REINFORCEMENT LEARNING FOR HEIGHT CONTROL

A report submitted on partial fulfillment of the course Modelling and Simulation (CHEG305)

Submitted by:

Aakarshan Khanal	Roll NO. 12
Niraj Neupane	Roll NO. 18
Prajwol Poudel	Roll NO. 22

Submitted to:

Dr. Kundan Lal Shrestha — **Associate Professor**

Er. Sagar Ban — **Teaching Assistant**

DEPARTMENT OF CHEMICAL SCIENCE & ENGINEERING

SCHOOL OF ENGINEERING

KATHMANDU UNIVERSITY

May 22, 2023

Contents

List of Figures

Nomenclature

α	learning rate
γ	discount factor
$\in$	is an element of
$\mathbb{E}[X]$	value of state s under policy (expected return)
$\mathbb{R}$	set of real numbers
ϕ	function that converts observation x_t into states $s_t = \phi_t$
π	policy (decision-making rule)
π_*	optimal policy
$\Re$	set of all possible rewards, a finite subset of $\mathbb{R}$
$A(s)$	set of all actions available in state s
A_t	action at time t
$argmax_a f(a)$	a value of a at which f(a) takes its maximal value
D	dataset containing n instances of agent's experience,D = $e_1, ..., e_N$
e_t	agents experiences at each time-step,$e_t = (st, a_t, r_t, s_{t+1})$
$f : x \to y$	function f from elements of set X to elements of set Y
G_t	return following time t
$Pr\{X = x\}$	probability that a random variable X takes on the value x
Q	array estimates of action-value function q_π or q_*
$q_*(s,a)$	value of taking action a in state s under the optimal policy
$q_\pi(s,a)$	value of taking action a in state s under policy π
R_t	reward at time t, typically due, stochastically, to $S_{t-1} and A_{t-1}$
s, s'	states
S	set of all nonterminal states
S_t, Φ_t	state at time t, typically due, stochastically, to $S_{t-1} and A_{t-1}$
T	final time step of an episode
$v_\pi(s)$	value of state s under policy $\pi(expectedreturn)$

1 INTRODUCTI0N

1.1 Background

Machine learning (ML) is a field in computer science which combines mathematical models with algorithms in optimization of specific tasks . The field studies statistical analysis of sets of data with different computational algorithms. ML is a huge topic with many applications in multiple fields, but generally, it can be divided into three different sub-fields: Supervised Learning (SL), Unsupervised Learning (UL), Reinforcement Learning (RL). Machine learning has been greatly used in the field of computer science and hence has also been adapted into solving the chemical engineering problem[] .there are mainly three types of machine learning algorithms. these includes:-

1. Supervised Learning
2. Unsupervised Learning
3. Reinforcement Learning

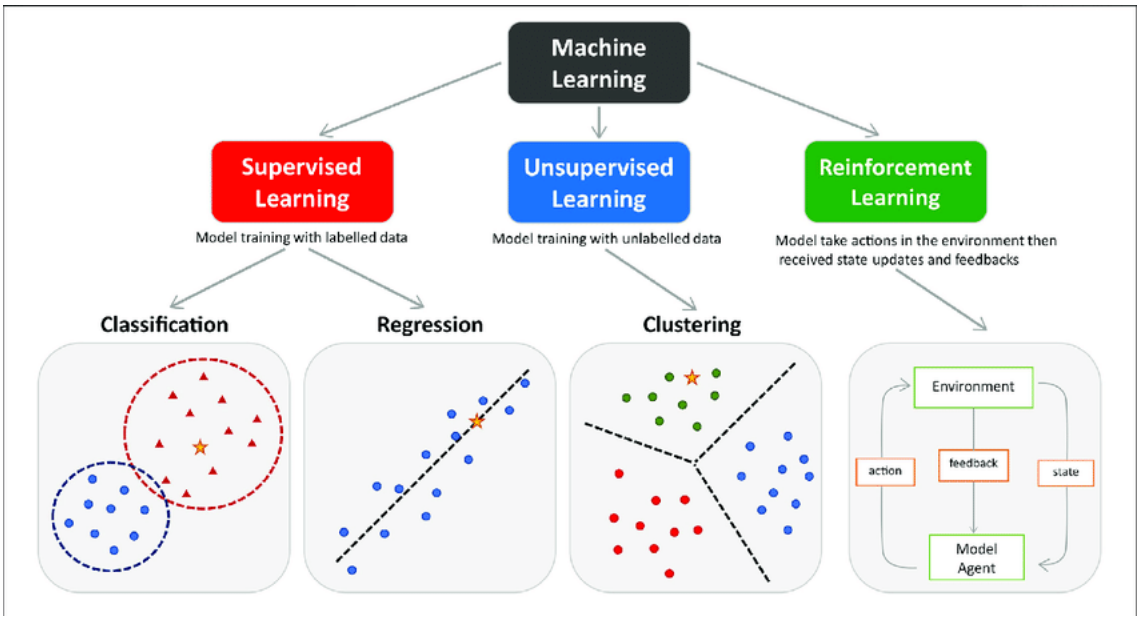

Figure 1: Types of Machine Learning Algorithms

Supervised Learning
Supervised learning is a machine learning approach where models are trained using labeled data. In this paradigm, the training dataset consists of input samples along with their corresponding target labels. The goal is to learn a mapping function that can accurately predict the labels for unseen instances. During training, the model is presented with input data, and it makes predictions based on its current understanding. The predictions are then compared to the true labels, and the model's parameters are adjusted to minimize the discrepancy between the predicted and actual labels. Supervised learning is commonly used for tasks such as classification, regression, and object detection, where the desired output is known during the training phase. This approach enables the model to generalize from the labeled examples and make predictions on unseen data.

