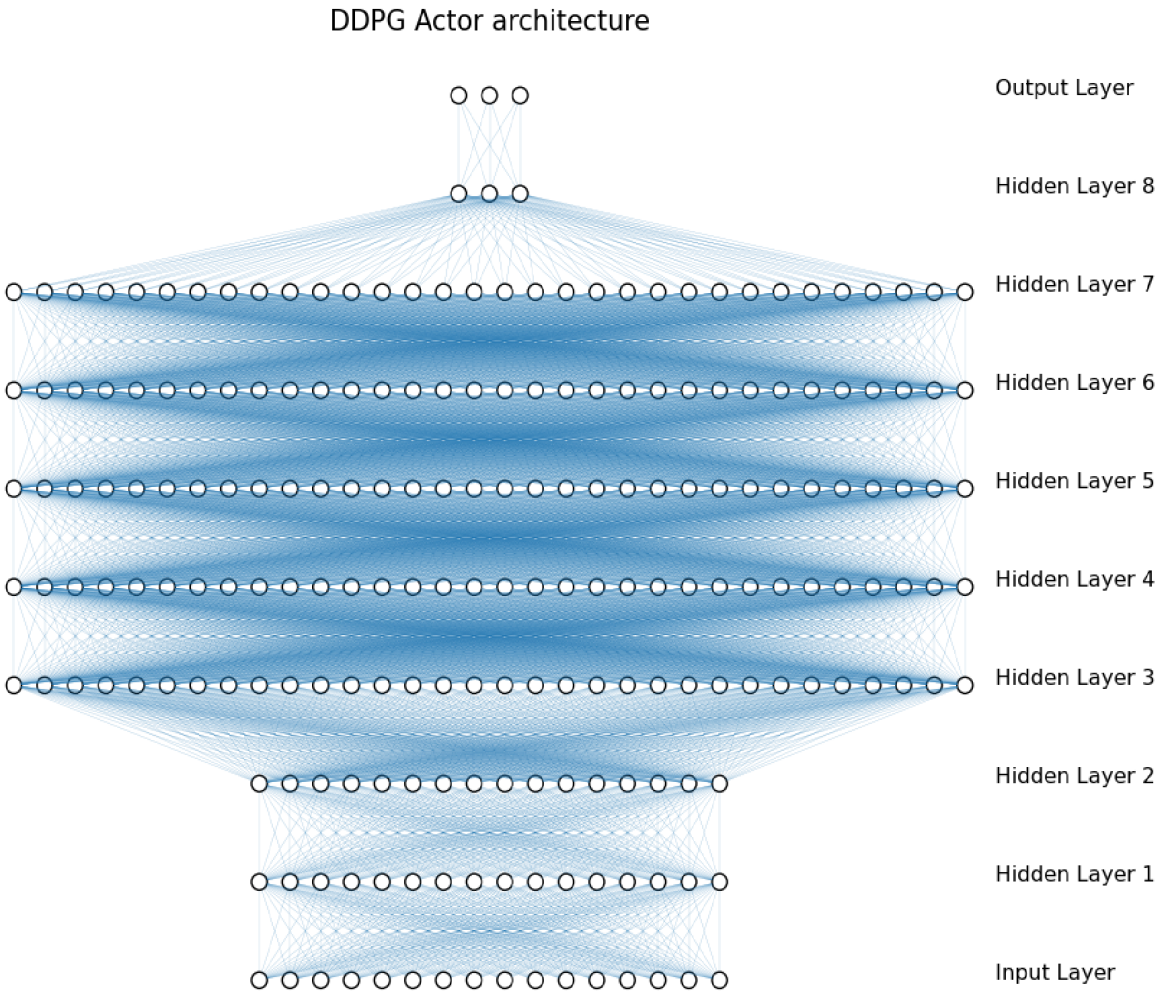

Figure 4.10: The structure of the Actor network with all hidden layers.

