Machine Learning Methods for Control of a Double Pendulum on A Cart

Jadon Duby (30008202), Cole Runions-Kahler (30102473)
Nathan Enerson (30065688), Steven Susanto (30073781)

April 12th 2023

1 Abstract

In this project we will develop machine learning algorithms to control a chaotic system known as the double pendulum on a cart. The double pendulum on a cart system is composed of two pendulums connected by a joint which hangs from a cart on a sliding track. The cart can be controlled with an input force which causes the cart to shift left or right across the track, and thus directs the swinging pendulums into an unstable equilibrium position. The double pendulum is a popular tool in academic settings because it illustrates the intricate, chaotic, output of a seemingly simple physical system [8]. There will be three stages to our project. First, we will derive the equations of motion governing a double pendulum system. Second, we will devise a machine learning algorithm to hold the double pendulum in the three unstable equilibrium from a starting position that is close to stable. The ability to control and stabilize a system such as this is important as a robust way of testing the ability of a machine learning algorithm to adapt to a changing physical system to meet a requested outcome. The problem of controlling a double pendulum has been studied as a machine learning problem and with more traditional robotics control methods such as feed forward control [6]. Machine learning methods are of particular importance because more complex chaotic systems do not always have such straightforward control schemes and therefore more flexible methods are desired.

2 Introduction

Control Theory, is the field of study involved in manipulating dynamical systems to achieve a desired outcome. In the case of our Double Pendulum on a Cart System (DPOAC), we want to use Closed-loop control, which involves measuring the output of the system and feeding it back into a controller where it is measured and corrected, then an input from the controller is fed into the system to achieve an outcome [3]. The control of a DPOAC system will involve measuring the state of the current system, and inputting an action to adjust the system state by applying a force horizontally to the cart, either left or right. The Machine learning methods used to enable the action, measure the outcome, adjust the parameters, and repeat these steps until a desired state is achieved, are applicable in a broad range of scenarios where control of a dynamic system is desired. The DPOAC acts as a simple to understand yet exceedingly complex example of such a system.

3 Mathematical Model of the DPOAC

3.1 Description of the double pendulum on a cart system

The system is as follows. Two pendulum systems with mass components m_1 and m_2 are attached to each other with the with the first pendulum system being attached by a rigid rod of length L_1 to a cart which is allowed to move horizontally in either direction along a track, and the second pendulum system is attached to the first via another rigid rod of length L_2 . Each pendulum system is allowed to freely rotate around the point at which it is joined either to the cart or the first pendulum. Compared to a single pendulum system, which follows a predictable easy to follow path, the equations which control the motion of the double pendulum system are significantly more complex, and unpredictable.

The notation here represents the following $\dot{\theta} = \frac{d\theta}{dt}$ where t is time, and $\ddot{\theta} = \frac{d^2\theta}{dt^2}$, and so on, where $\dot{\theta}$ and θ and all such parameters are treated as independent of each other when the derivative of the functions with respect to such a parameter is taken. The following parameters and their derivatives with respect to time are used to describe the locations and velocities of the pendulum components, with all other parameters being constants such as the masses m_i in the pendulum components and cart, and the lengths L_i of the pendulums themselves [13]:

- θ_0 = the horizontal position of the cart along the track.
- θ_1=the angle created by the rod of the first pendulum and the line of rest, which points perpendicularly from the cart path.
- θ_2=the angle created by the rod of the second pendulum and the line of rest.

Using Newtonian Mechanics and summing the kinetic energies T defined as $T = \frac{1}{2}mv^2$ of the cart T_0, and the first and second masses T_1 and T_2. m_i is the respective mass and v_i is the respective velocity defined as the change in position with respect to time ie. $\frac{d\theta}{dt}$:

$$T_{tot} = T_0 + T_1 + T_2$$

and summing the potential energies V using $V = mgh$ where g is the gravitational constant acceleration and h is its vertical position defined in the system. The potential energy of the cart is zero because the cart is restricted to horizontal motion only.

$$V_{tot} = V_1 + V_2$$

The following equations for kinetic and potential energies are from a system with the center of masses at the middle of the rods. This adds an additional term which includes the rotational kinetic energies of the rods, containing the moment of inertia I of a rod [13] Many steps in the derivation have been excluded for the sake of brevity. In certain scenarios involving double pendulums, the rods of the pendulums are defined as being mass-less, with the mass component only being found at the end of each pendulum, while in other scenarios the mass is defined as at the center of each rod, as is the following case. These particular equations are from Crowe-Wright's derivation, and l_1, l_2 are defined as the half lengths of the rods, while L_1 and L_2 are the full lengths [3].

