BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

APPLICATION OF REINFORCEMENT LEARNING IN SMART HOME MANAGEMENT

APLIKACE POSILOVANÉHO UČENÍ V ŘÍZENÍ SMART HOME

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **GABRIEL BIEL**
AUTOR PRÁCE

SUPERVISOR **doc. Ing. VLADIMÍR JANOUŠEK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

155030

Ústav:	Ústav inteligentních systémů (UITS)
Student:	**Biel Gabriel**
Program:	Informační technologie
Název:	**Aplikace posilovaného učení v řízení Smart Home**
Kategorie:	Umělá inteligence
Akademický rok:	2023/24

Zadání:

1. Prostudujte problematiku posilovaného učení a jeho aplikací.
2. Seznamte se metodami a nástroji pro řízení Smart Home (např. Home Assistant) a s dostupnými senzory a možnostmi interakce uživatele se systémem.
3. Pro vhodný subsystém Smart Home navrhněte inteligentní řízení (agenta), které bere v úvahu interakce uživatele se systémem a učí se mu vycházet vstříc tím, že směřuje k minimalizaci interakcí typu "nevyhovuje mi aktuální nastavení teploty" apod. Aplikujte zde vhodným způsobem princip posilovaného učení nebo se tímto principem inspirujte.
4. Navržený inteligentní systém řízení realizujte za použití vhodných prostředků propojitelných se simulovaným i reálným prostředím.
5. Realizovaný systém řízení otestujte v reálném i v simulovaném prostředí. Vyhodnoťte dosažené výsledky.

Literatura:
Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:
První 3 body zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Janoušek Vladimír, doc. Ing., Ph.D.**
Vedoucí ústavu:	Hanáček Petr, doc. Dr. Ing.
Datum zadání:	1.11.2023
Termín pro odevzdání:	9.5.2024
Datum schválení:	6.11.2023

Abstract

This thesis investigates how machine learning can improve smart home management by focusing on optimizing temperature control and boosting energy efficiency. Specifically, it examines and compares two sophisticated reinforcement learning algorithms, Deep Q-Learning (DQL) and Proximal Policy Optimization (PPO). These models are tested in a simulated environment that replicates real-world conditions to evaluate their effectiveness in adapting to user behaviors and environmental changes. The study finds that the PPO model is particularly effective due to its stability and ability to predict when occupants will return, thus maintaining a comfortable temperature more efficiently. This research offers valuable insights into the practical applications of AI technologies in smart homes.

Abstrakt

Táto práca skúma, ako môže strojové učenie zlepšiť riadenie inteligentných domácností s dôrazom na optimalizáciu riadenia teploty a zvýšenie energetickej účinnosti. Konkrétne sa porovnávajú dva pokročilé algoritmy posilňovaného učenia, Deep Q-Learning (DQL) a Proximal Policy Optimization (PPO). Tieto modely sú testované v simulovanom prostredí, ktoré napodobňuje reálne podmienky, aby sa zhodnotila ich schopnosť prispôsobiť sa správaniam užívateľov a zmenám v prostredí. Ukázalo sa, že model PPO je obzvlášť účinný vďaka svojej stabilite a schopnosti predpovedať návrat obyvateľov. Tento výskum ponúka cenné poznatky o praktických aplikáciách AI technológií v inteligentných domácnostiach.

Keywords

machine learning, smart home, temperature control, reinforcement learning, deep q-learning, dql, dqn, proximal policy optimization, ppo, predictive models

Klíčová slova

strojové učenie, inteligentné domácnosti, regulácia teploty, posilňované učenie, deep q-learning, dql, dqn, proximal policy optimization, ppo, prediktívne modely

Reference

BIEL, Gabriel. *Application of Reinforcement Learning in Smart Home Management.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Vladimír Janoušek, Ph.D.

Application of Reinforcement Learning in Smart Home Management

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of doc. Ing Vladimír Janoušek Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Gabriel Biel
May 8, 2024

Acknowledgements

My sincere gratitude to my supervisor, doc. Ing Vladimír Janoušek Ph.D., for all of his helpful advice and insightful remarks.

Contents

List of Figures

Chapter 1

Introduction

Home is more than just a place to live in our daily lives. It is a personal territory where everything can be adjusted according to our own needs to feel safe and comfortable. The rapid development of technology has allowed such a concept as "smart homes" to become its integral part. The use of intelligent systems in the living space has become not a luxury but often a necessity. Such houses can guarantee residents not only comfortable living conditions but also energy savings.

Smart home technology has evolved from simple automated tasks to advanced systems that can predict human behavior to improve their functions. An important factor in this growth has been the emergence of the Internet of Things (IoT), which has provided the infrastructure needed for data collection and advanced remote control features. Although smart home management has advanced significantly with existing solutions, there is still room for growth in terms of houses' capacity to learn and easily integrate into dynamic, effective systems.

The primary goal of this thesis is to explore the application of reinforcement learning techniques to improve smart home management, specifically focusing on the optimization of temperature regulation. This research delves into the development and evaluation of two advanced machine learning models: Deep Q-Learning (DQL) and Proximal Policy Optimization (PPO). These models are suppose to maintain comfortable indoor temperatures and enhance energy efficiency effectively across a range of home environments. A key aspect of these models is their ability to dynamically adjust to changes in user behavior and environmental conditions, as well as their capability to anticipate the return of occupants, ensuring that optimal conditions are met upon their arrival.

These objectives are tested within a sophisticated simulation environment that mimics the dynamics of a real-world environment. The models' performance is thoroughly analyzed, focusing on their efficiency in energy management, comfort maintenance, and their agility in adapting to and predicting changes within the home environment. This investigation not only provides a detailed comparative analysis of the capabilities and challenges faced by each model but also provide insights into the practical implications of deploying such technologies in smart homes.

I am particularly interested in this subject because I am fascinated by the positive possibilities of AI on our typical daily lives. The cooperation of artificial intelligence algorithms and home automation can alter the way we interact with the locations where we usually spend the most time. We can make homes smarter by allowing them to adapt to our needs and requirements without a specific set of instructions.

The structure of this thesis is crafted to guide the reader through the complexities of applying AI to smart homes. Chapter 2 begins with key terminology and reviews current smart home automation solutions. Chapter 3 discusses the theoretical framework for the chosen reinforcement learning models and their selection rationale. Chapter 4 details the goals and methodology for implementing these models in a simulated environment. Chapters 5 and 6 cover the implementation of the simulation environment and algorithms, respectively. Finally, Chapter 7 evaluates the models' performance and approaches.

Through this structured exploration, the thesis aims to contribute to the ongoing development of smart home technology, offering a pathway to more sustainable and personalized home environments.

Chapter 2

Smart Home

This thesis is situated within the context of smart homes, environments where automation and connectivity enhance residential life. This chapter defines what smart homes are, the specific behaviors desired from these systems, and how they use sensor data to manage internal conditions effectively.

2.1 What Smart Home Represents

Smart homes are designed to autonomously manage various household functions, thereby reducing the manual tasks typically required of occupants. This automation extends across several domains, from security systems to energy management, aiming to increase both the convenience and efficiency of household operations. [15].

In smart homes, just like a human observes before interacting, the system itself requires sensory input to make informed decisions. These decisions are based on data collected from a network of sensors that monitor different environmental parameters. The objective is to create a living space that not only reacts to the changes but also possibly anticipates the needs of its occupants to optimize both comfort and energy usage.

2.2 Sensory Data for Temperature Regulation

The focus of this thesis is on temperature regulation within smart homes, aiming to ensure continuous comfort and maintain energy efficiency. To effectively mimic a human caretaker's decisions regarding temperature management, a smart home system requires access to specific critical pieces of information.

There are many factors that influence indoor temperature, such as sunlight exposure, window and door openings, humidity, or heat emitted by appliances and occupants. But for simplicity, this thesis focuses on the most controllable and measurable factors. For example, sunlight exposure can significantly affect room temperatures but is challenging to predict and manage due to its variability. Likewise, monitoring heat from appliances and occupants is theoretically possible but impractical due to the added complexity and minimal practical benefit. Instead, the system should be able to operate effectively with some level of noise. So, the essential data points for temperature regulation might include:

- Current indoor and outdoor temperatures,
- Occupancy status of the home,

- Operational status of the heating system.

By focusing on these data, the system can make effective and practical adjustments. This streamlined approach aids in straightforward implementation while capturing the most significant factors affecting energy consumption and temperature dynamics. The goal is to develop a system that is effective in its function and practical in terms of cost and complexity, achieving an optimal balance between between precision and practicality.

2.2.1 Role of Internet of Things

In order to further enhance the performance of smart homes, the Internet of Things technologies are integrated into the system. IoT brings the connectivity and intelligence necessary for collecting and using sensory information when installing certain pieces of software. In this sense, every-day objects and devices, such as sensors and actuators, can send and receive data over the internet. The IoT devices are integral components of smart homes and transmit data to one centralized system that processes the data in them and sends feedback informing the activities to undertake for home management at that particular time not only does it optimize home operational processes but also dramatically enhances user convenience and energy management [2].

2.3 Desired Behavior

Since the primary goal of smart home temperature regulation is to strike a balance between energy efficiency and occupant comfort, the system needs to anticipate the occupants' needs based on typical patterns and adjust temperatures accordingly.

Figure 2.1 depicts a desired pattern of temperature adjustment in response to changes in occupancy, optimizing both comfort and energy usage. Graph is color-coded, with different colors representing various operational modes:

- Blue indicates periods where the system maintains optimal comfort temperatures.
- Gray signifies times when the target is energy savings.
- Red shows when the system is actively increasing the temperature to reach comfort levels.
- Vertical dotted lines mark changes in occupancy, highlighting the system's reactive adjustments.

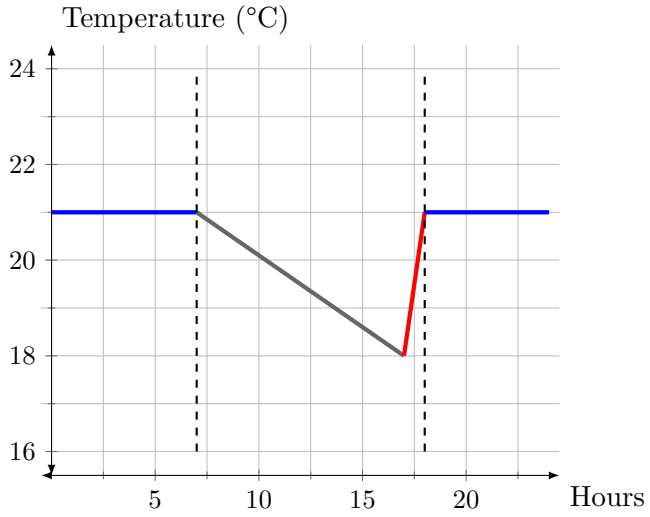

Figure 2.1: Desired temperature regulation over time in response to occupancy changes, showing different operational modes for optimal energy use and comfort.

2.4 Current Solutions for Temperature Management

Current solutions for managing home temperatures incorporate a range of technologies, from basic programmable devices to advanced systems for adaptive control. This section explores different approaches, outlining their functionalities, limitations, and how they contribute to energy efficiency and occupant comfort.

2.4.1 Programmable Thermostats

With most programmable thermostats, a user is allowed to identify varying temperatures during times of the day when an occupant is predicted to be away or sleeping. Despite achieving incredible energy savings, these devices continue working on the set schedule and do not adjust to varying conditions such as full occupancy or certain weather conditions. For example, if a user decided to adjust the temperature at the periods when occupants were still scheduled to leave or in the office working, the temperatures would not be adjusted until it was the next time on the schedule [6].

2.4.2 Learning Systems

More sophisticated than programmable thermostats, learning systems such as the Nest Thermostat [13] or [9] use algorithms to learn from occupant behaviors, adjusting the heating and cooling settings to match daily patterns. These systems observe the times when occupants are typically at home and adjust temperatures accordingly. They can eventually learn autonomously to adjust to changes in the occupants' schedules, such as a variation in the work pattern or some time off, etc. Nonetheless, these systems generally involve the recognition of patterns over time, and they might not be able to react appropriately to sudden and unofficial changes in occupancy or activity, including an unscheduled return from work or a sporadic party.

2.4.3 Predictive and Adaptive Systems

Beyond programmable and learning thermostats, predictive and adaptive systems represent an even more advanced stage of smart temperature management. Systems such as PSTC [28], integrate multiple data sources, including real-time weather forecasting and advanced occupancy sensors, to dynamically adjust heating and cooling operations. Such systems are capable of anticipating changes rather than merely reacting to them, potentially offering the highest level of energy efficiency and comfort customization. For example, a system might start heating the home in anticipation of a snowstorm predicted to hit just as the homeowner returns, or cool down more aggressively if it detects an increase in indoor occupancy unexpectedly.

However, despite their advanced capabilities, these systems have significant drawbacks. The complexity and cost of installation and maintenance can be inaccessible. Additionally, their reliance on accurate and timely data from external sources like weather forecasts, which can be unreliable, introduces operational risks. Errors in data interpretation may lead to inefficient temperature adjustments, potentially increasing energy use instead of reducing it. Moreover, the required internet connectivity raises serious privacy and security concerns due to continuous data collection.

Each of these approaches presents strengths and weaknesses in the context of smart home temperature management. While programmable and learning thermostats offer improvements over manual controls, predictive systems represent the cutting edge of technology with the capability to truly optimize energy use and comfort through dynamic adjustments.

2.5 Rationale for Using Reinforcement Learning

Employing Reinforcement Learning (RL) in smart home temperature regulation presents a promising avenue to overcome the limitations of current systems. RL offers the capability to learn and adapt from ongoing interactions with the environment, promising a more flexible and efficient approach.

RL algorithms are capable of handling environments where conditions are in constant flux, making them ideal for smart homes where both external weather conditions and occupancy can vary significantly. An RL agent can process data from various sensors to make informed decisions, continuously refining its strategies to optimize both energy efficiency and comfort. For example, by learning from patterns of occupancy and external temperatures, an RL system might learn adjust the home's temperature in advance, ensuring comfort upon occupants' return while minimizing energy usage. Let's try to achieve just that!

Chapter 3

Reinforcement Learning

Having defined the Smart Home setting, this chapter shifts focus to the reinforcement learning (RL) components that underpin the decision-making processes within this environment. Here, we first explore the fundamental concepts of RL, laying the groundwork for a deeper understanding of how agents interact with their surroundings, make decisions, and learn from their experiences. This exploration covers the essential elements of the environment, the agents operating within it, and the dynamics of their interactions through states, actions, and rewards.

Following the foundational overview, the chapter delves into specific algorithms that are going to be further examined and compared—Deep Q-Learning (DQL) and Proximal Policy Optimization (PPO). These sections will detail the theoretical bases, architectural considerations, and unique attributes of each algorithm. The discussion aims to provide a clearer understanding of these algorithms and their parts, in preparation for their practical implementation and evaluation.

3.1 Fundamental Concept

This section explains the fundamental components and principles of reinforcement learning, along with key terms that will be referenced throughout the thesis.

3.1.1 Environment

In reinforcement learning, the environment is the environment in which the agent functions. It encompasses all the investment agents interact with adaptively and is essentially characterized by a set of states and events that explicitly define the dynamics of the surroundings. The environment is essential from the outset as it influences the agent's point of view and disciplines what actions can be taken. [30].

State, Action

A state, in general, represents the current perspective of the environment. Specifically, a state can be seen as a particular, tangible and up-to-date interpretation, defined by all considerations that should be taken into consideration which describe the current environment. Actions, however, are all the feasible judgments or moves the agent can do with due respect to the state of the environment. The relationship of States with Actions is

a lingering pattern in which the action performed on a specific state triggers self-directed changes [30].

Reward

The concept of reward is another cornerstone of reinforcement learning since it serves as feedback from the environment. It is the instant numerical return the agent receives after performing an action in a given state. It acts as an indicator to establish how successful or unsuccessful that action was. The reward structure is crucial since it shapes how the agent learns and behaves; it is an essential component of the learning process since it dictates the agent's actions. Adequate rewarding signals allow the agent to differentiate between actions that benefit it and those that are detrimental to quickly learn the effective behaviors that maximize the sum of rewards it can get in the long run [30].

3.1.2 Agent

Having specified the environment where the agent operates, attention is now turned to the agent itself. The agent in reinforcement learning interacts with the environment by perceiving states, performing actions, and receiving rewards. This interaction is driven by internal mechanisms such as neural networks and policies, which dictate how the agent learns and decides [30].

Policy

Simply said, the policy π is dictating the course of actions to be taken in given states. It can be deterministic, where a specific action is chosen for a given state, or stochastic, where actions are selected based on a probability distribution. The policy guides the agent's decisions by leveraging the outputs from the neural network to enhance the likelihood of achieving higher rewards. The development and refinement of policies are central to improving the agent's performance in navigating the environment [30].

Formally, if A is the set of all actions and S is the set of all states, then the policy $\pi(a|s)$ gives the probability of taking action $a \in A$ when in state $s \in S$. The ultimate goal of an RL agent is to discover an optimal policy π^* that maximizes the expected cumulative reward over time, which can be mathematically expressed as:

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t | \pi\right], \tag{3.1}$$

where:

- $\arg\max_\pi$ denotes the operation of finding the policy π that results in the maximum value of the expected reward calculation. This operation is known as the „argument of the maximum."
- $\mathbb{E}[\cdot]$ symbolizes the expected value of the expression enclosed. In this context, it calculates the average outcome over many possible sequences of states and actions, given the stochastic nature of the policy and the environment.
- $\sum_{t=0}^{\infty} \gamma^t R_t$ is the sum of discounted rewards, extending from the current time step $t = 0$ to infinity. This sum represents the total reward expected to be accumulated over the future, reflecting all rewards R_t received at each time step.

- γ^t is the discount factor raised to the power of the time step t. The discount factor, a value between 0 and 1, determines the present value of future rewards. Rewards received further in the future are weighted less heavily, making γ^t decrease as t increases.
- R_t is the reward received at time t, determined by the environment's response to the agent's actions.

This formulation seeks to find the policy π that maximizes the total expected discounted reward over an infinite timeline, enabling the agent to make decisions that optimally balance immediate and future rewards.

3.1.3 Value Functions: Q-values and V-values

Value functions are fundamental in reinforcement learning and serve as a cornerstone for both policy development and decision-making processes. The following subsection will investigate the scope and definition of Q-values and V-values.

Q-values (Action-Value Functions)

Q-values, or action-value functions, estimate the expected utility of taking a given action in a given state and following a specific policy thereafter. The Q-value for a state-action pair (s, a) can be formally defined as the expected return starting from state s, taking the action a, and thereafter following policy π. Q-values help in determining the best action to take in a state by comparing the values of possible actions from that state [30].

V-values (State-Value Functions)

V-values, or state-value functions, represent the expected return for a state under a specific policy. For any state s, the V-value is the expected return of starting in s and following policy π thereafter. V-values are particularly useful for evaluating the goodness of states and are often used in conjunction with Q-values to inform policy improvements and action selection [30].

Bellman Equation

The Bellman equation is fundamental to many RL algorithms, particularly DQL, which is discussed in the following section. It provides a recursive decomposition for the value function, without which the optimal policy and value function are hardly calculable. Basically, in Q-learning, this equation is used to update the Q-values in an updating step, which is essential in assuring convergence towards the optimal policy. The equation is as follows:

$$Q(s, a) = R(s, a) + \gamma \max_{a'} Q(s', a'), \tag{3.2}$$

where $R(s, a)$ is the reward for doing action a in state s, and γ is the discount factor, and s' is the new state after taking action a. This enables the agent to weigh in the expected utility of its actions and hence choose its course of action that will ensure maximization of cumulative future rewards [30].

3.1.4 Learning process

The agent regards the environment, then selects an action that the agent believes will lead to the best outcomes. The environment subsequently returns an observation and a numerical reward following each action taken. The numerical reward specifies how well the action was executed in relation to the agent's goals. The whole process of observation, action, and reward is shown in Figure 3.1 as a continual interaction loop between the two.

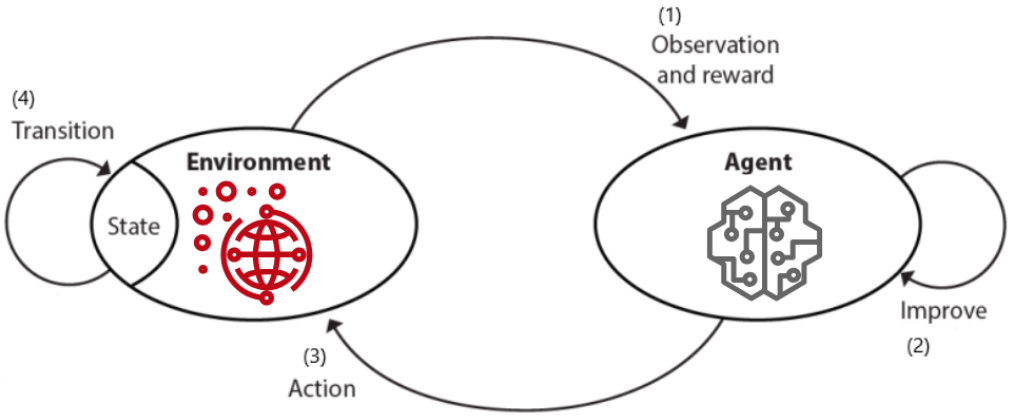

Figure 3.1: The interplay between the agent and the environment through states, actions, and rewards. Inspired by: [22]

In this basic interaction cycle of reinforcement learning, the agent initiates the cycle by attentively observing the environment (1), a critical step that informs its decision-making policy. With observations and past rewards, the agent then engages in a process of self-improvement (2), continuously trying to adapt to the task at hand. With a refined policy, the agent proceeds to take an action (3) towards the environment, aiming to steer the situation to its advantage. The environment responds to this intervention by transitioning into a new state (4), often causing a change in the internal state due to the agent's actions and the dynamics of the environment. This new state sets the stage for the next observational input to the agent, continuing the RL cycle. Through this ongoing loop of observe-improve-act-respond, the agent iteratively enhances its ability to act optimally within the environment, embodying the essence of learning through iterative experience.