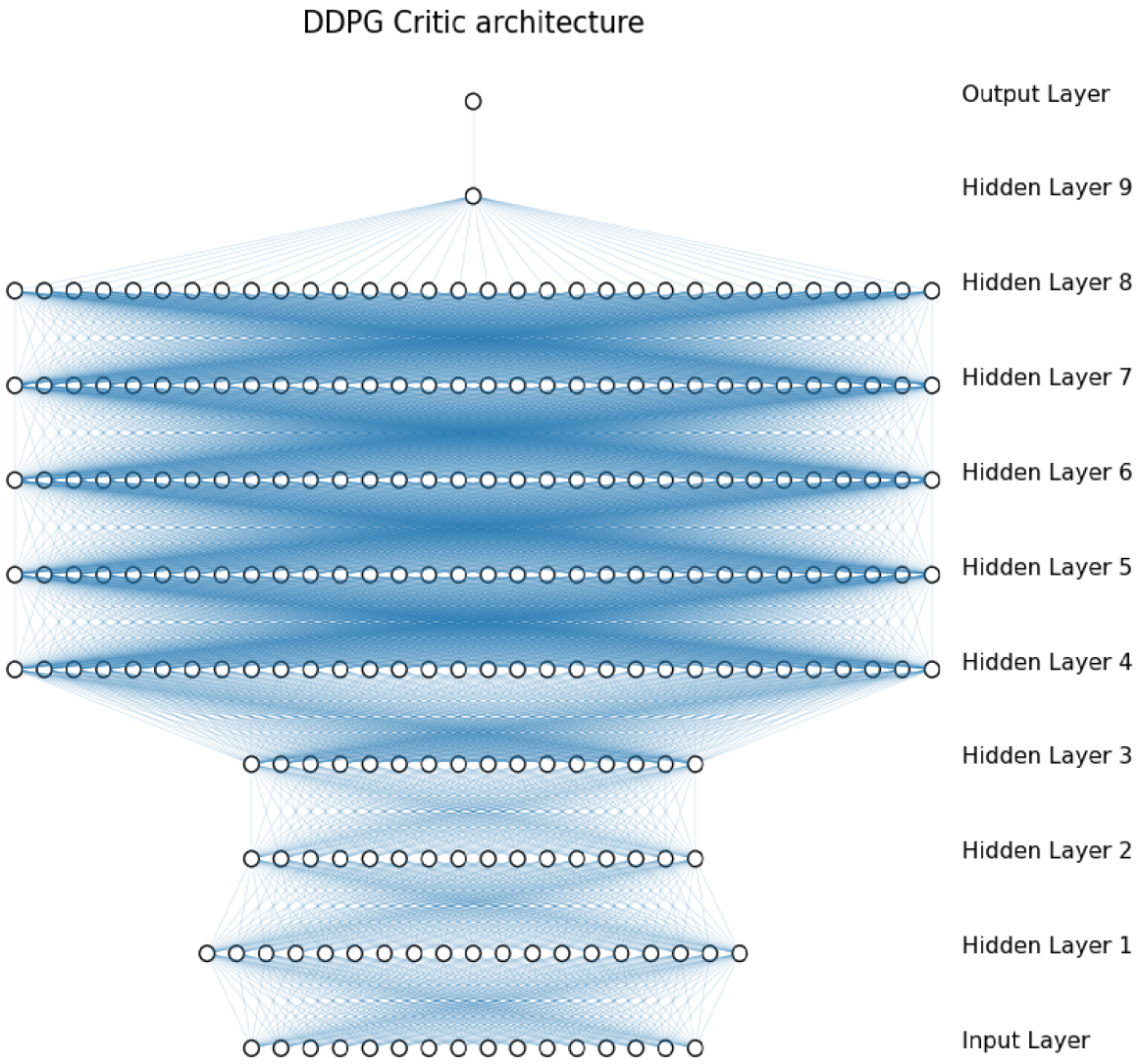

Figure 4.11: The structure of the Critic network with all hidden layers.

In the final structure of both the actor and critic networks, the dropout layer was placed in the middle of the network as opposed to the end of a network which is where dropout layers are most commonly found. Dropout layers allow the model to find more robust features of the environment (Budhiraja, 2018) and it is noted by Budhiraja, A. however, that the dropout layers increase time to converge by almost double. The benefit of more robust features allows the robot to perform better under more circumstances and since the use of the network does not extend into the robot itself, the long convergence time is almost irrelevant.

5. Analysis of Findings

In the training of the model, the DDPG agent performed well in learning from the environment and showed that it was improving as seen from Figure 4.5 on page 20. Also during testing it was shown that the DDPG network was performing worse on average than the NAF agent run by Christos Kouppas in a parallel study. The NAF agent had achieved an average steps per second of 20, whereas in most preliminary tests the DDPG agent performed at an average of 17 steps per second.

Table 4.2 on page 22 reveals that the DDPG was better than hypothesised, Christos Kouppas expressed concern for the agent only performing well within one direction of force applied to the robot. This is because the model will have to recognise that the forces may be opposite but that they can be applied in both directions with the same result, the table mentioned shows that the DDPG agent was able to resolve this issue with some success. Outlined in episode 3 in Table 4.2, this value shows a higher than expected reward for a failure state of the model. In previous iterations of this table as shown in Table 4.1 on page 21, the agent performed very poorly, every occurrence of the 386 steps means failure to stabilize the robot.

In the results as seen in Figure 4.5 on page 20 the model does learn successful responses to one direction of force and learns it quickly. The problem with the results of this graph are that the DDPG agent still fails to adapt to both the force directions enacted on the robot. On the graph outlining episode steps (coloured green) in Figure 4.5 on page 20, the agent jumps from 360-390 steps to 3000 steps constantly. These huge jumps are caused by the alternating forces being used by the simulation. Where the 3000 indicates a successful simulation, the 360-390 data points are failures from opposite forces.

6. Conclusion of Findings and Future Continuation

The final conclusion of the report is that the agent can learn to stabilize from one direction very well, in the cases of 3000 steps performed, however, the agent was not able to look at forces from the opposite direction and translate them for successful stabilization of the robot itself.

In a future study, reinforced by Christos Kouppas' suggestions, the model should have an LSTM layer at the start of the network. The LSTM layer allows the robot to successfully balance the robot using both force directions. The NAF agent run by Christos Kouppas uses this in a most recent addition and proves that it can be more successful. The addition of the new layer is not a quick solution as for an LSTM (Long Short-term Memory), this would require a lot of restructuring of the code to get it implemented. In addition, with all the unforeseen slowdowns during the project, this was unable to be fulfilled in time during this study.