$$\begin{aligned}
T_0 &= \frac{1}{2}m_0\dot{\theta}^2 \\[2em]
T_1 &= \frac{1}{2}m_1(\dot{x}_1^2 + \dot{y}_1^2) + \frac{1}{2}I_1\dot{\theta}_1^2 \\
&= \frac{1}{2}m_1\left[(\dot{\theta}_0 + l_1\theta\dot{c}os\theta_1)^2 + (-l_1\dot{\theta}_1 sin\theta)^2\right] + \frac{1}{2}I_1\dot{\theta}_1^2 \\
&= \frac{1}{2}m_1\dot{\theta}_0^2 + \frac{1}{2}(m_1l_1^2 + I_1)\dot{\theta}_1^2 + m_1l_1\dot{\theta}_0\dot{\theta}_1 cos\theta_1 \\[2em]
T_2 &= \frac{1}{2}m_2(\dot{x}_2^2 + \dot{y}_2^2) + \frac{1}{2}I_2\dot{\theta}_2^2 \\
&= \frac{1}{2}m_2\left[(\dot{\theta}_0 + L_1\dot{\theta}_1 cos\theta_1 + l_2\dot{\theta}_2 cos\theta_2)^2 + (-L_1\dot{\theta}_1 sin\theta_1 - l_2\dot{\theta}_2 sin\theta_2)^2\right] + \frac{1}{2}I_2\dot{\theta}_2^2 \\
&= \frac{1}{2}m_2\dot{\theta}_0^2 + \frac{1}{2}m_2L_1^2\dot{\theta}_1^2 + \frac{1}{2}(m_2l_2^2 + I_2)\dot{\theta}_2^2 + m_2L_1\dot{\theta}_0\dot{\theta}_1 cos\theta_1 + m_2l_2\dot{\theta}_0\dot{\theta}_2 cos\theta_2 + m_2L_1l_2\dot{\theta}_1\dot{\theta}_2 cos(\theta_1 - \theta_2) \\[2em]
V_1 &= m_1gy_1 \\
&= m_1gl_1 cos\theta_1 \\[2em]
V_2 &= m_2gy_2 \\
&= m_2g(L_1 cos\theta_1 + l_2 cos\theta_2)
\end{aligned} \tag{1}$$

3.2 Derivation of the Dynamic equations of motion

The Lagrangian of a system is a function defined as the kinetic energy minus the potential energy,

$$L = T - V$$

$$L = T_0 + T_1 + T_2 - V_1 - V_2$$

Newtons formalism needs directions for its forces so vectors are required, however the Lagrangian only needs scalar quantities since it deals with Kinetic and Potential energies. The Lagrangian also has the same form for any chosen generalized coordinates [13].

The Lagrangian derivation in the system described by Crowe-Wright is as follows [3]:

$$L = \frac{1}{2}(m_0+m_1+m_2)\dot{\theta}_0^2 + \frac{1}{2}(m_1 l_1^2 + m_2 L_1^2 + I_1)\dot{\theta}_1^2 + \frac{1}{2}(m_2 l_2^2 + I_2)\dot{\theta}_2^2 + (m_1 l_1 + m_2 L_1)\dot{\theta}_0\dot{\theta}_1 cos\theta_1$$

$$+m_2 l_2 \dot{\theta}_0\dot{\theta}_2 cos\theta_2 + m_2 l_2 L_1 \dot{\theta}_1\dot{\theta}_2 cos(\theta_1 - \theta_2) - g(m_1 l_1 + m_2 L_2)cos\theta_1 - m_2 g l_2 cos\theta_2$$

This is shown as an example of the complexity of the equations governing a seemingly simple physical system. Taking the following derivatives will give us the equation of motion with respect to variable θ :

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}}\right) - \frac{\partial L}{\partial \theta} = \vec{q}$$

Where q is the vector of generalized forces. This is the Euler-Lagrange equation, and the parameters for which we apply the previous function and derive the equation of motion are θ_0, θ_1 , and θ_2. The only force applied to the system is $u(t)$ which is applied horizontally to the cart giving the following system of equations [3]:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}_0}\right) - \frac{\partial L}{\partial \theta_0} = u(t)$$

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}_1}\right) - \frac{\partial L}{\partial \theta_1} = 0$$

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}_2}\right) - \frac{\partial L}{\partial \theta_2} = 0$$

which is represented as the following vector of forces:

$$\vec{q} = \begin{bmatrix} u(t) \\ 0 \\ 0 \end{bmatrix} \tag{2}$$

The Euler-Lagrange Equations require a significant amount of algebra in their derivations, and end up being complicated and cumbersome, so their derivation and final forms will be omitted in this paper. What is important however is that they are functions of the cart position and angle parameters θ_0, θ_1, and θ_2, and their corresponding linear and angular velocities and accelerations, $\dot{\theta}_0$, $\dot{\theta}_1$, $\dot{\theta}_2$,and $\ddot{\theta}_0$, $\ddot{\theta}_1$, $\ddot{\theta}_2$. The following notations and methods are used by Crowe-Wright [3] and Bogdanov[2]. Once the Euler-Lagrange Equations have been obtained for θ_0, θ_1, and θ_2, the result is a non-linear second order system that we can represent as

$$D(\theta)\ddot{\theta} + C(\theta,\dot{\theta})\dot{\theta} + G(\theta) = H(u)$$

where $\ddot{\theta}$, and $\dot{\theta}$, are vectors of their corresponding second and first derivative parameters for θ_0, θ_1, and θ_2, and D, C, G and H are specified Matrices containing the Euler Lagrange Equations.

Matrices allow for a condensing of the Euler-Lagrange equations into a more manageable form. We can further convert this system to a first-order problem by doing the following:

$$\ddot{\theta} = -D^{-1}C\dot{\theta} - D^{-1}G + D^{-1}Hu$$

and we then set

$$x = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}, \dot{x} = \begin{bmatrix} \dot{\theta} \\ \ddot{\theta} \end{bmatrix}$$

and get as a result

$$\dot{x} = A(x)x + L(x) + B(x)u$$

a first order system where A, B, and L are the following Matrices containing D^{-1}, C, G, H, and I, the identity matrix.

$$A(x) = \begin{bmatrix} 0 & I \\ 0 & -D^{-1}C \end{bmatrix}, L(x) = \begin{bmatrix} 0 \\ D^{-1}G \end{bmatrix}, B(x) = \begin{bmatrix} 0 \\ D^{-1}H \end{bmatrix}$$

D is a symmetric and non singular matrix, meaning it is square with a non-zero determinant, and is thus invertible, with its inverted form being also symmetric [3] The matrices containing the derived Euler-Lagrange Equations can then be written into code, in the form of this first order system, and we then can use numerical methods, such as the Runge-Kutta, to solve this first order ODE system, and simulate it.

