Balancing an Inverted Double Pendulum with Reinforcement Learning

Cem Alptürk[1]   **Pukashawar Pannu**[2]

[1]`ce5368al-s@student.lu.se`   [2]`pu6218pa-s@student.lu.se`

Abstract: Controlling an inverted double pendulum on a cart is an inherently difficult problem. A way to control such a system is to use the current state to determine an action that moves the cart appropriately to balance the double pendulum. One such way is by reinforcement learning. The approach in this project uses a model free controller trained with Deep Q-learning to control the double pendulum. Benefits of Deep Q-learning over regular Q-learning is that the controller can be used for situations that are not presented during training. This has been confirmed in the results. An easier problem to control, the single pendulum problem, was used to confirm the implementation of the learning algorithm. Balancing the double pendulum is a more difficult problem due to its chaotic nature and the controller being limited to a single degree of freedom. Results show that the double pendulum can be balanced indefinitely for certain initial conditions. The agent was trained for a limited number of episodes limiting the amount of experience and knowledge it could gather about the problem. We believe that by tuning hyperparameters, network size and training one could train an agent to balance the double pendulum for broader initial conditions and more optimally.

1. Introduction

The goal of this project is to teach an agent to control a double pendulum on a cart to be standing upright, in its equilibrium point, using reinforcement learning. A double pendulum is a highly chaotic system, where its behavior heavily depends on its initial conditions. Controlling the system with a cart limited to a single degree of freedom is a difficult but solvable problem [6]. Applying control theory requires in-depth knowledge of the system dynamics. However, with reinforcement learning one can achieve similar results without having any knowledge about the system. The aim is to create an agent that can learn from its own experiences to control the double pendulum, by applying a force on the cart in the horizontal direction, such that it stays upright as seen in Figure 1. Although this is an academic problem, the results could be applied to any type of learning problem since the agent is unaware of what it is working on [5].
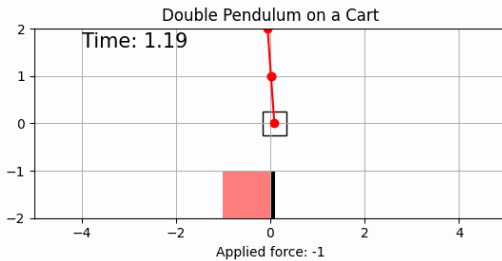

Figure 1. A double pendulum being controlled by a trained agent that is applying $-1N$ in the horizontal direction.

1.1 Reinforcement Learning

The main idea of reinforcement learning is to learn to make decisions based on past experiences [8]. In this case this means to train an agent to perform the best action possible in order to control the pendulums. For each action taken by the agent, the environment returns a new state and a reward for that action. The reward is calculated based on how close the system is to its equilibrium point. The goal of training the agent is to optimize the total sum of rewards until a terminal state is reached. This project implements the Deep Q-Learning [5] method for training the agent.

2. Modeling

In order to train the agent it needs to interact with the system. Since a physical system was unavailable it was decided to represent the system virtually by simulating it. The system consists of a cart that can move in the horizontal direction and two pendulums that are coupled to each other such that there is no collision between any parts. Since this project is not about modeling the double pendulum, but rather about controlling it, the equations of motion have been borrowed from [7]. The states of the system are represented by,

$$x := \begin{bmatrix} y & \dot{y} \end{bmatrix}^T, \quad y := \begin{bmatrix} q & \theta_1 & \theta_2 \end{bmatrix}^T \quad (1)$$

- q is the position of the cart on the horizontal direction.

- θ_1 is the angle of the inner pendulum with respect to the vertical axis in the clockwise direction.

- θ_2 is the angle of the outer pendulum with respect to the vertical axis in the clockwise direction.

- $\dot{q}$ is the velocity of the cart in the horizontal direction.

- $\dot{\theta}_1$ is the angular velocity of the inner pendulum.

- $\dot{\theta}_2$ is the angular velocity of the outer pendulum.

The system parameters are represented by,

- $m = 1\,[kg]$ is the mass of the cart.

- $m_1 = 0.1\,[kg]$ is the mass of the inner pendulum.

- $m_2 = 0.1\,[kg]$ is the mass of the outer pendulum.

- $l_1 = 1\,[m]$ is the length of the inner pendulum.

- $l_2 = 1\,[m]$ is the length of the outer pendulum.

- $g = 9.81\,[m/s^2]$ is the gravitational acceleration.

