Trajectory Planning for Non-Differentially Flat Systems using Reinforcement Learning framework

Master Thesis

Ecole Polytechnique Fédérale de Lausanne

Student : Oscar Jenot

Supervisor : Phillippe Müllhaupt

Assistant : Willson Sudarsandhari Shibani

June 11, 2021

Table of Contents

Introduction

Motivation and objectives

Differentially flat systems such as a car with n-trailers and some classes of cranes have the physical or geometric properties of flatness. Trajectory planning and the construction of associated inputs are obtained by basic differentiation of a sufficiently smooth path of the flat outputs. The relationship between flatness and controllability is tight, as flat systems can have their output linearized. Non-differentially flat systems can be controlled by different methods like high frequency control (Fliess et al., 1995), and / or by adding control variables to make the system differentially flat.

The objective of this project is to study the resolution of this problem using machine learning, and more specifically reinforcement learning, in the case of non-flat systems obtained by a perturbation of flat systems, for example by changing the geometry and / or removing control variables.

The methods used will be to create a pure neural network based controller, trained based on a model free reinforcement learning framework, in order to determine if such method can accomplish the task of controlling underactuated nonlinear systems.

The exemples that will be treated are three classes of cranes. The cart pendulum, an underactuated overhead crane (transporting a masspoint), and a fixed length crane transporting a vertical load will be presented in the case study. These systems lose the flatness property when the crane displaces a very long object vertically.

One question is to know to what extent the knowledge of the flatness property of the unperturbed system helps in solving the variational problem of the perturbed case.

The case study shows that a simple Deep Q-Network reinforcement learning algorithm is able to perfectly control a flat crane. When adding complexity to the system, that same learning algorithm is struggling to stabilize the system and oscillate around the equilibrium point.

Structure

The report is organized as follow :

- **Background** : The concepts of differential flatness and reinforcement learning for optimal control is introduced.
- **Empirical Research:** A brief overview of some examples where reinforcement learning was used for optimal control, and their strengths and weaknesses.
- **Methodology** : Firstly, an explanation on how to implement a reinforcement learning environment specific to control systems is presented, followed by a presentation on

the theory of optimal control policy in a markov environment. The learning algorithms and class of Neural Network used in the simulations will be presented.

- **Case Study** : We will investigate the resolution of the flat and non-flat crane systems using reinforcement learning algorithms, in an intent to contribute to the still developing research in the field.
- **Conclusion and Discussion :** A summary of the work done, the methods used, results, and a prospect for future work will be done.

Background

Understanding Flatness Theory

In order to understand the project's challenges, the characteristics of differentially flat systems must be comprehended. Understanding the properties of flat systems (such as the simple crane and the standard n-trailer system) and how to control them will help us understand what perturbations may cause these systems to become non-flat. One of the main publication on the subject (Fliess et al., 1995) describes the differential flatness property as followed :

"One major property of differential flatness is that (...) the state and input variables can be directly expressed, without integrating any differential equation, in terms of the flat output and a finite number of its derivatives. (...) Let us emphasize on the fact that this property may be extremely useful when dealing with trajectories: from y trajectories, x and u trajectories are immediately deduced" (Fliess et al., 1995, p. 3).

In the previous publication, the exemples of the flat crane and the flat (standard) n-trailer system are presented and solved via elementary properties of planar curves. The dynamic equations found in Fliess et al. (1995), Rouchon et al. (1993), Fliess et al. (1993) and Sudarsandhari Shibani et al. (2011) for the flat crane and the inverted cart pendulum will be used to build our artificial-intelligence (AI) based reinforcement learning (RL) model, in order to solve their motion planning using this different approach. Understanding which geometric perturbations will make these systems lose their flatness is crucial in order to determine if an AI is able to solve the perturbed (non-flat system) problem.

Reinforcement Learning for System Control

Reinforcement learning (RL) is a field of machine learning (along with supervised learning and unsupervised learning) modeled as a Markov decision process (MDP).

RL is a framework based on an *agent-environment* interface. The *agent* (the controller, in the engineering field), interacts with the *environment* (the controlled system or the plant). In each discrete time step, the agent takes an *action* (the control signal). Depending on the action taken, the state of the environment will change. The new state, and a numerical reward is then given by the environment to the agent. Iteratively, the agent decides on a new action to take. The policy π^* is a function of the state-action space that specifies which action the agent should take for each given state. The objective of the RL problem is to find an optimal policy π^* so that the agent maximizes the cumulative reward in each episode. A wide variety of different algorithms (e.g. DQN Deep Q-Network) can be used to solve different types of RL problems. This discrete-time stochastic control process called the Markov Decision process (MDP) is represented in Figure 1 for a RL agent-environment interface. (Stutton and Barto, 2018; Li et al., 2017).

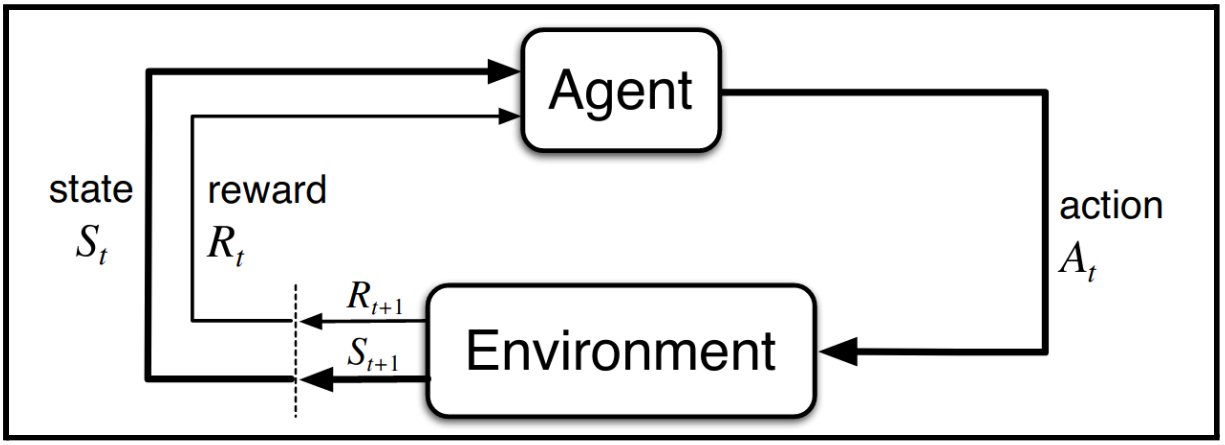


Figure 1: The agent-environment interaction (Stutton and Barto, 2018, p. 48).

Reinforcement learning has been used to solve a diverse set of problems, such as gaming (Google DeepMind's AlphaGo Zero)[1], autonomous driving (AWS DeepRacer)[2], robotic manipulation (Google AI's robots)[3], and much more. RL is also used in optimal control theory and automatic control fields, as it "offers powerful algorithms to search for optimal controllers of systems with nonlinear, possibly stochastic dynamics that are unknown or highly uncertain." (Buşoniu et al., 2018).

Thanks to MathWorks[4], an analogy between an AI-based RL system's control architecture and a classic automatic control scheme is shown in Figure 2.

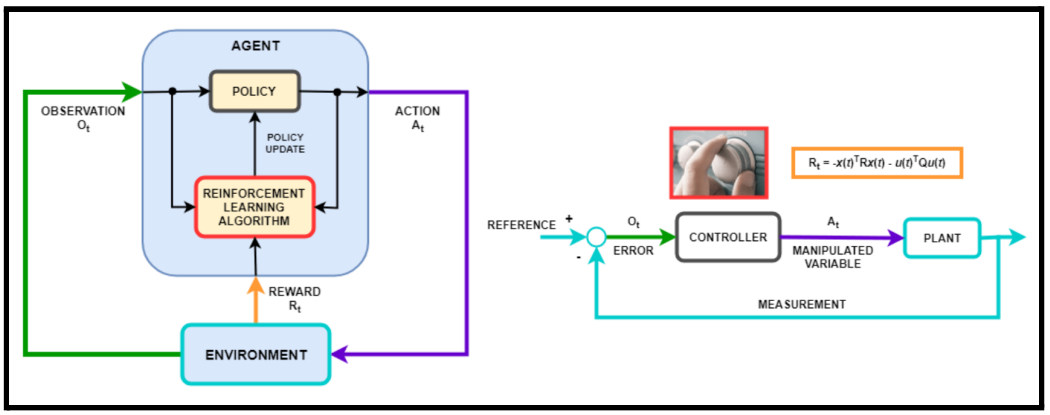


Figure 2: Reinforcement Learning for Control Systems Applications (n.d.).

[1] (n.d.). AlphaGo | DeepMind. Retrieved April 30, 2021, from https://deepmind.com/research/alphago/
[2] (n.d.). AWS DeepRacer. Retrieved April 30, 2021, from https://aws.amazon.com/deepracer/
[3] (n.d.). Google AI. Retrieved April 30, 2021, from https://ai.google/
[4] (n.d.). Reinforcement Learning for Control Systems Applications - MATLAB. Retrieved April 30, 2021, from https://www.mathworks.com/help/reinforcement-learning/ug/reinforcement-learning-for-control-systems-applications.html

Empirical research

As seen in the previous part, the analogy between a controller and a reinforcement learning framework is very close in its structure. In this part, we will look at exemples where RL methods have been successful in finding the optimal controleur and trajectory of complexe nonlinear, and perhaps non-flat systems.

Using reinforcement learning models for controlling complex systems are quite reliable, and stay robust even with large perturbations.

Bejar & Moran (2018) have developed a reinforcement learning based control strategy for parking autonomous truck-trailers. A deep deterministic policy gradient (DDPG) algorithm was used to train the neural network based controller and shows similar performance to classic controllers. The trailer system studied here was flat but the controller did not require a linearization procedure.

Ding & Wiering, (2018) tried to solve the problem of the offshore crane set down operation (with external disturbance such as wind, waves, etc.) and comparing different RL models' performance. Their models' performance shows some optimism for the use of RL in trajectory planning of mechanics.

Optimal control versus Reinforcement Learning

The Cart-Pole problem is a very basic system to experiment designing controllers in both the reinforcement learning and control engineering field. It seems like a perfect example to compare the advantages and disadvantages, the similarities and differences of each control strategy.

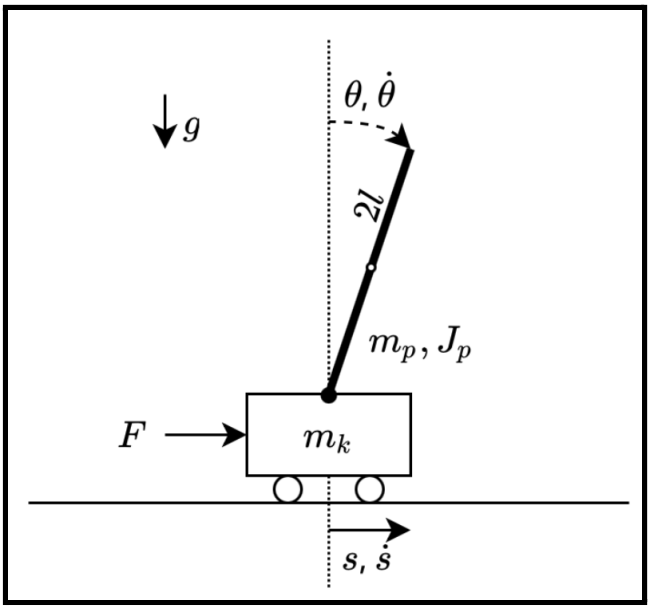


Figure 3: Dynamics of the Cart-Pole, image by Paul Brunzema (2021).

In Paul Brunzema[5]'s article *Optimal Control with OpenAI Gym,* the author uses the Cart-Pole problem to compare Reinforcement Learning with Optimal Control. The objectif on the problem is to balance a pole on a cart for a certain threshold of time. By performing a linearization around the upper equilibrium of the system, an LQR controller is designed. The results are impressive because designing the controller instead of learning it using RL results in convergence for 100% of the episodes, and needs zero iteration of training.

If we compare it to Matthew Chan[6]'s implementation of the same problem using a RL Q-Learning Policy, the model needs to be trained for 136 iterations before converging.

In this case, the LQR is very performant and finds the optimal control strategy. But the model of the system was known, since the system's dynamics were given to the controller. Using reinforcement learning to solve the same problem does not require any prior knowledge of the system, since it will learn the model through an algorithm.

A lot of examples that are partially solved using reinforcement learning seem to solve uncomplicated systems. When complexe systems are solved, they often require the use of approximation tools in order to simplify the environment that the model has to be trained of, such as in Chen et al. (2019)'s double pendulum cart. Furthermore, since RL requires discrete time steps and is often trained on an unperturbed environment, implementing the model to real life experiment that encounters changing operating conditions is a challenge (Sindhu Padakandla, 2005). Solving complexe real life environment's can require the use of a combination of AI methods and classic control theory to adapt to changing operating conditions (Rosolia & Borrelli, 2018).

	Optimal Control	Reinforcement Learning
Model requirement	Model based	Model free / model based
Optimal control algorithms	Stays fixe	Iteratively improves

Table 1 : Main differences between optimal control and RL.

The promising advances made in deep neural network based controllers and the more recent use of reinforcement learning framework to train them makes us believe that this tool might be an interesting to adopt for our case study.

In the following part, the concepts of RL environment applied to control systems, followed by an overview of the AI and RL tools used in the case study will be presented.

[5] "Optimal Control with OpenAI Gym. Comparing Optimal Control and ...." https://towardsdatascience.com/comparing-optimal-control-and-reinforcement-learning-using-the-cart-pole-swing-up-openai-gym-772636bc48f4. Accessed 10 Jun. 2021.

[6] "Cart-Pole Balancing with Q-Learning | by Matthew Chan | Medium." 13 Nov. 2016, https://medium.com/@tuzzer/cart-pole-balancing-with-q-learning-b54c6068d947. Accessed 10 Jun. 2021.

Methodology

Environment

In order to interact with the Reinforcement learning software, a program representing the environment should be written. The environment should have the following traits (Sewak et al., 2020):

A state representation. A state representation or observation space is the state of the system at a current time step. The state representation should have a Markov property, which means that the future state of the environment should only be determined from the present state, thus be independent of the past history (Grabski, 2015).

For physical systems, the equation of motion in the form of second order ordinary differential equations (ODE) is ideal. By transforming the equation of motion into continuous second order ODEs, we can use a numerical integrator in order to obtain the next state.

In the case of a physical system with the states being the position and velocities, the Euler method (Euler, 1792) is used to produce an explicit discrete state at the next time step.

For our case study, given a position vector x and velocity vector x', an computed acceleration vector x'' and a step size of τ, the formula to get the new state is :

$$x_{t+1} = x_t + \tau \cdot x'_t$$

$$x'_{t+1} = x'_t + \tau \cdot x''_t$$

The state can be passed as a parameter to other internal functions of the environment (to compute que equations of motions for example).

An action representation. The action representation is the input that the agent gives to the environment. It can be an integer (for a constant signal), an array (for multiple inputs), or any information that will interact with the environment's state representation. Depending on the learning algorithm used, a discrete action state can be demanded (for Deep Q-Networks for example).

A reward signal. (Sutton., & Barto (2018)) The reward is an essential module of the environment as it will be used by the learning algorithm to determine the optimal policy. The reward is a Reel number and should only be a function of the current action, state and next state *a, s, s'*, and random amount (a constant for example).

Designing a reward signal should be focused on the goal that needs to be reached by the agent rather than how it should be reached, or you could prevent the learning algorithm from finding the optimal policy.

In our case study, a very large sparse reward is needed when the system is stabilized at the desired equilibrium and thus achieves the goal. Since the state space is infinite (all positions, velocities, angles and angular velocities can be explored), the model does not perform well

at randomly finding the target. It is thus not able to dig the sparse reward. In order to help the agent converge to the optimal solution, smaller intermittent rewards will be added in order to guide the crane to the target and help him follow the right directions. This was made mainly to reduce computational time, and does not seem to affect the optimal trajectory chosen for the solved system.

In practice, reward shaping is often lest to trial-and-error. While designing the reward signal, we need to judge the performance of a reward by observing if the agent is translating the reward to the goals that we want him to reach.

A step function. This function is the link between the action, state, and reward. It should take the action as an input, and return the next state *s',* and the immediate reward *R'*, and some other information (like a 'done' signal if the episode is solved) to the agent.

A concrete implementation of the three environments can be found in the Appendix.

Reinforcement Learning Algorithm

Finding the optimal control policy in Markov environment

As said before, to obtain a control policy for our systems, we propose the use of reinforcement learning algorithms. Such tools have various levels of sophistication, but all rely on a cumulated expected returns frameworks for efficiency.

The use of RL is based on the concept of learning through interactions, which can be described as a Markov Decision Process (MDP) (Arulkumaran et al., 2017). Indeed, the controls of our crane systems will be achieved in a sequence of states, where the probability of transitioning to the next state depends on the interpretation of the previous state (Sewak et al. 2020). This transition of state represents the *decision.*

In an MDP, the sequential decision-making formalization relies on a decision maker (the agent) selecting an action ***a*** from a given state ***s*** and receiving a numerical reward *R* for transitioning to a new state s_{t+1}. In this sense and assuming *s, a* and *R* have finite numbers of elements, the goal of the MDP and the agent will be to maximize its expected return G, that is the cumulative rewards it receives until the final time step T (Sewak et al., 2020)**.**

$$G_t = R_{t+1} + R_{t+2} + \ldots + R_T$$

For that, the agent is informed of its state ($s_t \in S$) and selects an action ($a_t \in A$), giving for each time step a state-action pair (s_t, a_t). As the environment transitions to s_{t+1} , the agent receives the reward ($R_{t+1} \in R$), allowing it for evaluating the relevance of taking a_t from s_t.

In the context of the present research, a valid control policy will allow the agent to find the optimal trajectory of the control problem. Therefore, it is possible to divide the agent-environment interactions necessary to reach this objective in subsequences. In this sense, the task is episodic, and an episode ends when the agent manages to reach the

target state. Nevertheless, during the learning phase, the number of timesteps before T are not known. Therefore, an arbitrary number of maximum time-steps is set, after which each episode will end even if the target state is not reached at that point.

It follows that both **T** and related G_{T-1} can be infinite. Therefore, to allow the expected return to converge, a **discount rate** γ is introduced and the goal of the agent is updated:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots$$

When selecting actions depending on its state, the agent will continue to consider future rewards but will favor immediate ones, especially if γ is close to 0.

Optimal policies and value-functions

The agent's selection of an action follows a policy that maps a given state to the probabilities of selecting each possible action from that state. In this sense, if the agent follows the policy π at time *t*, $\pi(s)$ represents the probability of it selecting action *a* in state *s* such that $A_t = a$ if $S_t = s$.

The state-value function v is computed for evaluating a given policy π. It expresses the expectation (E) of the discounted return of following π from a given state s for the agent. The state-value function for a given time *t* can be expressed as:

$$v_\pi(s) = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s\right]$$

The expected cumulative reward from taking action *a* under policy π when in state *s* on the other hand, will be given for time *t* by an action-value or q.function:

$$q_\pi(s,a) = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s,\ A_t = a\ \right]$$

The output of this q.function expresses the value of a given state-action (s,a) pair at t, in terms of its contribution to the expected return.

In system control problems for instance, the goal is to find a policy π that will yield more return for the agent than any other. Several policies may satisfy this requirement and share the same optimal state-value function (Sewak et al., 2020):

$$v_*(s) = max\ v_\pi(s)$$

Similarly, the largest expected return achievable by any policy π for each state-action pair will be given by the optimal q-function, defined as:

$$q_*(s,a) = max\ q_\pi(s,a)$$

The q-function must satisfy the Bellman optimality equation:

$$q_*(s,a) = E\left[R_{t+1} + \gamma \cdot max\, q_*(s',a')\right]$$

Where s' is the state from which the best possible action a' can be taken at *t+1*. This optimal q.value, $q_*(s',a')$ defines the agent's target-q (Sewak et al., 2020).

Q.Learning

Based on the optimal action-value function, the agent can now determine the actions that maximize the q.value for each state and search the optimal policy. To that end, a reinforcement learning method can be used to iteratively update q-values for each (s,a) until the q-function converges to q_*. These processes and corresponding algorithms are referred to as q.learning, since they enable the agent to take into account its previous experiences by updating its q.values.

In the starting state of the first episode, the agent has no information about its environment. All q.values are equal to zero and the agent cannot differentiate the values of different possible actions. Therefore explorative interaction with the environment is needed for the agent to understand different rewards yielded by various (s,a) pairs. Nevertheless, to reach its goal of maximizing expected returns, the agent must 'learn' about its environment and exploit valuable known (s,a) and reward combinations to its advantage. The tradeoff between exploration and exploitation is formalized by the **exploration rate** ε which equals 1 at the initial episode in our case. This means that for its first interaction with the environment, the agent will have a 100% chance to act via exploration. Then, the agent is set to adopt an epsilon greedy strategy that favors exploitation, with the intent of maximizing reward returns. For implementing the greedy strategy, ε is decayed as the agent learns about its environment. To that end, an **exploration decay** rate is set for reducing the chances of the agent taking exploratory actions from one time step to the other. Practically, ε is decayed at the beginning of each episode, setting the probability of the agent choosing exploration over exploitation within the steps to come. For each time-step, a number between 0 and 1 is generated: if the number is greater than ε, the agent selects an action via exploitation, otherwise exploration will be carried out by choosing a random action.

As mentioned, the agent receives a reward to inform the value of taking an action in a given state. This information is used to update the (s,a) pairs' q.value according to the Bellman equation and target-q. Over time, q.learning aims at iteratively comparing the q.value and q_* for any given (s,a) pair. This allows the agent to select actions that will reduce the difference between the two values, the *loss,* maximizing its expected returns.

$$q_*(s,a) - q(s,a) = loss$$

$$E\left[R_{t+1} + \gamma \cdot max\, q_*(s',a')\right] - E\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right] = loss$$

In order to account for the agent's acquired experience and changes in expected returns, the q.value compared to q_* is not given by the calculation of a new q.value for a given (s,a) pair, but rather an update. The **learning rate** α is set for balancing how much information will be kept from the q.values calculated in previous timesteps in the updated q.value. Throughout the process, α remains constant. If α=1 there will be no trace of old q.values in the *q(s,a)* update. In this sense, α determines the agent's learning speed by decaying the influence of past experiences so that the q.value updated results are a weighted average of old and new values.

Based on the parameters discussed and for a given (s,a) pair at time t, the q.learning algorithm updates the associated q.value such that:

$$q^{new}(s,a) = (1 - \alpha)\, q(s,a) + \alpha(R_{t+1} + \gamma \cdot max\, q(s', a'))$$

Going back to the control problem and the greedy policy adopted for its resolution, the q.function can now be updated and used to inform the agent of the value of following a given policy for a given state at a given time:

$$v^{\pi}(s_t) = max\, q^{\pi}(s_t, a)$$

Therefore, the agent will follow a policy π and chose action a for a given state if and only if:

$$q^{\pi}(s,a) > v^{\pi}(s)$$

At this point, the agent has all the statistical tools needed for q.learning in order to improve its policies for controlling the system. Nevertheless, convergence of q(s,a) in q* may require that all states are infinitely visited and each action is taken an infinite number of times, especially in complex environments. For avoiding this issue and supporting policy optimization, combining reinforcement learning with neural networks has proven efficient (Arulkumaran, 2017; Sewak et al. 2020).

Deep q.learning

Instead of using value iteration for finding the optimal q.function, reinforcement learning can be conducted using a function approximation (Sewak et al. 2020). For that, a deep neural network is added to the RL model in order to approximate the optimal q.function based on the Bellman equation. This process and the network that supports it are referred to as deep q.learning and deep q.learning network (DQN).

For our systems, the network accepts states of the environment as input, (position, angle, etc.). Then, 2 fully connected hidden layers output the q.values for all possible actions that can be taken based on the input and output of the action space dimension. The use of the term "deep" is related to the use of several layers in the network model. In the context of the present research, the objective on the network is that for a given state, a forward pass of the DQN will allow for the action that better maximizes the expected return to be chosen by the agent.

Fully connected deep networks are structure agnostic, meaning they require no special assumptions regarding the input they accept. Therefore, such networks may be outperformed by some tailored to specific problems, yet fully connected networks are used as function approximators in a variety of applications (Bosagh Zadeh & Ramsundar, 2018). It can be said that fully connected networks represent the straightest forward approach to neural networks, literally but also for application ease. Indeed, a fully connected neural network simply consists of a series of fully connected layers that can be understood as a function from R^m to R^n. Now by posing $x \in R^m$ as the layers' input and $y_i \in R$ the *i*-th output of the fully connected layer, then the output can me computed such as:

$$y_i = \sigma(w_1 x_1 + \ldots + w_m x_m)$$

Here σ is a (nonlinear) activation function and w_i **the learning parameters** or weights of the network. The total output of the fully connected layer will be given by the sum of its individual outputs:

$$y = \begin{matrix} \sigma\left(w_1 x_1 + \ldots + w_m x_m\right) \\ \vdots \\ \sigma\left(w_{n,1} x_1 + \ldots + w_{n,m} x_m\right) \end{matrix}$$

As mentioned, the RL problem is to find the optimal policy for each (s,a) pair. For that and avoiding the need to perform an exhaustive research through all possible q.values, the deep neural network used needs to learn an approximation of the q.function. This action-value function approximation, referred to as q.theta, is defined as:

$$\hat{Q}_\theta(s, a \mid \theta) \sim q_*(s, a)$$

Where θ represents the learning parameter vector, that is the values for all the weights in the network.

Going back to q.learning theory, the DQN's objective is to minimize the difference between the action-value function approximation and the optimal q.value, while satisfying the Bellman Equation. In this sense, the objective of the DQN can be expressed as:

$$\sum_{e \in E} \sum_{t=0}^{T} \hat{Q}\left(s_t, a_t \mid \theta\right) - \left(r_t + \gamma \cdot max\, \hat{Q}\left(s_{t+1}, a' \mid \theta\right)\right)$$

Note that this objective function is fully differentiable, which will allow the DQN to find gradients and perform stochastic gradient descent for minimizing it (Bosagh Zadeh & Ramsundar, 2018).

Training the network

To meet its objective, the DQN will undergo a training where the weights of all the connections it has will be defined. During this phase, many training examples are feeded to the network and the weights are iteratively modified in the goal to reduce the *loss*. To

optimize the weights and encourage the agent to take actions that lead to high rewards, the stochastic gradient descent (SGD) algorithm is used to train individual neurons (Buduma & Locascio, 2017). Typically used in supervised learning, SDG can be applied to RL for optimizing the weights by using policy gradients. This method allows for scaling the *loss* by the expected return of an action, so, for example, if the model choses an action yielding negative return, the *loss* is amplified (Buduma & Locascio, 2017). This amplification of penalties in case of bad decision is strengthened by using the log probability of the model choosing that action. In other words, in the context of RL, SGD minimizes the *loss* as it optimizes the expression:

$$arg\ min\ \ -\sum_{t} G_t \log(\, p(a_t|\, x_t;\ \theta\,))$$

As a reminder: a_t is the action taken at time step *t*, x_t the layers' input (which will be the state for neurons of the first layer) and G_t the discounted expected return. In practice, the objective function is researched by using the ADAM optimizer, which adjusts the weights θ of the model according to the policy gradients. In order to expand the learning to all layers of the network (that may be defined as $f(\theta,\ x)$), the backpropagation algorithm is used with the logic of computing $\frac{\delta f}{\delta \theta}$ (Bosagh Zadeh & Ramsundar, 2018).

For speeding up the learning process, at every iteration, SGD is not computed on the full dataset but rather on a subset of it. This method, referred to as *minibatching,* aims at supporting the *loss* convergence towards zero by allowing for more gradient descent steps to be taken with the same amount of computations (Buduma & Locascio, 2017). The weight update performed through backpropagation can be expressed as:

$$\Delta w_{ij} = \sum_{k \in minibatch} \in y_i^{(k)} y_j^{(k)} \left(1 - y_j^{(k)}\right) \frac{\delta E^{(k)}}{\delta y_j^{(k)}}$$

Where *i* is a neuron from the previous layer and y_j the output of neuron on layer observed. Also, *E* is the mean squared error (loss) between each element of the input *x* and target *y* so that $w_{ij}^{(k)}$ can be understood as the weight of the connection between the ith neuron from layer *k* and the jth neuron in layer *k+1.*

Target q.network

Going back to the objective of the DQN, the *loss* is here defined as the difference between the predicted q.values for state s and that of state s+1 in the case where the best possible action is selected. The fact that this function is doubly dependent on the parameters of the model bares the risk of feedback loops arising during the learning, caused by high correlation of q.values updates (Buduma & Locascio, 2017).

Therefore, a second network is added to the model. Referred to as target network, its aim is to mirror the prediction network but with less frequent updates so that it becomes a more stable data point for loss minimization. In practice, this means that instead of updating a

single network frequently with itself, the weights for the approximation of the q.function will be drawn from the prediction network. The approximation of the q.value at next time step if the best action is selected ($\hat{Q}(s_{t+1}, a' | \theta)$) will be drawn from the target network. By updating the target network only every few batches of the training, the q.values updates become more stable and improve the learning of a valid q.function (Bosagh Zadeh & Ramsundar, 2018).

Experience replay

The last hyperparameter added to the model also aims at reducing the instability of q.learning, this time by breaking the high correlation that may exist between action-state pairs that follow each other. To that end, the agent's experiences are stored in a memory buffer table from which a batch of samples are drawn (Sewak et al., 2020). The experiences are stored as tuples (s_t, a_t, R_t, s_{t+1}) and allow for the computation of the *loss* function and subsequent optimization of the network, while reducing the risk of overfitting for the learned function. Nevertheless, for the agent to fully exploit its learning, the batch sample used in experience replay must be updated so that it does not contain too old experiences.

Learning outcome

The deep q.learning network may now approximate the q.function that represents the value of a given state-action (s,a) pair at time t, in terms of its contribution to the expected return.