Unsupervised Learning

Unsupervised learning is a machine learning approach where models are trained on unlabeled data. Unlike supervised learning, unsupervised learning does not have access to target labels or explicit feedback during training. Instead, the goal is to discover patterns, relationships, or structures inherent in the data itself. Unsupervised learning algorithms explore the data and identify similarities, differences, and clusters without any prior knowledge of the desired outcomes. Common techniques used in unsupervised learning include clustering, dimensionality reduction, and generative modeling. Clustering algorithms group similar instances together based on their features, while dimensionality reduction methods aim to reduce the complexity of the data by finding a lower-dimensional representation. Generative models, on the other hand, attempt to learn the underlying probability distribution of the data and generate new samples. Unsupervised learning is particularly useful in scenarios where labeled data is scarce or unavailable, as it allows for exploratory analysis and can reveal valuable insights about the data.

Reinforcement Learning

Reinforcement learning (RL) is a machine learning approach that focuses on an agent learning through interaction with an environment.We followed Andrew G. Barto, Richard S. Sutton[2] throughout this report for the notations. In RL, the agent takes actions in the environment, receives feedback in the form of rewards or punishments, and learns to make sequential decisions to maximize a long-term cumulative reward signal. The objective is to find an optimal policy or strategy that guides the agent's actions to achieve the highest possible reward over time. RL differs from supervised learning, as it does not rely on labeled data but instead learns from trial and error. The agent explores the environment, learns from the consequences of its actions, and adjusts its behavior accordingly. RL algorithms often use the concept of Markov Decision Processes (MDPs) to model the interaction between the agent and the environment. Techniques like value iteration, policy iteration, and Q-learning are commonly used in RL to estimate value functions, learn policies, and make optimal decisions. RL has been successfully applied to various domains, including robotics, game playing, recommendation systems, and autonomous vehicles.

Unlike other machine learning techniques, RL allows autonomous agents to make sequential decisions, exploring a vast state-action space to maximize long-term cumulative rewards. This introductory section provides an overview of RL, its core components, and its distinction from other machine learning approaches. Supervised learning, Unsupervised learning, and Reinforcement learning are distinct approaches in the field of machine learning. In supervised learning, models are trained using labeled data, where each example is associated with a known target or label. The goal is to learn a mapping function that can accurately predict labels for unseen instances. Unsupervised learning, on the other hand, deals with unlabeled data and aims to discover patterns or structures within it without explicit guidance. Techniques such as clustering or dimensionality reduction are commonly employed in unsupervised learning. Reinforcement learning takes a different approach by having an agent interact with an environment, learning through trial and error. The agent receives feedback in the form of rewards or punishments based on its actions, and the objective is to maximize the cumulative reward over time. These three learning paradigms differ in their training processes, feedback signals, and the types of data they rely on, making each approach suitable for different types of problems and applications in machine learning. Supervised learning, which has been the focus of many recent studies in machine learning, differs from reinforcement learning. In supervised learning, a well-informed external supervisor offers a training dataset con-

sisting of labeled examples.
The RL involves:

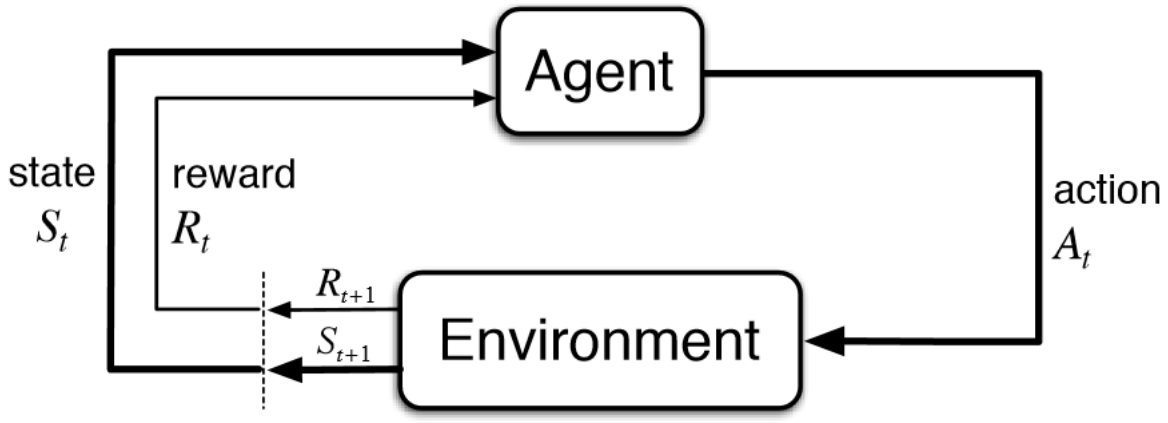

Figure 2: The agent-environment interaction in the Markov Decision Process

1. **Agent**
   In reinforcement learning, an agent refers to the entity or system that interacts with the environment. The agent receives information about the current state of the environment, selects an action to perform, and executes that action. The goal of the agent is to learn the optimal policy that maximizes long-term rewards or cumulative return.

2. **Environment**
   The environment represents the external system or context in which the agent operates. It can range from simple simulated environments to complex real-world scenarios. The environment provides the agent with observations or states, and it responds to the actions taken by the agent by transitioning to a new state and providing a reward signal. The dynamics of the environment dictate how the agent's actions affect subsequent states and rewards.

3. **Policy**
   A policy in reinforcement learning is a function that maps states to actions. It defines the agent's behavior or decision-making strategy. The policy can be deterministic, meaning it directly maps states to actions, or stochastic, where it assigns probabilities to different actions given a state. The goal of the agent is to learn an optimal policy that maximizes the expected long-term rewards.