3.3 Problem definition: description of holding in upright stable positions

Our goal is to create and control a DPOAC system and achieve the following:

- Start the system near a 'Hold up position' (an unstable equilibrium position, specifically where both pendulums are upright), move the pendulums, via applying force to the cart, to that position and keep them there.

A system is at equilibrium when its state does not change when no external forces are acting upon it. In our system, that means when the pendulums and cart are at rest. The system is at a stable equilibrium when, if the system is disturbed, it will return to the original equilibrium position. In the case of the double pendulum, the only stable equilibrium position is when both pendulums are in a downwards hanging position. The system is at an unstable equilibrium when if disturbed, the system will tend to move away from the equilibrium position. In our case, there are three unstable equilibrium's for the system, which if disturbed from these positions, they will tend to fall to the stable equilibrium position. The unstable equilibrium position of interest to us is when both pendulums are held upright, hence 'Hold up position'.

4 Mathematical Foundations of Reinforcement learning

Reinforcement Learning (RL) is one of many classifications of machine learning algorithms that deals with decision making in dynamic environments. The main actor in RL is called an agent. An agent constantly learns in an environment from its own actions through a series of trial and error. Where each action it performs, the environment sends either a feedback in the form of rewards or penalties. It's main objective is to maximize the total cumulative reward it receives over time.

The agent learns by adjusting its behaviour based on the feedback it receives to obtain the optimal strategy leading to the most reward over time. RL is commonly used in fields such as robotics, game development and finances where the optimal strategy may not be obvious as the environment is always changing. By using RL, agents can learn to adapt to changes in their environment which is perfect for the double pendulum on a cart problem.

Before getting into the mathematical framework to model the RL environment for our agents, it is important to know that agent's tasks can be classified into two categories depending on the problem. They are either episodic or continuous tasks.

1. In an episodic task, the agent interacts with the environment for a finite amount of time, after which the episode ends and the agent receives a final reward. Games such as chess or checkers are examples of an episodic task, where each game is a separate episode, and the agent's goal is to win as many games as possible.

2. In a continuous task, the agent interacts with the environment continuously, and there is no final timestep. In other words, the agent receives rewards continuously and its goal is to maximize the cumulative reward over an infinite time period. Examples include control problems in robotics.

Episodic tasks are often simpler to model and solve than continuous tasks, as the agent can focus on optimizing its behavior within a fixed episode. Continuous tasks, on the other hand, require the agent to balance exploration and exploitation[1] over an infinite time period, which can be more challenging. In our case, balancing the DPOAC system is technically a continuous task. However, we can apply a small trick to make it a hybrid of both episodic and continuous task, where we we would have a finite timestep for the agent to interact with but still apply the exploration and exploitation methods to achieve our goal, to balance the DPOAC into an upright position. Note that, for our problem, the agent the is the DPOAC itself. Now, We need to consider two of the following cases:

1. When the agent goes out of bounds; going out of bounds means either the first pendulum swings below the horizontal line where the cart moves or the cart moves out of its given range of track. In any of these cases, it is safe to say the our system has become chaotic and has failed to balance itself, we simply terminate the episode and restart the system back to its starting position, i.e. both pendulum close to an upright position.

2. When the agent successfully balances itself, we simply needs to keep track of how long it is able to balance without it going chaotic. That is, if the agent is able to balance for a certain period of time until the end of an episode, we assume that it can balance itself in an infinite amount of time. The challenge here would be to determine how long each episode should be.

4.1 Markov decision process

Markov decision process (MDP) is a framework to model decision making in a dynamic systems. The essence of a Reinforcement Learning (RL) problem can be formalized using the Markov Decision Process (MDP).

[1] The notion of exploration and exploitation will be further explained further down in the paper

The formal definition of a Markov Decision Process is a 4-tuple (S, A, T, R) for a given finite set of states S and A, a transition probability function $T : S \times A \to [0, 1]$ and a reward function $R : S \times A \to \mathbb{R}$[14]. We will explore each components individually.

1. **States.** The state space S is defined to be the set $s_1, \ldots, s_n$ where the cardinality of the state space $|S| = n$. A high level definition of each state $s_1, \ldots, s_n$ are that they are representations of the environment which stores information(s) at a given point in time.

2. **Actions.** The action space A is defined to be the set $a_1, \ldots, a_n$ where this set of actions can be taken in some state $s \in S$, denoted $A(s)$. It is important to note that not all actions can be taken in a state.

3. **Transition Probability Function.** Previously, by taking action $a \in A$ in some state $s \in S$, we call this a state-action pair, denoted (s, a), the system will transition from state s to some new state s'. We define T as follows

   $$T : S \times A \to p \text{ where } 0 \leq p \leq 1, p \in \mathbb{R}$$

   For instance, $T(s_t, a_t) = P(s'_{t+1}|s_t, a_t)$ for some probability between 0 and 1. It is also important to note that for all states s and actions a, $\sum_{s' \in S} T(s, a) = 1$, that is T defines a probability distribution over possible next states.

4. **Reward Function.** A reward function R gives back to the system a reward for transitioning to a new state after applying action on some old state. R is defined as follows

   $$R : S \times A \to \mathbb{R}$$

   For instance $R(s, a) = 3$ means taking action $a \in A$ at state $s \in S$ transitioning us to a new state s' where the reward is 3.