The equations of motions for this system [7], are adjusted such there is no friction or disturbances. The input to the system is a horizontal force that is acting on the cart, represented by $u\,[N]$.

$$M(y)\ddot{y} = f(y, \dot{y}, u) \qquad (2)$$

$$M(y) := \begin{bmatrix} m + m_1 + m_2 & l_1(m_1 + m_2)\cos\theta_1 & m_2 l_2 \cos\theta_2 \\ l_1(m_1 + m_2)\cos\theta_1 & l_1^2(m_1 + m_2) & l_1 l_2 m_2 \cos(\theta_1 - \theta_2) \\ l_2 m_2 \cos\theta_2 & l_1 l_2 m_2 \cos(\theta_1 - \theta_2) & l_2^2 m_2 \end{bmatrix} \qquad (3)$$

$$f(y, \dot{y}, u) := \begin{bmatrix} l_1(m_1 + m_2)(\dot{\theta}_1)^2 \sin\theta_1 + m_2 l_2(\dot{\theta}_2)^2 \sin\theta_2 + u \\ -l_1 l_2 m_2(\dot{\theta}_2)^2 \sin(\theta_1 - \theta_2) + g(m_1 + m_2)l_1 \sin\theta_1 \\ l_1 l_2 m_2(\dot{\theta}_1)^2 \sin(\theta_1 - \theta_2) + g l_2 m_2 \sin\theta_2 \end{bmatrix} \qquad (4)$$

3. System Design

A couple of reinforcement learning methods are Q-learning and Deep Q-learning.

3.1 Q-learning

Q-learning is a model free method [9] where the agent has no information about the system dynamics it is trying to control, thus it can be said that it is a black box model. The agent stores all its experiences in a Q-table that contains the quality of a state-action combination.

$$Q : S \times A \to \mathbb{R} \qquad (5)$$

$$Q(s, a) := \text{Quality of action } a \text{ given state } s$$

A trained agent decides on the action to perform by looking up the best action a given state s from the Q-table. In the case where multiple actions give the same values in the Q-table, the action is selected randomly among those. The selected action is performed on the environment, which is the inverted pendulum, and a reward is calculated based on the new state of the system. In the case where there is no previous knowledge about the current state in the Q-table, the action is selected randomly. Essentially the agent tries to find the action that will give the best reward.

$$a^* = \max_i Q(s, a_i) \qquad (6)$$

The agent uses an ϵ-greedy policy to decide on the next move, where the agent performs a random action with probability ϵ or performs the best action based on its current policy with probability $1 - \epsilon$. Having a high ϵ value will correspond to performing randomly and exploring the state-action space but will delay the convergence of the Q-table. Having a low ϵ value can cause the agent to rarely try new things but instead will try to exploit its current solution to the problem. A balance between these problems can be obtained by starting with a high ϵ value and decaying it over time to allow the agent to try new things in the beginning and slowly exploit the best solution it has found.

Each state-action pair in a trained Q-table, represents the expected value in Equation (7). These values represent the expected return of rewards for future actions if action a_t is taken at state s_t.

$$Q(s_t, a_t) = E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t, a_t] \qquad (7)$$

Training the agent corresponds to populating and updating the Q-table based on past experiences of the agent. After performing an action given the current state of the system, the agent receives a reward from the environment and the values in the Q-table are updated according to the state-action combination and the reward received. Equation (8) is known as the Bellman equation.

$$Q^{new}(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \max_i Q(s_{t+1}, a_i)) \qquad (8)$$

$$0 < \alpha \le 1, \quad 0 \le \gamma \le 1$$

After performing action a_t given the state s_t, the agent receives the new state s_{t+1} and the reward r_t. The value $Q(s_t, a_t)$ is updated by Equation (8). α is the learning rate and γ is the discount factor. Since the reward for a certain action might not come instantly, the discount factor tries to compensate for a delayed reward by estimating the reward that the agent will receive if it takes the best action a_{t+1} for the state s_{t+1}. Low values of γ will cause the agent to only consider near future rewards, while a γ value close to 1 will consider future rewards.

This method requires a discrete set of actions and states. For the double pendulum problem, this can be achieved by discretizing the states and actions. However since this is a chaotic system, it would require a fine discretization resulting in a massive Q-table. Although this can work in theory, there are practical implementation limitations such as limited memory size. This problem can be resolved by using the Deep Q-learning method.