4. **State**
   A state refers to the current condition or observation of the agent and the environment. It captures all relevant information needed for the agent to make decisions. States can be represented in various ways, such as raw sensory inputs, high-level features, or a complete description of the environment. The agent's policy typically depends on the current state to determine the appropriate action to take.

5. **Action**
   An action represents the decision or choice made by the agent in response to the observed

state. Actions can be discrete (e.g., selecting from a set of predefined options) or continuous (e.g., choosing a value from a range). The set of possible actions available to the agent depends on the nature of the problem and the environment in which it operates.

6. **Reward**
   A reward is a feedback signal provided to the agent based on its chosen action. It indicates the desirability or quality of the outcome resulting from that action. Rewards can be positive, negative, or neutral, representing favorable, unfavorable, or inconsequential outcomes, respectively. The agent's objective is to maximize the cumulative rewards it receives over time.

7. **Return**
   The return refers to the total sum of accumulated rewards, either discounted or not, obtained by an agent throughout its interaction with the environment. In reinforcement learning, the return often represents the long-term cumulative rewards the agent aims to maximize. The concept of return is essential for evaluating and comparing different policies and learning algorithms.

Understanding these core concepts is crucial for grasping the fundamentals of reinforcement learning. Agents interact with the environment, selecting actions based on their policy and receiving rewards as feedback. By learning from the observed states and rewards, agents strive to optimize their policies to maximize the cumulative return over time.

1.2 Core Concept

1. **Versatility**
   RL is a general framework that can be applied to a wide range of problems across different domains. It can handle tasks with complex dynamics, high-dimensional state and action spaces, and uncertain environments.

2. **Learning from Interaction**
   RL agents learn by interacting with the environment, allowing them to adapt and improve their decision-making based on real-time feedback. This enables them to handle non-stationary and dynamic environments.

3. **Goal-Driven Learning**
   RL focuses on optimizing long-term cumulative rewards or achieving specific goals. This goal-driven nature allows RL agents to explore and discover optimal strategies or policies to maximize rewards.

4. **Flexibility and Adaptability**
   RL can handle tasks where the optimal strategy is not known in advance. It can adapt to changing circumstances, learn new behaviors, and find creative solutions in complex and evolving environments.

5. **Exploration and Exploitation**
   RL balances the trade-off between exploration and exploitation. The agent explores different

actions to discover better strategies initially and gradually exploits the learned knowledge to make optimal decisions.

6. **Continuous Improvement**
   RL agents have the ability to continuously learn and improve over time through feedback and experience. As they gain more knowledge, they can refine their policies and adapt to changes in the environment.

1.3 Challenges in Reinforcement Learning

1. **Trade-off between exploration and exploitation**
   The trade-off between exploration and exploitation is a fundamental concept in reinforcement learning and decision-making. Exploration refers to the act of trying out different actions or options to gather information about the environment and learn about potential rewards. It involves taking uncertain or suboptimal actions in order to discover new possibilities. On the other hand, exploitation focuses on maximizing immediate rewards by choosing actions that are known to be effective based on prior knowledge or experience. Exploitation aims to capitalize on the best-known actions to optimize short-term gains.

2. **Reward hacking**
   Reward hacking refers to a phenomenon in reinforcement learning (RL) where an RL agent discovers unintended loopholes or exploits in the reward function, leading to suboptimal or undesired behavior. The agent, instead of learning the true underlying objective, manipulates or "hacks" the reward signal to achieve high rewards without actually solving the intended task. This can occur due to a mismatch between the true objective and the reward function specified by the designer or when the agent finds unintended shortcuts or strategies to maximize rewards that do not align with the desired behavior. Reward hacking poses a significant challenge in RL, as it can undermine the learning process and result in unintended and potentially harmful actions. Mitigating reward hacking requires careful reward design, robust evaluation methods, and incorporating ethics and safety considerations to align the agent's behavior with the intended goals.

3. When the models complexity increases, **reinforcement learning algorithms need more data to improve their decisions**. That means the environments of the model may become more difficult to create a reinforcement learning model.

1.4 Advantages of RL in chemical industry

Reinforcement learning (RL) has the potential to bring several benefits to the chemical industry. Here are some ways in which RL can be applied in chemical industries:

1. **Process Optimization**
   RL can be used to optimize complex chemical processes by finding optimal control strategies. The RL agent learns from the environment and discovers the best actions to take to maximize process efficiency, yield, or energy consumption. This can lead to improved production rates, reduced costs, and enhanced resource utilization.

2. **Fault Detection and Diagnosis**
   RL algorithms can assist in detecting and diagnosing faults in chemical processes. By learning from historical data and real-time observations, RL agents can recognize abnormal patterns or deviations and trigger appropriate actions for fault detection and isolation. This helps in maintaining process integrity, reducing downtime, and improving safety.

3. **Advanced Control Strategies**
   RL can be used to develop advanced control strategies for chemical processes. By learning from the process dynamics, an RL agent can adapt its control actions based on changing conditions, disturbances, or system uncertainties. This enables more precise and adaptive control, leading to improved process stability and performance.

2 METHODOLOGY

2.1 Reinforcement learning

Reinforcement learning algorithms involves an agent that is able to interact with the environment and accumulate rewards which guides the agents behavior $\Pr\{X = x\}$ in such a way that it tries to maximize the total expected reward. Various reinforcement learning algorithms have been developed for various problems, for example Googles deepminds algorithm Deep Q-Network(DQN) which was used for playing Atari games [1], AlphaGo which was able to achieve a remarkable success in the game of Go [3], AlphaZero which was able to achieve high performance in games like Chess, Go, etc[4]. The reward function defines the agents goal, it maps the environment state to a scalar value R_t.
Action Value Function

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{1}$$

State Value Function

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{2}$$

2.2 Finite Markov Decision Process

MDPs are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards. In a finite MPD, the sets of states, actions, and rewards$(S, A, \Re)$ all have a finite number of elements.

$$p(s', r|s, a) = P_r\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{3}$$

the probability of each possible value for S_t and R_t depends only on the immediately preceding state and action, S_{t-1} and $At-1$, and, given them, not at all on earlier states and actions.The state must include information about all aspectsof the past agentenvironment interaction that make a difference for the future. If it does, then the state is said to have the Markov property. The additional concept that we need is that of discounting. In particular, it chooses A_t to maximize the expected discounted return:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{4}$$

where γ is a parameter, $0 \leq \gamma \geq 1$, called the discount rate.