Ultimately, our goal is to maximize our cumulative reward over time. To do this, we introduce the concept of the expected return of the rewards at a given time step, that is we think of a return as the sum of future rewards. We mathematically define the sum of our rewards G_t at time t to be

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T \text{ where } T \text{ is the final timestep}$$

It is important to note that, by nature of the reinforcement learning problem, the training period of our agent depends on the system and can be up to an infinity timestep, which means that we do not have a final timestep. This implies that G_t could go up to infinity and this conflicts with what we want to achieve as it is not possible to maximize something that is infinity.

To remedy the issue, we introduce a concept called the discounted return with aims to make G_t finite. We define the discount rate $0 \leq \gamma < 1 \in \mathbb{R}$. This discount rate will be used to discount future rewards, and modify G_t to be the following

$$\begin{aligned} G_t &= \gamma^0 R_{t+1} + \gamma^1 R_{t+2} + \gamma^2 R_{t+3} + \ldots \\ &= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \end{aligned}$$

The idea behind a discounted return makes it so we care more about immediate rewards compared to future rewards and the further into the future, the less important the system considers it to be.

4.1.1 Policy and Value functions

Given an MDP (S, A, T, R), there are more than one type of policy. However, for our problem, we will define policy as a function, denoted π, that maps a given state to probabilities of selecting each possible action from that state, mathematically as follows

$$\pi : S \times A \to p \text{ where } 0 \leq p \leq 1, p \in \mathbb{R}$$

such that for state state $s \in S$, $\pi(s, a) \geq 0$ and $\sum_{a \in A} \pi(s, a) = 1$. This implies that achieving an optimal policy should yield us the maximum return. In application, we say an agent follows policy π at time t, then $\pi(s, a)$ is the probability of being in state s and taking action a.

Many Reinforcement Learning algorithms for MDPs compute optimal policies by learning value functions. Now we need to determine how good it is for an agent to take an action in a given state. We can think of a way to formalize this in terms of expected returns. We say value function is defined with respect to how the agent acts which is influenced by the policy it is following. There are two types of value functions, i.e. state-value function and action-value function, mathematically defined as follows [11]

We define a state-value function $V : S \to \mathbb{R}$ for policy π as follows

$$\begin{aligned} V_\pi(s, a) &= E_\pi[G_t | S_t = s] \\ &= E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| s_t = s\right] \text{ By definition of } G_t \\ &= E_\pi\left[R_{t+1} + \sum_{k=1}^{\infty} \gamma^k R_{t+k+1} \middle| s_t = s\right] \end{aligned} \tag{3}$$

Where R_{t+1} is the immediate reward for being in state s_t. Note that E_π denotes the expected value under policy π. In other words, V_π is the discounted expected return from starting at state s at time t and following policy π thereafter.

Similarly, we define the action-value function $Q : S \times A \to \mathbb{R}$ for policy π as follows

$$\begin{aligned} Q_\pi(s, a) &= E_\pi[G_t | S_t = s, A_t = a] \\ &= E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| s_t = s, a_t = a\right] \text{ By definition of } G_t \\ &= E_\pi\left[R_{t+1} + \sum_{k=1}^{\infty} \gamma^k R_{t+k+1} \middle| s_t = s, a_t = a\right] \end{aligned} \tag{4}$$

Where R_{t+1} is the immediate reward for performing the state action pair (s_t, a_t). And Q_π is the discounted expected return from starting at state s at time t then taking action a and following policy π thereafter.

A policy π is called better or equal than policy π', i.e. $\pi \geq \pi'$, if and only if $V_\pi(s) \geq V'_\pi(s)$ for all states $s \in S$. This also applies for action-value functions.

Conventionally, the function itself is also called the Q-function and the output of the function is called a Q-value. Notice that for a value function to follow an optimal policy means that the value function itself will be optimal. We define the optimal action-value function as the max of all possible value functions under any policy π, denoted $Q_* = \max_\pi Q_\pi$. That is, a value function that yields the biggest expected discounted return under any policy π. The same logic can be applied for state-value function.

4.1.2 Bellman's Optimality equation

A property of Q_* is that it must satisfy the following equation [5]

$$Q_*(s,a) = E[R_{t+1} + \gamma \max_{a'} Q_*(s',a')]$$

This is also called Bellman's optimality equation. We will proof this by deriving from our definition of Q_*.

Proof. Suppose the state action pair (s_t, a_t),

$$\begin{aligned}
Q_*(s_t,a_t) &= \max_\pi Q_\pi \\
&= \max_\pi \left(E_\pi \left[R_{t+1} + \sum_{k=1}^{\infty} \gamma^k R_{t+k+1} \middle| s_t = s, a_t = a \right] \right) && \text{by (4)} \\
&= \max_\pi (E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | s_t = s, a_t = a]) \\
&= \max_\pi (E_\pi[R_{t+1} + \gamma (R_{t+2} + \gamma R_{t+3} + \dots) | s_t = s, a_t = a]) \\
&= \max_\pi \left(E_\pi \left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \middle| s_t = s, a_t = a \right] \right) \\
&= \max_\pi (E_\pi[R_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1}) | s_t = s, a_t = a]) \\
&= E_\pi[R_{t+1} + \gamma \max_{a'} Q_*(s_{t+1}, a_{t+1}) | s_t = s, a_t = a] \\
&\text{By optimal action-value function definition}
\end{aligned}$$

□

Note that a similar derivation can be applied to derive the optimal state-value function. Notice the recursive function call from our derivation, Bellman's optimality equation states that, at time t, for any state action pair (s,a), the expected return for taking action a in state s will equal to R_{t+1},the reward for performing the State action pair (s,a) plus $\gamma \max_{a'} Q_*(s',a')$ is the maximum discounted return obtainable from all possible actions in the state you end up in (s',a'). We will see further down the paper how Bellman's equation is being used in application.