Therefore, as for q.learning, once the network has found an optimal q.function, expressing it in terms of optimal policy for the agent to follow is the last necessary step:

$$\pi(s; \theta) = argmax\ \hat{Q}^{*}(s, a'; \theta)$$

Neural Network Architecture

One single hidden layer with a finite number of nodes would be able to estimate continuous functions (Heaton, 2008), but only from a finite space to the other. As our observation space is infinite, at least two hidden layers are needed. A two hidden layer architecture can approximate any smooth mapping from infinite spaces (Heaton, 2008). Adding more hidden layers to our architecture can result in learning more complexe mapping functions, but would require a larger computation time, and is more used in computer vision (Heaton, 2008).

In order to prevent underfitting or overfitting, the correct number of neurons should be chosen. With a complexe data set, not having enough neurons might not detect the correct function approximation and complexity of our system and result in underfitting (Heaton, 2008). On the other hand, having too many neurons might learn an exact mapping of every state-action value of the trained set, but not adapt well to new inputs, and cause overfitting.

With that in mind, a trial and error method is required to optimize the network architecture, taking into account training results and testing results and computational time (Heaton, 2008).

Case Study

The objective of the crane case study is, using an iterative method, to create successive crane environments in order to model the non-flat crane's final environment. At each iteration, the system will be closer to reality. The Deep Q-Network RL model trained on each environment will only be lightly adapted and hyper parameters will remain the same for every new iteration. This way, their performance will be set side by side, and we will gain insights on the learning algorithm's capacity to solve complex systems.

If the learning algorithm is trained on a simple flat system and solves it, to what extent will it be able to solve a more complexe (en perhaps non-flat) system.

Tools used for the simulations

The chosen programming language for this project is Python[7] as it is a free open source interpreter with good readability and is the go-to language for machine learning since most major ML frameworks are Python-based. The OpenAI Gym toolkit will be used to create the necessary environments and agents, as it includes a complete RL library and visualisation, is open source and is widely used.

"OpenAI Gym is a toolkit for reinforcement learning research. It includes a growing collection of benchmark problems that expose a common interface, and a website where people can share their results and compare the performance of algorithms." (Brockman et al., 2016).

Many of the environments available in Gym are "Classic control" environments[8] such as the CartPole problem (which strongly resembles an inverted crane), or AcroBot (a double pendulum). Some of these systems are non-differentially flat, and developers have managed to develop RL models that find optimal policies so the trained agents are able to solve motion planning problems.

In the next part, with the case of the crane, we will create a simple environment (no friction, no disturbances, no maximum tension in the cables, etc.). Progressively, we will add constraints and modify the geometry and dynamics of our system in order to increase it's defect so it loses its flatness property.

Model used

In order to have a good comparison between the three environments Crane-v0, Crane-V1 and Crane-v2, the same agent and the same model was used.

As discussed in the previous part, a Deep Q-Network learning algorithm was implemented. Considering neural network architecture recommendations seen in the previous part, and through trial and error, a neural network of **two linear hidden layers of 64 Nodes** each, with the states as an input mapped to an output layer of the action size is used for training.

[7] (n.d.). Python.org. Retrieved April 30, 2021, from https://www.python.org/
[8] (n.d.). openai's gym's envs - OpenAI Gym. Retrieved April 30, 2021, from https://gym.openai.com/envs/

The objective is to create a simple benchmark model, and study how it performs, so it will only leave place for improvement. Another reason to not use a complexe Neural Network was computational time. A trade off between performance and computational time is important to take into account, since in practice, a AI-based model will never be used if needed to be trained for a few days for each new system geometry.

The **Adam Optimizer** was used to perform the stochastic gradient descent in order to minimize the weight. The Adam Optimizer is not used to its full potential since we fixed the learning rate of our model to a constant parameter.

The **discount factor** $\gamma = 0.995$ chosen is very high because we want to maximise the long term spars reward of solving the environment's control law and reaching and stabilising the target.

The chosen **learning rate** of $\alpha = 0.0005$was chosen at the bottom of the conventional learning rate intervals because for our complexe systems, we wanted to avoid underfitting the Q values in case of an isolated solved episode. Learning rates of $\alpha = 0.001$, $\alpha = 0.01$ and $\alpha = 0.1$was tried but resulted in a very poor and under-fitted learned model.

The Network is updated every 4 time-steps in order to reduce computational efforts but still take into account changes in state - action value functions.

We chose a high **exploration rate** decay ϵ_{decay} because we wanted a large amount of exploration in the first few hundred episodes in order to explore as much as possible of the state space. With an epsilon decay rate of :

$$\epsilon_{decay} = 0.997$$

The value of the exploration rate at time step $t + 1$ is

$$\epsilon_{t+1} = \epsilon_0 \cdot (\epsilon_{decay})^{t+1}$$

Which means that at episode 800, exploration rate will be $\epsilon \approx 1\%$ and the agent will almost always be in the exploitation phase. The minimum exploration rate during the training is set to $\epsilon_{end} = 0.01$ and the starting value is set to $\epsilon_{end} = 1$ (100% exploration).

The **max time step** is set to 1500 because by experience, when the environment was solved, it was always before timestep 1000. Increasing the max time step was not necessary since it will only increase the negative total reward obtained during a non solved episode and thus increase the interval of reward the learning algorithm encounters, distancing the model training from convergence.

The other parameters were chosen by trial and error, by comparing the training results between them.

γ	τ	α	Update every	ϵ_{start}	ϵ_{end}	ϵ_{decay}	Max time step	Buffer Size	Batch Size
Discount factor	Update of target paramet-ers	Learning Rate	Network Update	Explorati on	Exploitati on	Epsilon greedy policy	Max time steps	Replay buffer size	Minibatch size
0.995	1e-3	5e-4	4	1	0.01	0.997	1500	1e5	64

Table 2: Agent and Model parameters for the environment training

The learning algorithm was implemented by adapting the DQN model from from Mnih et al., (2015), implemented in python by Henri Chan[9] (MIT License Copyright (c) 2018 Henry Chan), and adapted and tuned for our environment and rewards.

[9] "kinwo/deeprl-navigation: Deep Reinforcement Learning ... - GitHub." 1 Sept. 2018, https://github.com/kinwo/deeprl-navigation. Accessed 8 Jun. 2021.

The Cart Pendulum (crane-v0)

In order to get knowledge on how to create a RL environment and agent using the Gym toolkit, a very elementary crane system will be created, based on the inverted pendulum's dynamics equations (Sudarsandhari Shibani et al., 2011), modified to be more easily programmed (Florian, 2005). A scheme of the inverted-pendulum system with some of the variables used for the environment is shown in Figure 4.

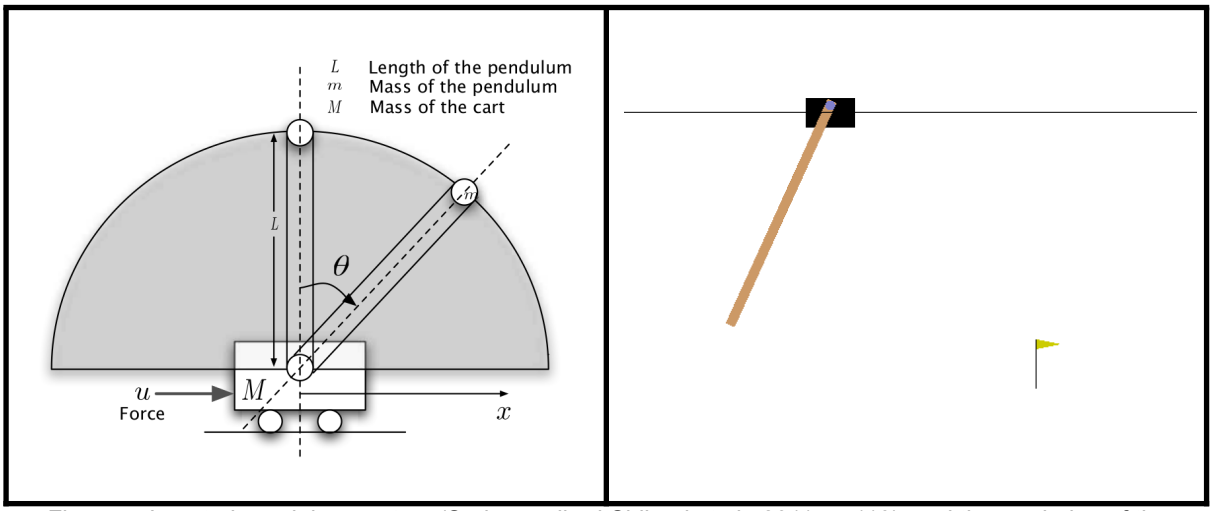


Figure 4: Inverted-pendulum system (Sudarsandhari Shibani et al., 2011, p. 110), and the rendering of the crane-v0 environment.

The frictionless motion equations of the cart pendulum are the following (Florian, 2005), with F being the input force :

$$\ddot{\theta} = \frac{g\sin\theta + \cos\theta\left(\frac{-F - m_p\, l\, \dot{\theta}^2 \sin\theta}{m_c + m_p}\right)}{l\left(\frac{4}{3} - \frac{m_p \cos^2\theta}{m_c + m_p}\right)}$$

$$\ddot{x} = \frac{F + m_p\, l\,(\dot{\theta}^2 \sin\theta - \ddot{\theta}\cos\theta)}{m_c + m_p}$$

Environnement's design

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum starts downward, and the goal is to stabilize it at another x-location of the cart (flag). The time step is $\tau = 0.2\ s$, and the Euler Method is used for the next state values.

The design and specifications of our custom environment are listed in the tables below :

State	x	x_dot	theta	theta_dot
Min	-∞	-∞	∞	-∞
Max	∞	∞	∞	∞

Table 3: Observation space for Crane-V0.

Action	0	1	2
Force F	-10 Newton	0	10 Newton

Table 4: Action space for Crane-V0.

Starting state and target	x	x_dot	theta_dot	theta_dot	Target position
Min	-2	-0.05	π - 0.05	-0.05	1
Max	-1	0.05	π + 0.05	0.05	1

Table 5: Reset state and target position for Crane-V0.

Episode Solved	x	x_dot	theta	theta_dot
Min	Target - 0.05	- 0.05	π - 0.03	- 0.03
Max	Target + 0.05	+ 0.05	π + 0.03	+ 0.03

Table 6: Episode solved conditions for Crane-V0.

The objective of the agent is to stabilize the crane to a chosen x-position, such as if it had to pick up a load located at the flag position. A visualization of the working environment is shown in Figure 4

Rewards

Finding the optimal rewards for the system to converge to the optimal solution was the most difficult part of the environment definition process. A trial-and-error based iteration was performed, based on the theory of reward seen in Sutton & barto (2018). In order to help the agent converge to the optimal solution, smaller intermittent rewards will be added in order to guide the crane to the crane to the target.

- A reward of R = -1 is sent every time step, in order for the agent to find the optimal solution as fast as possible.
    - As the agent did not find the target "by luck" that often, the agent behaved by going to an infinite position. By doing this, the environment broke as the

simulator could not accept values above 3.4028237×10^{38} (Float 32 max values). The episode was considered "solved" before the max time-step.

- A very large reward of R = 100 000 is added when the environment is stabilised at the goal (flag) position.
    - Since the agent did not manage to find the target "by luck", it sometimes never got that reward during training.
- A reward on the x position was added in order to penalise the distance of the cart to the target, and reward a close distance. In order to do this, a reward of $-\ ln(|Target\ -\ x|) + 1$ was awarded.
    - The agent now found the correct x position, but decided to oscillate at the Target position, thus not stabilising the pole.
- At a very close target position, a reward on the angular velocity was added : $-\ ln\left(\left|theta_{dot}\right|\right) \cdot\ 10\ +\ 1$. The +1 term is here to not penalise the agent when the correct x position is reached. The multiplier was added based on trial-and-error testing
    - Here, the agent managed to oscillate around the target position when theta_dot was zero.
- The sinus of the theta angle was added in the logarithm in order to be sure that the swinging rewards would actually be positive when the pole is stabilized.

The final rewards are listed in Table 7 :

Solved episode	$x \in R$	$Target\ -\ x \in [-\ 0.2,\ 0.2]$
100 000	$-\ ln(\|Target\ -\ x\|) + 1$	$-\ ln\left(\left\|theta_{dot}\right\| +\ sin(theta)\right) \cdot\ 10\ +\ 1$

Table 7: Rewards for Crane-v0

Once the agent knows at which state the high sparse reward is given, it determines an optimal policy (control law) to reach it from any initial state.

Results and Analysis

After approximately 28 minutes of training, an optimal policy is determined by the agent using the Deep Q-Network algorithm.

Figure 6 represents the score per episode of training. As you can see, around episode 1000, the model converges to an optimal solution. The training is considered complete if the average of the last 100 episodes of the training process is above the "solved episode" reward.

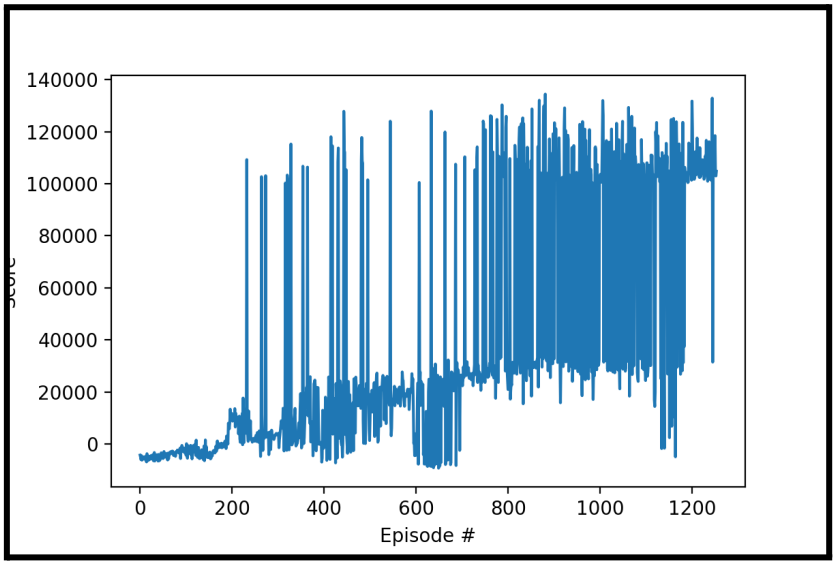


Figure 6: Score per episode during training

As you can see in the Appendix, the training has been done in 1705.75 seconds, or 28 minutes, using a laptop CPU (AMD Ryzen 4800). We see in Figure 6 that the model converges continuously to the optimal control system (for what the agent knows).

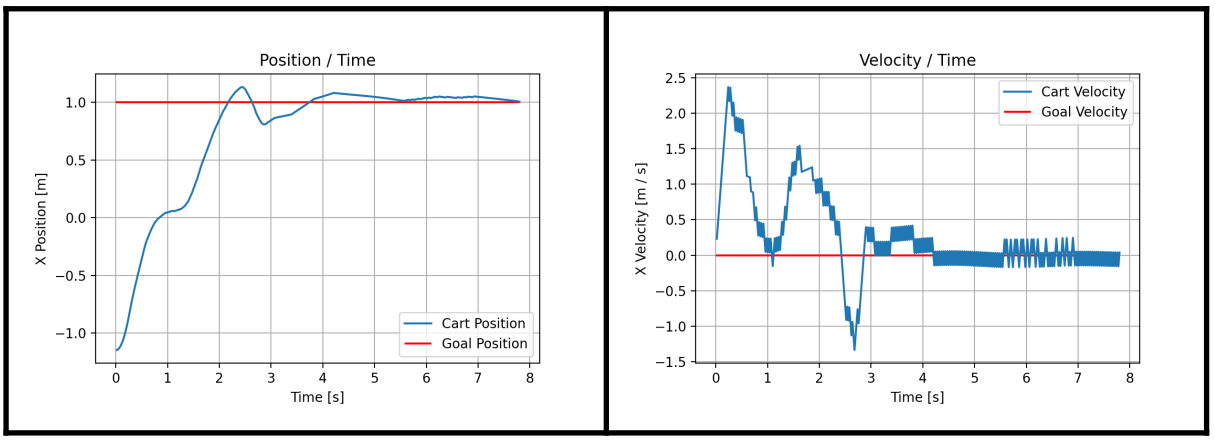
The plots of the states of our environment for a random initial condition is given in Figure 7.

The initial states for the following plots are :

x	x_dot	theta	theta_dot
-1.146	0.230	3.119	0.140

Table 8: Initial conditions for simulation

The environment is solved in 390 time steps, which represents 7.8 seconds.



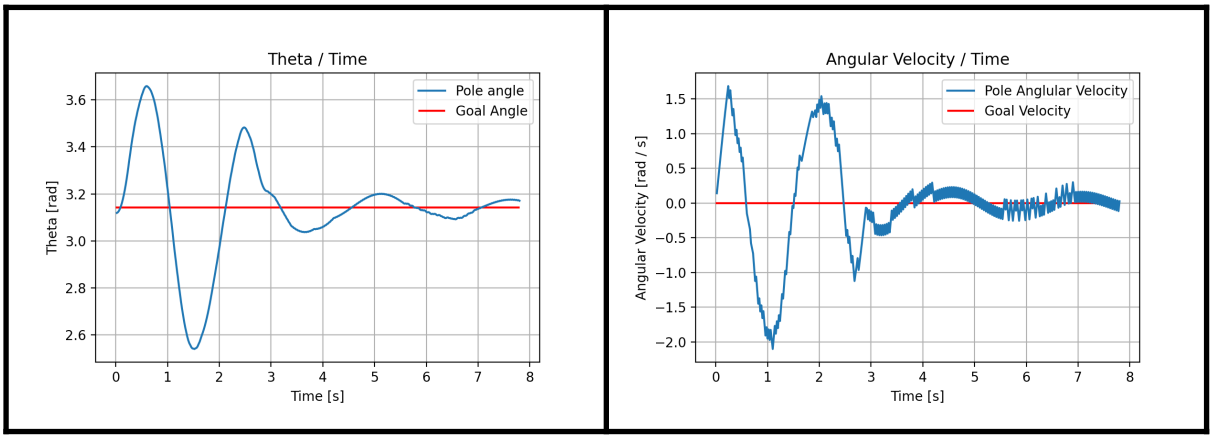


Figure 7: States against time plots for the solved Crane-v0 environment.

By running the simulation with the saved weights more than 100 times, the episode is solved 100% of the time, with an average time of 7.8 seconds, for initial random conditions stated in Table 5.

To apply this model to real life experiments, we can notice that the Target position is always set to 1m. Since the initial conditions are randomly selected, the trajectory can be computed for any starting point to any Target point.

One limitation is that we need to know the position of the Target during the definition of the environment, or else, the reward attribution will be based on no information. In order to solve this problem and be able to train a model without knowing the Target position, an idea would be to use a sensor on the cart to determine its distance from Target, and add this information to the state variables. With this improvement, the model could be trained with random initial conditions and random Target position.

What could be done is to stress our model by changing its initial conditions with initial conditions that are out of the bounds of Table 5, in order to know to what extent the system is robust to starting conditions he was not trained on. A good stressing could be :

- High initial velocity of the cart
- High initial angular velocity
- Starting position on the right side of the Target
- Initial angle on the top of the cart (theta = 0)

The Flat Crane (crane-v1)

The next version of the Crane (crane-v1), which is a realistic differentially flat frictionless crane, with a mass point *m* attached to a rolling up and down rope (Fliess et al. 1995) is presented below.

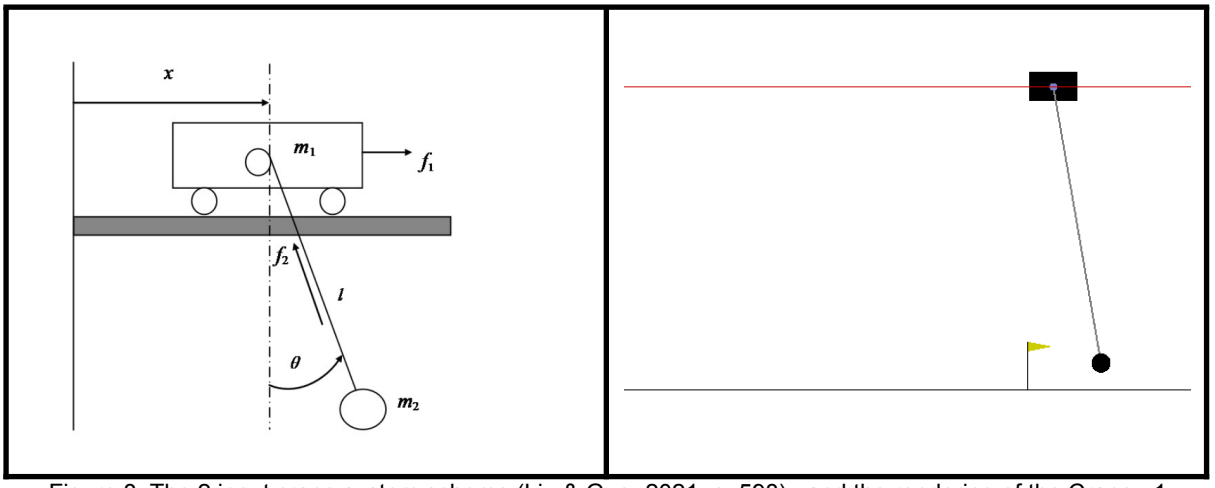


Figure 8: The 2 input crane system scheme (Liu & Guo, 2021; p. 598) , and the rendering of the Crane-v1 environment.

The frictionless equations of motions can be found in Liu & Guo, (2021). With f_1 being the force acting on the cart and f_2 the hoisting and lowering force of the rope. The rope is considered massless and rigid. The mass is considered being a masspoint.

$$\ddot{x} = (f_1 - f_2 \sin\theta)/m_1$$
$$\ddot{l} = f_2/m_2 + \dot{\theta}^2 l + g\cos\theta - (f_1 - f_2\sin\theta)\sin\theta/m_1$$
$$\ddot{\theta} = -((f_1 - f_2\sin\theta)\cos\theta/m_1 + 2\dot{l}\dot{\theta} + g\sin\theta))/l$$

Environment Design

A point mass is attached to an inextensible rope attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum starts downward, and the goal is to stabilize it at another x-location of the cart (Target), with the correct rope length. The time step is $\tau = 0.2\,s$, and the Euler Method is used for the next state values.

The design and specifications of our custom environment are listed in the tables below :

State	x	x_dot	theta	theta_dot	l	l_dot
Min	-∞	-∞	∞	-∞	0.1	-∞
Max	∞	∞	∞	∞	∞	∞

Table 9: Observation space for Crane-V1.

f_2 \ f_1	-10 Newton	0	10 Newton
-10 Newton	0	1	2
0	3	4	6
10 Newton	6	7	8

Table 10: Action space for Crane-V1.

Starting state and target	x	x_dot	theta	theta_dot	l	l_dot	Target position
Min	-2	-0.05	- 0.05	-0.05	1	-0.05	(1,0)
Max	-1	0.05	+ 0.05	0.05	2	0.05	(1,0

Table 11: Reset state and target position for Crane-V1.

Episode Solved	x	x_dot	theta	theta_dot	l	l_dot
Min	Target - 0.15	- 0.15	- 0.15	- 0.15	Target - 0.15	- 0.15
Max	Target + 0.15	+ 0.1(	0.15	+ 0.15	Target + 0.15	0.15

Table 12: Episode solved conditions for Crane-V1.

Rewards

For finding the rewards of the Crane V1 environment, we based ourselves on the method used for the Crane V0 environment, in the intent to have comparable results.

- A reward of R = -1 is sent every time step, in order for the agent to find the optimal solution as fast as possible.
    - As the agent did not find the target "by luck" that often, the agent behaved by going to an infinite position, infinite length, or infinite angle. By doing this, the environment broke as the simulator could not accept values above 3.4028237×10^{38} (Float 32 max values). The episode was considered "solved" before the max time-step.
- A very large reward of R = 10 000 000 is added when the environment is stabilised at the goal (flag) position.
    - Since the agent did not manage to find the target "by luck", never got that reward during training.

- A reward on the x position was added in order to penalise the distance of the cart to the target, and reward a close distance. In order to do this, a reward of $-\ ln(|Target\ -\ x|) \cdot 2 + 1$ was awarded. The +1 term is here to not penalise the agent when the correct x position is reached. The multiplier was added based on trial-and-error testing.
    - The agent now found the correct x position, but decided to oscillate at the Target position, thus not stabilising the pole. The length of the rope was never close to target length.
- At a very close target position, a reward on the angular velocity was added : $-\ ln\left(\left|theta_{dot}\right| + 3 \cdot sin(theta)\right) \cdot 15 + 5.$
- A reward on the length based on the same logarithmic function : $-\ ln\left(|l| -\ target_{length}\right)) \cdot 8 + 5.$
    - The + 5 terms were added to not penalise the agent when the correct x position is reached.
- The multipliers were set by trial-and-error:
    - If the length reward was more important, the agent would get the correct length and oscillate around the target position, neglecting the negative angle reward.
    - Inversely, if the angle reward was more important, the agent would find that setting the length the highest will result in easier stabilisation of the swing angle.
- An additional negative reward concerning l_dot was added to guide the agent to the correct length. $\pm \frac{l_{dot}}{6}$ depending if the rope should be hoisted or lowered.

The final rewards are listed in Table 13 :

Goal Achieved	$x \in R$	$Target\ -\ x \in [-\ 0.2,\ 0.2]$	$l < 3$ or $l > 3$
10 000 000	$-\ ln(\|Target\ -\ x\|) \cdot 2 + 1$	$-\ ln\left(\left\|theta_{dot}\right\| + 3 \cdot sin(theta)\right) \cdot 15 + 5$	$\pm \frac{l_{dot}}{6}$
		$-\ ln\left(\|l\| -\ target_{length}\right)) \cdot 8 + 5$	

Table 13: Rewards for Crane-v1

Results and Analysis

After approximately 30 minutes of training, a suboptimal policy is determined by the agent using the Deep Q-Network algorithm. The environment is not solved but the best policy was found near 600 training episodes, before diverging. This is certainly due to an

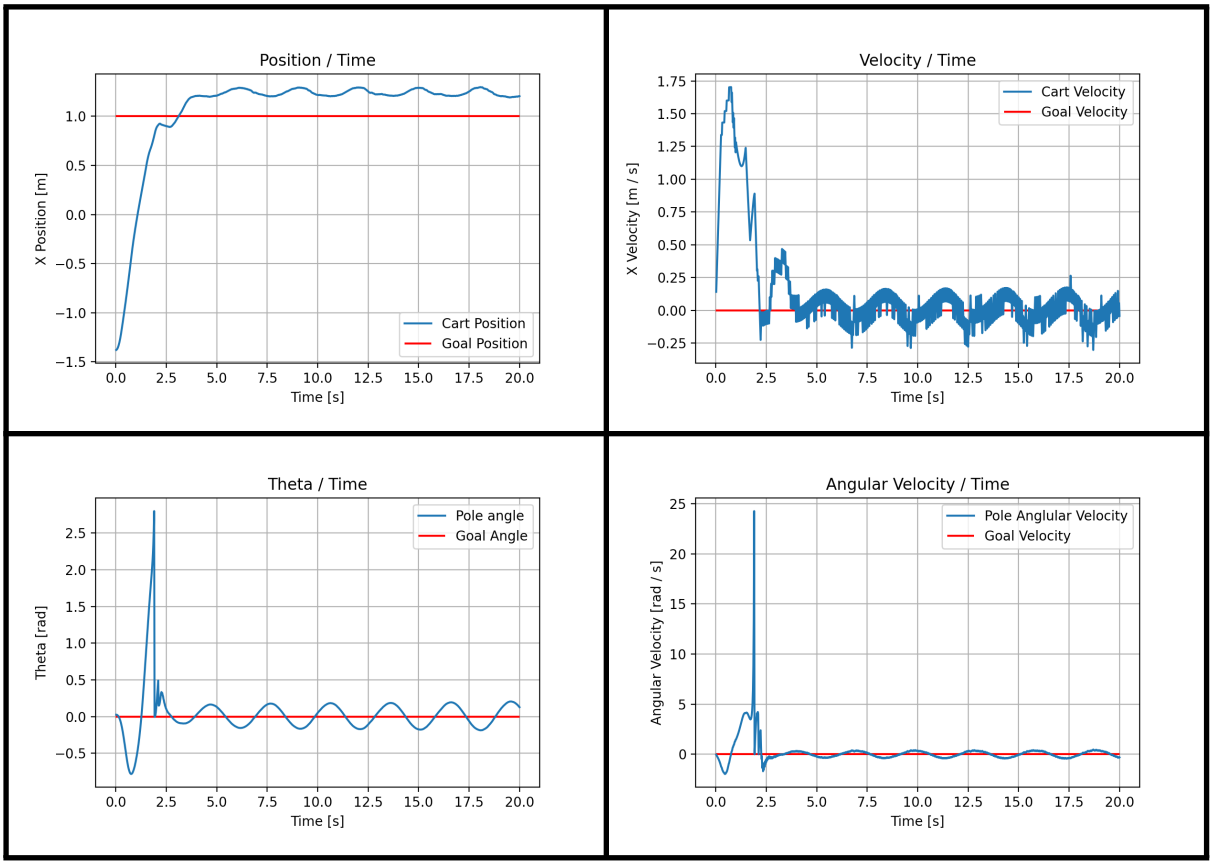
overestimation of the Q value in early stages, which is bad for our environment since its exploration is very large as the input space is infinite.[10]

The initial states for the following plots are :

x	x_dot	theta	theta_dot	l	l_dot
-1.437	0.183	0.013	-0.008	1.486	-0.053

Table 14: Initial conditions for simulation

The plots of the states of our environment for a random initial condition is given in Figure 8.



[10] "kinwo/deeprl-navigation: Deep Reinforcement Learning ... - GitHub." 1 Sept. 2018, https://github.com/kinwo/deeprl-navigation. Accessed 8 Jun. 2021.