Exploration and Exploitation

Moreover, following the same example, in accordance with the basic interaction process of agent and environment, the process of learning must fully address the trade-off between exploration and exploitation. As the agent learned from a set of episodes, the agent must be able to set the balance between trying new actions until it finds the optimal action which is the exploration, with the ratio of trying good as the good actions and executing bad actions as the bad actions, and exploiting known knowledge with the set of model in DRL itself to make decision based on current information.

The epsilon-greedy algorithm provides a means of balancing exploration with exploitation in reinforcement learning. With probability epsilon, the agent randomly picks an

action to explore unfamiliar options, while with probability one minus epsilon, it selects the best known choice based on prior experiences. In the early stages of learning, epsilon is set high to encourage diverse experimentation and discovery of the environment's dynamics. As training progresses though, epsilon is gradually decreased, placing heavier weight on applying accumulated insights. Through this adjustable approach, the learner continuously refines its developing understanding through both cautious exploitation and curious exploration, optimizing its selections while preventing hasty conclusions or stagnation. [30].

The dynamic adjustment of ϵ helps the agent to avoid local optima early in the learning process and to refine its strategy as it gains more experience. This method is particularly effective in environments where the optimal policy is not initially known and exploration can lead to significant improvements in performance.

3.1.5 Neural Network

Neural networks underlie many cutting-edge reinforcement learning systems, especially those operating in intricate or high-dimensional environments. These web-like computations adeptly synthesize functions key to rational decision-making, like importance ratings and behavioral blueprints [7].

Neurons and Network Architecture

A neural network contains processing elements called neurons, each acting independently as a simple calculating unit. Neurons accept inputs weighted by coefficients and produce an output through a nonlinear triggering rule. The arrangement of the system—how neurons are stacked and linked—notably affects its capacity to model complex correlations and interdependencies [14]. Connections between neurons strengthen or weaken as the network trains, developing an internal representation of the problem domain. With sufficient data, neural models can approximate nearly any discriminative or generative process.

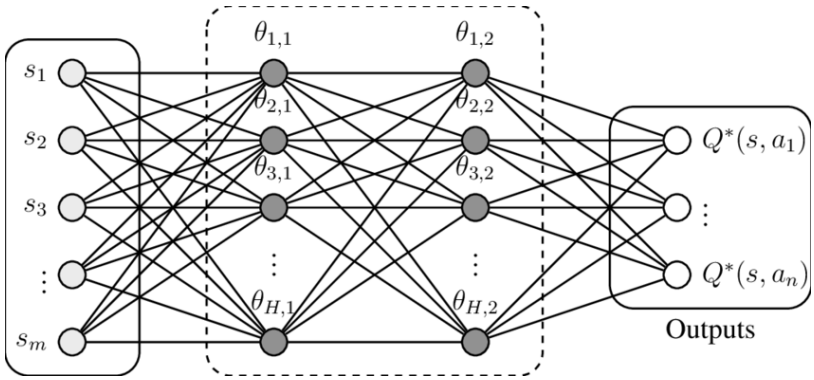

Figure 3.2: Architecture of a deep Q-network with state inputs and Q-value outputs for each possible action. Source: [19]

Activation Function

Activation functions play an integral role in neural networks by determining whether neurons fire or remain inactive. They introduce non-linear properties to neuron outputs, allowing networks to solve more intricate problems than simple linear prediction. Common

activation functions encompass the sigmoid, which caps outputs between 0 and 1 for binary classification, and the Rectified Linear Unit. ReLU immediately returns positive inputs while zeroing negatives, streamlining calculations and hastening convergence during training. Though efficient, the sigmoid sees less use due to ReLU excelling in speeding the learning of complex patterns from data. Networks leverage diverse activation functions to both classify and make sense of our richly nuanced world [7].

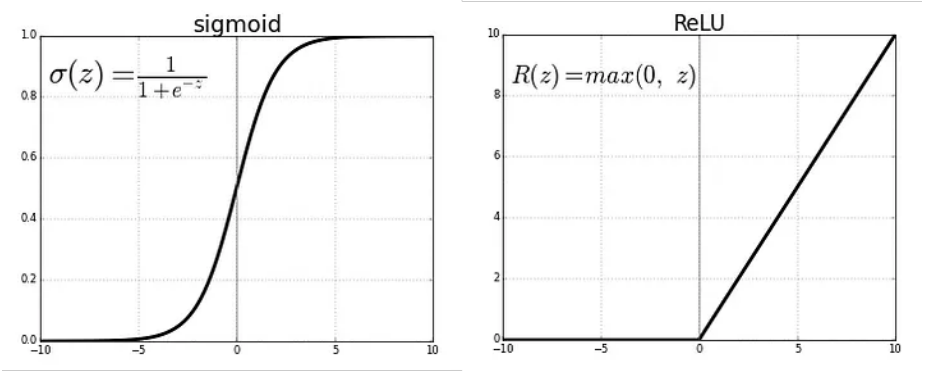

Figure 3.3: ReLU vs Sigmoid. Source: [27]

Forward Propagation and Activation

During forward propagation, inputs to the network are processed through successive layers, where each neuron's output becomes the input for the next layer. The output of each neuron is computed as a weighted sum of its inputs, transformed by a nonlinear activation function such as ReLU or sigmoid mention in the previous subsection. This nonlinearity is essential for the network's ability to capture complex patterns in data [7].

Backpropagation and Learning

Backpropagation is the principal method by which neural networks learn. It involves computing the gradient of the loss function— a measure of prediction error— with respect to each weight in the network. This gradient is then used to update the weights in a direction that minimizes the error, thereby improving the network's predictions over iterations [25].

Loss Functions and Optimization

The choice of a loss function and an optimization algorithm plays a crucial role in how neural networks learn from data. As for loss function, a Mean Squared Error (MSE) is often used when the task involves predicting continuous values. MSE measures how far off predictions are from actual outcomes, emphasizing larger errors more heavily than smaller ones, which helps in refining the accuracy of predictions [7].

Stochastic Gradient Descent (SGD) is a fundamental optimization algorithm that updates the model's weights incrementally, using a subset of data at each step. This method is efficient and can navigate the model toward optimal performance by minimizing the loss, such as MSE, over several iterations [3].

Adam optimization builds on the idea of SGD but adjusts the amount each weight is updated by calculating individual learning rates for different features. This helps in

converging faster and more effectively, especially in scenarios with large datasets or many parameters [11].

Role of Batches in Network Learning

Training neural networks involves several challenges, such as overfitting, where a model performs well on training data but poorly on unseen data. Solutions like regularization, dropout, and batch normalization are used to mitigate these issues, promoting generalization and improving the network's performance on new, unseen data [29].

3.1.6 Different Approaches to Learning

Understanding different learning approaches in reinforcement learning is crucial for designing agents that can adapt to various environments and tasks. These approaches directly influence the development and performance of learning algorithms.

On and Off Policy

On-policy methods involve the agent learning and refining a policy that it is concurrently executing. This means that the agent learns from the experiences gathered directly as a result of its actions. The policy is updated in a way that integrates new findings while attempting to perform optimally based on current knowledge. Proximal Policy Optimization (PPO) is an example of an on-policy method, where the objective function used for updating the policy is modified to maintain a balance between exploration and exploitation, helping to prevent excessively large policy updates that could destabilize the learning process [26].

Off-policy methods, on the other hand, allow the agent to learn an optimal policy that is decoupled from the policy it follows while exploring the environment. This approach enables the agent to learn from hypothetical decisions that it hasn't actually made, expanding its understanding and optimizing its strategy beyond the constrained scope of its current actions. Deep Q-Learning (DQL) exemplifies off-policy learning, where the agent uses a Q-function to evaluate the expected rewards of actions taken in various states. This function is continuously updated based on observed outcomes and hypothetical scenarios, guiding the agent toward more effective strategies over time [21].

These distinct approaches illustrate the versatility and depth of strategies available in reinforcement learning, allowing algorithms to be tailored to various types of tasks and environments. Each method has its advantages and suitable applications, depending on the specific requirements and goals of the reinforcement learning system.

3.1.7 Different Types of Agents

Figure 3.4 provides a schematic overview of these agent categories, illustrating the distinctions based on their reliance on models, policies, and value functions. This visual representation aids in understanding the foundational structure that underpins the various learning algorithms and their respective methodologies for policy derivation and value estimation.

Model-based vs. Model-free Agents

Model-based Agents: Model-based agents construct and maintain an internal representation of the environment, which they utilize to simulate and plan future actions. This

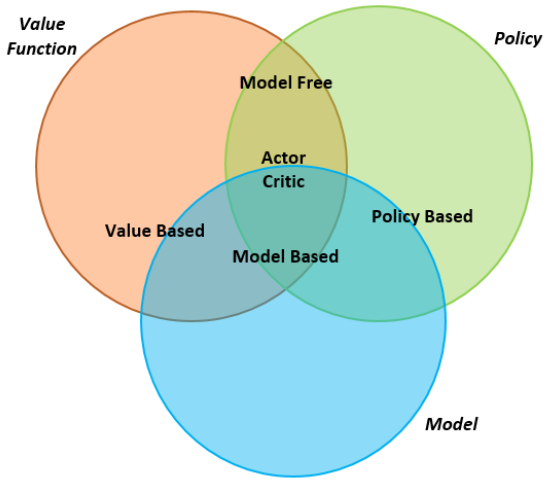

Figure 3.4: Venn diagram depicting the different RL methodologies.

internal model predicts the outcomes of potential actions, providing a foresight that is used to make informed decisions and learn effective policies with less trial-and-error interaction with the actual environment [10].

Model-free Agents: In contrast, model-free agents do not possess a model of the environment. Instead, they operate based on the principle of learning through direct experience. These agents learn the value of actions or the policy itself from each encountered state, relying on a historical record of rewards to inform their decision-making process without forecasting the specific consequences of their actions [30].

Value-based vs. Policy-based Agents

Value-based Agents: Value-based agents aim to estimate the value function, which represents the expected return of taking certain actions in specific states. The value function $V(s)$ for a state s or the action-value function $Q(s, a)$ for a state-action pair (s, a) are central to these methods. The action-value function, for instance, can be defined as:

$$Q(s, a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} Q(s', a')|s, a] \tag{3.3}$$

where R_{t+1} is the reward after taking action a in state s, γ is the discount factor, and s' is the subsequent state. Agents typically choose actions with the highest value, following a policy derived from $Q(s, a)$:

$$\pi(s) = \arg\max_{a} Q(s, a) \tag{3.4}$$

[30].

Policy-based Agents: Policy-based agents directly learn the policy $\pi(a|s)$, defining the likelihood of choosing action a in state s. This method is especially useful in environments

that have a large or continuous range of possible actions. To enhance the policy, gradient ascent is applied to maximize the expected return:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_\pi[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s,a)] \tag{3.5}$$

where:

- θ represents the parameters of the policy π.
- $J(\pi_\theta)$ is the objective function to be maximized, representing the expected return of following policy π parameterized by θ.
- $\nabla_\theta \log \pi_\theta(a|s)$ is the gradient of the logarithm of the policy probability, indicating how sensitive the probability of selecting action a in state s is to changes in the parameters θ.
- $Q^\pi(s,a)$ is the action-value function under policy π, estimating the expected return starting from state s, taking action a, and thereafter following policy π.
- $\mathbb{E}_\pi[\cdot]$ denotes the expected value over the distribution of actions and states as determined by policy π.

Use of Log Probabilities: Using logarithms of probabilities in policy gradient methods addresses numerical challenges associated with direct probability manipulations. Probabilities can be extremely small and prone to underflow, where very low values are rounded to zero, causing computational instability. The logarithmic transformation converts multiplicative probability operations into additive ones, enhancing numerical stability and simplifying the gradient computations. This approach ensures that even small probabilities significantly influence the learning process, crucial for environments with complex action spaces[23].

Actor-Critic: Actor-critic methods bridge value-based and policy-based strategies by using two components: an actor that suggests actions based on a policy $\pi(a|s)$, and a critic that evaluates these actions with a value function. This dual structure allows for simultaneous policy improvement and value estimation [12].

The mathematical foundation and detailed explanation of actor-critic methods are discussed in section 3.3.1.

3.2 Deep Q-Learning (DQL)

This thesis will implement and analyze the DQL algorithm, so let's start with an exploration of its basic principles.

Deep Q-learning has been established as one of the most remarkable methods in the history of RL algorithms. It is an off-policy algorithm that builds upon the traditional Q-learning algorithm by making use of deep neural networks for the representation of the Q-function. This advancement is the most crucial part that enables an agent to effectively deal with the environment of high-dimensional state space, an intrinsic aspect in complex or realistic settings [21].

3.2.1 Theoretical Foundations

Q-Learning Basics

Q-learning is a model-free off-policy algorithm used in reinforcement learning to find the optimal action-selection policy using a Q-function. The Q-function at its core, denoted as $Q(s, a)$, estimates the expected utility of taking an action a in a given state s and following a certain policy π. The core of the Q-learning algorithm updates the Q-values based on the equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)], \tag{3.6}$$

where α is the learning rate, γ is the discount factor, and r_{t+1} is the reward received after taking action a_t in state s_t. This update rule uses the Bellman equation to iteratively converge to the optimal Q-values, which can then dictate the best policy π^* by choosing the action with the highest Q-value in each state [30].

Exploration versus exploitation is a critical dilemma in Q-learning, where the agent must choose between exploring new actions to improve future rewards and exploiting known actions to maximize current rewards. This balance is often managed through strategies like ϵ-greedy, where ϵ represents the probability of choosing a random action, encouraging exploration.

Deep Learning Integration

DQL enhances traditional Q-learning by integrating neural networks to approximate the Q-function, represented as $Q(s, a; \theta)$ where θ are the neural network parameters. This approach tackles the scalability challenges of traditional Q-learning, which struggles with high-dimensional state spaces typical in applications like video games or robotics.

Neural networks enable DQL to generalize across vast state spaces efficiently, bypassing the need for a cumbersome Q-table by directly predicting Q-values from state inputs. This method not only reduces the computational burden but also improves the algorithm's ability to learn optimal policies more effectively.

By updating the neural network parameters θ through backpropagation, DQL aligns the predicted Q-values with the target Q-values from the Bellman equation, enhancing both the accuracy and efficiency of the learning process. This integration marks a significant step forward in applying reinforcement learning to more complex environments [21].

3.2.2 Mechanisms of DQL

Deep Q-learning further adds mechanisms to enhance both learning stability and efficiency. These mechanisms are very critical to the everyday challenges encountered in the application of deep learning to reinforcement learning.

Experience Replay

Experience Replay is an important mechanism to avoid the problems caused by correlated learning samples in DQL. The experiences of an agent at each time step are stored in a data set $D_t = \{e_1, \ldots, e_t\}$ which allows the network to re-use such experiences randomly. The process can avoid any correlation between consecutive learning samples and smoothing the learning process over a more diverse range of experiences. Consequently, revisiting old states and actions enables the network to learn in a relatively efficient manner and avoids typical problems of local minima or forgetting [17].

Target and Local Networks

DQL is based on two separate neural network architectures: a local network and a target network. The local network is continually learning, updating its weights based on the newly acquired experiences. In contrast, the weights of the target network are held fixed and replaced within a certain period. This kind of separation is essential since it helps stabilize the learning updates. If a single network were used, then a constantly updating Q-value could either diverge or oscillate the learning behaviors since the network is chasing a moving target in an increasingly volatile benchmark [21].

Target Network Update Frequency

The stability of the learning process depends mainly on how often updates from the local network to the target network's weights are made. Too frequent updates, in this case, unsettle the learning process because the target values will change quickly. Infrequent updates, on the other hand, lead to poor and slow learning because the learning agent does not have time to adjust properly to frequent environment changes. Such a standard procedure often followed is to update the target network after a few thousand steps or after a fixed number of episodes as given in.

Soft Update

Soft updates smoothly mix the weights of the local network into the target network with a controlled rate defined by an interpolation parameter τ (commonly a very small number). This way, the method makes sure the weights of the target network change slowly and learning updates stay stable. The update formula can be written as:

$$\theta_{\text{target}} = \tau\theta_{\text{local}} + (1 - \tau)\theta_{\text{target}}, \tag{3.7}$$

where, here, θ_{target} and θ_{local} are the parameters of target and local networks, respectively. It smooths out large policy deviations that result from an abrupt update and helps in avoiding oscillations while making the convergence smoother [16].

Periodical Target Update Strategy

In the periodical target update strategy, the target network's weights are updated at fixed intervals, a method that relies on the consistency of updates rather than the gradual convergence provided by soft updates. Frequent updates can cause the learning process to become unstable because the target values change too rapidly, making it difficult for the agent to converge to a stable policy. Conversely, infrequent updates may result in a sluggish response to changes in the environment, as the target network does not reflect the most recent behaviors learned by the local network[21].

$$\theta_{\text{target}} = \theta_{\text{local}}, \tag{3.8}$$

where θ_{target} and θ_{local} are the parameters of the target and local networks, respectively. During the update, the target network's parameters are directly copied from the local network, effectively resetting the target to the latest policy without blending the parameters.

The choice between a periodical target update strategy and soft updates often depends on the specific requirements of the application and the dynamics of the environment in which the agent operates. The periodical update provides a simpler, albeit sometimes less flexible, approach to managing the stability-versus-responsiveness trade-off inherent in reinforcement learning tasks [21].

Hyperparameters

The learning process depends on various parameters that influence its overall effectiveness. Determining the correct values for many of these parameters combines theory and experimentation. Below is a list of these with short explanations:

- **Learning Rate**: The learning rate is a parameter used in controlling the speed with which the network updates predictions. High learning rates can make the network learn very fast but can also result in very unstable or even divergent training. On the other hand, a low rate can stabilize the training procedure but may also slow learning.
- **Mini-batch size**: The number of training examples in one iteration. While smaller batch sizes may make training proceed faster but less stably, larger batches can offer more stability at the expense of slower updates.
- **Discount Factor**: A parameter, usually denoted γ, that multiplies a reward at a given state to decide the value of the state. The smaller the discount factor, the more short-sighted the agent becomes in the sense that it emphasizes immediate rewards. A higher discount factor does, however, make the rewards down the line more important in the long run for the agent.
- **Replay buffer size**: The number of past experiences to be stored for replay. Larger buffers will store more experiences, thus grant the training richer data; at the same time, more memory and possible repetition of old data might be needed.
- **Interpolation Parameter** (τ): A parameter in algorithms that use soft updates, for instance, soft target network updates. This parameter controls how frequently target network parameters are updated with main network parameters; lower values, therefore, mean slow updates and, hence, stabilization.

- **Target Update Frequency (Periodical Target Update Strategy)**: This hyperparameter defines how often the target network's weights are updated with the weights from the local network in methods that use a periodic update strategy. Setting this value too low might lead to rapid changes in the target network, potentially causing instability during training. Conversely, a very high value may slow down the learning process as the target network lags behind the local network's improvements.
- **Learning Frequency (Soft Update Strategy)**: In contrast to the target update frequency, the learning frequency determines how often the learning updates occur in relation to soft updates. Lower frequencies can ensure that enough new data is gathered before each learning step, enhancing the robustness of updates. Higher frequencies may accelerate learning but at the risk of destabilizing the training process due to insufficient data accumulation between updates.

This is a process where one needs to understand and fine-tune hyperparameters such that maximum performance by the learning algorithm is availed with the desired behavior of the RL agent.

3.3 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a sophisticated reinforcement learning algorithm that has recently become very popular due to its remarkable results and relative simplicity. PPO was introduced by Schulman in 2017 [26]. The key innovation of PPO is the specially designed objective function that resolves the issues with sample efficiency and training stability typical for other policy gradient methods. This objective function involves clipping, which restricts the size of policy changes to reduce the deviations between the current and the new policies. This section presents the theoretical underpinning of PPO, detailing the mathematical formulations and providing an overview of the algorithm's functionality.

3.3.1 Key Concepts and Terms

Policy Gradient Methods

Policy Gradient Methods form a class of reinforcement learning techniques that optimize the policy directly. Unlike value-based methods (such as DLQ 3.2), which first estimate the value of actions or states and then derive a policy, policy gradient methods adjust the policy parameters θ directly in response to the expected return. This is achieved by calculating the gradient of the expected return with respect to the policy parameters and updating the parameters in the direction that maximizes the return [30].

The general approach involves computing an estimator of the policy gradient and using it to update the policy parameters iteratively. This gradient often involves computing expectations over a trajectory of states and actions, making these methods on-policy. The foundational formula for policy gradient updates can be expressed as:

$$\Delta\theta = \alpha \nabla_\theta \log \pi_\theta(s, a) R$$

where α is the learning rate, π_θ is the policy parameterized by θ, $\nabla_\theta \log \pi_\theta(s, a)$ is the gradient of the log-probability of the action taken, and R is the return from the state-action pair (s, a). PPO builds upon these concepts by introducing mechanisms to ensure efficient and stable updates.