4.2 Action spaces

In Reinforcement learning, the agent's actions can be categorized into a discrete or continuous action space. The nature of the problem determines which type of action space will be used.

- In a discrete action space, the agent can only do a finite number of feasible actions, which is commonly represented by a collection of discrete values. For instance, a game of tic-tac-toe, there are only a total of 9 possible actions an agent can take initially, where this collection of possible actions decreases as the game progresses. Or in a simple grid maze game where the agent looks for a way out, there are a total of 4 possible actions for an agent, i.e. $\{left, right, up, down\}$.
- In a continuous action space, the agent can take any action within a certain range or set of possible values, rather than being limited to a fixed set of possible actions. Continuous action spaces are often encountered in real-world problems, where there may be a large number of possible actions that an agent can take, each of which has a different impact on the environment. For instance, our problem DPOAC has a continuous action space as there is a range of how much force the agent has to apply on the cart.

RL problems with a discrete action spaces are often simpler to work with than a continuous action space. In a continuous action space, agents need to learn the ranges of their actions which could be infinitely many. A method to over come this challenge is to actually discretize the continuous action space. For instance, our problem has a cart that can move on an infinitely long horizontal plane. Where its action space is a scalar of how much force to apply on a particular direction[2]. Now, as an example, We can discretize the infinitely long plane into ranges such as $-5 \leq x \leq 0$, $0 < x \leq 5$ and the rest of the x-plane where x represents the horizontal plane. This gives us 3 possible locations of where the cart would be, implying that depending on depending on how much force is applied on the cart, we can be sure that the cart would be in a certain location. Mathematically, we have the range of the forces mapped to the horizontal location of the cart. This technique can be refined by defining smaller ranges so that the values are closer to the actual simulation.

5 Machine learning algorithms for the hold up problem

5.1 Overview of Q-learning

Q-learning is one example of a reinforcement learning algorithm. It can be applied to an MDP to learn an optimal policy which will be proven later. Q-learning achieves this by filling and updating a Q-table which maps state action pairs to quality or q values that measures the algorithms best understanding of how valuable that state action pair is.

In this paper we chose to study Q-Learning because it is one of the most well studied reinforcement learning algorithms and has been applied to the hold up of a double pendulum problem multiple different times to varying success. Additionally Q-Learning has many variations that offer more specific properties we an leverage such as linear approximation Q-Learning, deep Q-learning, double Q-learning and dueling deep Q-learning. We will study one of these variations that suit our problem later in the paper.

5.2 Training the Q-Learning Algorithm

In order to train the Q-Learning Algorithm five major steps must be taken and four of them must be repeated. These steps will be mentioned slightly out of order for ease of understanding. The

[2] Possible directions are left and right

Q-learning algorithm at its essence learns from making choosing what actions to take then seeing how those decisions affect what state the system ends up in and updating the Q-table based off that information.

5.2.1 Choosing an action

First the algorithm must decide an action to take based off the state the system is in currently. To do this the algorithm uses two primary strategies:

1. Exploration:

   The strategy of exploration is to decide what action to take completely randomly with all possible actions equiprobable. More formally given the system is in a state S then for every valid action in state S referred to as $a_n^S \in A^S$ for $n \in \mathbb{Z}^+$ with the maximum such number being m. Then $\forall n$ such that $a_n^S \in A^S,\ P(a_n^S) = \frac{1}{m}$.

   Exploration is particularly important early in the training process as it helps the algorithm understand more about how the different state action pairs interact with each other. It is very effective at giving the algorithm a wide breadth of general information but not specific information of how to solve the particular problem the algorithm is being applied to.

2. Exploitation:

   The strategy of exploitation is using a greedy approach that maximizes the Q value of each action. Using this type of approach is what will later turn the Q-table into a policy. More formally given the system is in a state S then for every valid action in state S referred to as a_n^S for $n \in \mathbb{Z}^+$ then the action that will be chosen to perform will be the one that satisfies the following condition: $\exists n$ such that $Q(S, a_n^S) = \text{Max}_i(Q(s, a_i^S))$. That is it has the maximum possible Q value from all available actions. It is important to note that in implementations of Q learning if there is no maximum then randomly one of the elements tied for maximum will be selected.

   Exploitation is more important closer to the end of training once the algorithm has a large amount of information on the system it can start refining its idea of how to solve the specific problem however initially it is useless because you are trying to exploit information you do not have yet.

Both exploitation and exploration are important when choosing an action to perform. They need to be balanced with exploitation weighted to happen more often later in the training process but exploration weighted to happen more often initially. This is achieved using the ϵ value.

The ϵ value is a probability value that determines the probability of doing one of the two above mentioned action decision strategy's. Formally it is defined as such: $\epsilon \in \mathbb{R}$ such that $0 \leq \epsilon \leq 1$ where ϵ is initialized to $\epsilon = 1$. Now a exploration based strategy will be used with a probability of ϵ and a exploitation based approach will be used with a probability of $1 - \epsilon$. However ϵ needs to be updated over time so that later exploitation is weighted more. This means we need to decrease ϵ as the training goes on. Note that normally ϵ never hits zero. The process of decreasing ϵ is defined as follows:

1. First the total number of training steps needs to be decided by the user. Normally this is determined by how long the user wants the algorithm to train for and this is defined by N. Also the desired end value of ϵ needs to be decided and stored in e
2. The variable n is defined to track the current step number the algorithm is on.
3. On each step we then use the above information to determine the rate defined by $r = \max(\frac{N-n}{N}, 0)$.
4. This rate r is then used to define the new ϵ value with the following formula $\epsilon = (1-e)r + e$

Note that if the user wants to start ϵ with a lower value then 1 the only change that would be needed in the above method would to change the 1 in the last step with their chosen starting value between 0 and 1. Using this method we can now decrease our ϵ which gives us our overall mixed strategy between exploration and exploitation for choosing an action to perform during training.

