Queen Mary, University of London
School of Engineering and Material Sciences
Final-Year Thesis

Deep Reinforcement Learning for Robotic Systems

Mughees Asif
BEng. Aerospace Engineering
Supervisor: Dr Angadh Nanjangud

April 2021

SCHOOL OF ENGINEERING AND MATERIALS SCIENCE
DEN318: THIRD YEAR PROJECT

APRIL 2021

DECLARATION

This report entitled

Deep Reinforcement Learning for Robotic Systems

Was composed by me and is based on my own work. Where the work of the others has been used, it is fully acknowledged in the text and in captions to table illustrations. This report has not been submitted for any other qualification.

Name: *Mughees Asif*

Signed: ..... M.A .....

Date: ..... 28/04/2021 .....

To Odysseus, Leonardo da Vinci, Alexandre Dumas, Alan Turing, Fyodor Dostoevsky, and Tupac.

"The future is independent of the past, given the present"—David Silver (2015)

Abstract

Artificial Intelligence can be described as the zeitgeist of modern times that is enabling autonomous control systems capable of high-order thinking. The increased production of data via the propagation of internet-based services, coupled with major advancements in mainframe hardware and software, have enabled a myriad of computational techniques leading to an intersection of multi-disciplinary machine learning research. At the cusp of this intersection lies: *Deep Reinforcement Learning* (DRL). A combination of Deep Learning and Reinforcement Learning, DRL is based on the cognitive functionality associated with human consciousness, whilst maintaining the ability to computationally 'learn' from new experiences. DRL has proven to be successful in a wide array of activities including developing proficiency in computer gaming and facilitating robotic locomotion in complex environments. This thesis aimed to develop an industry-ready template via the application of the Proximal Policy Optimisation (PPO) algorithm, developed by OpenAI in 2017, to balance an inverted double pendulum at a local stability point. An Advantage Actor-Critic (`A2C`) neural network configuration, in combination with PPO was implemented to train the agent. The results show the non-linearity of the problem with various starting angles tested and the highest accuracy was recorded at 91%. The thesis serves as a gateway for advanced research into DRL, and introduces the mathematical and computational concepts needed to produce autonomous control systems applicable in safety-, and mission-critical operations.

Keywords—Spacecraft Engineering, Lagrangian Mechanics, Deep Learning, Reinforcement Learning, Proximal Policy Optimisation, OpenAI

Acknowledgements

I would like to sincerely thank my project supervisor, guide and mentor, **Dr Angadh Nanjangud**, for his relentless pursuit in pushing my abilities to a superior level and for affording me the chance to participate in next-generation research.

I would also like to whole-heartedly thank the **School of Engineering and Material Sciences**, including all the lecturers/staff, who have been pivotal in my academic and professional development.

Fortune has an odd way of shuffling cards. Throughout human history, catastrophes have forged a new normal. For this reason, I would like to thank **myself** and my brother, **Sherdil**, for putting in the required time and effort. We do it for posterity.

I would also like to thank my **study group** and the **Engineering cohort of 2021**, who were indeed a wise counsel.

Contents

Tables

Abbreviations

DRL	deep reinforcement learning
CCW	counter-clockwise
w.r.t	with respect to
K.E.	kinetic energy
P.E.	potential energy
DoF	degree-of-freedom
rad	radians
PPO	proximal policy optimisation
A2C	advantage actor-critic
ReLU	rectified linear unit
GAE	generalised advantage estimation

Symbols

x	horizontal distance of the cart from a datum line
$\dot{x}$	velocity of the cart in the x-direction
$\ddot{x}$	acceleration of the cart in the x-direction
θ	angular displacement of the first pendulum
$\dot{\theta}$	angular velocity of the first pendulum
$\ddot{\theta}$	angular acceleration of the first pendulum
ϕ	angular displacement of the second pendulum
$\dot{\phi}$	angular velocity of the second pendulum
$\ddot{\phi}$	angular acceleration of the second pendulum
m	mass of the cart
u	horizontal force component
M	mass of the pendulum(s)
I	moment of inertia of the pendulum(s)
g	gravitation constant
$\mathcal{L}$	Lagrangian function
Y	activation function
w	weights
b	bias
a	input
∇	gradient
ω_-	gradient descent
ω_+	gradient ascent
α	learning rate
S	state
A	action
$\mathbb{P}$	transition probability
R	reward
γ	discount factor
Q	Q-function
V	Value-function
$\hat{A}$	advantage function
J	goal of reinforcement learning
$\mathbb{E}$	expected return
λ	smoothing parameter
ϵ	clipping policy

Figures

1 Autonomous Spacecraft Robotics

Apart from the space race of the Cold War in the 1960s, space travel in the 21st century is experiencing a resurgence of interest, partly due to pioneering private companies such as SpaceX, Virgin Galactic, and Blue Origin, but also due to a renewed interest in interstellar human exploration. The financial implications of this new-age space race have given rise to researchers developing cost-effective control systems capable of handling nebulous mission objectives, such as perturbative gravity fields and mercurial space environments. One of the main paradigms being researched into for autonomous space travel includes Deep Travel, which is built upon Artificial Intelligence (AI), and modelled on the cognitive abilities of the human mind to seek optimal courses of action. Currently, there are two autonomous spacecraft control systems school of thought:

- **Rule-based Autonomy (RA)**: The RA model uses pre-defined processes to classify different modes of the spacecraft whilst also distinguishing the possible transitions from one mode to another [1]. However, instituting rigid rules for an unpredictable environment infers problems from the outset.
- **Optimisation-based Autonomy (OA)**: The OA model encapsulates a spacecraft system into a framework of constrained optimisation, where the mission aims are the values being optimised and once achieved, are used as terminal conditions [1]. This method has a large computational overhead and pre-requires several realistic models developed via various testing phases.

Although, these two approaches have found seamless uses in the space industry, evidently, there is a growing need for the development of *task-agnostic* learning algorithms that will enable future spacecraft systems to make high-order autonomous decisions without human intervention.

1.1 Aims of the Project

The main aim of this thesis is to develop an industry-standard template for computationally modelling robotic systems, and to enable autonomous control systems for high-order thinking in a different environments. This research also aims to contribute to the open-source science community by enabling the proliferation of AI research that is accessible at *all* levels, and to complement the development of autonomous space engineering control systems for purposes such as docking and berthing of spacecraft, or the stablisation of a multi-stage rocket.

2 Inverted Double Pendulum

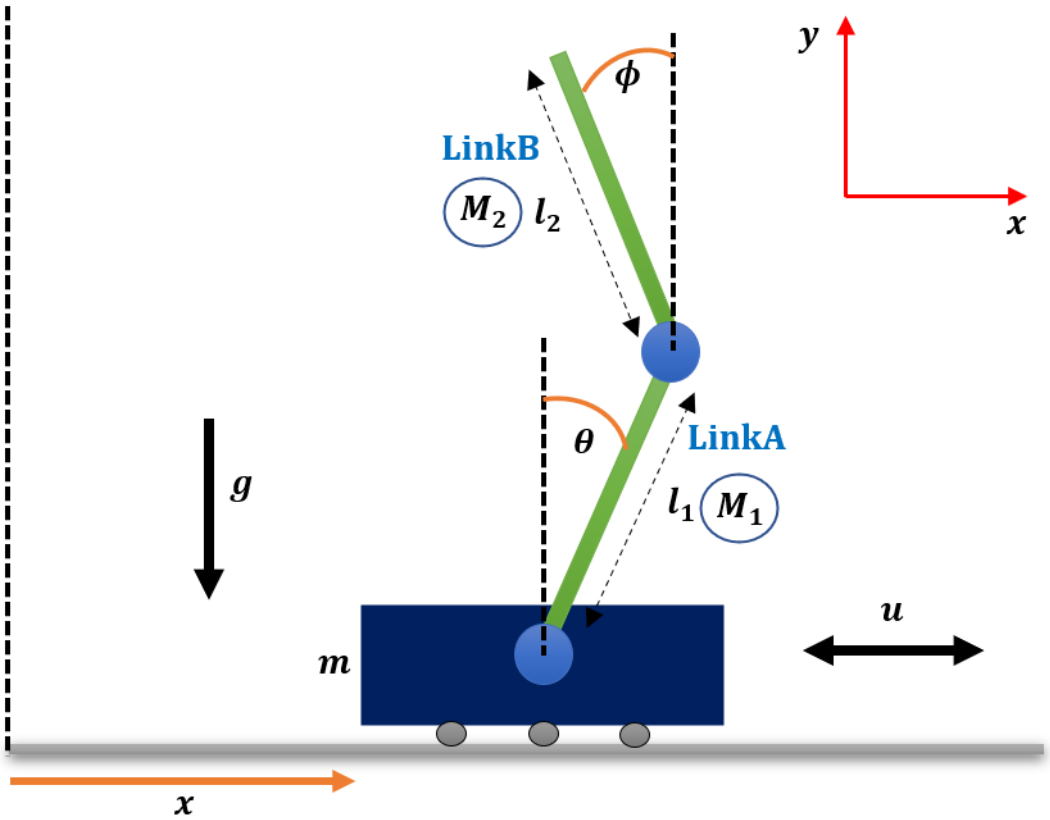


Figure 1: Free-body diagram

Table 1: System parameter notation

Parameter	**Notation**
Mass of the cart	m
Mass of the pendulum(s)	$M_1 = M_2 = M$
Length of the pendulum(s)	$l_1 = l_2 = l$
Angle of the first pendulum w.r.t the vertical (CCW+)	θ
Angle of the second pendulum w.r.t the first pendulum (CCW+)	ϕ
Moment of inertia of the pendulum(s)	$I_1 = I_2 = I$
Horizontal cart position	x
Horizontal force applied to the cart	u
Gravitational constant	g

An inverted double pendulum is a classic example of a holonomic, simple-to-build, non-linear, and chaotic mechanical system. The components of the system are outlined in Table 1, and Fig. 1 shows an under-actuated system as the degrees-of-freedom (DoF) outnumber the actuator (the cart), that is also responsible for parrying the conflicting motion of the pendulums due to the force acting on the cart.

3 Equations of Motion

3.1 Outline

This section highlights the derivation of the equations of motion and, consequently, the acceleration of each component, that will be used by the RL model to determine the amount of force required to balance the system at the equilibrium point i.e., $\theta = 0$ and $\phi = 0$. The derivation uses the Lagrangian Mechanics, as opposed to the classical Newtonian Mechanics due to [2]:

- Allows for an optimised-constraint approach to the chaotic double pendulum system. Instead of defining a vectorial definition of the different forces, as in classical Newtonian mechanics, the Euler-Lagrange equations allow the formulation of the equations of motion for any mechanical system using two scalar quantities: kinetic and potential energy.
- Through certain simplifying constraints, the study of this system transforms into an ideal and holonomic (total DoF is equal to the controllable DoF) study, where using the Euler-Lagrange equations proves less complex mathematically to describe the system. Moreover, the geometrical design of the components of the system are disregarded, in favour of generalised coordinates, thereby, further simplifying the development of an analytical solution.

Following constraints hold true for the subsequent derivation of the equations of motion:

- **Friction**: Frictional forces between the cart and ground are disregarded.
- **Movement**: The cart is constrained to move in a straight line i.e., horizontally in a two-dimensional Euclidean space: left or right.
- **Dimensions**: The pendulums are of equal length and mass.
- **Modelling**: The cart and the pendulums are modelled as point masses, as the "... simple model avoids the complicated multiple integrals associated with modelling the cart and pendulum bobs as bodies with shape" [3].

3.2 Mathematical Modelling

3.2.1 Kinetic and Potential Energy

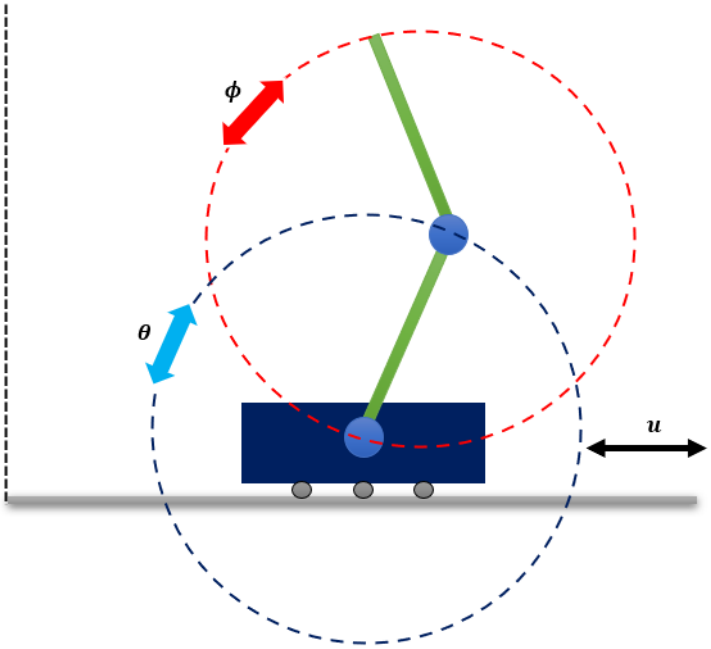


Figure 2: Translational and rotational motion of the system

The Euler-Lagrange method requires the total kinetic (K.E) and potential energy (P.E) of the system. From classical mechanics the total kinetic energy $K.E._T$ of the system in Fig. 2, which is a rigid body with planar motion, is comprised of the translational $K.E._{tr}$ and rotational $K.E._{ro}$ kinetic energy, and potential energy of all the components:

$$K.E._{tr} = \frac{1}{2}mv^2$$
$$K.E._{ro} = \frac{1}{2}I\omega^2$$
$$P.E = mg\Delta h$$

where, m is the mass of the component, v is the linear velocity, I is the moment of inertia, ω is the angular velocity, g is the gravitational constant, and Δh is the difference in the vertical position.

From the movement constraint defined in Section 3.1, the horizontal y_c and vertical x_c displacement of the cart c:

$$x_c = x(t) \qquad\qquad y_c = 0$$

The total kinetic and potential energy of the system is:

$$K.E._T = K.E._c + K.E._1 + K.E._2$$
$$P.E._T = P.E._c + P.E._1 + P.E._2$$

where,

$$
\begin{aligned}
K.E._c &= \frac{1}{2}m\dot{x}^2 \\
K.E._1 &= \left[\frac{1}{2}M\dot{x}^2\right] + \left[\frac{1}{2}\left(Ml^2 + I\right)\dot{\theta}^2\right] + \left[Ml\dot{x}\dot{\theta}\cos\theta\right] \\
&= \frac{1}{2}\left[M\dot{x}\left(\dot{x} + 2l\dot{\theta}\cos\theta\right) + \dot{\theta}^2(Ml^2 + I)\right] \\
K.E._2 &= \left[\frac{1}{2}M\dot{x}^2\right] + \left[\frac{1}{2}Ml^2\dot{\theta}^2\right] + \left[\frac{1}{2}M\dot{\phi}^2\left(Ml^2 + I\right)\right] + \left(Ml\dot{x}\dot{\theta}\cos\theta\right) + \\
&\qquad \left(Ml\dot{x}\dot{\phi}\cos\phi\right) + \left[\frac{1}{2}Ml\dot{\theta}\dot{\phi}\cos\left(\theta - \phi\right)\right] \\
&= \frac{1}{2}\left[\dot{x}^2 + l^2\dot{\theta}^2 + \dot{\phi}^2\left(Ml^2 + I\right) + Ml\dot{\theta}\dot{\phi}\cos\left(\theta - \phi\right) + 2Ml\dot{x}\left(\dot{\theta}\cos\theta + \dot{\phi}\cos\phi\right)\right]
\end{aligned}
$$

$$
\begin{aligned}
P.E._c &= 0 \\
P.E.1 &= Mgl\cos\theta \\
P.E.2 &= Mgl\cos\theta + Mgl\cos\phi
\end{aligned}
$$

$$
\begin{aligned}
\therefore \boldsymbol{K.E._T} = \frac{\mathbf{1}}{\mathbf{2}}\Big[&\boldsymbol{m\dot{x}^2 + M\dot{x}\left(\dot{x} + 2l\dot{\theta}\cos\theta\right) + \left(\left(Ml^2 + I\right)\left(\dot{\phi}^2 + \dot{\theta}^2\right)\right) +} \\
&\boldsymbol{\dot{x}^2 + l^2\dot{\theta}^2 + Ml\dot{\theta}\dot{\phi}\cos\left(\theta - \phi\right) + 2Ml\dot{x}\left(\dot{\theta}\cos\theta + \dot{\phi}\cos\phi\right)}\Big]
\end{aligned} \tag{1}
$$

$$
\therefore \boldsymbol{P.E._T} = \boldsymbol{Mgl(2\cos\theta + \cos\phi)} \tag{2}
$$