3.2 Deep Q-learning

In Deep Q-learning, the Q-table is replaced with a deep neural network [5] that tries to learn Q-values instead of memorizing them. A benefit of using Deep Q-learning is that is uses less memory and it has more predictive power on unseen states. The network takes in the system states as input arguments and predicts the Q-values.

$$NN_Q := \mathbb{R}^m \to \mathbb{R}^n \qquad (9)$$

where m is the state size and n is the size of the action space. The agent chooses the action to perform by picking the largest Q-value from the neural network prediction.
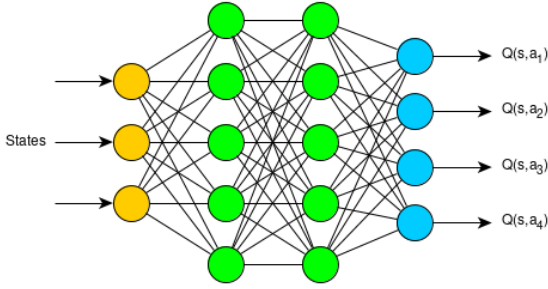

Figure 2. Illustration of a neural network with 3 state inputs and 4 Q-value outputs that can be mapped to actions.

The neural network is trained by a method called experience replay that is based on simulating the system and recording all states, rewards and actions taken in memory [5]. During experience replay, a random batch is taken from the memory and is used to train the network. The reason for taking random batches from memory and not sequential data, is to reduce the correlation between the data points. The simulation ends when it reaches a termination condition that is defined in the environment. For the double pendulum case the termination condition is when either of the pendula angles become larger than a defined maximum value. One full simulation from start to termination is called an episode. The stored experience is used to fit the neural network once there is enough experience stored in the memory.

$$Exp \leftarrow \begin{pmatrix} s_t, & a_t, & r_t, & T_t, & s_{t+1} \end{pmatrix} \qquad (10)$$

Where s_t is the state at time t, a_t the action taken by the system at time t, r_t the gained reward for the taken action, T_t indicating whether the system has terminated and s_{t+1} the next state in time. The experience is used to map a target for the neural network training. Targets are Q-values for each action given a state. Since the experience only holds data about the actual action that was taken for a given state the other Q-values for the same state have to be estimated to complete the entire target vector. There are several ways to estimate the unknown Q-values for a given state. Although the NN_Q network can be used to predict the target values for the other actions it causes the targets to be non-stationary leading to the network chasing its own tail [4]. A separate neural network, NN_{target}, was used to avoid the non-stationary targets problem. NN_{target} has the same network architecture as NN_Q. After a number of episodes the weights of NN_Q are mapped to NN_{target} to give better estimations of the unknown Q-values.

The actual value y_m replaces the corresponding predicted Q-value from $NN_{target}(\mathbf{s}_m)$. Since we only have the reward for one action in a given state, we replace the targets for the untaken actions with a prediction from the Q-network. This will result in a loss of 0 for these untaken actions, thus not affecting the training. Thus the network is trained only by the state-action pair that has been taken. This will however cause the implementation to be slower since a prediction has to be

---

Algorithm 1 Deep Q-Learning

for episode [0, max episodes] **do**
    while $t < t_{max}$ and T_t is **false do**   ▷ Simulate system
        $q_{values} \leftarrow NN_Q(s_t)$
        $a \leftarrow argmax(q_{values})$ ▷ a is index
        $s_{t+1}, r_t, T_t \leftarrow \text{Step}(ActionSpace[a])$
        $Exp \leftarrow \begin{pmatrix} s_t, & a_t, & r_t & T_t & s_{t+1} \end{pmatrix}$
        $\begin{pmatrix} \mathbf{s}, & \mathbf{a}, & \mathbf{r} & \mathbf{T} & \mathbf{s}_{t+1} \end{pmatrix} \leftarrow Exp$ ▷ Exp minibatch
        $y_m = \begin{cases} r_m & \text{terminal} \\ r_m + \gamma \max(NN_{target}(\mathbf{s}_{t+1,m})) & \text{otherwise} \end{cases}$
        $Loss \leftarrow (y_m - NN_Q(\mathbf{s}_{t+1,m})_{a_m})^2$
        Perform gradient descent on Loss
    end while
end for

---

made before each weight update. A custom loss function in `Keras` was implemented where the loss only depends on the Q-value of the taken action to solve this issue.

4. Implementation

The project was implemented in `Python`. A separate class was created for each component `Agent`, `Environment`, `Simulator` and `Controller`, as seen in Figure 3. The goal of the project is to produce a `Controller` object that can control the `Simulator`. `Environment` class contains the problem specifics such as `reward` and `termination` functions, and links the `Agent` with the `Simulator` containing the physical model dynamics. `Agent` contains the learning algorithm that trains and produces the `Controller`. The `Controller` is a simple object that produces the force to apply given the current state of the system.

$$C := \mathbb{R}^M \to \mathbb{R} \qquad (11)$$

All components are independent of each other making the structure modular. By using this structure it is easy to change for example the learning algorithm without having to modify anything else.