In other studies using DDPG, many use alternative variations of DDPG, such as DB-DDPG (Double Bootstrapped DDPG)(Zheng et al., 2018) and MADDPG (Multi-Actor DDPG)(Lowe et al., 2017). The two aforementioned variations are designed to improve exploration, the DDPG algorithm suffers in its exploration and the model is generally slow to converge. In the other studies, using the above alternative algorithms can overcome those shortcomings of the DDPG algorithm. Exploration in the DDPG algorithm is controlled by an exploration noise as shown in Algorithm 1 on page 14. The exploration noise is not very reliable within DDPG as it is quickly updated due to the nature of actor and critic where the critic reinforces the actors good choices.

7. Challenges and Discussion

7.1 Technical Challenges

The DDPG algorithm is more complex than most algorithms previously encountered and required more effort to get the initial tests working than was anticipated in the planning phase detailed in section 3.2 on page 11. This push back halted the final writing phase of subsequent parts related to the testing and analysis of the algorithm itself. The testing that came later in the timeline was constricted further to get the project finished within the time allocated and allow ample time to write up results and conclusions. The agent must run for a long period of time and this means waiting for results for days at a time and should any errors occur during the days of training, the agent must be restarted and run again.

7.1.1 Simulation

The simulation of the environment did not require much coding to get working and was mostly in-place using the right imports. Running the simulation during the testing phase results in dramatically slower training times and overall slower learning. In the testing phase the simulation could be run to show the robot perform what it has learnt from the training phase of the code. The simulation was not run during the testing though in the code it may be activated with a simple true/false change.

7.1.2 Networks

The creation of a network capable of extracting the information from the complex CoppeliaSim environment meant that the code designed for the pendulum DDPG would need to have more nodes to draw out the features. The network for the SARAH robot was higher in dimensions than the pendulum, where the pendulum had a one dimensional action the action dimension for the SARAH robot has three. The SARAH network differs from the pendulum network in that the pendulum example uses ReLU activation layers and this means that it will suffer from the Dying ReLU Problem. This problem means that when data inputs approach 0 and or are negative, the network can not learn from this data or perform backpropagation (MissingLink, 2019). The ReLU activation functions were initially replaced with Sigmoid, this allows better learning, although it performs slower than than ReLU (MissingLink, 2019). The article by MissingLink resulted in concluding that the TanH activation function was more suited to the inputs of the model. The TanH function was chosen over other functions such as Sigmoid as the Sigmoid function does not use zero centering and since the values of forces can range far above and below zero, the TanH activation function is more appropriate.

7.1.3 Coding

The coding of the network was not too complex due to the vast resources available online and the baseline code that was provided by Christos Kouppas to incorporate the CoppeliaSim robotics environment. The coding challenges came from adding in new features from DDPG and upgrading Tensorflow 1.15.0 to Tensorflow 2.2.0 which proved to be a massive undertaking. The upgrade to Tensorflow was needed to address a bug in the Keras-rl library using Tensorflow 1.15.0 that could not be resolved in Tensorflow 1.15.0 and was fixed with Tensorflow 2.2.0. Tensorflow 2.2.0 had depreciated some features from 1.15.0 and so these older features still required were imported using tf.compat.v1.(*function*). The dependencies of libraries had to all be checked since tensorflow is used by many packages it caused a lot of errors. After changing all the afflicted files and some import errors the Tensorflow 2.2.0 version was successfully implemented, this unforeseen bug in the Tensorflow 1.15.0 source added more time to development and slowed the project significantly. Some further changes were made to the DDPG agent file in order to be able to use the new Tensorflow 2.2.0, in the files, many imports were from Keras and some from Tensorflow, since you can not use Tensorflow and Keras separately for importing, all Keras imports were changed to Tensorflow.

The network diagrams produced from the model were produced using a widely used python executable file that can be altered for use of varying sizes of networks. These files are not self created as they already existed and would be a waste of time to recreate something similar. The graphs and network diagrams were created using matplotlib which was easy to implement at the end of the main code.

7.2 Personal Challenges

Time management during the project was difficult due to unforeseen setbacks mentioned in the above sections, the huge number of setbacks in both bugs with software that were out of personal control and a steep learning curve for the project. These factors made timing very challenging and ultimately the project ran short of time for testing an LSTM layer in the model. The LSTM layer requires a lot of adaptation to the current iteration of the model and with that in mind, it was decided not to add the layer and rate success of the DDPG algorithm in a simpler form.

7.3 Project Success

The success of the project is mixed, for around 50% of the tests conducted the agent performed very well, though as outlined in Section 6, the model could only learn from the one force direction.

The success of the model is based upon objectives 4, 5 and 8 in Section 1.4 on page 7.

In objective 4, the test success is based upon these two criteria: *i) The agent can perform 3000 steps successfully without failing. ii) In the 3000 steps the reward is above 30.* In many cases, this can illustrated in Figure 4.5 on page 20, which clearly shows during training the agent scoring above 30 for reward and achieving a max step count of 3000. The model however did not achieve the above 30 scores during the testing as shown in Figure 4.1 on page 20, where a maximum reward of 25.238 was achieved. Whilst many simulations ended with 3000 steps, the reward was not above 30 for any tests conducted using the trained model.