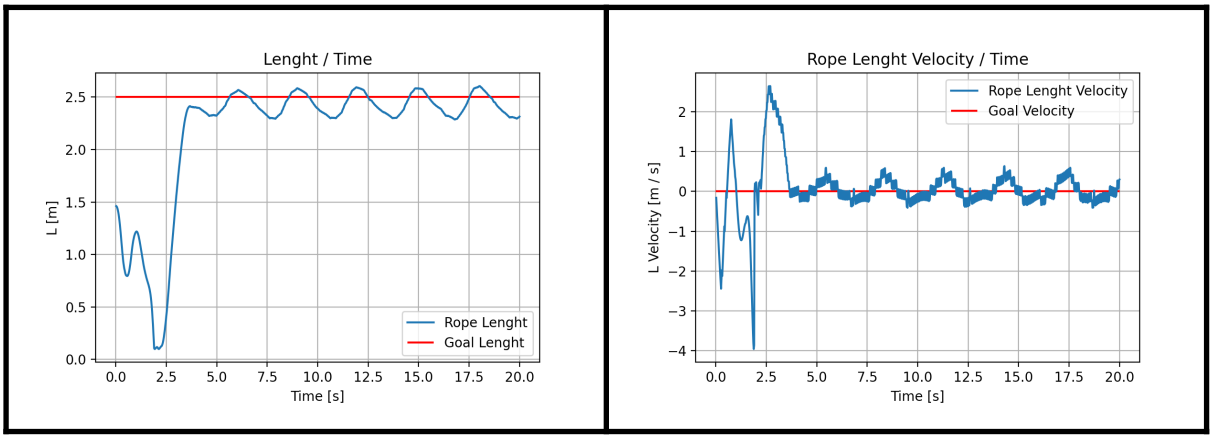
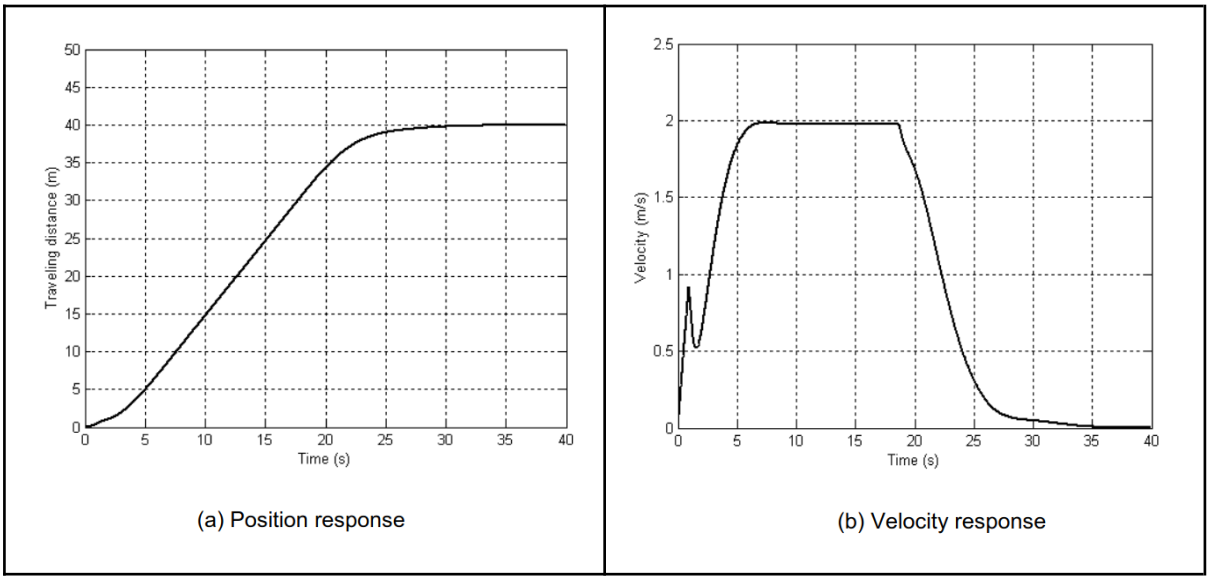


Figure 9: States against time plots for the solved Crane-v1 environment.

As previously said, the learning algorithm did not manage to solve the environment. Only a suboptimal solution was found. The system's length and angle oscillate after reaching a "close enough" x position.

If we inspect the length against the time plot, we see that the agent hoists the rope to the minimum allowed position, while moving the cart to the desired x Target position, and then lowers it down. It is at this moment that the system oscillates infinitely.

If we compare our trajectories with the two sliding mode controllers tracking a predefined trajectory found in Liu & Guo (2012), we can see some similarities.



(a) Position response

(b) Velocity response

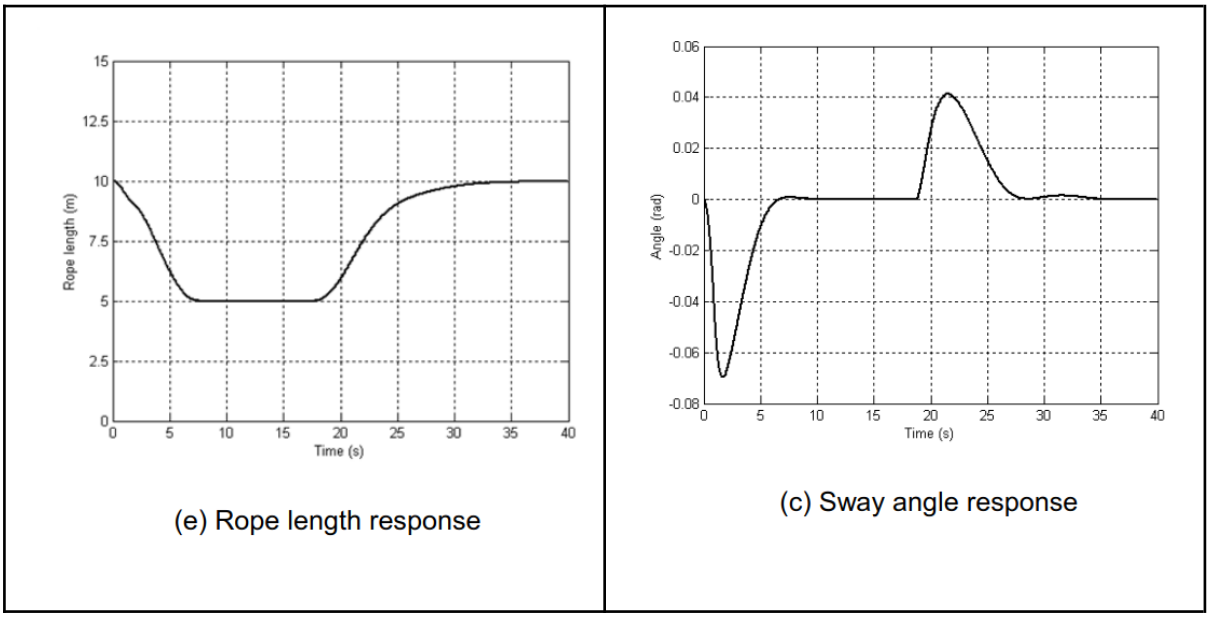


Figure 10: States against time plots for a trajectory tracking controller (Liu & Guo, 2012)

In Liu & Guo (2012)'s article, the others design two controllers. One to manage the hosting and lowering force, and the other one to manage the cart force, with a self-tuning fuzzy inference algorithm to adjust the relationship between the two controllers.

The Crane-v1 agent seems to have difficulties finding the right control law since it is controlling the two inputs at the same time. A more complex Neural Network might improve the model.

Another idea would be to follow the path of Liu & Guo (2012), and train one agent for the cart position and swing angle (such as in the Crane-v0 environment), and train a second agent to control the cable length.

We know from the Crane-v0 case study that if the length of the cable is maintained constant, the agent is able to stabilize the crane to the desired position. Another idea could be to use the agent of Crane-v0 for the cart force, and train a second agent to compensate for the length acceleration caused by the equations of motions.

The two previous suggestions might solve the problem but they have the drawback of interfering with the agent's optimal trajectory, in the case where it would be the only controller.

The Double Pendulum Crane (Crane-v2)

The next version of the Crane (crane-v2), is a non-flat crane. The crane transports a load of variable length. Unlike Crane-v1, the rope length is fixed, so there's only one input force on the cart position. It can be modeled as a double pendulum on a cart with the first pendulum's mass being infinitely small. The centers of mass of the links are in their centers, the pendulums are considered solid rods. The rope (first pendulum) is considered massless and rigid.

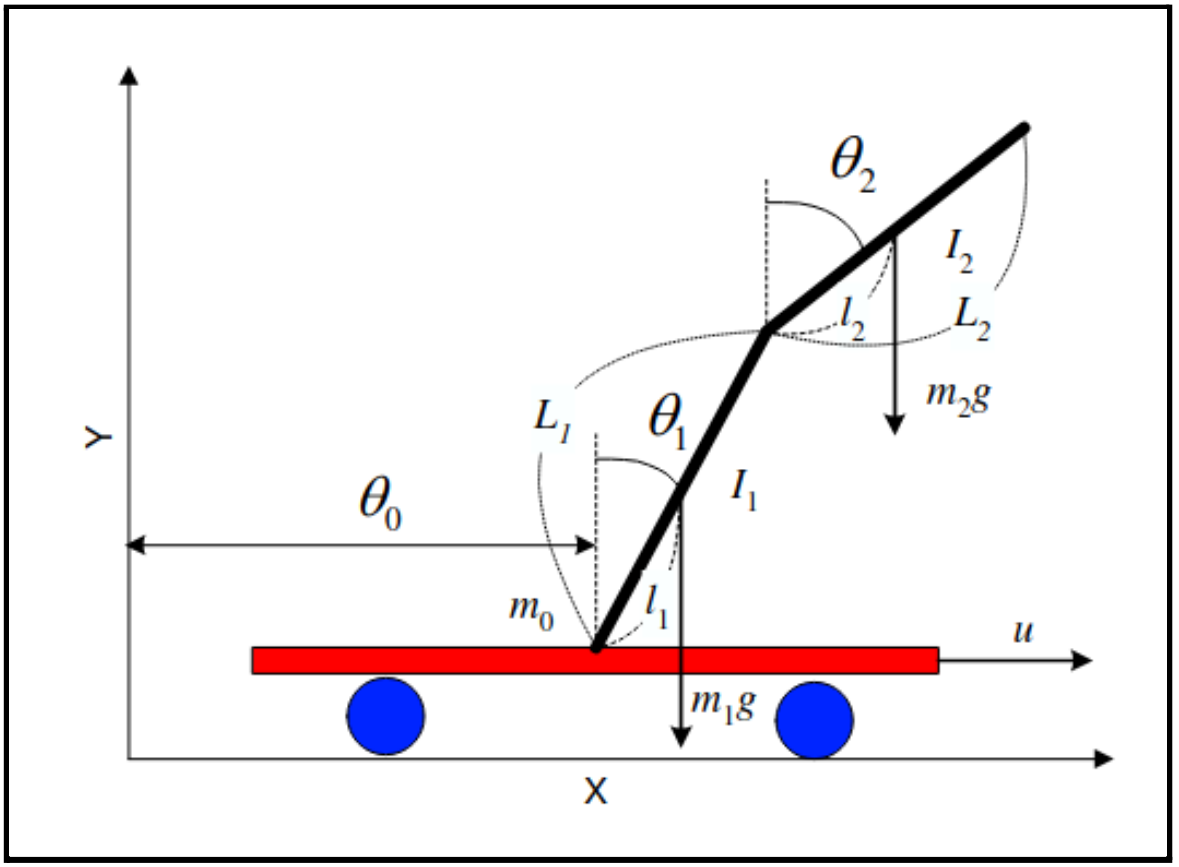


Figure 11 : The inverted double pendulum (Bogdavov, 2016)

The frictionless and non externally disturbed equations of motions of the double inverted pendulum are shown below (Bogdanov, 2016):

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

Where the matrix D, C, G, H are :

$$\mathbf{D}(\theta) = \begin{pmatrix} d_1 & d_2\cos\theta_1 & d_3\cos\theta_2 \\ d_2\cos\theta_1 & d_4 & d_5\cos(\theta_1-\theta_2) \\ d_3\cos\theta_2 & d_5\cos(\theta_1-\theta_2) & d_6 \end{pmatrix}$$

$$\mathbf{C}(\theta,\dot{\theta}) = \begin{pmatrix} 0 & -d_2\sin(\theta_1)\dot{\theta}_1 & -d_3\sin(\theta_2)\dot{\theta}_2 \\ 0 & 0 & d_5\sin(\theta_1-\theta_2)\dot{\theta}_2 \\ 0 & -d_5\sin(\theta_1-\theta_2)\dot{\theta}_1 & 0 \end{pmatrix}$$

$$\mathbf{G}(\theta) = \begin{pmatrix} 0 \\ -f_1\sin\theta_1 \\ -f_2\sin\theta_2 \end{pmatrix}$$

$$\mathbf{H} = (1\ 0\ 0)^T$$

Following the previous geometrical assumptions, we have :

$$\begin{aligned} d_1 &= m_0 + m_1 + m_2 \\ d_2 &= m_1 l_1 + m_2 L1 = (\frac{1}{2}m_1 + m_2)L_1 \\ d_3 &= m_2 l_2 = \frac{1}{2}m_2 L_2 \\ d_4 &= m_1 l_1^2 + m_2 L_1^2 + I_1 = (\frac{1}{3}m_1 + m_2)L_1^2 \\ d_5 &= m_2 L_1 l_2 = \frac{1}{2}m_2 L_1 L_2 \\ d_6 &= m_2 l_2^2 + I_2 = \frac{1}{3}m_2 L_2^2 \\ f_1 &= (m_1 l_1 + m_2 L_1)g = (\frac{1}{2}m_1 + m_2)L_1 g \\ f_2 &= m_2 l_2 g = \frac{1}{2}m_2 L_2 g \end{aligned}$$

Since the matrix $D(\theta)$ is non-singular, we can express the acceleration in function of the state variable, and via the Euler method, compute the next states.

Environment design

A vertical load is attached to a rope which is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum starts downward, and the goal is to stabilize it at another x-location of the cart (flag). The time step is $\tau = 0.2\ s$, and the Euler Method is used for the next state values.

The design and specifications of our custom environment are listed in the tables below :

State	x	x_dot	theta1	theta1_dot	theta2	theta2_dot	L2
Min	-∞	-∞	∞	-∞	∞	-∞	0
Max	∞	∞	∞	∞	∞	∞	∞

Table 15: Observation space for Crane-V2.

Action	0	1	2	3	4	5	6	7	8	9	10
Force	-5N	-4N	-3N	-2N	-1N	0N	1N	2N	3N	4N	5N

Table 16: Action space for Crane-V2.

Starting state and target	x	x_dot	theta1	theta1_dot	theta2	theta2_dot	L2
Min	-2	-0.05	π - 0.05	-0.05	π - 0.05	-0.05	1
Max	-1	0.05	π + 0.05	0.05	π + 0.05	0.05	1

Table 17: Reset state and target position for Crane-V2.

Episode Solved	x	x_dot	theta1	theta1_dot	theta1	theta1_dot
Min	Target - 0.1	- 0.1	π - 0.1	- 0.1	π - 0.1	- 0.1
Max	Target + 0.1	+ 0.1	π + 0.1	+ 0.1	π + 0.1	+ 0.1

Table 18: Episode solved conditions for Crane-V2.

The episode is considered solved if all angles are stabilized and the position of the cart stable at the target position.

Rewards

For finding the rewards of the Crane V1 environment, we based ourselves on the method used for the Crane V0 environment, in the intent to have comparable results. Since no solution was found during training, the initial rewards were kept. The rewards are listed in Table 19. A reward for the angular position is added in order to try to maintain the system in a non chaotic state.

Goal Achieved	$x \in R$	$Target - x \in [-0.2, 0.2]$

100 000	$-\ ln(\lvert Target\ -\ x\rvert) \cdot 3\ +\ 1$	$-\ ln\left(\left\lvert theta_{1_{dot}}\right\rvert +\ sin(theta_1)\right) \cdot 10\ +\ 1$
	$-\ \lvert theta_1\rvert - \lvert theta_2\rvert$	$-\ ln\left(\left\lvert theta_{2_{dot}}\right\rvert +\ sin(theta_2)\right) \cdot 10\ +\ 1$

Table 19: Rewards for Crane-v2

Results and Analysis

After training the learning algorithm for multiple hours, no episode was solved. Figure 12 shows that no convergence has occurs using our model

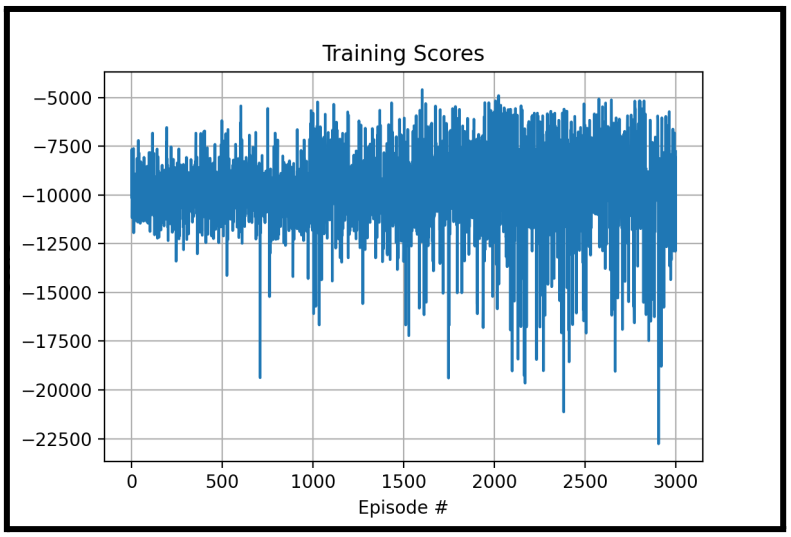


Figure 12: Score per episode for the Crane-v2 environment.

The state plots below show that the system is acting in a chaotic manner, diverging completely from the optimal solution. Changing the length of the Load did not result in an amelioration of the behavior of the system.

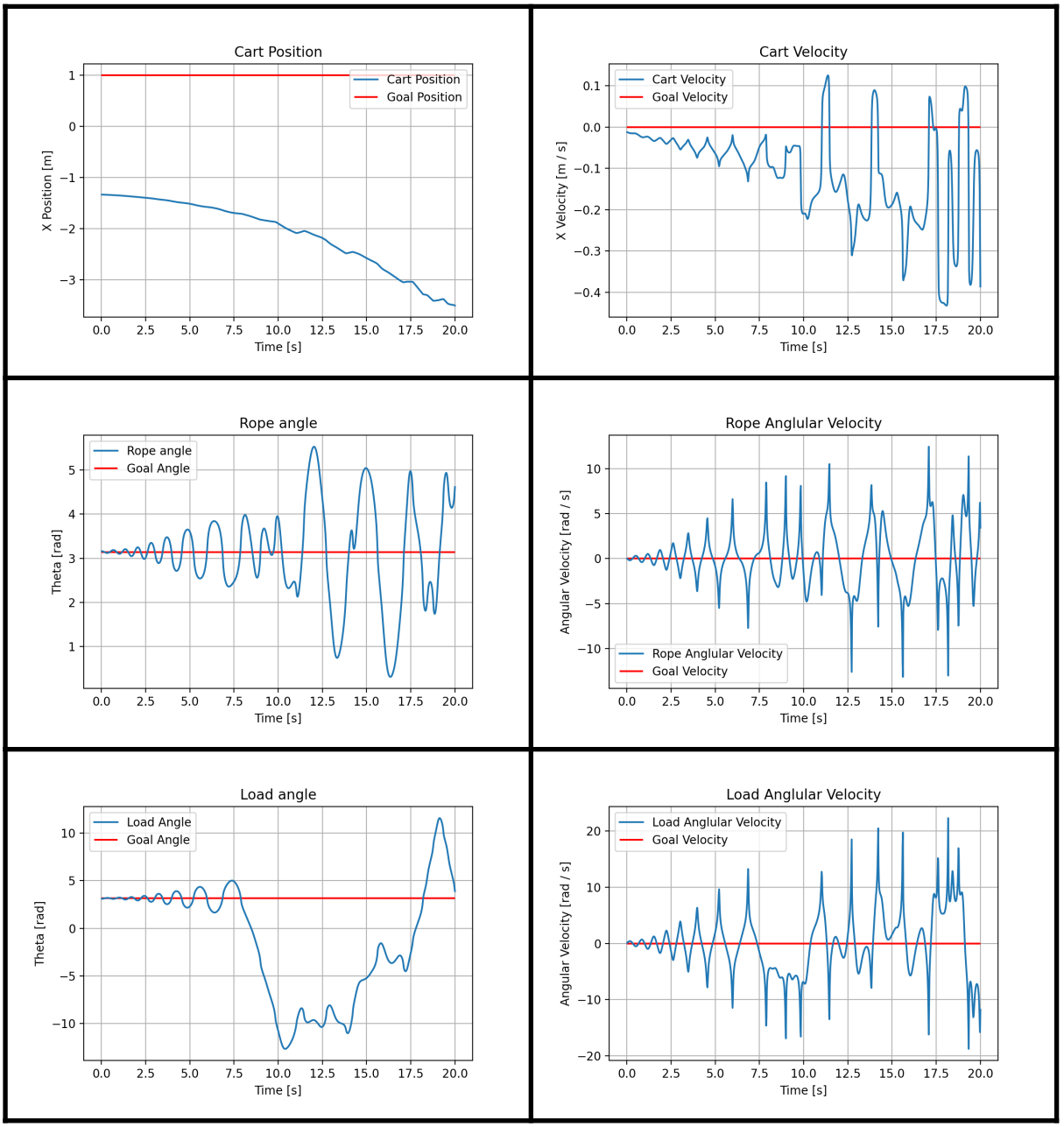


Figure 13: States against time plots for the non solved Crane-v2 environment.

The learning algorithm did not manage to solve the episode once and the system is behaving in a chaotic manner.

By looking at cases where such systems are controlled, we can get some insights.

Chen et al. (2019) manages to control a double pendulum crane using a pure neural network tracking controller that requires the linearization of the outputs around the equilibrium. They use a reference trajectory that the cart x position should follow, and the neural network is here to minimize the tracking error of the controller.

By performing a linearization and using small angles approximations, the non flat double pendulum cart system becomes flat, thus, a trajectory planning is known to be possible.

We believe that one of the reasons that the learning algorithm did not converge for the third system (Crane-v1) is that the agent never received the maximum reward. A larger training process might result in achieving this task.

The dynamics of the double pendulum cart are so complexe that adding more complexity to the neural network used will probably result in a better control function approximation.

Discussion and Conclusion

During this master project, we wanted to know to what extent the knowledge of the flatness property of the unperturbed system helps in solving the variational problem of the perturbed case. To answer this question, we asked ourselves if modern tools such as deep reinforcement learning would be able to solve the trajectory planning of complex systems, and more precisely non-differentially flat systems.

In order to do this, an overview on reinforcement learning was done, in order to deeply understand the concepts behind this framework and determine if it was able to give us insight on the research question. The study of the resolution of the optimal control problem of three different systems of 2D Cranes was done using a pure Deep-RL model.

A very simple and basic learning algorithm using a Deep Q-Network agent trained via a Neural Network was created in order to try to solve the optimal control problems of our systems. The choice of a simple algorithm was made in order to present a proof of concept that these new tools could / or could not solve complex systems. The learning algorithm used is the same for the three systems studied.

The first system, a simple flat crane, was successfully controlled.

The second Crane, a more realistic crane with 2 inputs (cable force and cart force), was almost solved with the same algorithm. A longer training period, the use of more complex algorithms, might have helped in finding the optimal solution of the control problem. The second system, which is a perturbation of the first one, is much more complex in its dynamic, and gives us hope that the reinforcement learning framework might be able to solve non-differentially flat and complexe systems.

The last example was a non-flat crane with one input. The system is similar to a double pendulum on a cart and is behaving with a chaotic behavior. Our Learning algorithm did not manage to solve the control problem for this system.

Thus, we can not conclude that the knowledge of the properties of the flat system helps in solving the non flat one, from a reinforcement learning agent point of view.

Calls for future research

By looking at the learning curves of the first systems (Crane-v0), it seems that the agent starts to make the right decisions when the goal is reached multiple times. The learning algorithm behaves in a way that the large sparse reward (solving the environment) needs to be rewarded during multiple episodes in order to converge to the optimal solution. The environment's rewards were modified in order to reduce the computational effort of getting the system through each possible state.

A lot of tools are available to solve these issues, such as imitation learning (Sutton & Barto, 2018). Imitation learning is a technique used to show the agent a correct trajectory to take. The agent will observe the path and possibly improve it. This method will result in less

computational effort but with a hope that the agent will use this known information to understand the goal, and possibly improve the trajectory.

Another idea would be to stabilize the system temporarily with a classic controller and add the RL agent on the setpoint. This avoids the learning algorithm to visit every state of the observation space.

Furthermore, many reinforcement learning algorithms have not been tested on these systems, and might improve the results. A Deep Deterministic Policy Gradients (DDPG) agent might to the task, as it is an Actor-Critic model, which are known to be good for continuous action space and control system theory[11].

The double pendulum non-flat crane environment is ready to use and can be requested from the author in order to train a more complex learning algorithm, and hopefully solve the optimal control problem.

[11] "kinwo/deeprl-continuous-control: Learning Continuous ... - GitHub." https://github.com/kinwo/deeprl-continuous-control. Accessed 11 Jun. 2021.

Bibliography

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. *ArXiv Preprint ArXiv:1606.01540*.

Arulkumaran, K., Deisenroth, M., Brundage, M., & Bharath, A. (2017). Deep Reinforcement Learning: A Brief Survey. *IEEE Signal Processing Magazine*, *34*(6), 26-38. https://doi.org/10.1109/msp.2017.2743240

Bosagh Zadeh, R., & Ramsundar, B. (2018). *TensorFlow for Deep Learning: From Linear Regression to Reinforcement Learning*. O'Reilly.

Buduma, N., & Locascio, N. (2017). *Fundamentals of deep learning* (1st ed., pp. 245-276).

Sewak, M., Sahay, S., & Rathore, H. (2020). Value-Approximation based Deep Reinforcement Learning Techniques: An Overview. *2020 IEEE 5Th International Conference On Computing Communication And Automation (ICCCA)*. https://doi.org/10.1109/iccca49541.2020.9250787

Bogdanov, Alexander. (2004). Optimal Control of a Double Inverted Pendulum on a Cart.

Li, F.F., Johnson J., & Yeung S. (2017, May 23). *Lecture 14: Reinforcement Learning* [PowerPoint presentation]. Stanford University, Stanford, CA, United States. http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf

Reinforcement Learning for Control Systems Applications. (n.d.). MathWorks. Retrieved April 29, 2021, from https://ch.mathworks.com/help/reinforcement-learning/ug/reinforcement-learning-for-control-systems-applications.html

Ding, M. & Wiering, M.A. (2018). *Reinforcement Learning For Offshore Crane Set-down Operations* (Master's Thesis, University of Groningen, Groningen, Netherland). Retrieved from https://fse.studenttheses.ub.rug.nl/18927/

Padakandla, S. (2020). A Survey of Reinforcement Learning Algorithms for Dynamically Varying Environments. *ArXiv, abs/2005.10619*.

Heaton, J. (2008). The number of hidden layers. Heaton Research Inc.

Florian, R. (2005). Correct equations for the dynamics of the cart-pole system.

Bejar, E., & Moran, A. (2018). Backing Up Control of a Self-Driving Truck-Trailer Vehicle with Deep Reinforcement Learning and Fuzzy Logic. *2018 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*. doi:10.1109/isspit.2018.8642777

Buccieri, D., Mullhaupt, P., & Bonvin, D. (2005). Spidercrane: Model And Properties Of A Fast Weight Handling Equipment. *IFAC Proceedings Volumes, 38*(1), 568-573. doi:10.3182/20050703-6-cz-1902.00750

Buşoniu, L., Bruin, T. D., Tolić, D., Kober, J., & Palunko, I. (2018). Reinforcement learning for control: Performance, stability, and deep approximators. *Annual Reviews in Control, 46*, 8-28. doi:10.1016/j.arcontrol.2018.09.005

Chen, Q., Cheng, W., Gao, L., & Fottner, J. (2019). A pure neural network controller for double-pendulum crane anti-sway control: Based on Lyapunov stability theory. *Asian Journal of Control, 23*(1), 387-398. doi:10.1002/asjc.2226

Editorial Board. (2018). *Annual Reviews in Control, 46*. doi:10.1016/s1367-5788(18)30180-9

Euler, L. (1792). *Institutiones calculi integralis*.

Fliess, M., Lévine, J., Martin, P., & Rouchon, P. (1995). Flatness and defect of non-linear systems: Introductory theory and examples. *International Journal of Control, 61*(6), 1327-1361. doi:10.1080/00207179508921959

Fliess, M., Lévine, J., Martin, P., & Rouchon, P. (1995). Flatness and defect of non-linear systems: Introductory theory and examples. *International Journal of Control, 61*(6), 1327-1361. doi:10.1080/00207179508921959

Fliess, M., Lévine, J., & Rouchon, P. (1993). Generalized state variable representation for a simplified crane description. *International Journal of Control, 58*(2), 277-283. doi:10.1080/00207179308923002

Grabski, F. (2015). Discrete state space Markov processes. *Semi-Markov Processes: Applications in System Reliability and Maintenance,* 1-17. doi:10.1016/b978-0-12-800518-7.00001-6

Liu, D., & Guo, W. (2012). Tracking Control for an Underactuated Two-Dimensional Overhead Crane. *Journal of Applied Research and Technology, 10*(4). doi:10.22201/icat.16656423.2012.10.4.383

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

Rosolia, U., & Borrelli, F. (2018). Learning Model Predictive Control for Iterative Tasks. A Data-Driven Control Framework. *IEEE Transactions on Automatic Control, 63*(7), 1883-1896. doi:10.1109/tac.2017.2753460

Rouchon, P., Fliess, M., Levine, J., & Martin, P. (n.d.). Flatness, motion planning and trailer systems. *Proceedings of 32nd IEEE Conference on Decision and Control*. doi:10.1109/cdc.1993.325686

Sewak, M., Sahay, S. K., & Rathore, H. (2020). Value-Approximation based Deep Reinforcement Learning Techniques: An Overview. *2020 IEEE 5th International Conference on Computing Communication and Automation (ICCCA)*.

doi:10.1109/iccca49541.2020.9250787

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. The MIT Press.