4.1 Simulator
The `Simulator` is responsible for the physical dynamics of the problem and for simulating the problem for given external inputs and time. System dynamics are computed using `odeint` function in `Scipy.integrate` package. For the double pendulum case, the simulator solves Equation (2) for $\ddot{y}$ to produce the next state using the ODE solver.

4.2 Environment
The reward function and termination condition is implemented in the `Environment` class. It acts as a connector between the `Agent` and the `Simulator`. This way when the reward or the termination needs to be changed it can be done without altering the learning algorithm (`Agent`) or the physical model (`Simulator`).

4.3 Evaluation
`Controller` is evaluated after a set of episodes in order to keep track of the controller performance. Performance is measured
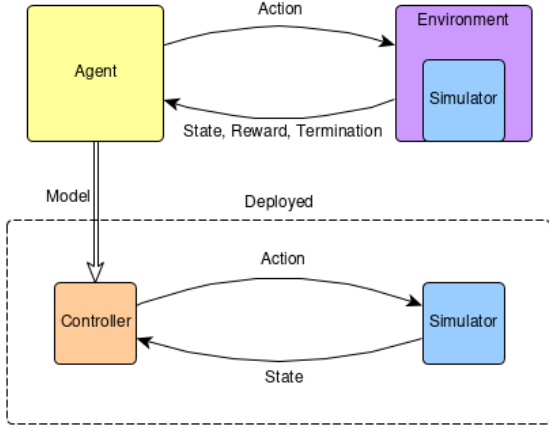
Figure 3. Code architecture.

by averaging the total reward for N_{eval} simulations of the dynamic system controlled by the `Controller`.

$$\frac{1}{N_{eval}} \sum_{i}^{N_{eval}} \sum_{j}^{M_i} R(s_j) \tag{12}$$

Notice that the number of simulation steps M_i for each simulation varies as a simulation may reach a termination condition sooner or later than another simulation. A satisfactory evaluation value can be used as a termination condition for the training to avoid overtraining. This mechanism has not been implemented. The training will continue until the max iterations have been reached. The model is saved if it has a better evaluation performance than the previous best model. This is to avoid overtraining the model.

5. Results

A simpler problem was used to confirm the implementation. The simpler problem being to balance an inverted single pendulum on a cart. The implementation architecture was used to simply replace the environment with one suitable for the single pendulum problem.

5.1 Balancing Inverted Single Pendulum

Equations from [1] were used to model the problem. System was terminated when the pendulum angle was greater than ±10 degrees. Equation (13) states the reward function.

$$R(s) = (\theta_{max} - |\theta|)/\theta_{max} \tag{13}$$

where $\theta_{max} = 10$ degrees. The value was chosen by trial and error, where 10 degrees gave the best overall performance.

Table 1 shows the neural network setup for the single pendulum problem. The weights of the network were initialized with uniform HE-initialization, which is an optimized weight initialization method when using Rectified Linear Unit (ReLU) activation functions [3].

Layer	Number of Neurons	Activation
Input	3	-
Hidden 1	20	ReLU
Hidden 2	40	ReLU
Output	3	Linear

Table 1. Network parameters for the single pendulum Q-network.

Although the single pendulum has 4 states, the position of the cart, x, was not used, to simplify training. Table 2 shows the training parameters for the single pendulum problem.

Max Angle	10
Step Size	0.02
Action Space	[-10,0,10]
Learning Rate	0.001
Memory	2000
Batch Size	32
Discount	0.9

Table 2. Parameters for the Deep Q-learning algorithm on the single pendulum problem.