Rewards-to-Go

Rewards-to-go, also known as the return, is a crucial concept in reinforcement learning that quantifies the total future reward expected from a given state, under the current policy. Mathematically, it is defined as:

$$R_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$$

where R_t represents the rewards-to-go from time t, r_k is the reward received at step k, T is the time at which the episode ends, and γ is the discount factor that weighs the importance of future rewards relative to immediate rewards [30]. This discount factor, typically between 0 and 1, ensures that rewards received sooner are prioritized over those received later, reflecting the uncertainty or decreasing significance of distant future events. In the context of PPO, rewards-to-go are used to calculate the advantage function (3.3.1), which in turn helps adjust the policy to favor actions that lead to higher future rewards [26].

Advantage Function

The Advantage Function, $A(s, a)$, is pivotal in reinforcement learning for quantifying the relative benefit of selecting a specific action a in a given state s over the policy's average. It is defined mathematically as:

$$A(s, a) = Q(s, a) - V(s)$$

where $Q(s, a)$ is the action-value function estimating the expected return of taking action a in state s, and $V(s)$ is the value function estimating the expected return from state s under the current policy. The concept of the advantage function, which is utilized in the actor-critic architecture (as mentioned in Subsection 3.3.1), facilitates policy updates by measuring the excess return of deviating from typical policy actions [30].

These formulations demonstrate how the advantage function underpins the interactions within the actor-critic architecture, guiding the policy updates in PPO towards more rewarding actions while ensuring stability and efficiency in learning.

Surrogate Objective

The Surrogate Objective function is used to ensure safe and efficient policy updates. This objective function allows the policy to be updated by taking significant, yet controlled, steps to improve performance, thereby avoiding the pitfalls of excessive policy variation which can lead to destabilization.

The surrogate objective in PPO is formulated as follows:

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)A_t) \right]$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ represents the ratio of the probability of taking action a_t under the new policy π_θ to the probability under the old policy $\pi_{\theta_{old}}$, and ϵ is a small constant, typically set to 0.1 or 0.2. The function uses clipping to bound $r_t(\theta)$, ensuring that the updates do not deviate too much from the old policy, thus maintaining the stability of the learning process. This mechanism effectively restricts the policy update step, mitigating the risk of harmful large updates that could degrade performance [26].

Actor-Critic Architecture

The Actor-Critic method is a fundamental architecture in modern reinforcement learning, prominently featured in PPO. This approach employs two neural network models: the actor, which determines the policy by mapping states directly to actions, and the critic, which evaluates these actions by estimating the value function [12] as shown in figure 3.5.

In PPO, policy updates are carried out through an interaction between two components: the actor and the critic. The actor is responsible for updating the policy parameters (θ) to maximize a surrogate objective function. This function is enhanced by the advantage function ($A_\phi(s, a)$), computed by the critic, which measures how much better an action (a) is compared to the average action at a given state (s). This metric allows the actor to prioritize more promising actions, effectively guiding the policy towards higher returns:

$$\text{Actor: } \Delta\theta = \nabla_\theta \log \pi_\theta(s, a) A_\phi(s, a)$$

The critic, on the other hand, updates its parameters (ϕ) to minimize the prediction error of the value function ($V_\phi(s)$), which estimates the expected return from each state. The

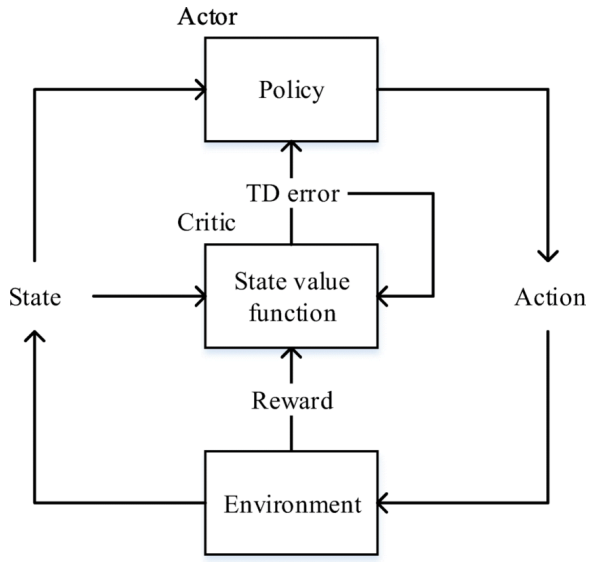

Figure 3.5: Illustration of an Actor-Critic Method, highlighting the interaction between the actor (policy model) and the critic (value model). The actor proposes actions based on the current policy, and the critic evaluates these actions, providing feedback to update the policy. Source: [33]

critic's updates help refine the baseline used to compute the advantage function, reducing the variance in the policy updates and aiding in policy stability:

$$\text{Critic: } \Delta\phi = (R - V_\phi(s))\nabla_\phi V_\phi(s)$$

Here, the advantage function is defined as $A_\phi(s, a) = R - V_\phi(s)$, where R is the reward following action a from state s. This structure allows PPO to balance exploration of new policies with exploitation of known rewarding actions, leading to more stable and effective learning [26].

Clipping Technique

The clipping technique is a crucial component of the surrogate objective, designed to maintain training stability by moderating the extent of policy updates. The clipping mechanism directly impacts the optimization process by bounding the probability ratios, ensuring they do not exceed predefined thresholds. This is described mathematically as:

$$\text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)$$

where $r_t(\theta)$ is the ratio of the new policy's probability of taking action a_t to the old policy's probability, and ϵ is a small constant, typically around 0.1 or 0.2. By clipping this ratio, PPO avoids making updates that are too large, which could potentially lead to destructive large shifts in policy behavior, destabilizing the learning progression.

The impact of this clipping on the optimization process is significant. It effectively creates a safety net that prevents the policy from moving too far from its previous state, promoting incremental learning and reducing the risk of catastrophic forgetting. This safeguard allows the algorithm to explore new strategies while ensuring that these explorations

do not undermine the accumulated knowledge base. So, in essence it is used to balance exploration and exploitation by moderating the updates to the policy, and therefore fostering more consistent and reliable convergence over time [26].

Figure 3.6: Taking smaller policy updates to improve the training stability. Inspired by: [8], Generated by [24] - Prompted by author

Figure 3.6 illustrates variations in policy updates: Blue arrows denote clipped, smaller steps promoting safer progress toward convergence, while red arrows signify longer steps with risks of falling off the cliff, underscoring the significance of smaller updates for optimal outcomes in PPO training.

Loss Function

The total loss function in PPO combines the clipped surrogate objective with terms for variance reduction and potentially an entropy bonus to encourage exploration:

$$L(\theta) = L^{CLIP}(\theta) - c_1 \cdot L^{VF}(\theta) + c_2 \cdot S[\pi_\theta](s)$$

Here, $L^{VF}(\theta)$ represents the value function loss, typically computed as the mean squared error of predicted and actual returns, and $S[\pi_\theta](s)$ is the entropy of the policy distribution, which helps to promote exploration by adding diversity to the policy actions. The constants c_1 and c_2 adjust the weighting of these additional components relative to the primary objective [26].

3.3.2 Training and Update Mechanism

Multiple Epochs per Data Sample

PPO uniquely leverages each batch of data across multiple epochs to refine the policy, enhancing the efficiency of learning from each set of interactions. In this context, an „epoch“ refers to one complete pass through the entire batch of data. By reusing data for several epochs, PPO extracts more value from each batch, improving sample efficiency and learning

stability. This repeated processing allows the algorithm to make more nuanced adjustments to the policy, reducing the likelihood of overlooking valuable information during updates. This approach contrasts with methods that use data once per update, ensuring that the agent learns more thoroughly from its experiences before discarding them [26].

Backpropagation Strategy

The backpropagation strategy in PPO involves calculating gradients of the loss function with respect to the policy parameters and applying these gradients to update the model. This process is carefully managed to maintain the stability of the learning updates. PPO's use of the surrogate objective function, combined with the clipping and multiple epochs, results in a stable gradient update pathway that minimizes the risk of catastrophic policy performance drops due to large, abrupt updates. The gradients are applied using optimizers like Adam, which adapt the learning rates based on the estimates of first and second moments of the gradients, further enhancing the stability and efficiency of the training process [11].

3.3.3 Hyperparameters

Similarly to DQL (3.2.2), PPO uses important parameters that affect training and performance. Both algorithms share some hyperparameters, such as the learning rate and discount factor, but PPO uses additional hyperparameters that reflect its learning behavior.

- **Clip Parameter (ϵ)**: The clip parameter is to moderate the policy update to avoid overly large updates, resulting in highly destructive policy changes. It is essentially setting the bounds on the size of the policy update. It clips the probability ratio to result in the new policy being within some range of the old one . This clipping mechanism is critical for keeping the training process stable, and it is vital in environments with large variance rewards [26].
- **Batch Size (Per Iteration)**: Differently, from DQL, that batch size is usually understood as the number of experience samples from the replay buffer to perform one update, PPO defines the batch size per iteration as the number of samples collected before each policy update. It is due to the on-policy nature of PPO that necessitates new samples for each iteration versus DQL on-policy characteristic that permits reusing experience from the replay buffer [26].
- **Number of Updates Per Iteration**: This parameter detailed the number of times the policy should be updated for each batch of data gathered, which I cannot find explicitly defined in DQL. While PPO says to update the policy n_steps after getting a batch of data, in practice, the policy is updated several times with the same batch. This makes more efficient use of the data and aids in faster convergence , which is specifically important for restrictive settings where harvesting data is expensive or slow.

3.4 Choosing Algorithms

In the realm of reinforcement learning, there are numerous algorithms with unique strengths and design principles that make them suitable for different kinds of problems. The decision to focus on DQL and PPO in this thesis was driven by several considerations that align with the specific requirements for smart home environment applications and also the different nature of the methodologies.

3.4.1 Deep Q-Learning:

DQL is an extension of the classic Q-Learning algorithm, enhanced with the power of deep neural networks to handle high-dimensional state spaces. This capability makes DQL particularly advantageous for smart home applications, where the agent must make decisions based on complex and often non-linear input data, such as temperature variations, energy consumption patterns, and user behaviors [21].

- **Off-Policy Learning:** DQL's off-policy nature allows it to learn optimal policies potentially faster by learning from the actions outside the current policy, which is beneficial for exploring a vast state space without compromising the learning of the current strategy [31].
- **Experience Replay:** The use of experience replay in DQL helps in stabilizing the neural networks' training by breaking the correlation between consecutive learning samples. This is crucial in dynamic environments where sensor inputs and user interactions can vary unpredictably [17].

3.4.2 Proximal Policy Optimization:

PPO, a policy gradient method, offers several advantages over other types of policy optimization, particularly in terms of training stability and ease of implementation. PPO's methodology allows for multiple epochs of learning using the same sample of data, enhancing efficiency and effectiveness in environments where data collection is costly or difficult [26].

- **On-Policy Iterative Updates:** PPO's on-policy nature and use of clipped surrogate objective functions prevent large updates, which helps in maintaining stable and reliable improvement. This feature is essential in real-world applications like smart homes, where drastic policy changes could lead to undesirable behaviors [26].
- **Adaptability and Simplicity:** PPO's balance between simplicity in hyperparameter tuning and adaptability to different environments makes it ideal for real-time applications where computational resources and response times are critical factors [26].

3.4.3 Consideration of Other Algorithms:

While DQL and PPO were selected for their specific advantages, other algorithms like Asynchronous Advantage Actor-Critic (A3C) and State-Action-Reward-State-Action (SARSA) were also considered. A3C offers fast learning times and good scalability but requires significant computational resources [20], making it less ideal for environments where simplicity

is prioritized. SARSA, while effective for learning risk-averse policies, tends to converge more slowly than DQL [30], making it less suitable for the efficiency-focused requirements of this study.

By analyzing the trade-offs between these algorithms, the choice was narrowed down to DQL and PPO, primarily due to their robustness, efficiency, and the substantial community support for practical implementation and troubleshooting. This strategic selection aims to maximize the applicability and impact of the research findings in real-world smart home automation.

Chapter 4

Design Proposal

This chapter outlines the design considerations for the project, laying the foundation for the detailed technical implementation discussed in subsequent chapters. The focus here is on establishing a clear understanding of the assignment, defining the project's primary goals, and describing the approach to be taken.

4.1 Description of the Assignment

This project is anchored in the exploration of reinforcement learning (RL) algorithms within smart home environments, with a particular focus on the regulation of room temperature. Utilizing sensory data such as indoor and outdoor temperatures, heater meter readings, and occupancy status to train an RL agent. The challenge lies in the smart home context, which presents a dynamic and complex environment. In this setting, the RL agent is required to make informed decisions that balance the comfort of the occupants with energy efficiency.

One of the critical aspects of this work involves addressing the disparity between simulated environments and real-world conditions. The creation of a simulation model is essential for several reasons. Firstly, it allows for the development and testing of RL algorithms in a controlled environment where variables can be manipulated to observe outcomes. Secondly, simulation models are an easy way to overcome the limitations of accessing real-time, large-scale smart home data due to privacy concerns and logistical constraints. These models provide a viable pathway to study the interactions between various elements within a smart home setting and the performance of RL algorithms under different conditions.

The dynamics of the environment play a significant role in the project. Smart homes are characterized by their changing states, influenced by external factors such as weather conditions and internal factors like occupancy patterns. These dynamics introduce complexities in maintaining optimal conditions, necessitating an intelligent system capable of adapting to continuously evolving scenarios.

A particularly challenging aspect of temperature regulation in smart homes is predicting the need for heating in alignment with occupancy patterns. The goal is not just to react to the presence of occupants but to anticipate their arrival, ensuring a comfortable environment upon their return. This predictive capability requires the system to learn and infer patterns from data, enabling preemptive heating adjustments. Similarly, there is a crucial need to optimize resource use, particularly energy consumption. The RL agent must learn to strike a balance between maintaining comfort and minimizing energy use, especially during periods of non-occupancy.

Lastly, the potential application of the learned model in real-life smart home systems, such as Home Assistant [1], is worth examining. By adapting the model's inputs and outputs to match the API specifications of smart home tools, the developed solution could be integrated into existing frameworks. This adaptability underscores the project's practical relevance, offering a pathway for transitioning from theoretical models to actual benefits in enhancing smart home automation and energy efficiency.

4.2 Goal of the Project

Building on the foundational understanding established in the previous chapters, this thesis aims to demonstrate and compare the effectiveness of two distinct reinforcement learning algorithms: Deep Q-Learning and Proximal Policy Optimization. These algorithms have been selected due to their differing approaches (detailed in 3.4) making them ideal candidates for evaluation and comparison in smart home temperature regulation scenarios.

Advancing this objective, the primary goal of this project is to conduct a thorough evaluation of DQL and PPO within the described smart home context. The analysis will specifically focus on their capability to intelligently regulate temperature by leveraging sensor data and accurately predicting occupancy patterns in a simulated environment. Success will be measured by the algorithms' ability to ensure occupant comfort and optimize energy consumption, especially in response to changes in occupancy.

To achieve this, a comparative analysis will be conducted to evaluate performance metrics, adaptability to dynamic environments, and efficiency in learning from complex sensor data inputs. It will delve into each algorithm's strengths and weaknesses, their responsiveness to environmental dynamics, and their potential for integration into real-world smart home systems. Ultimately, the study aims to determine which methodology is more effective and practical for smart home temperature regulation and why, providing a clear comparison of their respective advantages and limitations in this application.

4.3 Approach

This section transitions from the „what“ to the „how,“ presenting a step-by-step strategy for achieving the project's goals. It is organized into five distinct stages, each dedicated to a critical aspect of the project. These stages systematically advance the project from initial development to full implementation and integration. Below is a detailed enumeration of each stage, from developing the simulation environment and managing data, to implementing and integrating the solution with Home Assistant, and finally, evaluating the RL algorithms.

1. **Simulation Environment Development**

   The initial stage involves creating a sophisticated simulation environment to mimic the dynamics of a smart home. This environment will feature a modular design, encompassing components such as occupancy detection, temperature control, and heating system operations. Key to this stage will be the implementation of a reward mechanism to guide RL agents and the integration with the Gymnasium interface [4] for compatibility and standardization. The development of this simulation environment lays the groundwork for testing and evaluating RL algorithms in a controlled, yet realistic setting.

2. **Data Generation and Management**

   To effectively train RL algorithms, the simulation will generate synthetic sensory data, closely replicating real-world inputs from a smart home. Additionally, real outside temperature data for specific locations will be incorporated to enhance realism. A schedule module will simulate occupancy patterns, creating varied scenarios for the RL algorithms to adapt to. This combination of synthetic and real data ensures a comprehensive dataset for training, testing, and refining the algorithms.

3. **Implementation of RL Algorithms**

   This stage focuses on the development and implementation of two RL algorithms: DQL and PPO. For DQL, the process involves designing a neural network architecture, implementing experience replay for learning stability, and employing a target network to guide Q-value learning. PPO's implementation will center around designing a policy network, estimating advantages to assess action benefits, and applying a clipped objective function for stable policy updates. Both algorithms will be carefully coded and integrated into the simulation environment for subsequent training and evaluation.

4. **Integration with Home Assistant**

   The second to last stage involves creating an adapter module to bridge the RL algorithms with the Home Assistant API [1], enabling their deployment in actual smart home environments. This adapter will translate the algorithms' outputs into executable commands for Home Assistant and vice versa, ensuring seamless operation and user interaction. This integration represents the transition from theoretical development to practical application in enhancing smart home automation.

5. **Algorithm Evaluation**

   Finally, algorithm performance will be critically evaluated against key criteria, including energy efficiency, comfort level maintenance, learning efficiency, and adaptability to environmental changes. Various simulated scenarios will test the algorithms, employing quantitative metrics for a comprehensive comparative analysis. This evaluation will identify the strengths and weaknesses of each algorithm in the context of smart home temperature regulation, providing valuable insights into their differences and practical application.

Implementation Subsequent chapters will explore the practical application of the theoretical framework established in the earlier discussions. These chapters transition from design to implementation, detailing the actual development process used in the project.

The implementation follows the roadmap outlined in the previous section, advancing systematically from one stage to the next. It will include a presentation of the system architecture and key segments of the code, with a focus on providing a comprehensive explanation of the underlying principles for each phase of implementation. The aim is to show not only the 'how' but also the 'why'—enhancing understanding of how each step contributes to the overall goals of the project.

Chapter 5

Simulation Environment Development

In the first stage of implementation, the focus is on the development of the simulation environment. This section expands on the initial outline provided in the approach section, stage 1, detailing the aims and the adoption of the Gymnasium interface [4] for standardized simulation interactions. Additionally, it seeks to put some light on the organization of the internal structure and how the different modules work together.

The primary goal of this simulation environment is to create a configurable setting that emulates the complexity and dynamics of a smart home. This environment will facilitate the training, testing, and evaluation of the reinforcement learning models, providing a controlled space to simulate the various scenarios these algorithms will encounter in real-world applications.

5.1 Integration with the Gymnasium Interface

Adopting the Gymnasium interface [4] for the simulation environment ensures consistency and compatibility across a wide range of reinforcement learning scenarios. This widely-used interface provides a standardized set of methods for agent-environment interactions, including initializing states, executing actions, and resetting environments. By aligning with Gymnasium's structured API, the project's agents are equipped to operate seamlessly in diverse settings, enhancing their adaptability and effectiveness.

This standardization simplifies the development and evaluation process, making it straightforward to assess the performance of reinforcement learning algorithms.

5.2 High level architecture

The architecture of the environment, as depicted in Figure 5.1, is modular, facilitating a clear separation of concerns and specialization within the system. At its foundation, the environment is supported by two critical singleton modules: the Configuration Provider and the Time Manager.

The Configuration Provider is instrumental in parsing and distributing settings from a configuration file across the system, ensuring that all modules operate under consistent parameters, detailed further in subsection 5.5. In parallel, the Time Manager plays a

crucial role in maintaining a stable simulation timeline, for synchronizing the interactions and evolution of the environment over time.

Central to the environment are the domain-specific modules, namely the Temperature Manager, Occupancy Manager, and Heating System Manager. These modules are tasked with managing the dynamic states of the environment, such as internal and external temperatures, occupancy, and the status of the heating system. They not only supply the agent with current observations necessary for decision-making but also apply the agent's actions to the environment, thereby affecting its internal state. This bi-directional flow of information and influence is key to the adaptive and responsive nature of the environment.

Next, the inclusion of a Render Module provides an ability to see the environment's behavior. By graphically representing the evolving states and actions over time, it provides an intuitive visual feedback for observing the outcomes of the agent's interactions with the environment.

Sticking to the principles established by the Gymnasium interface, as mentioned in subsection 5.1, each module within the environment inherits this interface, ensuring a uniform approach to updating states (`step()`), initializing conditions (`reset()`) and concluding simulations (`close()`).

