MASTER THESIS

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at the University of Applied Sciences Technikum Wien
Degree Program Renewable Urban Energy systems

Multi-objective optimization of building control to minimize operational cost with a machine learning approach

Andreas Rippl, BSc
Student Number: 1810578009

Supervisor 1: Simon Schneider, MSc
Supervisor 2: Christoph Gehbauer, MSc

Berkeley, December 21, 2020

FH University of Applied Sciences TECHNIKUM WIEN

Declaration of Authenticity

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (see Urheberrechtsgesetz/ Austrian copyright law as amended as well as the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I further declare that up to this date I have not published the work to hand, nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.”

Place, Date | Signature

Abstract

The transition to more renewable energy sources, will become an ever-greater challenge in the future. Herein, one critical issue, is the natural volatile generation of sun and wind powered power plants, which strain the power grid during peak generation times. Buildings as flexible consumers can help to relieve the stress on the power grids without having them significantly increase their capacity. Current control systems are often realized with proportional-integral-derivative (PID) controllers do not react to predictions. Whereas Model-predictive-controllers (MPC) represent an innovation in control, as they can, adapt the control strategy to the needs of the grid by means of weather forecasts and predictions on the occupancy of people. Nevertheless, their application requires a high accuracy regarding the thermal model of the building. These two state-of-the art controllers are compared with the developed controller in this thesis.

A so-called agent with reinforcement learning (RL) is trained to learn the necessary rules to control the room temperature in an office building. RL refers to the fact that the agent improves itself with the help of experience as it takes over the control. The goal of this thesis is to develop an agent that controls the heating and cooling system in a room of a new office building in Berkeley, California and the tint of the electrochromic window used for shading. The room is represented as a resistance and capacitance (RC) model and is controlled by the agent with the goal of minimizing operating costs for heating, cooling, artificial lighting and office equipment. The performance of the agent is compared to a PID controller and a perfect information MPC. Past studies of RL-algorithms have shown the potential of the for this thesis chosen Deep Deterministic Policy Gradient (DDPG), in regard to the typical RL benchmark games. The agent interacts with the RC-model during a training process, where the agent learns how to operate the HVAC-system and dynamic façade to gain the highest reward based on the reward function including the total energy and demand costs and any violation of room temperature boundary. The goal of the agent is to maximize the reward over all possible timesteps.

To enable a foresightedness, the agent uses the weather forecast, electricity tariff information and information about occupancy for the next 4 hours. An agent with a DDPG-algorithm in combination with a multi-layer perceptron network succeeds in its primary task of ensuring the room temperature but is not farsighted enough to lower the maximum demand, what leads to high demand costs. The further improvement with four hours of forecast data as inputs and a reward system based on multiple steps lead to a behavior where the agent precools and preheats the room with lower the peak load and therefore lower demand and total operation costs. The best network configurations and settings for the reward system are found with a gridsearch, where all preselected settings are combined in all variants.

The final trained agent is based on a DDPG algorithm in combination with a multi-layer perceptron network with three hidden layers with a layer size of 400 of the first hidden layer and 300 of the following hidden layers. A Gaussian noise process is used for exploration as the action noise and for the sampling of the training data a High-Value Prioritization Experience Replay Buffer is used. The PI controller as a benchmark controller is outperformed by the agent in terms of the optimization goals with total cost savings in a test week starting on August 1st of 11.49 $ (30.21%). A "perfect information" model as MPC, optimizes the room over the entire period and minimizes the energy costs compared to the PI controller by 54.02 %, which saves 20.55 $ in the test week. Compared to the MPC the operation costs with the agents are 9.06 $ (44.12%) higher.

Keywords: Control, Building Technologies, Electrochromic Window, Machine Learning (ML), Reinforcement Learning (RL)

Kurzfassung

Der Übergang zu mehr erneuerbaren Energiequellen wird in Zukunft zu einer immer größeren Herausforderung werden. Ein kritischer Punkt ist dabei die natürliche volatile Erzeugung von sonnen- und windbetriebenen Kraftwerken, die das Stromnetz zu Spitzenerzeugungszeiten besonders belasten. Gebäude als flexible Verbraucher können dazu beitragen, die Stromnetze zu entlasten, ohne deren Kapazität wesentlich erhöhen zu müssen. Regelungssysteme werden derzeit oft mit Proportional-Integral-Derivativ-(PID)-Reglern realisiert, können nicht auf Vorhersagen reagieren. Daher stellen Modell-Prädiktive Regler (MPC) eine Regelungsinnovation dar, da sie in der Lage sind, die Regelungsstrategie durch Berücksichtigung von Wettervorhersagen und Vorhersagen über die Belegung von Personen an die Bedürfnisse des Stromnetzes anzupassen. Ihre Anwendung erfordert jedoch eine hohe Genauigkeit der Gebäudemodells. Diese beiden derzeit angewendeten Regler werden mit dem in dieser Arbeit entwickelten Regler verglichen.