Figure 4 shows the agent performance over episodes. The agent starts to learn how to move the cart to increase the score after a few episodes. Eventually the reward gain stops increasing. The system was simulated with the maximum 200 steps for each episode, capping the max possible score to 200. However, 200 is only possible if the initial condition starts in the equilibrium point. The initial condition for each episode is randomized preventing the system from reaching 200 score in the evaluation.
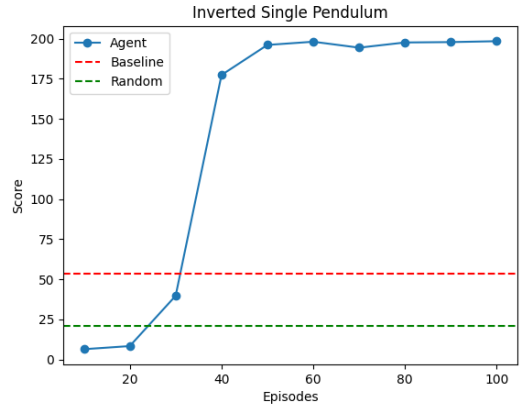

Figure 4. Evaluated controller score for every 10 episodes. The evaluation is measured as the average score of 10 full controlled simulations. Maximum number of steps for each evaluation iteration was 200, capping the maximum possible reward to 200.

The trained agent was used to control the single pendulum for initial conditions outside the trained domain, and the controller managed to bring back the pendulum to the known domain and balance it. This can be seen in Figure 5. Results indicate that the network has managed to generalize the problem. This would not be possible with a regular Q-table approach since the agent has never seen the states outside the training domain and would end up performing random actions.
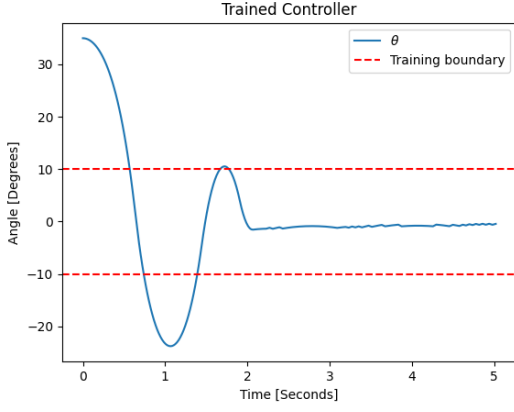
Figure 5. Changes in pendulum angle θ over time for controlled simulation of single pendulum on cart problem. The initial condition was $\theta = 35$ degrees.

Figure 6 and 7 shows the average Q-values during training for $\gamma = 0.1$ and $\gamma = 0.9$ respectively. It can be seen that the Q-values for a low γ value converges much faster while for the larger γ value, convergence has not happened at the end of the training.
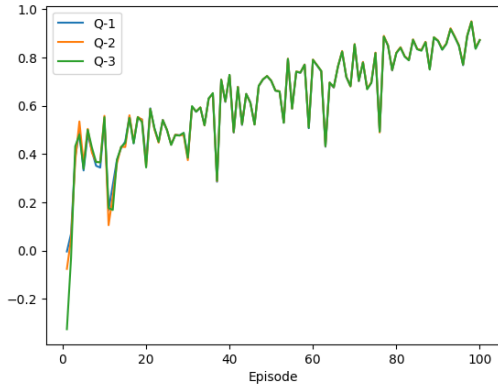

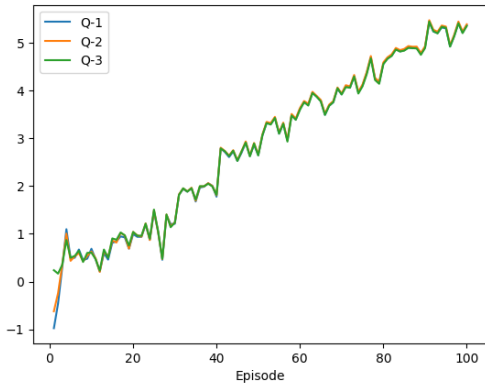**Figure 6.** Q-values for $\gamma = 0.1$.

Figure 7. Q-values for $\gamma = 0.9$.

Figure 8 shows the training result for $\gamma = 0.1$, while the

other hyperparameters remained the same. It can be seen that the score converges slightly faster since the Q-values converge faster. This is due to the algorithm only taking the instant and near future rewards in to consideration, thus being short sighted. However the overall performance showed that, with $\gamma = 0.1$ the agent was unable to generalize and even had issues with initial conditions that were close to θ_{max}. This can also be seen in the dip during training in Figure 8.
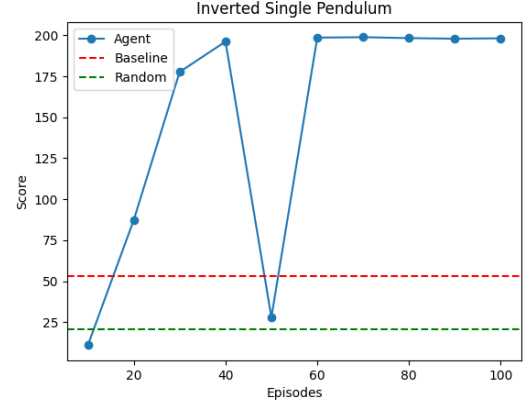

Figure 8. Evaluation of the controller with the same conditions in Table 2, except that $\gamma = 0.1$.