Willson, S. S., Mullhaupt, P., & Bonvin, D. (2011). A quotient method for designing nonlinear controllers. *IEEE Conference on Decision and Control and European Control Conference*. doi:10.1109/cdc.2011.6160803

Appendix

- James001 : Training agent for Crane-v1
- James002 : Training agent for Crane-v1
- James003 : Training agent for Crane-v2
- CraneEnv.py : CRANE-v0 Environment
- CraneEnv_1.py : CRANE-v1 Environment
- CraneEnv_2.py : CRANE-v2 Environment

James001

June 11, 2021

1 Crane V0 model training and simulations

Adapted from https://github.com/kinwo/deeprl-navigation (MIT License Copyright (c) 2018 Henry Chan)

Start Environement and create DQN Agent

[1]:
```
import gym
import numpy as np

env = gym.make('crane-v0') #Load the environement
```

1.1 Agent

The DQN Agent adapted from https://github.com/kinwo/deeprl-navigation (MIT License Copyright (c) 2018 Henry Chan). The following block is also found in /model_script.py file in the current working directory

[2]:
```
import numpy as np
import random
from collections import namedtuple, deque

#from model_script import QNetwork # UNCOMMENT IF YOU ARE NOT IN A JUPYTER␣
↪NOTEBOOK

import torch
import torch.nn.functional as F
import torch.optim as optim

BUFFER_SIZE = int(1e5)                  # replay buffer size
BATCH_SIZE = 64                         # minibatch size
GAMMA = 0.995            #was 0.99      # discount factor
TAU = 1e-3                              # for soft update of target parameters
LR = 5e-4                               # learning rate
UPDATE_EVERY = 4                        # how often to update the network

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
#device = torch.device('cuda:0')
```

```
class Agent():
    """Interacts with and learns from the environment."""

    def __init__(self, state_size, action_size, seed):
        """Initialize an Agent object.

        Params
        ======
            state_size (int): dimension of each state
            action_size (int): dimension of each action
            seed (int): random seed
        """



        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)

        # Q-Network
        self.qnetwork_local = QNetwork(state_size, action_size, seed).to(device)
        self.qnetwork_target = QNetwork(state_size, action_size, seed).
↪to(device)
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=LR)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)
        # Initialize time step (for updating every UPDATE_EVERY steps)
        self.t_step = 0

    def step(self, state, action, reward, next_state, done):
        # Save experience in replay memory
        self.memory.add(state, action, reward, next_state, done)

        # Learn every UPDATE_EVERY time steps.
        self.t_step = (self.t_step + 1) % UPDATE_EVERY
        if self.t_step == 0:
            # If enough samples are available in memory, get random subset and␣
↪learn
            if len(self.memory) > BATCH_SIZE:
                experiences = self.memory.sample()
                self.learn(experiences, GAMMA)

    def act(self, state, eps=0.):
        """Returns actions for given state as per current policy.

        Params
```

```
        ======
            state (array_like): current state
            eps (float): epsilon, for epsilon-greedy action selection
        """
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        self.qnetwork_local.eval()
        with torch.no_grad():
            action_values = self.qnetwork_local(state)
        self.qnetwork_local.train()

        # Epsilon-greedy action selection
        if random.random() > eps:
            return np.argmax(action_values.cpu().data.numpy())
        else:
            return random.choice(np.arange(self.action_size))

    def learn(self, experiences, gamma):
        """Update value parameters using given batch of experience tuples.

        Params
        ======
            experiences (Tuple[torch.Variable]): tuple of (s, a, r, s', done)␣
↪tuples
            gamma (float): discount factor
        """
        states, actions, rewards, next_states, dones = experiences

        # Get max predicted Q values (for next states) from target model
        Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].
↪unsqueeze(1)
        # Compute Q targets for current states
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

        # Get expected Q values from local model
        Q_expected = self.qnetwork_local(states).gather(1, actions)

        # Compute loss
        loss = F.mse_loss(Q_expected, Q_targets)
        # Minimize the loss
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # ------------------- update target network ------------------- #
        self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)

    def soft_update(self, local_model, target_model, tau):
```

```
        """Soft update model parameters.
        _target = *_local + (1 - )*_target

        Params
        ======
            local_model (PyTorch model): weights will be copied from
            target_model (PyTorch model): weights will be copied to
            tau (float): interpolation parameter
        """
        for target_param, local_param in zip(target_model.parameters(),␣
↪local_model.parameters()):
            target_param.data.copy_(tau*local_param.data + (1.
↪0-tau)*target_param.data)


class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, action_size, buffer_size, batch_size, seed):
        """Initialize a ReplayBuffer object.

        Params
        ======
            action_size (int): dimension of each action
            buffer_size (int): maximum size of buffer
            batch_size (int): size of each training batch
            seed (int): random seed
        """

        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state",␣
↪"action", "reward", "next_state", "done"])
        self.seed = random.seed(seed)

    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        experiences = random.sample(self.memory, k=self.batch_size)

        states = torch.from_numpy(np.vstack([e.state for e in experiences if e␣
↪is not None])).float().to(device)
```

```
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if␣
↪e is not None])).long().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if␣
↪e is not None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e.next_state for e in␣
↪experiences if e is not None])).float().to(device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is␣
↪not None]).astype(np.uint8)).float().to(device)

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)
```

1.2 Deep Q_Network Model

A 2 linear hidden layer of 64 nodes each is created, with relu activation function.

[3]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

class QNetwork(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=64,␣
↪fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
```

```
        return self.fc3(x)
```

1.3 DQN Agent Training

1.3.1 Create DQN Agent

[4]:
```
import torch
import time
from collections import deque

#from agent_script import Agent    # UNCOMMENT IF YOU ARE NOT IN A JUPYTER␣
 ↪NOTEBOOK

import matplotlib.pyplot as plt
%matplotlib inline

state_size=4
action_size=3
seed=0

agent = Agent(state_size=4, action_size=3, seed=0)
```

In order to know when the environement is solved, we compute the moving score (the total rewards per episode) average over the last 100 episodes. If the moving average is over a chosen thershold (target_scores), the model is then saved to 'model_weight_name'.

For the Crane_v0 environement, the target score is 100 000, since it is the reward obtained by the agent when finding the flag.

[5]:
```
model_weight_name = 'checkpoint_precise_2.pth'


def dqn(n_episodes=10000, max_t=2000, eps_start=1.0, eps_end=0.01, eps_decay=0.
 ↪997, target_scores=100000.0):
    """Deep Q-Learning.

    Params
    ======
        n_episodes (int): maximum number of training episodes
        max_t (int): maximum number of timesteps per episode
        eps_start (float): starting value of epsilon, for epsilon-greedy action␣
 ↪selection
        eps_end (float): minimum value of epsilon
        eps_decay (float): multiplicative factor (per episode) for decreasing␣
 ↪epsilon
        target_scores (float): average scores aming to achieve, the agent will␣
 ↪stop training once it reaches this scores
```

```
    """
    start = time.time()                # Start time
    scores = []                        # list containing scores from each episode
    scores_window = deque(maxlen=100)  # last 100 scores
    eps = eps_start                    # initialize epsilon

    for i_episode in range(1, n_episodes+1):
        # Reset env and score at the beginning of episode
        env_info = env.reset()                                  # reset the environment
        state = env.state                                       # get the current state
        score = 0                                               # initialize the score

        for t in range(max_t):
            action = agent.act(state, eps)
            env_info = env.step(action)                         # send the action to the environment
            next_state = env_info[0]                            # get the next state
            reward = env_info[1]                                # get the reward
            done = env_info[2]                                  # see if episode has finished

            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward
            if done:
                print("    Episode finished after {} timesteps".format(t+1))
                #print("final state is :", state)
                #print("Reward is : ", score)
                break

        scores_window.append(score)       # save most recent score
        scores.append(score)              # save most recent score
        eps = max(eps_end, eps_decay*eps) # decrease epsilon

        print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)), end="")

        if i_episode % 100 == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))

        if np.mean(scores_window)>=target_scores:
```

```
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.
↪2f}'.format(i_episode, np.mean(scores_window)))
            torch.save(agent.qnetwork_local.state_dict(), model_weight_name)
            break

    time_elapsed = time.time() - start
    print("Time Elapse: {:.2f}".format(time_elapsed))

    return scores

scores = dqn(n_episodes=2000, max_t=1500, eps_start=1.0, eps_end=0.01,␣
↪eps_decay=0.997, target_scores=100000.0)
```

```
Episode 100     Average Score: -4280.83
Episode 200     Average Score: -1434.59
Episode 232     Average Score: 1968.738    Episode finished after 705 timesteps
Episode 264     Average Score: 5094.82    Episode finished after 623 timesteps
Episode 273     Average Score: 6476.31    Episode finished after 251 timesteps
Episode 300     Average Score: 7440.57
Episode 316     Average Score: 6790.40    Episode finished after 310 timesteps
Episode 320     Average Score: 7800.94    Episode finished after 1485 timesteps
Episode 321     Average Score: 8737.19    Episode finished after 589 timesteps
Episode 328     Average Score: 9867.86    Episode finished after 1010 timesteps
Episode 354     Average Score: 10126.66    Episode finished after 629 timesteps
Episode 364     Average Score: 12208.22    Episode finished after 681 timesteps
Episode 400     Average Score: 13053.45
Episode 415     Average Score: 13356.43    Episode finished after 1103 timesteps
Episode 418     Average Score: 13778.51    Episode finished after 839 timesteps
Episode 430     Average Score: 11929.81    Episode finished after 582 timesteps
Episode 431     Average Score: 13016.34    Episode finished after 1045 timesteps
Episode 443     Average Score: 14651.57    Episode finished after 1321 timesteps
Episode 444     Average Score: 15915.44    Episode finished after 982 timesteps
Episode 445     Average Score: 16988.54    Episode finished after 962 timesteps
Episode 448     Average Score: 18144.50    Episode finished after 488 timesteps
Episode 481     Average Score: 17383.89    Episode finished after 1102 timesteps
Episode 482     Average Score: 18254.33    Episode finished after 1353 timesteps
Episode 483     Average Score: 19354.75    Episode finished after 639 timesteps
Episode 484     Average Score: 20468.21    Episode finished after 1116 timesteps
Episode 495     Average Score: 22203.83    Episode finished after 296 timesteps
Episode 500     Average Score: 23639.19
Episode 544     Average Score: 22190.52    Episode finished after 1490 timesteps
Episode 600     Average Score: 18213.24
Episode 607     Average Score: 17693.67    Episode finished after 148 timesteps
Episode 633     Average Score: 16447.75    Episode finished after 1362 timesteps
Episode 663     Average Score: 14165.85    Episode finished after 1343 timesteps
Episode 686     Average Score: 14110.01    Episode finished after 1185 timesteps
Episode 700     Average Score: 16093.09
```

```
Episode 706     Average Score: 17342.40    Episode finished after 692 timesteps
Episode 728     Average Score: 21077.50    Episode finished after 454 timesteps
Episode 732     Average Score: 22622.20    Episode finished after 569 timesteps
Episode 733     Average Score: 23516.84    Episode finished after 751 timesteps
Episode 746     Average Score: 25769.87    Episode finished after 1322 timesteps
Episode 747     Average Score: 27087.13    Episode finished after 1095 timesteps
Episode 751     Average Score: 28839.76    Episode finished after 1033 timesteps
Episode 762     Average Score: 31443.02    Episode finished after 1495 timesteps
Episode 764     Average Score: 31576.69    Episode finished after 1459 timesteps
Episode 767     Average Score: 33245.21    Episode finished after 1065 timesteps
Episode 776     Average Score: 35590.93    Episode finished after 1377 timesteps
Episode 777     Average Score: 36909.77    Episode finished after 983 timesteps
Episode 780     Average Score: 37774.89    Episode finished after 666 timesteps
Episode 781     Average Score: 38646.14    Episode finished after 619 timesteps
Episode 782     Average Score: 39508.82    Episode finished after 403 timesteps
Episode 786     Average Score: 40583.06    Episode finished after 333 timesteps
Episode 787     Average Score: 40534.85    Episode finished after 1449 timesteps
Episode 788     Average Score: 41920.65    Episode finished after 410 timesteps
Episode 789     Average Score: 42712.79    Episode finished after 479 timesteps
Episode 790     Average Score: 43460.00    Episode finished after 402 timesteps
Episode 791     Average Score: 44226.70    Episode finished after 382 timesteps
Episode 792     Average Score: 44990.60    Episode finished after 666 timesteps
Episode 793     Average Score: 45827.58    Episode finished after 692 timesteps
Episode 795     Average Score: 46701.95    Episode finished after 1267 timesteps
Episode 796     Average Score: 47941.82    Episode finished after 1416 timesteps
Episode 800     Average Score: 49045.14
Episode 804     Average Score: 48987.04    Episode finished after 713 timesteps
Episode 815     Average Score: 48805.46    Episode finished after 790 timesteps
Episode 816     Average Score: 49694.09    Episode finished after 442 timesteps
Episode 821     Average Score: 50686.48    Episode finished after 455 timesteps
Episode 822     Average Score: 51501.36    Episode finished after 532 timesteps
Episode 823     Average Score: 52326.65    Episode finished after 365 timesteps
Episode 824     Average Score: 53104.06    Episode finished after 795 timesteps
Episode 825     Average Score: 54043.83    Episode finished after 1059 timesteps
Episode 829     Average Score: 54243.01    Episode finished after 1187 timesteps
Episode 830     Average Score: 55207.94    Episode finished after 751 timesteps
Episode 831     Average Score: 56019.97    Episode finished after 782 timesteps
Episode 832     Average Score: 56883.01    Episode finished after 1300 timesteps
Episode 834     Average Score: 56055.01    Episode finished after 1394 timesteps
Episode 835     Average Score: 57050.12    Episode finished after 464 timesteps
Episode 836     Average Score: 57820.68    Episode finished after 511 timesteps
Episode 842     Average Score: 58879.70    Episode finished after 813 timesteps
Episode 843     Average Score: 59760.17    Episode finished after 295 timesteps
Episode 844     Average Score: 60556.57    Episode finished after 347 timesteps
Episode 850     Average Score: 59481.41    Episode finished after 600 timesteps
Episode 852     Average Score: 59310.15    Episode finished after 1432 timesteps
Episode 865     Average Score: 58624.85    Episode finished after 724 timesteps
Episode 867     Average Score: 59435.09    Episode finished after 934 timesteps
```

```
Episode 868     Average Score: 59467.61    Episode finished after 1450 timesteps
Episode 871     Average Score: 60652.23    Episode finished after 351 timesteps
Episode 873     Average Score: 61442.30    Episode finished after 306 timesteps
Episode 875     Average Score: 62296.39    Episode finished after 311 timesteps
Episode 877     Average Score: 62118.29    Episode finished after 1127 timesteps
Episode 878     Average Score: 62213.97    Episode finished after 1239 timesteps
Episode 879     Average Score: 63213.89    Episode finished after 489 timesteps
Episode 880     Average Score: 64053.02    Episode finished after 305 timesteps
Episode 881     Average Score: 63975.99    Episode finished after 1450 timesteps
Episode 885     Average Score: 63527.67    Episode finished after 282 timesteps
Episode 886     Average Score: 64216.61    Episode finished after 334 timesteps
Episode 888     Average Score: 63206.87    Episode finished after 355 timesteps
Episode 889     Average Score: 63195.09    Episode finished after 320 timesteps
Episode 890     Average Score: 63167.69    Episode finished after 855 timesteps
Episode 891     Average Score: 63312.24    Episode finished after 289 timesteps
Episode 893     Average Score: 62428.80    Episode finished after 718 timesteps
Episode 894     Average Score: 62441.16    Episode finished after 269 timesteps
Episode 895     Average Score: 63177.53    Episode finished after 162 timesteps
Episode 898     Average Score: 62039.15    Episode finished after 941 timesteps
Episode 899     Average Score: 62948.79    Episode finished after 864 timesteps
Episode 900     Average Score: 63870.74
Episode 901     Average Score: 63987.93    Episode finished after 1132 timesteps
Episode 902     Average Score: 64967.64    Episode finished after 298 timesteps
Episode 903     Average Score: 65760.24    Episode finished after 273 timesteps
Episode 904     Average Score: 66528.73    Episode finished after 1000 timesteps
Episode 905     Average Score: 66643.87    Episode finished after 297 timesteps
Episode 906     Average Score: 67494.66    Episode finished after 301 timesteps
Episode 907     Average Score: 68319.60    Episode finished after 183 timesteps
Episode 910     Average Score: 69191.43    Episode finished after 332 timesteps
Episode 911     Average Score: 69914.59    Episode finished after 907 timesteps
Episode 912     Average Score: 70798.28    Episode finished after 268 timesteps
Episode 918     Average Score: 69845.01    Episode finished after 301 timesteps
Episode 922     Average Score: 69928.87    Episode finished after 966 timesteps
Episode 923     Average Score: 70045.78    Episode finished after 1376 timesteps
Episode 924     Average Score: 70287.77    Episode finished after 325 timesteps
Episode 925     Average Score: 70166.61    Episode finished after 957 timesteps
Episode 927     Average Score: 70153.40    Episode finished after 163 timesteps
Episode 928     Average Score: 70885.51    Episode finished after 178 timesteps
Episode 929     Average Score: 71610.61    Episode finished after 150 timesteps
Episode 930     Average Score: 71382.08    Episode finished after 889 timesteps
Episode 931     Average Score: 71445.53    Episode finished after 818 timesteps
Episode 933     Average Score: 70498.86    Episode finished after 301 timesteps
Episode 935     Average Score: 70452.87    Episode finished after 187 timesteps
Episode 936     Average Score: 70394.06    Episode finished after 246 timesteps
Episode 937     Average Score: 70350.70    Episode finished after 209 timesteps
Episode 939     Average Score: 71016.48    Episode finished after 771 timesteps
Episode 942     Average Score: 71827.99    Episode finished after 230 timesteps
Episode 943     Average Score: 71713.35    Episode finished after 791 timesteps
```

```
Episode 947     Average Score: 71047.32    Episode finished after 284 timesteps
Episode 948     Average Score: 71789.64    Episode finished after 153 timesteps
Episode 949     Average Score: 72532.59    Episode finished after 338 timesteps
Episode 955     Average Score: 71636.72    Episode finished after 162 timesteps
Episode 956     Average Score: 72352.18    Episode finished after 1152 timesteps
Episode 957     Average Score: 73280.53    Episode finished after 686 timesteps
Episode 959     Average Score: 74052.52    Episode finished after 367 timesteps
Episode 962     Average Score: 74641.20    Episode finished after 1127 timesteps
Episode 967     Average Score: 74696.30    Episode finished after 1074 timesteps
Episode 969     Average Score: 73676.95    Episode finished after 1026 timesteps
Episode 970     Average Score: 74511.22    Episode finished after 186 timesteps
Episode 972     Average Score: 74455.99    Episode finished after 576 timesteps
Episode 973     Average Score: 75216.22    Episode finished after 151 timesteps
Episode 976     Average Score: 74426.71    Episode finished after 433 timesteps
Episode 977     Average Score: 75163.19    Episode finished after 1066 timesteps
Episode 980     Average Score: 73392.64    Episode finished after 396 timesteps
Episode 981     Average Score: 73419.19    Episode finished after 308 timesteps
Episode 983     Average Score: 73052.34    Episode finished after 152 timesteps
Episode 989     Average Score: 71351.60    Episode finished after 729 timesteps
Episode 990     Average Score: 71457.47    Episode finished after 682 timesteps
Episode 991     Average Score: 71426.86    Episode finished after 738 timesteps
Episode 994     Average Score: 70728.21    Episode finished after 603 timesteps
Episode 996     Average Score: 70084.07    Episode finished after 133 timesteps
Episode 999     Average Score: 69878.16    Episode finished after 156 timesteps
Episode 1000    Average Score: 69715.03
    Episode finished after 273 timesteps
Episode 1001    Average Score: 70415.56    Episode finished after 357 timesteps
Episode 1002    Average Score: 70217.90    Episode finished after 320 timesteps
Episode 1003    Average Score: 70219.49    Episode finished after 198 timesteps
Episode 1004    Average Score: 70197.63    Episode finished after 434 timesteps
Episode 1005    Average Score: 70021.01    Episode finished after 896 timesteps
Episode 1006    Average Score: 70168.65    Episode finished after 1495 timesteps
Episode 1007    Average Score: 70455.15    Episode finished after 1206 timesteps
Episode 1009    Average Score: 70709.16    Episode finished after 627 timesteps
Episode 1010    Average Score: 71497.28    Episode finished after 703 timesteps
Episode 1011    Average Score: 71586.54    Episode finished after 670 timesteps
Episode 1013    Average Score: 70829.94    Episode finished after 827 timesteps
Episode 1014    Average Score: 71674.31    Episode finished after 947 timesteps
Episode 1015    Average Score: 72681.81    Episode finished after 383 timesteps
Episode 1020    Average Score: 72523.28    Episode finished after 990 timesteps
Episode 1021    Average Score: 73385.80    Episode finished after 349 timesteps
Episode 1022    Average Score: 74061.19    Episode finished after 429 timesteps
Episode 1024    Average Score: 72868.30    Episode finished after 144 timesteps
Episode 1025    Average Score: 72830.33    Episode finished after 1293 timesteps
Episode 1026    Average Score: 72835.66    Episode finished after 400 timesteps
Episode 1031    Average Score: 70517.93    Episode finished after 733 timesteps
Episode 1034    Average Score: 69722.84    Episode finished after 376 timesteps
Episode 1035    Average Score: 70457.88    Episode finished after 391 timesteps
```

```
Episode 1036    Average Score: 70489.37    Episode finished after 1476 timesteps
Episode 1037    Average Score: 70689.51    Episode finished after 644 timesteps
Episode 1041    Average Score: 69990.85    Episode finished after 1031 timesteps
Episode 1044    Average Score: 69179.31    Episode finished after 229 timesteps
Episode 1046    Average Score: 69877.82    Episode finished after 499 timesteps
Episode 1048    Average Score: 69924.13    Episode finished after 888 timesteps
Episode 1049    Average Score: 70081.13    Episode finished after 1176 timesteps
Episode 1051    Average Score: 70259.90    Episode finished after 798 timesteps
Episode 1052    Average Score: 71051.22    Episode finished after 671 timesteps
Episode 1053    Average Score: 71885.81    Episode finished after 546 timesteps
Episode 1054    Average Score: 72672.21    Episode finished after 652 timesteps
Episode 1057    Average Score: 71826.99    Episode finished after 306 timesteps
Episode 1058    Average Score: 71722.60    Episode finished after 930 timesteps
Episode 1062    Average Score: 72095.70    Episode finished after 1495 timesteps
Episode 1063    Average Score: 72150.01    Episode finished after 496 timesteps
Episode 1065    Average Score: 72814.10    Episode finished after 663 timesteps
Episode 1066    Average Score: 73636.35    Episode finished after 1027 timesteps
Episode 1068    Average Score: 73633.65    Episode finished after 444 timesteps
Episode 1069    Average Score: 74446.40    Episode finished after 1250 timesteps
Episode 1071    Average Score: 73794.05    Episode finished after 468 timesteps
Episode 1072    Average Score: 74507.64    Episode finished after 370 timesteps
Episode 1075    Average Score: 73730.46    Episode finished after 1234 timesteps
Episode 1076    Average Score: 74694.64    Episode finished after 463 timesteps
Episode 1077    Average Score: 74687.06    Episode finished after 682 timesteps
Episode 1078    Average Score: 74583.85    Episode finished after 550 timesteps
Episode 1079    Average Score: 75361.56    Episode finished after 475 timesteps
Episode 1080    Average Score: 76065.17    Episode finished after 548 timesteps
Episode 1084    Average Score: 74497.57    Episode finished after 630 timesteps
Episode 1085    Average Score: 75275.81    Episode finished after 210 timesteps
Episode 1086    Average Score: 76103.72    Episode finished after 481 timesteps
Episode 1087    Average Score: 76861.61    Episode finished after 431 timesteps
Episode 1088    Average Score: 77557.62    Episode finished after 225 timesteps
Episode 1090    Average Score: 77482.54    Episode finished after 902 timesteps
Episode 1091    Average Score: 77510.43    Episode finished after 838 timesteps
Episode 1094    Average Score: 77438.46    Episode finished after 587 timesteps
Episode 1096    Average Score: 77426.44    Episode finished after 367 timesteps
Episode 1097    Average Score: 77456.22    Episode finished after 445 timesteps
Episode 1100    Average Score: 77385.85
Episode 1101    Average Score: 76644.07    Episode finished after 304 timesteps
Episode 1102    Average Score: 76626.00    Episode finished after 567 timesteps
Episode 1106    Average Score: 74315.58    Episode finished after 724 timesteps
Episode 1107    Average Score: 74067.72    Episode finished after 494 timesteps
Episode 1108    Average Score: 73843.04    Episode finished after 314 timesteps
Episode 1109    Average Score: 74564.58    Episode finished after 350 timesteps
Episode 1111    Average Score: 73656.35    Episode finished after 873 timesteps
Episode 1112    Average Score: 73646.85    Episode finished after 727 timesteps
Episode 1113    Average Score: 74386.98    Episode finished after 634 timesteps
Episode 1114    Average Score: 74342.47    Episode finished after 633 timesteps
```

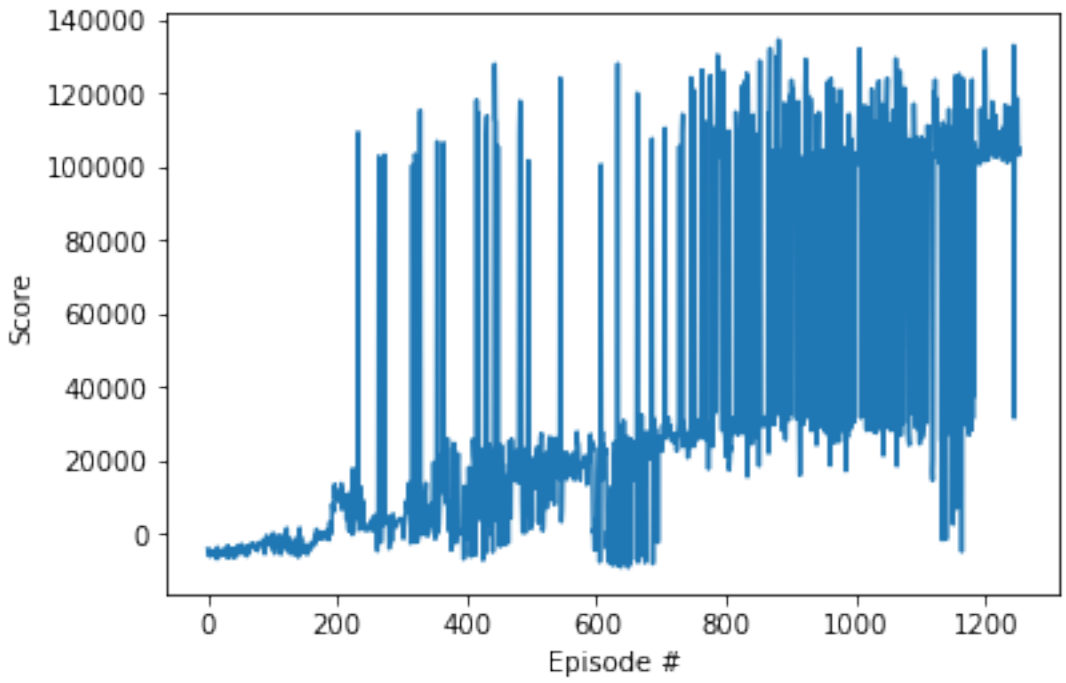
```
Episode 1115    Average Score: 74208.32    Episode finished after 357 timesteps
Episode 1116    Average Score: 74195.57    Episode finished after 515 timesteps
Episode 1117    Average Score: 74929.94    Episode finished after 371 timesteps
Episode 1121    Average Score: 74444.03    Episode finished after 1231 timesteps
Episode 1122    Average Score: 74614.25    Episode finished after 969 timesteps
Episode 1123    Average Score: 74725.96    Episode finished after 1338 timesteps
Episode 1124    Average Score: 75678.68    Episode finished after 722 timesteps
Episode 1125    Average Score: 75730.14    Episode finished after 493 timesteps
Episode 1126    Average Score: 75547.12    Episode finished after 1138 timesteps
Episode 1127    Average Score: 75677.87    Episode finished after 580 timesteps
Episode 1128    Average Score: 76418.58    Episode finished after 616 timesteps
Episode 1129    Average Score: 77175.69    Episode finished after 858 timesteps
Episode 1130    Average Score: 77992.80    Episode finished after 251 timesteps
Episode 1131    Average Score: 78702.77    Episode finished after 632 timesteps
Episode 1133    Average Score: 78612.73    Episode finished after 505 timesteps
Episode 1135    Average Score: 78297.40    Episode finished after 781 timesteps
Episode 1136    Average Score: 78370.31    Episode finished after 214 timesteps
Episode 1137    Average Score: 78143.83    Episode finished after 667 timesteps
Episode 1138    Average Score: 78081.25    Episode finished after 647 timesteps
Episode 1141    Average Score: 78515.19    Episode finished after 634 timesteps
Episode 1142    Average Score: 78453.41    Episode finished after 578 timesteps
Episode 1143    Average Score: 79259.37    Episode finished after 570 timesteps
Episode 1144    Average Score: 80085.52    Episode finished after 1036 timesteps
Episode 1145    Average Score: 80225.61    Episode finished after 475 timesteps
Episode 1146    Average Score: 80978.95    Episode finished after 663 timesteps
Episode 1148    Average Score: 80974.81    Episode finished after 598 timesteps
Episode 1149    Average Score: 80856.46    Episode finished after 553 timesteps
Episode 1150    Average Score: 80641.38    Episode finished after 603 timesteps
Episode 1152    Average Score: 80312.20    Episode finished after 1371 timesteps
Episode 1154    Average Score: 79308.01    Episode finished after 616 timesteps
Episode 1155    Average Score: 79240.29    Episode finished after 1381 timesteps
Episode 1157    Average Score: 79981.46    Episode finished after 1331 timesteps
Episode 1158    Average Score: 79963.16    Episode finished after 1081 timesteps
Episode 1159    Average Score: 79989.08    Episode finished after 1087 timesteps
Episode 1160    Average Score: 80872.33    Episode finished after 1465 timesteps
Episode 1162    Average Score: 81560.93    Episode finished after 583 timesteps
Episode 1163    Average Score: 81356.87    Episode finished after 955 timesteps
Episode 1165    Average Score: 81227.58    Episode finished after 737 timesteps
Episode 1166    Average Score: 81147.49    Episode finished after 1302 timesteps
Episode 1167    Average Score: 81192.42    Episode finished after 702 timesteps
Episode 1169    Average Score: 81263.16    Episode finished after 690 timesteps
Episode 1170    Average Score: 81087.92    Episode finished after 400 timesteps
Episode 1171    Average Score: 81864.63    Episode finished after 392 timesteps
Episode 1172    Average Score: 81873.24    Episode finished after 969 timesteps
Episode 1173    Average Score: 81992.66    Episode finished after 397 timesteps
Episode 1174    Average Score: 82725.90    Episode finished after 1214 timesteps
Episode 1176    Average Score: 82493.42    Episode finished after 578 timesteps
Episode 1177    Average Score: 82520.11    Episode finished after 715 timesteps
```