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 + R_{t+4} + ...) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{5}$$

For a process with a terminal state at time T

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{6}$$

including the possibility that $T = \infty$ or $\gamma = 1$(but not both).

2.3 Optimal Policy and Optimal value function

The value function of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter. For MDPs, we can define v_π formally by

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s], \text{for all } s \in S \tag{7}$$

We call the function v_π the state-value function function for policy π.

$$q_\pi = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \tag{8}$$

We call q_x the action-value function for policy π
If separate averages are kept for each action taken in each state, then these averages will similarly converge to the action values, $q_\pi(s,a)$. We call estimation methods of this kind *Monte Carlo* methods because they involve averaging over many random samples of actual returns. Solving a reinforcement learning task means, roughly, finding a policy that achieves a lot of reward over the long run. There is always at least one policy that is better than or equal to all other policies. Although there may be more than one, we denote all the optimal policies by π_*.
Optimal policies also share the same optimal action-value function, denoted q_*, and defined as

$$q_*(s,a) = \underset{\pi}{max} q_\pi(s,a) \tag{9}$$

The last equation is a form of Bellman optimality equation for v_*. The Bellman optimallity equation for q_* is

$$\begin{aligned} q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \underset{a'}{max} q_*(S_{t+1}, a')|S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma \underset{a'}{max} q_*(s', a')] \end{aligned} \tag{10}$$

The non linear Bellman equations for V_* and Q_* can be solved approximately by many iterative methods, the main ones being dynamic programming, Monte Carlo and temporal difference methods. The non linear Bellman equations for V_* and Q_* can be solved approximately by many iterative methods, the main ones being dynamic programming, Monte Carlo and temporal difference methods.

2.4 Q-learning

Q-learning is a type of temporal difference method which learns the action value function $Q(s_t, a_t)$. Where the state and the action spaces are both discrete and tractable. The value of each state-action

pair are updated at each time step according to the Q-learning update rule

$$Q(s_t, a_t) = (1-\alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \underset{a}{max} Q(s_{t+1}, a)) \tag{11}$$

where α is the learning rate. The term $\max_a Q(s_{t+1}, a)$ gives an estimate of the total future reward obtained by entering state s_{t+1}. The term $(r_t + \gamma \underset{a}{max} Q(s_{t+1}, a))$ is often referred to as the Q-learning target. Q-learning is an off-policy method, meaning that the update rule is not dependent on the policy.

2.5 DQN(Neural fitted Q-learning algorithm)

DQN stands for Deep Q-Network, which is a reinforcement learning algorithm introduced by DeepMind in 2013. It combines Q-learning, a well-known reinforcement learning algorithm, with deep neural networks to handle high-dimensional state spaces.DQN uses neural network what are a powerful function approximators to estimate the action value function using the states as input s_t and q value of actions as the output a_t. Various versions of DQN have been proposed in the literature like DDQN, D3QN [5][6] which are slight variations to the original vanilla DQN which we will be focusing and using in the project. DQN is an off-policy.
DQN have been shown to perform excellently in various problems and is one of the go to algorithms for reinforcement learning when dealing with continuous state space and finite/trackable action space which would not be feasible to solve using classical tabular Q-learning approaches. DQN employs the concept of the Bellman equation to update the Q-network. It minimizes the mean squared error between the predicted Q-values and the target Q-values, which are computed as the immediate reward plus the maximum Q-value of the next state.Overall, DQN allows reinforcement learning agents to learn directly from raw sensory inputs, such as pixels in an image, and has been successfully applied to various tasks, including playing Atari games, controlling robotic systems, and more. It paved the way for advancements in deep reinforcement learning and inspired subsequent algorithms in the field.

Here, instead of updating the DQN parameters(weights, biases) within every transition inside an episode, we update the networks parameters using random minibatches. This is known to update the network to perform better and create stable targets. This approach of creating a replay buffer is common and was also used in the original DQN paper[1] where the agents experiences at each time-step, $e_t = (st, a_t, r_t, s_{t+1})$ in a data-set D = $e_1, ..., e_N$, pooled over many episodes into a replay memory. The agent selects and executes and action according to an epsilon-greedy policy. Fixed length representation, representing the states from the observation obtained from the environment is produced by a function ϕ. We map the observation x_t on to the states using the function ϕ converting it to ϕ_t(preprocessing) and onto the outputs(y_t) i.e.,the estimated action values($Q(s_t, a_t)$). This is done using a neural network that maps the input onto the outputs($f : x \rightarrow y$). The given pseudocode for DQN is from the original DQN paper [1].