Ein sogenannter Agent mit Reinforcement Learning (RL) wird trainiert, um die notwendigen Regeln zur Regelung der Raumtemperatur in einem Bürogebäude zu erlernen. RL bezieht sich auf die Tatsache, dass sich der Agent mit Hilfe von eigener gesammelter Erfahrung verbessert, während er die Regelung übernimmt. Das Ziel dieser Arbeit ist es, einen Agenten zu entwickeln, der das Heiz- und Kühlsystem in einem Raum eines neuen Bürogebäudes in Berkeley, Kalifornien, sowie die Verschattung mittels elektrochromen Fensters regelt. Der Raum wird als Widerstands- und Kapazitätsmodell (RC-Modell) dargestellt und durch den Agenten geregelt, mit dem Ziel, die Betriebskosten für Heizung, Kühlung, künstliche Beleuchtung und Bürogeräte im Vergleich zu einem PID-Regler und einem perfect information-MPC zu minimieren. Bereits durchgeführte Studien von RL-Algorithmen haben das Potential des Deep Deterministic Policy Gradient (DDPG), der als Algorithmus für den Agenten gewählt wird, in den zum Benchmark verwendeten Spielen gezeigt. Der Agent interagiert mit dem RC-Modell während eines Trainingsprozesses, in dem der Agent lernt, wie das HVAC-System und die dynamische Fassade zu regeln sind, um die höchstmögliche Belohnung zu erhalten. Die Belohungsfunktion beruht dabei einschließlich auf den gesamten Energie- und Bedarfskosten und jeder Über- oder Unterschreitung der Raumtemperaturgrenzen. Das Ziel des Agenten ist es, die Belohnung über alle möglichen Zeitschritte zu maximieren.

Der Agent verwendet dazu die Wettervorhersage, Informationen über den Stromtarif und die Personenbelegung, um dem Agenten eine Weitsichtigkeit zu ermöglichen. Der Agent mit einem DDPG-Algorithmus in Kombination mit einem multi-layer perceptron Netzwerk erfüllt seine primäre Aufgabe, die Raumtemperatur sicherzustellen, ist aber nicht weitsichtig genug, um die vom Stromnetz bezogene Spitzenleistung zu senken, was zu hohen Netznutzungsentgelten führt. Die weitere Verbesserung mit vier Stunden Vorhersagedaten als Input und ein auf mehreren Schritten basierendes Belohnungssystem führen zu einem

Verhalten, bei dem der Agent den Raum vorkühlt und vorheizt, um die Netznutzungsentgelte zu senken. Die besten Konfigurationen des Neuronalen Netzes und Einstellungen für das Belohnungssystem werden mit einer Rastersuche gesucht, bei der alle vorgewählten Einstellungen in allen Varianten kombiniert werden.

Der endgültig ausgebildete Agent basiert auf einem DDPG-Algorithmus in Kombination mit einem multi-layer perceptron Netzwerk mit zwei versteckten Layern mit einer Layergröße von 400 im ersten versteckten Layer und 300 in den folgenden Layern. Der PI-Regler als Benchmark wird vom Agenten in Bezug auf die Optimierungsziele mit einer Gesamtkosteneinsparung in einer Testwoche, die am ersten August startet, von 11,49 $ (30,21 %) übertroffen. Ein "perfect Information Modell" als MPC optimiert den Raum über den gesamten Zeitraum der Testwoche und verringert die Energiekosten im Vergleich zum PI-Regler um 54,02 %, das 20,55 $ Einsparung in dieser Testwoche entspricht. Im Vergleich zum MPC sind die Betriebskosten mit dem Agenten um 9,06 $ (44,12 %) höher.

