TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Reinforcement Learning for Swing Up and Balancing of Three-Dimensional Humanoid Model

Author:
Panagiotis PAPADIMITRIOU

Supervisor:
Associate Prof.
Michail G. LAGOUDAKIS
Committee:
Prof. Michalis ZERVAKIS
Dr. Vasilis DIAKOLOUKAS

A thesis submitted in fulfillment of the requirements for the degree of Engineering Diploma

in the

Intelligent Systems Laboratory
School of Electrical and Computer Engineering

April 6, 2021

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

Διπλωματική Εργασία

Ενισχυτική Μάθηση για Αιώρηση και Ισορροπία ενός Τρισδιάστατου Ανθρωποειδούς Μοντέλου

Συγγραφέας:
Παναγιώτης
ΠΑΠΑΔΗΜΗΤΡΙΟΥ

Επιβλέπων:
Αναπληρωτής Καθηγητής
Μιχαήλ Γ. ΛΑΓΟΥΔΑΚΗΣ
Εξεταστική επιτροπή:
Καθηγητής
Μιχάλης ΖΕΡΒΑΚΗΣ
Δρ. Βασίλης ΔΙΑΚΟΛΟΥΚΑΣ

Διπλωματική εργασία
για την απόκτηση του διπλώματος Μηχανικού

στο

Εργαστήριο Προγραμματισμού και Τεχνολογίας Ευφυών Υπολογιστικών Συστημάτων
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

6 Απριλίου 2021

Declaration of Authorship

I, Panagiotis PAPADIMITRIOU, declare that this thesis titled, "Reinforcement Learning
for Swing Up and Balancing
of Three-Dimensional Humanoid Model" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for an engineering degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Panagiotis Papadimitriou

Date: April 6, 2021

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Engineering Diploma

**Reinforcement Learning
for Swing Up and Balancing
of Three-Dimensional Humanoid Model**

by Panagiotis PAPADIMITRIOU

Reinforcement Learning, as a subfield of Artificial Intelligence and Machine Learning, has gained a lot of traction in recent years. From trained agents playing video games or chess at expert level to self-driving cars in the streets, a lot of groundbreaking results have been achieved thanks to advances in Reinforcement Learning. The combination of Reinforcement Learning and Robotics has the additional advantage that agents trained in simulation could eventually be carried over to real robots that can be utilized in varying tasks to aid humans. In this diploma thesis, we construct a 3-dimensional humanoid model hanging below a horizontal bar (an acrobat) within a realistic simulation environment, based on humanoid model originally made for walk learning experiments. The goal of the agent that controls the actions of the humanoid model is to swing up and eventually balance the humanoid model on the bar. The challenge in this problem is the high-dimensional and continuous state and action space, since the model has 19 degrees of freedom (joints) and 17 actuators (motors), a case where conventional learning approaches do not apply. We try out two Reinforcement Learning algorithms: Deep Deterministic Policy Gradient (DDPG) and Advantage Actor-Critic (A2C) to train the agent using thousands of trials and we demonstrate the learning progress. A simple reward scheme was adopted that rewards the agent proportionally to the height reached at any time, but does not reveal any information about the nature of the problem. Through the extensive experimentation we conducted with both algorithms and some variations of the model, we deduced that the most efficient algorithm and a better fit to the problem at hand was DDPG, which through some tuning of the problem parameters yielded satisfying results. The resulting agent after learning is able to complete the task in most trials from any starting pose.

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

Περίληψη

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Δίπλωμα Μηχανικού

Ενισχυτική Μάθηση
για Αιώρηση και Ισορροπία
ενός Τρισδιάστατου Ανθρωποειδούς Μοντέλου

Παναγιώτης ΠΑΠΑΔΗΜΗΤΡΙΟΥ

Η Ενισχυτική Μάθηση, ως υποπεδίο της Τεχνητής Νοημοσύνης και της Μηχανικής Μάθησης, έχει γίνει αρκετά δημοφιλής τα τελευταία χρόνια. Από εκπαιδευμένους πράκτορες που παίζουν βιντεοπαιχνίδια ή σκάκι σε επίπεδο εμπειρογνωμόνων έως και αυτοοδηγούμενα οχήματα στους δρόμους, έχουν επιτευχθεί πολλά πρωτοποριακά αποτελέσματα χάρη στις εξελίξεις στην Ενισχυτική Μάθηση. Ο συνδυασμός της Ενισχυτικής Μάθησης και της Ρομποτικής έχει το πρόσθετο πλεονέκτημα ότι πράκτορες εκπαιδευμένοι σε προσομοίωση θα μπορούσαν τελικά να μεταφερθούν σε πραγματικά ρομπότ που μπορούν να χρησιμοποιηθούν σε ποικίλες εργασίες για να βοηθήσουν τους ανθρώπους. Σε αυτή τη διπλωματική εργασία, κατασκευάζουμε ένα τρισδιάστατο μοντέλο ανθρωποειδούς που κρέμεται από μια οριζόντια ράβδο (ένας ακροβάτης) μέσα σε ένα ρεαλιστικό περιβάλλον προσομοίωσης, βασιζόμενοι σε μοντέλο ανθρωποειδούς που αρχικά κατασκευάστηκε για πειράματα μάθησης για βάδισμα. Ο στόχος του πράκτορα που ελέγχει τις κινήσεις του μοντέλου ανθρωποειδούς είναι να αιωρηθεί προς τα πάνω και τελικά να εξισορροπήσει το μοντέλο ανθρωποειδούς πάνω στη ράβδο. Η πρόκληση σε αυτό το πρόβλημα είναι ο πολυδιάστατος και συνεχής χώρος κατάστασης και δράσης, καθώς το μοντέλο έχει 19 βαθμούς ελευθερίας (αρθρώσεις) και 17 ενεργοποιητές (κινητήρες), μια περίπτωση όπου οι συμβατικές προσεγγίσεις μάθησης δεν εφαρμόζονται. Δοκιμάζουμε δύο αλγόριθμους ενισχυτικής μάθησης: Deep Deterministic Policy Gradient (DDPG) και Advantage Actor-Critic (A2C) για να εκπαιδεύσουμε τον πράκτορα χρησιμοποιώντας χιλιάδες δοκιμές και καταδεικνύουμε την πρόοδο της μάθησης. Εφαρμόστηκε ένα απλό σχήμα ανταμοιβής που επιβραβεύει τον πράκτορα ανάλογα με το ύψος που έχει φτάσει ανά πάσα στιγμή, αλλά δεν αποκαλύπτει πληροφορίες σχετικά με τη φύση του προβλήματος. Μέσα από τον εκτεταμένο πειραματισμό που πραγματοποιήσαμε και με τους δύο αλγόριθμους και με κάποιες παραλλαγές του μοντέλου, καταλήξαμε στο συμπέρασμα ότι ο πιο αποτελεσματικός αλγόριθμος και η καλύτερη προσέγγιση στο πρόβλημα ήταν ο DDPG, ο οποίος μέσω κάποιων ρυθμίσεων των παραμέτρων του προβλήματος απέδωσε ικανοποιητικά αποτελέσματα. Ο πράκτορας που προέκυψε μετά τη μάθηση μπορεί να πετύχει τον στόχο στις περισσότερες δοκιμές ξεκινώντας από οποιαδήποτε αρχική στάση.

Acknowledgements

First and foremost I would like to express my gratitude to my supervisor Associate Prof. Michail G. Lagoudakis for his guidance and advice throughout my studies. I would also like to thank my colleagues for their assistance and support…

Contents

List of Figures

Dedicated to my beloved supporting family…

Chapter 1

Introduction

The human brain and body movements, have always been a challenging task and an inspiration for the field of Artificial Intelligence. The human intellect has been the milestone in generating software that can mimic human behaviour. The simplest human behaviours and actions such as standing up, walking and speaking that a person learns in the earliest stages of its life can be quite challenging for a computer software to learn.
The main idea of training a human or an animal how to behave correctly is to reward them when they follow the correct behaviours. This is the same idea that Reinforcement Learning is based on, where an agent is trained to do a task on which it has no prior knowledge on and step by step through trial and error with rewards or penalties the agent will potentially learn to do the task at hand. With recent advancements in these fields, Reinforcement Learning agents can be successful in tasks with high complexity, including real life or simulated robotic environments, with the end goal being the use of properly trained robots to be used in society.
Reinforcement Learning combined with robotics could eventually provide reliable robots that can help humans with numerous demanding tasks where a human could struggle or be used in the place of a human where there is danger involved. Robots could also be trained in other miscellaneous tasks as the one described below.

1.1 Problem Statement

In this thesis we aim to formulate a problem and find its solution using Reinforcement Learning. The problem is the training of a 3D humanoid model, that hangs below a bar, to swing up and maintain balance on the bar. To do so, the model has to be designed using a physics engine, the environment has to be created and the algorithms have to be implemented. In Reinforcement Learning, the agent interacts with the environment and receives rewards, based on its actions and states it reaches. The states in this problem are the humanoid model's joint positions and velocities and the actions are torques applied on the joints' actuators, both presented in the form of vectors. Rewards are manifested, when the environment is designed.
The agent has to be trained in an environment that follows emulated real-world physics laws and restrictions, such as gravitation pull and body moves that a normal human being can perform. For that the MuJoCo physics engine is chosen.
The environment is based on already existing OpenAi Gym environments and the algorithms used to try and solve the manufactured problem are chosen through a multitude of Reinforcement Learning algorithms and are Deep Deterministic Policy Gradient and Advantage Actor-Critic.

1.2 Related Work

When experimenting with OpenAi Gym, one of the environments to start from is the Cartpole-V1 environment [8]. In Cartpole-v1 (Figure 1.1) there is a pole attached on a cart that moves left or right on a track. In the environment the goal is to keep the pole standing upright, whilst applying force to the cart (-1 or 1) to move slightly left or right.

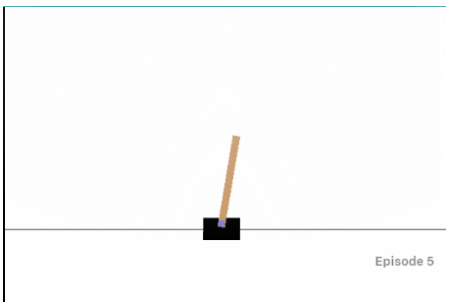


FIGURE 1.1: Cartpole-v1 Gym Environment

Another basic OpenAi Gym problem is the Pendulum-v0 [11] (Figure 1.2). In this environment the goal is to swing up and balance the pendulum by controlling its angular velocity with applied force.

FIGURE 1.2: Pendulum-v0 Gym Environment

Moving on to more complex environments, still in the 2D field, we come accross the Acrobot-v1 [6] (Figure 1.3), an environment like the Pendulum-v0, but with two links connected with an extra joint and consequently with more freedom of movement and more complexity.

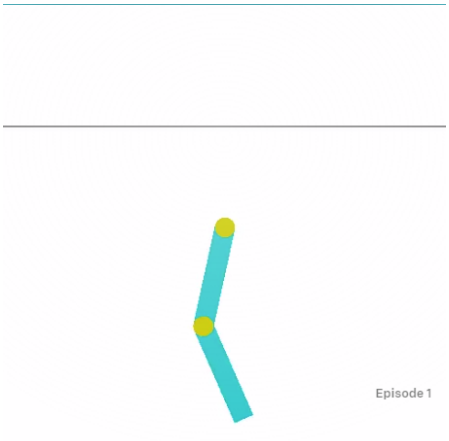


FIGURE 1.3: Acrobot-v1 Gym Environment

Looking through the 3D MuJoCo environments, we see InvertedDoublePendulum-v2 [10] (Figure 1.4), which combines the previous environments in the 3-Dimensional space.

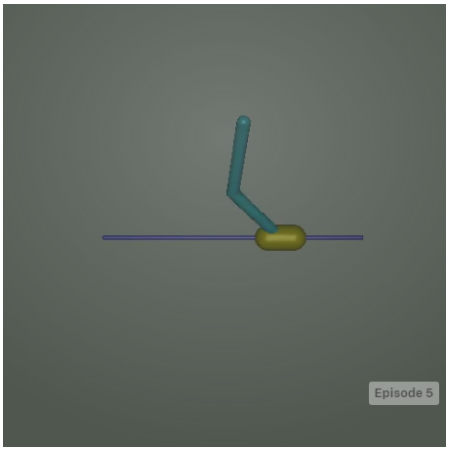


FIGURE 1.4: InvertedDoublePendulum-v2 Gym Environment

Another known MuJoCo environment is the Humanoid-v2 [9] (Figure 1.5), which is a humanoid robotic model with 19 joints, that tries to walk in the upright position without falling down by applying torques to 17 joints.

FIGURE 1.5: Humanoid-v2 Gym Environment

Combining the goal of the previous environments, the swing up and balance, with the high complexity of the humanoid environment we got the idea of the environment and task of this thesis.

Chapter 2

Background

2.1 MuJoCo

As stated on the MuJoCo website [**MuJoCo**] "MuJoCo stands for Multi-Joint dynamics with Contact. It is being developed by Emo Todorov for Roboti LLC. Initially it was used at the Movement Control Laboratory, University of Washington, and has now been adopted by a wide community of researchers and developers". MuJoCo is a physics engine used to render and simulate environments, it provides a number of gym environments and models in the form of .xml files. The physics engine itself is implemented in C++, but the open-ai gym environment used here makes use of the python wrappers provided by MuJoCo. The MuJoCo library generally requires a paid 1-year license, but there is also a student license which is free.

2.1.1 MuJoCo Models

The MuJoCo models are implemented in .xml files, which contain a number of bodies, geoms, joints, cameras, light, etc.

- **Body**–Multiple bodies are used to construct a model. These bodies are connected to each other in a tree-like topology, called the kinematic tree, meaning that to connect a body to another in order to create the model the bodies have to have a parent-child relationship inside that tree. The top-level (root) body is called the worldbody.
- **Geom**– Geoms are elements attached to each body and define its appearance and collision properties.
- **Joint**–Joint elements create the degrees of freedom between child and parent bodies. The absence of a joint in a body means that a body is rigidly attached to its parent. Cameras and lights define the rendering part of the models and scene brightness.