Figure 9 shows the distribution of actions taken for an agent that was trained with a larger action space. This plot shows that most of these actions were rarely used except for ± 10. When an agent was trained with the larger action space for the same number of episodes, the overall performance was worse than in Figure 4. This showed that a larger action space requires more training in order to get the same performance. In this case training time can be reduced by only using the actions that were most used by the agent.
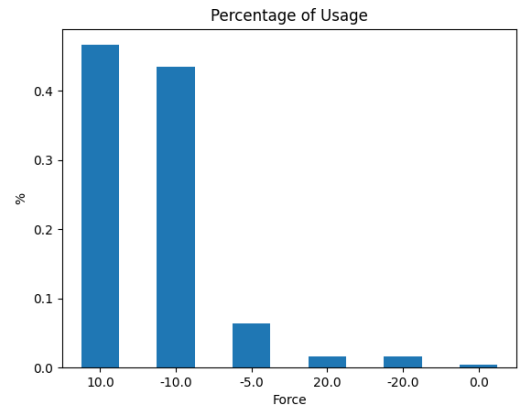

Figure 9. The distribution of used actions for an action space of $[-20, -10, -5, 0, 5, 10, 20]$. The model was trained with the same hyperparameters in Table 2.

5.2 Balancing Inverted Double Pendulum
The termination condition is given by Equation (14),

$$|\theta_1| > \theta_{max} \text{ or } |\theta_2| > \theta_{max} \qquad (14)$$

where θ_1 is the angle of the inner pendulum and θ_2 is the angle of the outer pendulum. $\theta_{max} = 15$ degrees was chosen in order to train the agent in a strict state space close to the equilibrium point. Due to the limited degrees of freedom of the system, higher values of θ_{max} can allow the pendulums to move to positions where they can not be recovered from. Reward is given by Equation (15),

$$r(s) = 1 - \left(\frac{|\theta_1|}{\theta_{max}} + \frac{|\theta_2|}{\theta_{max}} \right) - \left(\dot{\theta_1}^2 + \dot{\theta_2}^2 \right)$$

$$R(s) = \begin{cases} -10 & \text{if (14)} \\ \max(0, r(s)) & \text{otherwise} \end{cases} \quad (15)$$

where the agent receives a bonus of 1 for being alive, a penalty for being further away from the equilibrium point given by $\left(\frac{|\theta_1|}{\theta_{max}} + \frac{|\theta_2|}{\theta_{max}} \right)$, and a penalty for the velocity of the pendulums given by $\dot{\theta_1}^2 + \dot{\theta_2}^2$. A penalty of -10 is given to the agent if it crosses the termination boundary. This penalty is designed to discourage the agent from moving outside the boundary. The total score received in an episode depends on the step size of the simulation and the duration that the agent is able to survive.

Table 3 shows the neural network setup for the double pendulum problem. The weights of the network were initialized with uniform HE-initialization similar to the single pendulum problem. The position of the cart has not been used as input to simplify training.

Layer	Number of Neurons	Activation
Input	5	-
Hidden 1-5	[30, 20, 40, 30, 30]	ReLU
Hidden 6-10	[40, 30, 60, 60, 100]	ReLU
Hidden 11-15	[40, 30, 70, 70, 70]	ReLU
Hidden 16-20	[30, 30, 30, 30, 30]	ReLU
Output	3	Linear

Table 3. Network parameters for the double pendulum Q-network.

Table 4 shows the training parameters for the double pendulum problem.

Max Angle	15
Step Size	0.02
Action Space	$[\pm40, \pm10, \pm5, \pm1, 0]$
Learning Rate	0.00005
Memory	2000
Batch Size	32
Discount	0.9

Table 4. Parameters for the Deep Q-learning algorithm on the double pendulum problem.