```
Episode 1178    Average Score: 82532.65    Episode finished after 362 timesteps
Episode 1180    Average Score: 81747.64    Episode finished after 1206 timesteps
Episode 1182    Average Score: 81983.20    Episode finished after 531 timesteps
Episode 1184    Average Score: 82915.33    Episode finished after 476 timesteps
Episode 1185    Average Score: 82904.81    Episode finished after 407 timesteps
Episode 1186    Average Score: 82945.44    Episode finished after 315 timesteps
Episode 1187    Average Score: 82920.86    Episode finished after 355 timesteps
Episode 1188    Average Score: 82920.94    Episode finished after 264 timesteps
Episode 1189    Average Score: 82933.35    Episode finished after 337 timesteps
Episode 1190    Average Score: 83649.71    Episode finished after 270 timesteps
Episode 1191    Average Score: 83497.55    Episode finished after 251 timesteps
Episode 1192    Average Score: 83451.63    Episode finished after 221 timesteps
Episode 1193    Average Score: 84167.17    Episode finished after 825 timesteps
Episode 1194    Average Score: 85033.46    Episode finished after 600 timesteps
Episode 1195    Average Score: 85041.36    Episode finished after 896 timesteps
Episode 1196    Average Score: 85898.67    Episode finished after 595 timesteps
Episode 1197    Average Score: 85951.55    Episode finished after 542 timesteps
Episode 1198    Average Score: 86013.89    Episode finished after 446 timesteps
Episode 1199    Average Score: 86776.51    Episode finished after 342 timesteps
Episode 1200    Average Score: 87546.29
    Episode finished after 1317 timesteps
Episode 1201    Average Score: 88578.68    Episode finished after 1084 timesteps
Episode 1202    Average Score: 88772.01    Episode finished after 316 timesteps
Episode 1203    Average Score: 88708.99    Episode finished after 531 timesteps
Episode 1204    Average Score: 89490.03    Episode finished after 590 timesteps
Episode 1205    Average Score: 90234.84    Episode finished after 455 timesteps
Episode 1206    Average Score: 90942.48    Episode finished after 273 timesteps
Episode 1207    Average Score: 90892.78    Episode finished after 667 timesteps
Episode 1208    Average Score: 90978.47    Episode finished after 588 timesteps
Episode 1209    Average Score: 91019.12    Episode finished after 393 timesteps
Episode 1210    Average Score: 91034.01    Episode finished after 799 timesteps
Episode 1211    Average Score: 91848.18    Episode finished after 943 timesteps
Episode 1212    Average Score: 91913.52    Episode finished after 838 timesteps
Episode 1213    Average Score: 91967.11    Episode finished after 567 timesteps
Episode 1214    Average Score: 91913.78    Episode finished after 483 timesteps
Episode 1215    Average Score: 91920.58    Episode finished after 979 timesteps
Episode 1216    Average Score: 91935.25    Episode finished after 357 timesteps
Episode 1217    Average Score: 91935.67    Episode finished after 619 timesteps
Episode 1218    Average Score: 92039.69    Episode finished after 301 timesteps
Episode 1219    Average Score: 92893.80    Episode finished after 457 timesteps
Episode 1220    Average Score: 93777.94    Episode finished after 409 timesteps
Episode 1221    Average Score: 94615.85    Episode finished after 524 timesteps
Episode 1222    Average Score: 94470.94    Episode finished after 764 timesteps
Episode 1223    Average Score: 94423.87    Episode finished after 679 timesteps
Episode 1224    Average Score: 94270.32    Episode finished after 470 timesteps
Episode 1225    Average Score: 94271.55    Episode finished after 629 timesteps
Episode 1226    Average Score: 94335.08    Episode finished after 310 timesteps
Episode 1227    Average Score: 94168.58    Episode finished after 354 timesteps
```

```
Episode 1228    Average Score: 94155.41    Episode finished after 438 timesteps
Episode 1229    Average Score: 94169.19    Episode finished after 729 timesteps
Episode 1230    Average Score: 94178.12    Episode finished after 577 timesteps
Episode 1231    Average Score: 94226.62    Episode finished after 860 timesteps
Episode 1232    Average Score: 94338.42    Episode finished after 506 timesteps
Episode 1233    Average Score: 95128.12    Episode finished after 498 timesteps
Episode 1234    Average Score: 95138.56    Episode finished after 350 timesteps
Episode 1235    Average Score: 96165.91    Episode finished after 381 timesteps
Episode 1236    Average Score: 96099.95    Episode finished after 586 timesteps
Episode 1237    Average Score: 96155.49    Episode finished after 742 timesteps
Episode 1238    Average Score: 96274.61    Episode finished after 441 timesteps
Episode 1239    Average Score: 96285.93    Episode finished after 417 timesteps
Episode 1240    Average Score: 97016.48    Episode finished after 619 timesteps
Episode 1241    Average Score: 98048.04    Episode finished after 401 timesteps
Episode 1242    Average Score: 97973.76    Episode finished after 480 timesteps
Episode 1243    Average Score: 97911.61    Episode finished after 394 timesteps
Episode 1244    Average Score: 97884.83    Episode finished after 1363 timesteps
Episode 1246    Average Score: 97327.80    Episode finished after 311 timesteps
Episode 1247    Average Score: 97243.37    Episode finished after 423 timesteps
Episode 1248    Average Score: 98015.88    Episode finished after 582 timesteps
Episode 1249    Average Score: 98042.92    Episode finished after 445 timesteps
Episode 1250    Average Score: 98049.97    Episode finished after 968 timesteps
Episode 1251    Average Score: 98185.73    Episode finished after 600 timesteps
Episode 1252    Average Score: 99208.51    Episode finished after 415 timesteps
Episode 1253    Average Score: 99220.89    Episode finished after 664 timesteps
Episode 1254    Average Score: 100087.02
Environment solved in 1254 episodes!    Average Score: 100087.02
Time Elapse: 1705.75
```

1.3.2 Score plot for each episodes during training

[6]:
```
# plot the scores
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores)
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.savefig('plots/model_training.png', dpi = 200) #UNCOMMENT TO SAVE PLOT
plt.show()
```



1.4 Analysis and inspections

1.4.1 Watch the agent running (Using saved weights)

[7]:
```
# load the weights from file
agent.qnetwork_local.load_state_dict(torch.load('checkpoint_precise_2.pth'))
#agent.qnetwork_local.load_state_dict(torch.load('checkpoint_precise.pth'))



env_info = env.reset()                                  # reset the environment
state = env.state                                       # get the current state
score = 0                                               # initialize the score

t = 0
while True:
    env.render()
    time.sleep(0.008)
    action = agent.act(state)                           # select an action
    env_info = env.step(action)                         # send the action to the␣
 ↪environment
    next_state = env_info[0]                            # get the next state
    reward = env_info[1]                                # get the reward
    done = env_info[2]                                  # see if episode has finished
    score += reward                                     # update the score
```

```
    state = next_state  # roll over the state to next time step
    t += 1
    if done:                                           # exit loop if episode␣
↪finished
        print("final state is :", state)
        print("Total steps : ", t)
        break

print("Score: {}".format(score))
env.close()
```

```
final state is : [ 1.01893092e+00  2.26878363e-03  3.12455368e+00
-2.64702502e-02]
Total steps :  512
Score: 103213.0819224108
```

1.4.2 Calculate time steps and time before solving environement

[14]:
```
#Choose model
agent.qnetwork_local.load_state_dict(torch.load('checkpoint_precise.pth'))

env_info = env.reset()                                 # reset the environment
state = env.state                                      # get the current state
score = 0                                              # initialize the score

t = 0
while True:
    env.render()
    #time.sleep(0.02)                                   # Actual time step
    action = agent.act(state)                          # select an action
    env_info = env.step(action)                        # send the action to the␣
↪environment
    next_state = env_info[0]                           # get the next state
    reward = env_info[1]                               # get the reward
    done = env_info[2]                                 # see if episode has finished
    score += reward                                    # update the score
    state = next_state  # roll over the state to next time step
    t += 1
    if done:                                           # exit loop if episode␣
↪finished
        print("\r final state is :", state)
        print("\r Total steps : ", t)
        print("\r Total time is : ", env.tau * t)
        break

print("Score: {}".format(score))
env.close()
```

```
 final state is : [ 0.99739885  0.0387832   3.17148444 -0.00326196]
 Total steps :  393
 Total time is :  7.86
Score: 105153.49475149131
```

1.4.3 Plot state graphs against time

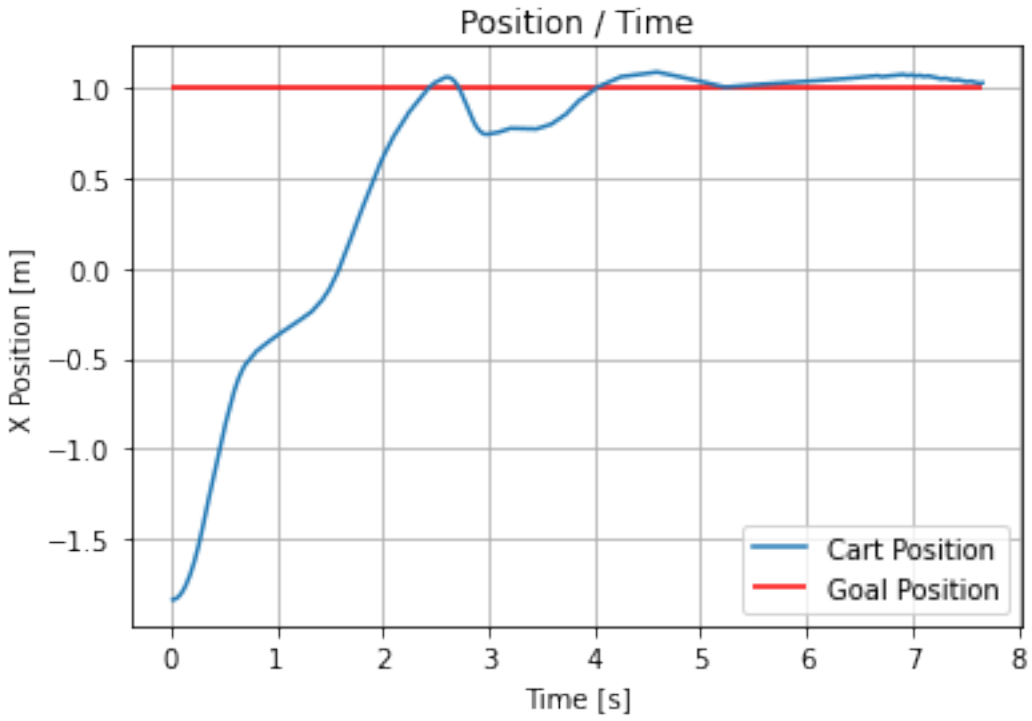
[15]:
```
#Choose model
agent.qnetwork_local.load_state_dict(torch.load('checkpoint_precise.pth'))
agent.qnetwork_local.state_dict() #UNCOMMENT TO USE LAST TRAINED WEIGHTS


env_info = env.reset()                                  # reset the environment
state = env.state                                       # get the current state
score = 0                                               # initialize the score

arr_x = []
arr_x_dot = []
arr_theta = []
arr_theta_dot = []
arr_t = []

t = 0.0
while True:
    env.render()
    #time.sleep(0.008)
    action = agent.act(state)                           # select an action
    env_info = env.step(action)                         # send the action to the␣
 ↪environment
    next_state = env_info[0]                            # get the next state
    reward = env_info[1]                                # get the reward
    done = env_info[2]                                  # see if episode has finished
    score += reward                                     # update the score
    state = next_state  # roll over the state to next time step
    t += 1.0
    arr_t.append(t)
    arr_x.append(state[0])
    arr_x_dot.append(state[1])
    arr_theta.append(state[2])
    arr_theta_dot.append(state[3])

    if done:                                            # exit loop if episode␣
 ↪finished
        print("Total steps : ", t)
        print("Total time is : ", env.tau * t)
        break
arr_t = 0.02*np.array(arr_t)
```

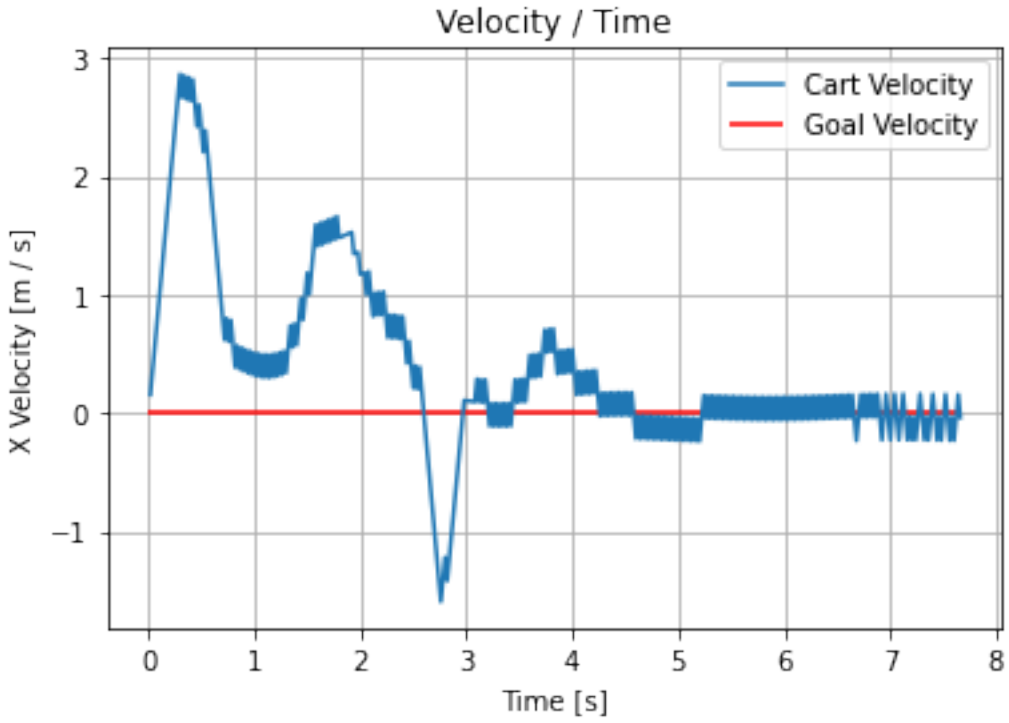
```
print("Score: {}".format(score))
env.close()
```

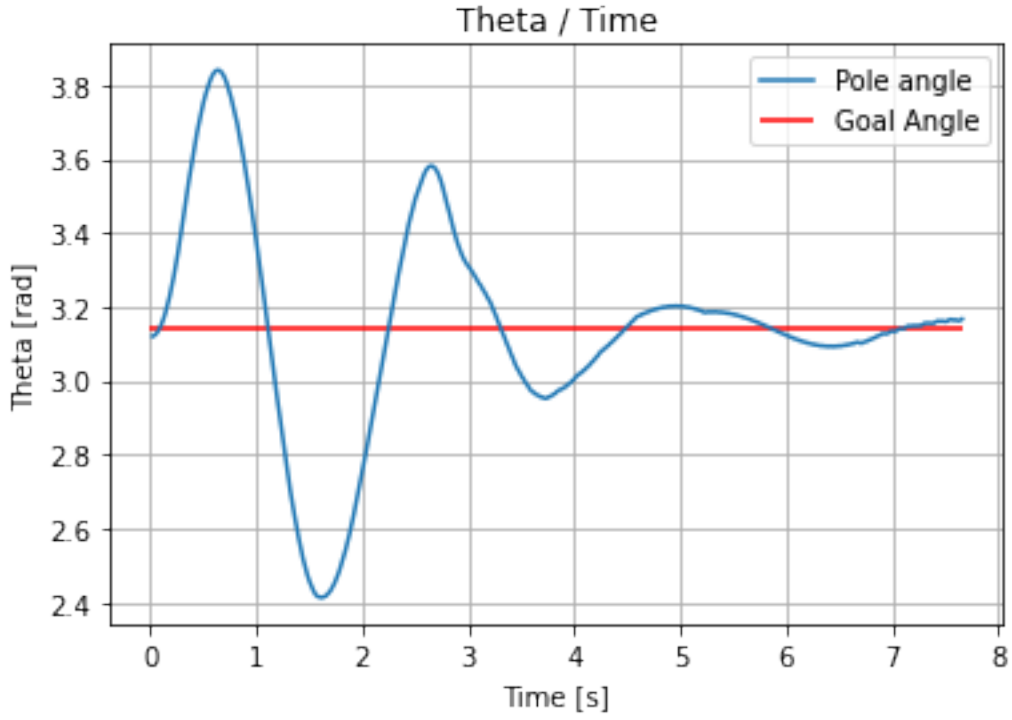
```
Total steps :  383.0
Total time is :  7.66
Score: 104468.93876048437
```

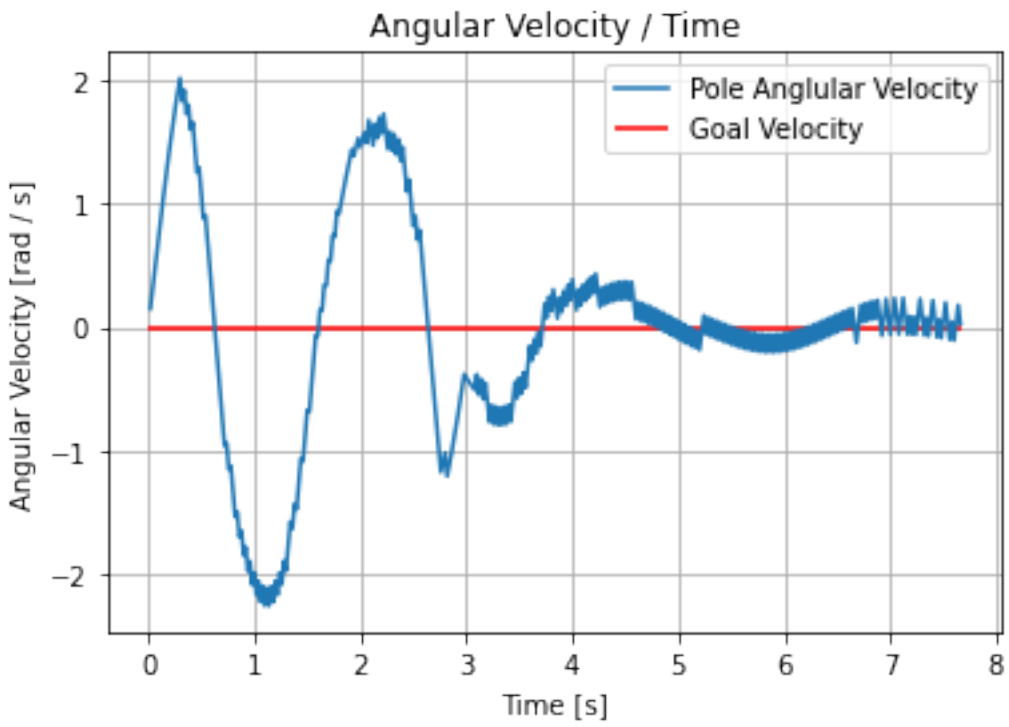
[16]:
```
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(arr_t, arr_x, label='Cart Position')
plt.ylabel('X Position [m]')
plt.xlabel('Time [s]')
plt.title('Position / Time')
plt.hlines(1.0, 0, arr_t[-1], colors='r', linestyles='solid', label='Goal␣
 ↪Position')
plt.legend()
plt.grid()
#plt.savefig('plots/model_1_x.png', dpi = 200)
plt.show()

fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(arr_t, arr_x_dot, label = 'Cart Velocity')
plt.ylabel('X Velocity [m / s]')
plt.xlabel('Time [s]')
plt.title('Velocity / Time')
plt.hlines(0.0, 0, arr_t[-1], colors='r', linestyles='solid', label='Goal␣
 ↪Velocity')
plt.legend()
plt.grid()
#plt.savefig('plots/model_1_x_dot.png', dpi = 200)
plt.show()

fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(arr_t, arr_theta, label = 'Pole angle')
plt.ylabel('Theta [rad]')
plt.xlabel('Time [s]')
plt.title('Theta / Time')
plt.hlines(np.pi, 0, arr_t[-1], colors='r', linestyles='solid', label='Goal␣
 ↪Angle')
plt.legend()
plt.grid()
#plt.savefig('plots/model_1_theta.png', dpi = 200)
plt.show()

fig = plt.figure()
```

```
ax = fig.add_subplot(111)
plt.plot(arr_t, arr_theta_dot, label = 'Pole Anglular Velocity')
plt.ylabel('Angular Velocity [rad / s]')
plt.xlabel('Time [s]')
plt.title('Angular Velocity / Time')
plt.hlines(0.0, 0, arr_t[-1], colors='r', linestyles='solid', label='Goal␣
 ↪Velocity')
plt.legend()
plt.grid()
#plt.savefig('plots/model_1_theta_dot.png', dpi = 200)
plt.show()

initial_state = [arr_x[0], arr_x_dot[0], arr_theta[0], arr_theta_dot[0]]
print('initial state is : ', initial_state)
```




Velocity / Time
Cart Velocity
Goal Velocity
X Velocity [m / s]
Time [s]
3
2
1
0
−1
0
1
2
3
4
5
6
7
8


Theta / Time
Pole angle
Goal Angle
Theta [rad]
Time [s]
3.8
3.6
3.4
3.2
3.0
2.8
2.6
2.4
0
1
2
3
4
5
6
7
8




```
initial state is :  [-1.8381194721045466, 0.1541213842217936, 3.12001952945492,
0.14986158839055636]
```

James002

June 11, 2021

1 Crane V1 model training and simulations

Adapted from https://github.com/kinwo/deeprl-navigation (MIT License Copyright (c) 2018 Henry Chan)

Start Environement and create DQN Agent

[1]:
```
import gym
import numpy as np

env = gym.make('crane-v1') #Load the environement
```

1.1 Agent

The DQN agent can be found below

[2]:
```
import numpy as np
import random
from collections import namedtuple, deque

#from model import QNetwork # UNCOMMENT IF YOU ARE NOT IN A JUPYTER NOTEBOOK

import torch
import torch.nn.functional as F
import torch.optim as optim

BUFFER_SIZE = 5*int(1e5)   # replay buffer size
BATCH_SIZE = 64            # minibatch size
GAMMA = 0.995 #was 0.99    # discount factor
TAU = 1e-3                 # for soft update of target parameters
LR =  5e-4                 # learning rate
UPDATE_EVERY = 4           # how often to update the network

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class Agent():
    """Interacts with and learns from the environment."""

    def __init__(self, state_size, action_size, seed):
```

```
        """Initialize an Agent object.

        Params
        ======
            state_size (int): dimension of each state
            action_size (int): dimension of each action
            seed (int): random seed
        """



        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)

        # Q-Network
        self.qnetwork_local = QNetwork(state_size, action_size, seed).to(device)
        self.qnetwork_target = QNetwork(state_size, action_size, seed).
↪to(device)
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=LR)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)
        # Initialize time step (for updating every UPDATE_EVERY steps)
        self.t_step = 0

    def step(self, state, action, reward, next_state, done):
        # Save experience in replay memory
        self.memory.add(state, action, reward, next_state, done)

        # Learn every UPDATE_EVERY time steps.
        self.t_step = (self.t_step + 1) % UPDATE_EVERY
        if self.t_step == 0:
            # If enough samples are available in memory, get random subset and␣
↪learn
            if len(self.memory) > BATCH_SIZE:
                experiences = self.memory.sample()
                self.learn(experiences, GAMMA)

    def act(self, state, eps=0.):
        """Returns actions for given state as per current policy.

        Params
        ======
            state (array_like): current state
            eps (float): epsilon, for epsilon-greedy action selection
        """
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
```

```
        self.qnetwork_local.eval()
        with torch.no_grad():
            action_values = self.qnetwork_local(state)
        self.qnetwork_local.train()

        # Epsilon-greedy action selection
        if random.random() > eps:
            return np.argmax(action_values.cpu().data.numpy())
        else:
            return random.choice(np.arange(self.action_size))

    def learn(self, experiences, gamma):
        """Update value parameters using given batch of experience tuples.

        Params
        ======
            experiences (Tuple[torch.Variable]): tuple of (s, a, r, s', done)␣
↪tuples
            gamma (float): discount factor
        """
        states, actions, rewards, next_states, dones = experiences

        # Get max predicted Q values (for next states) from target model
        Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].
↪unsqueeze(1)
        # Compute Q targets for current states
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

        # Get expected Q values from local model
        Q_expected = self.qnetwork_local(states).gather(1, actions)

        # Compute loss
        loss = F.mse_loss(Q_expected, Q_targets)

        # Minimize the loss
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # ------------------- update target network ------------------- #
        self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)

    def soft_update(self, local_model, target_model, tau):
        """Soft update model parameters.
        _target = *_local + (1 - )*_target

        Params
```

```
        ======
            local_model (PyTorch model): weights will be copied from
            target_model (PyTorch model): weights will be copied to
            tau (float): interpolation parameter
        """
        for target_param, local_param in zip(target_model.parameters(),␣
↪local_model.parameters()):
            target_param.data.copy_(tau*local_param.data + (1.
↪0-tau)*target_param.data)


class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, action_size, buffer_size, batch_size, seed):
        """Initialize a ReplayBuffer object.

        Params
        ======
            action_size (int): dimension of each action
            buffer_size (int): maximum size of buffer
            batch_size (int): size of each training batch
            seed (int): random seed
        """

        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state",␣
↪"action", "reward", "next_state", "done"])
        self.seed = random.seed(seed)

    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        experiences = random.sample(self.memory, k=self.batch_size)

        states = torch.from_numpy(np.vstack([e.state for e in experiences if e␣
↪is not None])).float().to(device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if␣
↪e is not None])).long().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if␣
↪e is not None])).float().to(device)
```

```
        next_states = torch.from_numpy(np.vstack([e.next_state for e in␣
↪experiences if e is not None])).float().to(device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is␣
↪not None]).astype(np.uint8)).float().to(device)

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)
```

1.2 Deep Q_Network Model

A 2 linear hidden layer of 64 nodes each is created, with relu activation function.

[3]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

class QNetwork(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=64,␣
↪fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)



    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

[4]:
```
import torch
import time
from collections import deque
#from agent import Agent   # UNCOMMENT IF YOU ARE NOT IN A JUPYTER NOTEBOOK
import matplotlib.pyplot as plt
%matplotlib inline

state_size=8        #State size of  environement
action_size=9       #Action size of enviroenemnt
seed=0
agent = Agent(state_size=8, action_size=9, seed=0) #setting the agent's␣
 ↪parameters
```

1.3 DQN Agent Training

Here, we save the model every 100 timesteps, in order to observe how the agent performs over it's training process. The files are located in the working derictory inder the name 'Episode_###.path'.

In order to know when the environement is solved, we compute the moving score (the total rewards per episode) average over the last 100 episodes. If the moving average is over a chosen thershold (target_scores), the model is then saved to 'model_weight_name'.

For the Crane_v0 environement, the target score is 10 000 000, since it is the reward obtained by the agent when finding the flag.

[5]:
```
model_weight_name = 'model_2.pth'

def dqn(n_episodes=1300, max_t=1500, eps_start=1.0, eps_end=0.01, eps_decay=0.
 ↪996, target_scores=200000.0):
    """Deep Q-Learning.