Objective 5 was to create a DDPG capable of training the SARAH robot, the model does train the robot and the robot can be assisted by the model. The robot itself takes the output of the model and since the main outcome of the study was to use the values obtained by the model to achieve a stable robot, since only around 50% of values are successful, the project was not as successful at training the model as initially thought.

Objective 8 has been successful, although no data can be obtained from the NAF agent run by Christos Kouppas, the DDPG agent performed worse than the NAF agent. DDPG performed, at best, in some episodes around 22 steps per second, the NAF agent performed 20 consistently. The average steps per second of the DDPG agent was around 17 most of the time.

7.4 Conclusion

To conclude the study, the project has had some success, the overall goal for the study was to see if the DDPG agent was better than the NAF agent. It was hypothesised by Christos Kouppas that it would not be better than the agent used by him. From the results, even the values that the model learnt successfully were lower in both reward and steps per second achieved during training than that of the NAF agent. The testing hypothesis proposed by Kouppas was found to be true for this model that has been devised in this paper. The proposed algorithm detailed on page 1 performed the task with varying results, the premise of the algorithm works but with tweaking the layers to include an LSTM layer would most likely yield better results.

8. Index

8.1 Acronyms

A2C - Advantage Actor Critic
ADC - Actor Duelling Critic

DBDDPG - Double Bootstrapped Deep Deterministic Policy Gradient
DDPG - Deep Deterministic Policy Gradient
DPG - Deterministic Policy Gradient
DQN - Deep Q-learning Network
DRL - Deep Reinforcement Learning

LSTM - Long Short Term Memory

MADDPG - Multi-Actor Deep Deterministic Policy Gradient
MSBE - Means Squared Bellman Error

NAF - Normalized Advantage Function
NN - Neural Network

RL - Reinforcement Learning

SARAH - Safe Agile Robust Autonomous Host
SOUP - Self-Adaptive Double Bootstrapped DDPG

UI - User Interface
VM - Virtual Machine
VPN - Virtual Private Network

Bibliography

Achiam, J. and Morales, M. (2020). *Deep Deterministic Policy Gradient*. Available online at: <https://spinningup.openai.com/en/latest/algorithms/ddpg.html>. Accessed on 20th March 2020.

Budhiraja, A. (2018). *Learning Less to Learn Better-Dropout in (Deep) Machine learning*. Available online at: <https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5>. Accessed on 9th August 2020.

Chollet, F. (2015). *Keras: The Python Deep Learning library*. Available online at: https://keras.io/. Accessed on 26th March 2020.

Chollet, F. (2020). *Why use Keras?* Available online at: <https://keras.io/why-use-keras/>. Accessed on 28th March 2020.

Efron, B. and Tibshirani, R. J. (1994). *An Introduction to the Bootstrap*. Chapman Hall/CRC. Referenced: 7th May 2020.

Guillaume, M., Olivier, S., and Nicolas, P. (2019). *The Problem With DDPG: Understanding Failures in Deterministic Environments with Sparse Rewards*. Available online at: <https://openreview.net/pdf?id=HyxnH64KwS>. Accessed on 4th May 2020.

Hossny, M., Iskander, J., Attia, M., and Saleh, K. (2020). *Refined Continuous Control of DDPG Actors via Parametrised Activation*. Available online at: <https://arxiv.org/pdf/2006.02818.pdf>. Accessed on 17th August 2020.

Islam, R., Gomorokchi, M., Precup, D., and Henderson, P. (2017). *Reproducibility of Benchmarked Deep Reinforcement Learning Tasks for Continuous Control*. Available online at: <https://arxiv.org/pdf/1708.04133.pdf>. Accessed on 6th May 2020.

Karagiannakos, S. (2018). *The idea behind Actor-Critics and how A2C and A3C improve them*. Available online at: <https://theaisummer.com/Actor_critics/>. Accessed on 1st May 2020.

Kouppas, C., Meng, Q., King, M., and Majoe, D. (2019). Controlling a bipedal robot with pattern generators trained with reinforcement learning. pages 16–19.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). *Continuous control with deep reinforcement learning.* Available online at: <https://arxiv.org/pdf/1509.02971.pdf>. Accessed on 1st May 2020.

Liu, C., Lonsberry, A., Nandor, M., Audu, M., Lonsberry, A., and Quinn, R. (2019). *Implementation of Deep Deterministic Policy Gradients for Controlling Dynamic Bipedal Walking. Biomimetics*, 4(1):28. Accessed on 8th May 2020.

Lowe, R., WU, Y., Tamar, A., Harb, J., Pieter Abbeel, O., and Mordatch, I. (2017). *Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments*. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 6379–6390. Curran Associates, Inc.

MissingLink (2019). *7 Types of Activation Functions in Neural Networks: How to Choose?* Available online at: <https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/>. Accessed on 15th August 2020.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., and et al. (2015). *Human-level control through deep reinforcement learning. Nature*, 518(7540):529–533.

Pitonakova, L., Giuliani, M., Pipe, A., and Winfield, A. (2018). *Feature and Performance Comparison of the V-REP, Gazebo and ARGoS Robot Simulators. Towards Autonomous Robotic Systems Lecture Notes in Computer Science*, page 357–368. Accessed on 8th May 2020.