5.2.2 Initializing and Updating the Q-table

As mentioned earlier the Q-table maps all state action pairs to a Q value. Formally $\forall a \in A, s \in S$ where S, A are the state and action space of the system respectively $Q(s,a) = q_{s,a}$ where $q_{s,a} \in \mathbb{R}$, $\forall s \in S, a \in A$. The q table is the most important part of this algorithm, it is how the algorithm represents the information it has gained through its training and what it iteritvley is updating to refine. Every step of training the Q table is updated to reflect the information to reflect how good of a decision the algorithm believes the one it just made is. Due to the fact that we update the Q table recursively it needs to be initialized with values. The Q-table is most commonly initialized with all 0's. Although occasionally it will be initialized with positive values to encourage more exploration even when an exploitation based approach is chosen. Now that the Q-Table is initialized it can be updated.

In order to update the Q-table a modified version of the Bellman optimality equation mentioned earlier is used. The formula looks as follows:

$$Q(s,a) = (1-\alpha)(Q(s,a)) + \alpha\{R_{t+1} + \text{Max}_{a'}(Q'(s',a'))\} \tag{5}$$

Each of the symbols appearing in the formula will be broken down:

- s and a are the state action pair the algorithm is updating the Q-value for.
- α is what is called the "learning rate" and affects how large the Q-values change on each step, a larger α leads to large changes in the Q-value. It is preferred to keep α as a smaller value as smaller changes at a time lead to more consistent results. Although larger values can be used if there is a limited amount of training time so bigger changes need to be made. The user determines what value they think this should be based off their time constraint and understanding of the problem.
- R_{t+1} is the reward given by the system for the state action pair s, a. The reward's for different state action pairs need to be initialized by the user before running the algorithm.

- γ is the discount factor that was mentioned earlier, it is the weight that affects how significant future rewards are. A lower γ leads to future actions being weighted less where as a higher γ weights future actions more.
- s' is the state that the system ends up in after performing the state action pair s, a. This information is obtained from the system after it simulates performing a while in state s using the runge kutta method.
- $\text{Max}_{a'}(Q'(s', a'))$ is the maximum possible Q-value obtainable in s' based o pairing it with any valid action a'. Essentially this is just the highest possible Q-value value that could be gotten on the next step given what we chose to do on this step.

Every step this must be applied to update the Q-table. The above formula is what makes Q-learning what it is.

5.2.3 The Process

Now that we have mentioned all of the steps in training the Q-learning algorithm we can put it all together with the following process of training a Q-learning algorithm:

1. First the problem must be able to be simulated in code or the real world problem must be able to be translated into states and actions for the algorithm to interact with. For our paper this was the initial process of deriving the equations of motion and simulating it in code with the runge kutta method. This means that the state space, action space and rewards all need to be defined. Also which states are labled as terminal states and which are labled as starting states will be mandatory later.
2. Second the values that are needed to be chosen by the user must be decided. These include α, γ, N, e. All values that the algorithm needs to initialize will also be initialized such as ϵ
3. Next the Q-table must be initialized with values by the user normally this is 0 as mentioned above.
4. Next the system is setup into one of the initial states so that it is ready for further training.
5. Now that the system is in a state s it must chose an action to be executed this is done using the above mentioned ϵ probability mixed strategy between exploration and exploitation
6. After an action is decided we perform said action this can be done via simulation of some type or physically doing something in the real world.
7. The system will now respond to our action and return the reward for the state action pair named R_{t+1} and the next state performing that action puts us into s'
8. Now we have the information needed to update $Q(s, a)$ based off the above formula.
9. Next we increment n and update the ϵ value
10. Finally if $n \leq N$ and s' is a terminal state we loop back to step 4. If s' is not a terminal state then we go back to step 5.

As you can see from the last step this process will repeat itself untill $n > N$. Meaning that the larger the N value the longer this process will take and the more accurate the result should be. After this method is completed we will be left with a Q-table that is full of values that have been iterated over many times.

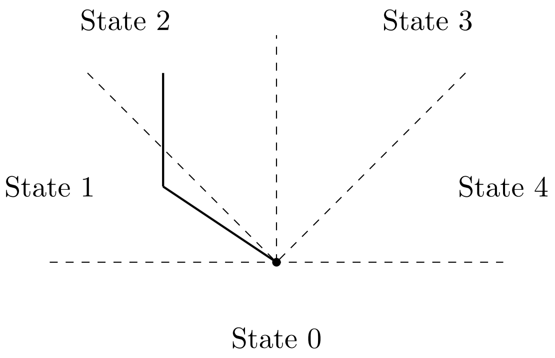
5.2.4 Finding a Policy

After training is complete we have a Q-table that is full of values that represent how good each state action pair is from the algorithm's trials during training. In order to turn this into a policy we must make the Q-Table a map from states to actions. This is normally done by simply taking the highest Q-value action for each state to be the action that the said state maps to. Formally this is $\pi\mathcal{S} \to \mathcal{A} = \max_{a\in\mathcal{A}(s)} Q(s,a), \ \forall s \in \mathcal{S}$. By construction this is very similar to our exploitation based approach. This means if we had an optimal Q-table it could be turned into an optimal policy using this method. However obtaining an optimal Q-table is a much more difficult problem. It has been proven by Francisco S. Melo[10] that the Q-learning method described above converges to the optimal Q-table and therefore the optimal policy as long as every state action pair is visited infinitely often. Although this is useful as it theoretically establishes the convergence of Q-learning and establishes its theoretical optimally visiting every state infinitely often is impossible in a real setting. The larger the state and action spaces become the more unrealistic visiting every state action pair regularly becomes.