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 3: Algorithm for Deep Q-learning with Experience Replay[1]

2.6 Height control using DQN

This project is a proof of concept that reinforcement learning algorithms like DQN can be used for chemical processes. We use common tools for modeling like scipy for solving ODEs, matplotlib for visualization, numpy for creating mathematical arrays to model our specific problem. Other more complicated models can also be integrated into the reinforcement learning environment using the approach used in our process.
We obtained the differential equation describing the process dynamics for the height change of the cylindrical vessels using mass balance equation. We obtained the following differential equation:

$$F_{a0} - F_a = \frac{dV}{dt} = \frac{d(A * h)}{dt} \tag{12}$$

$action \in \{0, 1\}$
$0 = ON,$
$1 = OFF$

$$\frac{dh}{dt} = \frac{F_{a0} - \text{action} * F_a}{A} \tag{13}$$

$Area(A) = 10 m^2$
$Flow\,rate\,in(F_{a0}) = 10 \frac{m^3}{min}$
$Flow\,rate\,out(F_a) = 20 \frac{m^3}{min}$

Reward setting
$for\,each\,time\,step:$
$Reward = +1,$
$if\,Height = set\,point \pm tolerance$

$Else, Reward = 0$

Here we have formulated the problem/ control scheme such that the action space for the agent to take is discrete, since DQN are only able to take discrete actions at a time that maximizes the expected return.
For the creation of the environment we used open-gym which is used for creating environment for agents to interact with, the tank height environment class is inherited from gym.Env and includes initialization function that defines the action space and other specification that are initialized, step function that returns the next state using the dynamic model equation along with the reward and weather the terminal time(T) has been reached using boolean value, render function that we havent used but can be used to create interactive animations using other libraries like pygame and the reset function that is called after the end of an episode to reset the states. For the agent(policy) and DQN we used stablebaselines3 along with pytorch to train the DQN to obtain an optimal policy π_*.

The input of the DQN, i.e state was a vector of size two, which included the current height and the set point, which was mapped onto the arbitrarily chosen hidden layers of size 2, each having the size of 64 neurons which is a default setting in the stablebaselines3. DQN, which was mapped on the output having dimension 2, which represents the Q-value for on and off action respectively.

Training was done with 400,000 timesteps(samples) on a cpu to obtain an optimal policy for a level controller.

3 RESULT

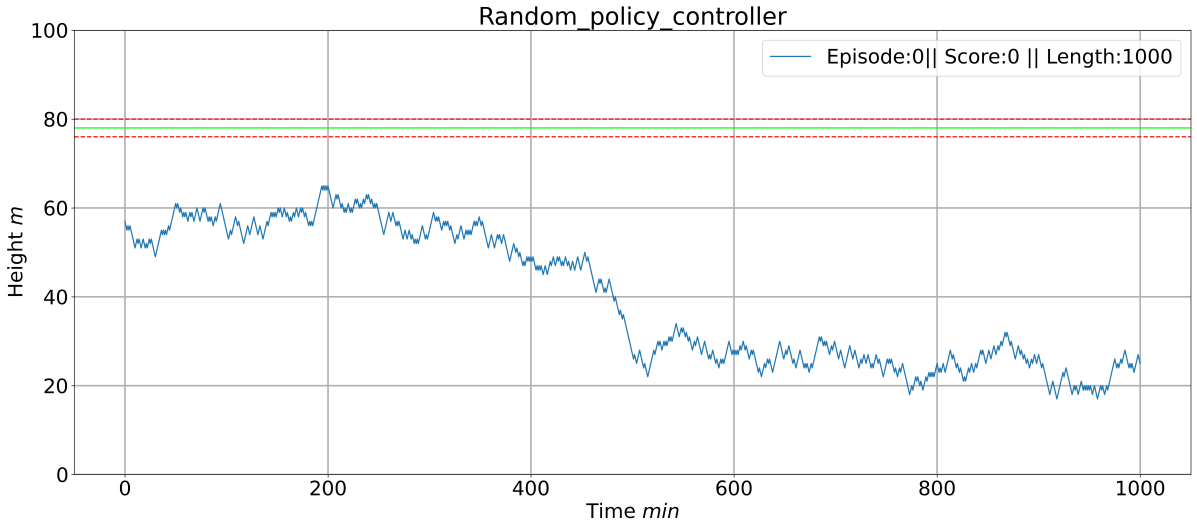

Figure 4: Random policy controller

The control policy when set at random action without any training was unable to maintain the set point and the total accumulated reward was only by pure chance. Without any training the policy isn't able to maintain the set point height. As observed on Fig 4 the random policy without any training wasn't able to maintain the height of the tank. The green line is the set point and the dashed red line above and below the set point line is the tolerance range where the agent is able to accumulate reward if the height measurement is able to reach the range. For figure 4 we see that the total Score was 0, i.e the total accumulated undiscounted reward for the episode was 0 when the total length of the simulation was 1000 minutes.