3.2.2 Euler-Lagrange Equations

The *Principle of Least Action* states that for any given mechanical system the action S has "...its mathematical formulation based on the geometry of space and time, and on the concept of energy. Velocity is more important than acceleration, and force, the key quantity that occurs in equations of motion, becomes a secondary, derived concept. This is helpful, as velocity is simpler than acceleration, and energy is something that is intuitively better understood than force" [4]. Therefore, S can be mathematically defined as:

$$
S = \int_{t_0}^{t_1} K.E._T - P.E._T \ \ dt
$$

The Lagrangian $\mathcal{L}$ can be used to characterise the state of the system:

$$
\mathcal{L} = K.E._T - P.E._T
$$

$$
S = \int_{t_0}^{t_1} \mathcal{L} \ \ dt
$$

$$
\begin{aligned}
\because \mathcal{L} = \frac{1}{2}\Big[&m\dot{x}^2 + M\dot{x}\left(\dot{x} + 2l\dot{\theta}\cos\theta\right) + \left(Ml^2 + I\right)\left(\dot{\phi}^2 + \dot{\theta}^2\right) + \dot{x}^2 + l^2\dot{\theta}^2 + \\
&Ml\dot{\theta}\dot{\phi}\cos\left(\theta - \phi\right) + 2Ml\dot{x}\left(\dot{\theta}\cos\theta + \dot{\phi}\cos\phi\right)\Big] - Mgl\left(2\cos\theta + \cos\phi\right)
\end{aligned} \tag{3}
$$

The standard Euler-Lagrange formulation for a system is:

$$\frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{x}} - \frac{\partial \mathcal{L}}{\partial x} = 0 \tag{4}$$

However, since the system is three-dimensional with an external force u applied to it, the Lagrange-D'Alembert Principle can be used to add a generalised force Q^P to Eq. 4. The principle is stated as the total work done by applied forces in a virtual displacement δx is equal to zero.

$$\begin{aligned}
&\implies \int_{t_0}^{t_1} \Big(\delta\mathcal{L} \,-\, Q^P \cdot \delta x\Big) dt \,=\, 0 \\
&\implies \int_{t_0}^{t_1} \Bigg(\Big[\, \frac{d}{dt}\Big(\frac{\partial \mathcal{L}}{\partial \dot{x}}\Big) \,-\, \frac{\partial \mathcal{L}}{\partial \dot{x}} \,\Big] \cdot \delta x \,-\, Q^P \cdot \delta x \Bigg) dt \,=\, 0 \\
&\implies \int_{t_0}^{t_1} \Bigg(\Big[\, \frac{d}{dt}\Big(\frac{\partial \mathcal{L}}{\partial \dot{x}}\Big) \,-\, \frac{\partial \mathcal{L}}{\partial \dot{x}} \,-\, Q^P \,\Big] \cdot \delta x \Bigg) dt \,=\, 0 \\
&\therefore \frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{x}} - \frac{\partial \mathcal{L}}{\partial x} = Q^p
\end{aligned}$$

Therefore, the three equations of motion for the system equate to:

$$\frac{\boldsymbol{d}}{\boldsymbol{dt}}\frac{\boldsymbol{\partial \mathcal{L}}}{\boldsymbol{\partial \dot{x}}} - \frac{\boldsymbol{\partial \mathcal{L}}}{\boldsymbol{\partial x}} = \boldsymbol{u}$$

$$\boldsymbol{Ml}\Big[\Big(\boldsymbol{2\ddot{\theta}\cos\theta + \ddot{\phi}\cos\phi}\Big) - \Big(\boldsymbol{2\dot{\theta}^2\sin\theta + \dot{\phi}^2\sin\phi}\Big)\Big] + \boldsymbol{\ddot{x}(M+m+1)} = \boldsymbol{u} \tag{5}$$

$$\frac{\boldsymbol{d}}{\boldsymbol{dt}}\frac{\boldsymbol{\partial \mathcal{L}}}{\boldsymbol{\partial \dot{\theta}}} - \frac{\boldsymbol{\partial \mathcal{L}}}{\boldsymbol{\partial \theta}} = \boldsymbol{0}$$

$$\boldsymbol{2Ml}\Big[\boldsymbol{\ddot{x}\cos\theta - g\sin\theta} + \frac{\boldsymbol{1}}{\boldsymbol{4}}\Big(\boldsymbol{\dot{\phi}^2\sin(\theta-\phi) + \ddot{\phi}\cos(\theta-\phi)}\Big)\Big] + \boldsymbol{\ddot{\theta}(I + l^2(M+1))} = \boldsymbol{0} \tag{6}$$

$$\frac{\boldsymbol{d}}{\boldsymbol{dt}}\frac{\boldsymbol{\partial \mathcal{L}}}{\boldsymbol{\partial \dot{\phi}}} - \frac{\boldsymbol{\partial \mathcal{L}}}{\boldsymbol{\partial \phi}} = \boldsymbol{0}$$

$$\boldsymbol{Ml}\Big[\boldsymbol{\ddot{x}\cos\phi - g\sin\phi} + \frac{\boldsymbol{1}}{\boldsymbol{2}}\Big(\boldsymbol{\ddot{\theta}\cos(\theta-\phi) - \dot{\theta}^2\sin(\theta-\phi)}\Big)\Big] + \boldsymbol{\ddot{\phi}(I + Ml^2)} = \boldsymbol{0} \tag{7}$$

3.2.3 Linearisation and Acceleration

Table 2: Linerisation of the equations of motion using small-angle approximations

$\sin\theta \approx \theta$	$\sin\phi \approx \phi$
$\cos\theta = 1$	$\cos\phi = 1$
$\dot{\theta}^2 = 0$	$\dot{\phi}^2 = 0$
$\sin(\theta-\phi) \approx \theta - \phi$	$\cos(\theta-\phi) = 1$

Through a quick perusal of Eq. 5, 6 & 7, the movement of the double pendulum is best described as experiencing the *butterfly effect*: the dependence of a non-linear system on

the initial starting conditions. If the system is set to commence movement with a small starting θ and ϕ, a linear conventional multi-degree freedom of movement is observed [5]. However, as the initial conditions are incrementally altered to larger starting angles, the system displays a chaotic and non-linear movement pattern. Therefore, Eq. 5, 6 & 7 need to be linearised by using small-angle approximations to ensure local stability at the equilibrium point i.e., the upright vertical position, where $\theta = 0$ and $\phi = 0$.

Using the linearisation in Table. 2, the equations of motion are:

$$Ml(2\ddot{\theta} + \ddot{\phi}) + \ddot{x}\left(M + m + 1\right) = u$$
$$2Ml\left(\ddot{x} - g\theta + \frac{1}{4}\ddot{\phi}\right) + \ddot{\theta}\left(I + l^2\left(M + 1\right)\right) = 0$$
$$Ml\left(\ddot{x} - g\phi + \frac{1}{2}\ddot{\theta}\right) + \ddot{\phi}\left(I + Ml^2\right) = 0$$

Re-arranging and solving for the accelerations:

$$\ddot{x} = \frac{u - Ml(2\ddot{\theta} + \ddot{\phi})}{M + m + 1} \tag{8}$$
$$\ddot{\theta} = -\frac{2Ml\left(\ddot{x} - g\theta + \frac{1}{4}\ddot{\phi}\right)}{I + l^2\left(M + 1\right)} \tag{9}$$
$$\ddot{\phi} = -\frac{Ml\left(\ddot{x} - g\phi + \frac{1}{2}\ddot{\theta}\right)}{I + Ml^2} \tag{10}$$

Solving Eq. 8, 9 and 10 simultaneously:

$$\ddot{\boldsymbol{x}} = \frac{\begin{array}{l}-\boldsymbol{M^2gl^2}\left[\boldsymbol{\theta}\left(\boldsymbol{4I} + \boldsymbol{Ml}\left(\boldsymbol{4l} - \boldsymbol{1}\right)\right) + \boldsymbol{\phi}\left(\boldsymbol{I} + \boldsymbol{Ml}\left(\boldsymbol{l} - \boldsymbol{1}\right) + \boldsymbol{l^2}\right)\right] \\ +\boldsymbol{u}\left(\boldsymbol{I^2} + \boldsymbol{Il^2}\left(\boldsymbol{2M} + \boldsymbol{1}\right) + \boldsymbol{Ml^2}\left(\boldsymbol{Ml^2} - \frac{\boldsymbol{1}}{\boldsymbol{4}}\boldsymbol{M} + \boldsymbol{l^2}\right)\right)\end{array}}{\boldsymbol{X}} \tag{11}$$

$$\ddot{\boldsymbol{\theta}} = \frac{\boldsymbol{Ml}\left[\begin{array}{l}-\boldsymbol{2Mgl}\left(\frac{\boldsymbol{1}}{\boldsymbol{4}}\left(\boldsymbol{M} + \boldsymbol{m} + \boldsymbol{1}\right) - \boldsymbol{Ml}\right)\boldsymbol{\phi} \\ +\boldsymbol{2g}\left(\boldsymbol{I}\left(\boldsymbol{M} + \boldsymbol{m} + \boldsymbol{1}\right) + \boldsymbol{Ml^2}(\boldsymbol{m} + \boldsymbol{1})\,\boldsymbol{\theta}\right. \\ \left.-\boldsymbol{2u}\left(\boldsymbol{I} + \boldsymbol{Ml}\left(\boldsymbol{l} - \frac{\boldsymbol{1}}{\boldsymbol{4}}\right)\right)\right.\end{array}\right]}{\boldsymbol{X}} \tag{12}$$

$$\ddot{\boldsymbol{\phi}} = \frac{\boldsymbol{Ml}\left[\begin{array}{l}-\boldsymbol{Mgl}\left(-\boldsymbol{M}\left(\boldsymbol{4l} - \boldsymbol{1}\right) + \boldsymbol{m} + \boldsymbol{1}\right)\boldsymbol{\theta} \\ +\boldsymbol{g}\left(\boldsymbol{I}\left(\boldsymbol{M} + \boldsymbol{m} + \boldsymbol{1}\right) + \boldsymbol{Ml^2}\left(\boldsymbol{m} - \boldsymbol{3M} + \left(\frac{\boldsymbol{m+1}}{\boldsymbol{M}}\right) + \boldsymbol{2}\right)\right)\boldsymbol{\phi} \\ -\boldsymbol{u}\left(\boldsymbol{I}(\boldsymbol{I} + \boldsymbol{1}) + \boldsymbol{Ml}\left(\boldsymbol{l} - \boldsymbol{1}\right)\right)\end{array}\right]}{\boldsymbol{X}} \tag{13}$$

where,

$$X = I\left[I + l^2 + m\left(I + l^2\right)\right] - M^2l^2\left[M\left(2l\left(2l - 1\right) + \frac{1}{4}\right) + \left(3I - l^2\left(m - 1\right) + \frac{1}{4}\left(m + 1\right)\right)\right] + M\left(I^2 + l^2\left[I\left(2m + 3\right) + l^2(m + 1)\right]\right)$$

Therefore, Eq. 11, 12 and 13, were used in the RL model to map desirable actions to given states of the two pendulums using the force on the cart i.e., acceleration of each component.

4 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) is an amalgamation of Deep Learning and Reinforcement Learning, both of which are a subset within Machine Learning. The two nascent paradigms allow an agent to define an objective directly from raw inputs, whilst using minimal domain knowledge [6]. DRL has recently experienced wide-spread adoption by different entities, ranging from government institutions that have developed intelligent monitoring systems, to pharmaceutical companies that have used DRL to expedite vaccine development, evident from the recent COVID-19 crisis. This rapid pace of industrial adoption can be attributed to major advances in mainframe hardware, computational engineering techniques, and improved research into the domain.

4.1 Deep Learning

Deep Learning (DL) is a neuro-scientific technique that emulates the structure of the nervous system by leveraging computational learning algorithms, more commonly known as neural networks, to learn abstraction from randomly generated data, enable extraction of salient features from given information, and to identify hierarchies of the identified features. Algorithms, in vanilla computer science, are defined as a set of instructions structured sequentially to produce an output, but in the DL domain, algorithms are mathematical functions designed to place importance on user-defined parameters from stochastic data.

4.1.1 Neural Networks

DL removes the need for human intervention that is witnessed in classical machine learning at the data cleansing stage, where the data is labelled and categorised for inputting into a classification algorithm. Instead, a neural network is designed using various 'layers' (Fig. 3), from which each layer extracts a particular feature that is defined as a quantifiable and individual property of the system under consideration [7]. The layers are formed using 'nodes', where each node has an underlying mathematical architecture comprising of an activation function Y that consists of weights w, biases b and inputs a:

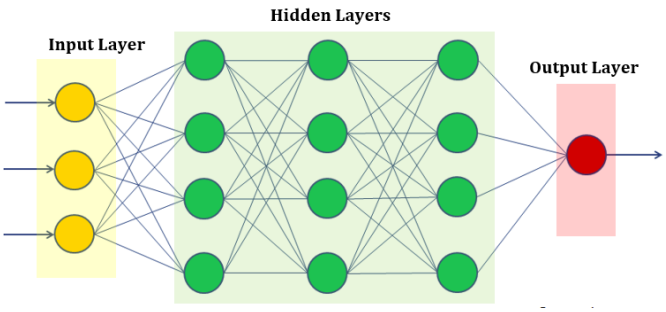
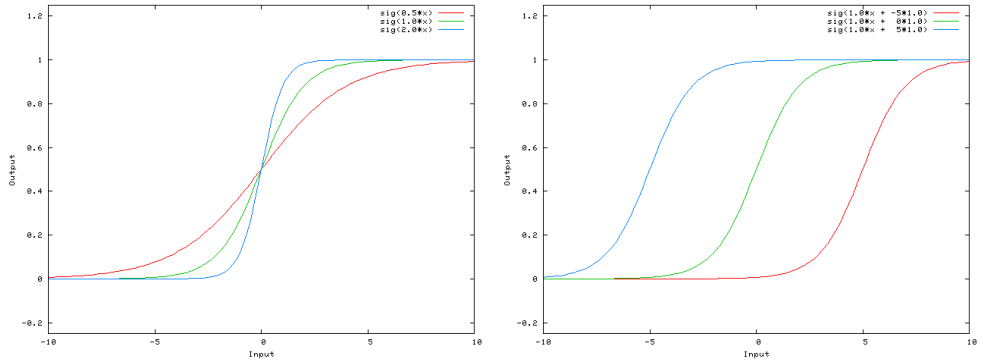


Figure 3: Artificial neural network

$$Y = \sum_{i=1}^{n} (w_i a_i) + b$$

Weights are defined as the controlling signal or the strength of the connection between each node which allows the neural network to decide how much influence each input has on each output [8].

The keen-eyed reader would have recognised that the equation representing the activation function Y resembles the equation of a line, $y = mx + c$, where the y-intercept c is replaced by the term bias b. Biases are constant values designed to add variability to the weights being sampled at each node, thereby, allowing the neural network to store a richer representation of the input as mapped to the defined weights [9]. The term, effectively, shifts the activation function from left to right, and vice versa (Fig. 4a to 4b), thereby, letting the neural network modulate the required intercept of the parameter space needed for successful learning.



(a) Bias = 0 (b) Bias $\neq$ 0

Figure 4: Impact of bias values on the activation function [9]

An activation function is designed to act as a threshold value which determines if the particular neuron's output should be forwarded or negated. If the output is within limits, the activation function at each node 'fires' the output along the neural network for further analysis, enabling complex and non-linear relationships to be determined.

Each node is connected to all the nodes in the next layer, and the output of each node in the current layer acts as the input for the subsequent node in the next layer. Therefore, a neural network learns to map a set of inputs to a set of outputs using the data provided. This process defines a neural network, and the forward propagation of feature extraction allows the predictive model to learn and adapt to increasingly complicated tasks.

4.1.2 Gradient Descent

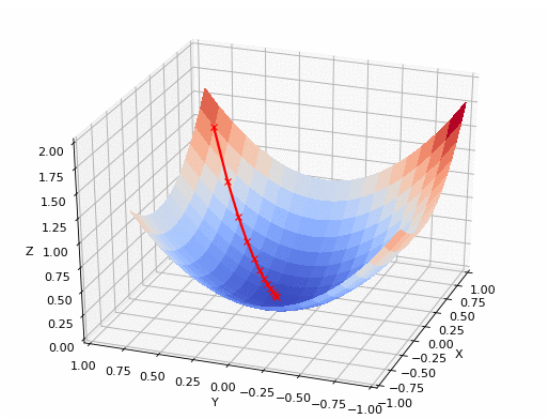


Figure 5: Gradient descent [10]

A neural network starts training by randomly assigning weights (stochastic), which correlate to an output, and then optimising the weights until the *user-defined* output is achieved. The process is repeated several times, where at each iteration a loss function determines how far the current output is from the desired output. "The [loss] function has its own curve and gradients. The slope of this curve [shows] how to update [the] parameters to make the model more accurate" [10].