5.3 Linear Q-learning

While the q-table is incredible powerful for some reinforcement learning problems, where the state space is small, it is impractical to model the large MDPs that arise in problems such as chess and go. Claud Shannon estimates the number of possible position of chess to be 10^{42}[12]. It would be impossible to hold a this many positions in the memory of a computer. In continuous action spaces used in robotics it is impossible to represent the action space with a q -table. To overcome this challenge we introduce the idea of a linear function approximator. rather than calculate every state of a q-table, we approximate the true the optimal policy q_π using a linear combination of state-action features, ϕ.

We define an action space as the figure below, where the state of the system is defined by the location of the end of the second pendulum.



We define two actions that can take place in each state, apply a force left or apply a force right. We can simply take each features to be the continuous values given by each control variable, namely

θ_0, θ_1, θ_2, $\dot{\theta}_0$, $\dot{\theta}_1$, $\dot{\theta}_2$. We also need a feature that captures the relationship between the two arms. We take another feature to be the horizontal difference between the end of either arm. Finally we take the difference between the tip of the second arm and 2*L. We take the dot product of each feature vector and a vector of weights. It is these weights that the linear Q-learning algorithm will be optimizing.

$$\phi(s,a) = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \dot{\theta}_0 \\ \dot{\theta}_1 \\ \dot{\theta}_2 \\ y_2 - y_1 \\ 2L - y_2 \end{bmatrix} \cdot \begin{bmatrix} W_1 \\ W_2 \\ W_3 \\ W_4 \\ W_5 \\ W_6 \\ W_7 \end{bmatrix}^{\top}$$

The Q function becomes:

$$Q(s,a) = \sum_{i=0}^{n} \phi(s,a) \cdot W_i$$

Then for any given state we calculate the Q value for each action, left or right and choose the the larger of the two.

6 Gradient Descent

To find an optimal approximation to our Q-policy we iteratively update our weight vectors using gradient descent. We define a loss function

$$L = \frac{1}{2}(Q - \hat{Q})^2 \tag{6}$$

which is the mean squared error of the difference between our current estimate of the Q-function and our last estimate. Since this is a vector valued function we take the gradient with respect to the weight :

$$\begin{aligned} \nabla_w L &= \frac{\partial Q}{\partial Q}(\frac{1}{2}(Q - \hat{Q})^2) \\ &= (Q - \hat{Q}) \cdot \frac{\partial Q}{\partial Q}(Q - \hat{Q}) \\ &= (Q - \hat{Q}) \cdot \frac{\partial Q}{\partial Q}(\vec{\phi} \cdot \vec{w} - \vec{\phi} \cdot \hat{\vec{w}}) \\ &= (Q - \hat{Q})\vec{\phi} \end{aligned} \tag{7}$$

Using a bellman optimal update rule we get:

$$\Delta w = \alpha(R_t + 1 + \gamma \hat{q}(s_{t+1}, a_{t+1}, w) - \hat{q}(s_t, a_t, w))\vec{\phi}$$

In this way, the weights that get adjusted the most are those that are associated with the features that take the largest value. Our final algorithm is as follows:

```
Algorithm 1: Linear Q-Learning
  Input: Initial parameters $\mathbf{w}_0$
  Output: Optimal Q-function $\mathbf{q}^*$
  repeat
      Start new episode;
      Initialize state $s_0$ and time step $t = 0$;
      repeat
          Choose action $a_t$ from state $s_t$ by $\epsilon$-greedy policy with respect to Q-function;
          estimate $\hat{\mathbf{q}}(s_t, \mathbf{a}, \mathbf{w}_t)$;
          Take action $a_t$;
          Observe reward $r_{t+1}$ and next state $s_{t+1}$;
          Compute error: $\delta_t = r_{t+1} + \gamma \max_{\mathbf{a}} \hat{\mathbf{q}}(s_{t+1}, \mathbf{a}, \mathbf{w}_t) - \hat{\mathbf{q}}(s_t, a_t, \mathbf{w}_t)$;
          Update parameters: $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \delta_t \phi(s_t, a_t)$;
          Increment time step: $t \leftarrow t + 1$;
      until failed episode or maximum time steps reached;
  until convergence;
  return $\mathbf{q}^*(s, a) = \hat{\mathbf{q}}(s, a, \mathbf{w})$
```

7 Results and Conclusion

7.1 Our Implementation

We have successfully implemented a code simulation of the DPOAC problem. It is based of the cartpole.py[3] file from openAI gym's original environment. We modified the code to use the first order differential equations from the results above to represent the mathematical model of the DPOAC system. We then used an existing module from scipy to call Runge-Katta of order 8 to solve our system. We render the image every 0.02 seconds to represent each timestep. Some constraints include

- We have a $10s$ duration for each episode. Meaning we have a total of 500 timesteps per episode.
- We limit the range of how far the Cart is able to move to only fit the screen. [4]