2.2 Reinforcement Learning

Reinforcement Learning (RL) is one of the subcategories of Machine Learning. It is a mix of supervised learning (SL) and unsupervised learning (UL). In contrast to SL being a static function (e.g. a classifier that gives you a certain result for a certain input), RL is more like a loop of actions chosen to achieve a certain goal with respect to time. The major characteristic of RL is that there is no target for each step, instead there is an end-goal that the agent is trying to achieve. That means that the RL agent has to figure out the optimal sequence of actions that would lead to the end goal and not the optimal action for each step. Basic Definitions of RL follow:

- **Agent** – The learning and acting part of a Reinforcement Learning problem, which tries to maximize the rewards it is given by the Environment. The Agent is the model which RL scientists try to design.
- **Environment** – The environment can be everything which isn't the Agent; everything the Agent can interact with, either directly or indirectly. The environment changes as the Agent performs actions.
- **State, Action, Reward** – A state can be described as the configuration of the environment at a certain point in time. E.g. in a game of chess environment the state is described by the position of each piece on the board. An action is a move that the agent can make. E.g. in the same example actions are all the possible moves of each pawn. The reward is a return, a number, that the agent gets after each step or episode. Rewards are used to improve agent training.
- **Policy** – The policy π is essentially the rule that the agent follows in order to take an action a at a state s, for example in a greedy policy the optimal action is the action with the highest value.
- **Off-Policy / On-Policy** – RL algorithms can be either Off-Policy or On-Policy, the difference between them is in the way that an algorithm estimates the optimal policy. While in the exploration part of training On-Policy algorithms will use the same policy that the algorithm behaves with to update the Q-Values, an Off-Policy algorithm will use a different one.
- **Q-Value** – Q-value is the expected discounted reward for an agent that follows a policy π and takes an action a based on that policy, at a state s given by :

$$Q^{\pi}(s,a) = R_s(a) + \gamma \sum_{s'} P_{ss'}[\pi(a)] V^{\pi}(s') \tag{2.1}$$

  with $R_s(a)$ being the reward value at state s after taking action a, γ the discount factor, $P_{ss'}[\pi(a)]$ being the probability of moving from state s to s' after taking action a following the policy π and $V^{\pi}(s')$ the value of state s'.
- **Markov Decision Process** -A MDP describes a reinforcement learning system. In a MDP, we describe the environment with the state-transition probability, which is the probability of arriving at a state s' at a time $t+1$ and getting the reward R given the state s at time t and taking the action a.

$$p(S_{t+1}, R_{t+1} | S_t, A_t) \tag{2.2}$$

  An MDP consists basically of an agent and an environment interacting with each other. The agent gets the current state from the environment and decides based on a policy what action to take. After the action is done the environment is updated based on that action and returns a new state and a reward. The agent then again reads the new state and decides an action and the same process repeats.
- **Policy Gradient** – Policy gradient algorithms are widely used in reinforcement learning problems with continuous action spaces. The basic idea is to represent the policy by a parametric probability distribution $\pi_{\theta}(a|s) = P[a|s;\theta]$ that stochastically selects action a in state s according to parameter vector θ. Policy gradient algorithms typically proceed by sampling this stochastic policy and adjusting the policy parameters in the direction of greater cumulative reward.

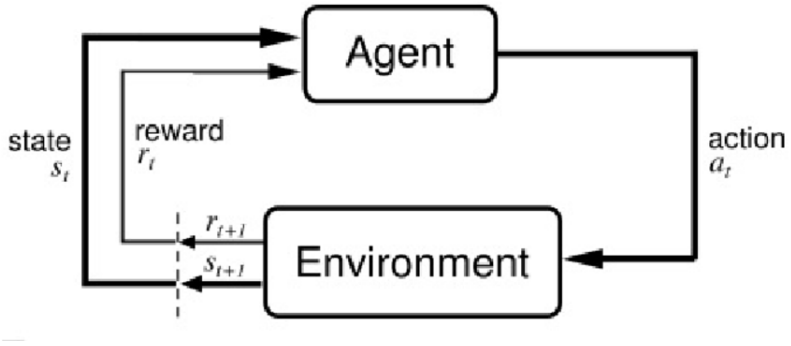


FIGURE 2.1: Reinforcement Learning Framework [14]

2.2.1 Q-Learning

Q-Learning [16] is a model-free RL algorithm. In Q-learning the agent given that the state is s evaluates the actions available by the immediate reward that the action yields after it is chosen and an estimate of the next state/states rewards with respect to the state-transition probability described previously. The Q-Value equation is :

$$Q(s,a) = R_s(a) + \gamma \max_{a'} Q(s',a') \tag{2.3}$$

with γ being the discount factor used to achieve convergence. By trying-exploring all the actions available in all states repeatedly the algorithm generates a table of size $S[s_1,...,s_n](statespace) \times A[a_1,...,a_m](actionspace)$. Based on that table the best actions are chosen based on long-term discounted reward. At the beginning of the training either the table values are given or set to 0 and some of the initial actions are chosen randomly in order to explore the action space and get a base for the Q-values. As the training proceeds the values are updated based on the following equation and they become more and more correct.

Q-Value update Equation :

$$Q'(s_t,a_t) = (1-\alpha)\cdot Q(s_t,a_t) + \alpha \cdot (r_t + \gamma \cdot \max_{a'} Q(s_{t+1},a')) \tag{2.4}$$

Overall in Q-Learning the agent is tasked with finding an optimal policy π, one that maximizes the total discounted expected reward. The action selection is based on maximizing the Q-Value :

$$a_t = arg\max_{a'} Q(s_t,a') \tag{2.5}$$

2.2.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) [3] are information processing or computation models used for computations, especially applied in machine learning, that are inspired from the way the human brain performs. A computer can be superior to a human brain in functions, such as data storage or complex math, but it can struggle in learning behaviours or in recognizing even the simplest patterns, in which a human brain excels at.

From the above observation, the ANNs were inspired. A brain has many neurons that send signals through it and process the data that it receives, in the same way ANNs have processing units that are simplified versions of the human neurons, called also neurons.

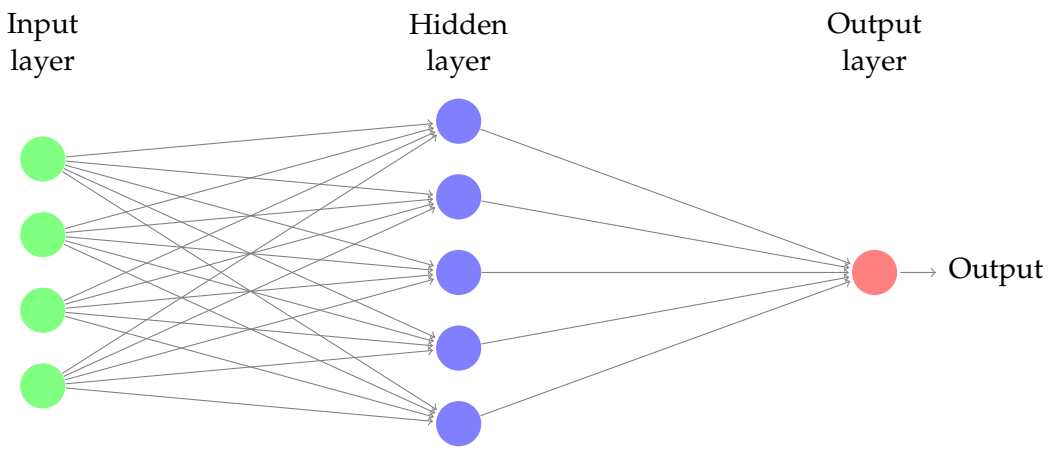


FIGURE 2.2: Simple Artificial Neural Network

The neurons in a NN do not have a random topology, each neuron can either be an input, an output or hidden, based on its placement. The simplest topology (Figure 2.2) consists of one input layer, that takes inputs from the outside world, computer or algorithm, an output node that outputs the data to the outside world, computer or algorithm (depending on the use of the ANN) and a hidden layer placed between these two layers, the hidden neurons take inputs from other neurons and passes it through to other ones. Each layer's neurons are connected to all the next layer's neurons, making the NN fully connected. The connections passing data from neuron to neuron have an important role in the ANN, they weight the data that is passed through. In most NNs the multiple inputs of a neuron are summed, a bias (b) can also be in the sum, and in less common implementations substracted.
All the above leads to the mathematical calculation that if the neuron's output is x_i and the weight is w_{ij} the next neuron's j input is

$$y_j = \sum_i x_i w_{ij} + b \tag{2.6}$$

Each neuron that outputs info can have an activation function f, making its output $x_j = f(y_j)$ Figure 2.3. Activation functions are a vital part of the NN training, when the data are complex, because they aid in mapping the output values to a desired range that is more manageable. Two of the most common activation functions are :

- **ReLU**

$$R(z) = max(0, z) \tag{2.7}$$

- **Tanh**

$$tanh(z) = \frac{e^{2z} - 1}{e^{2z} + 1} \tag{2.8}$$

Neuron connections can either connect each neuron to the next layer, making the ANN Feedforward or can have more complex connextions, to the previous layers making them Recurrent.
ANNs can be Multilayer perceptrons (MLPs). MLP is a category of feedforward ANN sometimes loosely connected to any feedforward ANN, sometimes strictly to refer to networks composed of multiple layers of perceptrons (with threshold activation).

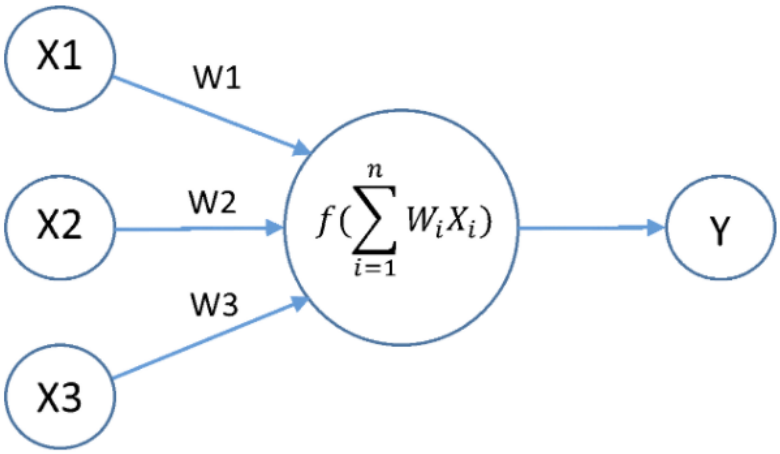


FIGURE 2.3: Mathematical Visualization

2.2.3 Deep Q-Learning

Although Q-learning yields good results for games with small state spaces, most RL problems nowadays have large state and action spaces, making Q-Learning not the ideal tool for solving them. The Q-Table that the algorithm makes use of, highly increases the time and space complexity of the algorithm for RL tasks with large states spaces. An accurate example are video games, in which the state at a certain time point is the configuration of pixels on the screen. This configuration of pixels even for a low resolution game, like atari games, makes of a very large state space. E.g. Atari-Breakout has a resolution of 210x160x3, even if it is converted to black and white there are still 210x160=33600 pixels making the state space and Q-table great in size thus rendering the Q-Learning method almost unusable, considering that the agent would require multiple weeks to be trained for a simple game. In [**Derele**] deep learning agent trained to play atari games using a Convolutional Neural Network (CNN) instead of Q-table with raw pixels as inputs and a value function estimating future rewards as output is introduced. The NNs used are referred to as Deep Q-Networks (DQNs). Two new terms are introduced here: the loss function and the experience replay. The Neural Network with weights θ, known as Q-Netrwork, is trained by minimising a sequence of loss functions $L_i(\theta_i)$. In supervised learning the loss function is defined as

$$(Target - Prediction)^2, \tag{2.9}$$

whereas in Q-learning there is no predifined target, but rather an estimate of a target depending on the model's prediction and the Loss Function looks as follows :

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim p(\cdot)}[(y_i - Q(s,a))^2], \tag{2.10}$$

where y_i is the estimated target

$$\mathbb{E}_{s'}[r + \gamma \max_{a'} Q(s',a')|s,a] \tag{2.11}$$

The experience replay mechanism makes use of the replay memory that stores the agent's past experiences of each time-step, random mini-batches of these experiences are used to perform Q-updates on the NN's weights. This technique smooths out the learning and helps avoid divergence in the parameters.