The gradient is a measure of the change in all the values of the weights w.r.t the change in error/loss. Gradient descent is an optimisation algorithm that allows for the minimisation of the loss function, where the gradient is calculated at each step to find the fastest route to the global minima in the input space i.e., the *steepest descent* (Fig. 5).

$$\omega_{-} \leftarrow \omega - \alpha \cdot \nabla_{\omega} \sum_{1}^{m} L_m(\omega)$$

where, ω represents the weights vector, which lies in the xyz plane, $L_m(\omega)$ is the loss function, α is the learning rate and ∇_{ω} is the gradient of the parameter space of the problem. The gradient is the partial derivative of the function at that point in the parameter space, where each derivative denotes the directional component of the variables towards the *steepest ascent* [11]. Since, the steepest descent is needed, it is *subtracted* from the weights to get the steepest descent towards the global minima. The learning rate α determines how large a step the network should take in the direction of the descent. Too large a learning rate and the network will overshoot, and a small learning rate will increase computational time.

4.2 Reinforcement Learning

Based on the cognitive development theory by Jean Piaget (1896-1980), a fundamental way humans conduct the process of learning comes from the ability to make mistakes and learn i.e., cause-and-effect, or Reinforcement Learning (RL). RL is a computational learning model that is "...situated in between supervised learning and unsupervised learning, [and] deals with learning in sequential decision-making problems in which there is limited feedback" [12]. In the computational domain, the aim of training a model would involve the maximisation of a cumulative numerical reward in a closed-loop network, where the model is not explicitly programmed with a set of starting instructions, but rather employs a stochastic decision-making process, akin to a Monte Carlo simulation [13].

4.2.1 Markov Decision Process

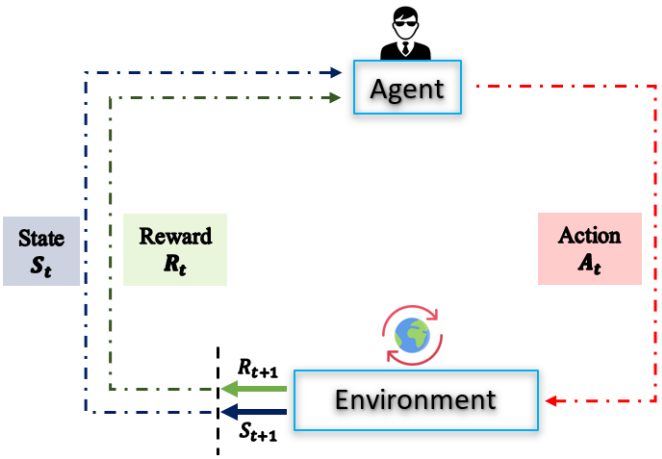


Figure 6: Markov Decision Process (MDP)

RL adheres to the Markov Decision Process (MDP) which computationally describes the interaction of an agent with a simulated environment. Fig. 6 displays a characteristic MDP, where the environment is responsible for setting the 'state' that instigates an 'action' by the agent, from where a numerical 'reward' is imparted based on the desirability of the action. "MDPs are a classical formalisation of sequential decision-making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those, the future rewards. Thus

MDPs involve delayed reward [aggregation], and the need to trade-off immediate and delayed reward [strategies]" [13]. MDP can be represented by the 5-parameter tuple, $(S, A, \mathbb{P}, R, \gamma)$:

- **State** S_t: State is a function of the historical information (experience) stored through a sequence of observations and corresponding actions within an observable time frame.
- **Action** A**:** Actions represent finite changes in the state of the environment. Each action once committed is assigned a 'value' (scalar) which acts as a ranking system for desirability based on the reward outcome.
- **Transition Probability** $\mathbb{P}$: In essence, most cases of RL utilise the Markov Property which is poetically described by David Silver (2015) as, "The future is independent of the past, given the present" [6]. Mathematically, the property can be described as:

  $$\mathbb{P}\left[S_t + 1 \mid S_t\right] = \mathbb{P}\left[S_t + 1 \mid S_1, \ldots, S_t\right]$$

  The above equation demonstrates the Markov Property as the probability $\mathbb{P}$ of the next state S_{t+1} in relation to the current state S_t, is equal to the probability of the next state if all the previous states $S_1, \ldots, S_t$ are disregarded. This encapsulates the fundamental basis of MDP, where the future state is independent of the history i.e., the current state offers sufficient statistical information to model the future [6].
- **Reward** R: The reward is a numerical scalar value relayed by the environment to the agent that defines the desirability of a completed action.

  $$\text{Total Expected Reward} = \sum_{i=1}^{T} \gamma^{i-1} r^i$$

  The discount factor γ, bounded between 0 and 1, represents the agents reward aggregation method, where $\gamma = 0$ equates to the agent employing myopic strategies that allow for immediate reward collection, whilst $\gamma = 1$ ensures delayed gratification based on the sum total of the future rewards.

In addition to the above, for continuous action spaces, the symbiosis between the state-action pairs is represented by the Q function, which is a predictor of the future reward and highlights how good an action is for the given state [6]:

$$Q^{\pi}\left(s_t, a_t\right) = \mathbb{S} \times \mathbb{A} \rightarrow \mathbb{R}$$

π is the policy which is defined as the probability distribution of actions given a state.

The value function V equals the expected total reward and is emblematic of the desirability of state-action pair [14]:

$$V^{\pi}\left(s_t\right) = \mathbb{E}\left[\sum_{i=1}^{T} \gamma^{i-1} r^i\right]$$

The optimal value function V^* for all state-action pairs is the maximisation of the above equation:

$$V^*\left(s_t\right) = \max_{\pi} V^{\pi}(s)$$

Therefore, the optimal value function is the maximum of the optimal Q function over all possible actions [14]:

$$V^*(s_t) = \max_a Q^*(s_t, a_t)$$

The optimal policy π^* correlates to the maximum value function and allows the agent to find the best possible action for the *highest* reward:

$$\therefore \pi^*(s_t) = \arg\left[\max_a Q^*(s_t, a_t)\right]$$

The advantage function is the difference between the Q function and the value function, which gives an indication of the desirability of the current policy π_t.

$$\hat{A}^t = Q^\pi(s_t, a_t) - V^\pi(S_t)$$

If the policy is not optimal ($\pi_t \neq \pi^*$), the function recognises that a certain policy must exist ($Q^\pi(s_t, a_t) < V^\pi(s_t)$) that would yield a positive reward. If the policy is optimal ($\pi_t = \pi^*$) , the function will return 0 as the there is no room for improvement ($Q^\pi(s_t, a_t) = V^\pi(s_t)$).

4.2.2 Gradient Ascent

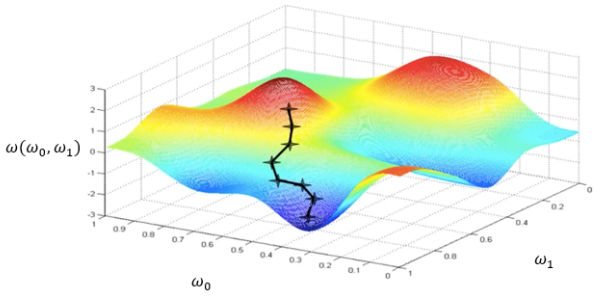


Figure 7: Gradient ascent [15]

The goal J of RL is to maximise the 'expected' reward r via a defined set of parameters ω (weight and biases) that generate a congregation of states and actions τ, whilst adhering to a policy E_π [15]:

$$J(\omega) = E_\pi[r(\tau)]$$

where,

$$E_\pi[r(\tau)] = \sum_1^m L_m(\omega)$$

$$\therefore J(\omega) = \nabla_\omega \sum_1^m L_m(\omega)$$

Evident from Fig. 7, the gradient is *added* (as opposed to gradient descent from Section 4.1.2) to propel the agent towards the global maxima of the parameter space, thereby, *maximising* the cumulative reward:

$$\omega_- \leftarrow \omega + \alpha \cdot \nabla_\omega \sum_1^m L_m(\omega)$$

Stochastic gradient ascent is an on-line algorithm, where each weight is individually and incrementally added to update the agent as new states and actions are generated. This ensures optimum exploration and decreases computational overhead.

5 Proximal Policy Optimisation

Proximal Policy Optimisation (PPO) is a first-order optimisation and on-policy algorithm that clips the policy update at each time step, thereby, limiting the deterioration of a policy or a huge deviance from a good policy. On-policy algorithms use a definitive policy (defined as the action required at each state) to ensure that the behaviour policy (objective currently used) correlates with the target policy (objective to be learnt), which translates into the agent using and consistently improving the *same* policy update (interaction with the environment). PPO leverages a surrogate clipped objective to maintain environmental desirability by ensuring the exclusion of bad policy updates and using previously obtained good policy updates to keep the agent in an optimum training zone [16]. In addition, stochastic gradient ascent is implemented that allows for the maximisation of the reward, thereby, reducing the need for immense computational resources, abstract hyper-parameter tuning, and degrading sample efficiency, whilst maintaining agent learning stability. Eq. 14 displays the mathematical formulation of PPO[1]:

$$\mathbf{L^{PPO}(\boldsymbol{\theta})} = \hat{\mathbb{E}}_{\boldsymbol{t}}[\mathbf{L^{CLIP}(\boldsymbol{\theta})} - \boldsymbol{c_1}\boldsymbol{L^{VF}(\boldsymbol{\theta})} + \boldsymbol{c_2}\boldsymbol{S}[\boldsymbol{\pi_\theta}](\boldsymbol{s_t})] \tag{14}$$

where,

- $\mathrm{L}^{\mathrm{CLIP}}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}^t, \mathrm{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}^t)]$

  - $r_t(\theta)\ \hat{A}^t$: The surrogate objective is the probability ratio between a new policy and an older policy multiplied with the advantage function, to maximise the gradient that yields a high positive advantage.

    $$\therefore r_t(\theta)\hat{A}^t = \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\right)\hat{A}^t$$

  - ϵ: The clipping policy—recommended with a value of 0.2 [16]. This ensures the agent is continuously training within the 0.8 to 1.2 range, thereby ensuring the elimination of excessive and frequent policy updates prone to deterioration.

  - $\mathrm{clip}(r_t(\theta),\ 1-\epsilon,\ 1+\epsilon)\ \hat{A}^t$: clipped version of the surrogate objective, where the probability ratio is truncated to remove unnecessary policy updates.

    $$\mathrm{clip}(\epsilon, \hat{A}^t) = \begin{cases} (1+\epsilon)\ \hat{A}^t & \hat{A}^t \geq 0 \\ (1-\epsilon)\ \hat{A}^t & \hat{A}^t < 0 \end{cases}$$

    If the advantage is positive for a given state-action pair, the clipped objective constrains the policy update, thus, curtailing the new policy update to not maximise away from the previous stable policy π_{old}.

    If the advantage is negative, the objective function limits the minimisation of the policy update, thus again, curtailing the new policy update to not drop below the previous stable policy.

[1] θ represents the parameter values of the input space, as opposed to the $\angle\theta$ from Section 3.1.

  - min[...]: "...the minimum of the clipped and unclipped objective, so the final objective is a lower bound (i.e., a pessimistic bound) on the unclipped objective...ignore the change in probability ratio when it would make the objective improve, and include it when it makes the objective worse" [16].

- $c_1 L^{VF}(\theta)$: Determines desirability of the current state, in relation to a defined hyperparameter c_1.
- $c_2 S[\pi_\theta](s_t)$: The entropy term which ensures optimum exploration of an environment using Gaussian Distribution [6], in relation to a defined hyperparameter c_2.

6 Computational Implementation

6.1 Pseudo-code[1]

Algorithm 1 PPO$^{\text{CLIP}}$

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0.
2: **for** $i = 0, 1, 2, \ldots$ **do**
3: Collect trajectories by running policy $\pi_i = \pi(\theta_i)$.
4: Compute the rewards R.
5: Compute advantage estimation via G.A.E. on the current value function.
6: Update the policy by maximising the clipped objective via gradient ascent:

$$\mathrm{L}^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}^t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}^t)]$$

7: Calculate the loss function via gradient descent, and back-propagate the error value
to fine-tune the trajectories at the next iteration.
8: **end for**

6.2 Neural Networks

Two neural networks were setup in a synchronous Advantage Actor-Critic style (`A2C`) style with a multi-layer perceptron feed-forward architecture, where the `ActorNetwork` executed the policy (policy-based), while the `CriticNetwork` was responsible for ensuring the actions taken by the actor are suitable for the defined outcome (value-based). "A [critic] in `A2C` waits for all the parallel actors to finish [the] work before updating the global parameters and then in the next iteration, parallel actors start from the same policy. The synchronised gradient update keeps the training more cohesive and potentially [makes achieving] convergence faster" [17]. Two distinct advantages of the methodology include:

1. Consider a Monte Carlo simulation where the number of possible outcomes is infinite. By defining the `A2C` style, the number of possible outcomes from a stochastic policy can be narrowed down to a set of desirable actions.

[1] The complete code is available in Appendix A.

2. The computational overhead is reduced as the critic acts as an overarching shield which restrains the actor from unnecessarily exploring new policy updates i.e., performing sub-optimal actions.

The `ActorNetwork` and `CriticNetwork` were both setup through a `PyTorch` implementation of one input layer, two hidden layers, and one output layer, all accommodated in a sequential container that allowed for the storage of the states. The input dimensions were passed through the first fully connected layer (each node connects to all nodes in the next layer), which usually applies a linear transformation to the input parameters, thereby enabling the input dimensions to match the *required* dimensions of the `PyTorch` library for forward propagation through the neural network.

However, linear transformations do not allow for complex mapping of actions and states, therefore, the next layers were designed to apply a rectified linear unit activation function (ReLU) [18]. ReLU is a monotonic activation function that simplifies the output by clipping all negative values to zero (Fig. 8), thereby acting as a linear function with the added flexibility of a non-linear function, which decreases overall training time by ensuring some neurons *do not* activate, when not needed for the output. The second hidden layer was designed with the same methodology.

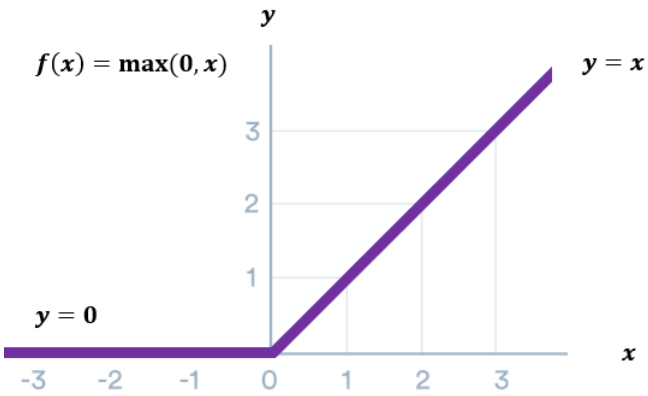


Figure 8: Rectified Linear Unit (ReLU)

The last layer for the `ActorNetwork` had an underlying softmax activation function that takes in the input dimensions and outputs a vector of probability distributions that sum to 1, thus, defining the correlation of the inputs to the eventual output. "Neural networks end in a penultimate layer which outputs real-valued scores that are not conveniently scaled, and which may be difficult to work with. Here the softmax [function] is very useful because it converts the scores to a normalised probability distribution, which can be displayed to a user or used as input to other systems" [19].

6.3 Agent

The `Agent` class defined the PPO algorithm with several features, and the default instantiation of the class was with the parameters defined in Table 3. The agent is a critical component of the overall architecture of RL that utilises the neural network(s) to determine if an action is suitable. The agent pulls in observations (Markovian state and corresponding action) from the environment and processes the observations using the policy function $\pi(a|s)$ to maximise a cumulative reward R. For the purposes of the project, an `ExperienceCollector` class was first implemented that allowed the storage of states, actions, log probabilities, values, and rewards. The aforementioned parameters were stored in batches to allow the data to seamlessly forward propagate through the neural network(s).

Table 3: Agent parameters

Parameter	**Value**
Discount factor γ	0.99
Learning rate α	$3e^{-4}$
Smoothing parameter λ	0.95
Clipping policy ϵ	0.2
Batch size	5
Number of epochs	4

The discount factor γ with a value of 0.99 ensured the delayed gratification by orientating towards a long-term reward R. The learning rate α with a value of $3e^{-4}$ displays the size of the 'step' the agent will take towards the steepest descent to minimise the loss function.

Advantage is defined to be a measure of the desirability an action for a particular state. The Generalised Advantage Estimation (G.A.E.) is an algorithm that calculates the advantage function $\hat{A}^t$ and utilises the smoothing parameter λ that is "...used for reducing the variance in training, which [in turn] makes [the training] more stable" [13].