    Params
    ======
        n_episodes (int): maximum number of training episodes
        max_t (int): maximum number of timesteps per episode
        eps_start (float): starting value of epsilon, for epsilon-greedy action␣
 ↪selection
        eps_end (float): minimum value of epsilon
        eps_decay (float): multiplicative factor (per episode) for decreasing␣
 ↪epsilon
        target_scores (float): average scores aming to achieve, the agent will␣
 ↪stop training once it reaches this scores
    """
    start = time.time()                # Start time
    scores = []                        # list containing scores from each␣
 ↪episode
    scores_window = deque(maxlen=100)  # last 100 scores
```

```
    eps = eps_start                    # initialize epsilon

    for i_episode in range(1, n_episodes+1):
        # Reset env and score at the beginning of episode
        env_info = env.reset()                                  # reset the␣
↪environment
        state = env.state                                       # get the current␣
↪state
        score = 0                                               # initialize the␣
↪score

        for t in range(max_t):
            action = agent.act(state, eps)
            env_info = env.step(action)                         # send the action to␣
↪the environment
            next_state = env_info[0]                            # get the next state
            reward = env_info[1]                                # get the reward
            done = env_info[2]                                  # see if episode has␣
↪finished

            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward
            if done:
                print("\n Episode finished after {} timesteps".format(t+1))
                print("\n final state is :", state)
                print("\n Reward is : ", score)
                break

        scores_window.append(score)       # saving the most recent score
        scores.append(score)              # saving the most recent score
        eps = max(eps_end, eps_decay*eps) # decrease of the epsilon value


        print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.
↪mean(scores_window)), end="")

        if i_episode % 100 == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.
↪mean(scores_window)))
            temp_model_weight_name = 'Episode_{}.pth'.format(i_episode)
            torch.save(agent.qnetwork_local.state_dict(),␣
↪temp_model_weight_name)

        if np.mean(scores_window)>=target_scores:
```

```
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.
 ↪2f}'.format(i_episode, np.mean(scores_window)))
            torch.save(agent.qnetwork_local.state_dict(), model_weight_name)
            break

    time_elapsed = time.time() - start
    print("Time Elapse: {:.2f}".format(time_elapsed))

    return scores

scores = dqn(n_episodes=1500, max_t=1500, eps_start=1.0, eps_end=0.01,␣
 ↪eps_decay=0.997, target_scores=10000000.0)

# plot the scores


fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores)
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.title('Training Scores')
plt.ylim(0,200000)
plt.grid()
#plt.savefig('plots/model_training.png', dpi = 200)
plt.show()
```

```
Episode 100     Average Score: -3969.27
Episode 200     Average Score: 3409.393
Episode 300     Average Score: 23349.56
Episode 400     Average Score: 29454.76
Episode 403     Average Score: 37312.47
 Episode finished after 263 timesteps

 final state is : [ 0.85394743 -0.10502536  0.13972463  0.02307469  2.6424625
-0.11131322
  1.          0.        ]

 Reward is :  10195930.55498117
Episode 500     Average Score: 118817.88
Episode 521     Average Score: 20739.251
 Episode finished after 556 timesteps

 final state is : [0.99944954 0.10081445 0.13379506 0.13954825 2.63855876
0.01224912
 1.         0.        ]
```

```
 Reward is :  10494347.836950721
Episode 524     Average Score: 127555.84
 Episode finished after 405 timesteps

 final state is : [ 1.12599146  0.05368165  0.15014753 -0.08265099  2.58049004
0.1238463
  1.          0.        ]

 Reward is :  10310822.79713091
Episode 600     Average Score: 244197.14
Episode 647     Average Score: 31909.325
 Episode finished after 573 timesteps

 final state is : [ 0.85988265  0.14275462 -0.12597401  0.05559806  2.35281866
-0.04616643
  1.          0.        ]

 Reward is :  10304205.156921364
Episode 648     Average Score: 134640.83
 Episode finished after 399 timesteps

 final state is : [ 0.89417387 -0.0647649  -0.14247715  0.01007087  2.46974975
0.14684254
  1.          0.        ]

 Reward is :  10088862.139571585
Episode 700     Average Score: 260362.59
Episode 800     Average Score: 2878.6580
Episode 825     Average Score: 15154.59
 Episode finished after 495 timesteps

 final state is : [ 0.89100194  0.11704317  0.1308887   0.1412988   2.35031599
-0.14422304
  1.          0.        ]

 Reward is :  10062953.187467009
Episode 900     Average Score: 113196.48
Episode 1000    Average Score: 56474.14
Episode 1007    Average Score: 59977.43
 Episode finished after 1451 timesteps

 final state is : [ 0.90705112 -0.09222657 -0.02091674  0.04364559  2.61285118
-0.07013578
  1.          0.        ]

 Reward is :  10714545.061859686
Episode 1008    Average Score: 167171.63
 Episode finished after 379 timesteps
```

```
 final state is : [ 0.91857128 -0.1255766  -0.02298296 -0.09955415  2.64907534
-0.10425144
  1.          0.        ]

 Reward is :  10249641.493185386
Episode 1100    Average Score: 225159.43
Episode 1200    Average Score: -15361.47
Episode 1300    Average Score: -14239.81
Episode 1400    Average Score: 22366.108
Episode 1450    Average Score: 34919.97
 Episode finished after 605 timesteps

 final state is : [0.85329626 0.11502913 0.12468338 0.06534624 2.59017
0.07007034
 1.         0.        ]

 Reward is :  10148654.368939364
Episode 1453    Average Score: 150735.02
 Episode finished after 902 timesteps

 final state is : [ 0.92615044  0.01614251 -0.14947131  0.11777681  2.48696465
-0.0591223
  1.          0.        ]

 Reward is :  10530245.60758538
Episode 1454    Average Score: 255729.19
 Episode finished after 1174 timesteps

 final state is : [ 1.05514293  0.14645649 -0.10492286 -0.03370872  2.61801705
0.06117462
  1.          0.        ]

 Reward is :  10775762.993256772
Episode 1457    Average Score: 367725.52
 Episode finished after 1335 timesteps

 final state is : [0.91735891 0.06286596 0.0940211  0.14369615 2.53144169
0.11504205
 1.         0.        ]

 Reward is :  10443874.935163798
Episode 1458    Average Score: 466574.41
 Episode finished after 564 timesteps

 final state is : [ 0.95694793 -0.05436414 -0.14612674 -0.14537259  2.61727134
0.00866986
  1.          0.        ]
```

```
 Reward is :  10159304.436219815
Episode 1500    Average Score: 583216.84
Time Elapse: 1473.47
```

Since the environement is not solved, we choose the best model from where the score plot converges before diverging. After trial and error, it is found to be arround episode 600. We renamed the model "Episode_600.pth' to 'model.pth' and visualised it in the next block.

In order to plot the state against time, we save the data points in local arrays.

[20]:
```
#Choose model
#agent.qnetwork_local.state_dict()

agent.qnetwork_local.load_state_dict(torch.load('model.pth'))

env_info = env.reset()                                  # reset the environment
state = env.state                                       # get the current state
score = 0                                               # initialize the score

#Initialising the state arrays
arr_x = []
arr_x_dot = []
arr_theta = []
arr_theta_dot = []
arr_l = []
arr_l_dot = []
arr_t = []

t = 0
max_t = 1000                                            # Number of timesteps
for t in range(max_t) :
    env.render()
    #time.sleep(0.008)
    action = agent.act(state)                           # select an action
    env_info = env.step(action)                         # send the action to the␣
 ↪environment
    next_state = env_info[0]                            # get the next state
    reward = env_info[1]                                # get the reward
    done = env_info[2]                                  # see if episode has finished
    score += reward                                     # update the score
    state = next_state                                  # roll over the state to␣
 ↪next time step
    t += 1.0

    #Saving states in state matrix for the plot
    arr_t.append(t)
    arr_x.append(state[0])
```

```
    arr_x_dot.append(state[1])
    arr_theta.append(state[2])
    arr_theta_dot.append(state[3])
    arr_l.append(state[4])
    arr_l_dot.append(state[5])



    if done:                                        # exit loop if episode␣
 ↪finished
        print("Total steps : ", t)
        print("Total time is : ", env.tau * t)
        break

arr_t = 0.02*np.array(arr_t)                        # Switching time steps for␣
 ↪times in seconds
print("Score: {}".format(score))
env.close()
```

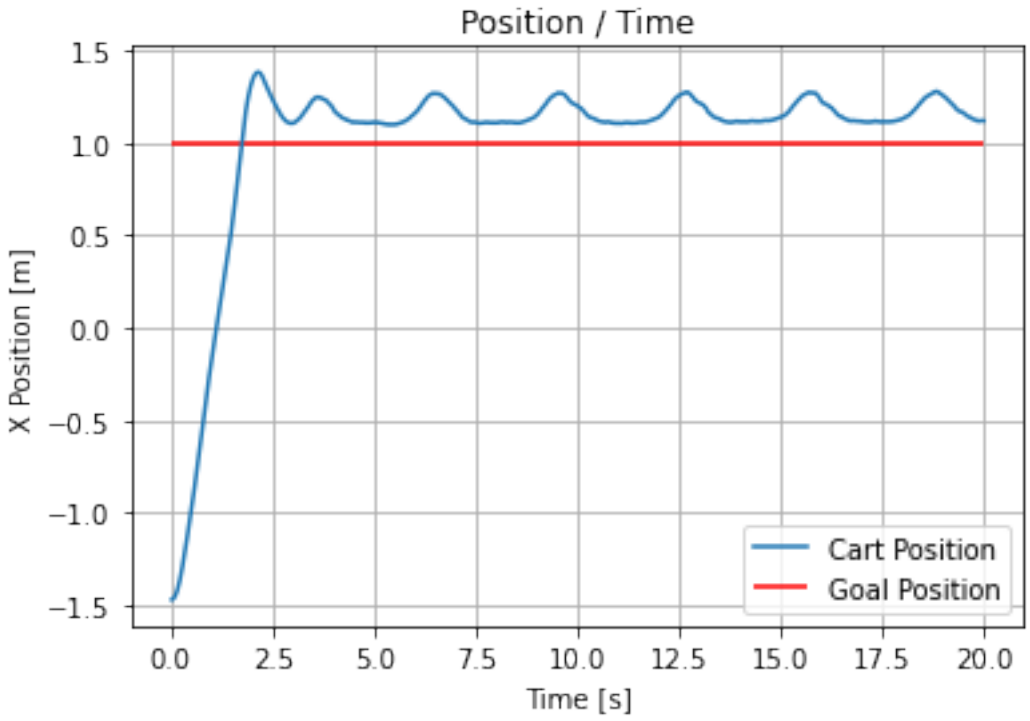
```
Score: 28335.093728179087
```

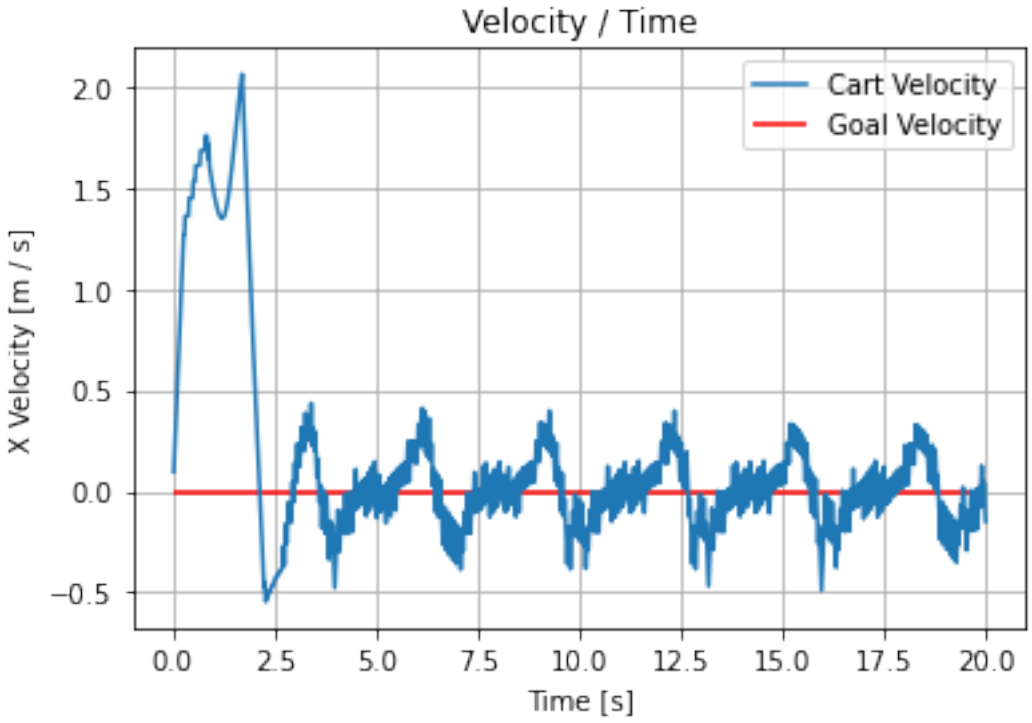
1.3.1 States against time plots

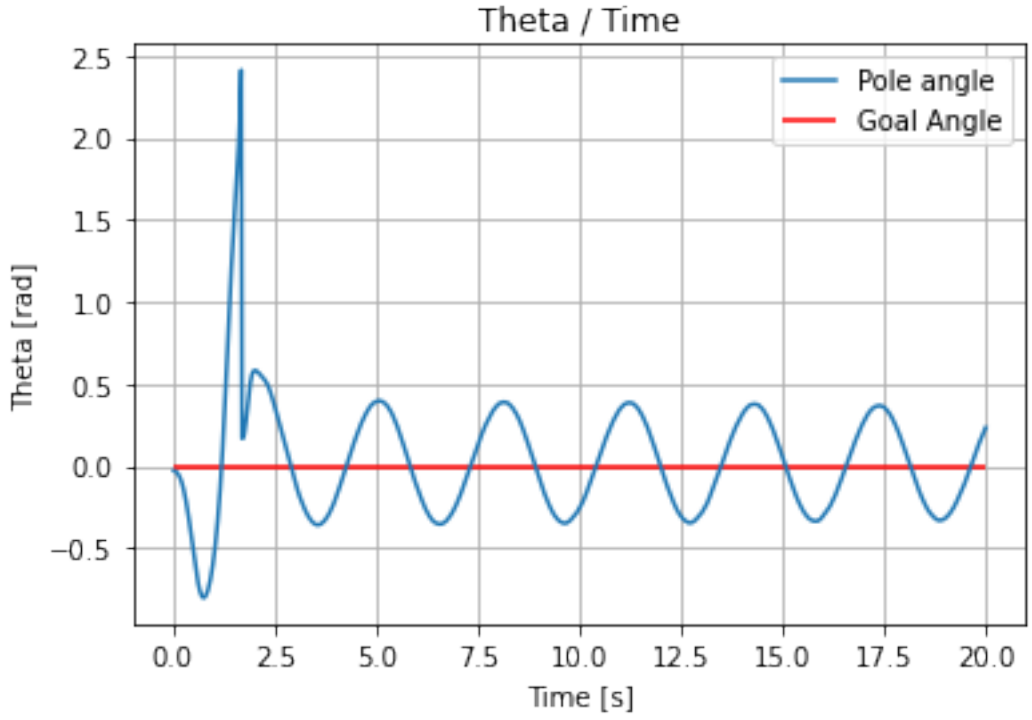
[31]:
```
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(arr_t, arr_x, label='Cart Position')
plt.ylabel('X Position [m]')
plt.xlabel('Time [s]')
plt.title('Position / Time')
plt.hlines(1.0, 0, arr_t[-1], colors='r', linestyles='solid', label='Goal␣
 ↪Position')
plt.legend()
plt.grid()
#plt.savefig('plots/model_3_x.png', dpi = 200)  #UNCOMMENT TO SAVE PLOT
plt.show()

fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(arr_t, arr_x_dot, label = 'Cart Velocity')
plt.ylabel('X Velocity [m / s]')
plt.xlabel('Time [s]')
plt.title('Velocity / Time')
plt.hlines(0.0, 0, arr_t[-1], colors='r', linestyles='solid', label='Goal␣
 ↪Velocity')
plt.legend()
plt.grid()
#plt.savefig('plots/model_3_x_dot.png', dpi = 200)   #UNCOMMENT TO SAVE PLOT
```

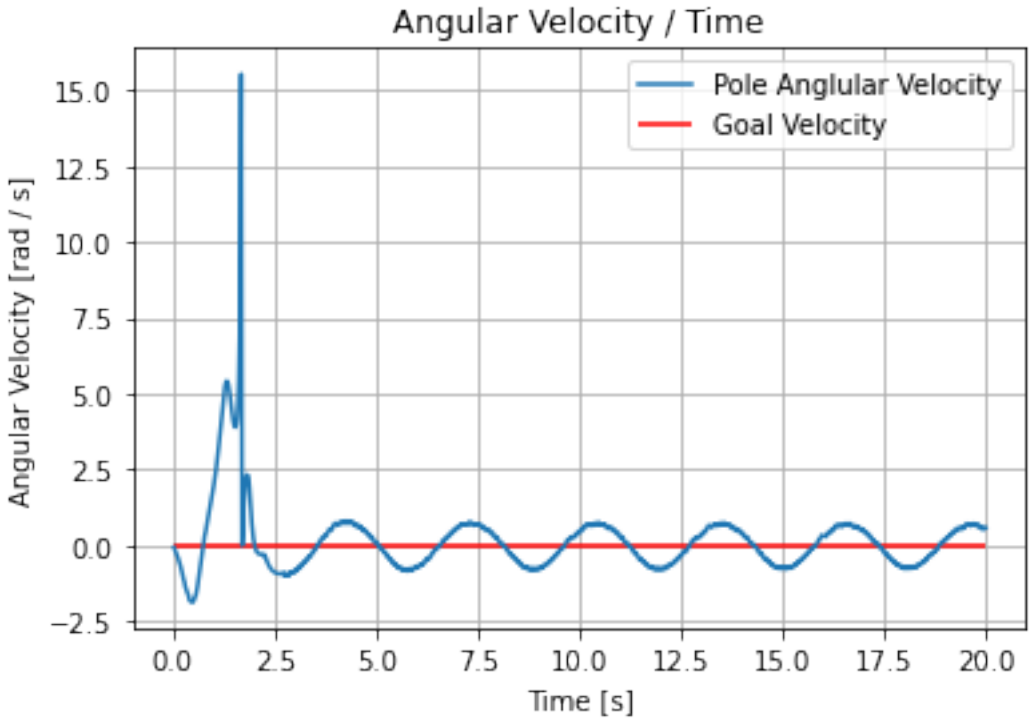
```
plt.show()

fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(arr_t, arr_theta, label = 'Pole angle')
plt.ylabel('Theta [rad]')
plt.xlabel('Time [s]')
plt.title('Theta / Time')
plt.hlines(0, 0, arr_t[-1], colors='r', linestyles='solid', label='Goal Angle')
plt.legend()
plt.grid()
#plt.savefig('plots/model_3_theta.png', dpi = 200)   #UNCOMMENT TO SAVE PLOT
plt.show()

fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(arr_t, arr_theta_dot, label = 'Pole Anglular Velocity')
plt.ylabel('Angular Velocity [rad / s]')
plt.xlabel('Time [s]')
plt.title('Angular Velocity / Time')
plt.hlines(0.0, 0, arr_t[-1], colors='r', linestyles='solid', label='Goal␣
 ↪Velocity')
plt.legend()
plt.grid()
#plt.savefig('plots/model_3_theta_dot.png', dpi = 200)   #UNCOMMENT TO SAVE PLOT
plt.show()

fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(arr_t, arr_l, label = 'Rope Lenght')
plt.ylabel('L [m]')
plt.xlabel('Time [s]')
plt.title('Lenght / Time')
plt.hlines(2.5, 0, arr_t[-1], colors='r', linestyles='solid', label='Goal␣
 ↪Lenght')
plt.legend()
plt.grid()
#plt.savefig('plots/model_3_l.png', dpi = 200)   #UNCOMMENT TO SAVE PLOT
plt.show()

fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(arr_t, arr_l_dot, label = 'Rope Lenght Velocity')
plt.ylabel('L Velocity [m / s]')
plt.xlabel('Time [s]')
plt.title('Rope Lenght Velocity / Time')
```

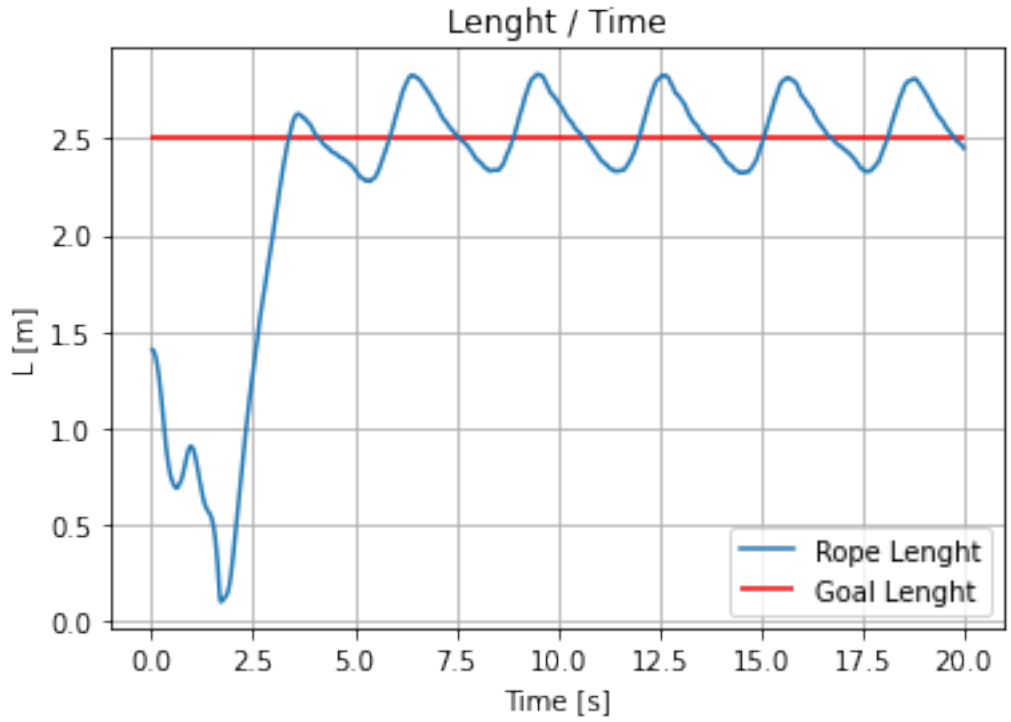
```
plt.hlines(0.0, 0, arr_t[-1], colors='r', linestyles='solid', label='Goal␣
 ↪Velocity')
plt.legend()
plt.grid()
#plt.savefig('plots/model_3_l_dot.png', dpi = 200)   #UNCOMMENT TO SAVE PLOT
plt.show()
```

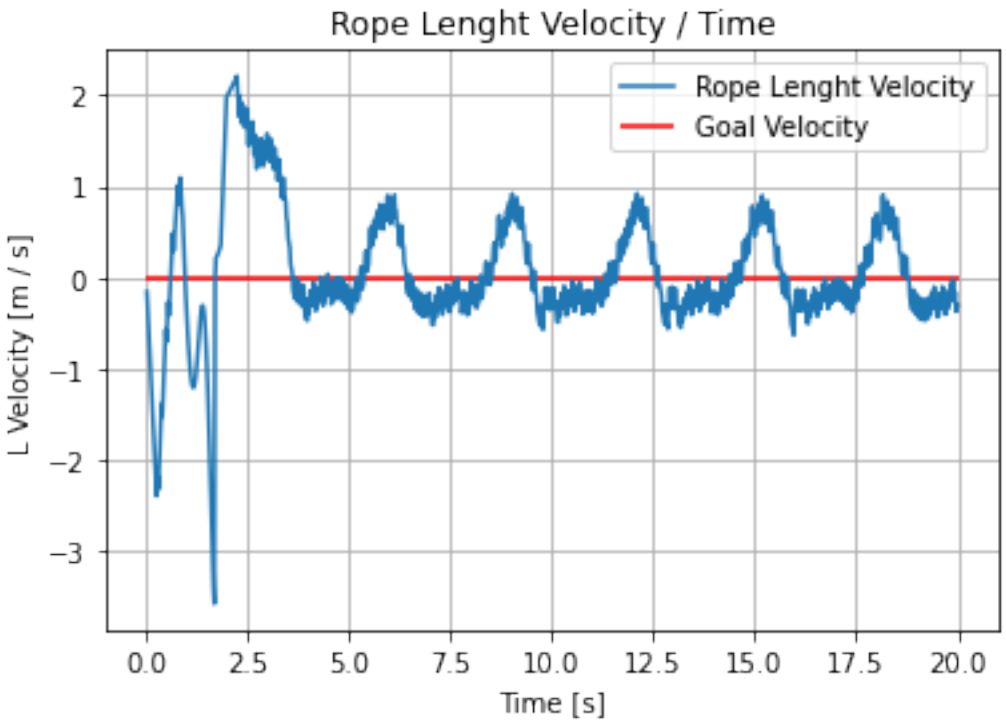



Velocity / Time
Cart Velocity
Goal Velocity
X Velocity [m / s]
2.0
1.5
1.0
0.5
0.0
−0.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]


Theta / Time
Pole angle
Goal Angle
Theta [rad]
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]


Angular Velocity / Time
Pole Anglular Velocity
Goal Velocity
Angular Velocity [rad / s]
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−2.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]


Lenght / Time
L [m]
2.5
2.0
1.5
1.0
0.5
0.0
Rope Lenght
Goal Lenght
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]


Rope Lenght Velocity / Time
Rope Lenght Velocity
Goal Velocity
L Velocity [m / s]
Time [s]
2
1
0
−1
−2
−3
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

[49]:

James003

June 11, 2021

1 Crane V2 model training and simulations

Adapted from https://github.com/kinwo/deeprl-navigation (MIT License Copyright (c) 2018 Henry Chan)

Start Environement and create DQN Agent

[1]:
```
import gym
import numpy as np

env = gym.make('crane-v2') #Load the environement
```

1.1 Agent

The DQN agent can be found below

[2]:
```
import numpy as np
import random
from collections import namedtuple, deque

#from model import QNetwork # UNCOMMENT IF YOU ARE NOT IN A JUPYTER NOTEBOOK

import torch
import torch.nn.functional as F
import torch.optim as optim

BUFFER_SIZE = int(1e5)      # replay buffer size
BATCH_SIZE = 64             # minibatch size
GAMMA = 0.995 #was 0.99     # discount factor
TAU = 1e-3                  # for soft update of target parameters
LR =  5e-4                  # learning rate
UPDATE_EVERY = 4            # how often to update the network

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class Agent():
    """Interacts with and learns from the environment."""

    def __init__(self, state_size, action_size, seed):
```

```
        """Initialize an Agent object.

        Params
        ======
            state_size (int): dimension of each state
            action_size (int): dimension of each action
            seed (int): random seed
        """



        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)

        # Q-Network
        self.qnetwork_local = QNetwork(state_size, action_size, seed).to(device)
        self.qnetwork_target = QNetwork(state_size, action_size, seed).
↪to(device)
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=LR)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)
        # Initialize time step (for updating every UPDATE_EVERY steps)
        self.t_step = 0

    def step(self, state, action, reward, next_state, done):
        # Save experience in replay memory
        self.memory.add(state, action, reward, next_state, done)

        # Learn every UPDATE_EVERY time steps.
        self.t_step = (self.t_step + 1) % UPDATE_EVERY
        if self.t_step == 0:
            # If enough samples are available in memory, get random subset and␣
↪learn
            if len(self.memory) > BATCH_SIZE:
                experiences = self.memory.sample()
                self.learn(experiences, GAMMA)

    def act(self, state, eps=0.):
        """Returns actions for given state as per current policy.

        Params
        ======
            state (array_like): current state
            eps (float): epsilon, for epsilon-greedy action selection
        """
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
```

```
        self.qnetwork_local.eval()
        with torch.no_grad():
            action_values = self.qnetwork_local(state)
        self.qnetwork_local.train()

        # Epsilon-greedy action selection
        if random.random() > eps:
            return np.argmax(action_values.cpu().data.numpy())
        else:
            return random.choice(np.arange(self.action_size))

    def learn(self, experiences, gamma):
        """Update value parameters using given batch of experience tuples.

        Params
        ======
            experiences (Tuple[torch.Variable]): tuple of (s, a, r, s', done)␣
↪tuples
            gamma (float): discount factor
        """
        states, actions, rewards, next_states, dones = experiences

        # Get max predicted Q values (for next states) from target model
        Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].
↪unsqueeze(1)
        # Compute Q targets for current states
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

        # Get expected Q values from local model
        Q_expected = self.qnetwork_local(states).gather(1, actions)

        # Compute loss
        loss = F.mse_loss(Q_expected, Q_targets)

        # Minimize the loss
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # ------------------- update target network ------------------- #
        self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)

    def soft_update(self, local_model, target_model, tau):
        """Soft update model parameters.
        _target = *_local + (1 - )*_target

        Params
```

```
        ======
            local_model (PyTorch model): weights will be copied from
            target_model (PyTorch model): weights will be copied to
            tau (float): interpolation parameter
        """
        for target_param, local_param in zip(target_model.parameters(),␣
↪local_model.parameters()):
            target_param.data.copy_(tau*local_param.data + (1.
↪0-tau)*target_param.data)


class ReplayBuffer:
    """Fixed-size buffer to store experience tuples."""

    def __init__(self, action_size, buffer_size, batch_size, seed):
        """Initialize a ReplayBuffer object.

        Params
        ======
            action_size (int): dimension of each action
            buffer_size (int): maximum size of buffer
            batch_size (int): size of each training batch
            seed (int): random seed
        """

        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state",␣
↪"action", "reward", "next_state", "done"])
        self.seed = random.seed(seed)

    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self):
        """Randomly sample a batch of experiences from memory."""
        experiences = random.sample(self.memory, k=self.batch_size)

        states = torch.from_numpy(np.vstack([e.state for e in experiences if e␣
↪is not None])).float().to(device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if␣
↪e is not None])).long().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if␣
↪e is not None])).float().to(device)
```

```
        next_states = torch.from_numpy(np.vstack([e.next_state for e in␣
↪experiences if e is not None])).float().to(device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is␣
↪not None]).astype(np.uint8)).float().to(device)

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)
```

1.2 Deep Q_Network Model

A 2 linear hidden layer of 64 nodes each is created, with relu activation function.

[3]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

class QNetwork(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=64,␣
↪fc2_units=64):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)



    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

1.3 DQN Agent Training

1.3.1 Create DQN Agen

[4]:
```
import torch
import time
from collections import deque
#from agent import Agent   # UNCOMMENT IF YOU ARE NOT IN A JUPYTER NOTEBOOK
import matplotlib.pyplot as plt
%matplotlib inline

state_size=7        #State size of  environement
action_size=11       #Action size of enviroenemnt
seed=0
agent = Agent(state_size=7, action_size=11, seed=0) #setting the agent's␣
↪parameters
```

Here, we save the model every 100 timesteps, in order to observe how the agent performs over it's training process. The files are located in the working derictory inder the name 'Episode_###.path'.

In order to know when the environement is solved, we compute the moving score (the total rewards per episode) average over the last 100 episodes. If the moving average is over a chosen thershold (target_scores), the model is then saved to 'model_weight_name'.

For the Crane_v0 environement, the target score is 10 000 000, since it is the reward obtained by the agent when finding the flag.

[5]:
```
model_weight_name = 'model.pth'

def dqn(n_episodes=1300, max_t=1500, eps_start=1.0, eps_end=0.01, eps_decay=0.
↪996, target_scores=200000.0):
    """Deep Q-Learning.