2.2.4 Policy Gradients

In [15] the argument that sole value function approach has limitations is made. Firstly it is oriented toward finding deterministic policies, whereas the optimal policy is often stochastic, selecting different actions with specific probabilities. Secondly, an arbitrarily small change in the estimated value of an action can cause it to be, or not be selected.
To improve these limitations, the policy gradient methods are introduced. Rather than approximating a value function and using that to compute a deterministic policy, we approximate a stochastic policy directly using an independent function approximator with its own parameters. In policy gradient methods, the reward function is calculated as :

$$J(\theta) = \sum_{s\in S} d^\pi(s) V^\pi(s) = \sum_{s\in S} d^\pi(s) \sum_{a\in A} \pi_\theta(a|s) Q^\pi(s,a) \tag{2.12}$$

where $d^{\pi}(s)$ is the stationary distribution of the corresponding Markov chain for π_θ (on-policy state distribution under π).
To maximize the returns using gradient ascent with the above expression is ploblematic, because it depends on both the action selection (directly determined by π_θ) and the stationary distribution of states following the target selection behavior (indirectly determined by π_θ). Given that the environment is generally unknown, it is difficult to estimate the effect on the state distribution by a policy update. And the policy gradient theorem to ease things is introduced :

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta \sum_{s\in\mathcal{S}} d^{\pi}(s) \sum_{a\in\mathcal{A}} Q^{\pi}(s,a)\pi_\theta(a|s) \\ &\propto \sum_{s\in\mathcal{S}} d^{\pi}(s) \sum_{a\in\mathcal{A}} Q^{\pi}(s,a)\nabla_\theta \pi_\theta(a|s) \end{aligned} \tag{2.13}$$

2.2.5 Deep Determenistic Policy Gradient

In Deep Q-Learning there is always talk about the value function Q(s,a). In DQN, the state and action that describe a Q-value are not both inputs of the NN, instead the state is an input and there are a number of outputs, one for each possible action at state s.
In some RL problems including the one in this thesis, the action space is continuous meaning that there are, if not infinite possible actions, a large number of them. Having infinite possible actions makes the use of DQNs impossible, because that would require NNs with an infinite number of outputs. As it is stated in [2], DQN cannot be straightforwardly applied to continuous domains, since it relies on finding the action that maximizes the action-value function, which in the continuous valued case requires an iterative optimization process at every step.
A new algorithm is introduced Deep Deterministic Policy Gradient (DDPG) which is based on the Deterministic Policy Gradient (DPG) [12] with some modifications inspired by DQN. DDPG makes use of the actor-critic principle, in which there are two basic models interacting with each other, the actor and the critic. The actor is a function approximator tasked with choosing an optimal action at a state s and the critic is another function approximator that evaluates the action the actor chose. In DDPG there is a determenistic policy $\mu(s)$ which is represented by a NN with input the state s that outputs an optimal action a (Actor).
The action is chosen based on the equation :

$$a_t = \mu(s_t|\theta^{\mu}) \tag{2.14}$$

The above equation is not sufficient enough to ensure proper exploration of the environment so a Gaussian Noise $\mathcal{N}$ is added :

$$a_t = \mu(s_t|\theta^{\mu}) + \mathcal{N}_t \tag{2.15}$$

The critic is also represented by a NN that takes as input the state s and action a which is the output of the policy network and outputs the Q-Value. Both the μ and Q networks have two instances, the main and the target network.
As in DQN, the DDPG algorithm makes use of experience replay buffers that store previous state-action transitions. The main actor is being updated at each step with the goal of maximizing the total expected rewards and the critic in order to minimize the loss function.
A new way of updating the target networks is used in DDPG, called "soft" target

updates. In "soft" target updates rather than copying over all the weights from the main network periodically, a fraction of the weights is copied over at each step as shown below :

$$\theta^{\mu\prime} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu\prime} \tag{2.16}$$

$$\theta^{Q\prime} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q\prime} \tag{2.17}$$

with $\tau << 1$ providing a slow rate of updates that ensures a great stability in learning.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ
Initialize target network Q' and actor μ' with weights $\theta^{Q\prime} \leftarrow \theta^Q$ and $\theta^{\mu\prime} \leftarrow \theta^\mu$
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 observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{R}$
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu\prime})|\theta^{Q\prime})$
    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 :

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

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

  end for
end for

2.2.6 Advantage Actor-Critic (A2C)

In [4] multiple asynchronous RL algorithms are described, one of them being Asynchronous Advantage Actor-Critic (A3C). As it is obvious from its name A3C is an actor critic algorithm, meaning that there are two NNs the actor or the policy network $\pi(a|s,\theta_\pi)$ and the critic or value network $V(s,\theta_v)$. The advantage A in A3C is the term used to measure the difference between the actual return at a state s and the state's value $V(s)$

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t) \tag{2.18}$$

where $Q(s_t, a_t)$ is the expected return for selecting an action a at a state s and $V(s_t)$ is the expected return from state s. The update performed on the policy network can be seen as $\nabla_{\theta'} \log \pi(a_t, s_t; \theta') A(s_t, a_t; \theta)$ and the value network is updated with the

loss function being the squared error between the actual return and the predicted value $L_v = (Q(s_t, a_t) - V(s, \theta_v))^2$.
To encourage exploration and avoid premature convergence to sub-optimal policies the entropy of the policy H is added to the policy loss function making the final policy update function :

$$\nabla_{\theta'} \log \pi(a_t, s_t; \theta') A(s_t, a_t; \theta) + \beta \nabla_{\theta'} H(\pi(s_t; \theta')) \tag{2.19}$$

with β being a regularization constant that controls the strength of the regularization term.
The word Asynchronous in the name of A3C comes from the framework proposed in [4], in this framework multiple actor-learners called workers each of them seeing its own version of the environment running on threads of the same CPU. The idea behind this setup is that each worker can have different policies and explore different parts of the environment, making it more efficient than a single agent exploring the environment. Each of the workers has its own policy and value networks and there is also a global target network that gets updated slowly.
The algorithm makes use of the n-step update method to update the networks meaning that n number of steps are taken in the environment to calculate the returns.
In an openai blog post ([7]) a variation of the A3C algorithm was introduced called Advantage Actor-Critic (A2C), taking out the asynchronous part of A3C. As stated in the post " AI researchers wondered whether the asynchronus element improved performance , or if it was just an implementation detail that allowed for faster training with a CPU-based implementation.", which encouraged them to experiment with a synchronous deterministic variation of A3C, which proved to yield equal or better results (on GPUs) than the original A3C. In the A2C implementation the workers wait for each other to finish their segment (n-steps of the environment) and then perform the synchronous update. One major difference with A3C is that since the updates are now synchronous there is no need for each worker to have its own Neural Network instead there is one global NN that each worker uses a copy of and the updates are performed over the average results of the workers.

2.3 OpenAI Gym

OpenAI Gym [1] is a toolkit for reinforcement learning research. It essentially provides a selection of environments on which RL algorithms can be applied.
It can be considered as a benchmark for RL researchers to test agents on. Different agents can be tested on the same environments that the OpenAI Gym provides and their results can be compared with its scoreboard setting. As stated in [1], the result sharing aspect of the OpenAI Gym is aimed towards sharing code and ideas for agents and environments, rather than a competion.
It contains a variety of environments in the categories listed below :

- Classic Control and toy text : small-scale tasks from the RL literature.
- Algorithmic : perform computations, such as adding multi-digit numbers and reversing sequences.
- Atari : a selection of classic Atari games for agents to learn to play.
- 2D and 3D robots : simulation of robots with control problems using the MuJoCo engine.

The environment created and used in this thesis falls in the final category of 3D robots on the MuJoCo engine. All available environments can be found here [5].

Chapter 3

Experimental Setup

3.1 Overview

In this chapter a description of the procedure that was followed to design and implement the model, setup the training environments and implement the training algorithms and the libraries used in each part.

In Section 3.2 the MuJoCo humanoid model design and details are described and a description of the OpenAi gym environment. In Section 3.3.1 the DDPG algorithm implementation and in Section 3.3.2 the baselines library and the A2C implementation are described.

3.2 Environment Setup

3.2.1 MuJoCo Model

For the creation of a MuJoCo environment a XML file is required. This XML file is the model that will be simulated.

One of the most well-known OpenAi gym MuJoCo environments is the 'Humanoid-v2' which is a 3d humanoid robot designed in order for the agent to learn how to walk and keep walking without falling, the 'Humanoid-v2' can be seen in Figure 3.1.

Part of the XML file code for the humanoid can be seen in Figure 3.2. This part is the code that designs and specifies the arms hands and their joints.

The environment used in the experiments of this thesis is a modified version of the environment described above. The problem here is different to the initial one, that one was for the humanoid to learn how to walk. Now the goal is for the humanoid, that starts in the initial position of hanging on a bar, that is shown in the simulate tool that MuJoCo provides in Figure 3.3, to learn via training to swing up and eventually balance itself on that bar, similar to the well known CartPole and Pendulum problems, but in a significantly more complex environment.

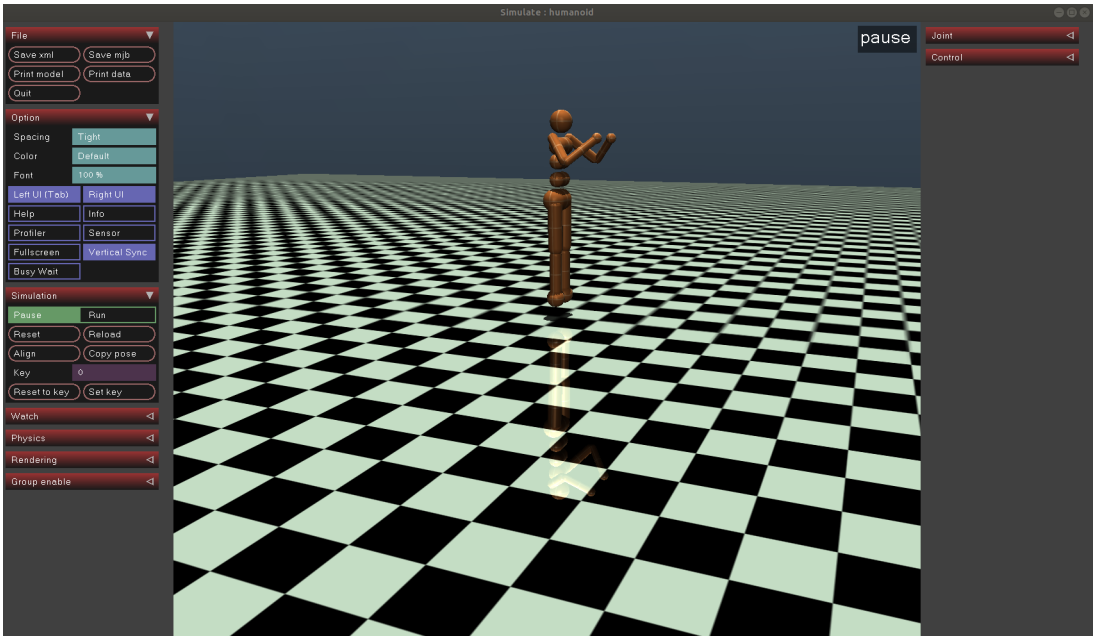


FIGURE 3.1: Humanoid-v2 initial state.

```
<body name="right_upper_arm" pos="0 -0.17 0.06">
    <joint armature="0.0068" axis="2 1 1" name="right_shoulder1" pos="0 0 0" range="-85 60" stiffness="1" type="hinge"/>
    <joint armature="0.0051" axis="0 -1 1" name="right_shoulder2" pos="0 0 0" range="-85 60" stiffness="1" type="hinge"/>
    <geom fromto="0 0 0 .16 -.16 -.16" name="right_uarm1" size="0.04 0.16" type="capsule"/>
    <body name="right_lower_arm" pos=".18 -.18 -.18">
        <joint armature="0.0028" axis="0 -1 1" name="right_elbow" pos="0 0 0" range="-90 50" stiffness="0" type="hinge"/>
        <geom fromto="0.01 0.01 0.01 .17 .17 .17" name="right_larm" size="0.031" type="capsule"/>
        <geom name="right_hand" pos=".18 .18 .18" size="0.04" type="sphere"/>
        <camera pos="0 0 0"/>
    </body>
</body>
<body name="left_upper_arm" pos="0 0.17 0.06">
    <joint armature="0.0068" axis="2 -1 1" name="left_shoulder1" pos="0 0 0" range="-60 85" stiffness="1" type="hinge"/>
    <joint armature="0.0051" axis="0 1 1" name="left_shoulder2" pos="0 0 0" range="-60 85" stiffness="1" type="hinge"/>
    <geom fromto="0 0 0 .16 .16 -.16" name="left_uarm1" size="0.04 0.16" type="capsule"/>
    <body name="left_lower_arm" pos=".18 .18 -.18">
        <joint armature="0.0028" axis="0 -1 -1" name="left_elbow" pos="0 0 0" range="-90 50" stiffness="0" type="hinge"/>
        <geom fromto="0.01 -0.01 0.01 .17 -.17 .17" name="left_larm" size="0.031" type="capsule"/>
        <geom name="left_hand" pos=".18 -.18 .18" size="0.04" type="sphere"/>
    </body>
</body>
```

FIGURE 3.2: Body elements for humanoid hands from XML file.

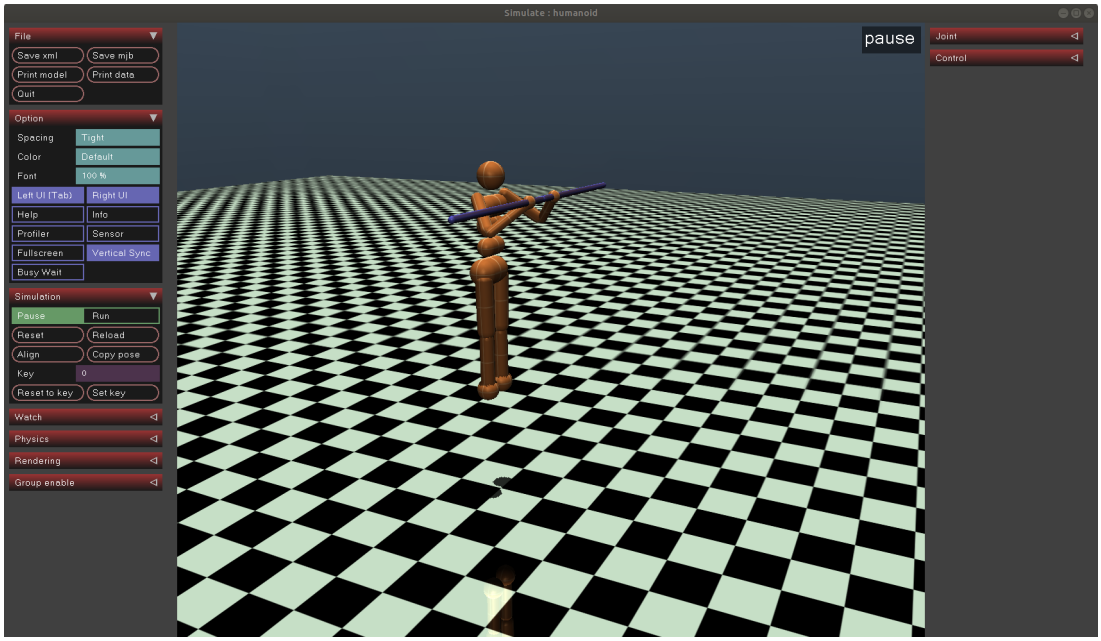


FIGURE 3.3: Humanoid Bar initial state.