The goal of the agent is to balance the pendulum over a long period of time and maximise the total reward by ensuring $\theta = 0$ and $\phi = 0$, over the course of the number of pre-defined games. To achieve this, the agent used the rewards r_{t+n} that were collected at each time step n and the calculated reward r_t was accumulated for all the executed actions. For example, if the agent balanced one pendulum but not the other, the algorithm would store how close the agent came to balancing the other pendulum, and then use this stored information over the course of the next iterations to provide instructive guidance to the agent. Even if the agent does not balance both pendulums at the next time step, the advantage is stored, thus ensuring a cascading effect over the remaining number of games. Mathematically, G.A.E. is described as [20]:

$$\hat{A}_t^t = \delta + \left(\gamma \cdot \lambda \cdot m_t \cdot \hat{A}_{t+1}^t\right)$$

where,

$$\delta = r_t + (\gamma \cdot V\left(s_{t+1}\right) \cdot m_t) - V(s_t)$$

where, m_t is a 'mask' value that is a numerical indicator of which state to initiate the next batch of games from, if the previous state fully balanced the pendulum i.e., *fully* restart if the previous game was successful. The clipping policy ϵ with a value of 0.2 as suggested by Schulman et al. (2017) [16], modulates the percentage change of the policy updates at each time step. The batch size, with a value of 5, shows the number of samples that need to be processed before the agent should update the learning methodology. The number of epochs, with a value of 4, denotes the number of complete passes of the *complete* training dataset through the neural network.

Lastly, the loss function was also implemented by taking the log ratio between the new policy and the old policy ($\log \pi_{\text{new}} / \log \pi_{\text{old}}$). In combination with ϵ, the clipped ratio was used to determine by how much % the policy should update at each time step. The actor loss was implemented as the minimum value between the clipped ratio and the

unclipped ratio, to maximise the positive reward returning actions. The critic loss was defined as the mean squared error of the returning states from the environment and the advantage estimation, which allows to regulate the extent of the penalisation of the actor, when straying away from good policies. The critic loss was also combined with a hyper-parameter with a value of 0.5 that ensured the penalisation of the actions was curtailed to 50% as opposed to completely disregarding an undesirable action. This reduced the computational overhead, as 50% of the action was stored, regardless of the outcome, therefore, enabling the agent to consequently use this stored information on the next time steps, instead of a completely new beginning.

6.4 Environment

"The environment is a modelled as a stochastic finite state machine with inputs (actions sent from the agent) and outputs (observations and rewards sent to the agent)" [13].

A directional limitation was first implemented, where the game would be registered as a failed attempt, if the agent accelerated the cart further than 2.4m in the left/right directions. This ensured that environment space was concise, and the agent was not trained to use infinite space as a means to stabilise the pendulums. In addition, various starting angles were explored, and as mentioned in Section 3.2.3, as the angles become larger, the chaotic behaviour of the inverted double pendulum system increases computational complexity, and therefore, angle limitation was also implemented, where, a game was registered as failed when $\theta, \phi \approx 12^\circ$ or 0.2 rad.

The states that were sent from the environment to the agent included the distance x and the corresponding linear velocity $\dot{x}$, the angles θ & ϕ and the corresponding angular velocities $\dot{\theta}$ & $\dot{\phi}$. The state information allowed the agent to adjust the corresponding actions, as required, to achieve the requisite output (Fig. 9h). The equations describing the acceleration of the system from Section 3.2.3 were used to control the action of the agent at each iteration. For example, if the linear acceleration generated the instability of the pendulums at an arbitrary iteration, the agent would store this information, and for all consequent iterations modulate the linear acceleration around this baseline to ensure the same situation does not repeat. Moreover, two auxiliary states were defined: right and left caused by a force u.

Two reward states were also defined: 0 for failing a game by surpassing the componential thresholds described previously, and 1 for successfully achieving the end state i.e., a vertically stable pendulum.

7 Results

7.1 Numerical Data

Table 4: Computational system parameters

Parameter	Value
Gravitational constant g/ms^2	9.81
Cart mass m/kg	1
Pendulum mass M/kg	0.2
Pendulum length l/m	0.50
Force magnitude u/N	10
Moment of inertia I/kgm^2	$3.33\mathrm{e}^{-3}$

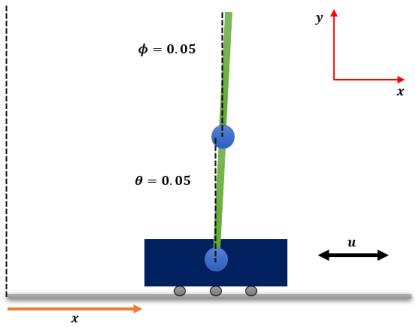
Table 5: Overall results for $10\mathrm{e}^3$ episodes

$\boldsymbol{\theta}$ (rad)	$\boldsymbol{\phi}$ (rad)	**Cumulative reward** (R)	**Total time** (min)	**Averaged accuracy** (%)
0.05	0.05	72,728	29.14	70.12
0.10	0.10	62,919	27.45	62.86
0.15	0.15	56,180	17.28	56.57
0.05	0.10	62,744	22.12	63.39
0.10	0.05	72,783	23.4	72.84
0.00	0.15	55,859	17.44	55.74
0.15	0.00	91,709	28.11	91.91

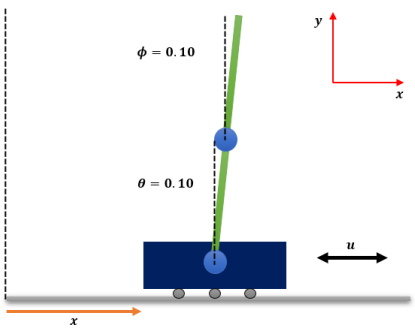
Table 6: Experimental results over various iterations

$\boldsymbol{\theta}$ (rad)	$\boldsymbol{\phi}$ (rad)	**No. of episodes**	**Reward** (r_t)	**Accuracy** (%)	**Time taken** (min)
0.05	0.05	100	720	72.00	0.14
		500	3,595	71.90	1.20
		1,000	7,213	72.13	2.31
		5,000	31,146	62.29	14.45
		10,000	72,283	72.30	29.14
0.10	0.10	100	623	62.30	0.12
		500	3,157	63.14	1.11
		1,000	6,302	63.02	2.10
		5,000	31,465	62.93	11.40
		10,000	62,919	62.92	27.45
0.15	0.15	100	563	56.30	0.10
		500	2,811	56.22	0.52
		1,000	5,636	56.36	1.58
		5,000	28,907	57.81	8.54
		10,000	56,180	56.18	17.28
0.05	0.10	100	634	63.40	0.13
		500	3,127	65.54	1.17
		1,000	6,258	62.58	2.26
		5,000	31,340	62.68	12.57
		10,000	62,744	62.74	22.12
0.10	0.05	100	734	73.40	0.14
		500	3,630	72.60	1.12
		1,000	7,279	72.79	2.28
		5,000	36,317	72.63	12.33
		10,000	72,783	72.78	23.40
0.00	0.15	100	555	55.50	0.10
		500	2,783	55.66	0.51
		1,000	5,578	55.78	1.44
		5,000	27,951	55.90	9.44
		10,000	55,859	55.86	17.44
0.15	0.00	100	922	92.20	0.19
		500	4,590	91.80	1.28
		1,000	9,200	92.00	3.26
		5,000	45,931	91.86	13.50
		10,000	91,709	91.71	28.11

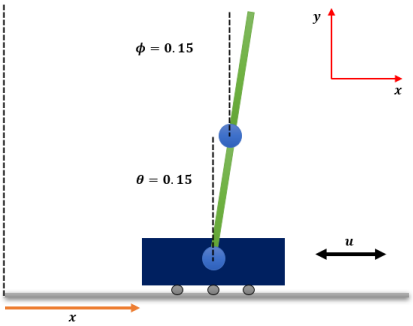
7.2 Visual Representation



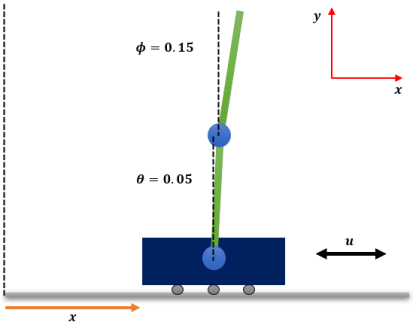
(a) $\theta = 0.05,\ \phi = 0.05$



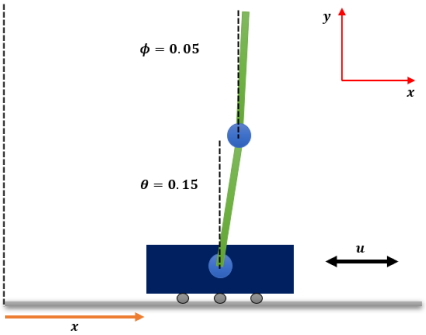
(b) $\theta = 0.10,\ \phi = 0.10$



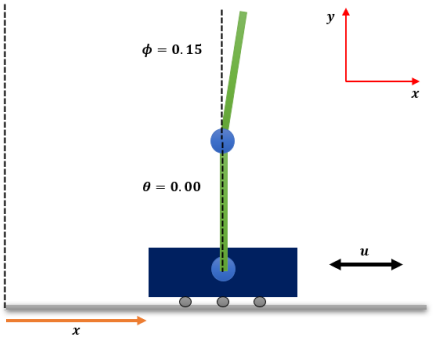
(c) $\theta = 0.15,\ \phi = 0.15$



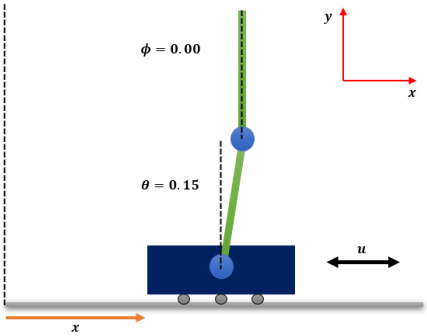
(d) $\theta = 0.05,\ \phi = 0.10$



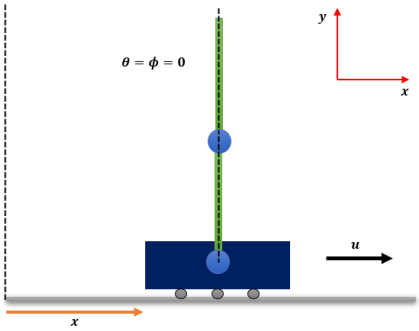
(e) $\theta = 0.10,\ \phi = 0.05$



(f) $\theta = 0.00,\ \phi = 0.15$



(g) $\theta = 0.15,\ \phi = 0.00$



(h) *Desired result*: $\theta = 0.00,\ \phi = 0.00$

Figure 9: Angle variations

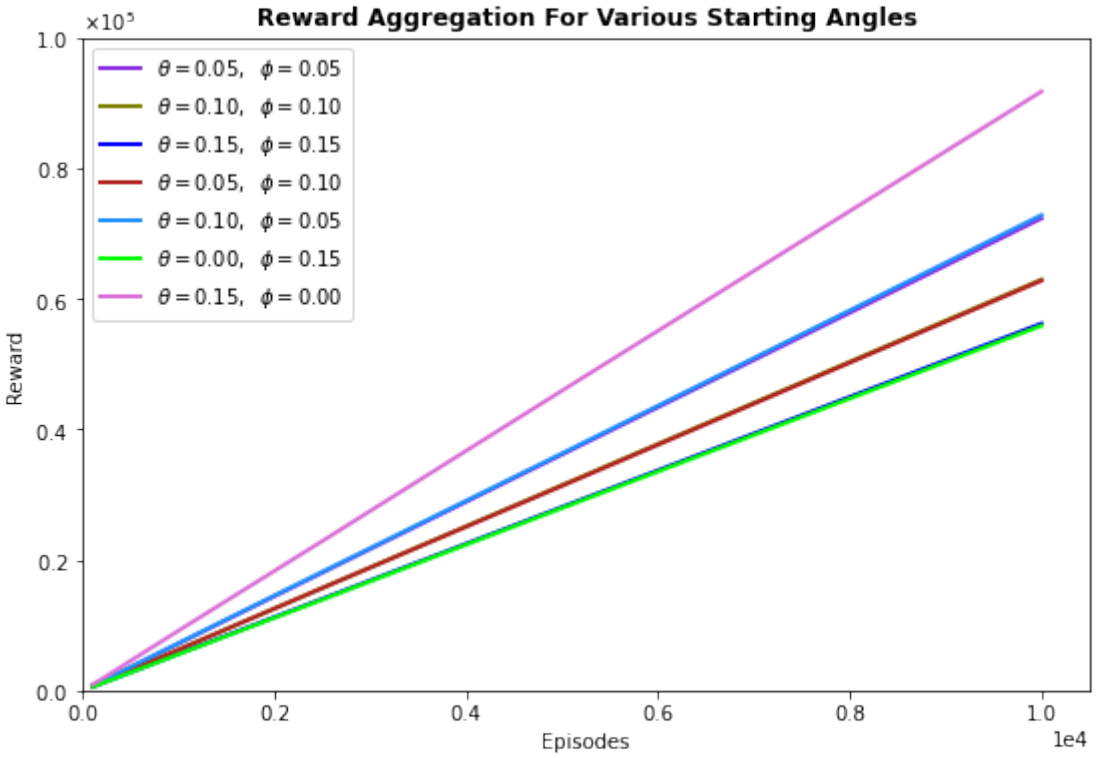


Figure 10: Reward aggregation[1]

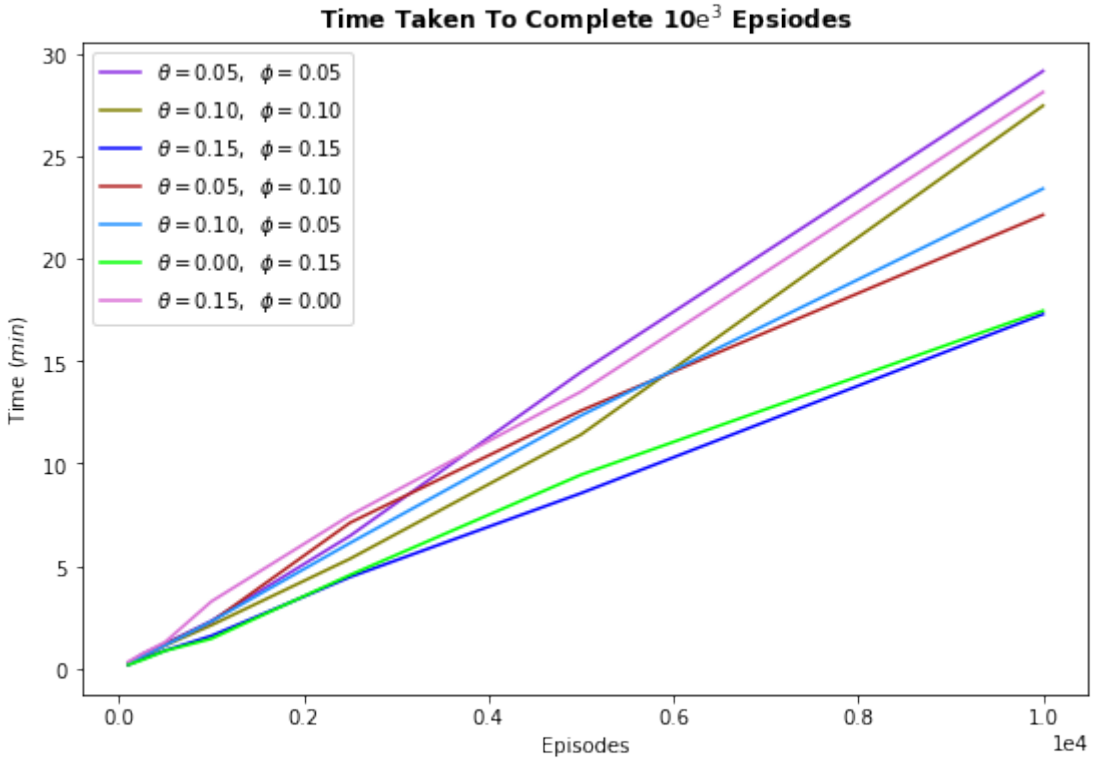


Figure 11: Time aggregation

[1]Individual graphical representation available in Appendix B.