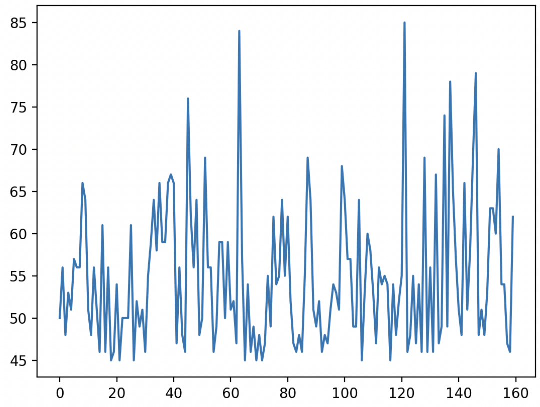
We tried implementing Q-learning for our problem. The agent starts randomly at ± 0.1 radian away for both the pendulums from the unstable upright position. Below are some additional constraints we used to end an episode and consider it as a fail during the learning process of our agent

- If the cart goes out of bounds or out of screen
- If either one of the pendulums falls below the horizontal line

[3] This is a file that simulates movement of single pendulum on a cart

[4] We left it the same as the original cartpole.py parameters

However, our result shows that the Q-table was not able converge. Below is a sample result graph we plotted after every 160 episodes (x-axis) and the number of timestep the agent was able to stay up before the episode ends (y-axis) [5].



However, after running over 8000 episodes, on average, the agent can only balance up to 55.15625 timesteps which is equivalent to approximately 1.1 seconds per 10 seconds episode. This is likely because our action and state space is very large and Q-learning is more suitable for simpler problems with smaller and discrete action space. In this regard, although Q-learning has been successfully implemented on a single pendulum on a cart before [9]. However, when it comes to a double pendulum on a cart problem, the chaotic nature of the system makes it extremely hard for our agent to learn.

7.2 Future Work

Future work for us includes implementing the linear approximation Q-learning we have studied and broke down earlier in the paper. If given more time we would have implemented it as well. We assume that the linear approximation Q-learning would be suitable for our problem because the large state space lead us to difficulties training a normal Q-learning algorithm. Linear approximation Q-learning has also been used for this problem successfully before. Feature vector selection is a critical part of linear approximation Q-learning and by implementing the methods for feature vector selection outlined in Gaston Baudat's paper[1] We would expect to see improvements in accuracy over previous linear approximation Q-learning implementations to the hold up problem before. As these well defined methods for feature vector selection have not been used yet for this problem.

Future work could also include using the model based reinforcement learning algorithm PILCO[4] to attempt to solve even harder problems such as the ”Swing up” problem in which the DPOAC system starts with both arms pointing downwards and then PILCO must use the cart to swing the system up into different equilibrium positions or the ”hold up” position. Once we have both of these future results completed we would have enough different results and implementations to do a significant statistical comparison between the various results and conclude which was the most

[5] This is out of 500 timesteps

successful using different metrics.We could also compare against other implementations of machine learning algorithms and classical robotic solutions for the DPOAC system.

7.3 Conclusion

The DPOAC is a chaotic system that can be controlled by machine learning algorithms as shown by Gustafsson[7]. However in order to achieve this a more advanced variation of Q-learning must be used or a different algorithm entirely. This is because of the continuous nature of the angles. Turning these continuous angles into a discrete state space proves to be a challenging task as you must weigh between too few states and losing alot of information in the conversion and too many states which make training the algorithm require far too much time. One example would be a successful implementation of a single pendulum on a cart with q learning required over 100 states[9], if we extended this same model to the DPOAC it would be over 10000 states. In order to alleviate this methods such as Linear approximation Q-learning or more advanced reinforcement learning algorithms are required. This paper served as a good breakdown of the techniques used to simulate the DPOAC and the machine learning method that we attempted to apply to it and a more advanced method that has been shown to work before. It also served as a illustration of one of the weaknesses of the base Q-learning algorithm and why a user may want to utilize Deep Q-learning or linear approximation Q-learning for future use or even other algorithms that are more suited to continuous state-action spaces.

References

[1] G. Baudat and F. Anouar. Feature vector selection and projection using kernels. *Neurocomputing*, 55(1):21–38, 2003. Support Vector Machines.

[2] Alexander Bogdanov. Optimal control of a double inverted pendulum on a cart. *Department of Computer Science & Electrical Engineering, OGI School of Science & Engineering, OHSU Technical Report CSE-04-006*, 2004.

[3] Ian J P. Crowe-Wright. Control theory: The double pendulum inverted on a cart. *https://digitalrepository.unm.edu/math_etds/132*, 2018.

[4] Marc Deisenroth and Carl Rasmussen. Pilco: A model-based and data-efficient approach to policy search. pages 465–472, 01 2011.

[5] Sutton R. S. & Barto A. G. *Reinforcement learning: An introduction (2nd ed.)*. 2018.

[6] Knut Graichen, Michael Treuer, and Michael Zeitz. Swing-up of the double pendulum on a cart by feedforward and feedback control with experimental validation. *Automatica*, 43(1):63–71, 2007.

[7] Fredrik Gustafsson. Control of inverted double pendulum using reinforcement learning. *Retrieved from Stanford University CS229*, 2016.

[8] R. Levien and Sze Tan. Double pendulum: An experiment in chaos. *American Journal of Physics - AMER J PHYS*, 61:1038–1044, 11 1993.

[9] Cy Liou. Q-learning demo (solve the inverted pendulum problem), 2005.

[10] Francisco S. Melo. Convergence of q-learning: a simple proof, 2008.

[11] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 4 edition, 2022.

[12] Claude E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41(314), March 1950.

[13] John R. (John Robert) Taylor. *Classical mechanics. Sausalito, Calif. :University Science Books, ISBN 13: 9781891389221.* 1939-. (2005).

[14] Martijn van Otterlo and Marco A Wiering. Markov decision processes: Concepts and algorithms. 2012.