Firstly, the bar was added as a child to the worldbody, as a body placed at a fixed position in the plane. To specify the position of a body the x, y, z coordinates have to be set, for the bar the x, y, z values are $0, 0, 2.06$. We define the z coordinate set to 2.06, as $HEIGHT$.

To define how the bar looks, a geom element has to be created. The bar consists of 3 geoms, the first one is for the bar itself and it is a capsule type geom as most of the geoms on the humanoid model, the other two geom type elements are two capsules at which the humanoid's hands are attached to, as described below. In the Humanoid-v2 walking problem the humanoids hands do not have any particular use, thus only being geoms. But in this balancing problem the hands are a main component, as they are the part that attaches the humanoid to the bar. In order to be able to do that in MuJoCo, they were made into body elements with the geoms as their child.

In order to attach two bodies in MuJoCo, an equality constraint has to be established. Equality constraints essentially keep a distance at an axis between two geoms constant during a simulation. Equality constraints were set between each hand and the bar, but that was not enough as the humanoid was sliding sideways on the bar. To solve that, the two geoms mentioned above were added at the spots on the bar that the hands should be on and one more equality constraint was used to connect each of the hands at the desired spot and keep the humanoid from sliding sideways.

In MuJoCo physics the bodies can move by having joints that provide a freedom of movement.
The joints have 'motors' and 'gears' that in a free translation describe the bodies strength by defining how sharp or delicate the moves can be. As the problem of swinging up and balancing the humanoid on the bar would require more torque on the upper body of the humanoid compared to the walking problem, the gear value of both the left and right shoulders and elbows was increased.

To set the optimal values for the problems many tests were done with different gear combinations with different outcomes, e.g. for some the humanoid lacked the ability to swing itself above the level of the bar due to the gear setting being too low, or on the other extreme, when setting the gears to a higher than needed setting, the moves would be too sharp and the humanoid would arrive to a position curled up around the bar and not moving.

3.2.2 OpenAi Gym

An OpenAi gym 3D robot MuJoCo environment has a python file with 3 basic functions :

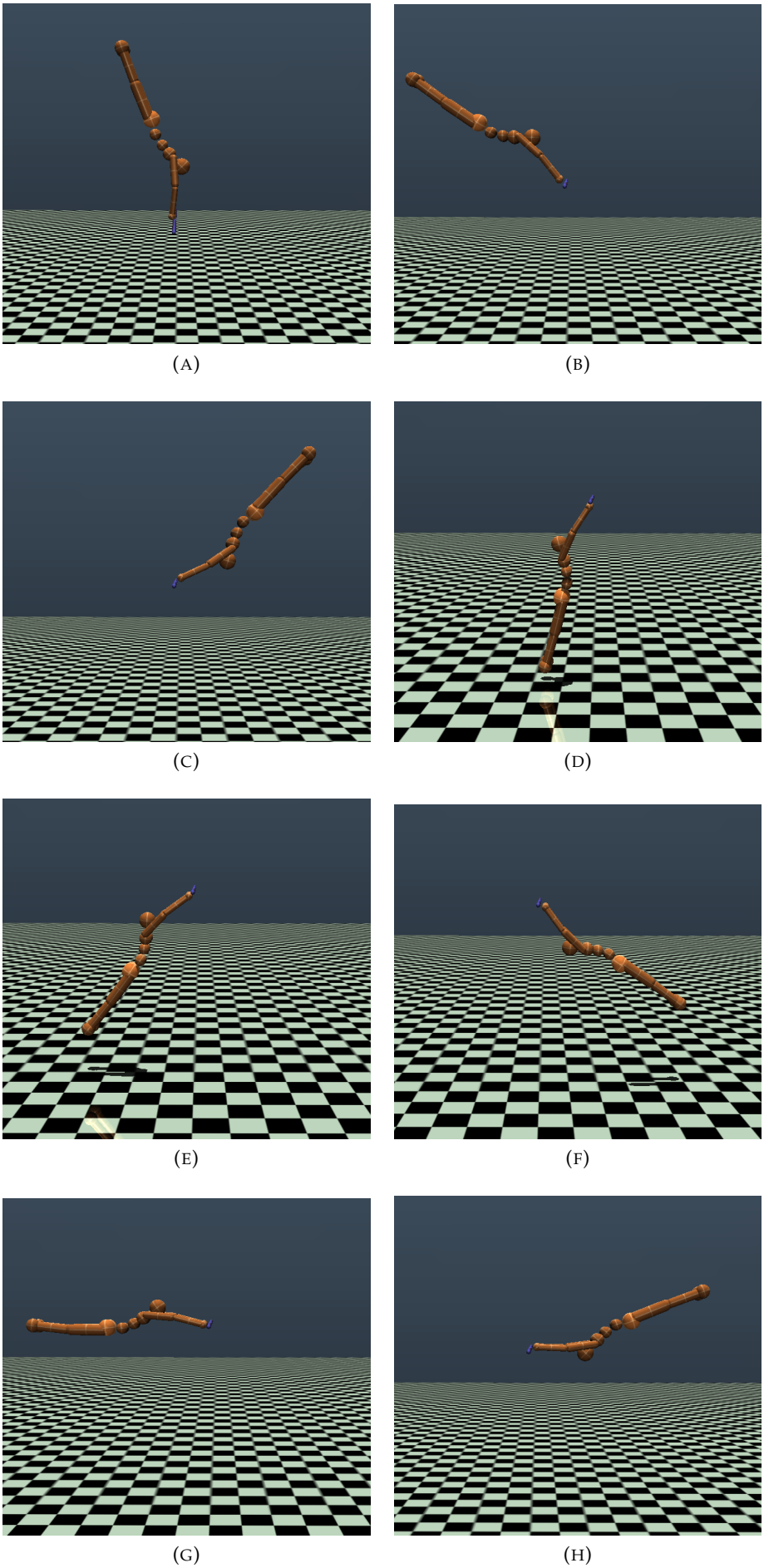
- **init** : in this function the XML file to load in the environment is defined.
- **step** : runs one timestep in the environment by taking an action as input, performing that action in the environment and returning the observation, reward, done from the new state that the action lead to.
- **reset** : resets the model to its initial state, by resetting the joints degrees of freedom. In MuJoCo that is done by setting values to qpos and qvel, the joints position and velocity, because the states in MuJoCo environments are described by the qpos and qvel vectors.

In the step function, the reward function is presented. In the experiment of this thesis, the reward is calculated as the sum of the position of each of the humanoid's bodies on the vertical axis (y) minus 2.06 which is the position the bar is placed on

the y axis, referred to as $HEIGHT$ in Section 3.2.1. This means that as the humanoid sits under the bar the rewards will be negative and as some of its components get above the bar the rewards will get closer to positive values and be at its max when the humanoid is doing a handstand on the bar.

$$R = \sum_{n=1}^{15} (xpos[n, 2] - HEIGHT) \tag{3.1}$$

In the reset function the model is reset to its initial position as shown in Figure 3.3. During the training experiments, it was obvious that the agent did not have enough knowledge of the upper part of the environment, essentially the positions where positive rewards would be awarded, and the humanoid would not swing up to a sufficient point.
To fix the lack of knowledge for the environment some randomness for exploration was introduced. At each new episode the humanoid would spawn at one of eight random poses around the bar shown in Figure 3.4 so that eventually full knowledge of the environment would be acquired.
To do this the simulate tool, provided by MuJoCo was used to set the humanoid at the desired poses. Using the simulate tool, it is possible to apply forces on the humanoid model and see the position and velocity values for the humanoid's bodies and joints. Enough force was applied on the humanoid for it to spin arround the bar, the simulation was paused at the desired poses. For each of those poses, the joint positions were copied and saved in the humanoid.py file. Each time the reset function is called one of the poses is randomly chosen using the random.choice() python function, and the returned pose is passed with the set_state() function of the MuJoCo environment.



(A) (B) (C) (D) (E) (F) (G) (H)

FIGURE 3.4: Possible initial positions

3.3 Algorithms

3.3.1 Deep Deterministic Policy Gradient

For the DDPG algorithm to be implemented in code as explained in Section 2.2.5 we need Neural Networks the Q and the μ and their respective target networks. To create those NNs, the TensorFlow library was used.
Both the Q and μ NNs have the same architecture. This means that they have the same size of hidden layers, in this experiment 300 neurons and one hidden layer. As of the activation function, both NNs use the same one, in this instance ReLU for the hidden layers, with the difference that, since the Q network has as output real numbers, future rewards, it needs no output activation function, whereas the output activation function for the μ NN is tanh.
The μ NN takes as input the states and outputs a number between [-1,+1] because the activation function (tanh) outputs a value between -1 and +1 which is matched to an action by multiplying with the max action value of the environment. For the Q NN the inputs are the action and state concatenated with the action being the output of the μ NN.
The replay buffer is implemented with numpy arrays, one for each of the following : state, action, reward, next state, and done (s, a, r, s', d). To use the buffer samples in the learning process the desired number of samples are randomly picked from the buffer, which essentially is the arrays mentioned here. For the DDPG algorithm, we have a learning rate of 0.001, the gamma is 0.99 and the decay factor is 0.995. The replay buffer has a size of 1000 samples and the random batch of samples used is 100.
At the beginning of the training procedure, for a certain amount of steps, the agent does not use its policy. This means that a random number of steps are taken to encourage better exploration and for the replay buffer to be filled, this number of steps is set to 100000. In the experiments presented in this thesis, the episode size was set to 1000 and 500 steps, for a multitude of total training episodes. The test episodes are recorded in video using the Monitor function of the gym wrappers.
The implementation does not test the agent as soon as the training is finished, as would be usual. Instead the agent is tested periodically, in this implementation every 25 training episodes, 5 testing episodes are done. This proved to be really helpful in evaluating the progress the agent is making with the training and especially with the recorded videos, so that it is easier to translate the values of the episode rewards into the actual movement of the humanoid and observing the patterns that are being developed.
Finally, both training and testing results are saved using the numpy library's savez function and the learning curves are generated using plot from matplotlib.

3.3.2 Advantage Actor-Critic

The Baselines library is a python library that provides a multitude of ready implemented Reinforcement Learning algorithms (A2C, ACER, ACKTR, DDPG, DQN, GAIL, HER, PPO1, PPO2, SAC, TD3, TRPO).
The Stable Baselines library provides an improved set of the implementations of the Baselines algorithms. To use this library it had to be installed in the python3 directory. To train the agent a python file has to be created in which the desired algorithm is imported and the environment the agent will be trained on is defined. The model is defined as the algorithm in this instance being A2C, which takes as arguments the

environment and the policy, which in most cases is a MultiLayer Perceptron (MLP). The learn function is called for the desired total timeteps over all the parallel environments. The trained model can be saved and loaded with the save and load functions. After all the training steps are done, the agent can be tested either in a timestep limited episode setup or for a sequence of timesteps without reseting the environment, in this experiment, for testing, 2000 episodes of 500 timesteps were ran. The results are collected and plotted in the same manner as in DDPG (with numpy and matplotlib), so that they can be comparable.

Chapter 4

Experiments and Results

4.1 Humanoid Bar Balance without random initial position

For the first experiments that were performed on the environment, the humanoid model was created and the training begun without any other modifications. And the initial position of each episode is the same, the one in Figure 3.3.

4.1.1 DDPG

For the first experiments the DDPG algorithm was used for training the model for an episode length of 1000 steps and a total of 40000 episodes.
The reward function for the results presented below is :

$$R = MassCenter - HEIGHT \tag{4.1}$$

with the MassCenter being the center of mass of the humanoid with regard to the y axis and HEIGHT the height in the plane that the bar is placed on. This gives us as the reward the distance between the humanoid and the bar, with it having positive values, when the humanoid is above the bar, and negative values, when it sits below it.

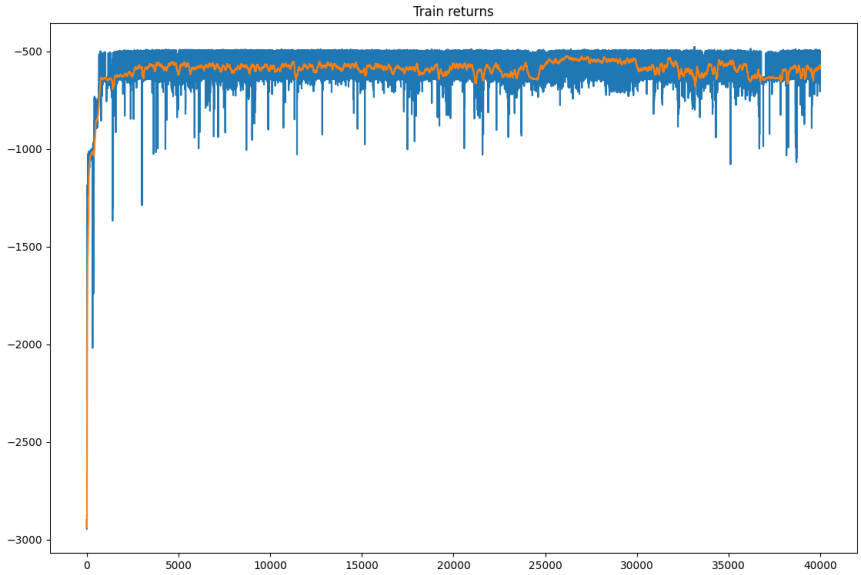


FIGURE 4.1: Training results

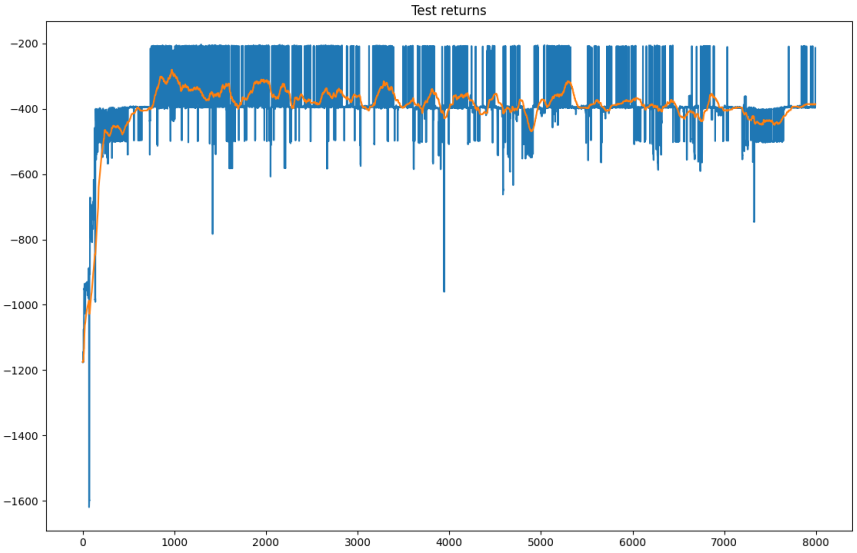


FIGURE 4.2: Testing results

From the above training (Figure 4.1) and testing (Figure 4.2) results we can observe that the minimum episode reward is close to -3000 making an average step reward of -3. Considering that at the first episodes the humanoid seats hanging below the bar this is the lowest reward achievable. From this, we can realize that the optimal reward would be 3000. If that reward is achieved it would mean that the humanoid balances on the bar for the whole episode.
In the training results we see that the agent reaches a maximum reward of -500 near the 1000 episode mark and the algorithm converges to that reward for the episodes to come. We can easily conclude that the maximum reward is far from optimal and there is a big difference between the theoretical max reward (3000) and the experimental one (-500). The best pose recorded for the humanoid in this training experiment is shown in Figure 4.3.