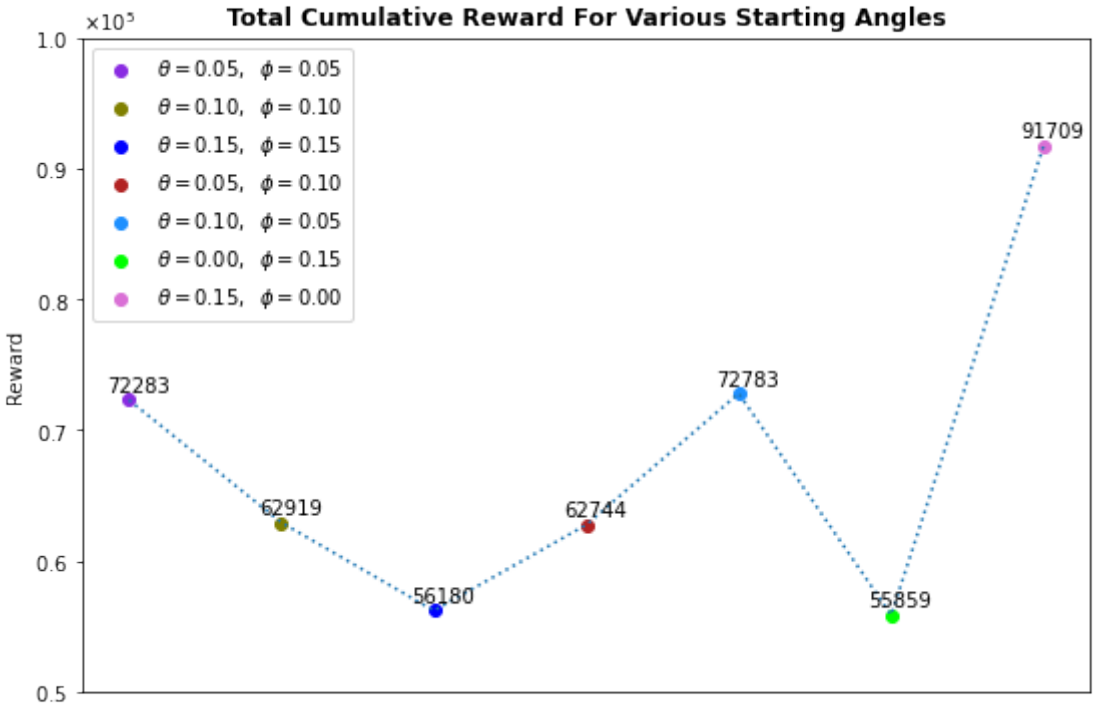


Figure 12: Cumulative rewards

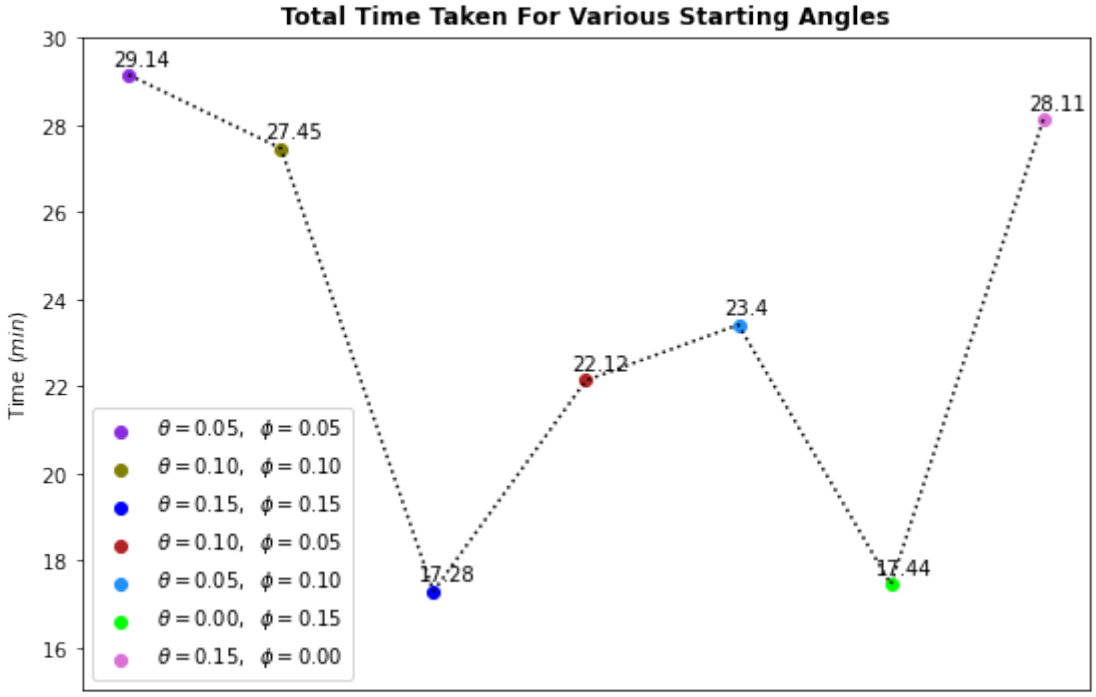


Figure 13: Total time taken

8 Discussion

The main aim of this thesis was to introduce the concept of DRL to the author and reader by designing a basic framework that extrapolates laboratory experimentation to real-world applications. The baseline model of the inverted double pendulum can be used to analyse non-linearity and chaotic mechanical systems with varying DoF. In addition, the integration of PPO, as a control system, represents a standard open-source template that can be used on a myriad of different systems by future researchers.

8.1 Analysis

Table 4 displays the initial parameters of the experiment and the corresponding values used by the algorithm to balance the system, i.e. $\theta = \phi = 0$ rad (Fig. 9h). As stated in the initial constraints of the modelling (Section 3.1), the system consisted of two pendulums with equal mass and length, and consequently, the moment of inertia was also equal with a value of $3.33e^{-3}$ kgm^2. The moment of inertia was limited to reflect the axis of the system passing through the centre of mass for both pendulums, which stipulates the fact that the pendulums were assumed to be infinitely thin and rigid, and have constant mass density in the middle. This transformed into another simplification technique designed to reduce the complexity of evaluating a chaotic system mathematically and computationally. In addition, the force applied was 10 N depending on the reward feedback, for example, if the force was applied to the left side of cart and the system overshot the angles of the pendulums away from the desired outcome, then the agent was designed to apply a negative force of the same magnitude, which translates into the application of the same force magnitude, but from the *right* side to offset the overshooting effects of the previous force application. Moreover, the cart was always heavier than the two pendulums to ensure a stabilised platform of action onto which the force could act, and to offset the complexity associated with increasing the mass of the pendulums more than the cart.

A total of $10e^3$ episodes were played for *seven* variations of starting angles, and Table 6 shows the change in the angles of the pendulums per episode with the corresponding cumulative reward, and total time taken to execute the episodes. It is important to note that each specific episode comprised of 10 individual iterations of the game, so in total the game was played for $10e^6$ times and the total reward reflects the *individual* reward for each episode.

From Table 6, the first three angle variation combinations were incrementally increased equally for both the angles, θ and ϕ. The first iteration, where $\theta = \phi = 0.05$ rad displays the highest accuracy, and evident from Fig. 9a to 9c, as the angles are increased simultaneously for both pendulums, the accuracy *deteriorates*. As previously mentioned in Section 2, the system is highly chaotic and is appropriately described as experiencing the *butterfly effect*, where the initial conditions determine the complexity of controlling the system. "Sensitive dependence on initial conditions is the hallmark of chaos. This means that tiny separations $\delta x(t_0)$ between nearby initial conditions are amplified exponentially in time t:

$$\delta x(t) \backsim \delta x(t_0) e^{\lambda t}$$

where, $\delta x(t)$ is the separation between nearby trajectories and λ is a positive constant" [21].

The larger the starting angle, the more modulation of the force on the cart will be required to balance the pendulums. Subsequently, as the angles are changed to an uneven distribution between θ and ϕ, the accuracy increases only when $\phi < \theta$. Fig. 9f and 9g demonstrates this with the initial starting angles of the system, which when compared with Graph 12, displays the reduction in the cumulative reward when $\phi > \theta$. Moreover, the highest cumulative reward was observed for the $\theta = 0.15$ rad and $\phi = 0.00$ rad angle variation, therefore reinforcing the validity of the aforementioned statement.

From Graph 10, the linearity between the aggregated reward for each angle variation with increasing number of episodes is displayed. One of the ways to gauge how effective a RL model is to ensure that the agent returns a positive *linear* relationship between the reward aggregation and the number of executed episodes i.e., the model learns and sustains an accumulation in reward aggregation. Additionally, Graph 13 displays the total time that was taken by the agent to execute $10e^3$ episodes. The total time taken is emblematic of how long it took the agent to effectively contain the chaotic behaviour of the double pendulums, given an initial set of starting angle conditions. The longest time taken by the agent was for the $\theta = \phi = 0.05$ rad angle variation, and the shortest time taken was for when $\theta = \phi = 0.15$ rad. This might seem counter-intuitive to what has been just stipulated, but in fact, reinforces the point about the containment of chaotic behaviour. As mentioned in Section 6.4, the agent was designed to fail an episode if the cart moved 2.4m away from the starting position, and when θ & $\phi \geq$ 0.20 rad. It is true that the control of the pendulum system when started with larger angles would require an increasing number of computations, but it is important to note that the failing criteria is also *quicker* to be attained. The closer the system is to the failing criteria, the faster it will fail episodes and therefore, decrease the overall total time taken for the angle variations as the agent would fail the episode, and simply move to the next iteration.

The highest cumulative reward was recorded for when $\boldsymbol{\theta = 0.15}$ **rad** and $\boldsymbol{\phi = 0.00}$ **rad** with a value of **91,709** and an average accuracy of **91.91%**. This translates into the agent being able to balance the inverted double pendulum system approximately 9/10 games per episode. As mentioned previously, the most reward was accumulated in episodes where $\phi < \theta$, and the highest reward gained in any angle variation, again reiterates this point. If the reader imagines an inverted double pendulum with the starting position as $\theta = 0.15$ rad and $\phi = 0.00$ rad (Fig. 9g), one can immediately infer that balancing the bottom pendulum is far easier than balancing the top pendulum, as the distance between the horizontal force component u and ϕ is larger than between u and θ. The force applied to the cart causes larger perturbations in the angular acceleration of ϕ as the angular acceleration $\ddot{\phi}$ depends on $\ddot{\theta}$. Therefore, a small variation in the acceleration of the cart $\ddot{x}$ would result in large local disturbances for $\ddot{\theta}$, and thereby, accentuate the angular displacement for $\ddot{\phi}$.

8.2 Suggested Improvements

The experiment was modelled to simulate real-world robotic applications, however, several assumptions were added to the overall system to induce modelling simplicity, as the complexity grew throughout the testing phase. These assumptions have been addressed in this section and suggested improvements have been highlighted to increase the applicability of the model on mission-, business-, and safety-critical space engineering operations.

8.2.1 Adaptive Learning Rate

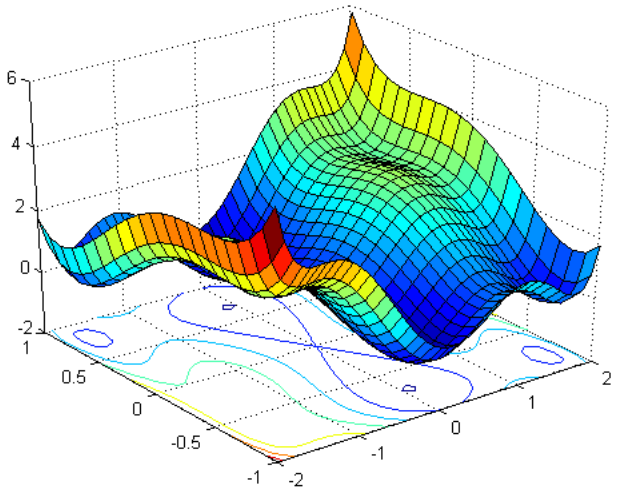


Figure 14: Complex gradient descent

The learning rate α was kept constant throughout the testing phase with a value of $3e^{-4}$, which determines the size of the step towards the steepest descent in the gradient descent algorithm used to minimise the loss function in the neural networks (Section 4.1.2). In essence, "The learning rate hyper-parameter controls the rate or speed at which the model learns. Specifically, it controls the amount of apportioned error that the weights of the model are updated with each time they are updated, such as at the end of each batch of training examples" [18]. It is imperative to note that striking the right balance between a slow or fast learning rate is key to optimising the neural networks, as a slow rate would increase computational overheard and the training loss, while a fast rate would cause diverging behaviour, where the gradient would induce a large update to the weights causing numerical overflow. This ties in closely with the problem of the gradient descent algorithm getting stuck in *local* minima, as opposed to travelling to the *global* minima. Fig. 14 demonstrates a complex gradient with multiple local minima, therefore allowing the reader to truly appreciate the complexity of navigating real-world modelling problems.

To offset the rigidity of the fixed learning rate, an Adaptive Learning Rate (ALR) can be introduced to the model. The ALR can be used to reduce the learning rate once the model plateaus, thereby, increasing the chances of learning by reducing the rate of learning, effectively forcing the model to analyse the data closely, whilst employing the knowledge gained from previous iterations.

8.2.2 Dying ReLU

The dying rectified linear unit (ReLU) refers to the problem when certain nodes in the neural network become dormant, and only output zero for *any* given input. As discussed in Section 6.2, the ReLU activation function clips *all* possible negative values to zero, thereby enabling linear and non-linear feature extraction. "A fundamental difficulty in training deep neural networks is the vanishing and exploding gradient problem. The dying ReLU is a kind of vanishing gradient, which refers to a problem when ReLU neurons become inactive and only output 0 for any input. It has been known as one of the obstacles in training deep feed-forward ReLU neural networks. To overcome this problem, a number of methods have been proposed. . .one approach modifies the network architectures. This includes but is not limited to the changes in the number of layers, the number of neurons, network connections, and activation functions. In particular, many activation functions have been proposed to replace the ReLU. However, the performance of other activation

functions varies on different tasks and data sets and it typically requires a parameter to be turned. Thus, the ReLU remains one of the popular activation functions due to its simplicity and reliability" [22]. A possible remedy is to use a *randomised asymmetric initialisation*, as opposed to stochastically choosing symmetric probability distributions for the initialisation of the neural network.

8.2.3 Kullback–Leibler (KL) Divergence

The PPO version used in this study performed the RL via the clipped surrogate objective as described in Section 5. However, the original paper by Schulman et al. [16], suggests an adaptive penalty co-efficient on the KL divergence that allows the algorithm to compute the probability distribution using a *reference* distribution. The nuances of KL divergence are beyond the scope of this thesis, but worth mentioning for incorporation into the future study of PPO.

8.2.4 Reward Shaping

RL algorithms are time-consuming, as knowledge is developed through the ratiocination of various action-state pairs, and collected observations from the environment. "Reward shaping is a method of incorporating domain knowledge into [RL], so that the algorithms are guided faster towards more promising solutions" [23]. Essentially, the incorporation of *variable* reward impartation, as opposed to a *fixed* reward of 0 and 1, as defined in this thesis, would allow the model to train faster, where an increasing/decreasing reward would provide an indicative guide for the agent to gauge action desirability.

8.2.5 Environment Setup

The author implemented a *new* environment by leveraging the vanilla `Cart-Pole-v0` environment setup by Sutton et al. (1983) [24]. The environment was manipulated to include the inverted double pendulum, where the states being returned were changed to reflect the addition of the added pendulum. The environment was verified through extensive testing and exhibited similar learning patterns as when a pre-built environment was observed, such as the OpenAI Gym. One of the main problems of RL is the relatively new nascence of the paradigm which is prone to non-standardised methodologies. Using a pre-built environment that has been verified through exhaustive testing with the watchful policy of trained academics should be the natural selection for anyone trying to solve a DRL problem. However, the author wished to understand the intricate nature of the problem, in addition to not being able to install the `MujoCo` library environment (`InvertedDoublePendulum-v2`) due to obtuse installation guidance and license requirement. Therefore, the environment was implemented from first principles, which is indeed ambitious, but highly prone to insidious software bugs. Future study should consider using a pre-built environment to easily transfer the algorithm between different environments and allow a reduction in the overall development time. Additionally, the standard number of episodes for verifying a reinforcement learning problem should be in the $> 10e^3$ domain. The author ran the experiment for a maximum of $10e^3$ episodes that is reflective of only the *minimum* threshold, and further study should consider the usage of a GPU to increase training capacity, whilst decreasing training time.

8.2.6 Simplifying Assumptions

The assumptions used for the model such as $M_1 = M_2 = M$ and $l_1 = l_2 = l$, simply do not reflect real-world applications. Additionally, the aspect of the pendulum starting from a set of initial angle conditions is improbable, where the pendulums hanging with a swing-up required is far more realistic. To reduce the complexity of the governing equations of motion, due to the lack of published mathematical modelling literature on an inverted double pendulum on a cart, the author introduced these assumptions to reduce development time and the pernicious effects of human error whilst solving complex differential equations. Indeed, every experiment that is carried out cannot fully simulate real-world conditions, however, caution should be taken when applying simplifications and processes must be put in place to ensure that the assumptions are *reasonable* and do not cause an enormous divergence of results recorded in an experimental setting, as opposed to real-world engineering.