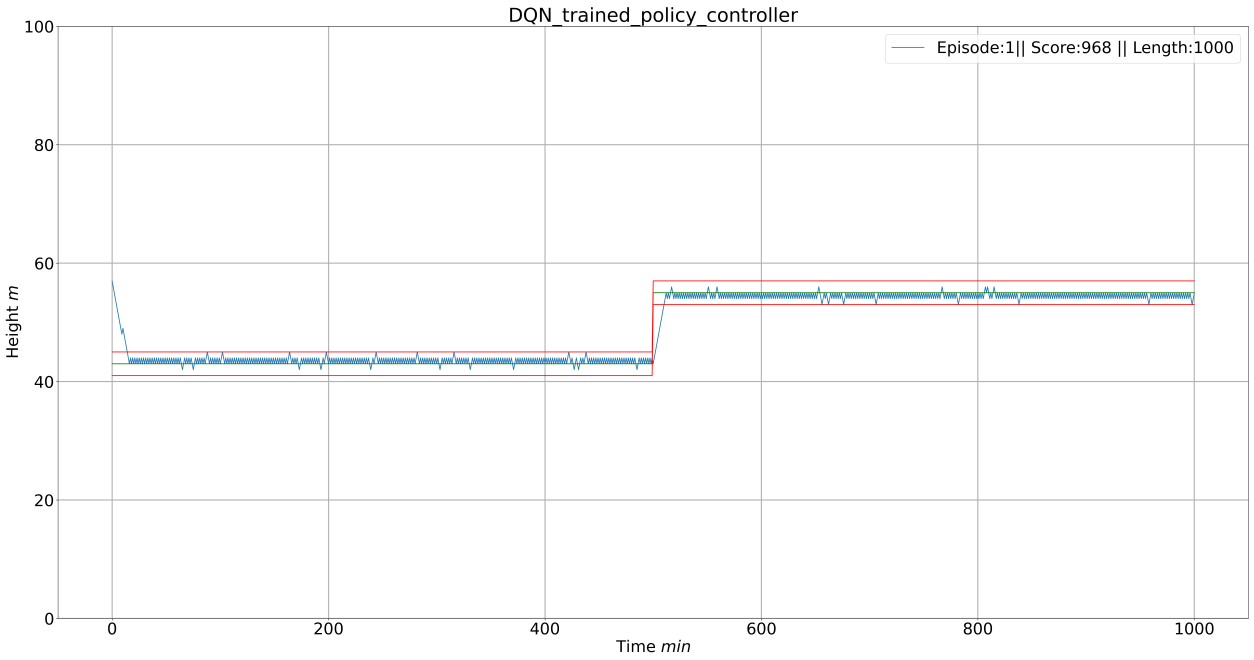

Figure 5: DQN trained policy controller with initial height 30 m

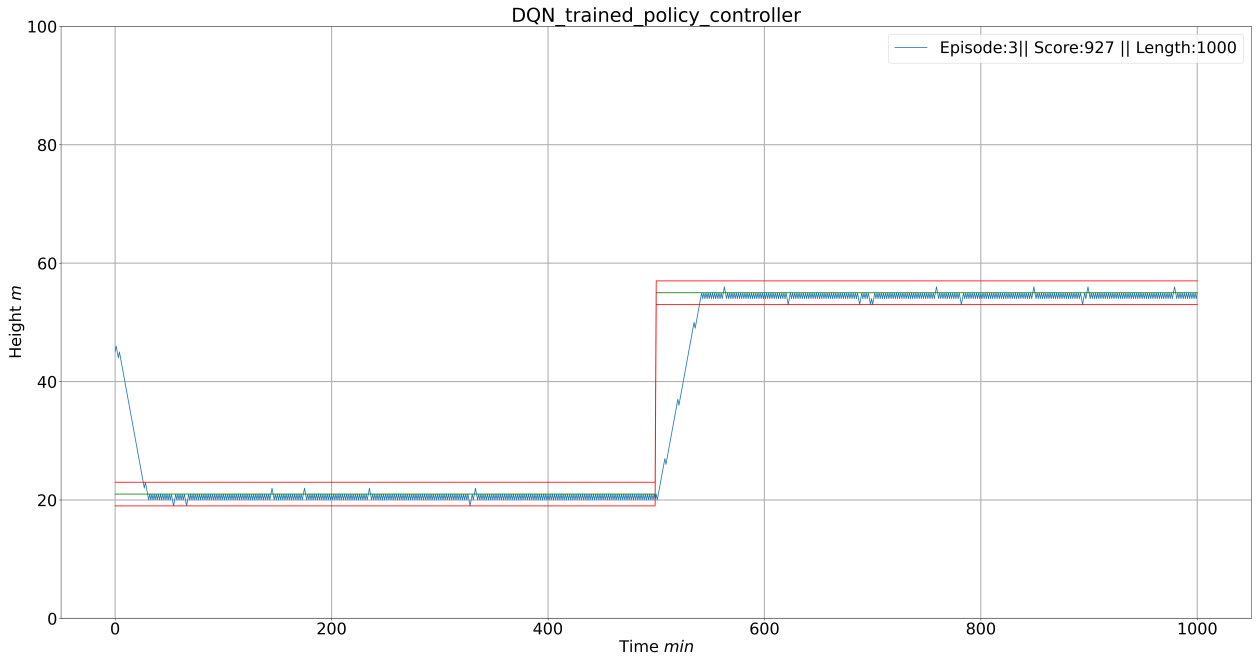

Figure 6: DQN trained policy controller with initial height 57 m

We observe that after 400,000 timesteps/samples of training the agent is able to increase the reward and maintain the level within the given setpoint's tolerance range. THe policy was trained on the default MlpPolicy from the stablebaselines3.DQN function where the hidden layer size is of 2 layers and each layer have 64 neurons and the activation used is rectified linear unit (ReLU). So the policy has 2 input layer, set point target and height measurement as input and the output of size 2 which are the action values of each actions(on or off) of the outlet flowrate. For better control of the hyperparameters and the neural network architecture we also have created the agent and the learning policy using pytorch and is available at the appendix(pytorchDQN.py and environment.py).

The results obtained for the DQN using stablebaselines3 are more stable without too much fluctuations in the total return and the policy was able to accumulate during the training process than the ones obtained form the pytorch implementation. The above results shown are from the DQN agent trained using stablebaselines3, which was able to maintain and control the tank height at the given set point after training and able to obtain total undiscounted reward greater than 900(968 and 927) during the performance testing phase as seen in figure 5 and figure 6.

4 CONCLUSION

While we used DQN for liquid level control successfully after training, we can use the same approach for designing the environment for other more complicated dynamic processes and can also design more arbitrary reward function which are not simply formulated as a set point problem but rather as process optimization(i.e desired product maximization, undesired product minimization, production/operating cost minimization,etc) problem. These methods are not limited by the type of inputs, states/inputs could not be only limited to sensor inputs like pressure, level, composition, temperature controller that are prevalent in process control but could also include sensor inputs like images or other arbitrary inputs that could be useful for determining an optimal policy could be easily employed[7].