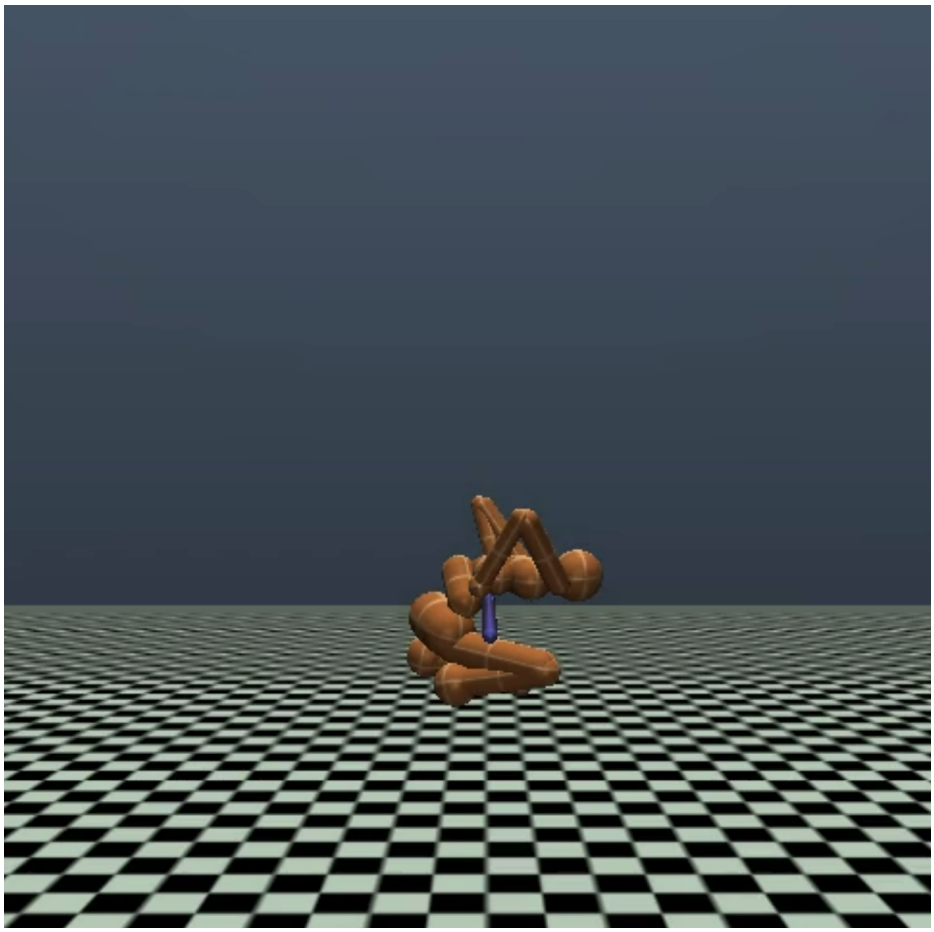


FIGURE 4.3: Humanoid pose

4.1.2 A2C

For the A2C algorithm, the parallel environments are set to eight, the same episode length of 1000 is selected and a total of 85000 episodes for each environment, making a total of 680000 episodes. The reward function is the same as the DDPG algorithm Equation 4.1.
The A2C algorithm seemed to work better (Figure 4.4) than the DDPG used above, reaching its maximum reward value earlier in the training procedure. The max reward (17) is the average over the eight environments and it is reached early in the simulation. Also it is observed that average rewards of -500 are systematically reached in the proceeding episodes, but the algorithm converges to lower rewards averaging around -1000.
The A2C algorithm also seems to be faster the DDPG implementation used in these experiments, considering that 8 virtual environments are being simulated in parallel, this seems to be due to the fact the A2C algorithm is loaded from the Baselines library, and being a high level python code, while DDPG is created from scratch.

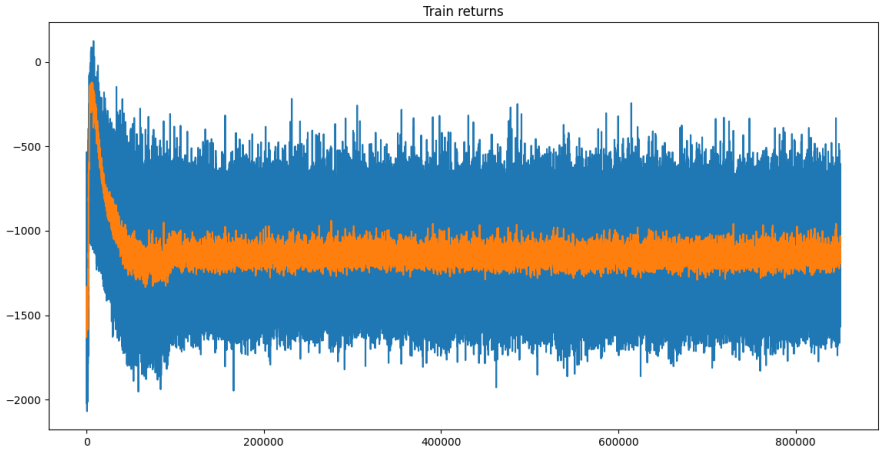


FIGURE 4.4: Training results

Although the first experiments show a stable training procedure and the humanoid seems to be making moves towards reaching higher positions in the plane, it is also obvious that the current experimental set up is sub-optimal and further changes and optimizations have to be made to aid the algorithms in training the agent to perform the correct task.
The main issue seems to be that the agent does not have enough information on the upper part of the plane as it is almost impossible for it to get consistently positive rewards for the humanoid being above the bar.
The solution to the problem is the randomization of the initial episode position of the humanoid around different angles on the bar, introduced and described in Section 3.2.2. The initial poses also produce more momentum aiding in the increased velocity required for the humanoid to swing up. Also the mass of the humanoid is reduced in order to make it easier for it to pull its weight upwards.

4.2 Humanoid Bar Balance with random initial position

In this section the results of the second version of the environment set up are presented for the two algorithms that were examined in the previous section.

4.2.1 DDPG

The episode length was reduced to 500 episodes and the total episodes for training are 10000 for the results presented below.
For the first part of the experiments with the randomness introduced in the initial positions, the reward function remained unchanged (Equation 4.1).
$R = MassCenter - HEIGHT$

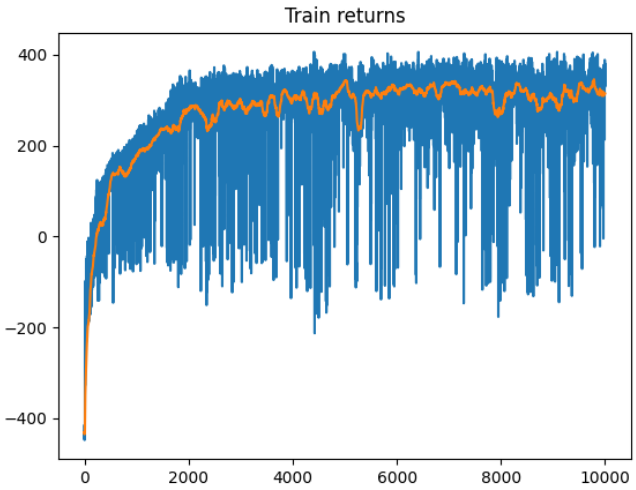


FIGURE 4.5: First Training results with random initial position

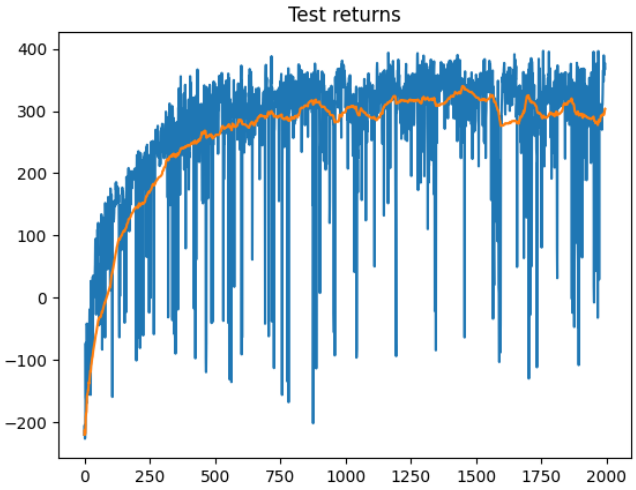


FIGURE 4.6: First Testing results with random initial position

From Figure 4.5 and Figure 4.6 on first sight it is easy to conclude that the training yielded much better results than the previous attempts. The minimum reward is near -400 for an episode with a total of 500 step and the maximum achieved is near +400.
To compare these results with the ones in the previous section we have to take into consideration that the episode size is reduced to half and that the mass of the humanoid is also reduced, making the gravitation pull less thus making the humanoid hang higher above the ground than previously without any training and consequently the minimum reward much less than before.
It is obvious that the random initial position provides much more knowledge of the environment to the agent as it achieves positive rewards that remain consistent throughout the training process. We also see that that when testing the agent the minimum rewards are lower than the training, which is normal considering that at the first testing episode the agent has already been trained for 25 episodes, with random initial position, which means that some of those episodes will begin at the best possible position for maximum reward (subfigure 3 in Figure 3.4).

For this version of the environment as the mass was changed, we opted not to use the mass center function for the reward equation, so the rewards used are the ones described in Section 3.2.2:

$$R = \sum_{n=1}^{15} (xpos[n,2] - HEIGHT) \tag{4.2}$$

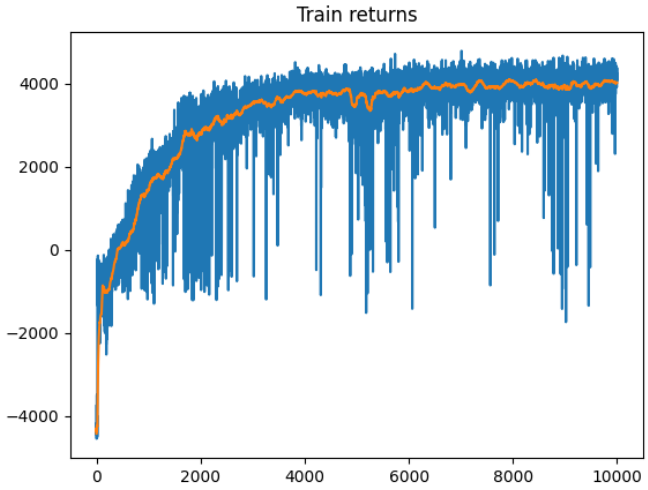


FIGURE 4.7: First Training results with random initial position and xpos rewards

From the results presented in Figure 4.7 and Figure 4.8 for the new environment version, the rewards in training vary between -4500 and 4500. The max reward of 4500 in 500 steps of the episode results in an average step reward of 9. Considering that the maximum step reward that was calculated during the simulation is near 13,4 and is acquired when the humanoid is holding its weight upright on the bar and the body is at full extension making the xpos[i,2] of each body part reaching its maximum value, the average of 9 is quite satisfactory for the problem examined here. As observed when rendering the agent, the humanoid's policy is correct in

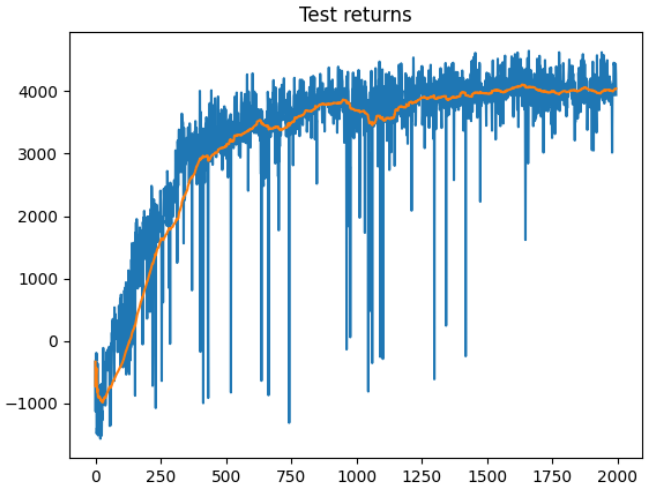


FIGURE 4.8: First Testing results with random initial position and xpos rewards

the meaning that the humanoid is constantly trying to achieve higher positions and swings up on the bar, whilst when achieving the swing up trying to balance on that position without falling below the bar for the most time possible. Due to the design of the parts of the humanoid that make contact with the bar, they are sphere type objects, and the bar itself being a cylinder there is very little contact area, for the humanoid to balance there indefinitely, thus making the humanoid to fall below the bar some steps during the simulation. As the reward become negative when below the bar, it is observed that the time that the humanoid spends below the bar is also tried to be minimized by the agent, and it is attempted to swing back up as soon as possible. Since these results proved to be near the optimal of what we were looking for, we went ahead and trained the agent for more episodes to see if there would be any improvement or if no further learning could be applied.