9 Conclusion

The aim of the thesis was to develop, firstly, an understanding of the DRL concept through a rigorous study of the inverted double pendulum, and secondly, build a open-source template for enabling the study of autonomous control systems accessible at all levels of academia. The control system was designed to balance both the pendulums to achieve equilibrium, i.e., $\theta = 0$ and $\phi = 0$, whilst ensuring the effective modulation of various system-specific parameters. The neural networks were implemented to direct the PPO algorithm to enable optimal actions that transformed into a maximised cumulative reward. Various reward aggregations for several starting angle conditions were recorded to verify the validity of the model. The highest accuracy and cumulative rewards were recorded when $\theta > \phi$, therefore, establishing the relationship between the force on the cart, and the angular displacements of the first pendulum and the second pendulum. Furthermore, various improvements were also suggested for future study that should be considered to optimise the development of truly autonomous control systems.

References

[1] Andrew Harris, Thibaud Teil, and Hanspeter Schaub. "Spacecraft decision-making autonomy using deep reinforcement learning". In: *29th AAS/AIAA Space Flight Mechanics Meeting, Hawaii*. 2019, pp. 1–19 (page 1).

[2] Raymond A. Serway. *Principles of Physics*. 2nd ed. Saunders College Pub., 1998 (page 3).

[3] Francesco Bullo and Andrew D. Lewis. *Geometric Control of Mechanical Systems*. Vol. 49. Texts in Applied Mathematics. New York-Heidelberg-Berlin: Springer Verlag, 2004. ISBN: 0-387-22195-6 (page 3).

[4] NS Manton. "The Principle of Least Action in Dynamics". In: (2013) (page 5).

[5] S Widnall. *Lecture L20 - Energy Methods: Lagrange's Equations*. 3rd ed. Massachusetts Institute of Technology (MIT), 2009, p. 6. URL: https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-07-dynamics-fall-2009/lecture-notes/MIT16_07F09_Lec20.pdf (page 7).

[6] David Silver. *RL Course by David Silver - Lecture 2: Markov Decision Process*. May 2015. URL: %5Curl%7Bhttps://www.youtube.com/watch?v=lfHX2hHRMVQ%7D (pages 8, 11, 14).

[7] Christopher M. Bishop. *Pattern recognition and machine learning*. Springer, 2016 (page 8).

[8] *Weights and Biases*. URL: https://docs.paperspace.com/machine-learning/wiki/weights-and-biases (page 8).

[9] Nate Kohl. *What is the role of the bias in neural networks?* Mar. 2010. URL: https://stackoverflow.com/questions/2480650/what-is-the-role-of-the-bias-in-neural-networks (page 9).

[10] *Gradient Descent*. 2017. URL: https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html (page 9).

[11] Ayoosh Kathuria. *Intro to Optimization in Deep Learning: Gradient Descent*. Dec. 2020. URL: https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/ (page 10).

[12] Martijn van Otterlo and Marco Wiering. "Reinforcement Learning and Markov Decision Processes". In: *Reinforcement Learning: State-of-the-Art*. Ed. by Marco Wiering and Martijn van Otterlo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–42. ISBN: 978-3-642-27645-3. DOI: 10.1007/978-3-642-27645-3_1. URL: https://doi.org/10.1007/978-3-642-27645-3_1 (page 10).

[13] 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 (pages 10, 11, 16, 17).

[14] Moustafa Alzantot. *Deep Reinforcement Learning Demystified (Episode 2)*. Oct. 2018. URL: https://medium.com/@m.alzantot/deep-reinforcement-learning-demysitifed-episode-2-policy-iteration-value-iteration-and-q-978f9e89ddaa (pages 11, 12).

[15] Sanyam Kapoor. *Policy Gradients in a Nutshell*. URL: https://towardsdatascience.com/policy-gradients-in-a-nutshell-8b72f9743c5d (page 12).

[16] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347 (pages 13, 14, 16, 26).

[17] Lilian Weng. *Policy Gradient Algorithms*. Apr. 2018. URL: https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html (page 14).
[18] Jason Brownlee. *A Gentle Introduction to the Rectified Linear Unit (ReLU)*. Jan. 2019. URL: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/ (pages 15, 25).
[19] Thomas Wood. *Softmax Function*. May 2019. URL: https://deepai.org/machine-learning-glossary-and-terms/softmax-layer (page 15).
[20] Chintan Trivedi. *Proximal Policy Optimization Tutorial (Part 2/2: GAE and PPO loss)*. Sept. 2019. URL: https://towardsdatascience.com/proximal-policy-optimization-tutorial-part-2-2-gae-and-ppo-loss-fe1b3c5549e8 (page 16).
[21] Troy Shinbrot, Celso A Grebogi, Jack Wisdom, and James A Yorke. "Chaos in a double Pendulum". In: *American Journal of Physics* 60 (1992), pp. 491–491. DOI: 10.1119/1.16860. URL: https://hal.archives-ouvertes.fr/hal-01403609 (page 23).
[22] Lu Lu. "Dying ReLU and Initialization: Theory and Numerical Examples". In: *Communications in Computational Physics* 28.5 (June 2020), pp. 1671–1706. ISSN: 1991-7120. DOI: 10.4208/cicp.oa-2020-0165. URL: http://dx.doi.org/10.4208/cicp.OA-2020-0165 (page 26).
[23] Marek Grzes. "Reward shaping in episodic reinforcement learning". In: (2017) (page 26).
[24] A. G. Barto, R. S. Sutton, and C. W. Anderson. "Neuronlike adaptive elements that can solve difficult learning control problems". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13.5 (1983), pp. 834–846. DOI: 10.1109/TSMC.1983.6313077 (page 26).

Appendix

A Computational Code

A.1 Imports

```
# mathematical
from sympy import symbols, Function, cos, sin, Eq, linsolve
from sympy.physics.mechanics import init_vprinting

# computational
import matplotlib.pyplot as plt
import math
import time
import gym
import seeding
import numpy as np
import torch as T
import torch.nn as nn
import torch.optim as optim
from torch.distributions.categorical import Categorical
```

A.2 Modelling

```
# format mathematical output
init_vprinting()

# initiliase variables
t = symbols('t')
m = symbols('m')
l = symbols('l')
M = symbols('M')
I = symbols('I')
g = symbols('g')
u = symbols('u')

x = Function(x)(t)
θ = Function(θ)(t)
ϕ = Function(ϕ)(t)

# cart
ẋ = x.diff(t)

# pendulum(s)
θ̇ = θ.diff(t)
θ̈ = θ̇_dot.diff(t)
ϕ̇ = ϕ.diff(t)
ϕ̈ = ϕ̇_dot.diff(t)
cos_theta = cos(θ)
sin_theta = sin(θ)
cos_thetaphi = cos(θ - ϕ)
cos_phi = cos(ϕ)
sin_phi = sin(ϕ)

# kinetic energy components
# cart - linear
```

```
k_1 = m*x_dot**2

# pendulum(s) - angular
k_2 = (M*ẋ*(ẋ + 2*l*θ̇*cos_theta) + θ̇**2*(M*(l**2)+I))
k_3 = (x_dot**2) + (l**2*θ̇**2) + (ϕ̇**2*(M*(l**2)+I) \
       + (M*l*θ̇*ϕ̇*cos_thetaphi) + \
       (2*M*l*ẋ*((θ̇*cos_theta) + (ϕ̇*cos_phi))))
# total kinetic energy
K = 0.5*(k_1 + k_2 + k_3)

# total potential energy
P = M*g*l*((2*cos_theta) + cos_phi)

# the lagrangian
L = K - P

# euler-lagrange formulation
"""
`expand()`: allows cancellation of like terms
`collect()`: collects common powers of a term in an expression
"""
euler_1 = Eq((L.diff(ẋ).diff(t) - \
L.diff(x)).simplify().expand().collect(x.diff(t, t)), u)

euler_2 = Eq((L.diff(θ̇).diff(t) - \
L.diff(θ)).simplify().expand().collect(θ.diff(t, t)), 0)

euler_3 = Eq((L.diff(ϕ̇).diff(t) - \
L.diff(ϕ)).simplify().expand().collect(ϕ.diff(t, t)), 0)

# linearise the system
matrix = [(sin_theta, θ), (cos_theta, 1), (θ̇**2, 0),
          (sin_phi, ϕ), (cos_phi, 1), (ϕ̇**2, 0),
          (sympy.sin(θ - ϕ), θ - ϕ), (sympy.cos(θ - ϕ), 1)]

linear_1 = euler_1.subs(matrix)
linear_2 = euler_2.subs(matrix)
linear_3 = euler_3.subs(matrix)

# simplify for linear and angular acceleration
final_equations = linsolve([linear_1, linear_2, linear_3], \
[x.diff(t, t), θ.diff(t, t), ϕ.diff(t, t)])

x_ddot = final_equations.args[0][0].expand().collect((θ, θ̇, \
x, ẋ, ϕ, ϕ̇, u, M, m, l, I)).simplify()
θ_ddot = final_equations.args[0][1].expand().collect((θ, θ̇, \
x, ẋ, ϕ, ϕ̇, u, M, m, l, I)).simplify()
ϕ_ddot = final_equations.args[0][2].expand().collect((θ, θ̇, \
x, ẋ, ϕ, ϕ̇, u, M, m, l, I)).simplify()
```

A.3 Neural Networks

```
class ActorNetwork(nn.Module):
    # constructor
    def __init__(self,
                 num_actions,
                 input_dimensions, learning_rate_alpha,
```

```
                fully_connected_layer_1_dimensions=256,
                fully_connected_layer_2_dimensions=256):

        # call super-constructor
        super(ActorNetwork, self).__init__()

        # neural network setup
        self.actor = nn.Sequential(
                # linear layers unpack input_dimensions
                nn.Linear(*input_dimensions, fully_connected_layer_1_dimensions),
                # ReLU: applies the rectified linear unit function element-wise
                nn.ReLU(),
                nn.Linear(fully_connected_layer_1_dimensions,
                            fully_connected_layer_2_dimensions),
                nn.ReLU(),
                nn.Linear(fully_connected_layer_2_dimensions,
                            num_actions),

                # softmax activation function: a mathematical function
                # that converts a vector of numbers
                # into a vector of probabilities, where the
                # probabilities of each value are proportional to the
                # relative scale of each value in the vector.
                nn.Softmax(dim=-1)
        )

        # optimizer: an optimization algorithm that
        # can be used instead of the classical stochastic
        # gradient descent procedure to update network
        # weights iterative based in training data
        self.optimizer = optim.Adam(self.parameters(),
                            lr=learning_rate_alpha)
        # handle type of device
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')
        self.to(self.device)
    # pass state forward through the NN: calculate
    # series of probabilities to draw from a distribution
    # to get actual action. Use action to get log
    # probabilities for the calculation of the two probablities
    # for the learning function
    def forward(self, state):
        dist = self.actor(state)
        dist = Categorical(dist)
        return dist

# [NOTE: See the above comments in the `ActorNetwork` for
        # individual function explanation]
class CriticNetwork(nn.Module):
    def __init__(self,
                input_dimensions,
                learning_rate_alpha,
                fully_connected_layer_1_dimensions=256,
                fully_connected_layer_2_dimensions=256):

        super(CriticNetwork, self).__init__()

        self.critic = nn.Sequential(
                nn.Linear(*input_dimensions,
```

```
                            fully_connected_layer_1_dimensions),
                nn.ReLU(),
                nn.Linear(fully_connected_layer_1_dimensions,
                            fully_connected_layer_2_dimensions),
                nn.ReLU(),
                nn.Linear(fully_connected_layer_2_dimensions, 1)
        )

        # same learning rate for both actor & critic ->
        # actor is much more sensitive to the changes in the underlying

        # parameters
        self.optimizer = optim.Adam(self.parameters(), lr=learning_rate_alpha)
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')
        self.to(self.device)

    def forward(self, state):
        value = self.critic(state)
        return value
```

A.4 Agent

```
# storage class
class ExperienceCollector:
    # constructor - init values to empty lists
    def __init__(self, batch_size):
        self.states_encountered = []
        self.probability = []
        self.values = []
        self.actions = []
        self.rewards = []
        self.terminal_flag = []

        self.batch_size = batch_size

    # generate batches - defines the number of samples
    # that will be propagated through the network
    def generate_batches(self):
        num_states = len(self.states_encountered)
        batch_start = np.arange(0, num_states, self.batch_size)
        idx = np.arange(num_states, dtype=np.int64)
        np.random.shuffle(idx) # shuffle to handle stochastic gradient descent
        batches = [idx[i:i+self.batch_size] for i in batch_start]

        # NOTE: maintain return order
        return np.array(self.states_encountered),\
                np.array(self.actions),\
                np.array(self.probability),\
                np.array(self.values),\
                np.array(self.rewards),\
                np.array(self.terminal_flag),\
                batches

    # store results from previous state
    def memory_storage(self,
                       states_encountered,
                       action,
```

```
                        probability,
                        values,
                        reward,
                        terminal_flag):
        self.states_encountered.append(states_encountered)
        self.actions.append(action)
        self.probability.append(probability)
        self.values.append(values)
        self.rewards.append(reward)
        self.terminal_flag.append(terminal_flag)

    # clear memory after retrieving state
    def memory_clear(self):
        self.states_encountered = []
        self.probability = []
        self.actions = []
        self.rewards = []
        self.terminal_flag = []
        self.values = []

# defines the agent
class Agent:
    def __init__(self,
                 num_actions,
                 input_dimensions,
                 gamma=0.99,
                 learning_rate_alpha=3e-4,
                 gae_lambda=0.95,
                 policy_clip=0.2,
                 batch_size=64,
                 num_epochs=10):
        # save parameters
        self.gamma = gamma
        self.policy_clip = policy_clip
        self.num_epochs = num_epochs
        self.gae_lambda = gae_lambda

        self.actor = ActorNetwork(num_actions,
                                  input_dimensions,
                                  learning_rate_alpha)
        self.critic = CriticNetwork(input_dimensions, learning_rate_alpha)
        self.memory = ExperienceCollector(batch_size)

    # store memory; interface function
    def interface_agent_memory(self,
                               state,
                               action,
                               probability,
                               values,
                               reward,
                               terminal_flag):
        self.memory.memory_storage(state,
                                   action,
                                   probability,
                                   values,
                                   reward,
                                   terminal_flag)

```

```
94
95      # choosing an action
96      def action_choice(self, observation):
97          # convert numpy array to a tensor
98          state = T.tensor([observation], dtype=T.float).to(self.actor.device)
99
100         # distribution for choosing an action
101         dist = self.actor(state)
102         # value of the state
103         value = self.critic(state)
104         # sample distribution to get action
105         action = dist.sample()
106
107         # squeeze to eliminate batch dimensions
108         probability = T.squeeze(dist.log_prob(action)).item()
109         action = T.squeeze(action).item()
110         value = T.squeeze(value).item()
111
112         return action, probability, value
113
114     # learning from actions
115     def learn(self):
116         # iterate over the number of epochs
117         for _ in range(self.num_epochs):
118             state_array, action_array, old_probability_array, values_array,\
119             reward_array, terminal_flag_array, batches = \
120                     self.memory.generate_batches()
121
122             values = values_array
123             # advantage
124             advantage = np.zeros(len(reward_array), dtype=np.float32)
125
126             # calculate advantage
127             for time_step in range(len(reward_array)-1):
128                 discount = 1
129                 advantage_time_step = 0
130                 # from Schulman paper -> the advantage function
131                 for k in range(time_step, len(reward_array)-1):
132                     advantage_time_step +=
133                         discount*(reward_array[k] + self.gamma*values[k+1]*\
134                             (1-int(terminal_flag_array[k])) - values[k])
135                     # multiplicative factor
136                     discount *= self.gamma*self.gae_lambda
137                 advantage[time_step] = advantage_time_step
138             # turn advantage into tensor
139             advantage = T.tensor(advantage).to(self.actor.device)
140
141             # convert values to a tensor
142             values = T.tensor(values).to(self.actor.device)
143             for batch in batches:
144                 states = T.tensor(state_array[batch], dtype=T.float)
145                                         .to(self.actor.device)
146                 old_probability = T.tensor(old_probability_array[batch])
147                                         .to(self.actor.device)
148                 actions = T.tensor(action_array[batch]).to(self.actor.device)
149
150                 # pi(theta)_new: take states and pass to Actor to get
151                 # the distribution for new probability
```

```
                dist = self.actor(states)
                critic_value = self.critic(states)
                # new values of the state according to the Critic network
                critic_value = T.squeeze(critic_value)
                # calculate new probability
                new_probability = dist.log_prob(actions)
                # probability ratio; probabilities taken
                # as exponential to get ratio
                probability_ratio = new_probability.exp() /
                                                    old_probability.exp()
                weighted_probability = advantage[batch] * probability_ratio
                weighted_clipped_probability = T.clamp(probability_ratio
                            , 1-self.policy_clip, 1+self.policy_clip)*
                                                        advantage[batch]
                # negative due to gradient ascent
                actor_loss = -T.min(weighted_probability,
                                        weighted_clipped_probability).mean()
                returns = advantage[batch] + values[batch]
                critic_loss = (returns-critic_value)**2
                critic_loss = critic_loss.mean()
                total_loss = actor_loss + 0.5*critic_loss
                # zero the gradients
                self.actor.optimizer.zero_grad()
                self.critic.optimizer.zero_grad()
                # back-propagate total loss
                total_loss.backward()
                self.actor.optimizer.step()
                self.critic.optimizer.step()
        # at end of epochs clear memory
        self.memory.memory_clear()
```