    Params
    ======
        n_episodes (int): maximum number of training episodes
        max_t (int): maximum number of timesteps per episode
        eps_start (float): starting value of epsilon, for epsilon-greedy action␣
↪selection
        eps_end (float): minimum value of epsilon
        eps_decay (float): multiplicative factor (per episode) for decreasing␣
↪epsilon
        target_scores (float): average scores aming to achieve, the agent will␣
↪stop training once it reaches this scores
    """
    start = time.time()                # Start time
    scores = []                        # list containing scores from each␣
↪episode
```

```
    scores_window = deque(maxlen=100)  # last 100 scores
    eps = eps_start                    # initialize epsilon

    for i_episode in range(1, n_episodes+1):
        # Reset env and score at the beginning of episode
        env_info = env.reset()                                  # reset the␣
↪environment
        state = env.state                                       # get the current␣
↪state
        score = 0                                               # initialize the␣
↪score

        arr_x = []
        arr_x_dot = []
        arr_theta1 = []
        arr_theta1_dot = []
        arr_theta2 = []
        arr_theta2_dot = []
        arr_t = []

        for t in range(max_t):
            action = agent.act(state, eps)
            env_info = env.step(action)                         # send the action to␣
↪the environment
            next_state = env_info[0]                            # get the next state
            reward = env_info[1]                                # get the reward
            done = env_info[2]                                  # see if episode has␣
↪finished

            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward

            arr_t.append(t)
            arr_x.append(state[0])
            arr_x_dot.append(state[1])
            arr_theta1.append(state[2])
            arr_theta1_dot.append(state[3])
            arr_theta2.append(state[4])
            arr_theta2_dot.append(state[5])
            if score > 100000:
                tarr_x = arr_x
                tarr_x_dot = arr_x_dot
                tarr_theta1 = arr_theta1
                tarr_theta1_dot = arr_theta1_dot
                tarr_theta2 = arr_theta2
                tarr_theta2_dot = arr_theta2_dot
```

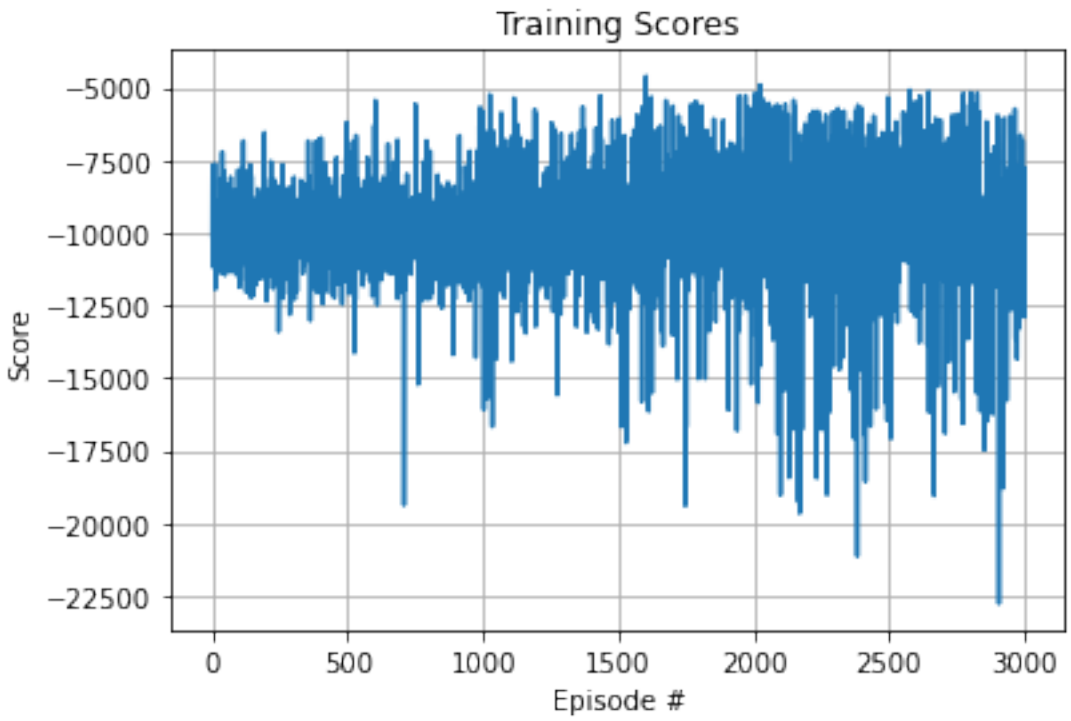
```
                tarr_t = arr_t
                temp_model_weight_name = 'Episode_score{}.pth'.format(score)
                torch.save(agent.qnetwork_local.state_dict(),␣
↪temp_model_weight_name)

            if done:
                print("\n Episode finished after {} timesteps".format(t+1))
                print("\n final state is :", state)
                print("\n Reward is : ", score)
                break

        scores_window.append(score)       # saving the most recent score
        scores.append(score)              # saving the most recent score
        eps = max(eps_end, eps_decay*eps) # decrease of the epsilon value


        print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.
↪mean(scores_window)), end="")

        if i_episode % 100 == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.
↪mean(scores_window)))
            temp_model_weight_name = 'Episode_{}.pth'.format(i_episode)
            torch.save(agent.qnetwork_local.state_dict(),␣
↪temp_model_weight_name)

        if np.mean(scores_window)>=target_scores:
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.
↪2f}'.format(i_episode, np.mean(scores_window)))
            torch.save(agent.qnetwork_local.state_dict(), model_weight_name)
            break

    time_elapsed = time.time() - start
    print("Time Elapse: {:.2f}".format(time_elapsed))

    return scores

scores = dqn(n_episodes=3000, max_t=1500, eps_start=1.0, eps_end=0.01,␣
↪eps_decay=0.997, target_scores=100000.0)
```

```
Episode 100     Average Score: -10042.78
Episode 200     Average Score: -9947.411
Episode 300     Average Score: -10042.79
Episode 400     Average Score: -9914.199
Episode 500     Average Score: -9937.186
Episode 600     Average Score: -9813.782
Episode 700     Average Score: -9769.14
```

```
Episode 800     Average Score: -9987.472
Episode 900     Average Score: -9973.65
Episode 1000    Average Score: -9878.62
Episode 1100    Average Score: -9742.19
Episode 1200    Average Score: -9569.14
Episode 1300    Average Score: -10109.39
Episode 1400    Average Score: -9832.775
Episode 1500    Average Score: -9741.588
Episode 1600    Average Score: -9813.78
Episode 1700    Average Score: -9906.16
Episode 1800    Average Score: -9395.68
Episode 1900    Average Score: -9537.58
Episode 2000    Average Score: -9689.27
Episode 2100    Average Score: -8842.66
Episode 2200    Average Score: -9792.707
Episode 2300    Average Score: -9696.70
Episode 2400    Average Score: -9434.11
Episode 2500    Average Score: -9830.64
Episode 2600    Average Score: -8967.83
Episode 2700    Average Score: -9466.92
Episode 2800    Average Score: -9644.810
Episode 2900    Average Score: -9667.75
Episode 3000    Average Score: -10296.13
Time Elapse: 2993.82
```

1.3.2 Score plot for each episodes

[6]:
```
# plot the scores


fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores)
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.title('Training Scores')
plt.grid()
plt.savefig('plots/model_training.png', dpi = 200)
plt.show()
```



Since the environement is not solved, we choose the best model from where the score plot converges before diverging. After trial and error, it is found to be arround episode 600. We renamed the model "Episode_600.pth' to 'model.pth' and visualised it in the next block.

In order to plot the state against time, we save the data points in local arrays.

[7]:
```
#Choose model
agent.qnetwork_local.state_dict()

#agent.qnetwork_local.load_state_dict(torch.load('model.pth'))

env_info = env.reset()                                  # reset the environment
state = env.state                                       # get the current state
score = 0                                               # initialize the score

#Initialising the state arrays
arr_x = []
arr_x_dot = []
arr_theta1 = []
arr_theta1_dot = []
arr_theta2 = []
arr_theta2_dot = []
arr_t = []
```

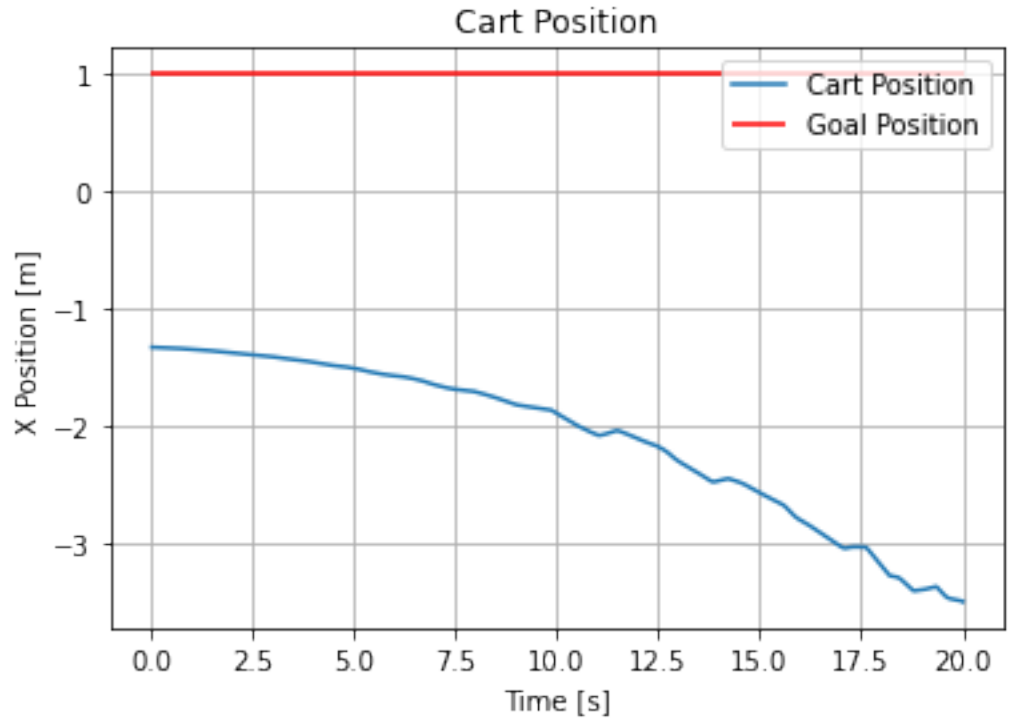
```
t = 0
max_t = 1000                                        # Number of timesteps
for t in range(max_t) :
    #env.render()
    #time.sleep(0.008)
    action = agent.act(state)                       # select an action
    env_info = env.step(action)                     # send the action to the␣
 ↪environment
    next_state = env_info[0]                        # get the next state
    reward = env_info[1]                            # get the reward
    done = env_info[2]                              # see if episode has finished
    score += reward                                 # update the score
    state = next_state                              # roll over the state to␣
 ↪next time step
    t += 1.0

    #Saving states in state matrix for the plot
    arr_t.append(t)
    arr_x.append(state[0])
    arr_x_dot.append(state[1])
    arr_theta1.append(state[2])
    arr_theta1_dot.append(state[3])
    arr_theta2.append(state[4])
    arr_theta2_dot.append(state[5])




    if done:                                        # exit loop if episode␣
 ↪finished
        print("Total steps : ", t)
        print("Total time is : ", env.tau * t)
        break

arr_t = 0.02*np.array(arr_t)                        # Switching time steps for␣
 ↪times in seconds
print("Score: {}".format(score))
env.close()
```

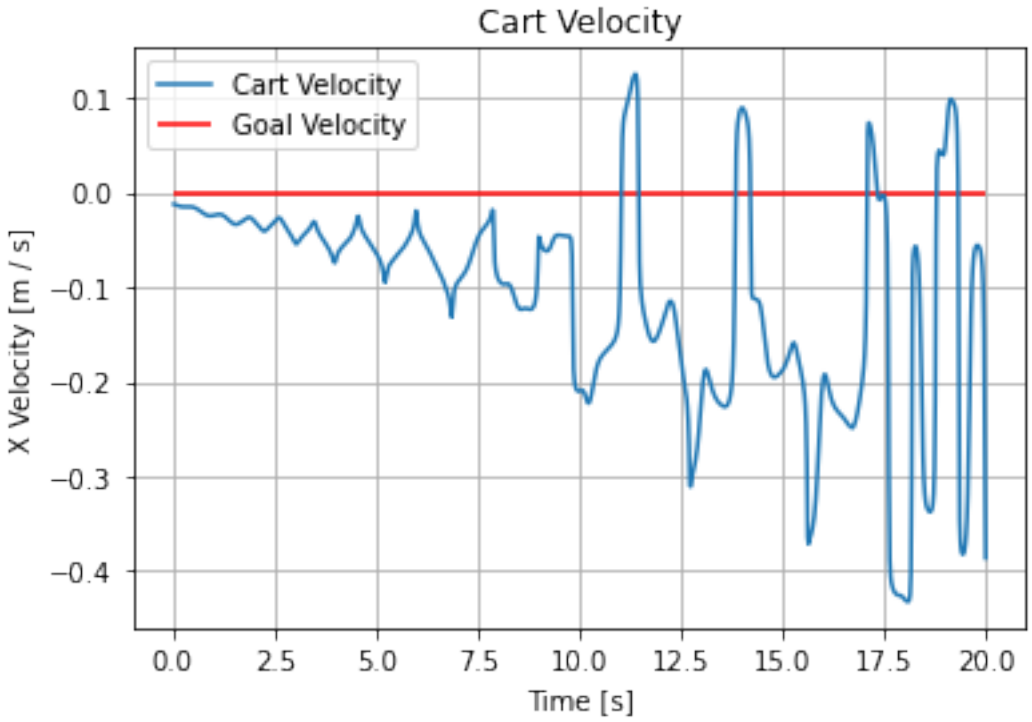
```
Score: -7468.544648733661
```

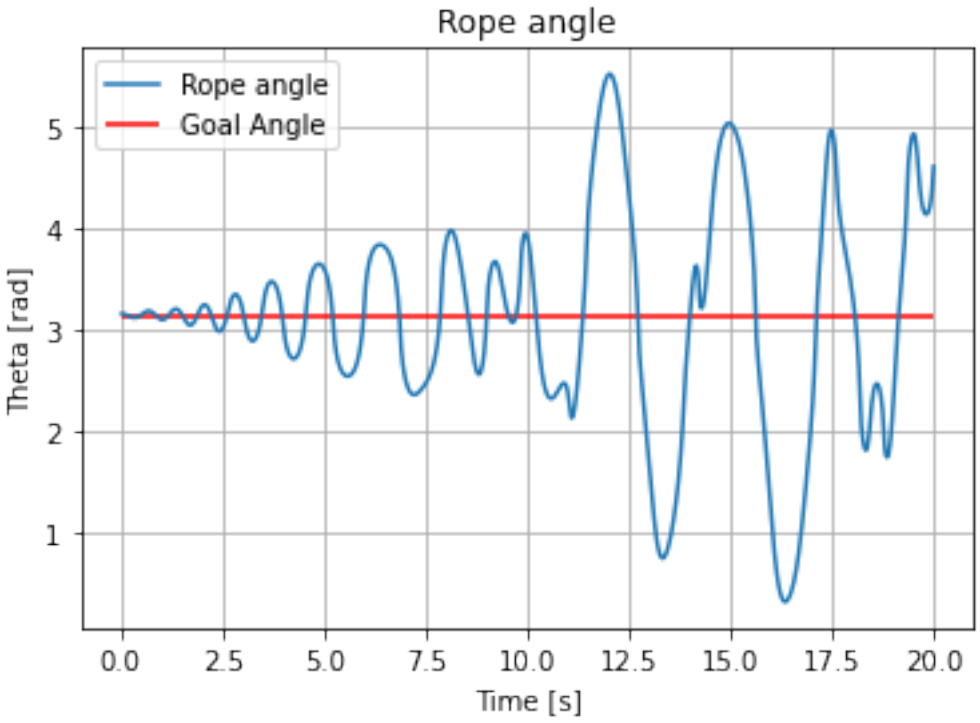
1.3.3 States against time plots

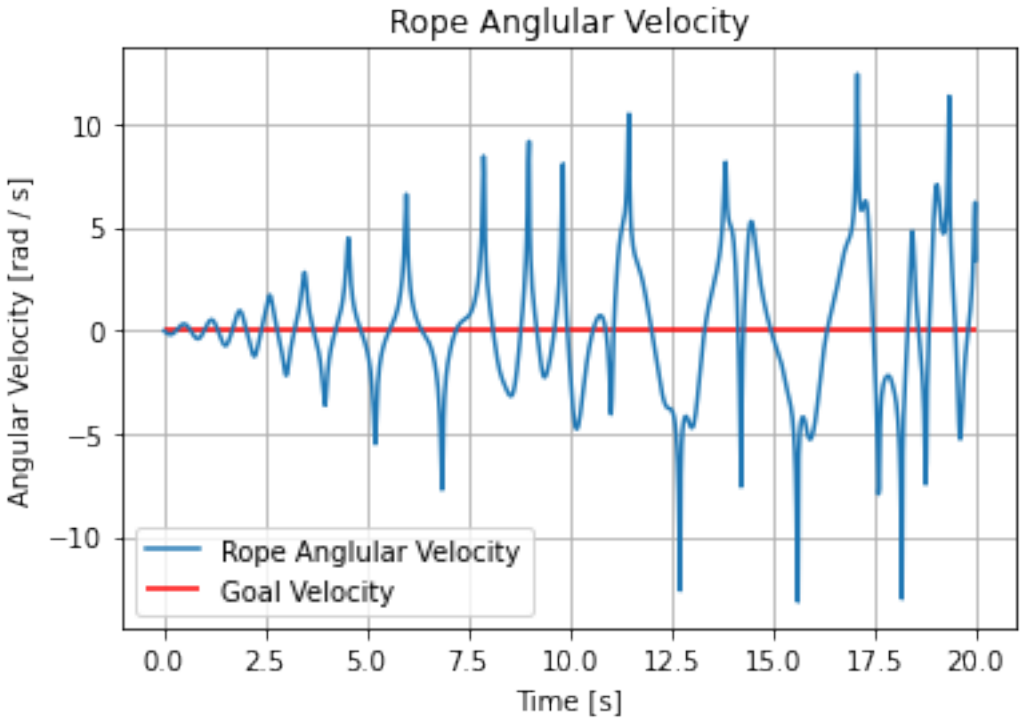
[8]:
```
import math

fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(arr_t, arr_x, label='Cart Position')
plt.ylabel('X Position [m]')
plt.xlabel('Time [s]')
plt.title('Cart Position')
plt.hlines(1.0, 0, arr_t[-1], colors='r', linestyles='solid', label='Goal␣
↪Position')
plt.legend()
plt.grid()
plt.savefig('plots/model_x.png', dpi = 200)  #UNCOMMENT TO SAVE PLOT
plt.show()

fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(arr_t, arr_x_dot, label = 'Cart Velocity')
plt.ylabel('X Velocity [m / s]')
plt.xlabel('Time [s]')
plt.title('Cart Velocity')
plt.hlines(0.0, 0, arr_t[-1], colors='r', linestyles='solid', label='Goal␣
↪Velocity')
plt.legend()
plt.grid()
plt.savefig('plots/model_x_dot.png', dpi = 200)   #UNCOMMENT TO SAVE PLOT
plt.show()

fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(arr_t, arr_theta1, label = 'Rope angle')
plt.ylabel('Theta [rad]')
plt.xlabel('Time [s]')
plt.title('Rope angle')
plt.hlines(math.pi, 0, arr_t[-1], colors='r', linestyles='solid', label='Goal␣
↪Angle')
plt.legend()
plt.grid()
plt.savefig('plots/model_theta1.png', dpi = 200)   #UNCOMMENT TO SAVE PLOT
plt.show()

fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(arr_t, arr_theta1_dot, label = 'Rope Anglular Velocity')
plt.ylabel('Angular Velocity [rad / s]')
```

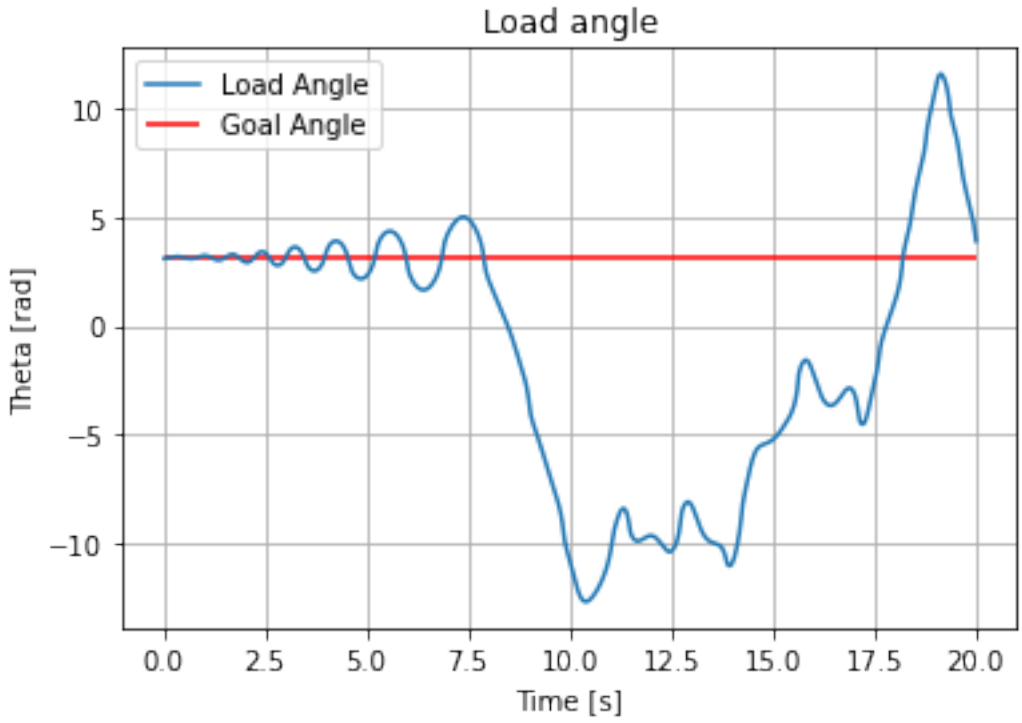
```
plt.xlabel('Time [s]')
plt.title('Rope Anglular Velocity')
plt.hlines(0.0, 0, arr_t[-1], colors='r', linestyles='solid', label='Goal␣
 ↪Velocity')
plt.legend()
plt.grid()
plt.savefig('plots/model_theta1_dot.png', dpi = 200)   #UNCOMMENT TO SAVE PLOT
plt.show()

fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(arr_t, arr_theta2, label = 'Load Angle')
plt.ylabel('Theta [rad]')
plt.xlabel('Time [s]')
plt.title('Load angle')
plt.hlines(math.pi, 0, arr_t[-1], colors='r', linestyles='solid', label='Goal␣
 ↪Angle')
plt.legend()
plt.grid()
plt.savefig('plots/model_theta2.png', dpi = 200)   #UNCOMMENT TO SAVE PLOT
plt.show()

fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(arr_t, arr_theta2_dot, label = 'Load Anglular Velocity')
plt.ylabel('Angular Velocity [rad / s]')
plt.xlabel('Time [s]')
plt.title('Load Anglular Velocity')
plt.hlines(0.0, 0, arr_t[-1], colors='r', linestyles='solid', label='Goal␣
 ↪Velocity')
plt.legend()
plt.grid()
plt.savefig('plots/model_theta2_dot.png', dpi = 200)   #UNCOMMENT TO SAVE PLOT
plt.show()
```

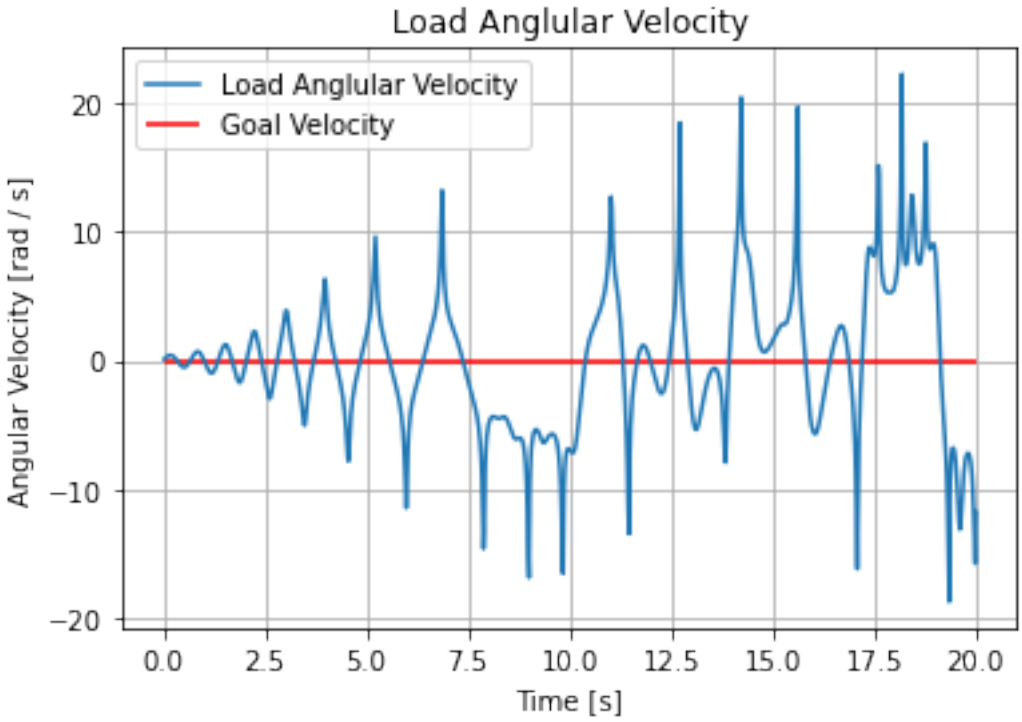

Cart Position
Cart Position
Goal Position
X Position [m]
1
0
−1
−2
−3
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]


Cart Velocity
Cart Velocity
Goal Velocity
X Velocity [m / s]
0.1
0.0
−0.1
−0.2
−0.3
−0.4
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]


Rope angle
Rope angle
Goal Angle
Theta [rad]
1
2
3
4
5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]


Rope Anglular Velocity
Rope Anglular Velocity
Goal Velocity
Angular Velocity [rad / s]
10
5
0
−5
−10
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]


Load angle
Load Angle
Goal Angle
Theta [rad]
10
5
0
−5
−10
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]


Load Anglular Velocity
Load Anglular Velocity
Goal Velocity
Angular Velocity [rad / s]
20
10
0
−10
−20
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Time [s]

[ ]:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Apr  8 17:33:46 2021

@author: oscarjenot

Adapted from Open AI Gym : https://github.com/openai/gym/tree/master/gym/envs/-
classic_control

Reference:
Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., &
Zaremba, W. (2016). Openai gym.
"""

"""
CRANE-v0 Environement
"""

import math
import gym
from gym import spaces, logger
from gym.utils import seeding
import numpy as np

class CraneEnv(gym.Env):


    metadata = {
        'render.modes': ['human', 'rgb_array'],
        'video.frames_per_second': 50
    }

    def __init__(self):
        self.gravity = 9.8
        self.masscart = 1.0
        self.masspole = 0.1
        self.total_mass = (self.masspole + self.masscart)
        self.length = 1  # actually half the pole's length
        self.polemass_length = (self.masspole * self.length)
        self.force_mag = 10.0
        self.tau = 0.02  # seconds between state updates
        self.kinematics_integrator = 'euler'

        #self.goal_position = self.np_random.uniform(low=-2, high=2)
        self.goal_position = 1.0

        # Angle at which to fail the episode
        self.theta_threshold_radians = 12 * 2 * math.pi / 360 + math.pi # 12 degrees + pi
rad
        #min theta_threshold_radian would be : self.theta_threshold_radians - 2 * 12 * 2
* math.pi / 360
        self.x_threshold = 2.4

        # Angle limit set to 12 degrees + 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 + 12 * 2 * math.pi / 360,
                         np.finfo(np.float32).max],
                        dtype=np.float32)
        low = np.array([-self.x_threshold * 2,
                         -np.finfo(np.float32).max,
                         self.theta_threshold_radians - 2 * 12 * 2 * math.pi / 360,
                         -np.finfo(np.float32).max],
                        dtype=np.float32)
```

```
        self.action_space = spaces.Discrete(3)

        self.observation_space = spaces.Box(low, 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 = self.state
        force = (action - 1) * self.force_mag
        costheta = math.cos(theta)
        sintheta = math.sin(theta)

        # For the interested reader:
        # https://coneural.org/florian/papers/05_cart_pole.pdf
        temp = (force + self.polemass_length * theta_dot ** 2 * sintheta) /
self.total_mass
        thetaacc = (self.gravity * sintheta - costheta * temp) / (self.length * (4.0 /
3.0 - self.masspole * costheta ** 2 / self.total_mass))
        xacc = temp - self.polemass_length * thetaacc * costheta / self.total_mass

        if self.kinematics_integrator == 'euler':
            x = x + self.tau * x_dot
            x_dot = x_dot + self.tau * xacc
            theta = theta + self.tau * theta_dot
            theta_dot = theta_dot + self.tau * thetaacc
        else:  # semi-implicit euler
            x_dot = x_dot + self.tau * xacc
            x = x + self.tau * x_dot
            theta_dot = theta_dot + self.tau * thetaacc
            theta = theta + self.tau * theta_dot

        self.state = (x, x_dot, theta, theta_dot)

        #REWARDS

        reward = -math.log(abs(self.goal_position - x)) - 1


        if abs(self.goal_position - x) < 0.2 :
            reward = reward - math.log((abs(theta_dot) + abs(sintheta))) * 10 + 1

        if x > self.goal_position - 0.05 and x < self.goal_position + 0.05 and x_dot > -
0.05 and x_dot <   0.05 and theta_dot > - 0.03  and theta_dot <   0.03 and theta >
math.pi  - 0.03 and theta <  math.pi + 0.03 :
            reward = reward + 100000.0

        done = bool(  x_dot > -0.05 and x_dot < 0.05 and x > self.goal_position - 0.05 and
x < self.goal_position + 0.05 and theta_dot > -0.03 and theta_dot < 0.03  and theta >
math.pi - 0.03 and theta < math.pi + 0.03 )
```

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

    def reset(self):
        #self.state = np.array([self.np_random.uniform(low=-2, high=-1),
self.np_random.uniform(low=-0.05, high=0.05), self.np_random.uniform(low=math.pi-0.05,
high=math.pi+0.05), self.np_random.uniform(low=-0.05, high=0.05) ])
        #Try with initial conditions thet model was not trained on
        self.state = np.array([self.np_random.uniform(low=2, high=4),
self.np_random.uniform(low=-0.05, high=0.05), self.np_random.uniform(low=math.pi-0.05,
high=math.pi+0.05), self.np_random.uniform(low=-0.05, high=0.05) ])
        return self.state

    def render(self, mode='human'):
        screen_width = 600
        screen_height = 400

        world_width = self.x_threshold * 2
        scale = screen_width/world_width
        carty = 300  # TOP OF CART
        polewidth = 10.0
        polelen = scale * (2 * self.length)
        cartwidth = 50.0
        cartheight = 30.0

        if self.viewer is None:
            from gym.envs.classic_control import rendering
            self.viewer = rendering.Viewer(screen_width, screen_height)

            l, r, t, b = -cartwidth / 2, cartwidth / 2, cartheight / 2, -cartheight / 2
            axleoffset = cartheight / 4.0
            cart = rendering.FilledPolygon([(l, b), (l, t), (r, t), (r, b)])
            self.carttrans = rendering.Transform()
            cart.add_attr(self.carttrans)
            self.viewer.add_geom(cart)

            l, r, t, b = -polewidth / 2, polewidth / 2, polelen - polewidth / 2, -
polewidth / 2
            pole = rendering.FilledPolygon([(l, b), (l, t), (r, t), (r, b)])
            pole.set_color(.8, .6, .4)
            self.poletrans = rendering.Transform(translation=(0, axleoffset))
            pole.add_attr(self.poletrans)
            pole.add_attr(self.carttrans)
            self.viewer.add_geom(pole)

            self.axle = rendering.make_circle(polewidth/2)
            self.axle.add_attr(self.poletrans)
            self.axle.add_attr(self.carttrans)
            self.axle.set_color(.5, .5, .8)
            self.viewer.add_geom(self.axle)

            self.track = rendering.Line((0, carty), (screen_width, carty))
            self.track.set_color(0, 0, 0)
            self.viewer.add_geom(self.track)

            flagx = (self.goal_position) * scale + screen_width / 2.0
            flagy1 = (carty - cartheight - polelen )
            flagy2 = flagy1 + 50
            flagpole = rendering.Line((flagx, flagy1), (flagx, flagy2))
            self.viewer.add_geom(flagpole)

            flag = rendering.FilledPolygon(
                [(flagx, flagy2), (flagx, flagy2 - 10), (flagx + 25, flagy2 - 5)]
            )
            flag.set_color(.8, .8, 0)
            self.viewer.add_geom(flag)
```

```
            self._pole_geom = pole

        if self.state is None:
            return None

        # Edit the pole polygon vertex
        pole = self._pole_geom
        l, r, t, b = -polewidth / 2, polewidth / 2, polelen - polewidth / 2, -polewidth /
2
        pole.v = [(l, b), (l, t), (r, t), (r, b)]

        x = self.state
        cartx = x[0] * scale + screen_width / 2.0  # MIDDLE OF CART
        self.carttrans.set_translation(cartx, carty)
        self.poletrans.set_rotation(-x[2])

        return self.viewer.render(return_rgb_array=mode == 'rgb_array')


    def close(self):
        if self.viewer:
            self.viewer.close()
            self.viewer = None
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Mai 10  8 13:32:43 2021

@author: oscarjenot
Adapted from Open AI Gym : https://github.com/openai/gym/tree/master/gym/envs/-
classic_control

Reference:
Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., &
Zaremba, W. (2016). Openai gym.
"""

"""
CRANE-v1 Environement
"""

import math
import gym
from gym import spaces, logger
from gym.utils import seeding
import numpy as np

class CraneEnv1(gym.Env):

    """
    Description
        An mass is attached to an inextensible rope attached by an un-actuated
        joint to a cart, which moves along a frictionless track.
        The pendulum starts downward, and the goal is to stabilysing
        it at an other x-location of the cart (flag).