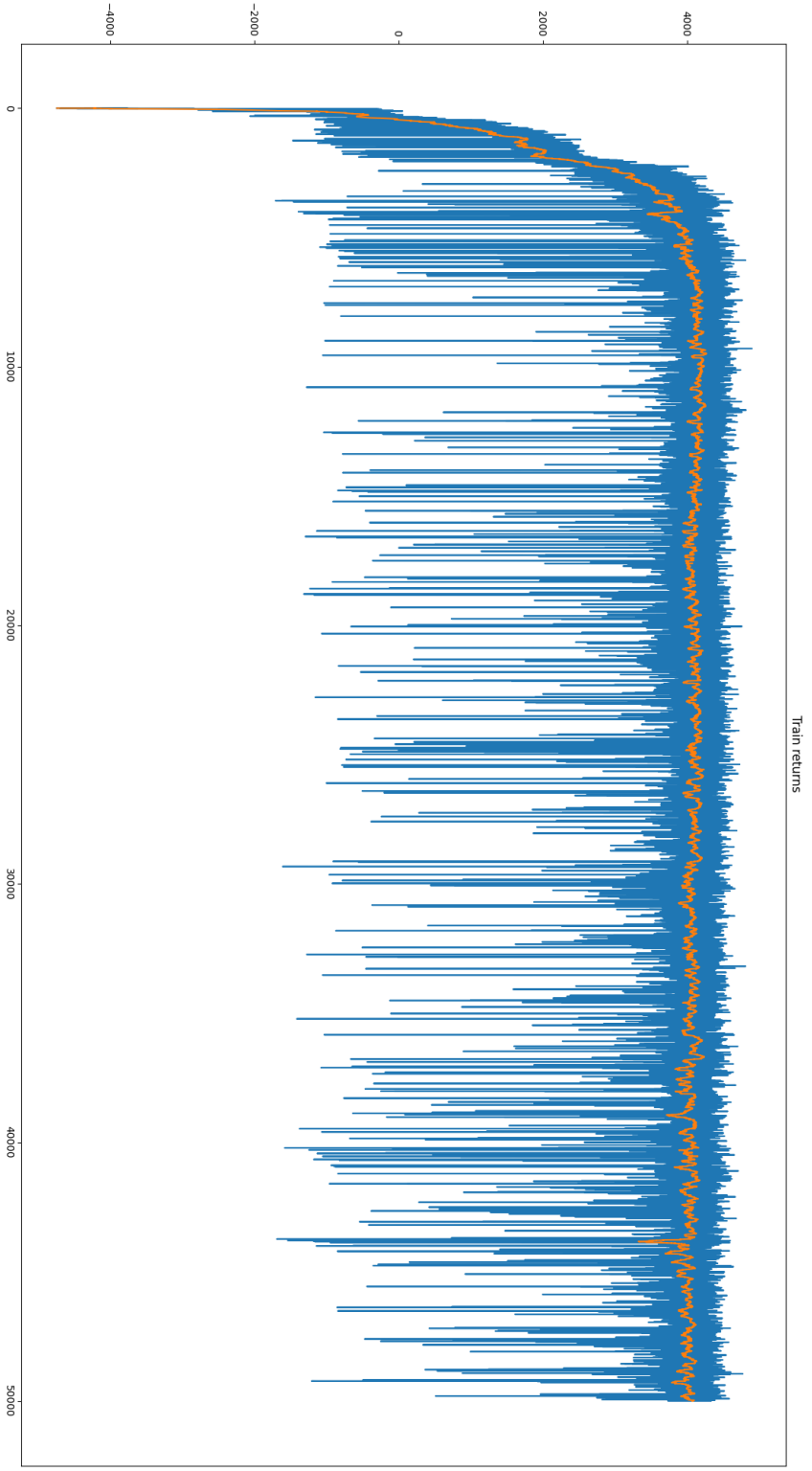


FIGURE 4.9: Training results for 50000 episodes

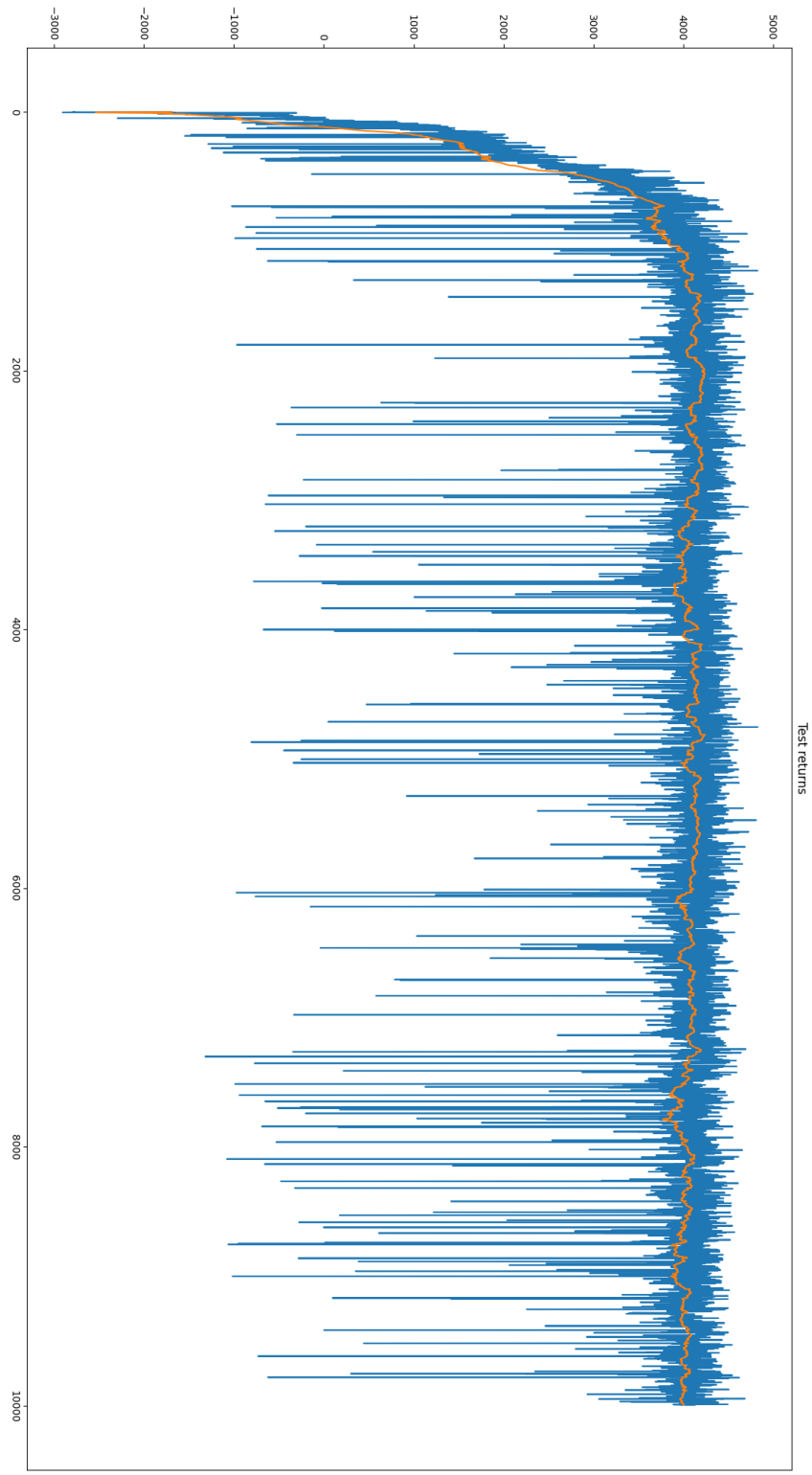


FIGURE 4.10: Testing results for 50000 episodes

From the above training Figure 4.9 and testing Figure 4.10 curves we can see that the maximum episode returns are achieved considerably early on the simulations, near the 5000 episode mark and the algorithm converges to that value, while not achieving a larger one for the rest of the episodes. The max return achieved is 4500, the same as before consequently with the same average step reward of 9. The behaviour is also similar to the one described above (in the 10000 episode simulation). One more interesting result deduced from the above curves is that the fluctuation in the episode rewards is normal for most parts averaging around 4000, but there are some spikes with negative or near 0 rewards, that produces an unwanted increase in variance. What is considered to be normal fluctuation is that of the rewards around 4000. To see if this is normal in MuJoCo environments, and especially "Humanoid-v2" which would be the closest environment to compare this thesis to, from online research the curve in Figure 4.11 for an agent training on the humanoid environment using the PPO algorithm was found in [13].

Comparing the two curves we see that the spikes described previously are not

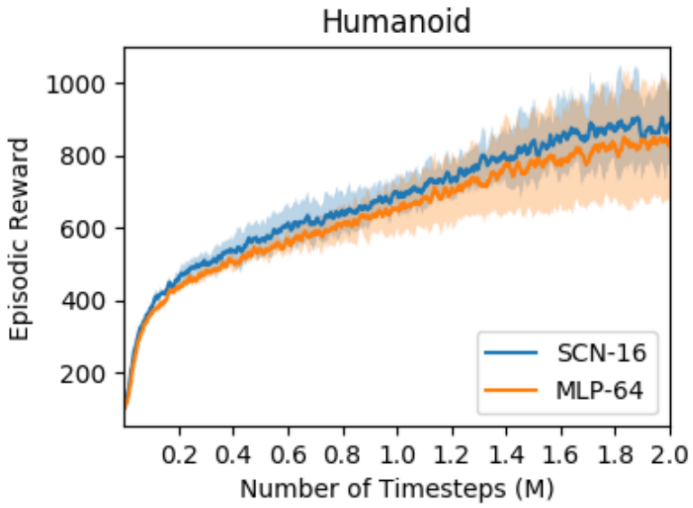


FIGURE 4.11: PPO trained Humanoid-v2 rewards

present in the "Humanoid-v2" training curve. By reviewing the testing footage it was deduced that the spikes are caused, because at some episodes the humanoid ends up tangled around the bar, rendering it unmovable, thus making the step rewards static at negative values, also making the total episode reward negative.

In Figure 3.3 we can see that there is space between the humanoid's torso parts. This gap is there also in the original humanoid.xml and it is believed that it serves the purpose of providing certain elasticity to the body's movement.

These gaps are the ones that cause the humanoid to be tangled with the bar, as they are large enough for the bar to fit between them and restrict the humanoid's movement freedom (Figure 4.12).

To improve the situation and produce learning curves without this many negative spikes, the spaces between torso parts were reduced, as it can easily be observed by comparing the two humanoid instances of Figure 4.13, which made the body more rigid and reduced the episode cases described above.

The new curves produced are shown in Figure 4.14 and Figure 4.15.