Shaw, A. (2019). *Why is Python so slow?* Available online at: <https://hackernoon.com/why-is-python-so-slow-e5074b6fe55b>. Accessed on 30th March 2020.

Silver, D., Lever, G., Degris, T., Weirstra, D., and Reidmiller, M. (2014). *Deterministic Policy Gradient Algorithms.* Available online at: <http://proceedings.mlr.press/v32/silver14.pdf>. Accessed on 8th May 2020.

Veromann, V.-J. (2019). *RAG B Traffic Light Rating System.* Available online at: <https://blog.weekdone.com/ragb-traffic-light-rating-system-expanding-established-design-patterns/>. Accessed on 27th March 2020.

Watts, M. (2019). *Reinforcement Learning (DDPG and TD3) for News Recommendation.* Available online at: <https://towardsdatascience.com/reinforcement-learning-ddpg-and-td3-for-news-recommendation-d3cddec26011>. Accessed on 4th May 2020.

Weng, L. (2018). Policy gradient algorithms. Available online at: <https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html>. Accessed on 5th May 2020.

Wu, M., Gao, Y., Jung, A., Zhang, Q., and Du, S. (2019). *The Actor-Dueling-Critic Method for Reinforcement Learning. Sensors*, 19(7):1547. Accessed on 7th May 2020.

Yoon, C. (2019). *Deep Deterministic Policy Gradients Explained.* Available online at: <https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b>. Accessed on 1st May 2020.

Zheng, Z., Yuan, C., Lin, Z., Cheng, Y., and Wu, H. (2018). *Self-Adaptive Double Bootstrapped DDPG. Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pages 3198 – 3204. Accessed on 6th May 2020.