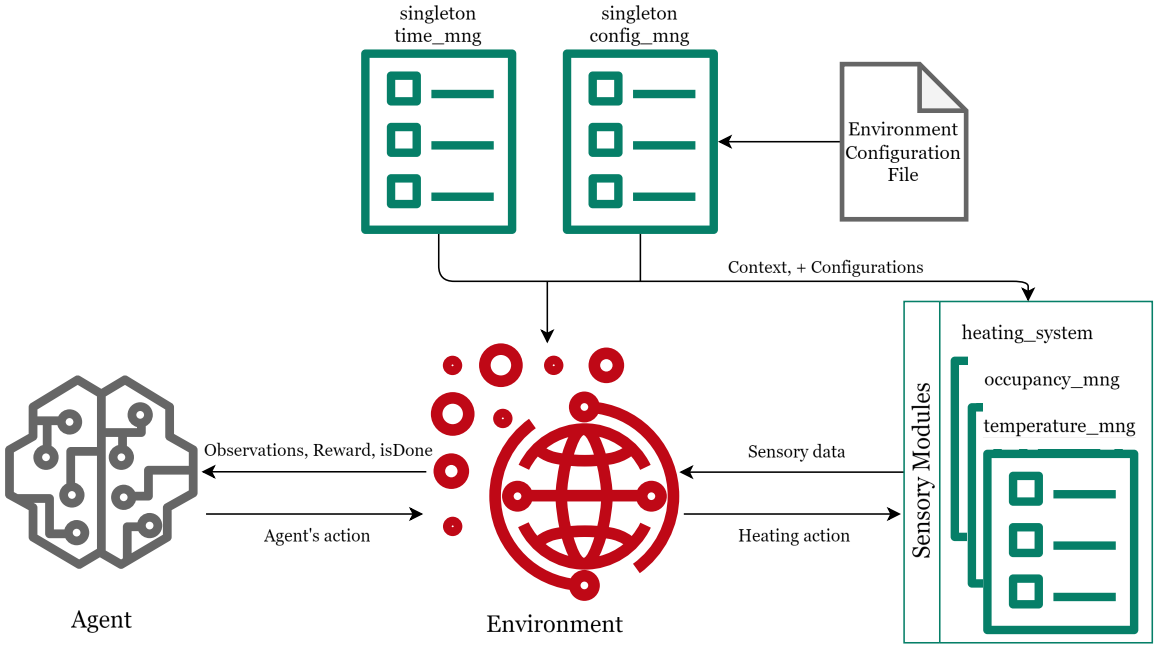

Figure 5.1: High level environment architecture

5.3 Observations and Actions

Observations

The environment provides the reinforcement learning agent with necessary information for making informed decisions. The observations are represented as a vector of continuous and discrete values. These observations are:

1. **Indoor Temperature**: A continuous value reflecting the interior temperature, which influences the comfort of occupants.

2. **Outdoor Temperature**: A continuous value indicating external temperature, which impacts indoor environmental conditions.

3. **Occupancy Status**: A binary value (0 or 1) that denotes whether the home is occupied, influencing priorities for comfort management and energy usage.

4. **Heater Status**: A continuous value that shows the current output level of the heating system.

5. **Hour and Day of the Week**: Discrete values providing time context to the agent for decision-making.

Actions

The action space is discrete and very simple. Only following five actions are available to the agent. Each action corresponds to a specific operation of the heating system, allowing the agent to control the environment's energy consumption and internal temperature:

1. **Maintain Current Level**: Keeps the heating system's output unchanged.

2. **Increase Heating by 1x, 2x, 3x Base Increment**: Increases the heaters output by the base accumulation rate.

3. **Turn Off Heating**: Reduces the heating output by the cooling rate, which may cool down the indoor environment unless offset by other heating inputs or insulation properties.

5.4 Dynamics of Internal State

The environment within smart home setting is characterized by a dynamic internal state. This state is influenced by three primary observables: temperature, heating system status, and occupancy. Understanding the intricacies of these observables is crucial for modeling the environment's behavior and optimizing the agent's interactions for energy efficiency and occupant comfort.

5.4.1 Comprehensive Temperature Dynamics

The temperature within the smart home environment exhibits dynamic behavior influenced by external conditions, the building's insulation quality, and the operational state of the Heating System. This model encapsulates these factors provides a semi-realistic (yet, still very simplified) simulation framework for training the reinforcement learning agent.

The change in indoor temperature (ΔT) over a given timestep is represented by the equation:

$$\Delta T = (\mathrm{OUT}_{\mathrm{temp}} - \mathrm{CUR}_{\mathrm{temp}}) \times (1 - \mathrm{INSULATION}) \times \mathrm{TIME\_FACTOR} + \Delta T_{\mathrm{HS}} \quad (5.1)$$

where:

- $\mathrm{OUT}_{\mathrm{temp}}$ and $\mathrm{CUR}_{\mathrm{temp}}$ represent the current outdoor and current indoor temperatures, respectively.

- INSULATION reflects the home's resistance to thermal exchange.
- TIME_FACTOR adjusts for the rate of temperature change according to the simulation timestep.

The Heating System's contribution (ΔT_{HS}) to this model is further defined by the operational mechanics of the system, which is designed to manage its heat energy through accumulation and dissipation processes resembling a radiator unit. This influence is shown in the heating system's temperature change equation:

$$\Delta T_{\text{HS}} = H_{\text{efficiency}} \times \left(e^{\left(\frac{H_{\text{energy}}}{T_{\text{base}}}\right)} - 1 \right) \times \max\left(1 - \frac{T_{\text{current}}}{T_{\text{max}}}, 0\right) \tag{5.2}$$

where:

- $H_{\text{efficiency}}$, H_{energy}, and T_{base} determine the efficiency and effect of the heat energy on indoor temperature.
- T_{current} and T_{max} are used to calculate the diminishing returns on heating as the indoor temperature approaches a maximum output level.

This unified model of temperature dynamics illustrates the complex interplay between the external environment, the home's thermal insulation, and the sophisticated operation of the Heating System. The exponential term $e^{\left(\frac{H_{\text{energy}}}{T_{\text{base}}}\right)}$ in the ΔT_{HS} equation reflects the non-linear relationship between the heating system's energy and the resulting temperature change, emphasizing the diminishing effectiveness as the desired maximal output of the heater is approached.

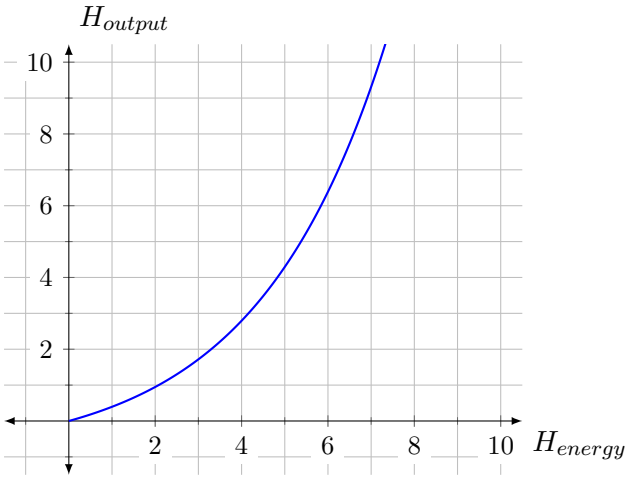

Figure 5.2: Exponential growth of output based on the energy provided, with $T_{base} = 3$.

5.4.2 Occupancy Status Dynamics

Occupancy within the smart home environment is dynamically managed based on daily schedules, random events, and real-time updates, providing a semi-realistic simulation context for the reinforcement learning agent.

Occupancy is determined through a combination of fixed schedules with variance and random events that can alter daily routines. Random events such as vacations or sick days

can override scheduled leave and return times. The system updates the occupancy status at each timestep.

All of the factors (schedules, random event probabilities, etc.) can be configured in the environment's configuration file.

5.4.3 Reward Function

The design of the reward function is a critical component, as it directly influences the agent's behavior and learning outcomes. The primary objective for the agent is to maintain a comfortable temperature when the environment is occupied, while minimizing energy use when unoccupied. Additionally, the optimal behavior would involve anticipating a person's arrival to pre-adjust the temperature accordingly, effectively optimizing both comfort and energy consumption.

Design Challenges

Defining an effective reward function proved challenging and required extensive experimentation to balance penalties and rewards appropriately. A significant difficulty was enabling the agent to anticipate future occupancy changes based on time observables. This anticipation was facilitated by introducing an extra motivational factor during the arrival of a person, enhancing the agent's ability to pre-adjust the environment's settings.

Reward Function Mechanics

The reward function is structured to differentiate between occupied and unoccupied states, with varying penalties and incentives:

$$R = \begin{cases} E, & \text{if not occupied} \\ (1 - C) \cdot E + C \cdot T, & \text{if occupied} \end{cases} \tag{5.3}$$

where:

- E represents the energy reward, penalizing high energy usage more significantly when the house is unoccupied.
- T denotes the temperature reward, which is weighted more heavily when the house is occupied, reflecting the preference for comfort.
- C is the comfort-to-cost preference factor, set to 0.65, indicating a higher weighting towards maintaining comfort when occupied.

Temperature and Energy Rewards

The temperature reward (T) is calculated based on the deviation from a target temperature, with an added motivational factor if occupancy is anticipated:

$$T = M \cdot \min\left(10, \frac{1}{|t_{\text{current}} - t_{\text{target}}|} - 10\right) \tag{5.4}$$

- t_{current} and t_{target} are the current and target temperatures, respectively.

- M is the motivation factor, which increases when an arrival is detected, enhancing the reward for pre-heating.

The function depicted in Figure 5.3 illustrates how the temperature reward, T, varies with the absolute temperature difference between the current and target temperatures. As shown, the reward is designed to steeply penalize large deviations from the target temperature, reflecting a rapid increase in incentive as the system approaches the desired comfort level. This graph effectively visualizes the non-linear relationship intended to motivate the maintenance of a near-target temperature, especially when anticipating the arrival of occupants.

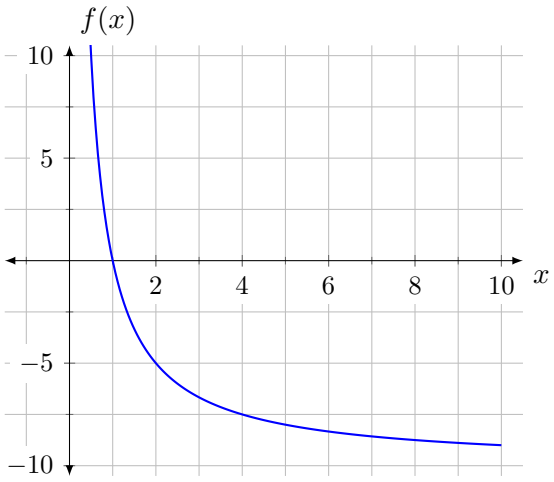

Figure 5.3: Temperature reward ($f(x)$) for temperature difference (x).

The energy reward (E) is designed to encourage lower energy consumption, particularly when the environment is unoccupied.

$$E = -10 \cdot \frac{H_{\text{energy}}}{H_{\text{max}}} \tag{5.5}$$

- H_{energy} is the current heat energy used by the heating system.
- H_{max} is the maximum heat energy capacity of the system.

This reward function setup ensures that the agent learns to manage energy and comfort efficiently, balancing immediate occupant needs with long-term energy conservation goals.

5.5 Dynamic Configuration of the Environment

The simulation environment is designed for easy customization, allowing adjustments to suit different scenarios. This flexibility enhances the model's applicability to a variety of conditions, emphasizing its adaptability. Below are the primary parameters that can be modified and their impact on the simulation.

5.5.1 General Settings

- **Minutes per Step**: Defines the temporal granularity of the simulation, dictating the duration in minutes that each simulation step represents.
- **Outside Temperature Record Step**: Specifies the interval in minutes at which the external temperature is recorded, influencing the simulation's responsiveness to external temperature fluctuations.
- **Start of Simulation**: Marks the commencement date and time for the simulation, providing a temporal anchor for the simulation's progression.
- **Temperature Data File**: Identifies the file path to the dataset containing historical or predictive temperature data, serving as the external temperature input for the simulation.

5.5.2 Temperature-Related Configuration

- **Starting and Target Temperatures**: Establish the initial conditions for the indoor temperature and the desired target temperature within the environment.
- **Insulation Quality**: A coefficient ranging from 0 to 1 that quantifies the effectiveness of the environment's thermal insulation.
- **Heater and Cooler Capacities**: Define the maximum temperature outputs for heating and cooling devices, alongside the maximum reading of the heating meter.
- **HVAC Efficiency and Meter Step**: Specify how effectively the heating/cooling outputs are converted into actual temperature changes and the rate at which the heating meter adjusts.

5.5.3 Occupancy and Schedule Configuration

- **Inhabitants**: Lists the occupants within the smart home environment, providing a basis for simulating occupancy patterns.
- **Weekly Schedule**: Outlines a typical weekly occupancy schedule for each inhabitant, including departure and return times, along with variance to simulate real-life unpredictability.
- **Random Event Parameters**: Detail the probabilities of unscheduled events affecting occupancy, such as sick days or vacations, and the potential variations in daily schedules.

These parameters allow for detailed customization of environmental. To see a configuration file used in tests, refer to the appendix section specified in A, which outlines the exact setup for all experiments.

5.6 Data Generation and Management

This second stage (from 4.3) is important to ensure that reinforcement learning algorithms are being trained and evaluated under conditions that try to mimic real-world scenarios.

Two types of input data are central to the simulation: outside temperature and occupancy schedules.

5.6.1 Outside Temperature Data

The Temperature Manager module (5.2) relies on authentic hourly temperature data from Basel (from [18]), covering a span of 10 years. This dataset provides a comprehensive baseline for simulating the varying external weather conditions that a smart home's temperature regulation system must adapt to. Since the simulation can work on steps that are less than an hour long, a linear interpolation is used to smoothly connect temperature data for these shorter intervals. Small data sample could be examined in appendix B.

5.6.2 Occupancy Schedules

An equally vital component of data generation process is the creation of occupancy schedules. The responsibility for generating these schedules lies on the Occupancy Manager module (5.2) within the simulation environment. This module crafts a new occupancy schedule daily, grounded in the parameters set within the environmental configuration (5.5.3). So, these schedules are not random but are instead based on a predefined configuration that aims to reflect realistic patterns of when occupants are likely to be present or absent from the home.

Chapter 6

Implementation of RL Algorithms

The focus now shifts to the practical application of reinforcement learning – stage 3 and 4 of 4.3. This chapter delves into the implementation of DQL and PPO algorithms, exploration of hyperparameters and neural network architectures for optimal performance and in the last section, the implementation of adater module for Home Assistant API [1].

6.1 Implementing a Deep Q-Learning Agent

6.1.1 DQL Agent Architecture

The architecture of the DQL agent is designed to facilitate continuous learning and decision-making through interaction with its environment (5). The primary responsibilities of the agent include collecting experiences, storing them in a replay memory, sampling batches of these experiences for training, updating its neural network models based on these samples, and using these models to select actions. A high-level workflow of these processes is shown in Figure 6.1. It illustrates the cyclical nature of experience collection, learning, and action decision in DQL.

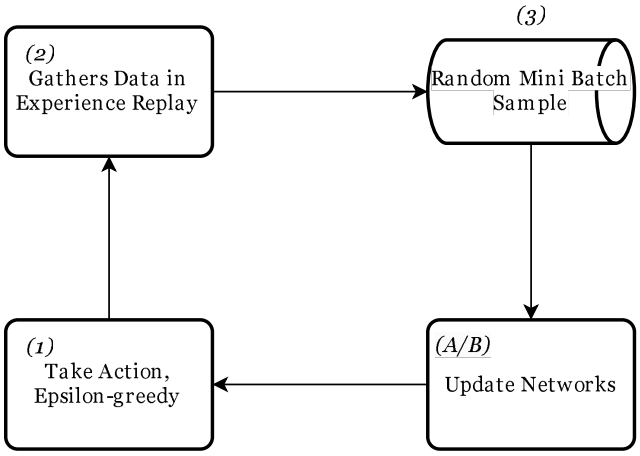

Figure 6.1: High-level workflow of the DQL agent. Inspired by: [5]

Note that each step in the workflow is numerically for clarity and will be referenced throughout this section to explain the specific functionalities implemented in the DQL agent. This structured approach ensures that each component of the agent's architecture

is clearly understood and effectively integrates with the others to form a cohesive learning mechanism.

6.1.2 Experience Replay

Before assembling everything, it is essential to have the Experience Replay class ready, as it plays a key role in steps 2 and 3. This class is essentially a container designed to manage the storage and retrieval of experiences—vectors derived from interactions with the environment.

Implementation of ReplayMemory

The 'ReplayMemory' is implemented as a cyclic buffer that maintains a set maximum size, storing experiences as tuples $(state, action, reward, next\_state, done)$. This setup ensures that once the capacity is reached, older experiences are overwritten by newer ones, thus keeping the data pool fresh and relevant to the current state of the environment.

Storing and Sampling Experiences: Class implements `push` and `sample` methods. The `push` method allows for new experiences to be added to the memory, ensuring that the memory remains updated with the most recent interactions. Conversely, the `sample` method retrieves a random batch of experiences from the memory, crucial for the network's training phase.

6.1.3 Implementing the Neural Network Class

Step 4 involves employing two neural network, the Local and Target Networks, to estimate and update Q-values based on experiences collected from the environment. To facilitate this, a `Network` class is needed, it extends PyTorch's `nn.Module`, to encapsulate the functionalities required for both networks.

Dynamic Class Implementation

The `Network` class serves as a customizable feed-forward neural network, which is defined as follows:

```
class Network(nn.Module):
    def __init__(self, in_dim, out_dim, hidden_layers=[64, 64],
                 activation=nn.ReLU, output_activation=nn.Softmax(dim=-1),
                 seed=42):
        # ... creates a network based on the configuration
```

This implementation provides a flexible architecture where the number and size of hidden layers can be dynamically adjusted, allowing for easy experimentation with different network configurations. This flexibility is crucial for exploring how variations in the neural network architecture impact the agent's performance. Parameter choices are explored in the Parameter Tuning section 6.3.

Updating Network Mechanism

Within the system architecture illustrated in Figure 6.1, two distinct approaches for updating the networks were explored, referred to as Approach A and Approach B. Each approach employs a different strategy for updating the local and target networks.

Initially, the implementation utilized a commonly used approach – Approach B, which entails a periodic replacement of the target network's weights with those of the local network (3.2.2). This approach, as depicted in Figure 6.1, is straightforward but resulted in less than satisfactory outcomes in terms of learning efficiency and stability.

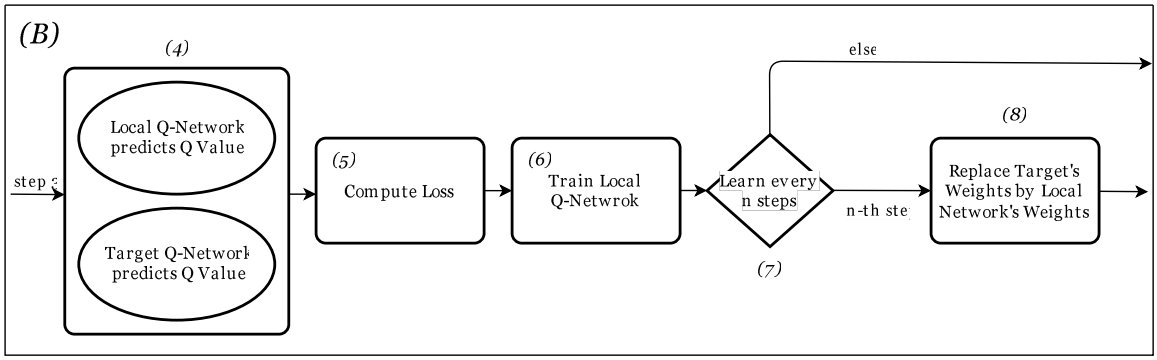

Figure 6.2: Approach B: Periodic update of target network weights.

Due to these initial challenges, a shift was made to Approach A, which incorporates soft updates (3.2.2). This method involves a gradual blending of the weights from the local network into the target network using a predefined interpolation parameter τ. Additionally, the local network undergoes periodic (typically quite frequent) learning updates based on new data accumulated from the environment interactions.

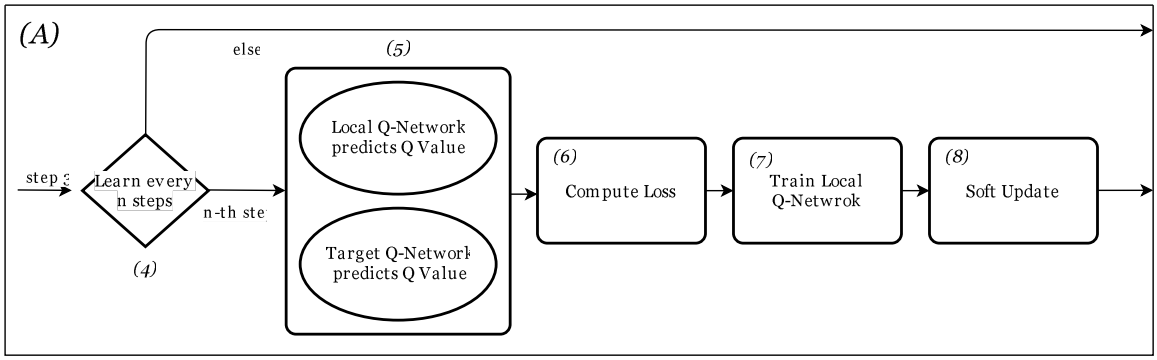

Figure 6.3: Approach A: Soft update of target network.

The significant improvement in performance with Approach A can be attributed to the more gradual integration of learned behaviors, which is particularly beneficial in environments characterized by complex and dynamic interactions. The soft update method allows the model to maintain a certain level of stability while still adapting quickly enough to effectively learn from new experiences.

The implementation going forward will utilize Approach A.

Initializing Hyperparameters

Let's start with initialization of hyperparameters. These parameters influence how the agent learns from its experiences and how it balances the exploration-exploitation. The key hyperparameters include learning rate, discount factor, and the sizes of the memory buffer and mini-batches and are explained in section 3.2.2. These need to be set up at the beginning of the agent's lifecycle.