A.5 Environment

```
"""
Adapted from the classic cart-pole system implemented by Rich Sutton et al.
Original `CartPole-v0` environment available here:
                http://incompleteideas.net/sutton/book/code/pole.c
Permalink: https://perma.cc/C9ZM-652R
"""
class DoubleInvertedPendulum(gym.Env):
    def __init__(self):
        self.gravity = 9.81                                             # m/s^2
        self.mass_cart = 1.0                                            # kg
        self.masspole_1 = 0.1                                           # kg
        self.masspole_2 = 0.1                                           # kg
        self.mass_pole = (self.masspole_1 + self.masspole_2)            # kg
        self.lengthpole_1 = 0.25                                        # m
        self.lengthpole_2 = 0.25                                        # m
        self.length = (self.lengthpole_1 + self.lengthpole_2)           # m
        self.force = 10.0                                               # N
        self.moment_inertia = (self.mass_pole*self.length**2) / 12      # kgm^2
        self.tau = 0.02                                                 # s
        self.kinematics_integrator = 'euler'
        self.theta = 0.05
        self.phi = 0.05

        # angle at which to fail the episode - ~ 0.2 rad, ~12 deg
```

```
        self.theta_threshold_radians = 12 * 2 * math.pi / 360
        self.phi_threshold_radians = 12 * 2 * math.pi / 360

        # distance of cart to fail episode
        self.x_threshold = 2.4

        # angle limit set to 2 * theta_threshold_radians so failing observation
        # is still within bounds.
        high = np.array([self.x_threshold * 2,
                         np.finfo(np.float32).max,
                         self.theta_threshold_radians * 2,
                         np.finfo(np.float32).max,
                         self.phi_threshold_radians * 2,
                         np.finfo(np.float32).max],
                        dtype=np.float32)

        self.action_space = gym.spaces.Discrete(2)
        self.observation_space = gym.spaces.Box(-high, high, dtype=np.float32)

        self.seed()
        self.viewer = None
        self.state = None
        self.steps_beyond_done = None

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

    def step(self, action):
        err_msg = "%r (%s) invalid" % (action, type(action))
        assert self.action_space.contains(action), err_msg

        x, x_dot, theta, theta_dot, phi, phi_dot = self.state
        force = self.force if action == 1 else -self.force
        ################################
        # denominator
        denominator_X = (self.moment_inertia**2) +
                        (self.moment_inertia*self.length**2) + \
        (self.moment_inertia*self.mass_cart*\
        (self.moment_inertia+self.length**2)) - \
        (self.mass_pole**3*self.length**2*\
        (4*self.length**2-2*self.length+0.25)) - \
        (self.mass_pole**2*self.length**2*(3*self.moment_inertia-\
        self.length**2*self.mass_cart-self.length**2\
            +0.25*self.mass_cart+0.25)) + \
        (self.mass_pole*(self.moment_inertia**2+2*self.moment_inertia*\
            self.length**2*self.mass_cart\
                +3*self.moment_inertia*self.length**2+self.length**4*\
                    self.mass_cart+self.length**4))

        # x-acceleration numerator
        numerator_x_acc = (force*(self.moment_inertia**2\
        +2*self.moment_inertia*\
        self.mass_pole*self.length**2\
            +self.moment_inertia*self.length**2+ \
             self.mass_pole**2*self.length**4-0.25*self.mass_pole**2\
             *self.length**2+self.mass_pole*self.length**4)) - \
        (self.mass_pole**2*self.gravity*self.length**2*\
```

```
            (4*self.moment_inertia+4*self.mass_pole**2\
                -self.mass_pole*self.length)*self.theta) - \
            (self.mass_pole**2*self.gravity*self.length**2*\
            (self.moment_inertia+self.mass_pole*self.length**2\
                -self.mass_pole*self.length+self.length**2)*self.phi)
        # final x-acceleration
        x_acc = numerator_x_acc / denominator_X

        # theta-acceleration numerator
        numerator_theta_acc = (self.mass_pole*self.length*\
            (-self.mass_pole*self.gravity*self.length*\
                 (-2*self.mass_pole*self.length+0.5*self.mass_pole\
                 +0.5*self.mass_cart+0.2)*self.phi) + \
            2*self.gravity*(self.moment_inertia*self.mass_pole+\
                self.moment_inertia*self.mass_cart\ +self.moment_inertia+\
            self.mass_pole*self.mass_cart*self.length**2+self.mass_pole*\
            self.length**2)*self.theta - force*(2*self.moment_inertia\
                +2*self.mass_pole*self.length**2-0.5*self.mass_pole*self.length))
        # final theta-acceleration
        theta_acc = numerator_theta_acc / denominator_X

        # phi-acceleration numerator
        numerator_phi_acc = (self.mass_pole*self.length*\
        (-self.mass_pole*self.gravity*self.length*(-4*self.mass_pole\
            *self.length+self.mass_pole+self.mass_cart+1)*self.theta) + \
                        (self.gravity*(self.moment_inertia*self.mass_pole\
                        +self.moment_inertia*self.mass_cart\
                    +self.moment_inertia-3*self.mass_pole**2*self.length**2+\
        self.mass_pole*self.length**2*self.mass_cart\
        +2*self.mass_pole*self.length**2\
               +self.length**2*self.mass_cart+self.length**2)*self.phi - \
            force*(self.moment_inertia+self.mass_pole*self.length**2\
            -self.mass_pole*self.length +self.moment_inertia**2)))
        # final phi-acceleration
        phi_acc = numerator_phi_acc / denominator_X
        ################################
        if self.kinematics_integrator == 'euler':
            x = x + self.tau * x_dot
            x_dot = x_dot + self.tau * x_acc
            theta = theta + self.tau * theta_dot
            phi = phi + self.tau * phi_dot
            theta_dot = theta_dot + self.tau * theta_acc
            phi_dot = phi_dot + self.tau * phi_acc
        else:  # semi-implicit euler
            x_dot = x_dot + self.tau * x_acc
            x = x + self.tau * x_dot
            theta_dot = theta_dot + self.tau * theta_acc
            theta = theta + self.tau * theta_dot
            phi_dot = phi_dot + self.tau * phi_acc
            phi = phi + self.tau * phi_dot

        self.state = (x, x_dot, theta, theta_dot, phi, phi_dot)
        done = bool(
            x < -self.x_threshold
            or x > self.x_threshold
            or theta < -self.theta_threshold_radians
            or theta > self.theta_threshold_radians
            or phi < -self.phi_threshold_radians
```

```
            or phi > self.phi_threshold_radians
        )

        if not done:
            reward = 1.0
        elif self.steps_beyond_done is None:
            self.steps_beyond_done = 0
            reward = 1.0
        else:
            self.steps_beyond_done += 1
            reward = 0.0

        return np.array(self.state), reward, done, {}

    def reset(self):
        self.state = self.np_random.uniform(low=-0.05, high=0.05, size=(6,))
        self.steps_beyond_done = None
        return np.array(self.state)
```

A.6 Test

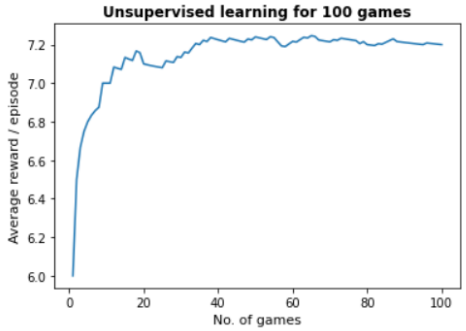
```
# utility function to measure time taken by each test run
def exec_time(start, end):
   diff_time = end - start
   m, s = divmod(diff_time, 60)
   h, m = divmod(m, 60)
   s,m,h = int(round(s, 0)), int(round(m, 0)), int(round(h, 0))
   print("Execution Time: " + "{0:02d}:{1:02d}:{2:02d}".format(h, m, s))

# start time of each test run
start = time.time()
# create environment
env = DoubleInvertedPendulum()
# number of samples processed before the model is updated
batch_size = 5
# a full training pass over the entire dataset such that
# each example has been seen once
num_epochs = 4
# controls the rate or speed at which the model learns
learning_rate_alpha = 3e-4
# create agent
agent = Agent(num_actions=env.action_space.n, batch_size=batch_size,
                learning_rate_alpha=learning_rate_alpha, num_epochs=num_epochs,
                input_dimensions=env.observation_space.shape)

# number of games
num_games = 100
# track best score: minimum score for the environment
best_score = env.reward_range[0]
# record score history
score_history = []
# a reward function that informs the agent how well
# its current actions and states are doing
learn_iters = 0
# track average score
average_score = 0
# number of steps means using one batch size of training data to train the model
```

```
num_steps = 0

for i in range(num_games):
    observation = env.reset()
    terminal_flag = False
    score = 0
    while not terminal_flag:
        # choose action based on the current state of the environment
        action, probability, value = agent.action_choice(observation)
        # get information back from environment
        observation_, reward, terminal_flag, info = env.step(action)
        # update step
        num_steps += 1
        # update score based on current reward
        score += reward
        # store transition in the agent memory
        agent.interface_agent_memory(observation,
                                     action,
                                     probability,
                                     value,
                                     reward,
                                     terminal_flag)
        if num_steps % num_steps == 0:
            agent.learn()
            learn_iters += 1
        observation = observation_
    score_history.append(score)
    average_score = np.mean(score_history[-100:])

    if average_score > best_score:
        best_score = average_score
    # format output
    if i+1 < 10:
        print('episode: ', i+1, '  |  score: %.0f' % score)
    elif i+1 < 100:
        print('episode: ', i+1, ' |  score: %.0f' % score)
    else:
        print('episode: ', i+1, '|  score: %.0f' % score)
episodes = [i+1 for i in range(len(score_history))]
# end time of each test run
end = time.time()
# display time taken by test run
print('---------')
exec_time(start,end)
# visualise learning
def plot_learning_curve(episode, scores):
    running_avg = np.zeros(len(scores))
    for i in range(len(running_avg)):
        running_avg[i] = np.mean(scores[max(0, i-100):(i+1)])
    plt.plot(episode, running_avg)
    plt.title(f"Unsupervised learning for {num_games} games", fontweight='bold')
    plt.xlabel('No. of games', fontsize=11)
    plt.ylabel('Average reward / episode', fontsize=11)

plot_learning_curve(episodes, score_history)
```

B Graphical Representation

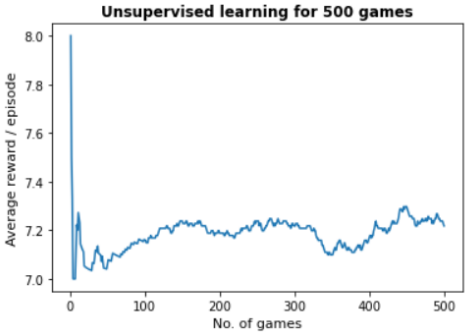
B.1 $\theta = 0.05,\ \phi = 0.05$

```
Execution Time: 00:00:14
---------
Total reward for 100 unsupervised episodes: 720.0
---------
```



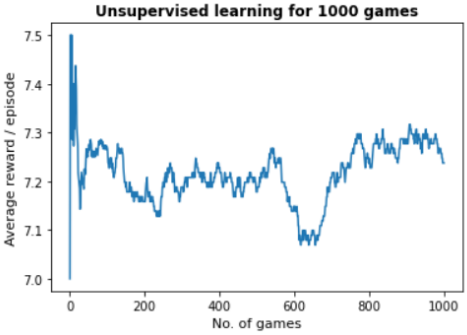
(a)

```
Execution Time: 00:01:20
---------
Total reward for 500 games: 3595.0
---------
```



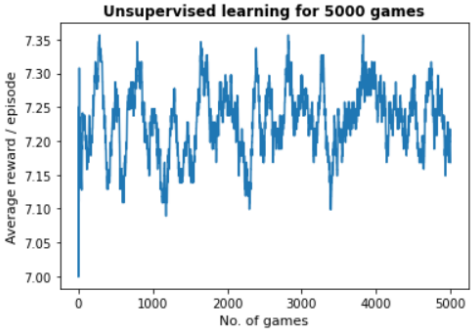
(b)

```
Execution Time: 00:02:31
---------
Total reward for 1000 games: 7213.0
---------
```



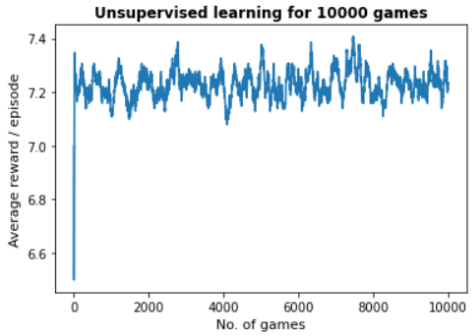
(c)

```
Execution Time: 00:14:45
---------
Total reward for 5000 games: 36146.0
---------
```



(d)

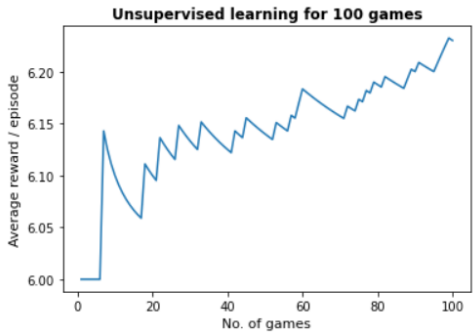
```
Execution Time: 00:29:14
---------
Total reward for 10000 games: 72283.0
---------
```



(e)

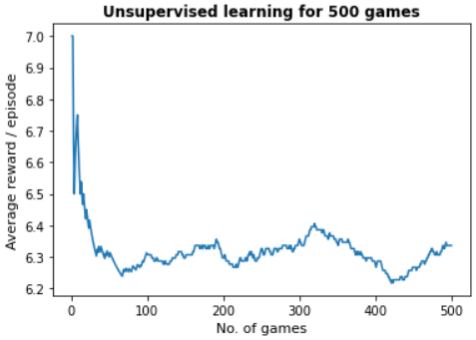
B.2 $\theta = 0.10,\ \phi = 0.10$

```
Execution Time: 00:00:12
---------
Total reward for 100 games: 623.0
---------
```



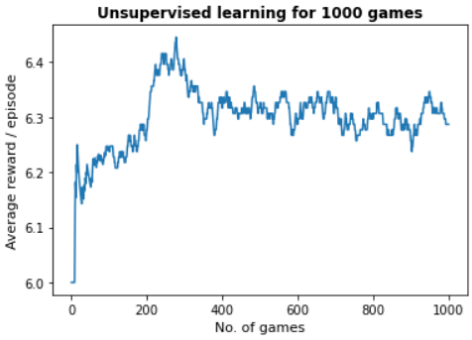
(a)

```
Execution Time: 00:01:11
---------
Total reward for 500 games: 3157.0
---------
```



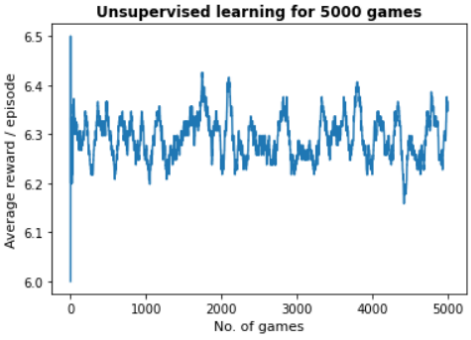
(b)

```
Execution Time: 00:02:10
---------
Total reward for 1000 games: 6302.0
---------
```



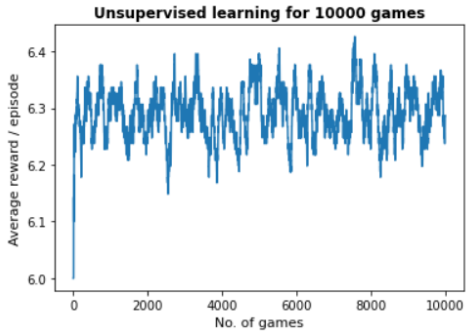
(c)

```
Execution Time: 00:11:40
---------
Total reward for 5000 games: 31465.0
---------
```



(d)