9. Appendix

9.1 Jordan-MSc-Code.py

```
#@Author - Christos Kouppas
#@Secondary Author - Jordan Phillips
##For code written by Jordan is outlined in comments, imports were also ov

##The dependencies to ensure proper running of the code are listed
within the readme accompanying this file.

##This code creates a DDPG agent based on 'rl.agents.ddpg', a model
is created for the actor and critic of the DDPG agent as well as target
networks respectively.
##The DDPG agent is then compiled and fit to the model.
##After completion of the training, graphs are drawn from the training
data.
##The agent is then tested using the new model just trained for 10
episodes where the output is displayed in the command line interface


from vrepper.core import vrepper
import time
import numpy as np
import math
import tensorflow as tf
from gym import spaces
import keras


import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

import cProfile

from rl.agents.ddpg import DDPGAgent

import rl.memory
import tensorflow.keras.backend as K
```

```
from gym.utils import seeding

#Import the layers needed by the model to train and use
from tensorflow.keras.layers import Dense, Flatten, Concatenate, LSTM,
Dropout
from tensorflow.python.keras.engine.input_layer import Input
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Dense, Activation, Flatten, Input, Con

#Use the Adam optimizer
from tensorflow.keras.optimizers import Adam

from rl.random import OrnsteinUhlenbeckProcess
import os

print(tf.__version__)

#Disable eager execution, since using tensorflow 2.2.0, it is enabled
by default
tf.compat.v1.disable_eager_execution()

goal_steps = 500
score_requirement = 50
initial_games = 1000

epochs = 10  # Number or repeat of the whole Training
inner_epochs = 5  # Number of repeat of each training session
time_frame = 10  # The length of each time window
action_wait_time = 100
test_games = 10

batch_size = 32
batch_depth = 16
num_cores = 10

if False:
    num_GPU = 0
    num_CPU = 1
else:
```

```
    num_GPU = 0
    num_CPU = 1

#Configure GPUs if they are being used
config =  tf.compat.v1.ConfigProto(intra_op_parallelism_threads=num_cores,
        inter_op_parallelism_threads=num_cores, allow_soft_placement=False,
        device_count = {'CPU': num_CPU, 'GPU': num_GPU})
config.gpu_options.per_process_gpu_memory_fraction = 0.75

#Create the session for the code to run within
session = tf.compat.v1.Session(config=config)
tf.compat.v1.keras.backend.set_session(session)

##################################Source 1: Christos Kouppas, Loughborough
University PhD Student

#set up the environment
class CartPoleVREPEnv():
    def __init__(self,headless=False):
        self.venv = venv = vrepper(headless=headless)
        venv.start()
        time.sleep(0.1)
        file_path = os.path.join(os.path.dirname(os.path.realpath
        (__file__)),'Sarah_vortex_CPG.ttt')
        print(file_path)
        venv.load_scene(file_path)
        time.sleep(0.1)
        # Pressure Sensors
        self.L_Force_Front = venv.get_object_by_name('L_Force_F')
        self.L_Force_Back = venv.get_object_by_name('L_Force_B')
        self.R_Force_Front = venv.get_object_by_name('R_Force_F')
        self.R_Force_Back = venv.get_object_by_name('R_Force_B')

        # Angle Sensors / Actuators
        self.L_Abduction = venv.get_object_by_name('L_Abduction')
        self.R_Abduction = venv.get_object_by_name('R_Abduction')
        self.L_Hip_Angle = venv.get_object_by_name('L_Hip_Angle')
        self.R_Hip_Angle = venv.get_object_by_name('R_Hip_Angle')
        self.L_Linear = venv.get_object_by_name('L_Hip_Linear')
        self.R_Linear = venv.get_object_by_name('R_Hip_Linear')
```

```
        # Structural Components
        self.Center = venv.get_object_by_name('Central_Addon')
        print('(CartPoleVREP) initialized')
        obs = np.array([np.inf]*16)
        act = np.array([1.]*3)
        self.seed()

        self.action_space = spaces.Box(-act,act)
        self.observation_space = spaces.Box(-obs,obs)
        self.Frequency = 0
        self.Center_mass = 0
        self.observation = 0
        self.parameters = 0
        self.Time = 0

    def self_observe(self):
        # observe then assign
        LFF_Force, LFF_Torque = self.L_Force_Front.read_force_sensor()
        LFB_Force, LFB_Torque = self.L_Force_Back.read_force_sensor()
        RFF_Force, RFF_Torque = self.R_Force_Front.read_force_sensor()
        RFB_Force, RFB_Torque = self.R_Force_Back.read_force_sensor()
        LA_angle = self.L_Abduction.get_joint_angle()
        RA_angle = self.R_Abduction.get_joint_angle()
        LH_angle = self.L_Hip_Angle.get_joint_angle()
        RH_angle = self.R_Hip_Angle.get_joint_angle()
        LL_length = self.L_Linear.get_joint_angle()
        RL_length = self.R_Linear.get_joint_angle()

        IMU = np.array([0.]*6)
        IMU[0] = self.venv.get_float_signal('A_CA_X')
        IMU[1] = self.venv.get_float_signal('A_CA_Y')
        IMU[2] = self.venv.get_float_signal('A_CA_Z')
        IMU[3] = self.venv.get_float_signal('G_CA_X')
        IMU[4] = self.venv.get_float_signal('G_CA_Y')
        IMU[5] = self.venv.get_float_signal('G_CA_Z')
        self.observation = np.array([
            IMU[0], IMU[1], IMU[2], IMU[3], IMU[4], IMU[5],
            # 6
            math.sqrt(LFF_Force[0]**2+LFF_Force[1]**2+LFF_Force[2]**2)
```

```
            /10,# 1
            math.sqrt(LFB_Force[0]**2+LFB_Force[1]**2+LFB_Force[2]**2)
            /10,    # 1
            math.sqrt(RFF_Force[0]**2+RFF_Force[1]**2+RFF_Force[2]**2)
            /10,    # 1
            math.sqrt(RFB_Force[0]**2+RFB_Force[1]**2+RFB_Force[2]**2)
            /10,    # 1
            LA_angle, RA_angle,      # 2
            LH_angle, RH_angle,      # 2
            LL_length, RL_length     # 2
            ]).astype('float32')

    # The step function which is called for each step the robot takes
    during the testing and training
    def step(self, actions):
        self.parameters = np.around(actions*100, 0)
        self.venv.set_float_signal('Parameter_1', self.parameters[0])
        self.venv.set_float_signal('Parameter_2', self.parameters[1])
        self.venv.set_float_signal('Parameter_3', self.parameters[2])

        self.venv.step_blocking_simulation()


        # observe again
        self.self_observe()


        # cost
        self.Center_mass = self.Center.get_position()
        self.Frequency = self.venv.get_float_signal('Oscillation_