Training Loop

The cyclic nature of architecture outlined in Figure 6.1 hints that some kind of training loop is needed. This loop is where the agent interacts with the environment to collect data, learn from it, and progressively improve its policy. Code bellow provides context for next implementation steps encapsulated in `step` method.

```
def train(self, total_timesteps=4*24*365*10):
  # ... define epsilon greedy parameters
  t_so_far = 0
  while t_so_far < total_timesteps:
    state, _ = self.env.reset(start_from_random_day=False)
    done = False
    while not done:
      action = self.get_action(state, epsilon)                # step 1
      next_state, reward, done, _, _ = self.env.step(action)
      self.step(state, action, reward, next_state, done)      # step 2 - 8
      state = next_state
      t_so_far += 1
    epsilon = self.update_epsilon(epsilon)
```

The loop manages environment resets and the flow of training. It handles interactions with the environment and encapsulates the steps shown in Figure 6.1, seperating step 1, `get_action`, and the learning steps in the `step` method.

Implementing Learning Process of Step Method

As outlined, the `step` method acts as the central point for the learning process. Here, experiences— states, actions, rewards, next states, and termination signals—are gathered and stored in the replay memory (step 2). When the memory has sufficient experiences to form a minibatch, the agent begins the learning sequence. Note that, Step 4 is executed prior to the batching process in Step 3 due to the need to synchronize updates with the correct intervals. Once all the conditions are met, the local Q-network is updated using the `update_policy` method (steps 5, 6, 7). Subsequently, step 8 is utilized to softly update the target Q-network.

```
def step(self, state, action, reward, next_state, done):
  self.memory.push((state, action, reward, next_state, done)) # step 2
  self.t_step = (self.t_step + 1) % self.learning_freqency    # step 4
  if self.t_step == 0 and len(self.memory.memory) > self.batch_size:
    experiences = self.memory.sample(self.batch_size)         # step 3
    self.update_policy(experiences, self.discount_factor)     # step 5,6,7
    self.soft_update_target_network()                         # step 8
```

Updates to Policies

So it comes to the core of learning process. DQL agent updates the policies, specifically the local Q-network, based on the experiences gathered from the environment. This process is integral to refining the decision-making capabilities of the agent over time. Following the diagram referenced earlier, the update process involves calculating the target Q-values for future states and updating the local Q-network to minimize the discrepancy between predicted and target Q-values.

Local Policy Update: The mechanism for updating the policy is implemented in the `update_policy` method. This method uses the experiences stored in the replay memory to perform the following steps:

```
def update_policy(self, experiences, discount_factor):
  states, actions, rewards, next_states, dones = experiences
  # Step 4: Compute Q values for next states from the target network
  next_q_targets = self.target_netw(next_states).detach().max(1)[0].unsqueeze(1)
  # Calculate the Q targets for current states using the Bellman equation
  q_targets = rewards + (discount_factor * next_q_targets * (1 - dones))
  # Get expected Q values from the local network for the current actions
  q_expected = self.local_netw(states).gather(1, actions)
  # Step 5: Compute the loss as the Mean Squared Error
  loss = F.mse_loss(q_expected, q_targets)
  # Step 6: Zero gradients, perform a backward pass, and update the weights.
  self.optimizer.zero_grad()                    # Clear existing gradients
  loss.backward()                               # Backpropagate the loss
  self.optimizer.step()                         # Update the network weights
```

Through these updates, the local Q-network gradually learns to predict more accurate Q-values, improving the agent's ability to make decisions that maximize future rewards. The decoupling of the target and local networks ensures that the updates are stable and that the learning progresses in a balanced manner, avoiding large oscillations or divergences in the policy.

Target Policy Update: The target network's parameters are periodically updated to gradually align with the local network, using a method the soft update 3.2.2. This process is controlled by the parameter τ (interpolation hyperparameter), which determines the degree to which the target network is adjusted towards the local network's parameters at each update cycle. This method helps maintain stability in the learning process by ensuring that the target values evolve smoothly over time.

```
def soft_update_target_network(self, local_policy, target_policy, tau):
    # Iteratively update the parameters of the target network
    for t_param, l_param in zip(target_policy.params(), local_policy.params()):
        # Perform the soft update using the blend parameter tau
        t_param.data.copy_(tau * l_param.data + (1.0 - tau) * t_param.data)
```

Implementing Decision-Making with the Act Method

To come full circle, interaction with its environment is the `act` method, which embodies the decision-making process. Leveraging the epsilon-greedy strategy, this method dynamically adjusts between exploring new actions and exploiting the agent's accumulated knowledge.

```
def act(self, state, epsilon=0.):
    self.local_qnetwork.eval()  # Set the network to evaluation mode
    state = torch.from_numpy(state).float().unsqueeze(0).to(self.device)
    with torch.no_grad():
        action_values = self.local_qnetwork(state)
    self.local_qnetwork.train()  # Set the network back to training mode
    # Epsilon-greedy action selection
    if random.random() > epsilon:
        return np.argmax(action_values.cpu().data.numpy())
    else:
        return random.choice(np.arange(self.action_size))
```

This method starts by preparing the state for evaluation and setting the local Q-network to evaluation mode, ensuring that the network's forward pass is used purely for inference without tracking gradients. The action values for the current state are then obtained by passing the state through the local Q-network. Based on the epsilon value—which decreases over time to shift from exploration to exploitation—the method decides whether to select the action with the highest Q-value or to choose an action randomly. By encapsulating the decision-making logic in the `act` method, the agent can seamlessly toggle between learning from novel experiences and leveraging its learned behaviors to navigate the environment effectively.

6.2 Implementing a Proximal Policy Optimization Agent

This section delves into the practical implementation of a PPO agent. The implementation follows guidelines from OpenAI's Spinning Up [23] in Deep Reinforcement Learning, specifically focusing on the PPO-Clip variant. The architecture of the agent, including its methods and responsibilities, is described in detail in subsequent sections.

6.2.1 Overall Structure

The implementation is guided by pseudocode from OpenAI's Spinning Up documentation, which was adapted to suit our use case in a smart home environment. The pseudocode provides a structured approach to the algorithm, as illustrated in Figure 6.4.

Algorithm 1 PPO-Clip

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}_t$.
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
  Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

  typically via stochastic gradient ascent with Adam.
  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

  typically via some gradient descent algorithm.
end for

Figure 6.4: PPO-Clip Pseudocode Implementation. Source:[23]

This pseudocode serves as the foundation for the agent class implementation.

6.2.2 Agent Implementation and Responsibilities

The agent class encapsulates various methods that are directly mapped from the steps outlined in the pseudocode. The following subsections will illustrate the connection between these steps and the corresponding implementation.

Step 1: Initialization of the PPO Agent

The initialization of the PPO agent involves setting up the actor and critic networks, configuring essential hyperparameters, and initializing optimizers.

Configuration Setup: For both the actor and critic, the neural network class implemented in the previous section (6.1.3) is utilized. The network's configurations and key hyperparameters such as learning rate, discount factor, clipping parameter, batch size, and

update frequency (3.3.3) are established to guide the training process. More about value selection in Parameter Tuning section (6.3).

Optimizer Initialization: Adam optimizers (section 3.3.2) are set up for both networks (actor, critic) to facilitate effective backpropagation, optimizing network parameters during training.

These steps equip the agent with the necessary configurations to efficiently learn and adapt within its training environment.

Step 2: Training Loop

The second step is represented by the training loop method, which orchestrates the entire learning process, managing the sequence of operations defined by the pseudocode:

```
def train(self, total_timesteps):
    t_so_far = 0
    i_so_far = 0
    while t_so_far < total_timesteps:
        batch_data = self.collect_batch_data()
        batch_lens = batch_data[-1]
        t_so_far += np.sum(batch_lens)
        i_so_far += 1
        self.update_policy(batch_data)
```

This implementation effectively splits the responsibilities of the learning process into distinct phases: data collection and policy updating. Each phase focuses on specific tasks, with data collection gathering necessary experiences and policy updating refining the agent's strategy based on those experiences.

Step 3, 4: Collect Set of Trajectories (Batch Data)

This function is pivotal in collecting batch data, which consists of multiple episodes of agent-environment interactions. It is integral for computing the subsequent rewards-to-go (3.3.1) and advantage estimatesre (3.3.1) (step 4 and 5), which are essential for the policy update phase.

Data Collection Mechanism: During each episode, the agent interacts with the environment to collect observations, actions, rewards, and log probabilities of the actions based on the current policy. The process begins with the environment's initial state and continues until a terminal state or episode cutoff is reached.

```
# Function to collect batch data
def collect_batch_data(self):
    batch_obs, batch_acts, batch_log_probs, batch_rews, batch_lens = [] ...
    t = 0
    while t < self.batch_size:
        obs, _ = self.env.reset()
        done = False
        while not done:
```

```
            action, log_prob = self.get_action(obs)
            obs, rew, done, _ = self.env.step(action)
            bath_obs, batch_acts, ... << obs, action, ...
            t += 1
    return batch_obs, batch_acts, ...
```

Action Selection: Action selection in the PPO agent is handled by the `get_action` method, which samples actions based on the output probabilities from the policy network. This process utilizes a Categorical distribution to ensure actions are chosen stochastically, reflecting the learned policy's probabilities.

```
# Selects an action based on the current policy
def get_action(self, obs):
    action_probs = self.actor(torch.tensor(obs)
    dist = torch.distributions.Categorical(probs=action_probs)
    action = dist.sample()
    return action.item(), dist.log_prob(action)
```

The observation (`obs`) is input to the actor network, which outputs action probabilities. These probabilities are used to instantiate a Categorical distribution, from which an action is sampled and will be used in the next steps. The selected action and its log probability are essential for the policy gradient calculations during the training phase.

Rewards-to-Go Computation: Rewards-to-go (3.3.1), which are crucial for calculating advantages, are computed by accumulating discounted rewards from the end of each episode to the beginning. Agent's method `compute_rtgs` ensures that each step's reward includes future rewards, adjusted by a discount factor. This concludes the step 4 of the PPO pseudocode.

Step 5: Advantage Calculation

Calculation of the advantage function $A(s, a)$ informs how beneficial it is to take an action a at state s compared to the average. The advantage is determined by the formula from section 3.3.1, but this implementation needs a little alteration (see π and ϕ_k):

$$A(s, a) = Q^{\pi}(s, a) - V_{\Phi_k}(s) \tag{6.1}$$

Here, $Q^{\pi}(s, a)$ represents the Q-value for a state-action pair, which we have pre-computed using rewards-to-go, and $V_{\Phi_k}(s)$ is the value of the state as predicted by our critic network for a current epoch. To obtain $V_{\Phi_k}(s)$, we utilize the `evaluate` method.

The `evaluate` method computes the value V for each observation in the batch from our critic network. It also returns the log probabilities of actions taken, which are useful for subsequent calculations in the policy update step(step 6). This dual output ensures that all the necessary components for calculating the advantage and updating the policy are available in one efficient step.

```
# Calculate V_{Phi_k}
V, _ = self.evaluate(batch_obs)
A_k = batch_rtgs - V
```

In practise, however, a normalization was needed to enhance stability and to ensure consistent training performance.

```
A_k = (A_k - A_k.mean()) / (A_k.std() + 1e-10)  # Prevent division by zero
```

Normalization of the advantages, though not specified in the original PPO pseudocode, is crucial in practice. It helps manage the scale differences across dimensions in the optimization algorithm, facilitating smoother and more stable training. This step effectively prepares the advantage values for the upcoming policy update steps. Tip was found in medium article [32].

Step 6: Updating the Actor Network Parameters

In Step 6 of the PPO algorithm, we update the parameters Θ of the actor network, leveraging the calculated probability ratios and advantages. This process might sound complicated, but in reality, it's quite manageable since most of the necessary data is either pre-calculated or readily obtained through existing subroutines.

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right) \tag{6.2}$$

Calculation of Probability Ratios: The point of this substep is to compute the probability ratios, which reflect the change in policy behavior over iterations:

$$\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} \tag{6.3}$$

To calculate this, we first retrieve the log probabilities of actions taken under the current policy parameters Θ and the log probabilities from the previous iteration Θ_k. Note that batch_log_probs was collected in step 2.

```
# Extract current and stored log probabilities for actions
_, curr_log_probs = self.evaluate(batch_obs, batch_acts)
ratios = torch.exp(curr_log_probs - batch_log_probs)
```

Surrogate Loss Functions: The surrogate loss functions, as explained in 3.3.1, help fine-tune the actor network by comparing how the actions taken according to the current policy (with parameters Θ) measure against those taken under the previous policy parameters (Θ_k). This comparison is made using calculated probability ratios and the advantages A_k:

$$min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), g(\epsilon, A^{\pi_{\theta_k}}) \right) \tag{6.4}$$

Let's start by computing the surrogate losses:

```
# Compute surrogate losses using clipped and unclipped ratios
surr1 = ratios * A_k #left side
# clamp binds arg 1 between arg 2 and arg 3 as lower and upper bounds
surr2 = torch.clamp(ratios, 1 - self.clip, 1 + self.clip) * A_k #right side
```

And now, the calculation of final actor loss:

```
actor_loss = (-torch.min(surr1, surr2)).mean()
```

This operation serves a specific purpose: the algorithm seeks to maximize the performance function, which aligns with improving policy. But since the optimizer used (Adam 3.3.2) is designed to minimize loss, negating the minimum of surrogate losses effectively turns the minimization process into one of maximization. The mean is taken to simplify the loss from a batch of data into a single scalar value, which can be efficiently processed by the optimizer.

Multiple Epochs in Policy Update: The PPO enhances training stability by updating the policy through multiple epochs, using the same batch of data. This is represented by the $\sum_{t=0}^{T}$ in the pseudocode and is implemented in the `update_policy` method, which performs several epochs of optimization:

```
for _ in range(self.n_updates_per_iteration): # T
    V, curr_log_probs = self.evaluate(batch_obs, batch_acts)
    A_k = batch_rtgs - V.detach()
    A_k = (A_k - A_k.mean()) / (A_k.std() + 1e-10)
    self._update_networks(A_k, batch_log_probs, curr_log_probs, V, batch_rtgs)
```

Each epoch attempts to refine the policy by recalculating the values and advantages, and applying updates to the network parameters. The `n_updates_per_iteration` hyperparameter controls the number of these optimization epochs, allowing for incremental improvements to the policy based on the same data, thereby maximizing learning efficiency from each batch.

Optimization Step: The final step involves backpropagation and optimization of the actor network:

```
# Perform backpropagation and update actor network parameters
self.actor_optim.zero_grad()
actor_loss.backward()
self.actor_optim.step()
```

This process uses the Adam optimizer to minimize the negative of the surrogate loss, effectively maximizing the policy performance function through stochastic gradient ascent.

This concise implementation efficiently encapsulates the core mechanics of Step 6 in the PPO algorithm, focusing on the practical application of theoretical concepts in a dynamic training environment.

Step 7: Updating the Critic Network Parameters

Step 7 involves updating the critic network parameters using the mean squared error (MSE) loss between the predicted state values and the actual rewards-to-go. This process refines the critic's accuracy in estimating the expected returns from given states, an essential aspect for effective policy updates.

Critic Network Optimization: To begin, an Adam optimizer is also set up specifically for the critic network to handle its parameter updates.

The MSE loss calculation involves comparing the predicted values $V_\Phi(s_t)$ from the critic network against the pre-computed rewards-to-go (step 4). The predicted values are obtained via the `evaluate` method as in step 5.

Following the extraction of V, the MSE between these predictions and the rewards-to-go is calculated using a built-in function from the `torch.nn` module:

```
# Calculate the mean squared error loss for the critic updates
critic_loss = nn.MSELoss()(V, batch_rtgs)
```

Backpropagation on the Critic Network: With the critic loss calculated, it is now possible to update the critic parameters using the critic's optimizer, similar to how the actor was updated in the previous code.

6.3 Parameter Tuning

As outlined in the sections 3.2.2 and 3.3.3, the performance of RL algorithms depends heavily on the correct choice of hyperparameters and the configuration of the neural network architecture. Given the complexity of determining the optimal parameters, an empirical approach has been adopted, experimenting with various combinations to discover the most effective ones.

6.3.1 Methodology

Configuration Trail: The methodology for testing each configuration involved consistent conditions across trails to ensure comparability. Each trial was held in the same simulated environment, and specific random seeds were set manually to the same initial conditions for all the sessions. In addition, each algorithm had an equal number of timesteps to learn from. The training period was 15 years in total, where each timestep equals 15 minutes. This translates to $4 \times 24 \times 365 \times 15$ timesteps.

To evaluate the performance of each algorithm, the focus was on the average results of the last 30 episodes of the environment. This measure provided an indication of the stability and effectiveness of the learning process towards the end of the training period. The evaluation helped to identify how well each algorithm adapted to the environment and optimized its policy over an extended duration, highlighting differences in learning outcomes and algorithm efficiency.

Approach: Given the vast number of possible combinations of hyperparameters and neural network configurations, testing every potential setup would be computationally infeasible due to the exponential nature of the combinations. To efficiently navigate this challenge, the tuning process is divided into stages: Initially, the best performing neural network configuration is identified using a default hyperparameter setting. Subsequently, the most effective hyperparameters are determined using this optimal network configuration. For a comprehensive assessment, the top five performing combinations of neural network and hyperparameter configurations are then evaluated over the last 30 episodes of training in various environments. This phased approach ensures a thorough exploration near the optimal configuration, balancing computational efficiency with the pursuit of high performance.

6.3.2 Top Performing Neural Network Configurations

The top five neural network configurations, as determined by their performance in the final training episodes, are listed below. These configurations represent the most successful architectures tested under a uniform hyperparameter settings.

DQL Configuration

Learning rate	0.0005
Batch size	100
Discount factor (γ)	0.99
Learning frequency	4
Interpolation	0.001
Replay buffer size	100000

PPO Configuration

Learning rate	0.0003
Batch size	2048
Discount factor (γ)	0.99
Updates/iteration	10
Clip	0.2

Figure 6.5: Hyperparameters used for identifying optimal neural network configutaion.

Network Layers	Average Score
32:128:256	1053.49
128:32:256	762.22
256:128:128	686.43
128:256:64	588.22
128:256:128	537.18

Table 6.1: PPO: Top five performing neural network configurations.

Network Layers	Average Score
32:256	-220.92
32:32:128	-415.15
256:128:256	-517.21
32:64:32	-904.48
32:64	-912.12

Table 6.2: DQL: Top five performing neural network configurations.

6.3.3 Top Performing Hyperparameter Configurations

Following the identification of the best neural network setups, the top hyperparameter settings are determined based on their performance when applied with the optimal network configurations. The table below showcases the five best hyperparameter setups.

Learning Rate	Discount (γ)	Batch size	Updates/iteration	Clip	Score
0.0005	0.95	4096.0	20.0	0.3	2057.9
0.0015	0.95	4096.0	10.0	0.2	1951.26
0.0005	0.95	1024.0	10.0	0.3	1849.28
0.0015	0.95	1024.0	10.0	0.3	1656.19
0.0005	0.9	4096.0	20.0	0.3	1643.64

Table 6.3: PPO: Top five performing hyperparameter configurations.

Learning Rate	Minibatch	Discount (γ)	Update Freq	Interpolation	Score
0.0005	128	0.99	4	0.025	-560.12
0.0005	96	0.95	4	0.025	-918.97
0.0005	96	0.95	4	0.001	-928.79
0.0005	128	0.95	16	0.001	-1002.38
0.0015	96	0.95	16	0.005	-1064.18

Table 6.4: DQL: Top five performing hyperparameter configurations.

6.3.4 Overall Top Performing Configurations

The final phase involves combining the best neural network and hyperparameter configurations to identify the overall top performing setups. These results, detailed in the tables above, provide insights into the optimal combinations for each algorithm.

Layers	Learning Rate	Discount (γ)	Batch size	Updates/iteration	Clip	Score
32:128:256	0.0005	0.95	4096.0	20.0	0.3	2057.9
32:128:256	0.0015	0.95	4096.0	10.0	0.2	1951.26
128:32:256	0.0005	0.95	4096.0	20.0	0.3	1919.0
32:128:256	0.0005	0.95	1024.0	10.0	0.3	1849.28
128:32:256	0.0005	0.9	4096.0	20.0	0.3	1829.31

Table 6.5: PPO: Top five combinations of the best-performing HP and NN configurations from previous runs.

Layers	Learning Rate	Minibatch	Discount (γ)	Update Freq	Interpolation	Score
32:64:32	0.0005	128	0.99	4	0.025	-138.49
32:32:128	0.0015	96	0.95	16	0.005	-240.74
32:64	0.0005	128	0.99	4	0.025	-506.26
32:256	0.0005	128	0.99	4	0.025	-560.12
32:256	0.0005	96	0.95	4	0.025	-918.97

Table 6.6: DQL: Top five combinations of the best-performing HP and NN configurations from previous runs.

Conclusion: Having identified the optimal configurations for both DQL and PPO, one can argue that they do not present the absolute best possible values. And he would be right. Given the limited computational resources available, this will have to suffice. The next chapter will utilize the best performing configurations and dives into a deep evaluation of the performances. Such an analysis would serve to determine which algorithm performs better under what conditions while also highlighting observed differences. The insights will be helpful in further fine-tuning the understanding of the strengths and limitations of each algorithm in the perspective of smart home automation.

For a comprehensive review of all tested configurations along with their performance metrics for both PPO and DQL, and details on the configuration of the environment, refer to the appendix C.

6.4 Integration with Home Assistant

The integration of learned agents with Home Assistant represents a transition from theoretical algorithm development to practical application. This section describes the development of an adapter module designed to facilitate communication between the RL algorithms and the Home Assistant API [1], thereby enabling the deployment of these algorithms in actual smart home settings.