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[2] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[3] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[4] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[5] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[6] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[7] Marc T Henry de Frahan, Nicholas T Wimer, Shashank Yellapantula, and Ray W Grout. Deep reinforcement learning for dynamic control of fuel injection timing in multi-pulse compression ignition engines. *International Journal of Engine Research*, 23(9):1503–1521, 2022.

Appendix

```
#{DQN_height_control.ipynb}
import numpy as np
from scipy.integrate import odeint
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
import gym
from gym import Env
from gym.spaces import Discrete, Box
import os
from stable_baselines3 import DQN
import random
A_tank=10 #m^2
qin=10 #m^3/min
qout=20 #m^3/min

def height_model(x, t, action=0):
    h = x[0]
    dhdt = (qin-action*qout)/A_tank
    return dhdt
class tank_env(Env):
    def __init__(self, set_point=None):
        self.action_space = Discrete(2)
        self.observation_space = Box(0, 100, shape=(2,))
        self.max_length = 1000
        if set_point==None:
            self.set_point = random.randint(20,80)
        else:
            self.set_point=set_point
        self.state = np.array([random.randint(20,80), self.set_point])

    def step(self, action):
        tn = np.linspace(0, 1, 2)
        sol = odeint(height_model, self.state[0], tn, args=(action,))
        self.state = np.array([sol[-1], self.set_point]).astype(dtype=float)
        self.max_length-=1
        reward=0
        if self.state[0]<=0.0 or self.state[0]>=100.0:
            done=True
        elif self.max_length<=0:
            done=True
        elif self.state[0]<=(self.set_point+2) and self.state[0]>=(self.set_point-2):
            done=False
            reward = + 1
```

```
            else:
                done=False
                reward = 0
            info={}
            return self.state, reward, done, info

        def render(self):
            pass
        def reset(self, set_point=None):
            self.state = np.array([random.randint(20,80), self.set_point]).astype(dtype=floa
            self.max_length = 1000
            if set_point==None:
                self.set_point = random.randint(20,80)
            else:
                self.set_point=set_point
            return self.state
    env = tank_env()

    episodes = 5
    for episode in range(episodes):
        height_data=[]
        state = env.reset()
        height_data.append(state[0])
        done = False
        score = 0
        length = 0
        while not done:
            length+=1
            action = env.action_space.sample()
            n_state, reward, done, info = env.step(action=action,)
            height_data.append(n_state[0])
            score+=reward

        plt.figure(figsize=(25,10))
        t = np.arange(0, len(height_data))
        plt.plot(t, height_data,label=f"Episode:{episode}|| Score:{score} || Length:{length}
        plt.ylim(0,100)
        plt.title("Random_policy_controller")
        plt.xlabel("Time $min$")
        plt.ylabel("Height $m$")
        plt.rc('font', size=25)
        plt.axhline(y=env.set_point,color="lime")
        plt.axhline(y=env.set_point+2, color="red", ls="--")
        plt.axhline(y=env.set_point-2, color="red", ls="--")
        plt.grid(which='both', linewidth=2)
```

```
    plt.legend()
    plt.savefig(f'Images/random_controller_Height_env{episode}.png', dpi=300)
    plt.show()

    print(f'Episode:{episode}|| Score:{score} || Length:{length} || Set_point1:{env.set_
env.close()
env= tank_env()
model = DQN.load("Saved_models/DQN_height_control", env=env)
episodes = 10
for episode in range(episodes):
    height_data=[]
    obs = env.reset()
    height_data.append(obs[0])
    done = False
    score = 0
    length = 0
    set_points = [0,0]
    set_points[0] = env.set_point
    while not done:
        length+=1
        action, _states = model.predict(obs)
        obs, reward, done, info = env.step(action)
        height_data.append(obs[0])
        score+=reward
        if length>=500:
            env.set_point = 55
            set_points[1] = env.set_point

    plt.figure(figsize=(40,20))
    t = np.arange(0, len(height_data))
    plt.plot(t, height_data,label=f"Episode:{episode}|| Score:{score} || Length:{length}
    plt.ylim(0,100)
    plt.title("DQN_trained_policy_controller")
    plt.xlabel("Time $min$")
    plt.ylabel("Height $m$")
    plt.rc('font', size=30)
    sp1=np.ones(500)*set_points[0]
    sp2=np.ones(len(height_data)-500)*set_points[1]
    sp=np.concatenate((sp1,sp2), axis=0)
    plt.plot(t,sp,color="green")
    plt.plot(t,sp-2,color="red")
    plt.plot(t,sp+2,color="red")
    plt.grid(which='both', linewidth=2)
    plt.legend()    plt.savefig(f'Images/DQN_policy_controller_Height_env{episode}.png',
    plt.show()
```

```
    print(f'Episode:{episode}|| Score:{score} || Length:{length}')
env.close()

#pytorch_DQN.py
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np

# not DDQN
class DeepQNetwork(nn.Module):
    def __init__(self, lr, input_dims, fc1_dims, fc2_dims, n_actions):
        super().__init__()
        self.input_dims = input_dims
        self.fc1_dims = fc1_dims
        self.fc2_dims = fc2_dims
        self.n_actions = n_actions
        self.fc1 = nn.Linear(*self.input_dims, self.fc1_dims) # what's *self.input_dims
        self.fc2 = nn.Linear(self.fc1_dims, self.fc2_dims)
        self.fc3 = nn.Linear(self.fc2_dims, self.n_actions)
        self.optimizer = optim.Adam(self.parameters(), lr=lr)
        self.loss = nn.MSELoss()
        self.device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

    def forward(self, state):
        x = F.relu(self.fc1(state.float()))