␣␣␣␣␣␣␣␣Frequency')

        # cost function
        cost = 2**(-(self.Frequency - 3)**2)
        if self.Time < 2:
            cost += 1e-6
        elif self.Time > 5:
            if self.Frequency < 0.5:
                cost = self.Time/3000
            else:
```

```
                cost *= self.Time/3000
        else:
            cost *= self.Time/30

        if (self.Center_mass[2] > 0.85) and (self.Center_mass[2] < 1)
        and (np.abs(self.Center_mass[0]) < 0.5) and \
                (np.abs(self.Center_mass[1]) < 0.5):
            done = 0
            if self.venv.get_integer_signal('Simulation_Stop') == 1:
                print('stopped')
                self.venv.stop_simulation()
                done = 1
                cost = 0
                print(self.parameters) #print the last step taken by
                the model on each episode

        else:
            done = 1
            cost = 0
            print(self.parameters) #print the last step taken by the
            model on each episode
            self.venv.stop_simulation()
        self.Time = self.venv.get_float_signal('Time')
        return self.observation, cost, done, {}

    def seed(self, seed=None):
        self.np_random, seed = seeding.np_random(seed)
        return [seed]

    def reset(self):

        self.venv.stop_simulation()
        time.sleep(0.01)
        self.venv.start_blocking_simulation()
        self.venv.step_blocking_simulation()
        self.venv.stream_integer_signal('Simulation_Stop')
        self.venv.stream_float_signal('A_CA_X')
        self.venv.stream_float_signal('A_CA_Y')
        self.venv.stream_float_signal('A_CA_Z')
        self.venv.stream_float_signal('G_CA_X')
```

```
            self.venv.stream_float_signal('G_CA_Y')
            self.venv.stream_float_signal('G_CA_Z')
            self.venv.stream_float_signal('Oscillation_Frequency')
            self.venv.stream_float_signal('Time')
            time.sleep(0.01)
            self.self_observe()

            return self.observation

        def destroy(self):
            self.venv.stop_simulation()
            self.venv.end()
            time.sleep(0.05)


def some_random_games_first():
    # Each of these is its own game.
    for episode in range(3):
        env.reset()
        # this is each frame, up to 200...but we wont make it that far.
        for t in range(100):
            # This will display the environment
            # Only display if you really want to see it.
            # Takes much longer to display it.
            # env.render()

            # This will just create a sample action in any
            environment.
            # In this environment, the action can be 0 or 1,
            which is left or right
            action = env.action_space.sample()

            # this executes the environment with an action,
            # and returns the observation of the environment,
            # the reward, if the env is over, and other info.
            observation, reward, done, info = env.step(action)
            if done:
                break
```

```
def neural_network_model(input_shape, trY):
    print('Creating um_Model...')
    model = Sequential()
    model.add(LSTM(nb_observation**2, dropout=0.1,
                batch_input_shape=(batch_size, input_shape[0],
                input_shape[1]), activation='tanh',
                    return_sequences=True, stateful=True))
    model.add(Dense(nb_observation**2, activation='sigmoid'))
    model.add(LSTM(nb_observation*5, dropout=0.1, activation='sigmoid',
    return_sequences=True, stateful=True))
    model.add(Dense(nb_observation*5, activation='sigmoid'))
    model.add(LSTM(nb_observation*5, dropout=0.1,
    activation='sigmoid',return_sequences=True,stateful=True))
    model.add(Dense(nb_observation*5, activation = 'sigmoid'))
    model.add(LSTM(nb_observation**2, dropout=0.1, activation='sigmoid',
    return_sequences=False,stateful=True))
    model.add(Dense(nb_observation**2, activation='sigmoid'))
    #model.add(LSTM(nb_observation**2, dropout=0.1, activation='sigmoid',
    return_sequences=False, stateful=True))
    model.add(Dense(trY, activation='linear', bias_initializer='ones'))
    # print(model.summary())
    return model


def V_model(input_shape):
    print('Creating V_Model...')
    V_model = Sequential()
    V_model.add(LSTM(nb_observation*5,  dropout=0.1,
                batch_input_shape=(batch_size, input_shape[0],
                input_shape[1]), activation='tanh',
                     return_sequences=True, stateful=True))
    V_model.add(Dense(nb_observation*5, activation='sigmoid'))
    V_model.add(LSTM(nb_observation*2, dropout=0.1, activation='sigmoid',
    return_sequences=True, stateful=True))
    V_model.add(Dense(nb_observation*2, activation = 'sigmoid'))
    V_model.add(LSTM(nb_observation*2, dropout=0.1,
    activation='sigmoid',return_sequences=True,stateful=True))
    V_model.add(Dense(nb_observation*2, activation = 'sigmoid'))
    V_model.add(LSTM(nb_observation*5, dropout=0.1,
    activation='sigmoid',return_sequences=False,stateful=True))
```

```
    V_model.add(Dense(nb_observation*5, activation='sigmoid'))
    #V_model.add(LSTM(nb_observation*5, dropout=0.1, activation='sigmoid',
    return_sequences=False, stateful=True))
    V_model.add(Dense(1, activation='linear'))
    # print(V_model.summary())
    return V_model


def L_model(input_shape, nb_actions):
    print('Creating L_Model...')
    action_input = Input(shape=(nb_actions,), name='action_input')
    observation_input = Input(input_shape, name='observation_input')
    x = Concatenate()([action_input, Flatten()(observation_input)])
    # x = concatenate([Flatten()(observation_input), action_input])
    # x =Reshape((time_frame, nb_observation+nb_actions))(x)
    # x =LSTM(nb_observation*5, activation='sigmoid',
    return_sequences=True, stateful=False)(x)
    x = Dense(nb_observation, activation='tanh')(x)
    # x =LSTM(nb_observation*5, activation='sigmoid',
    return_sequences=True, stateful=False)(x)
    x = Dense(nb_observation, activation='sigmoid')(x)
    x = Dense(nb_observation, activation='sigmoid')(x)
    x = Dense(nb_observation, activation='sigmoid')(x)
    x = Dense(nb_observation, activation='sigmoid')(x)
    # x =LSTM(nb_observation*5, activation='sigmoid',
    return_sequences=False, stateful=False)(x)
    x = Dense((nb_actions * nb_actions + nb_actions) // 2,