FIGURE 4.12: Humanoid not able to move

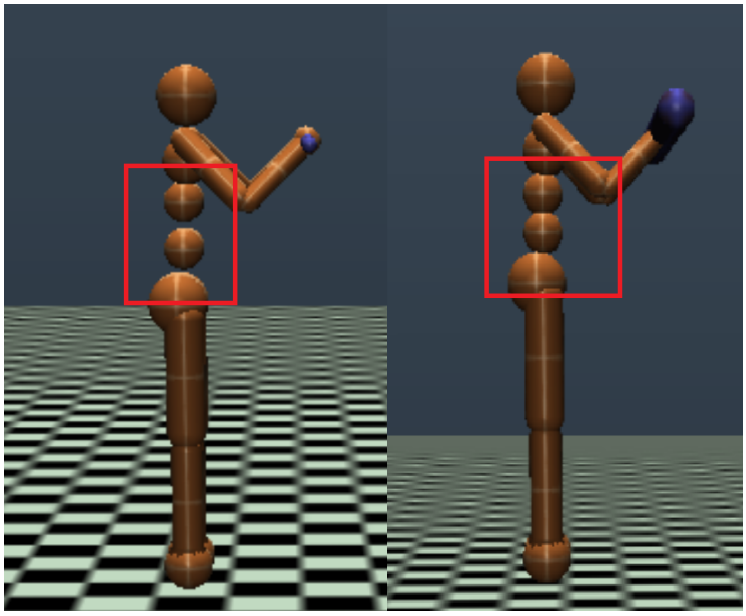


FIGURE 4.13: Initial and improved humanoid

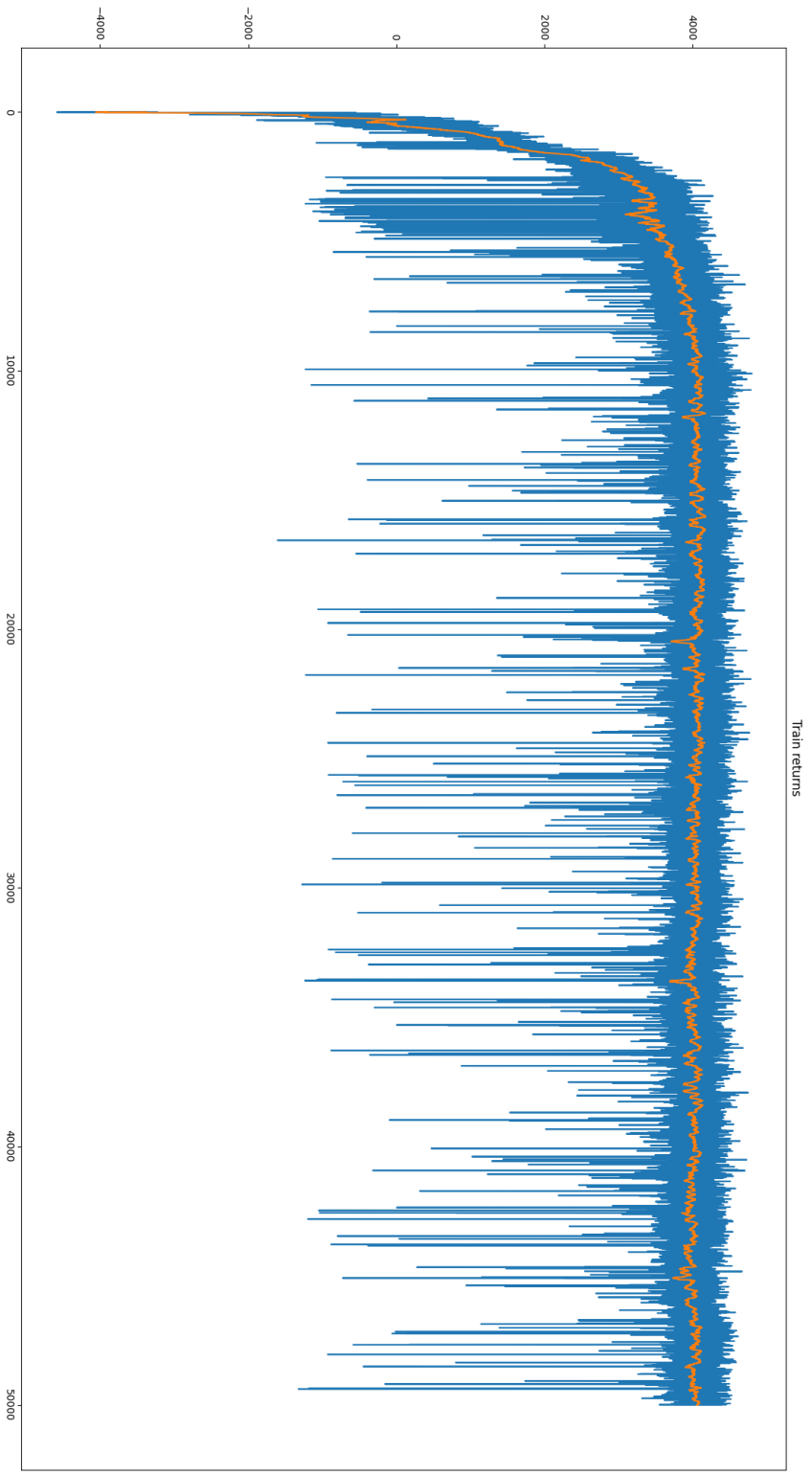


FIGURE 4.14: Improved Training results for 50000 episodes

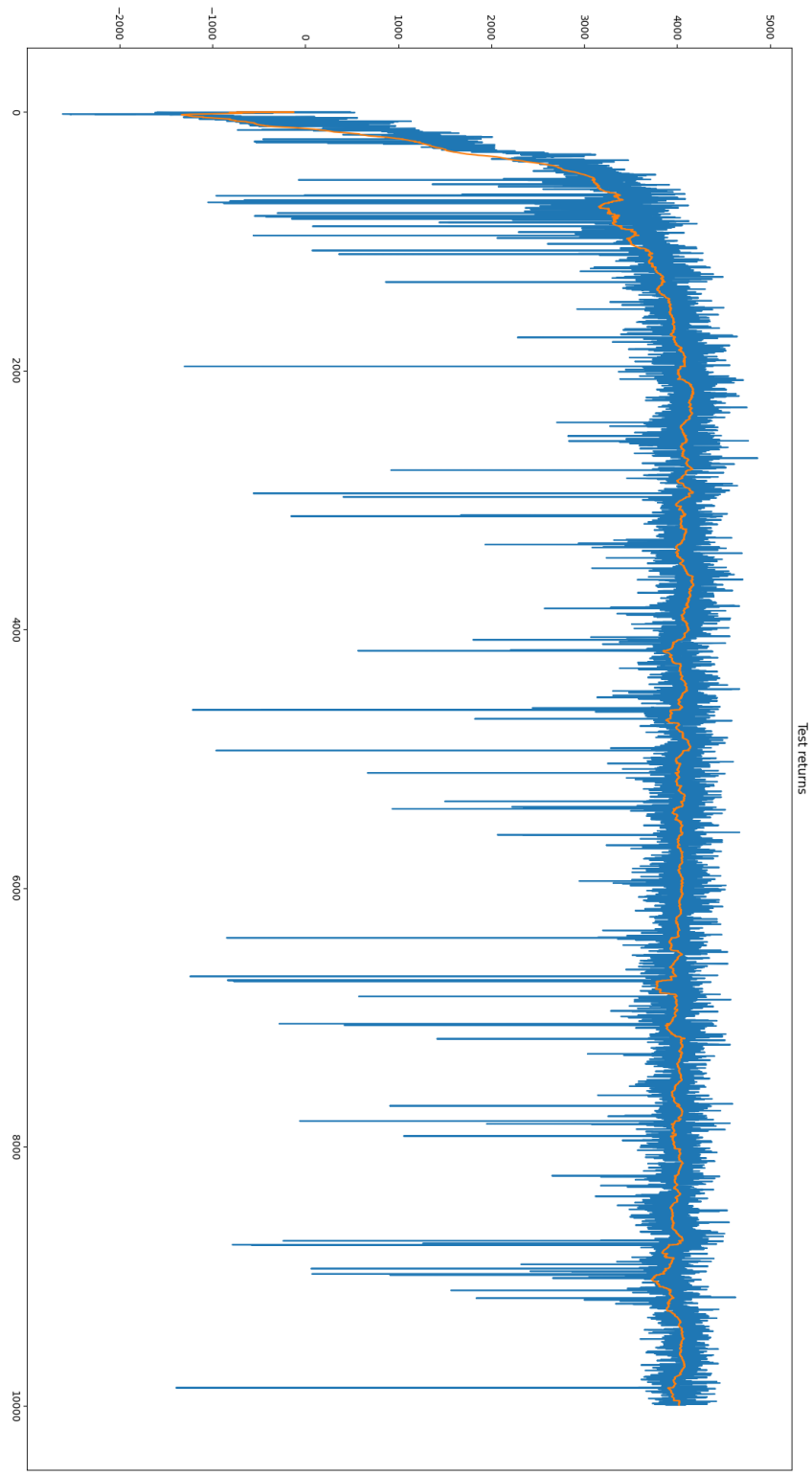
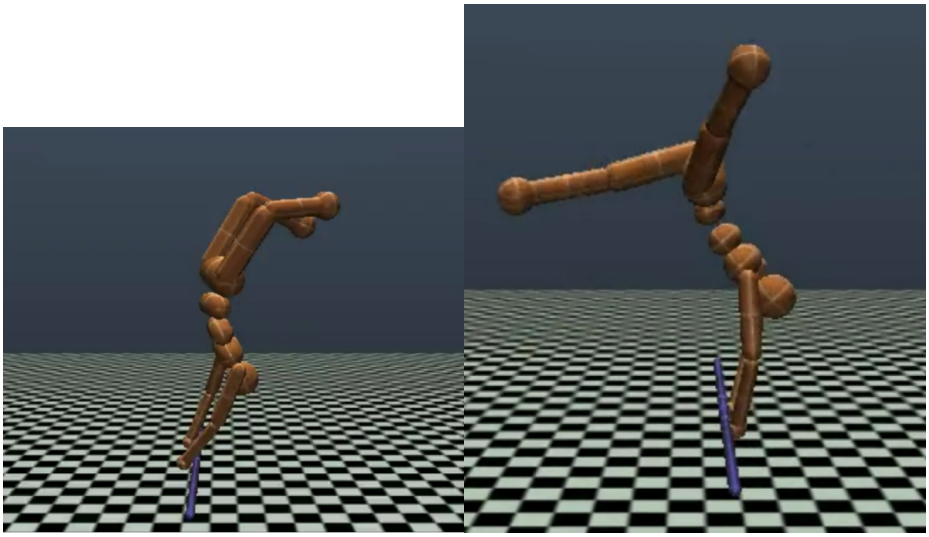
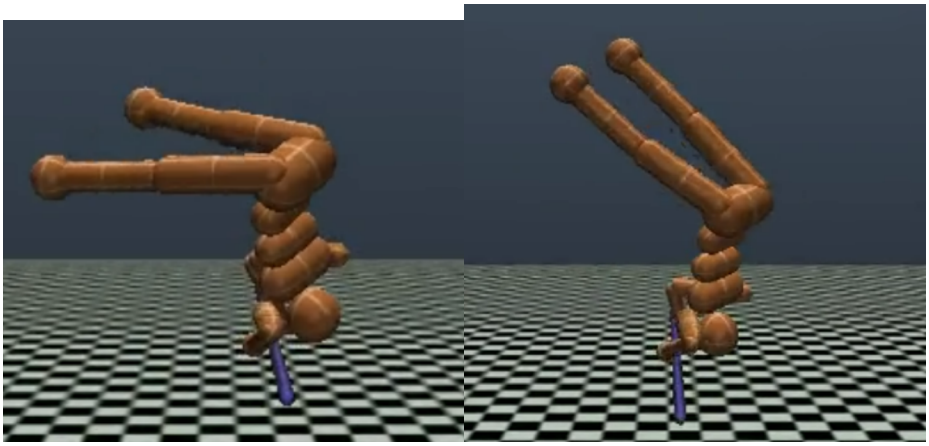


FIGURE 4.15: Improved Testing results for 50000 episodes



(A) (B)

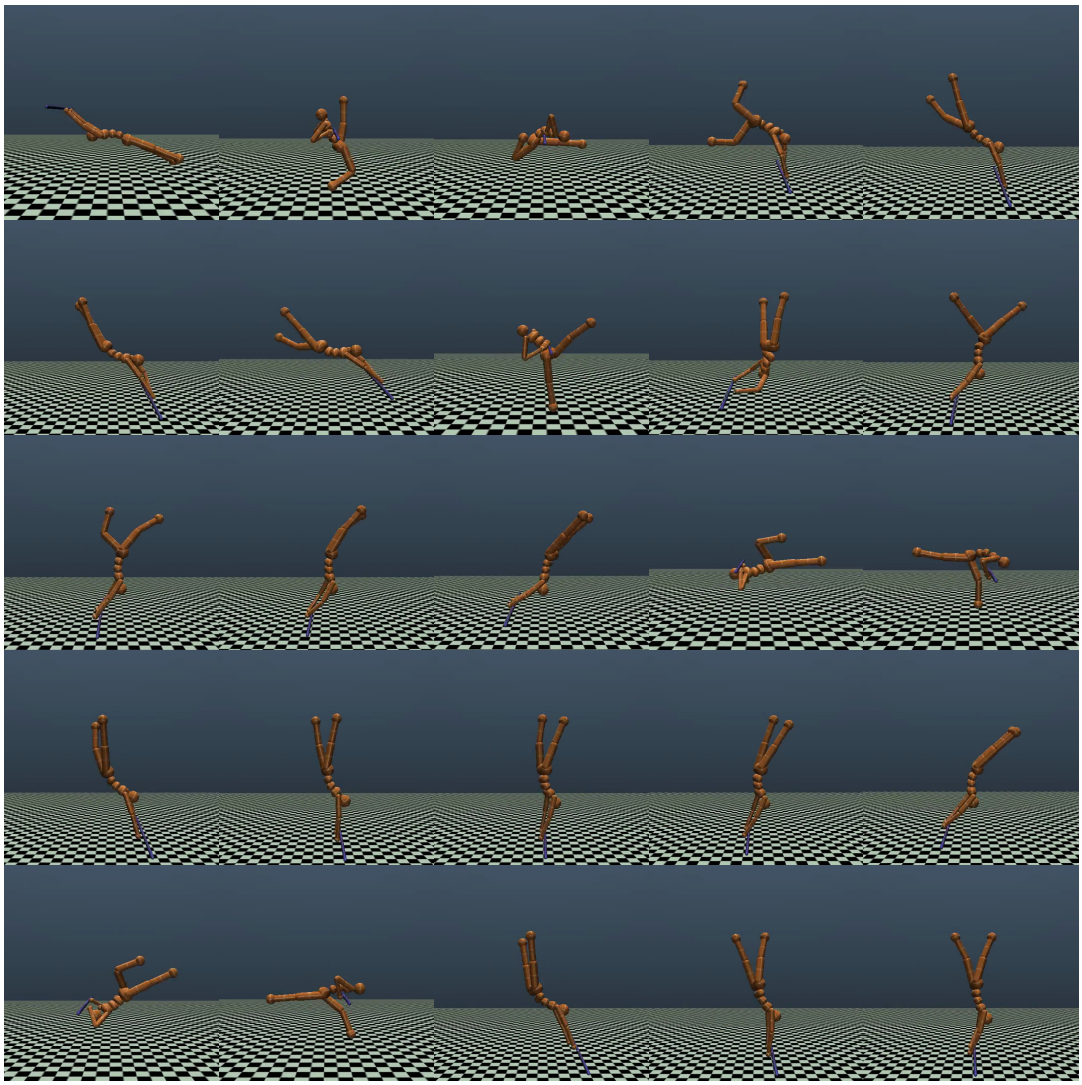


(C) (D)