Schlagwörter: Regelung, Gebäudetechnik, Elektrochromes Fenster, Maschinelles lernen (ML), Reinforcement Learning (RL)

Acknowledgements

My greatest gratitude goes to my supervisor Christoph Gehbauer, who has given me the opportunity to write my master thesis in Berkeley, California. Already in fall 2019 Christoph supported me completing all necessary documents for the application. Without his support even before my arrival in the U.S.A. I would not have had such a good start into the new topic. The ongoing support throughout the last nine months, especially for programming made the work much easier.
Thanks Christoph!

My UAS supervisor Simon Schneider, has agreed at the end of 2019 very spontaneously and at short notice to support me with my master's thesis. Especially in the last stressful period, the support in the completion of the master's thesis was very important.
Thanks, Simon!

Since the pandemic also shut down Berkeley and the Lawrence Berkeley National Laboratory before I could really start, the first weeks were exhausting. I received great support from Ellen Thomas, who took care of my ergonomics in the home-office and always had good camping tips at hand. The meetings of the Windows & Daylighting Group under the leadership of Christian Kohler were very refreshing with the "Ice-breakers" to escape the home office.
Thanks Ellen, Christian and the whole Windows & Daylighting Group!

A special thanks to my two housemates, Ulrike and Christian in the "Haste Mansion" for their support as professional readers and friends throughout the year. I have been able to learn a lot about chemistry, but that is enough for me now!
Thanks, Ulrike and Christian

I would like to thank my parents for their support during my entire studies in Vienna and my studies before. Without the time I got to take care of my education I would not have achieved all this.
Thanks, mom and dad!

Last of all I want to thank the Marshall Plan Foundation for their scholarship which financially supported the research stay.

Danksagung

Den größten Dank verdient mein Betreuer Christoph Gehbauer, der mir die Chance ermöglicht hat, meine Arbeit in Berkeley, Kalifornien zu schreiben. Bereits im Herbst 2019 hat mich Christoph dabei unterstützt alle notwendigen Anträge fristgerecht fertig zu stellen. Ohne die Unterstützung bereits vor Beginn meiner Ankunft in den U.S.A wäre mir der Start in das neue Thema nicht so gut gelungen. Auch das Programmieren wurde mir durch die laufende Hilfestellung in den letzten neun Monaten enorm erleichtert.
Danke Christoph!

Mein FH-Betreuer Simon Schneider, hat sich zu Jahresende 2019 sehr spontan und kurzfristig dazu bereit erklärt mich bei meiner Masterarbeit zu unterstützen. Besonders in der letzten stressigen Zeit war die Unterstützung bei der Fertigstellung der Masterarbeit sehr wichtig.
Danke Simon!

Da die Pandemie auch Berkeley und das Lawrence Berkeley National Laboratory lahmgelegt hat, bevor ich richtig starten konnte war besonders der Beginn sehr anstrengend. Besondere Unterstützung habe ich dabei von Ellen Thomas erhalten, die sich um meine Ergonomie am Heimarbeitsplatz gekümmert hat und auch immer gute Campingtipps parat hatte. Die Meetings der Windows & Daylighting Group unter der Leitung von Christian Kohler waren mit den „Ice-breakern" doch sehr erfrischend um dem Home-office zu entkommen.
Danke Ellen, Christian und das gesamte Windows & Daylighting Team!

Meinen beiden Mitbewohnern, Ulrike und Christian in der „Haste Mansion" danke ich für die Unterstützung als professionelle Korrekturleser und Ablenkung während des gesamten Jahres. Ich habe sehr viel über Chemie lernen dürfen, aber das ist jetzt auch genug!
Danke Urike und Christian

Meinen Eltern danke ich für die Unterstützung während meines gesamten Studiums in Wien und meiner gesamten Ausbildung. Ohne die Zeit mich um meine Bildung zu kümmern hätte ich das alles nicht erreicht.
Danke Mama und Papa!

Zum Schluss möchte ich noch der Marshall Plan Foundation für das Stipendium und damit die finanzielle Unterstützung danken.