        x = F.relu(self.fc2(x))
        actions_q = self.fc3(x)

        return actions_q


class Agent():
    def __init__(self, gamma, epsilon, lr, input_dims, batch_size, n_actions, max_mem_si
        self.gamma = gamma
        self.epsilon = epsilon
        self.eps_min = eps_end
        self.eps_dec = eps_dec
        self.lr = lr
        self.action_space = [i for i in range(n_actions)]
        self.mem_size = max_mem_size
        self.batch_size = batch_size
        self.mem_counter = 0
```

```
        self.Q_eval = DeepQNetwork(self.lr, input_dims=input_dims, fc1_dims=256, fc2_dim

        self.state_memory = np.zeros((self.mem_size, *input_dims), dtype=np.float32)
        self.new_state_memory = np.zeros((self.mem_size, *input_dims), dtype=np.float32)
        self.action_memory = np.zeros(self.mem_size, dtype=np.int32)
        self.reward_memory = np.zeros(self.mem_size, dtype=np.float32)
        self.terminal_memory = np.zeros(self.mem_size, dtype=np.bool)

    def store_transition(self, state, action, reward, state_, done):
        index = self.mem_counter % self.mem_size # this is easier than popping the memo
        self.state_memory[index] = state
        self.new_state_memory[index] = state_
        self.reward_memory[index] = reward
        self.action_memory[index] = action
        self.terminal_memory[index] = done

        self.mem_counter += 1

    def choose_action(self, observation):
        if np.random.random() > self.epsilon:
            state = torch.tensor([observation]).to(self.Q_eval.device)
            actions_q = self.Q_eval.forward(state)
            action = torch.argmax(actions_q).item()

        else:
            action = np.random.choice(self.action_space)

        return action

    def learn(self):
        if self.mem_counter < self.batch_size:
            return

        self.Q_eval.optimizer.zero_grad()

        max_mem = min(self.mem_counter, self.mem_size)
        batch = np.random.choice(max_mem, self.batch_size, replace=False)

        batch_index = np.arange(self.batch_size, dtype=np.int32)

        state_batch = torch.tensor(self.state_memory[batch]).to(self.Q_eval.device)
        new_state_batch = torch.tensor(self.new_state_memory[batch]).to(self.Q_eval.devi
        reward_batch = torch.tensor(self.reward_memory[batch]).to(self.Q_eval.device)
        terminal_batch = torch.tensor(self.terminal_memory[batch]).to(self.Q_eval.device
```

```
        action_batch = self.action_memory[batch]

        q_eval = self.Q_eval.forward(state_batch)[batch_index, action_batch]
        q_next = self.Q_eval.forward(new_state_batch)
        q_next[terminal_batch] = 0.0

        q_target = reward_batch + self.gamma*torch.max(q_next, dim=1)[0]

        loss = self.Q_eval.loss(q_target, q_eval).to(self.Q_eval.device)
        loss.backward()
        self.Q_eval.optimizer.step()

        self.epsilon = self.epsilon - self.eps_dec if self.epsilon > self.eps_min else s
```

```
#environment.py
import gym
from pytorch_DQN import Agent
# from utils import plotLearning
import numpy as np
import numpy as np
from scipy.integrate import odeint
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
import gym
from gym import Env
from gym.spaces import Discrete, Box
import os
from stable_baselines3 import DQN
import random



if __name__=="__main__":
    # env = gym.make('LunarLander-v2')
    # env = gym.make('CartPole-v0')
    A_tank=10 #m^2
    qin=10 #m^3/min
    qout=20
    A_out_pipe=1
    g=9.8
    def height_model(x, t, action=0):
        h = x[0]
    #     dhdt = A_tank*(qin - action*A_out_pipe*(np.sqrt(2*g*h)))
        dhdt = (qin-action*qout)/A_tank
        return dhdt
```

```
class tank_env(Env):
    def __init__(self, set_point=None):
        self.action_space = Discrete(2)
        self.observation_space = Box(0, 100, shape=(2,))
        self.max_length = 1000
        if set_point==None:
            self.set_point = random.randint(20,80)
        else:
            self.set_point=set_point
        self.state = np.array([random.randint(20,80), self.set_point])

    def step(self, action):
        tn = np.linspace(0, 1, 2)
        sol = odeint(height_model, self.state[0], tn, args=(action,))
        self.state = np.array([sol[-1], self.set_point]).astype(dtype=float)
        self.max_length-=1
        reward=0
        if self.state[0]<=0.0 or self.state[0]>=100.0:
            done=True
        elif self.max_length<=0:
            done=True
        elif self.state[0]<=(self.set_point+2) and self.state[0]>=(self.set_point-2)
            done=False
            reward = + 1
        else:
            done=False
            reward = 0
        info={}
        return self.state, reward, done, info

    def render(self):
        pass
    def reset(self, set_point=None):
        self.state = np.array([random.randint(20,80), self.set_point]).astype(dtype=
        self.max_length = 1000
        if set_point==None:
            self.set_point = random.randint(20,80)
        else:
            self.set_point=set_point
        return self.state


env = tank_env()
```

```
agent = Agent(gamma=0.99, epsilon=1.0, batch_size=32, n_actions=2, eps_end=0.01, inp
scores, eps_history = [], []
n_games = 100

for i in range(n_games):
    score = 0
    done = False
    observation = env.reset()

    while not done:
        env.render()
        action = agent.choose_action(observation)
        observation_, reward, done, info = env.step(action)
        score+=reward
        agent.store_transition(observation, action, reward, observation_, done)
        agent.learn()
        observation = observation_
    scores.append(score)
    eps_history.append(agent.epsilon)

    avg_score = np.mean(scores[-100:])

    print(f'episode{i}, scores{score}, average_score{avg_score}, epsilon{agent.epsil

x = [i+1 for i in range(n_games)]
```