    Observation
        Type: Box(4)
        Num     Observation               Min                     Max
        0       Cart Position             -4.8                    4.8
        1       Cart Velocity             -Inf                    Inf
        2       Pole Angle                -2pi rad (-360 deg)    2pi rad (360 deg)
        3       Pole Angular Velocity     -Inf                    Inf
        4       Rope Lenght               0                       Inf
        5       Rope Velocity             -Inf                    Inf
        6       x goal position           -2                      2
        7       y goal position           0                       2

    Coordinates : Positive angle is trigonometric angle, positive x position is
    to the right, pole lenght is positive from the cart to the mass. Pole lowering
    force is positive from mass to cart.

    Actions:
        Type: Box(2)
        Num   Action
        0     Driving force. Type: Discrete(3): accelerate to the Left (0),
              no acceleration (1), accelerate to the right (2).
        1     Pole lowering and hoisting force. Type: Discrete(3): hoist Pple (0),
              no force (1), lower pole (2).

    Reward:
        Reward of 0 is awarded if the agent reached the flag with a small tolerance.
        Tolerance on pole lenght : [y_flag_dist -0.05 , y_flag_dist +0.05] Rad.
        Tolerance on other Observations : [-0.05 , 0.05].
        Reward of -1 is awarded if the position of the agent is not on the flag.

    Starting State:
        Cart position is assigned a uniform random value in [-2 , 0].
        Cart velocity, pole angle and angular velocity are assigned a uniform random
```

```
        value in [-0.05 , 0.05].
        Pole lenght is assigned a uniform random value in [0 , 2].

    Episode Termination:
        Pole Angle is not longer in [-12, +12] degrees.
        Cart Position is more than 2.4 (center of the cart reaches the edge of
        the display).
        Episode length is greater than 200.
        Solved Requirements:
        The agent reached the flag with a small tolerance (position to be set up).

    """

    metadata = {
        'render.modes': ['human', 'rgb_array'],
        'video.frames_per_second': 50
    }


#_____________________________________________________________________________________

    def __init__(self):

        #Constants
        self.gravity = 9.81
        self.masscart = 1.0
        self.masspoint = 0.1

        self.tau = 0.02  # seconds between state updates
        self.force_cart = 5.0
        self.force_rope = 1.0

        self.kinematics_integrator = 'euler'
        #self.kinematics_integrator = 'semi-implicit euler'


        self.cart_height = 2.5

        #thresholds
        self.theta_threshold_radians = 12 * 2 * math.pi / 360 #12 degrees
        self.x_threshold = 2.4


        #Action space
        """
        UNCOMMENT FOR CONTINUOUS ACTION SPACE

        self.min_action = np.array([-1.0, -1.0]) # min cart force, min rope force
        self.max_action = np.array([1.0, 1.0])   # max cart force, max rope force

        self.action_space = spaces.Box(low = self.min_action,
                                       high = self.max_action,
                                       dtype=np.float.32)  # cart, rope
        """

        #COMMENT FOR USE OF CONTINUOUS ACTION SPACE
        self.action_space = spaces.Discrete(9)



        # Observation space and limits

        self.x_min = -(self.x_threshold * 2)
        self.x_max = self.x_threshold * 2
        self.theta_min = -(2*self.theta_threshold_radians)
        self.theta_max = 2*self.theta_threshold_radians
```

```
        self.l_min = 0
        self.l_max = np.finfo(np.float32).max
        self.velocity_min = -np.finfo(np.float32).max
        self.velocity_max = np.finfo(np.float32).max
        self.x_goal_min = -2
        self.x_goal_max =  2
        self.y_goal_min =  0
        self.y_goal_max =  2

        self.min_observation = np.array([self.x_min, self.velocity_min,
                                         self.theta_min, self.velocity_min,
                                         self.l_min, self.velocity_min,
                                         self.x_goal_min, self.y_goal_min],
                                        dtype=np.float32)

        self.max_observation = np.array([self.x_max, self.velocity_max,
                                         self.theta_max, self.velocity_max,
                                         self.l_max, self.velocity_max,
                                         self.x_goal_max, self.y_goal_max],
                                        dtype=np.float32)

        self.observation_space = spaces.Box(low = self.min_observation,
                                            high = self.max_observation,
                                            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, l, l_dot, x_goal, y_goal = self.state


        """
        UNCOMMENT FOR CONTINUOUS ACTION SPACE

        force_cart = (action[0]) * self.force_cart
        force_rope = (action[1]) * self.force_rope - self.masspoint * self.gravity
#removing masspoint * gravity brings the mass to an equilibrium when thetha acc is zero
        """

        """
        FOR DISCRETE ACTION SPACE, THIS IS THE FORCE / ACTION MATRIX :

            [  ][-1][ 0][ 1] force_cart
            [-1][ 0][ 1][ 2]
            [ 0][ 3][ 4][ 5]
            [ 1][ 6][ 7][ 8]
           force_
           rope

        """
```

```
        #Discrete action space / force mapping

        temp = np.array([[0, 1, 2],
                         [3, 4, 5],
                         [6, 7, 8]])

        if action in temp[:,0] :
            force_cart = -1 * self.force_cart
        if action in temp[:,1] :
            force_cart =  0 * self.force_cart
        if action in temp[:,2] :
            force_cart =  1 * self.force_cart

        if action in temp[0,:] :
            force_rope = -1 * self.force_rope - self.masspoint * self.gravity #removing
masspoint * gravity brings the mass to an equilibrium when thetha acc is zero
        if action in temp[1,:] :
            force_rope =  0 * self.force_rope - self.masspoint * self.gravity #removing
masspoint * gravity brings the mass to an equilibrium when thetha acc is zero
        if action in temp[2,:] :
            force_rope =  1 * self.force_rope - self.masspoint * self.gravity #removing
masspoint * gravity brings the mass to an equilibrium when thetha acc is zero


        # For the interested reader the dynamics of the system can be found here:
        # https://www.researchgate.net/publication/-
261295749_Tracking_Control_for_an_Underactuated_Two-Dimensional_Overhead_Crane

        costheta = math.cos(theta)
        sintheta = math.sin(theta)

        temp = ( force_cart - force_rope * sintheta )


        xacc = temp / self.masscart
        thetaacc = -( temp * costheta / self.masscart + 2 * l_dot * theta_dot +
self.gravity * sintheta ) / l
        lacc =  (force_rope / self.masspoint + (theta_dot)**(2) * l + self.gravity *
costheta  - temp * sintheta / self.masscart)


        if self.kinematics_integrator == 'euler':
            x = x + self.tau * x_dot
            x_dot = x_dot + self.tau * xacc
            theta = theta + self.tau * theta_dot
            theta_dot = theta_dot + self.tau * thetaacc
            l = l + self.tau * l_dot
            l_dot = l_dot + self.tau * lacc

        else:  # semi-implicit euler
            x_dot = x_dot + self.tau * xacc
            x = x + self.tau * x_dot
            theta_dot = theta_dot + self.tau * thetaacc
            theta = theta + self.tau * theta_dot
            l_dot = l_dot + self.tau * lacc
            l = l + self.tau * l_dot



        costheta = math.cos(theta)
        sintheta = math.sin(theta)

        self.x_goal_position = x_goal
        self.y_goal_position = y_goal
```

```
        # REWARDS
        reward = -1

        reward = reward -np.log(abs(1.0 - x)) * 2


        if abs(1 - x) < 0.5 :

            reward = reward + 3
            if l > 3 :
                reward = reward  -l_dot / 6
            if l < 2 :
                reward = reward + l_dot / 6


            reward = reward - np.log((abs(theta_dot) + 3*abs(sintheta))) * 15 + 5

            reward = reward - np.log(abs(l - abs(self.cart_height))) * 8 + 5



            #if ((abs(theta_dot) + abs(sintheta))) < 0.1:
                #reward = reward - math.log(abs(l - abs(self.cart_height))) * 7 + 50
                #reward = reward - (abs(l - abs(self.cart_height))) + 30

            #reward = reward - math.log(abs(l - abs(self.cart_height))) * 10
+10
            #reward = reward - abs(l - abs(self.cart_height))


        '''
        #PRECISE GOAL

        if (x > self.x_goal_position - 0.05     and x < self.x_goal_position + 0.05
            and x_dot > - 0.05          and x_dot <   0.05
            and theta_dot > - 0.05      and theta_dot <   0.05
            and sintheta > - 0.05          and sintheta <   0.05
            and l_dot > - 0.05          and l_dot <   0.05
            and l > (self.cart_height - self.y_goal_position) - 0.05  and l <
(self.cart_height - self.y_goal_position) + 0.05
            ):
            reward = reward + 200000.0


        done = bool(
            x > self.x_goal_position - 0.05     and x < self.x_goal_position + 0.05
            and x_dot > - 0.05          and x_dot <   0.05
            and theta_dot > - 0.05      and theta_dot <   0.05
            and sintheta > - 0.05          and sintheta <   0.05
            and l_dot > - 0.05          and l_dot <   0.05
            and l > (self.cart_height - self.y_goal_position) - 0.05
            and l < (self.cart_height - self.y_goal_position) + 0.05
        )
        '''

        if (x > self.x_goal_position - 0.15     and x < self.x_goal_position + 0.15
            and x_dot > - 0.15          and x_dot <   0.15
            and theta_dot > - 0.15      and theta_dot <   0.15
            and sintheta > - 0.15          and sintheta <   0.15
            and l_dot > - 0.15          and l_dot <   0.15
            and l > 2.5 - 0.15  and l < 2.5 + 0.15
            ):
            reward = reward + 10000000.0

        if (x > self.x_goal_position - 0.2     and x < self.x_goal_position + 0.2
            and x_dot > - 0.2          and x_dot <   0.2
```

```
        and theta_dot > - 0.2     and theta_dot <   0.2
        and sintheta > - 0.2        and sintheta <   0.2
        #and l_dot > - 0.1         and l_dot <   0.1
        and l > 2.5 - 0.3  and l < 2.5 + 0.3
        ):
        reward = reward + 10000.0
        #print('First l_position Reward !')


    done = bool(
        x > self.x_goal_position - 0.15     and x < self.x_goal_position + 0.15
        and x_dot > - 0.15         and x_dot <   0.15
        and theta_dot > - 0.15     and theta_dot <   0.15
        and sintheta > - 0.15      and sintheta <   0.15
        and l_dot > - 0.15         and l_dot <   0.15
        and l > 2.5 - 0.15         and l < 2.5+ 0.15
    )


    #Limits on observation values

    if theta_dot > 10**20 :
        #print('!!! Theta dimension problem !!!')
        #print(self.state)
        theta = self.np_random.uniform(low=-math.pi, high=math.pi)
        theta_dot = self.np_random.uniform(low=-0.1, high=0.1)
        thetaacc = 0
        x_dot = 0
        xacc = 0
        #reward = reward - 2000
        #CraneEnv1.reset(self)
        #x, x_dot, theta, theta_dot, l, l_dot, x_goal, y_goal = self.state
        #done = True

    if  l_dot  > 10**20 :
        #print('!!! Lenght dimension problem !!!')
        #print(self.state)
        #reward = reward - 2000
        theta = 1.0
        theta_dot = self.np_random.uniform(low=-math.pi, high=math.pi)
        thetaacc = 0
        x_dot = 0
        xacc = 0
        l = 1.0
        l_dot = 0
        lacc = 0
        #CraneEnv1.reset(self)
        #x, x_dot, theta, theta_dot, l, l_dot, x_goal, y_goal = self.state
        #done = True

    if l < 0.1 :
        #print('!!! Lenght Can not be negatif !!!')
        #reward = reward - 10
        #print(self.state)
        l = 0.1
        l_dot = 0
        lacc = 0
        theta = self.np_random.uniform(low=-0.2, high=0.2)
        theta_dot = 0
        thetaacc = 0
        #done = True

    '''
    if l < 0.1 :
        #print('!!! Lenght Can not be negatif !!!')
        #reward = reward - 10
```

```
            #print(self.state)
            l = 0.2
            l_dot = 0
            lacc = 0
            theta = self.np_random.uniform(low=-0.2, high=0.2)
            theta_dot = 0
            thetaacc = 0
            done = True
        '''

        #if x_dot > 10**20 :
            #print('!!! x_position dimension problem !!!')
            #print(self.state)
            #reward = reward - 2000
            #CraneEnv1.reset(self)
            #x, x_dot, theta, theta_dot, l, l_dot, x_goal, y_goal = self.state
            #done = True


        self.state = (x, x_dot, theta, theta_dot, l, l_dot, x_goal, y_goal)


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



#_____________________________________________________________________________________________________

    def reset(self):
        self.state = np.array([self.np_random.uniform(low=-2, high=-1),
self.np_random.uniform(low=-0.05, high=0.05), # x and x_dot
                               self.np_random.uniform(low=-0.05, high=0.05),
self.np_random.uniform(low=-0.05, high=0.05), # theta and thetha_dot
                               self.np_random.uniform(low=1, high=2),
self.np_random.uniform(low=-0.05, high=0.05), # l and l_dot
                               self.np_random.uniform(low=1.0, high=1.0),
self.np_random.uniform(low=0.0, high=0.0)])  #x_goal and y_goal

        self.x_goal_position = self.state[6]
        self.y_goal_position = self.state[7]

        self.steps_beyond_done = None
        return self.state


#_____________________________________________________________________________________________________

    def render(self, mode='human'):
        screen_width = 600
        screen_height = 400

        world_width = self.x_threshold * 2
        scale = screen_width/world_width

        carty = self.cart_height * scale  + screen_height/20 # screen_height/20 is the
vertical offset
        polewidth = 2.0
        polelen = scale * (self.state[4])
        cartwidth = 50.0
        cartheight = 30.0



        if self.viewer is None:
            from gym.envs.classic_control import rendering
```

```
            self.viewer = rendering.Viewer(screen_width, screen_height)

            l, r, t, b = -cartwidth / 2, cartwidth / 2, cartheight / 2, -cartheight / 2
            cart = rendering.FilledPolygon([(l, b), (l, t), (r, t), (r, b)])
            self.carttrans = rendering.Transform()
            cart.add_attr(self.carttrans)
            self.viewer.add_geom(cart)

            l, r, t, b = -polewidth / 2, polewidth / 2, polelen - polewidth / 2, -
polewidth / 2
            pole = rendering.FilledPolygon([(l, b), (l, t), (r, t), (r, b)])
            #pole.set_color(.8, .6, .4)
            pole.set_color(.5, .5, .5)
            self.poletrans = rendering.Transform()
            pole.add_attr(self.poletrans)
            pole.add_attr(self.carttrans)
            self.viewer.add_geom(pole)

            self.axle = rendering.make_circle(polewidth * 2)
            self.axle.add_attr(self.carttrans)
            self.axle.set_color(.5, .5, .8)
            self.viewer.add_geom(self.axle)

            self.mass = rendering.make_circle(self.masspoint * 100)
            self.mass.set_color(0, 0, 0)
            self.masstrans = rendering.Transform()
            self.mass.add_attr(self.masstrans)
            self.viewer.add_geom(self.mass)

            self.track = rendering.Line((0, carty), (screen_width, carty))
            self.track.set_color(0.8, 0, 0)
            self.viewer.add_geom(self.track)

            flagx = (self.x_goal_position) * scale + screen_width / 2.0
            flagy1 = (scale * self.y_goal_position + screen_height/20 ) # screen_height/-
20 is the vertical offset
            flagy2 = flagy1 + 50
            flagpole = rendering.Line((flagx, flagy1), (flagx, flagy2))
            self.viewer.add_geom(flagpole)

            flag = rendering.FilledPolygon(
                [(flagx, flagy2), (flagx, flagy2 - 10), (flagx + 25, flagy2 - 5)]
            )
            flag.set_color(.8, .8, 0)
            self.viewer.add_geom(flag)

            self.ground = rendering.Line((0, 0 + screen_height/20), (screen_width, 0 +
screen_height/20))
            self.ground.set_color(0, 0, 0)
            self.viewer.add_geom(self.ground)

            self._pole_geom = pole

        if self.state is None:
            return None

        # Edit the pole polygon vertex
        pole = self._pole_geom
        l, r, t, b = -polewidth / 2, polewidth / 2, polelen - polewidth / 2, -polewidth /
2
        pole.v = [(l, b), (l, t), (r, t), (r, b)]

        x = self.state
        cartx = x[0] * scale + screen_width / 2.0  # MIDDLE OF CART
        self.carttrans.set_translation(cartx, carty)
        self.poletrans.set_rotation(x[2]+ math.pi)
```

```
        massx = cartx + (math.sin(x[2]) * x[4])*scale
        massy = carty - (math.cos(x[2]) * x[4])*scale
        self.masstrans.set_translation(massx, massy)



        return self.viewer.render(return_rgb_array=mode == 'rgb_array')

#_____________________________________________________________________________

    def close(self):
        if self.viewer:
            self.viewer.close()
            self.viewer = None
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Jun  9 18:34:11 2021

@author: oscarjenot

Adapted from Open AI Gym : https://github.com/openai/gym/tree/master/gym/envs/-
classic_control

Reference:
Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., &
Zaremba, W. (2016). Openai gym.
"""


"""
CRANE-v2 Environement
"""

import math
import gym
from gym import spaces, logger
from gym.utils import seeding
import numpy as np
from numpy.linalg import inv

class CraneEnv2(gym.Env):



    metadata = {
        'render.modes': ['human', 'rgb_array'],
        'video.frames_per_second': 50
    }


#_____________________________________________________________________________________

    def __init__(self):

        #Constants
        self.gravity = 9.81
        self.m0      = 100.0         # mass cart
        self.m1      = 0.1           # mass rope
        self.m2      = 5.0           # mass load
        self.L1      = 1.0           # lenght rope
        #L2 = state[6]               # lenght load (in observation space)

        #Integration for next states
        self.tau = 0.02                 # seconds between state updates
        self.kinematics_integrator = 'euler'
        #self.kinematics_integrator = 'semi-implicit euler'

        #Target
        self.x_goal_position = 1

        #thresholds
        self.x_threshold = 2.4

        #Action space
        """
        UNCOMMENT FOR CONTINUOUS ACTION SPACE

        self.min_action = -5 # min cart force, min rope force
        self.max_action = 5  # max cart force, max rope force
```

```
        self.action_space = spaces.Box(low = self.min_action,
                                       high = self.max_action,
                                       dtype=np.float.32)  # cart
        """

        #COMMENT FOR USE OF CONTINUOUS ACTION SPACE
        self.action_space = spaces.Discrete(11)
        # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] is action space



        # Observation space and limits

        self.x_min = -(self.x_threshold * 2)
        self.x_max = self.x_threshold * 2
        self.theta_min = -np.finfo(np.float32).max
        self.theta_max = np.finfo(np.float32).max
        self.velocity_min = -np.finfo(np.float32).max
        self.velocity_max = np.finfo(np.float32).max
        self.L2_min = 0
        self.L2_max = 1


        self.min_observation = np.array([self.x_min, self.velocity_min,
                                         self.theta_min, self.velocity_min,
                                         self.theta_min,
self.velocity_min,
                                         self.L2_min ],
                                        dtype=np.float32)

        self.max_observation = np.array([self.x_max, self.velocity_max,
                                         self.theta_max, self.velocity_max,
                                         self.theta_max, self.velocity_max,
                                         self.L2_max],
                                        dtype=np.float32)

        self.observation_space = spaces.Box(low = self.min_observation,
                                            high = self.max_observation,
                                            dtype=np.float32)
        # [x, x_dot, theta1; theta1_dot, theta2, theta2_dot, L2] is observation space

        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, theta1, theta1_dot, theta2, theta2_dot, L2  = self.state


        """
        UNCOMMENT FOR CONTINUOUS ACTION SPACE

        force_cart = (action)
```

```
        """
        force = action - 5
        # [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5] is force action space

        # For the interested reader the dynamics of the system can be found here:
        # https://www.researchgate.net/publication/-
250107215_Optimal_Control_of_a_Double_Inverted_Pendulum_on_a_Cart

        # Constants for matrix form of Lagrange Equations
        m0 = self.m0
        m1 = self.m1
        m2 = self.m2
        L1 = self.L1
        L2 = L2

        d1 = m0 + m1 + m2
        d2 = (0.5 * m1 + m2 ) * L1
        d3 = 0.5 * m2 * L2
        d4 = (1/3 * m1 + m2) * L1**2
        d5 = 0.5 * m2 * L1 * L2
        d6 = 1/3 * m2 * L2**2
        f1 = (0.5 * m1 + m2) * L1 * self.gravity
        f2 = 0.5 * m2 * L2 * self.gravity

        costheta1 = math.cos(theta1)
        costheta2 = math.cos(theta2)
        costheta12 = math.cos(theta1 - theta2)
        sintheta1 = math.sin(theta1)
        sintheta2 = math.sin(theta2)
        sintheta12 = math.sin(theta1 - theta2)

        #Matrix for Lagrange Equations : D(q) * q_acc + C(q, q_dot) * q_dot + G(q) = H *
force

        D =      np.array([[d1           , d2*costheta1  , d3*costheta2   ],
                           [d2*costheta1 , d4            , d5*costheta12  ],
                           [d3*costheta2 , d5*costheta12 , d6             ]])

        C =      np.array([[0 , -d2*sintheta1*theta1_dot , -d3*sintheta2*theta2_dot],
                           [0 , 0                        , d5*sintheta12*theta2_dot],
                           [0 , -d5*sintheta12*theta1_dot, 0                       ]])

        G =      np.array([0 , -f1*sintheta1 , -f2*sintheta2 ])

        H =      np.array([1, 0, 0])

        D_inv = inv(D) #invers of matrix D

        #ODEs q_acc = (x_acc, theta1_acc, theta2_acc)
        # q_acc = D^(-1) * (-C*q_dot - G - H*force)


        q_dot =   np.array([x_dot,
                            theta1_dot,
                            theta2_dot])

        q_acc = np.dot( D_inv , ( -np.dot( C , q_dot ) -G - H*force ))

        x_acc      = q_acc[0]
        theta1_acc = q_acc[1]
        theta2_acc = q_acc[2]

        # Finding next state

        if self.kinematics_integrator == 'euler':
            x = x + self.tau * x_dot
```

```
        x_dot = x_dot + self.tau * x_acc
        theta1 = theta1 + self.tau * theta1_dot
        theta1_dot = theta1_dot + self.tau * theta1_acc
        theta2 = theta2 + self.tau * theta2_dot
        theta2_dot = theta2_dot + self.tau * theta2_acc


    else:  # semi-implicit euler
        x_dot = x_dot + self.tau * x_acc
        x = x + self.tau * x_dot
        theta1_dot = theta1_dot + self.tau * theta1_acc
        theta1 = theta1 + self.tau * theta1_dot
        theta2_dot = theta2_dot + self.tau * theta2_acc
        theta2 = theta2 + self.tau * theta2_dot


    #REWARDS
    sintheta1 = math.sin(theta1)
    sintheta2 = math.sin(theta2)

    flag = self.x_goal_position

    reward = -1

    reward = reward -np.log(abs(flag - x))*3 # x position reward

    reward = reward - abs(theta1 % math.pi)
    reward = reward - abs(theta2 % math.pi)


    if abs(flag - x) < 0.2 :
        reward = reward - np.log((abs(theta1_dot) + abs(sintheta1))) * 10 + 1
        reward = reward - np.log((abs(theta2_dot) + abs(sintheta2))) * 10 + 1



    if (        x     > flag - 0.1  and x          < flag + 0.1
            and x_dot      > - 0.1  and x_dot      <   0.1
            and theta1_dot > - 0.1  and theta1_dot <   0.1
            and sintheta1  > - 0.1  and sintheta1  <   0.1
            and theta2_dot > - 0.1  and theta2_dot <   0.1
            and sintheta2  > - 0.1  and sintheta2  <   0.1
            ):
        reward = reward + 100000.0

    done = bool(x     > flag - 0.1  and x          < flag + 0.1
            and x_dot      > - 0.1  and x_dot      <   0.1
            and theta1_dot > - 0.1  and theta1_dot <   0.1
            and sintheta1  > - 0.1  and sintheta1  <   0.1
            and theta2_dot > - 0.1  and theta2_dot <   0.1
            and sintheta2  > - 0.1  and sintheta2  <   0.1
            )

    #Limits on observation values

    if theta1_dot > 10**20 :
        print('!!! Theta dimension problem !!!')
        theta1 = self.np_random.uniform(low=-math.pi, high=math.pi)
        theta1_dot = self.np_random.uniform(low=-0.1, high=0.1)
        theta1_acc = 0
        #x_dot = 0
        #xacc = 0
        #reward = reward -10000
        done = True

    if theta2_dot > 10**20 :
```

```
            print('!!! Theta dimension problem !!!')
            theta2 = self.np_random.uniform(low=-math.pi, high=math.pi)
            theta2_dot = self.np_random.uniform(low=-0.1, high=0.1)
            theta2_acc = 0
            #x_dot = 0
            #xacc = 0
            #reward = reward -10000
            done = True

        self.state = (x, x_dot, theta1, theta1_dot, theta2, theta2_dot, L2)


        return np.array(self.state, dtype='float32'), reward, done, {}



#_____________________________________________________________________________________________

    def reset(self):
        self.state = np.array([self.np_random.uniform(low=-2,
high=-1)                    , self.np_random.uniform(low=-0.05, high=0.05), # x and x_dot
                               self.np_random.uniform(low=math.pi-0.05,
high=math.pi+0.05), self.np_random.uniform(low=-0.05, high=0.05), # theta1 and thetha1_dot
                               self.np_random.uniform(low=math.pi-0.05,
high=math.pi+0.05), self.np_random.uniform(low=-0.05, high=0.05), # theta2 and thetha2_dot
                               1.0])              # L2 lenght of the load

        self.steps_beyond_done = None
        return self.state


#_____________________________________________________________________________________________

    def render(self, mode='human'):

        if self.state is None:
            return None

        return self.viewer.render(return_rgb_array=mode == 'rgb_array')

#_____________________________________________________________________________________________

    def close(self):
        if self.viewer:
            self.viewer.close()
            self.viewer = None
```