Table of Contents

1 Introduction

Building envelopes play a crucial role in the energy performance of buildings, imposing an annual 21.3 quadrillion Btu (6,242.41 TWh) primary energy in the U.S. in 2019, which represents 28 % of the total primary energy consumption (U.S. Energy Information Administration 2020). The initiative of the U.S. Department of Energy launched an initiative for Grid-interactive Buildings whose aim is to optimize the interplay of energy efficiency, demand response, behind-the-meter generation and energy storage to enable more demand-side management possibilities. State-of-the-art control systems, such as Proportional Integral Derivative (PID) controls use conventional feedback and are rule-based. Specifically, they are reactionary (cannot consider future operation) and largely univariate (only consider a single variable) and often fail to deliver sustained performance over the time of the installation (Wang and Hong 2020). These controllers cannot consider future climatic conditions like predicted hot outside air temperatures and only react to the outside conditions, which leads to high peak loads for heating and cooling.

Model-predictive controls (MPC) on the other hand can take the future outside conditions into account and have proven the potential to save energy in simulations, as well as in real life buildings. The disadvantage of the MPC is the fact that as the name implicates a detailed model of the building must be programmed. Therefore, the development and calibration are cost intensive as every building is unique. That is the main reason for the limited application of, predictive control in real buildings.

Current investigations at the Lawrence Berkeley National Laboratory (LBNL) in California induce the application of machine learning (ML) in a building life cycle and show that ML is applicable in many stages of this life cycle (Hong et al. 2020). These studies already demonstrate the potential of ML to benefit the performance of the buildings. ML and the field of reinforcement learning (RL) is especially suited for the desired control strategies and can help to eliminate the developing and calibrating of detailed building models as known from MPC.

1.1 Motivation

The rising requirements for energy management, occupant interactions, on-site renewable generation, on-site storage, electric grid interfacing, etc., demand innovative control methods to integrate multiple subsystems. Furthermore, it becomes necessary to address the number of high-performance objectives, such as minimizing the use of energy, energy cost, increasing the demand response capacity, while satisfying the occupant comfort. Therefore, the control methods need to be responsive to real-time and forecasted conditions, consider the interaction of multiple subsystems, require minimal to no set-up and commissioning and have to be adaptable over the life of the installation.

First studies already indicate the high potential for energy savings, the current challenge hereby is the implementation in buildings to enable more electric loads and distributed Energy systems without reinforcement of the power grid.
Here, the latest study of the LBNL focusing on model predictive controllers (MPC) showed that a total energy cost saving of 28% is possible compared to state-of-the-art heuristic controllers (Gehbauer et al. 2020). The complexity of the building model necessary for the development of the controllers must be decreased to enable more buildings to have advanced building controls in order to path the way for renewable energy systems.

1.2 Aim of the Thesis and Scientific Question

The LBNL investigates the potential of MPC in an environment, where the shading system and the heating ventilating air conditioning (HVAC) system is controlled. Herein, the constraints in form of occupancy comfort (e.g. indoor temperature control) and cost savings have to be considered. Therefore, the aim of this thesis is the implementation of an agent that aims to minimize the total energy costs and the peak electricity load, while ensuring the comfort parameters for the occupants.

Within this framework, the following question needs to be answered to improve existing control strategies:

- Which Reinforcement Learning (RL) methodology is best suited for the control of building technology to further reduce total energy costs compared to state-of-the-art controllers and MPC controllers?

1.3 Approach

At the beginning of the work, a fundamental understanding of state-of-the-art ML approaches and of RL in particular, needs to be gained. RL is a powerful deep learning (DL) technique in the field of artificial intelligence (AI). The most renowned successes of DL were achieved in the video game and board game sector. For example, an agent trained with RL defeated the world champion in the game “Go” which was considered to be impossible due to the complexity of the game (DeepMind 2016). Based on the gained knowledge the RL agent shall be developed in open-source based programming language Python which supports modules and packages that make it suitable for ML applications.

The development of the agent will be performed entirely in Python with the ML framework TensorFlow. The RL agent must regulate the heating and cooling of the building, as well as the control of the dynamic façade as a shading device. The costs for electricity will be compared with a MPC system programmed in python with the module pyomo. The comparison will be performed in the context of California, using the electricity tariff for medium office buildings of Pacific Gas and Electricity (E-19) as a time-of-use (TOU) tariff.

2 Methodology

A plethora of research and development has already been conducted in the field of ML. In the following chapter, the relevant methods employed to answer the scientific question posed in this thesis are summarized and explained.

Some of the tools used to develop the agent are prescribed by the LBNL to enable the communication with existing programs and environments. All used tools and programs used are freeware to ensure reproducibility.