Design of the Home Assistant Adapter

The Home Assistant Adapter serves as the intermediary that translates the numerical outputs from the RL agents into http requests that are interpretable by Home Assistant, impacting actual devices. It also provides ability to read the states and events from Home Assistant and turn them into inputs for the RL algorithms. This bidirectional communication ensures that the algorithms can effectively control smart home devices based on real-time data and user interactions.

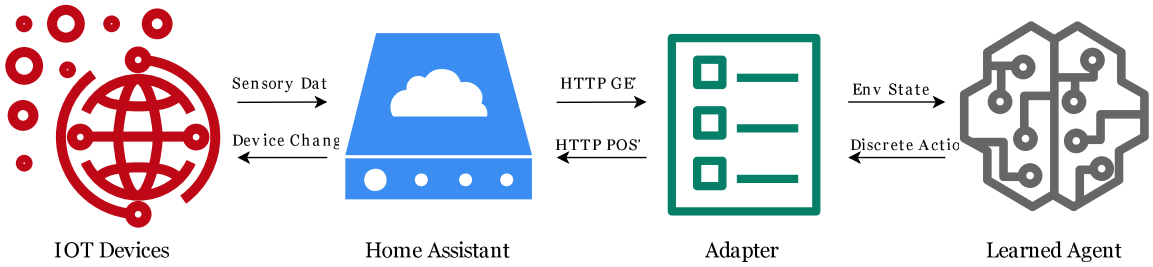

Figure 6.6: Learned model controlling IoT devices in smart home using Home Assistant Adapter.

Interaction Workflow

The interaction between the RL agents and Home Assistant through the adapter is defined by the following workflow:

- **Data Collection:** The adapter fetches current state data from Home Assistant's sensors, which is translated into the format required by the RL agent.
- **Decision Making:** Based on this input, the RL agent evaluates and determines the optimal actions according to its learning.
- **Action Execution:** The adapter then translates these decisions back into commands that are sent to Home Assistant for device control, such as adjusting heating unit.

This workflow ensures that the intelligent agent can interact with real smart home environment.

Chapter 7

Algorithm Evaluation

This chapter showcases the core value of the thesis by evaluating the performance and behavior of the algorithms under study. Having implemented and identified the best configurations for each algorithm, this section aims to provide a comprehensive assessment of the final results, emphasizing specific performance metrics and observed behaviors.

Note on Graphical Representation: All graphs presented in this chapter are color-coded for clarity and ease of interpretation. The color blue represents data corresponding to the PPO algorithm, while orange is used to denote the DQL algorithm. This color scheme is consistently applied across all visual data to facilitate a direct and efficient comparison between the performances of the two algorithms.

7.1 Methodology

The evaluation process is based on a number of fundamental indicators that are intended to capture both performance and stability, together with certain behavioural elements that are important for smart home environments. These include the algorithm's ability to keep the environment at comfortable temperatures, save energy, deal with sporadic events like unplanned occupancy changes, and anticipate people's arrival.

DQL Configuration

Hidden Layers	32:32:128
Activation f.	ReLU
Output Activ. f.	Softmax
Seed	42
Learning rate	0.0005
Batch size	96
Discount factor (γ)	0.95
Learning frequency	16
Interpolation	0.005
Replay buffer size	100000

PPO Configuration

Hidden Layers	32:128:256
Activation f.	ReLU
Output Activ. f.	Softmax
Seed	42
Learning rate	0.0005
Batch size	4096
Discount factor (γ)	0.95
Updates/iteration	20
Clip	0.3

Figure 7.1: Configurations used for evaluation based on Parameter Tuning section (6.3).

It is important to mention that this evaluation uses the optimal configuration for PPO and the second-best configuration for DQL, as the latter demonstrated higher performance

peaks but lesser stability. For results regarding the best performing DQL configuration from the Parameter Tuning section, please refer to the appendix D.

7.2 Performance and Stability

To assess performance and stability, the best performing configurations of each algorithm (identified in 6.3) are tested across multiple environment seeds (yielding different episodes). This approach aims to evaluate the consistency of the algorithms and to identify any variations in performance that may arise due to different initial conditions or stochastic elements within the environment.

Performance Interpretation in General: Based on the reward function detailed in 5.4.3, achieving a score around zero is indicative of very good performance. This conclusion is drawn from the structure of the reward function, where positive rewards are granted primarily when the system maintains the environment within the comfort temperature zone without getting significant penalties for heating. This suggests that a score approaching zero effectively balances maintaining comfort with minimizing energy use. The tables below present scores across multiple environments with varying seeds, where each score represents a checkpoint in the learning process across 100 years of simulated data. Specifically, each score reflects the average of the last 10 episodes prior to reaching each checkpoint, providing a snapshot of the agent's performance stability and effectiveness at that stage of learning.

Seed	**20 years**	**40 years**	**60 years**	80 years	100 years	**Avg**
Seed 1	2771.96	539.84	671.73	734.59	734.59	1090.54
Seed 2	2151.24	423.38	1022.43	510.84	510.84	923.75
Seed 3	2153.62	148.05	545.75	747.30	747.30	868.40
Seed 4	1968.15	505.94	893.53	824.63	824.63	1003.37
Seed 5	1696.21	-114.30	848.74	658.93	658.93	749.70
Avg	2148.24	300.58	796.44	695.26	695.26	

Table 7.1: PPO: Performance overview in multiple environments.

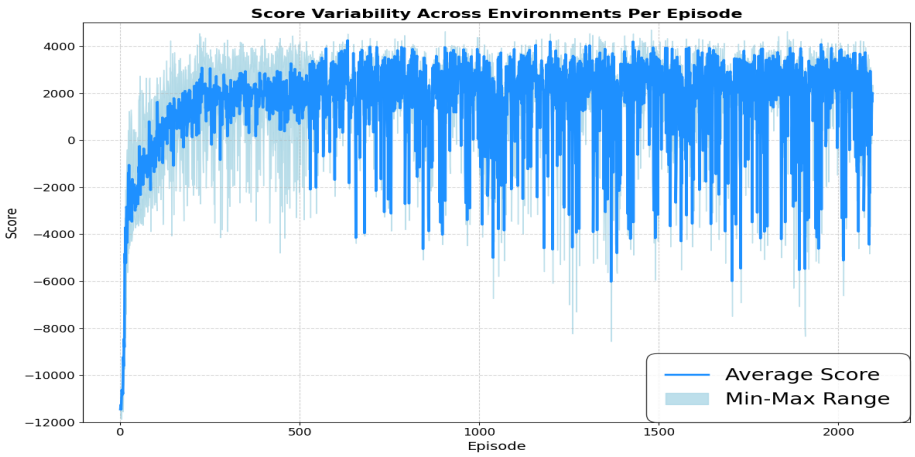

Figure 7.2: PPO: Stability overview in multiple environments.

Interpreting PPO Performance

The performance data from Table 7.1 for PPO shows variability in scores across different seeds, but it generally presents a trend towards improvement or stabilization over time. For example, the scores tend to converge around similar values by the 100-year checkpoint across different seeds, which suggests that PPO is able to learn and adapt effectively to the environment over an extended period. The average scores decrease significantly from the 20-year to the 100-year mark, indicating a learning curve where the algorithm optimizes its policy to better handle the environment dynamics. Notably, Seed 5 displays an impressive recovery, turning a negative score into a positive trajectory by the 100-year checkpoint, highlighting the potential for significant improvement even from poor initial conditions.

Seed	**20 years**	**40 years**	**60 years**	80 years	100 years	**Avg**
Seed 1	-2280.56	-2985.57	-4175.33	-5844.48	-5844.48	-4226.08
Seed 2	-2643.19	14.59	-1212.67	-3881.56	-3881.56	-2320.88
Seed 3	-1117.42	-2729.79	-4784.14	1423.72	1423.72	-1156.78
Seed 4	-1436.19	-11606.35	-11606.35	-11606.35	-11606.35	-9572.31
Seed 5	-1324.46	-1921.78	-1897.19	2577.82	2577.82	2.44
Avg	-1760.36	-3845.78	-4735.14	-3466.17	-3466.17	

Table 7.2: DQL: Performance overview in multiple environments.

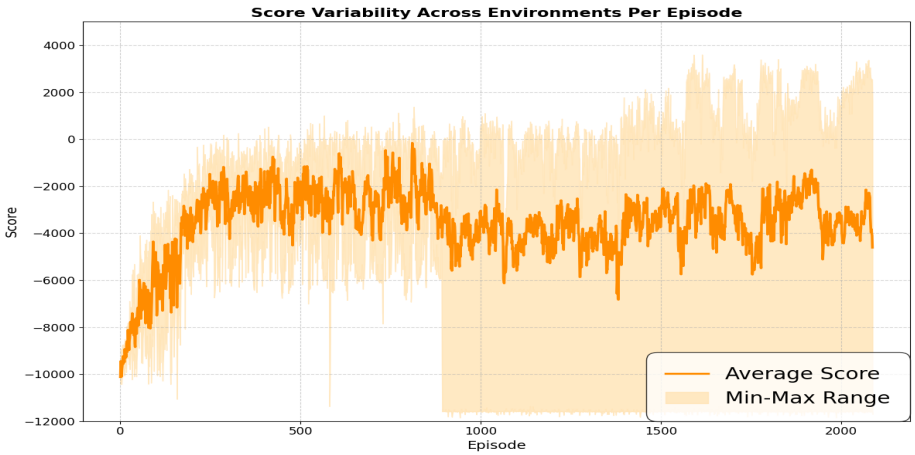

Figure 7.3: DQL: Stability overview in multiple environments.

Interpreting DQL Performance

Conversely, DQL's performance in Table D.1 shows a less favorable trend. The scores are significantly lower, with many values remaining negative throughout the simulation period, which implies persistent difficulties in achieving the desired outcomes. Specifically, Seed 4 and Seed 2 exhibit particularly challenging scenarios, where scores do not only remain negative but worsen considerably over time. This could indicate that DQL struggles with specific types of environmental changes or with maintaining stability in its learning process, possibly due to less effective handling of the exploratory aspects of the environment or a less robust response to the reward structure.

7.3 Behavioral Analysis

Following the performance assessment from the previous section, the behavior of the best-trained agents is analyzed in detail. For PPO, it involves the agent from the environment with seed 1, and for DQL, the agent is from seed 5. The aim is for behavior that mimics the desired behavior described in figure 2.1. This analysis focuses on the agents' operational effectiveness in:

1. Maintaining desired temperature levels within the home, thereby ensuring occupant comfort.
2. Optimizing energy usage, reflecting the agents' efficiency and economic impact.
3. Responding to random events, such as a resident unexpectedly staying home or leaving, which tests the adaptability of the algorithms.
4. Anticipating the arrival of persons, a critical factor that significantly enhances the proactive capabilities of the smart home system.

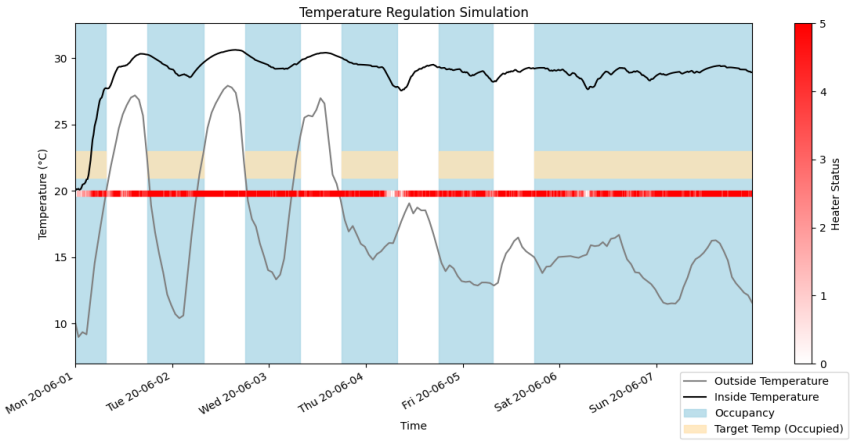

Figure 7.4: Reference behavior where random action is taken.

7.3.1 Temperature Comfort

This subsection assesses the agents' ability to maintain a comfortable temperature, focusing specifically on performance during the winter season when adjustments are most needed.

Analysis of winter performance (Figure 7.5) reveals that both agents effectively maintain temperatures within the comfort zone. The PPO algorithm tends to keep the temperature closer to the middle or slightly higher within the comfort range, possibly prioritizing occupant comfort over energy efficiency. In contrast, the DQL generally maintains temperatures at the lower end of the comfort zone, which might reflect a strategy that slightly favors energy conservation over optimal comfort.

7.3.2 Energy Economy

Economic use of energy is an important aspect of performance evaluation, especially when considering the cost implications. This subsection assesses how effectively each algorithm

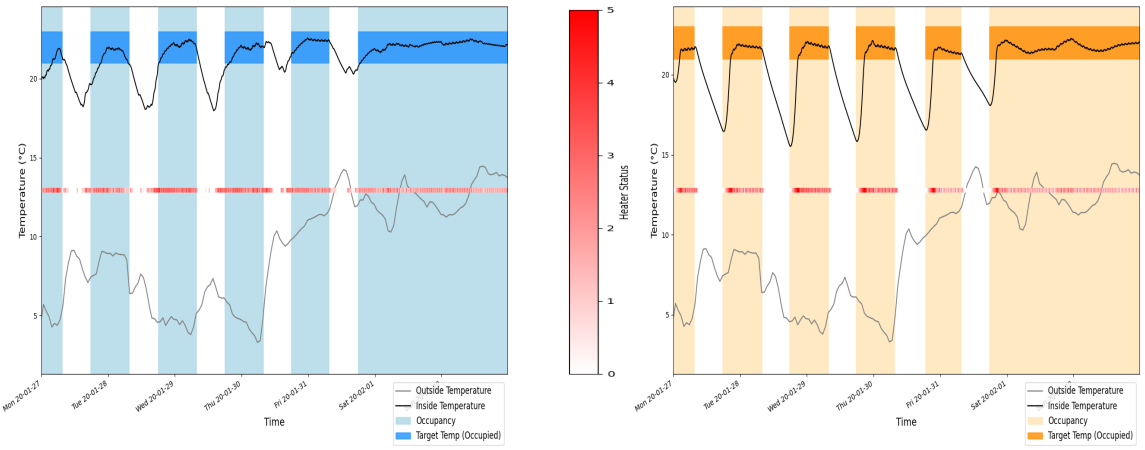

Figure 7.5: Winter season behavior.

minimizes energy usage, particularly when no one is home, which directly contributes to energy conservation and cost savings.

This evaluation is situated during the summer season (Figure 7.6). During this time the challenge intensifies as the indoor temperature may naturally increase due to external conditions. Thus, an efficient agent should leverage these natural temperature increases to minimize artificial heating. The focus of this analysis is on the heater's status, represented by a red-white line in the middle of each graph.

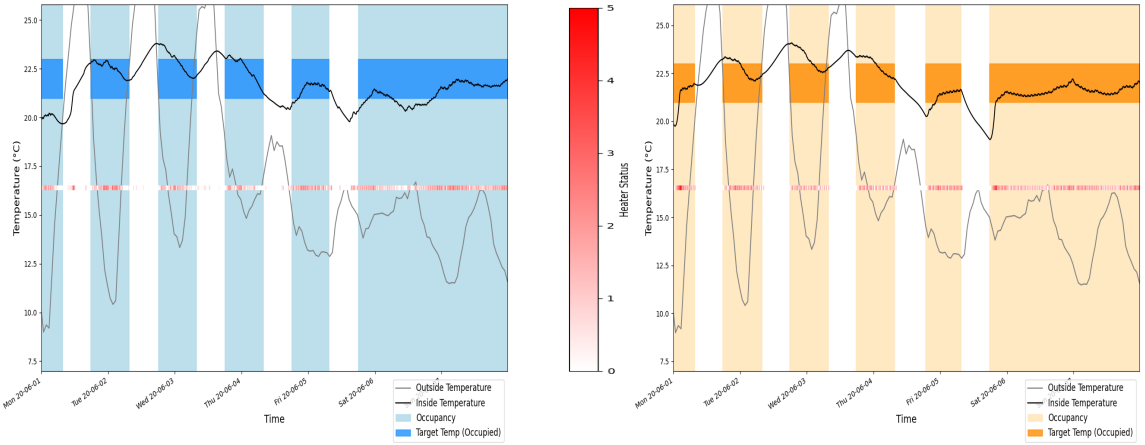

Figure 7.6: Summer season behavior.

Interestingly, the DQL algorithm appears to excel in this aspect. Observations from the charts indicate that during periods of unoccupancy, which are highlighted in white, the heater remains mostly off under DQL's control, showcasing effective energy conservation. DQL also manages to maintain comfortable indoor temperatures throughout the day, suggesting a well-tuned balance between comfort and energy efficiency.

On the other hand, the PPO algorithm exhibits a different pattern. While it generally keeps the heater off during unoccupied periods, there are occasional activations that do

not seem to disrupt overall energy efficiency significantly. However, it is worth noting a particular oversight on Monday during unoccupied hours. The PPO fails to anticipate a rise in external temperatures, which could naturally help increase the indoor temperature without the need for heating. This missed opportunity for additional energy savings highlights a potential area for improvement in PPO's predictive capabilities or temperature management strategies.

Overall, both algorithms demonstrate strengths in energy management, with DQL showing a slight edge in utilizing periods of natural warmth and minimizing unnecessary heater use.

7.3.3 Non-typical Schedule

The ability to recognize and adapt to non-typical schedules is a capability that distinguishes advanced reinforcement learning systems from simple programmable thermostats. For real-world applications where everyday habits could change suddenly, this capacity is essential.

This analysis evaluates how well the agents manage a non-typical schedule, particularly focusing on a week where Monday is unexpectedly unoccupied, and Thursday sees occupancy atypical of the regular pattern. Furthermore, there are variations in arrival times, adding another layer of complexity to the scenario. The agents' behavior during this schedule can be seen in Figure 7.7, which illustrates how each algorithm adapts to these irregularities.

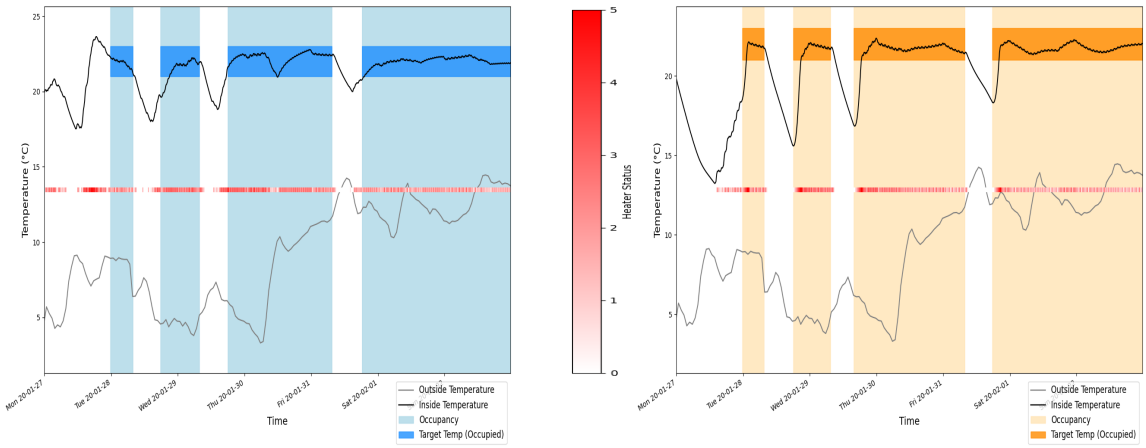

Figure 7.7: Behavior during non-typical schedule.

The behavior analysis of Figure 7.7 reveals that both algorithms perform commendably under these conditions. Each demonstrates a capacity to economize energy use effectively on days without occupancy and to maintain comfortable indoor temperatures during unexpected occupancy on typically unoccupied days. This performance is indicative of the agents' ability to learn and react not just to fixed or frequently repeating patterns but to dynamically assess and respond to changes in the environment.

7.3.4 Arrival Anticipation

One of the key goals for the agents is to develop the ability to heat the house in advance in anticipation of a person's arrival. Achieving this would elevate the agent from a mere

reactive system to an adaptive one. A system capable of proactively managing the home environment. This task was complicated by the incorporation of unexpected schedule changes into the training scenarios as discussed earlier.

In the evaluation of the agents' performances as depicted in figures 7.5, 7.6, and 7.7, it was observed that the PPO algorithm partially succeeded in learning to anticipate arrivals. This was evident from its ability to activate heating systems ahead of time. In contrast, the DQL struggled with this aspect of the task. DQL demonstrated a considerable lack of understanding in adapting its strategy to accommodate for anticipated changes in occupancy, often failing to pre-heat the home effectively.

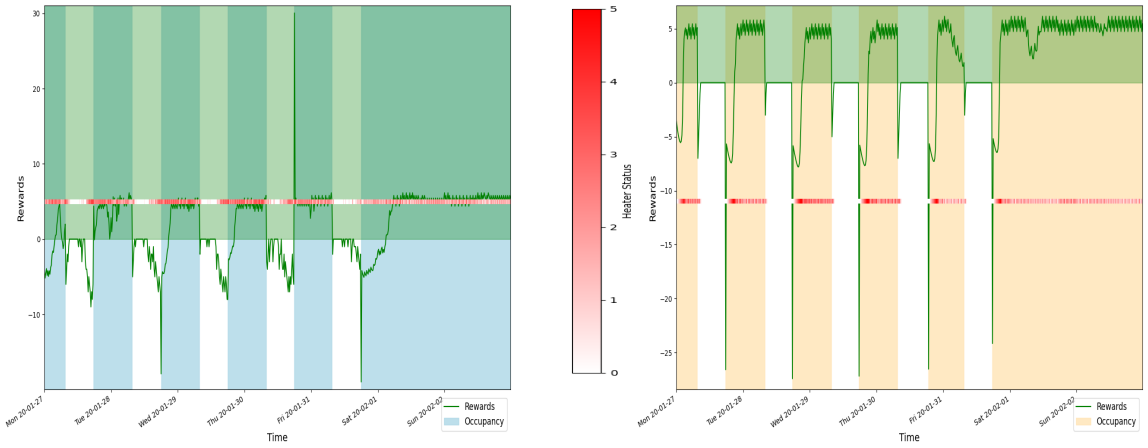

Figure 7.8: Rewards step by step on the winter season dataset 7.5.