    activation='linear')(x)  # ((nb_actions * nb_actions +
    nb_actions) / 2))
    L_model = Model(inputs=[action_input, observation_input],
    outputs=x)
    # print(L_model.summary())
    return L_model

##Clear the graph of the session to remove any mess left over
by the session before - Jordan
K.clear_session()
tf.compat.v1.reset_default_graph()
```

```
##set up the VREP environment
env = CartPoleVREPEnv(headless=True)
np.random.seed(123)
env.seed(123)
assert len(env.action_space.shape) == 1
nb_actions = env.action_space.shape[0]

nb_observation = env.observation_space.shape[0]
some_random_games_first()


print("Random Played. Starting Population.")

memory = rl.memory.SequentialMemory(limit=100000, window_length=1)
random_process = OrnsteinUhlenbeckProcess(size=nb_actions, theta=.15,
mu=0., sigma=.3)

##################################################################End of source 1

# Create a placeholder for the input shape to ensure that this is
enforced by the first layer of the actor - Jordan
input_shape_j = np.empty([16,1])

#####################################################################
#BELOW IS CODE CREATED BY JORDAN PHILLIPS B623995###########
#####################################################################

##########################ACTOR-------START#####################################
# The actor is created as a sequential model and all layers are
configured, at the end, the model structure is outputted

actor = Sequential()

actor.add(Flatten(input_shape=(1,) + env.observation_space.shape))
actor.add(Dense(16))
actor.add(Activation('tanh'))
actor.add(Dense(32))
actor.add(Activation('tanh'))
actor.add(Dropout(0.05))
```

```
actor.add(Dense(32))
actor.add(Activation('tanh'))
actor.add(Dense(nb_actions))
actor.add(Activation('tanh'))

print(actor.summary())


#############################ACTOR-------END#########################################

# Create the variables for the input to the critic network and flatten
the observation input to match the shape of the network
action_input = Input(shape=(nb_actions,), name='action_input')
observation_input = Input(shape=(1,) + env.observation_space.shape,
name='observation_input')
flattened_observation = Flatten()(observation_input)


#############################CRITIC-------START#####################################
# Create the critic network, ensure that the Reshape() comes from
tensorflow, a workaround to a bug encountered during creation
# Print the summary of the network to the user in the command line
interface


x = Input(batch_shape=(None, 16))
x = tf.keras.layers.Reshape((16,))(x)
x = Concatenate()([action_input, flattened_observation])
x = Dense(16)(x)
x = Activation('tanh')(x)
x = Dense(32)(x)
x = Activation('tanh')(x)
x = Dropout(0.05)(x)
x = Dense(32)(x)
x = Activation('tanh')(x)
x = Dense(1)(x)
x = Activation('tanh')(x)

critic = Model(inputs=[action_input, observation_input], outputs=x)
```

```
print(critic.summary())

##########################CRITIC----------END#####################################

# Create the ddpg agent using the models that are defined above, define
learning rate for target networks within and the discount factor, gamma
# gamma = 0.9 is the discount factor of future rewards
ddpg = DDPGAgent(nb_actions=nb_actions, actor=actor, critic=critic,
critic_action_input=action_input,
                                memory=memory, batch_size=32,
                                nb_steps_warmup_critic=5000,
                                nb_steps_warmup_actor=5000,

                                random_process=random_process, gamma=0.9,
                                target_model_update=5e-3)


# .compile() is used to configure the model with losses and metrics.
# The learning rate of actor and critic are entered as arguments below
respectfully
ddpg.compile([Adam(lr=5e-4), Adam(lr=5e-3)], metrics=['mae'])

# show the metrics of the model that can be analysed in graphs
print(ddpg.metrics_names)


# .fit() is used to train the DDPG model
# 3000 max steps specified by Christos Kouppas
# Test the agent until it reaches 1 million total steps and print the
output in a verbose 2 styling.
history = ddpg.fit(env, nb_steps = 1000000, visualize=False, verbose=2)

# set up variables to store agent data from training to be used in graphs
history_dict = history.history
print(history_dict.keys())
episode_rew = history.history['episode_reward']
episode_steps = history.history['nb_episode_steps']
number_steps = history.history['nb_steps']
```

```
# create a variable to store the amount of episodes completed in the
1 million step limit,
# as this is a variable amount the range will need to be able to
cater to this changing number
num_of_eps = len(episode_rew)
num_of_eps = range(0,num_of_eps)

print(num_of_eps)

##################################Print two 2D line graphs in a subplot
of height = 2

plt.subplot(2, 1, 1)
plt.plot(num_of_eps, episode_rew, 'r')
plt.title('Graphs')
plt.ylabel('Episode Reward')

plt.subplot(2, 1, 2)
plt.plot(num_of_eps, episode_steps, 'g')
plt.xlabel('Episode')
plt.ylabel('Episode Steps')

plt.show()

##################################Print a 3D scatter graph

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

x_set = episode_rew
y_set = episode_steps
z_set = num_of_eps

ax.scatter(x_set, y_set, z_set, c='r', marker='o')

ax.set_xlabel('Episode Reward')
ax.set_ylabel('Episode Steps')
ax.set_zlabel('Episode')

plt.show()
```

```
##################################Print a 2D scatter graph

plt.scatter(episode_rew, episode_steps, c='r', alpha=0.5)
plt.title('A scatter graph of total steps taken with corresponding
reward.')
plt.xlabel('Episode Reward')
plt.ylabel('Episode Steps')
plt.show()

plt.scatter(num_of_eps, episode_rew, c='b', alpha=0.5)
plt.title('A scatter graph of reward per episode.')
plt.xlabel('Episode')
plt.ylabel('Episode Reward')
plt.show()

#################################################################
###############Testing the model that has been trained
#################################################################
#The model is now tested for 10 episodes to see how it performs
using the training it has undergone


print("Online Testing")
env.destroy
time.sleep(1)
env2 = CartPoleVREPEnv(headless=True)
time.sleep(1)
ddpg.test(env2, nb_episodes=10, visualize=False,
nb_max_episode_steps=3000)

# end simulation
print('simulation ended. leaving in 5 seconds...')
time.sleep(1)
```