2.1 Programming Language – Python 3.8

Python is an open-source programming language which is administrated by the non-profit corporation Python Software Foundation (Python Software Foundation 2020). The language use is widely spread amongst industry due to its flexibility. It is used for web-development, scientific and numeric or software development. Python is an interpreted, object-oriented high-level programming language with a clear syntax.

Figure 1 shows the popularity of programming languages based on raw data based on Google Trends. The numbers show the share of how often a programming tutorial for the corresponding language has been searched in 2020. By a large margin, python is the most popular programming language with a share of more than 30 % of searches on Google.

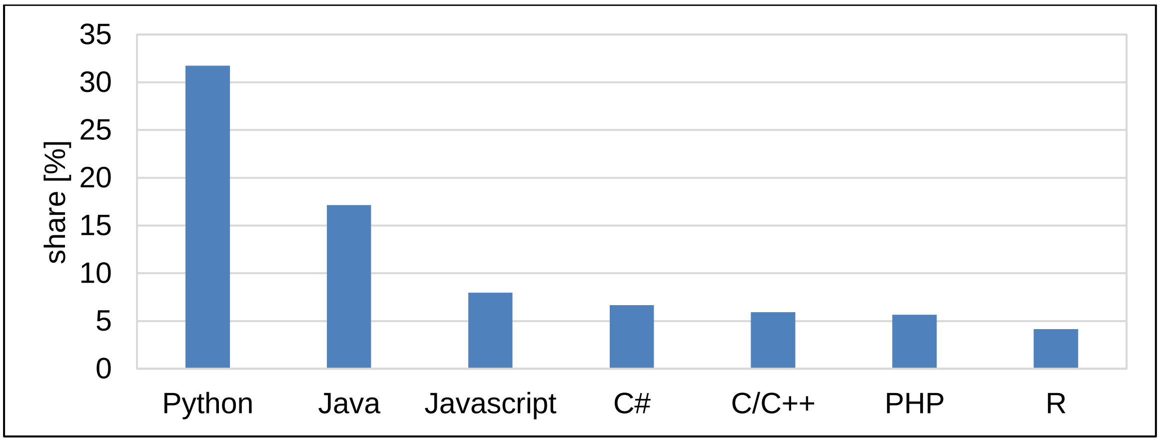

Figure 1: Worldwide PYPL PopularitY of Programming Language in 2020 (modified according to (Pierre 2020))

2.1.1 Machine Learning Framework

The variety of ML frameworks was studied by Jeff Hale who described the popularity of different ML frameworks with a power ranking based on online Job Listings, Google Search Volume, Medium Articles, ArXiv Articles, GitHub Activity and others (Figure 2) (Hale 2019). With applied weights, Tensorflow is the most popular framework for machine learning followed by Keras and Pytorch.