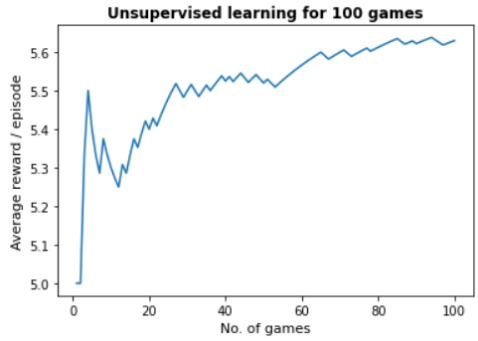
```
Execution Time: 00:27:45
---------
Total reward for 10000 games: 62919.0
---------
```



(e)

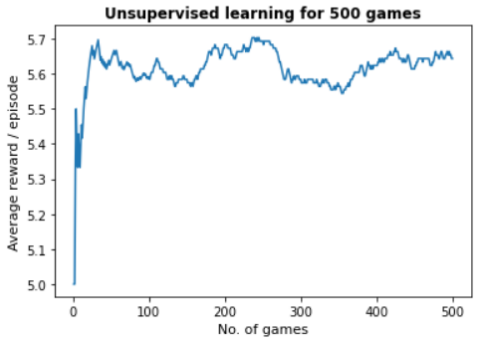
B.3 $\theta = 0.15,\ \phi = 0.15$

```
Execution Time: 00:00:10
---------
Total reward for 100 games: 563.0
---------
```



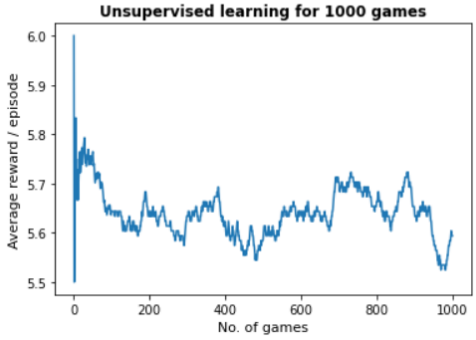
(a)

```
Execution Time: 00:00:52
---------
Total reward for 500 games: 2811.0
---------
```



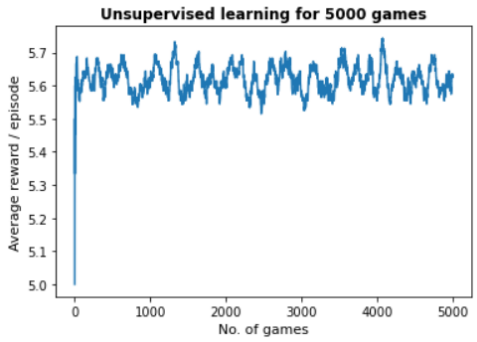
(b)

```
Execution Time: 00:01:58
---------
Total reward for 1000 games: 5636.0
---------
```



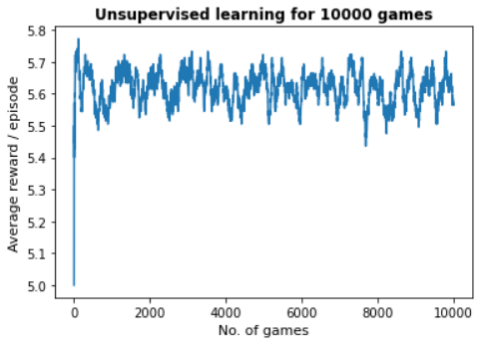
(c)

```
Execution Time: 00:08:54
---------
Total reward for 5000 games: 28097.0
---------
```



(d)

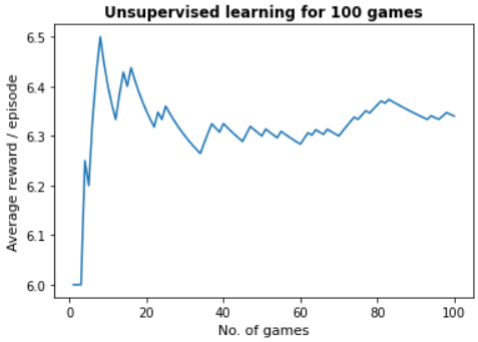
```
Execution Time: 00:17:28
---------
Total reward for 10000 games: 56180.0
---------
```



(e)

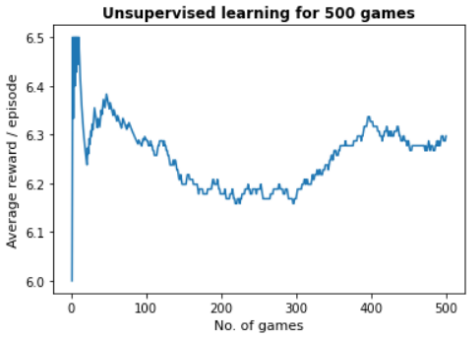
B.4 $\theta = 0.05,\ \phi = 0.10$

```
Execution Time: 00:00:13
---------
Total reward for 100 games: 634.0
---------
```



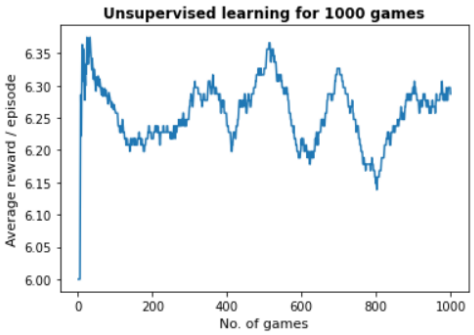
(a)

```
Execution Time: 00:01:17
---------
Total reward for 500 games: 3127.0
---------
```



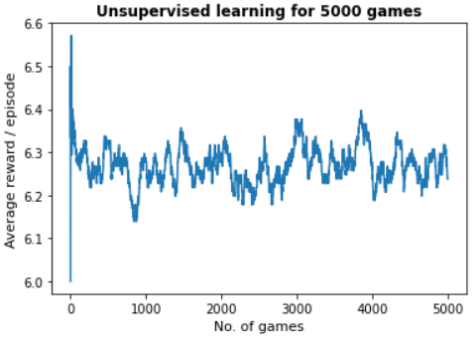
(b)

```
Execution Time: 00:02:26
---------
Total reward for 1000 games: 6258.0
---------
```



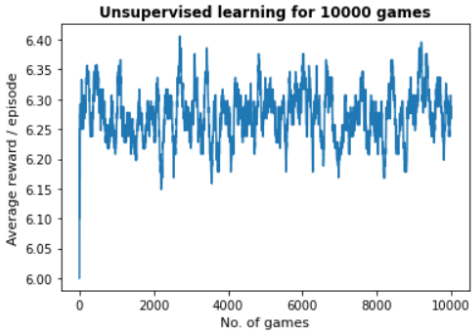
(c)

```
Execution Time: 00:12:57
---------
Total reward for 5000 games: 31340.0
---------
```



(d)

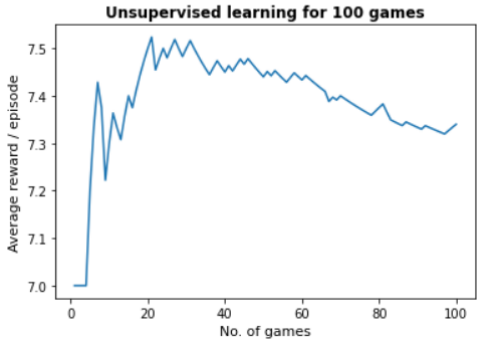
```
Execution Time: 00:22:12
---------
Total reward for 10000 games: 62744.0
---------
```



(e)

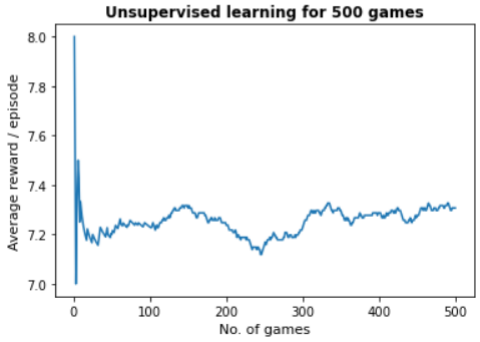
B.5 $\theta = 0.10,\ \phi = 0.05$

```
Execution Time: 00:00:14
---------
Total reward for 100 games: 734.0
---------
```



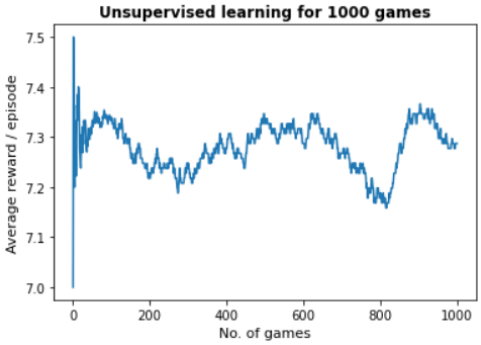
(a)

```
Execution Time: 00:01:12
---------
Total reward for 500 games: 3630.0
---------
```



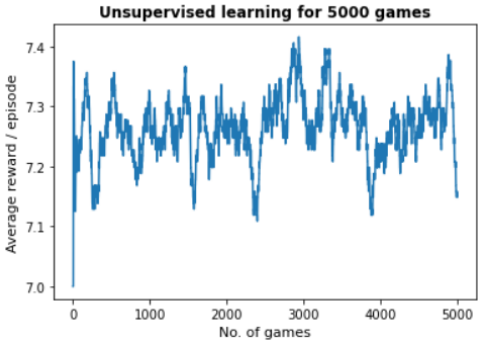
(b)

```
Execution Time: 00:02:28
---------
Total reward for 1000 games: 7279.0
---------
```



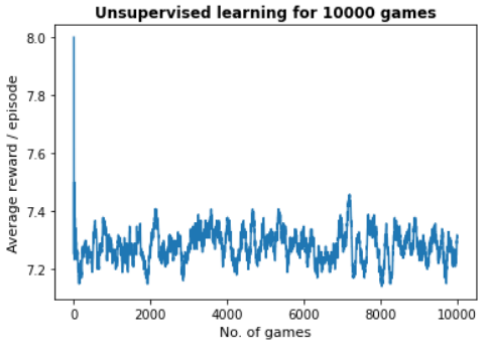
(c)

```
Execution Time: 00:12:33
---------
Total reward for 5000 games: 36317.0
---------
```



(d)

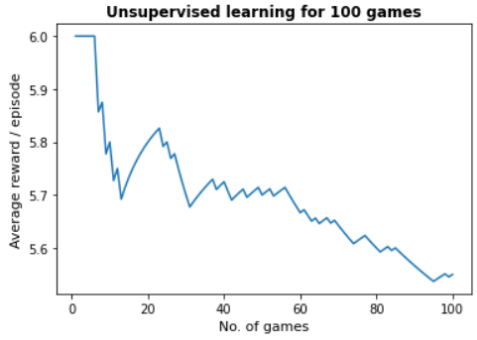
```
Execution Time: 00:23:40
---------
Total reward for 10000 games: 72783.0
---------
```



(e)

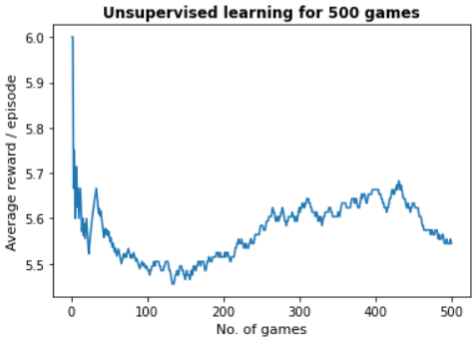
B.6 $\theta = 0.00,\ \phi = 0.15$

```
Execution Time: 00:00:10
---------
Total reward for 100 games: 555.0
---------
```



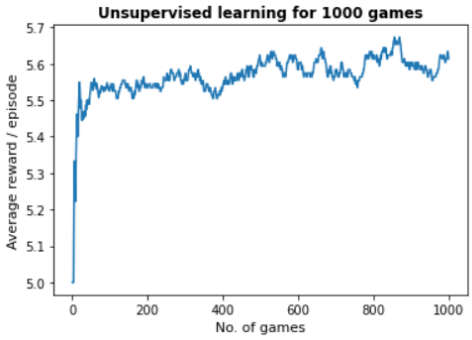
(a)

```
Execution Time: 00:00:51
---------
Total reward for 500 games: 2783.0
---------
```



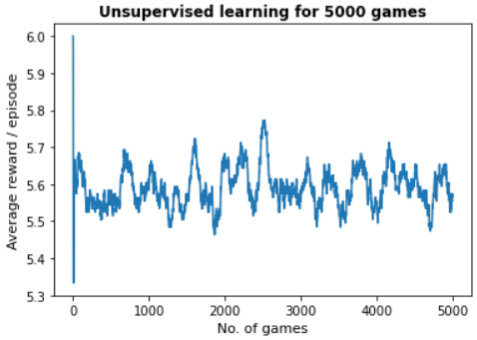
(b)

```
Execution Time: 00:01:44
---------
Total reward for 1000 games: 5578.0
---------
```



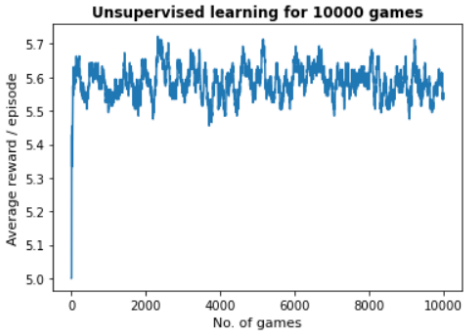
(c)

```
Execution Time: 00:09:44
---------
Total reward for 5000 games: 27951.0
---------
```



(d)

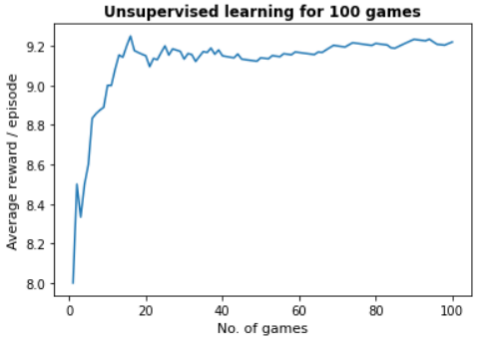
```
Execution Time: 00:17:44
---------
Total reward for 10000 games: 55859.0
---------
```



(e)

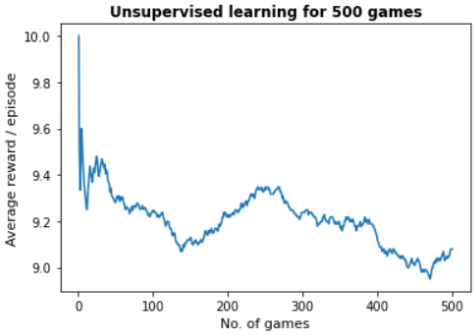
B.7 $\theta = 0.15,\ \phi = 0.00$

```
Execution Time: 00:00:19
---------
Total reward for 100 games: 922.0
---------
```



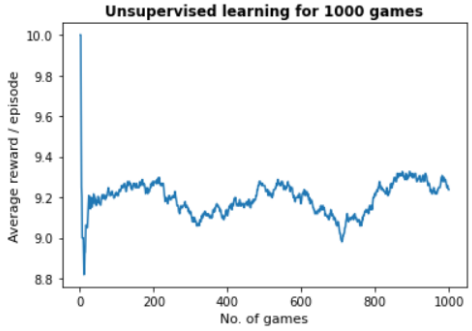
(a)

```
Execution Time: 00:01:28
---------
Total reward for 500 games: 4590.0
---------
```



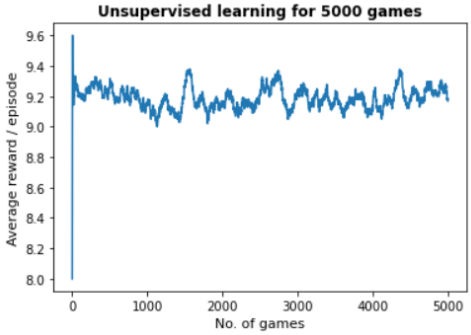
(b)

```
Execution Time: 00:03:26
---------
Total reward for 1000 games: 9200.0
---------
```



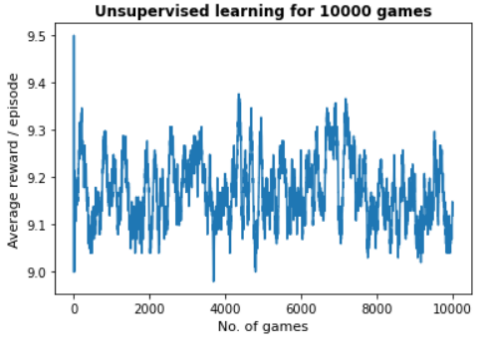
(c)

```
Execution Time: 00:13:50
---------
Total reward for 5000 games: 45931.0
---------
```



(d)

```
Execution Time: 00:28:11
---------
Total reward for 10000 games: 91709.0
---------
```



(e)