This discrepancy in anticipatory behavior highly contributes to the differing performance scores observed across episodes. The reward function, as outlined in section 5.4.3, incorporates a multiplier that significantly increases the reward when the system successfully prepares the home environment for an arriving occupant. The Figure 7.8 illustrates the performance regarding rewards. PPO shows some proficiency in minimizing penalties associated with unpreparedness or even securing substantial rewards by ensuring the home is appropriately conditioned upon arrival.

Exploring Factors Influencing Learning Success in Anticipatory Actions The varying success in learning to anticipate future events, might be deeply rooted in their distinct algorithmic frameworks. PPO, operating as a policy-based approach, directly optimizes the policy function, enabling a dynamic adaptation strategy that is useful for tasks requiring foresight and proactive adjustments. This capability is significantly supported by PPO's use of the advantage function, which evaluates the relative benefits of potential actions facilitating strategic decisions that prioritize long-term benefits over short-term gains. Moreover, the methodological design of PPO incorporates gradient clipping and multiple passes over the same data, promoting a stable learning progression that accommodates gradual behavioral adjustments based on environmental cues and predictive learning.

Conversely, DQL, which embodies a value-based learning paradigm, concentrates on maximizing immediate rewards through Q-value estimations. This approach inherently prioritizes short-term outcomes and can restrict the algorithm's capacity for strategic foresight necessary for anticipatory behaviors. The primary reliance on an ϵ-greedy strategy

for exploration may not provide sufficient opportunity for the algorithm to encounter and learn from less immediately rewarding but strategically important actions. As a result, DQL lacks the flexibility to effectively adapt its policy based on potential future benefits, thus hindering its ability to perform tasks that require understanding and reacting to future conditions.

7.4 Evaluation Conclusion

This chapter provided a detailed comparison of PPO and DQL across several critical performance metrics within a smart home environment. This analysis covered temperature management, energy efficiency, adaptability to non-typical schedules, and anticipatory behaviors.

The results demonstrate PPO's superior adaptability, particularly in anticipating occupant behaviors and adjusting to variable schedules, which are essential for effective smart home management. PPO's ability to balance comfort with energy efficiency suggests its potential for real-world applications. In contrast, DQL, while robust in maintaining energy efficiency, showed limitations in scenarios requiring dynamic adaptation and predictive actions. Additionally, the DQL training took roughly 30% more time than PPO, across the same amount of episodes.

Overall, the review emphasizes that PPO's adaptable and proactive methodology makes it better suited for intricate, practical applications. Still, given DQL's particular strengths, it may be possible to improve it further or create a hybrid approach that combines elements of both algorithms. Lastly, this analysis highlights how reinforcement learning has great potential for developing smart home technology. It makes recommendations for further study and analysis to improve these algorithms for a wider and more efficient use.

Chapter 8

Conclusion

The overarching goal of this thesis was to explore the application of reinforcement learning techniques to enhance the management and optimization of smart home environments. Central to this research was the development and refinement of models designed to maintain comfortable living conditions while improving energy efficiency. These models were tasked with dynamically adapting to changes in user behavior and learning to anticipate the return of occupants, thereby ensuring optimal conditions upon their arrival.

This goal was accomplished through the implementation of a simulation environment that mimics realistic home dynamics and the development of two advanced reinforcement learning models: Deep Q-Learning (DQL) and Proximal Policy Optimization (PPO). These models were deployed within the simulation environment to critically assess their performance and their approaches to optimizing smart home management. The following evaluation analysed their ability to effectively manage energy usage and comfort, adapting dynamically to unexpected user behaviors and environmental conditions.

The results of these evaluations showed that both models were adept at adapting to the environment and regulating temperature to maintain both comfort and efficiency. Although both models handled unexpected occupancy events competently, the PPO model was found to be more suitable due to its more stable and faster learning capabilities. Additionally, the PPO model alone demonstrated the ability to anticipate an occupant's arrival and adjust the heating in advance.

Reflecting back, I have gained meaningful insight into both the potential and difficulties of applying artificial intelligence in practical settings. This experience has been very enriching, pushing the boundaries of my technical skills while deepening my appreciation for the nuances of machine learning and its real impacts in the world around us.

Looking ahead, exploring optimization techniques to enhance the predictive prowess of the DQL model would prove intriguing. Moreover, while this environment functioned as an effective proof of concept, crafting a more realistic and customized approach developed to seamlessly integrate these models into specific smart home environments leveraging an implemented Smart Home Adapter could yield benefits.

Bibliography

[1] *REST API Documentation* [online]. Home Assistant, November 2023 [cit. 2024-03-20]. Available at: https://developers.home-assistant.io/docs/api/rest/.

[2] Atzori, L., Iera, A. and Morabito, G. The Internet of Things: A survey. *Computer Networks*. Elsevier. 2010, vol. 54, no. 15, p. 2787–2805.

[3] Bottou, L. Large-Scale Machine Learning with Stochastic Gradient Descent. In: Lechevallier, Y. and Saporta, G., ed. *Proceedings of COMPSTAT'2010*. Heidelberg: Physica-Verlag HD, 2010, p. 177–186. ISBN 978-3-7908-2604-3.

[4] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J. et al. *OpenAI Gym* [online]. OpenAI, 2016 [cit. 2023-12-07]. Available at: https://gym.openai.com/.

[5] Doshi, K. *Reinforcement Learning Explained Visually* [online]. Towards Data Science, December 2020 [cit. 2023-12-13]. Available at: https://towardsdatascience.com/reinforcement-learning-explained-visually-part-5-deep-q-networks-step-by-step-5a5317197f4b.

[6] Energy, U. D. of. *Programmable Thermostats* [online]. U.S. Department of Energy [cit. 2023-04-15]. Available at: https://www.energy.gov/energysaver/thermostats.

[7] Goodfellow, I., Bengio, Y. and Courville, A. *Deep Learning*. 1st ed. MIT Press, November 2016. ISBN 0262035618.

[8] Hui, J. *RL Proximal Policy Optimization (PPO) Explained* [online]. Medium, September 2018 [cit. 2024-04-10]. Available at: https://jonathan-hui.medium.com/rl-proximal-policy-optimization-ppo-explained-77f014ec3f12.

[9] Inc., E. *Ecobee SmartThermostat: Comfort made smarter* [online]. 2018 [cit. 2024-04-15]. Available at: https://www.ecobee.com/en-us/smart-thermostats/.

[10] Kaelbling, L. P. et al. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*. 1996, vol. 4, p. 237–285. Available at: https://www.jair.org/index.php/jair/article/view/10166.

[11] Kingma, D. P. and Ba, J. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*. 2014. Available at: https://arxiv.org/abs/1412.6980.

[12] Konda, V. R. and Tsitsiklis, J. N. Actor-Critic Algorithms. *SIAM Journal on Control and Optimization*. Society for Industrial and Applied Mathematics. 2000, vol. 42, no. 4, p. 1143–1166. DOI: 10.1137/S0363012999363500.

[13] Labs, N. *Meet the Nest Learning Thermostat* [online]. 2013 [cit. 2024-04-15]. Available at: https://nest.com/thermostats/nest-learning-thermostat/overview/.

[14] LeCun, Y., Bengio, Y. and Hinton, G. Deep learning. *Nature*. Nature Publishing Group. 2015, vol. 521, p. 436–444. DOI: 10.1038/nature14539. ISSN 0028-0836. Available at: https://www.nature.com/articles/nature14539.

[15] Lee, S. and Choi, D.-H. Federated Reinforcement Learning for Energy Management of Multiple Smart Homes With Distributed Energy Resources. *IEEE Transactions on Industrial Informatics*. IEEE. November 2022, vol. 18, p. 488–497. DOI: 10.1109/TII.2021.3056015. ISSN 1551-3203. Available at: https://ieeexplore.ieee.org/document/9357127.

[16] Lillicrap, T. P. et al. Continuous control with deep reinforcement learning. *ArXiv*. 2015, abs/1509.02971. Available at: https://arxiv.org/abs/1509.02971.

[17] Long Ji, L. Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching. *Machine Learning*. Kluwer Academic Publishers. 1992, vol. 8, 3-4, p. 293–321. DOI: 10.1007/BF00992699. Available at: https://link.springer.com/article/10.1007/BF00992699.

[18] Meteoblue. *Historical Weather Data for Basel, Switzerland* [online]. Meteoblue [cit. 2024-01-19]. Available at: https://www.meteoblue.com/en/weather/archive/export.

[19] Mismar, F. B. *Structure of the neural network used for the Deep Q-learning Network implementation* [online]. ResearchGate, August 2018 [cit. 2024-02-19]. Available at: https://www.researchgate.net/figure/Structure-of-the-neural-network-used-for-the-Deep-Q-learning-Network-implementation-with_fig1_327050635.

[20] Mnih, V. et al. Asynchronous Methods for Deep Reinforcement Learning. *International Conference on Machine Learning*. June 2016. DOI: 10.5555/3045390.3045594. Available at: http://proceedings.mlr.press/v48/mniha16.html.

[21] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J. et al. Human-level control through deep reinforcement learning. *Nature*. Nature Publishing Group. February 2015, vol. 518, no. 7540, p. 529–533. Available at: https://doi.org/10.1038/nature14236.

[22] Morales, M. *Grokking Deep Reinforcement Learning*. 1st ed. Manning Publications, November 2020. ISBN 1617295450.

[23] OpenAI. Spinning Up in Deep RL. *Introduction to Policy Optimization* [online]. OpenAI, 2018 [cit. 2024-01-30]. Available at: https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html.

[24] OpenAI. *DALL-E 3* [online]. OpenAI, 2023 [cit. 2024-04-10]. Available at: https://openai.com/dall-e-3.

[25] Rumelhart, D. E., Hinton, G. E. and Williams, R. J. Learning representations by back-propagating errors. *Nature*. Nature Publishing Group. October 1986, vol. 323, p. 533–536. DOI: 10.1038/323533a0. ISSN 0028-0836. Available at: https://www.nature.com/articles/323533a0.

[26] Schulman, J. et al. Proximal Policy Optimization Algorithms. *ArXiv preprint arXiv:1707.06347*. August 2017, vol. 2. Available at: https://arxiv.org/abs/1707.06347.

[27] Sharma, S. *Activation Functions in Neural Networks* [online]. Towards Data Science, September 2017 [cit. 2024-01-29]. Available at: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[28] Soudari, M. et al. Predictive smart thermostat controller for heating, ventilation, and air-conditioning systems. *Proceedings of the Estonian Academy of Sciences*. Tallinn, Estonia: Teaduste Akadeemia Kirjastus (Estonian Academy Publishers). 2018, vol. 67, no. 3, p. 291–299. DOI: 10.3176/proc.2018.3.11. Accessed: 2024-01-30. Available at: https://www.kirj.ee/31932/?tpl=1061&c_tpl=1064.

[29] Srivastava, N. et al. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*. June 2014, vol. 15, p. 1929–1958. ISSN 1532-4435. Available at: http://jmlr.org/papers/v15/srivastava14a.html.

[30] Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. 2nd ed. MIT Press, 2018. ISBN 0262039249.

[31] Watkins, C. J. C. H. *Learning From Delayed Rewards* [online]. Cambridge, UK, 1989. [cit. 2024-04-22]. Dissertation. King's College, Cambridge. Available at: https://www.cs.rhul.ac.uk/~chrisw/thesis.html.

[32] Yu, E. Y. *Coding PPO from Scratch with PyTorch* [online]. Medium, September 2020 [cit. 2024-02-13]. Available at: https://medium.com/@eyyu/coding-ppo-from-scratch-with-pytorch-part-2-4-f9d8b8aa938a.

[33] Zhao, X. et al. *Applications of asynchronous deep reinforcement learning based on dynamic updating weights*. Springer, 2019. DOI: 10.1007/s10489-018-1296-x. Available at: https://www.researchgate.net/figure/The-general-framework-of-Actor-Critic_fig2_327512420.

Appendix A

Environment configuration

Configuration used for algorithm evaluation and testing:

```
"settings": {
  'minutes_per_step': 15,
  'outside_temperature_record_step': 60,
  'temperature_data_file': 'data/basel_10_years_hourly.csv'},
"temperature": {
  'starting_temperature': 20,
  'target_temperature': 22,
  'insulation_quality': 0.95,
  'heater_at_max': 40,
  'heating_meter_at_max': 5,
  'cooler_at_max': 5,
  'hvac_efficiency': 0.05,
  'meter_step': 0.25},
"schedule": {
  "people": ["p1"],
  "weekly_schedule": {
    "Monday": {"p1": {"leave": "08:00", "return": "18:00", "variance": 15}},
    "Tuesday": {"p1": {"leave": "08:00", "return": "18:00", "variance": 15}},
    "Wednesday": {"p1": {"leave": "08:00", "return": "18:00", "variance": 15}},
    "Thursday": {"p1": {"leave": "08:00", "return": "18:00", "variance": 15}},
    "Friday": {"p1": {"leave": "08:00", "return": "18:00", "variance": 15}},
    "Saturday": {"p1": {"at_home": "true"}},
    "Sunday": {"p1": {"at_home": "true"}},},
  'random_event_chance': 0.1,
  'random_event': {
    'sick-day': 0.10,
    'early-return': 0.2,
    'late-return': 0.2,
    'early-leave': 0.2,
    'late-leave': 0.2,
    'out-of-town': 0.10},
  'random_event_max_duration': 180,}
```

Appendix B

Temperature Data in Simulation

A small csv dataset of historical temperature readings from Basel, Switzerland, containing timestamps (`YYYYMMDDTHHMM`) and corresponding temperatures (°C).

```
...
20220118T0000,1.0402454
20220118T0100,0.7802454
20220118T0200,0.5602454
20220118T0300,0.6802454
20220118T0400,0.64024544
20220118T0500,0.6902454
20220118T0600,0.6902454
20220118T0700,0.89024544
20220118T0800,1.4302454
20220118T0900,2.5702453
20220118T1000,3.7902455
20220118T1100,4.6202455
20220118T1200,5.630245
20220118T1300,5.9302454
20220118T1400,6.0602455
20220118T1500,5.610245
20220118T1600,3.8502455
20220118T1700,3.1402454
20220118T1800,2.6002455
20220118T1900,1.7802454
20220118T2000,1.2102454
20220118T2100,1.3902454
20220118T2200,0.9002454
20220118T2300,0.9302454
20220119T0000,1.2102454
...
```

Appendix C

Parameter Tuning Results

Network Layers	Score	Network Layers	Score	Network Layers	Score
32:128:256	1053.49	256:256:32	-271.44	128:32:32	-1310.58
128:32:256	762.22	32:64:256	-271.61	64:64	-1483.49
256:128:128	686.43	64:256:128	-282.16	256:32:32	-1502.96
128:256:64	588.22	128:256:256	-283.08	256:128:64	-1507.83
128:256:128	537.18	256:32:128	-297.83	64	-1551.85
256:256:64	512.70	64:32:128	-327.19	64:32:32	-1606.89
32:256:32	458.68	32:32:256	-329.41	32:32:32	-1750.98
32:256:256	381.09	64:128:256	-341.76	128:128	-1860.68
256:32	321.78	32:32:64	-374.04	64:256:32	-2064.00
128:128:64	311.43	64:128:32	-383.58	256:64:128	-2227.97
128:64:32	186.16	128:128:128	-388.65	256:256:128	-2747.66
32:256:128	158.75	256:64:256	-388.76	256	-2803.21
128:64:256	127.12	256:64:64	-399.80	64:256	-2882.71
128:32:64	110.81	64:256:256	-404.84	64:32:256	-3038.53
32:128:64	109.35	64:64:256	-481.70	256:256	-4312.45
32:128:128	98.91	128:64:128	-495.22	256:256:256	-4844.81
64:256:64	70.92	64:32	-508.61	128	-11326.28
128:128:256	68.23	128:32:128	-520.56	256:64	-11589.10
128:64:64	48.67	32:64:128	-563.81	32	-11589.10
32:128:32	20.25	32:64:32	-575.17	64:128:128	-11589.10
64:128:64	-73.64	32:256:64	-638.15	32:64	-11593.89
128:256	-96.83	256:32:256	-692.75	32:256	-11596.29
32:32:128	-125.17	256:64:32	-751.31	64:128	-11597.49
128:32	-141.99	64:64:64	-800.51	128:64	-11610.96
256:32:64	-172.47	32:32	-964.58		
64:64:128	-189.53	32:128	-980.74		
128:256:32	-202.09	32:64:64	-988.27		
256:128:32	-214.96	256:128:256	-1069.36		
256:128	-217.26	64:32:64	-1211.56		
128:128:32	-252.26	64:64:32	-1250.79		

Table C.1: PPO: Performance of Network Layer Combinations.

Network Layers	Score	Network Layers	Score	Network Layers	Score
32:256	-220.92	128:32:64	-2017.45	32:128	-3569.67
32:32:128	-415.15	256:32:32	-2091.22	32:256:256	-3595.8
256:128:256	-517.21	32:64:64	-2107.57	64:32	-3717.57
32:64:32	-904.48	32:32:64	-2115.63	64:64	-3963.38
32:64	-912.12	256:256:64	-2134.44	128:64:32	-4096.07
64:32:256	-952.51	32:32:32	-2180.98	32:128:128	-4219.97
64:256:64	-953.06	128:256:64	-2249.5	128:128:128	-4296.39
128:32:256	-970.95	32	-2338.35	256:128:128	-4389.15
64:128:32	-1016.24	256:32:64	-2418.74	64:256:128	-4649.28
64:128:128	-1017.16	64:32:32	-2433.67	128:64:64	-4832.71
64:32:64	-1126.67	32:32:256	-2493.51	128:128:32	-4928.41
256:64:32	-1145.94	64:64:128	-2502.71	64:32:128	-5086.66
64:64:32	-1219.11	256:128:64	-2704.02	128:64:128	-5121.68
256:128:32	-1343.23	32:128:64	-2859.39	256:256:32	-5185.89
64:64:64	-1355.17	256:64:256	-2880.79	256:64	-5187.42
128:64:256	-1528.08	64:64:256	-2977.11	64:256:32	-5293.0
64	-1536.71	256:32:128	-3019.66	64:128	-5699.99
128:128:64	-1566.17	32:64:256	-3039.4	64:256	-5752.42
32:32	-1610.54	64:128:64	-3056.13	256:64:64	-5853.35
128	-1643.38	128:64	-3077.94	128:256:256	-9597.62
64:256:256	-1688.85	32:128:256	-3091.94	128:128:256	-11274.01
32:256:32	-1794.28	128:32	-3192.72	256:32:256	-11274.01
64:128:256	-1799.48	128:32:32	-3204.9	256:256:128	-11274.01
256:32	-1887.25	128:128	-3292.61	128:256:256	-11274.01
32:256:64	-1899.95	32:128:32	-3299.39	128:256	-11323.7
32:256:128	-1917.25	256:32	-3309.35	256:256	-11323.7
128:256:32	-1923.76	256:128	-3365.99	256	-11323.7
128:32:128	-1973.57	32:64:128	-3559.21	256:256:256	-11323.7

Table C.2: DQL: Performance of Network Layer Combinations.