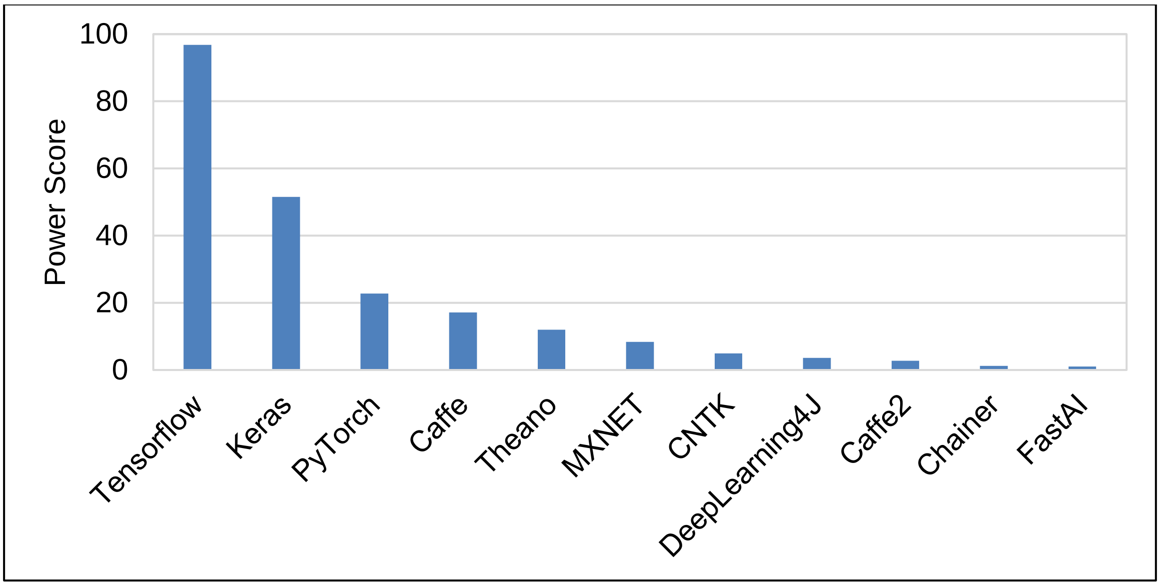

Figure 2: ML Framework Power Scores 2018 (modified according to (Hale 2019))

The further development of deep learning frameworks lead to a new survey by Hale where the growth of the leading frameworks in 2019 was observed as presented in Figure 3 (Hale 2020). The leading frameworks currently are Tensorflow with Keras as the high-level application programming interface (API) and Pytorch with fast.ai. According to these results Tensorflow is the most in demand framework, as well as fastest the growing.

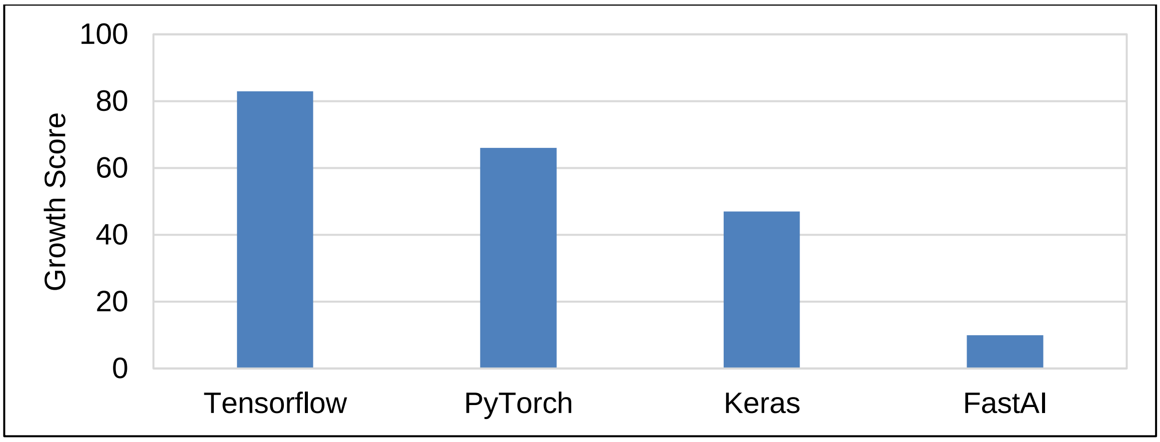

Figure 3: DL Framework Six-Month Growth Scores 2019 (modified according to (Hale 2020))

The open-source library **Tensorflow (version 2.3.0)** was developed by Google and was intended for the spam filter of Gmail before it was available to the public in 2015 (Open Data Science 2019). As shown before, TensorFlow is currently one of the most popular ML frameworks and is widely employed for DL. TensorFlow can run on various platforms, such as Linux, macOS, Windows and on the mobile platforms iOS android or on Raspberry Pi. For performance reasons, the library is written in C++, but the API is also available in Python and others. TensorFlow can be executed on the central processing unit (CPU) or on the graphics processing unit (GPU) with enabled multiprocessing to boost the performance. One of the biggest advantages of TensorFlow is the possibility to work with low-level, as well as with high-level API.

Keras (version 2.4.0) as a high-level API was launched in 2015 and became the framework for developers due a clean API and the possibility to use it with different DL libraries as the backend such as TensorFlow, Theano or CNTK (Google Inc. 2019). In 2019 with TensorFlow 2.0, Keras was integrated and now is the standard interface, when developing DL environments.

The python package **pyomo (version 5.7)** is used for developing the MPC is an open-source package which provides a variety of different optimization models (Sandia National Laboratories 2019). The high-level programming language has the advantage of usability over other algebraic modelling languages.