Figure 10 shows the agents training performance over 2000+ episodes. During training the agent was able to balance the pendulum for up to ten seconds. Episodes were terminated if the agent lasted for ten seconds to speed up training, even though this terminates the episode, the agent is not given a penalty. A comparison between the agent's performance, baseline and a random controller is also illustrated in the figure. The trained agent performs much better than the baseline and the random controller.
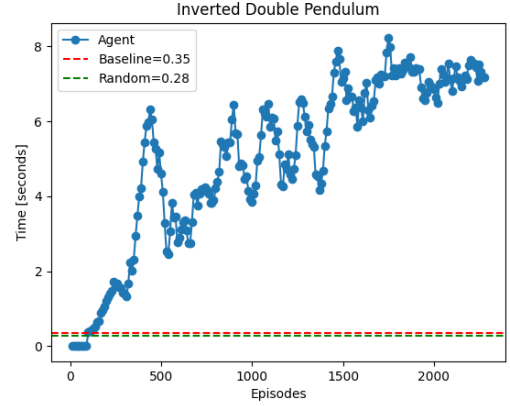

Figure 10. Evaluated controller score for every 10 episodes. The evaluation is measured as the average score of 10 full controlled simulations. The baseline is measured by evaluating 200 simulations without a controller which lasted on average for 0.35 seconds. A controller that performs actions randomly survives an average time of 0.28 seconds, estimated the same way as the baseline.

Figure 11 shows the change in θ_1 and θ_2 for a simulation with the trained controller where the initial conditions were outside the training boundary. This shows that the agent was able to generalize since it has never seen these states.
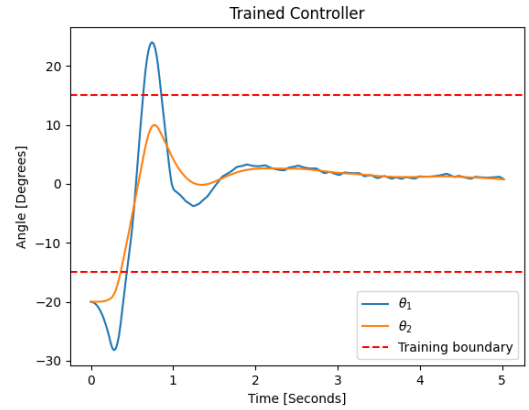

Figure 11. Change in pendulum angles controlled by trained agent for initial conditions set as $\theta_1 = \theta_2 = -20°$.

Figure 12 shows the force distribution for initial conditions that were outside of the training region. The agent uses the larger forces initially to get the pendulums closer to the equilibrium and then starts using the smaller values to keep it there.

5.3 Training Time

The training time of the agent heavily depends on the hyperparameters and the overall performance of the agent. In the first few episodes, due to the agents lack of experience the episodes will terminate faster, thus resulting in shorter episodes. As the agent gains more experience, it learns to survive for longer,
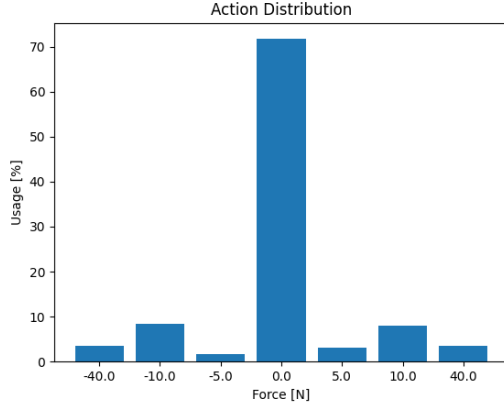
Figure 12. Force distribution for the trained agent controlling the pendulums with initial conditions of $\theta_1 = \theta_2 = -20°$.

increasing the computation time for a single episode. Table 5 and 6 show the computation and simulation time differences for a change in the termination condition.

Single Pendulum		
Episode	Computation	Simulation
1	0.72 s	0.56 s
50	20.73 s	3.5 s
100	28.52 s	4 s

Table 5. Computation and simulation times for episodes with hyperparameters specified in Table 2.

Single Pendulum		
Episode	Computation	Simulation
1	0.76 s	0.6 s
50	6.03 s	0.86 s
100	26.77 s	4 s

Table 6. Computation and simulation times for episodes with hyperparameters specified in Table 2, with a modification of $\theta_{max} = 20$ degrees.

Double Pendulum		
Episode	Computation	Simulation
1	2.95 s	0.8 s
300	36.74 s	7.18 s
2000	60.24 s	10 s

Table 7. Computation and simulation times for episodes with hyperparameters specified in Table 4.

6. Discussion

The double pendulum is a highly chaotic system and with the cart limited to a single degree of freedom it is difficult to control. In situations where the agent receives two states that are very close to each other, the applied action may result in very different outcomes which makes the learning of the expected reward for that action challenging. We have found that deeper networks perform better than wider ones for this problem. This may be due to wider networks having less non-linearity compared to deeper ones.