LR	BS	DF	Freq	In.	Score	LR	BS	DF	Freq	In.	Score	LR	BS	DF	Freq	In.	Score
0.0005	128.0	0.99	4.0	0.025	-560.12	0.0005	64.0	0.95	4.0	0.001	-2295.56	0.0005	128.0	0.99	8.0	0.025	-4874.83
0.0005	96.0	0.95	4.0	0.025	-918.97	0.0005	96.0	0.95	16.0	0.005	-2303.49	0.0005	96.0	0.99	4.0	0.001	-4907.78
0.0005	96.0	0.95	4.0	0.001	-928.79	0.0005	128.0	0.9	8.0	0.025	-2330.19	0.0015	96.0	0.95	16.0	0.001	-5073.61
0.0005	128.0	0.95	16.0	0.001	-1002.38	0.0005	96.0	0.99	16.0	0.025	-2363.23	0.0005	96.0	0.9	16.0	0.001	-5196.79
0.0015	96.0	0.95	16.0	0.005	-1064.18	0.0005	96.0	0.95	4.0	0.005	-2397.18	0.0005	128.0	0.99	4.0	0.005	-5439.61
0.0005	96.0	0.99	8.0	0.025	-1226.27	0.0005	128.0	0.95	8.0	0.005	-2441.56	0.0015	96.0	0.9	8.0	0.001	-5448.86
0.0005	64.0	0.9	8.0	0.001	-1229.17	0.0005	128.0	0.99	16.0	0.025	-2450.97	0.0015	64.0	0.9	16.0	0.005	-9204.97
0.0005	64.0	0.99	4.0	0.001	-1306.59	0.0005	128.0	0.9	8.0	0.001	-2478.13	0.0005	64.0	0.95	16.0	0.005	-9204.97
0.0005	96.0	0.99	4.0	0.005	-1395.87	0.0005	96.0	0.9	4.0	0.025	-2488.6	0.0015	128.0	0.95	8.0	0.005	-9360.28
0.0005	128.0	0.99	16.0	0.005	-1418.71	0.0005	128.0	0.95	16.0	0.025	-2500.31	0.0015	128.0	0.99	8.0	0.025	-9360.28
0.0005	64.0	0.9	4.0	0.025	-1421.88	0.0005	128.0	0.9	16.0	0.005	-2520.98	0.0015	96.0	0.9	16.0	0.025	-9466.78
0.0005	128.0	0.99	8.0	0.001	-1428.38	0.0005	96.0	0.9	4.0	0.001	-2601.8	0.0015	64.0	0.95	8.0	0.025	-9521.85
0.0005	64.0	0.99	4.0	0.025	-1451.4	0.0005	96.0	0.99	8.0	0.001	-2611.13	0.0015	96.0	0.99	8.0	0.001	-9580.78
0.0005	64.0	0.9	4.0	0.001	-1476.92	0.0005	128.0	0.95	4.0	0.025	-2625.59	0.0015	96.0	0.95	8.0	0.005	-9580.78
0.0005	96.0	0.9	8.0	0.005	-1495.36	0.0005	128.0	0.95	16.0	0.005	-2628.2	0.0015	96.0	0.95	4.0	0.001	-9595.74
0.0005	64.0	0.99	4.0	0.005	-1500.14	0.0005	128.0	0.9	16.0	0.001	-2640.31	0.0015	128.0	0.9	4.0	0.001	-9604.8
0.0005	96.0	0.9	16.0	0.005	-1565.59	0.0005	96.0	0.99	8.0	0.005	-2706.44	0.0015	128.0	0.9	4.0	0.005	-9604.8
0.0015	128.0	0.95	16.0	0.001	-1595.65	0.0005	96.0	0.99	16.0	0.005	-2716.05	0.0015	128.0	0.99	4.0	0.005	-9604.8
0.0005	128.0	0.9	4.0	0.005	-1597.13	0.0005	96.0	0.95	8.0	0.001	-2746.18	0.0015	64.0	0.95	4.0	0.005	-9682.71
0.0005	128.0	0.95	4.0	0.005	-1656.34	0.0005	96.0	0.99	16.0	0.001	-2895.92	0.0015	128.0	0.95	8.0	0.025	-11274.54
0.0005	64.0	0.9	16.0	0.025	-1702.33	0.0005	64.0	0.95	8.0	0.001	-3003.31	0.0015	96.0	0.99	4.0	0.005	-11277.43
0.0005	64.0	0.99	8.0	0.001	-1715.28	0.0005	96.0	0.9	8.0	0.025	-3010.72	0.0015	96.0	0.99	4.0	0.025	-11277.43
0.0005	64.0	0.9	8.0	0.005	-1715.98	0.0015	96.0	0.99	16.0	0.005	-3037.06	0.0015	96.0	0.9	4.0	0.005	-11277.43
0.0005	64.0	0.95	8.0	0.025	-1788.68	0.0005	64.0	0.9	8.0	0.025	-3049.64	0.0005	64.0	0.99	16.0	0.001	-11280.63
0.0005	128.0	0.9	8.0	0.005	-1821.67	0.0005	96.0	0.9	8.0	0.001	-3134.56	0.0005	64.0	0.95	16.0	0.001	-11280.63
0.0005	128.0	0.95	4.0	0.001	-1854.06	0.0005	96.0	0.95	8.0	0.005	-3355.13	0.0015	64.0	0.95	16.0	0.001	-11280.63
0.0005	64.0	0.99	8.0	0.025	-1889.98	0.0005	128.0	0.95	8.0	0.025	-3368.69	0.0015	96.0	0.99	8.0	0.025	-11281.99
0.0005	128.0	0.99	8.0	0.005	-1918.44	0.0005	64.0	0.9	16.0	0.005	-3486.28	0.0015	96.0	0.95	8.0	0.001	-11281.99
0.0005	64.0	0.99	8.0	0.005	-1920.99	0.0005	96.0	0.95	16.0	0.001	-3515.83	0.0015	128.0	0.9	16.0	0.005	-11285.04
0.0005	96.0	0.95	8.0	0.025	-1933.57	0.0015	128.0	0.95	4.0	0.005	-3588.06	0.0015	128.0	0.95	16.0	0.025	-11285.04
0.0005	64.0	0.95	8.0	0.005	-1952.89	0.0005	96.0	0.99	4.0	0.025	-3821.88	0.0015	64.0	0.9	4.0	0.025	-11286.18
0.0015	96.0	0.95	16.0	0.025	-1985.45	0.0005	64.0	0.95	4.0	0.025	-4026.09	0.0015	64.0	0.95	4.0	0.001	-11286.18
0.0005	128.0	0.95	8.0	0.001	-2031.12	0.0005	64.0	0.9	16.0	0.001	-4071.46	0.0015	96.0	0.99	16.0	0.001	-11290.24
0.0005	128.0	0.9	16.0	0.025	-2035.58	0.0005	128.0	0.9	4.0	0.001	-4302.36	0.0015	96.0	0.99	16.0	0.025	-11290.24
0.0005	96.0	0.95	16.0	0.025	-2075.77	0.0005	64.0	0.9	4.0	0.005	-4345.98	0.0015	128.0	0.99	4.0	0.001	-11303.02
0.0005	64.0	0.99	16.0	0.005	-2084.16	0.0005	128.0	0.99	4.0	0.001	-4363.6	0.0015	128.0	0.9	8.0	0.005	-11319.05
0.0005	128.0	0.9	4.0	0.025	-2161.72	0.0015	96.0	0.9	16.0	0.005	-4618.48	0.0015	128.0	0.9	8.0	0.025	-11319.05
0.0005	64.0	0.95	16.0	0.025	-2187.7	0.0005	96.0	0.9	4.0	0.005	-4642.17	0.0015	128.0	0.99	8.0	0.001	-11319.05
0.0005	128.0	0.99	16.0	0.001	-2198.58	0.0015	96.0	0.9	16.0	0.005	-4686.47	0.0015	128.0	0.99	8.0	0.005	-11319.05
0.0005	96.0	0.9	16.0	0.025	-2212.64	0.0005	64.0	0.95	4.0	0.005	-4701.66	0.0015	96.0	0.99	4.0	0.001	-11321.49

Table C.3: DQL: Performance of HP combinations. 3.2.2

LR	DF	BS	Up/Iter	Clip	Score	LR	DF	BS	Up/Iter	Clip	Score	LR	DF	BS	Up/Iter	Clip	Score
0.0005	0.95	4096.0	20.0	0.3	2057.9	0.0005	0.95	4096.0	10.0	0.1	1056.7	0.0015	0.9	1024.0	20.0	0.2	1.11
0.0015	0.95	4096.0	10.0	0.2	1951.26	0.0005	0.95	2048.0	5.0	0.2	1011.78	0.0015	0.99	2048.0	5.0	0.3	-18.42
0.0005	0.95	1024.0	10.0	0.3	1849.28	0.0005	0.9	1024.0	20.0	0.3	987.46	0.0005	0.99	1024.0	10.0	0.1	-64.7
0.0015	0.95	1024.0	10.0	0.3	1656.19	0.0005	0.95	1024.0	5.0	0.1	985.34	0.0005	0.99	2048.0	10.0	0.1	-77.86
0.0005	0.9	4096.0	20.0	0.3	1643.64	0.0005	0.95	4096.0	10.0	0.2	969.5	0.0015	0.95	1024.0	20.0	0.3	-254.82
0.0005	0.9	4096.0	20.0	0.2	1634.75	0.0015	0.9	4096.0	20.0	0.3	906.86	0.0015	0.95	2048.0	20.0	0.1	-264.97
0.0015	0.95	2048.0	10.0	0.3	1623.15	0.0015	0.9	1024.0	5.0	0.2	896.51	0.0015	0.99	2048.0	10.0	0.2	-304.28
0.0005	0.95	2048.0	10.0	0.2	1557.64	0.0015	0.9	2048.0	20.0	0.1	891.52	0.0005	0.95	1024.0	20.0	0.3	-340.64
0.0015	0.95	4096.0	20.0	0.1	1533.32	0.0015	0.95	2048.0	20.0	0.2	868.68	0.0015	0.99	2048.0	5.0	0.2	-360.28
0.0015	0.9	2048.0	10.0	0.2	1531.65	0.0005	0.9	1024.0	20.0	0.2	833.02	0.0005	0.99	2048.0	20.0	0.1	-380.52
0.0005	0.9	2048.0	20.0	0.2	1513.35	0.0015	0.95	2048.0	5.0	0.1	804.09	0.0015	0.95	2048.0	20.0	0.3	-418.63
0.0005	0.95	2048.0	10.0	0.1	1468.54	0.0015	0.9	2048.0	20.0	0.3	763.44	0.0005	0.99	2048.0	5.0	0.3	-433.41
0.0005	0.95	4096.0	20.0	0.2	1457.08	0.0015	0.9	2048.0	20.0	0.2	759.14	0.0015	0.99	2048.0	10.0	0.1	-443.25
0.0005	0.9	4096.0	20.0	0.1	1447.96	0.0015	0.99	4096.0	10.0	0.3	753.74	0.0005	0.95	4096.0	5.0	0.1	-483.71
0.0005	0.95	1024.0	10.0	0.2	1441.33	0.0005	0.9	1024.0	10.0	0.3	745.95	0.0005	0.95	4096.0	5.0	0.2	-544.99
0.0015	0.9	4096.0	20.0	0.1	1431.65	0.0015	0.9	1024.0	10.0	0.2	684.77	0.0015	0.95	4096.0	20.0	0.3	-555.91
0.0005	0.9	2048.0	20.0	0.3	1427.42	0.0005	0.99	4096.0	20.0	0.2	652.71	0.0015	0.99	4096.0	10.0	0.2	-599.79
0.0005	0.99	4096.0	20.0	0.3	1418.94	0.0015	0.9	2048.0	10.0	0.1	649.02	0.0015	0.95	1024.0	20.0	0.1	-756.72
0.0015	0.9	4096.0	20.0	0.2	1382.5	0.0005	0.99	1024.0	20.0	0.2	590.87	0.0005	0.99	2048.0	20.0	0.3	-797.32
0.0005	0.95	2048.0	20.0	0.3	1380.04	0.0005	0.99	2048.0	20.0	0.2	585.1	0.0015	0.95	1024.0	20.0	0.2	-805.65
0.0005	0.95	4096.0	20.0	0.1	1373.82	0.0005	0.95	1024.0	5.0	0.3	571.61	0.0015	0.99	2048.0	5.0	0.1	-823.72
0.0005	0.95	2048.0	20.0	0.1	1326.53	0.0015	0.95	1024.0	10.0	0.1	513.8	0.0015	0.9	1024.0	5.0	0.1	-837.16
0.0015	0.9	4096.0	10.0	0.1	1312.55	0.0015	0.99	4096.0	20.0	0.2	482.0	0.0005	0.99	1024.0	5.0	0.1	-850.9
0.0005	0.9	2048.0	10.0	0.1	1288.92	0.0005	0.9	4096.0	10.0	0.1	446.09	0.0005	0.99	4096.0	10.0	0.3	-860.98
0.0005	0.9	1024.0	10.0	0.1	1276.26	0.0005	0.99	2048.0	5.0	0.2	432.33	0.0005	0.99	4096.0	10.0	0.1	-902.05
0.0005	0.99	2048.0	10.0	0.2	1268.09	0.0005	0.95	2048.0	5.0	0.1	377.72	0.0015	0.99	4096.0	5.0	0.3	-914.28
0.0005	0.95	1024.0	10.0	0.1	1249.88	0.0015	0.95	4096.0	20.0	0.2	305.41	0.0005	0.99	4096.0	20.0	0.1	-981.23
0.0005	0.99	2048.0	10.0	0.3	1231.48	0.0005	0.95	1024.0	20.0	0.2	277.97	0.0015	0.9	2048.0	5.0	0.1	-1016.65
0.0015	0.9	1024.0	10.0	0.3	1212.17	0.0015	0.9	1024.0	20.0	0.3	270.91	0.0005	0.9	1024.0	5.0	0.1	-1028.15
0.0015	0.95	2048.0	5.0	0.2	1211.41	0.0005	0.9	1024.0	10.0	0.2	269.35	0.0015	0.95	1024.0	10.0	0.2	-1105.68
0.0015	0.95	2048.0	10.0	0.1	1207.58	0.0015	0.9	1024.0	10.0	0.1	226.61	0.0015	0.99	2048.0	10.0	0.3	-1350.74
0.0015	0.95	2048.0	10.0	0.2	1204.0	0.0005	0.9	1024.0	20.0	0.1	200.27	0.0015	0.99	1024.0	10.0	0.3	-1418.02
0.0015	0.95	4096.0	10.0	0.3	1190.61	0.0005	0.95	2048.0	20.0	0.2	182.88	0.0005	0.99	1024.0	20.0	0.3	-1454.96
0.0015	0.9	2048.0	10.0	0.3	1143.47	0.0005	0.99	1024.0	10.0	0.2	180.82	0.0005	0.99	1024.0	5.0	0.2	-1475.41
0.0005	0.95	2048.0	10.0	0.3	1117.82	0.0015	0.95	1024.0	5.0	0.3	168.09	0.0005	0.99	1024.0	10.0	0.3	-1481.64
0.0005	0.9	2048.0	20.0	0.1	1112.47	0.0015	0.9	4096.0	10.0	0.2	164.82	0.0015	0.99	4096.0	10.0	0.1	-1514.97
0.0015	0.95	4096.0	10.0	0.1	1099.71	0.0015	0.95	1024.0	5.0	0.2	81.07	0.0005	0.99	1024.0	20.0	0.1	-1545.15
0.0015	0.95	1024.0	5.0	0.1	1097.07	0.0005	0.95	1024.0	20.0	0.1	19.63	0.0015	0.9	1024.0	20.0	0.1	-1546.49
0.0005	0.95	4096.0	10.0	0.3	1066.93	0.0005	0.99	4096.0	10.0	0.2	15.08	0.0005	0.99	4096.0	5.0	0.2	-1556.07
0.0015	0.95	4096.0	5.0	0.2	1062.91	0.0015	0.99	1024.0	10.0	0.2	9.62	0.0005	0.9	2048.0	10.0	0.2	-1584.42

Table C.4: PPO: Performance of HP combinations. 3.3.3

DQL: Performance of the best-performing HP and NN configurations.

Layers	Learning Rate	Minibatch	DiscountF.	L.Freq	Interp.	Score
[32:64:32]	0.0005	128	0.99	4	0.025	-138.49
[32:32:128]	0.0015	96	0.95	16	0.005	-240.74
[32:64]	0.0005	128	0.99	4	0.025	-506.26
[32:256]	0.0005	128	0.99	4	0.025	-560.12
[32:256]	0.0005	96	0.95	4	0.025	-918.97
[32:256]	0.0005	96	0.95	4	0.001	-928.79
[32:256]	0.0005	128	0.95	16	0.001	-1002.38
[32:256]	0.0015	96	0.95	16	0.005	-1064.18
[32:64]	0.0005	96	0.95	4	0.001	-1383.55
[256:128:256]	0.0005	128	0.95	16	0.001	-2368.71
[32:64]	0.0005	128	0.95	16	0.001	-2762.79
[32:64:32]	0.0015	96	0.95	16	0.005	-2961.09
[32:64:32]	0.0005	96	0.95	4	0.025	-3045.01
[32:32:128]	0.0005	128	0.95	16	0.001	-3481.06
[32:32:128]	0.0005	96	0.95	4	0.025	-3687.24
[32:64:32]	0.0005	128	0.95	16	0.001	-3696.9
[32:64:32]	0.0005	96	0.95	4	0.001	-3972.55
[32:32:128]	0.0005	96	0.95	4	0.001	-4274.4
[32:64]	0.0005	96	0.95	4	0.025	-4649.27
[32:32:128]	0.0005	128	0.99	4	0.025	-5978.72
[32:64]	0.0015	96	0.95	16	0.005	-6005.56
[256:128:256]	0.0005	96	0.95	4	0.025	-9595.74
[256:128:256]	0.0005	128	0.99	4	0.025	-9604.8
[256:128:256]	0.0005	96	0.95	4	0.001	-11321.49
[256:128:256]	0.0015	96	0.95	16	0.005	-11334.81

PPO: Performance of the best-performing HP and NN configurations.

Layers	Learning Rate	Discount Factor	Batch Size	Ups/Iter	Clip	Score
[32:128:256]	0.0005	0.95	4096	20	0.3	2057.9
[32:128:256]	0.0015	0.95	4096	10	0.2	1951.26
[128:32:256]	0.0005	0.95	4096	20	0.3	1919.0
[32:128:256]	0.0005	0.95	1024	10	0.3	1849.28
[128:32:256]	0.0005	0.9	4096	20	0.3	1829.31
[256:128:128]	0.0005	0.9	4096	20	0.3	1767.85
[32:128:256]	0.0015	0.95	1024	10	0.3	1656.19
[32:128:256]	0.0005	0.9	4096	20	0.3	1643.64
[128:256:128]	0.0005	0.9	4096	20	0.3	1609.56
[128:256:64]	0.0005	0.9	4096	20	0.3	1563.49
[256:128:128]	0.0015	0.95	4096	10	0.2	1553.21
[128:32:256]	0.0015	0.95	4096	10	0.2	1462.37
[128:256:128]	0.0015	0.95	4096	10	0.2	1386.53
[128:256:128]	0.0005	0.95	4096	20	0.3	1340.44
[128:256:64]	0.0005	0.95	4096	20	0.3	1310.54
[128:256:128]	0.0005	0.95	1024	10	0.3	1282.67
[256:128:128]	0.0005	0.95	1024	10	0.3	1198.23
[128:256:64]	0.0015	0.95	4096	10.0	0.2	1156.05
[256:128:128]	0.0005	0.95	4096	20	0.3	1090.6
[128:32:256]	0.0015	0.95	1024	10	0.3	1047.36
[128:32:256]	0.0005	0.95	1024	10	0.3	1022.21
[128:256:64]	0.0005	0.95	1024	10	0.3	854.92
[128:256:64]	0.0015	0.95	1024	10	0.3	267.53
[256:128:128]	0.0015	0.95	1024	10	0.3	140.21

Appendix D

Alternative Evaluation Results

Seed	20 years	40 years	60 years	80 years	100 years	Avg
Seed 1	-1473.87	-4593.00	-4544.84	-2571.20	-2571.20	-3150.82
Seed 2	-2469.42	-1155.93	-3831.09	-683.55	-683.55	-1964.71
Seed 3	-3684.60	-5465.59	-5952.23	-4109.54	-4109.54	-4664.30
Seed 4	-5293.44	-2584.35	-2845.69	-3418.44	-3418.44	-3512.07
Seed 5	-565.75	-4288.26	-1830.64	-2267.19	-2267.19	-2243.81
Avg	-2697.42	-3617.43	-3790.90	-2429.98	-2429.98	

Table D.1: DQL: Performance overview in multiple environments of the best configuration from section 6.3.

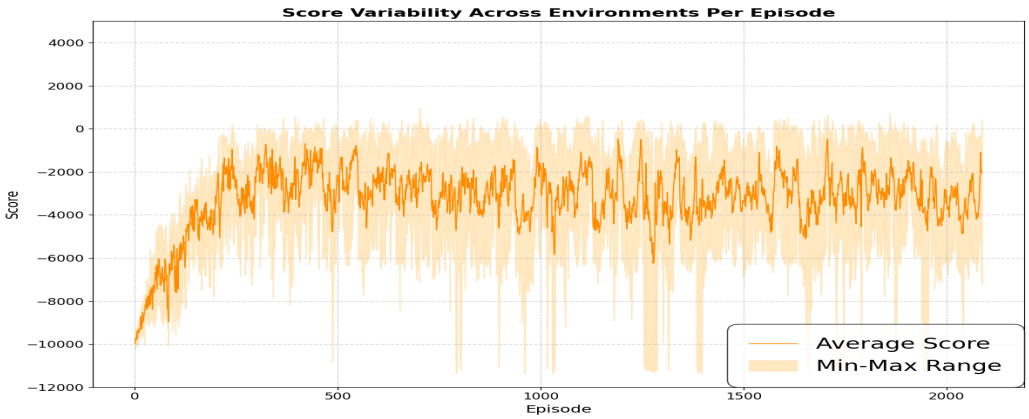

Figure D.1: DQL: Stability overview in multiple environments of the best configuration from section 6.3.

Appendix E

Contents of the Attached Media

This appendix provides an overview of the file structure for the supplementary material included with the thesis. The structure includes source code, documentation, and data results from experiments.

Root Directory

- **README.md** - Instructions and general information about the repository.
- **xbielg00.pdf** - This document.

doc (Documentation Diagrams)

results (Data from Experiments)

- **README.md** - Descriptions of the results and data structure.
- **evaluation** - Evaluation results of the models.
- **parameter_tuning** - Results from the tuning of model parameters.

src (Source Code)

- **algorithms** - Implementations of the DQL and PPO algorithms.
- **data** - CSV files containing data used in experiments.
- **env** - Environment modules and configurations for the simulation.
- **main.py** - Main script to run the project. Enables training and testing.
- **run_configurations.py** - Script that runs all configurations in a folder.
- **run_evaluation.py** - Script for final evaluation of configuration.
- **makefile**, **requirements.txt** - Build script and dependencies.
- **tools** - Utility scripts including argument parsing and configuration management.