FIGURE 4.16: Balanced positions

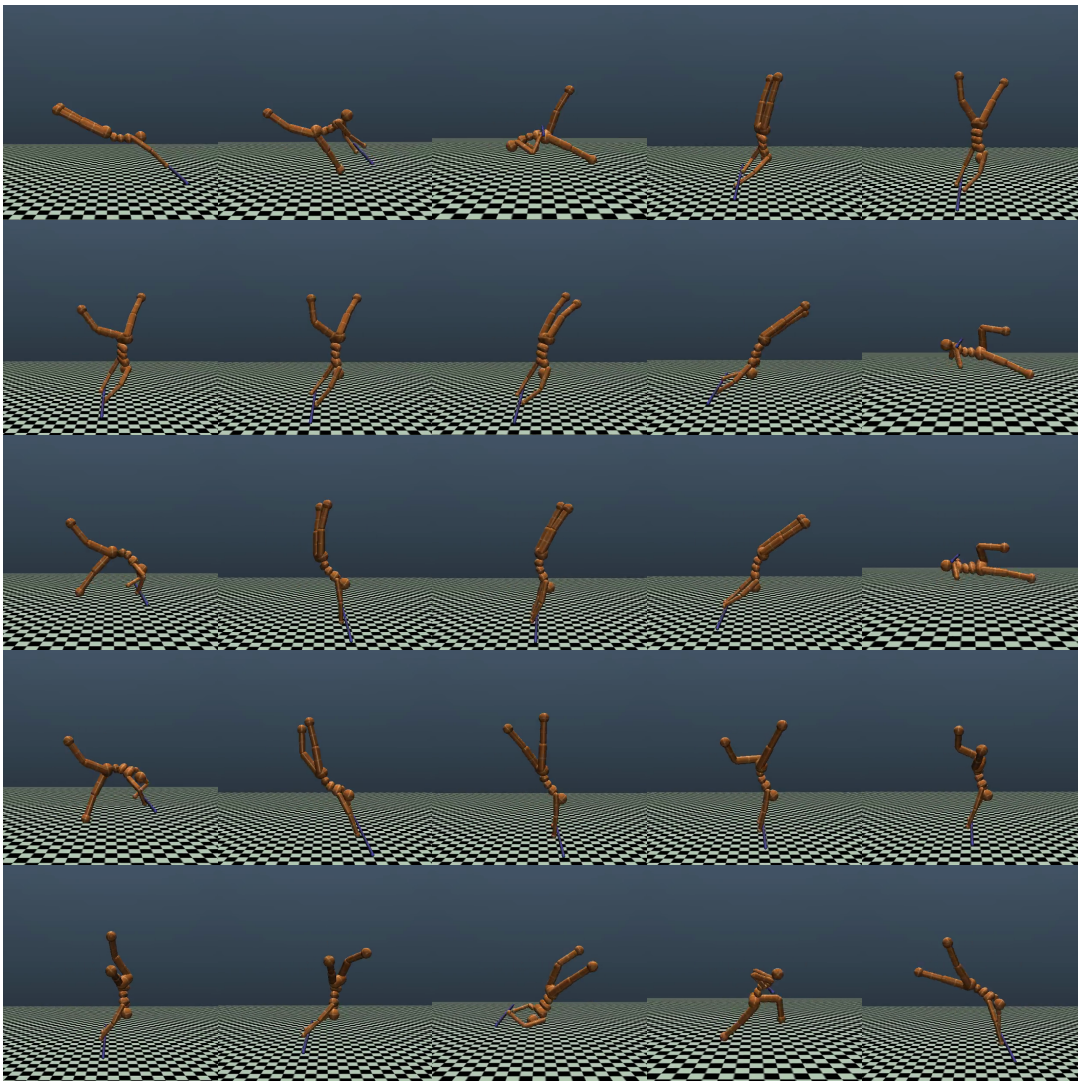
Comparing Figure 4.9 to the improved one Figure 4.14 we see that the average of negative spikes was 35 in 10000 episodes and in the improved version it reduced to an average 15 in the same number of episodes. Also in the agent testing curve Figure 4.14 we can see that negative spikes are much less and 3 or 4 in 2000 testing episodes, with the exception of the first 2000 testing episodes in which the agent still has not converged to the max reward value.
In Figure 4.16 some images of positions that the humanoid managed to balance at for several seconds throughout the simulations are presented.

FIGURE 4.17: Episode frames from a right side initial position.



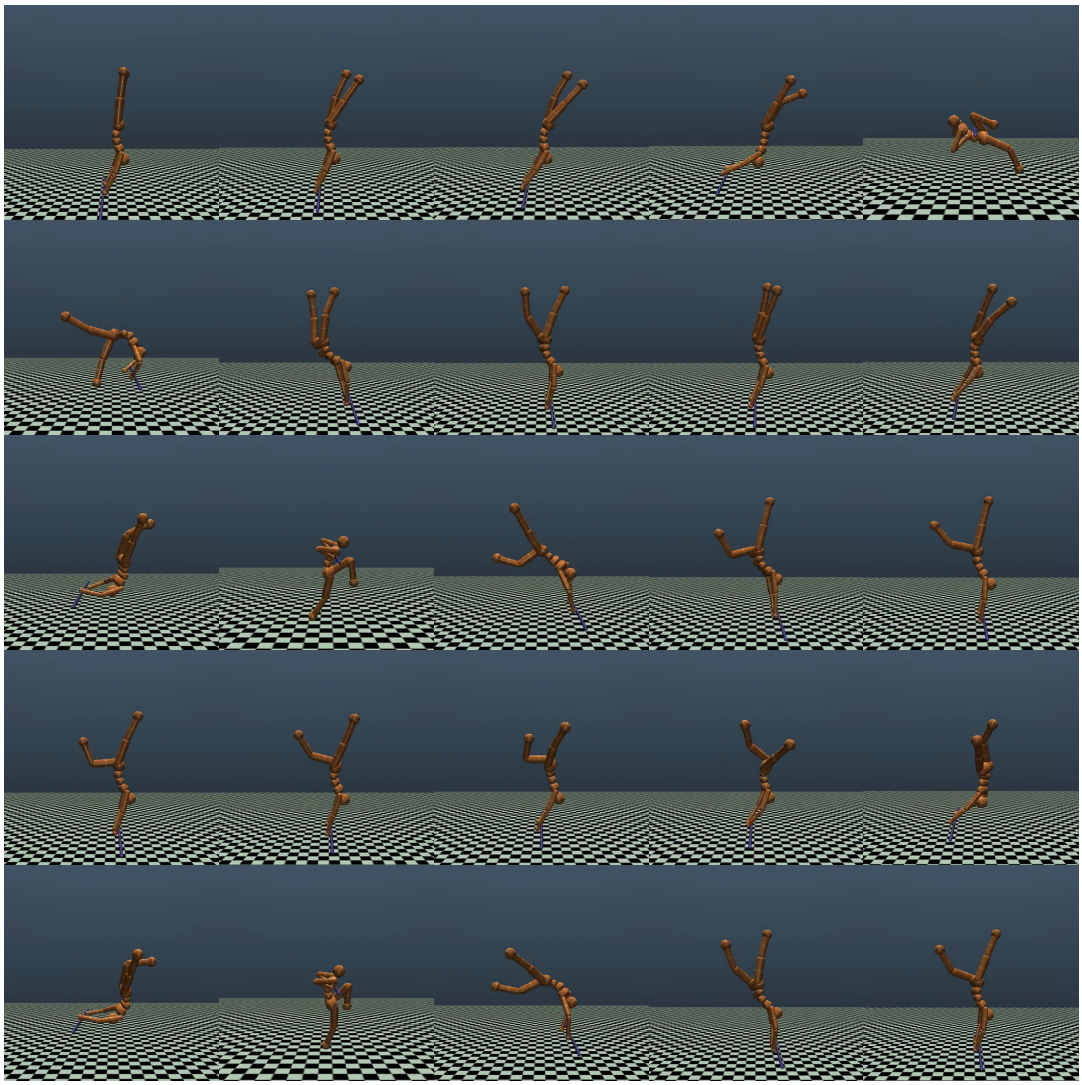
In Figure 4.17 25 frames of an episode of a trained agent starting the testing episode at a position on the right side of the bar are presented.

FIGURE 4.18: Episode frames from a left side initial position.



In Figure 4.18 25 frames of an episode of a trained agent starting the testing episode at a position on the left side of the bar are presented.

FIGURE 4.19: Episode frames from an upright initial position.



In Figure 4.19 25 frames of an episode of a trained agent starting the testing episode at an upright position on the bar are presented.

4.2.2 A2C

For the A2C implementation the same set-up rewards and episodes were used. The A2C algorithm seemed really promising at the first experiments in Section 4.1.2 as it reached a local optimum really early on in the training and it was faster than the DDPG implementation. But as it can be deduced by the learning curve produced, it yielded below par results in these experiments (with the random initialization). We believe that the randomness introduced in this step of the experiments complexes the situation more for the algorithm to work, although it produces better knowledge of the environment for the agent. Considering that to implement A2C two or more environments have to be simulated in parallel, the randomness that we introduce to the agent leads to eight environments each with a random of eight initial positions, this situation produces too much information for the agent to correlate and to find a policy that serves the goal. To evaluate the A2C, the average reward across the parallel environments is presented Figure 4.20, which also due to the randomness does not reach the optimal rewards that the DDPG algorithm produced. So, as the DDPG implementation was sufficient enough to train the agent correctly the idea of using or improving the A2C implementation was abandoned.

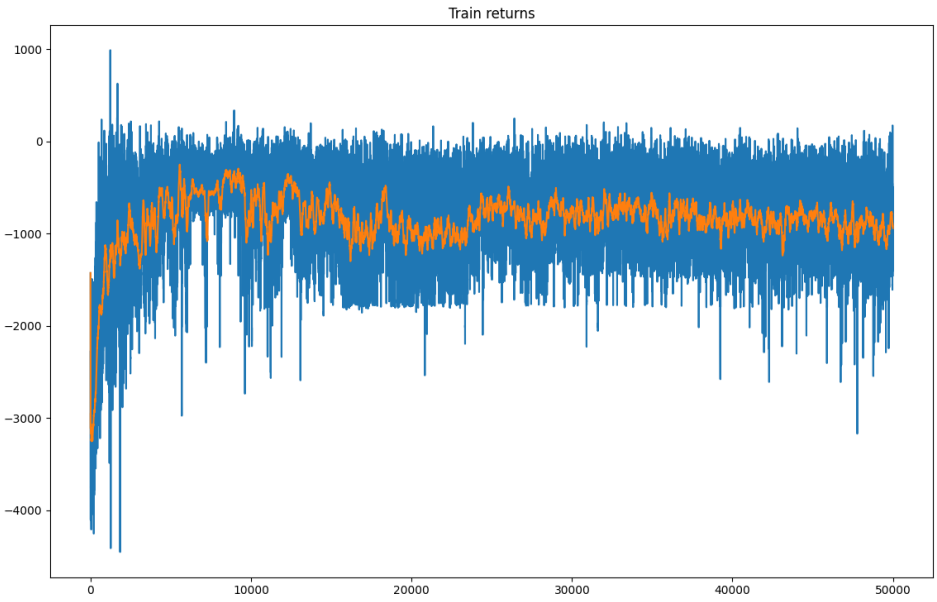


FIGURE 4.20: A2C average for 8 environments

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The goal of this thesis was to train an agent using Reinforcement Learning on the gym environment created, that is based on 'Humanoid-v2', to swing up on a bar and balance a humanoid 3D robot model. To do so, the xml model that was based on the humanoid.xml had to be created and the environment altered in order to fit the new task, of swinging up and balancing. To do so, DDPG and A2C algorithms approaches were explored, which are considered to be state-of-the-art algorithms alongside others, such as TRPO or PPO2. The algorithms were tested and evaluated, and the problem was solved with DDPG, although at first sight A2C seemed to work better time-wise. The process of reaching the end result included many experiments with different environment set-ups eventually getting the best results described in Section 4.2.1.

5.2 Future Work

Based on this project an interesting and possible improvement would be to experiment on the same concept, but with the states being visual representations of the environment and train the agent using Convolutional Neural Networks that are proven to work great with image recognition and are used to train agents on Atari games.
In our experiments, the humanoid would always eventually fall below the bar due to the shape of the hands and the bar, both being sphere or capsule like bodies with too little contact area. As designing more detailed hand elements in MuJoCo would be quite challenging, a potential redesign of the model using other physics engines such as Gazebo or the Unity engine, would be interesting. But, for this experiment a new environment from scratch should be designed outside the OpenAi Gym as there are not ready-made environments of other 3D physics engines in the Gym library.
As for further improvements on this project, other algorithms in the Baselines library could be tested including the aforementioned TRPO and PPO2.
Last, a very interesting expansion of this project would be to apply the trained agent on a real life robotic setup and not just the 3D simulation, although this would be really challenging and extreme resources would be required.

Bibliography

[1] Greg Brockman et al. "OpenAI Gym". In: *CoRR* abs/1606.01540 (2016). arXiv: 1606.01540. URL: http://arxiv.org/abs/1606.01540.

[2] Timothy Lillicrap et al. "Continuous control with deep reinforcement learning". In: *CoRR* (Sept. 2015).

[3] Ms. Sonali. B. Maind and Ms. Priyanka Wankar. "Research Paper on Basic of Artificial Neural Network". In: *International Journal on Recent and Innovation Trends in Computing and Communication* 2 (1) (2014), pp. 96–100. URL: https://doi.org/10.17762/ijritcc.v2i1.2920.

[4] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[5] OpenAI. *A toolkit for developing and comparing reinforcement learning algorithms*. URL: https://gym.openai.com/envs/.

[6] OpenAI. *Acrobot-v1 Gym environment*. URL: https://gym.openai.com/envs/Acrobot-v1/.

[7] OpenAI. *baselines-acktr-a2c*. URL: https://openai.com/blog/baselines-acktr-a2c/.

[8] OpenAI. *CartPole-v1 Gym environment*. URL: https://gym.openai.com/envs/CartPole-v1/.

[9] OpenAI. *Humanoid-v2 Gym environment*. URL: https://gym.openai.com/envs/Humanoid-v2/.

[10] OpenAI. *InvertedDoublePendulum-v2 Gym environment*. URL: https://gym.openai.com/envs/InvertedDoublePendulum-v2/.

[11] OpenAI. *Pendulum-v0 Gym environment*. URL: https://gym.openai.com/envs/Pendulum-v0/.

[12] David Silver et al. "Deterministic Policy Gradient Algorithms". In: *31st International Conference on Machine Learning, ICML 2014* 1 (June 2014).

[13] Mario Srouji, Jian Zhang, and Ruslan Salakhutdinov. "Structured Control Nets for Deep Reinforcement Learning". In: (Feb. 2018).

[14] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[15] Richard S. Sutton et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Advances in Neural Information Processing Systems 12* (2000), pp. 1057–1063.

[16] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning". In: *Machine Learning 8* (1992), pp. 279–292.