After playing around with the exploration rate ϵ, we have found that a dynamic exploration rate performs better for this problem, rather than a static one. Higher exploration rate forces the agent to take actions that it normally would avoid causing it to explore the action space more. For Q-learning, this is a desired thing since the purpose is to fill and update the Q-table. However, a high exploration rate leads to the agent exploring things that it should avoid, rather than finding the best action to perform. In Q-learning, this is not an issue since the Q-values of the state-action pairs do not directly affect each other. However, for Deep Q-learning, every weight update affects all Q-values for the state-action pairs. In the case where the learning rate is too low, the agent will almost always prefer to take the same actions which can cause overfitting in the network. This is why we preferred to train the agent with a dynamic exploration rate, where the agent performs randomly in the beginning and over time becomes more deterministic. However, we still have a minimum ϵ, to allow the agent to explore during the later episodes of training. This prevents the agent from overfitting to states close to the equilibrium point. For the same reason, we preferred to use a strict termination condition. Since the agents purpose is to control the pendulums around the equilibrium, there is little to no advantage in allowing the pendulums to move too far from the equilibrium point.

When we used a discount factor closer to 1, we found that it takes longer for the Q-values to converge, and they converged to larger values, as seen in Figure 7. Although this specific problem does not have any delayed rewards, such as reaching a goal in a game, it made more sense to use a high discount factor γ, so that the agent can take into account the consequences of its actions, such as if an action makes the pendulum overshoot the equilibrium point or not. Our results indicated that using a higher γ value helped the agent generalize much better and was more resilient to initial conditions outside the training boundary.

As seen in Table 7 tuning hyperparameters is a difficult process. Due to the time the training takes, evaluating the performance of a different set of hyperparameters is challenging. Since the training process highly depends on the hyperparameters, the time it takes for one episode to finish can be vastly different for different hyperparameters. Aside from the time it takes to get the results, evaluating the effect of the hyperparameters is not straight forward. Even when using the same set of hyperparameters, we can get different results. Trying to estimate the effect of a new hyperparameter is problematic. Due to these reasons, a trial and error approach was more effective. We believe that with the right hyperparameters, the double pendulum can be controlled more optimally.

The duration that the agent survives for an episode does not directly correlate with the agents performance. Different hyperparameters, reward functions and termination conditions affect the duration the agent is able to survive, which makes evaluating hyperparameters challenging. As seen in Table 5 and 6, the agent was trained with a more flexible termination condition, which allowed it to survive longer in an episode,

however it takes more episodes for the Q-values to converge since the state-action space is much larger than before which takes more training time.

Other model-free methods such as Linear Q-learning [10] and Double Deep Q-learning [2] can be implemented and perhaps give similar or better results. Another interesting problem would be to train a swing-up controller. This can be achieved by modifying the reward function in a way that it encourages the agent to swing the pendulums up and balance them around the equilibrium. This can be considered for future work.

References

[1] R. Florian. "Correct equations for the dynamics of the cart-pole system" (2005).

[2] H. van Hasselt, A. Guez, and D. Silver. "Deep reinforcement learning with double q-learning". *CoRR* **abs/1509.06461** (2015). arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461.

[3] K. He, X. Zhang, S. Ren, and J. Sun. "Delving deep into rectifiers: surpassing human-level performance on imagenet classification". *IEEE International Conference on Computer Vision (ICCV 2015)* **1502** (2015). DOI: 10.1109/ICCV.2015.123.

[4] T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, A. Sendonaris, G. Dulac-Arnold, I. Osband, J. Agapiou, J. Leibo, and A. Gruslys. "Learning from demonstrations for real world reinforcement learning" (2017).

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. "Playing atari with deep reinforcement learning" (2013).

[6] L. Moysis. *Balancing a double inverted pendulum using optimal control and laguerre functions*. 2016. DOI: 10.13140/RG.2.1.2948.6486.

[7] Z. Neusser and M. Valasek. "Control of the double inverted pendulum on a cart using the natural motion". *Acta Polytechnica* **53** (2013). DOI: 10.14311/AP.2013.53.0883.

[8] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction Second Edition*. 2018.

[9] C. Watkins and P. Dayan. "Technical note: q-learning". *Machine Learning* **8** (1992), pp. 279–292. DOI: 10.1007/BF00992698.

[10] Y. Zheng, S. Luo, and Z. Lv. "Control double inverted pendulum by reinforcement learning with double cmac network". In: vol. 4. 2006, pp. 639–642. DOI: 10.1109/ICPR.2006.416.