3 Machine Learning

In 1942, the idea of AI was born in the USA when it was mentioned in the science fiction short story called “Runaround” by Isaac Asimov (Haenlein and Kaplan 2019). At the same time, a machine called “The Bombe” for deciphering Enigma, an encryption device used for secure communications by the German military in the second world war was developed by the English mathematician Alan Turing. The ability to decipher Enigma led to Turing’s seminal paper “Computing Machinery and Intelligence” in 1950 which stated, that, for a machine to be intelligent, it needs to respond in a manner that it is not differentiable from a human being (Turing 1950). These criteria are a benchmark for the intelligence of machines considered to be AI-systems. The first machine that matched this criterion was called ELIZA, it was able to simulate a conversation with a human and was developed between 1964 and 1966 at MIT. The System used for ELIZA was a so-called “Expert System” in which rules are programmed assuming that human intelligence can be formalized with a top-down “if-then” approach. The same system was used in IBM’s Deep Blue in 1997 which was able to beat the reigning chess world champion Gary Kasparov.

A more technical definition for ML was stated by Tom M. Mitchell in 1997: “A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."(Mitchell 1997, p.2). Ethem Alpaydin describes the task of ML as a optimization of a performance criterion using example data and experience (Alpaydin 2010). These definitions are still valid today and based on them different algorithms and approaches have evolved.

The next big milestone for Artificial Intelligence (AI) was made in 2015 by Google with the program "Alpha-Go" which can play the board game Go and was able to beat Lee Sedol the reigning world champion (Haenlein and Kaplan 2019). Figure 4 shows two children playing Go on a board with black and white stones which are placed anywhere on the grid and cannot be moved afterwards (DeepMind 2016). The goal of Go is to capture as much free space and surround as many of the opponent's stones until no more move is possible. This leads to 10^{17} possible board configurations.

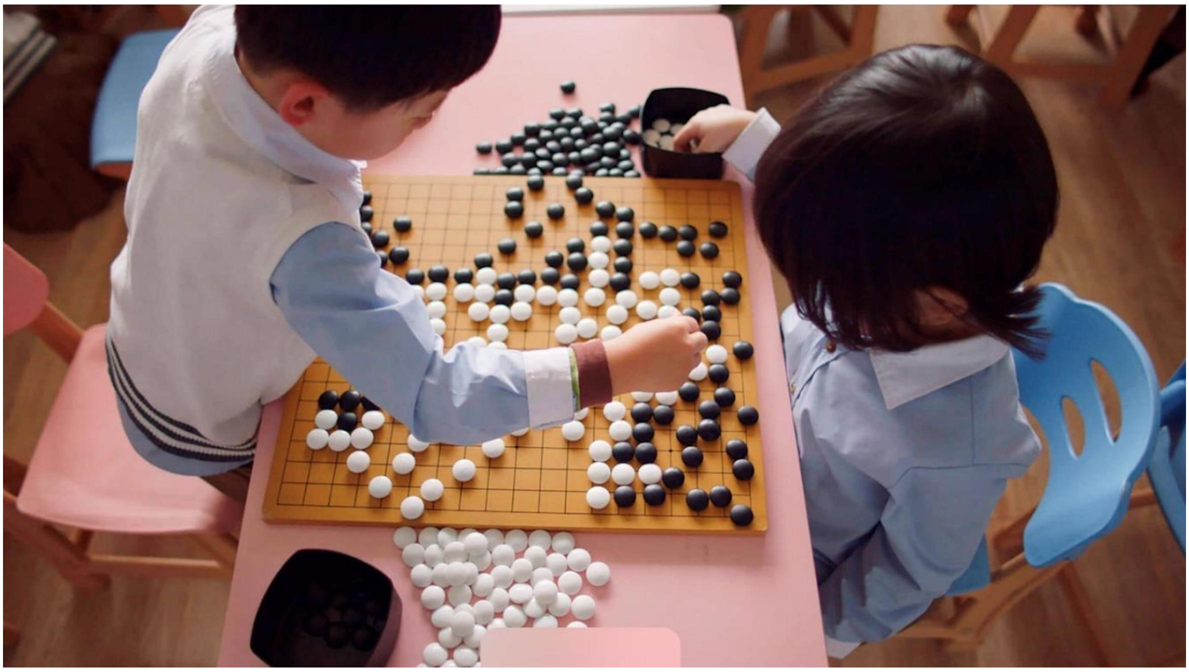

Figure 4: Children playing Go on a regular Go board (DeepMind 2016)

The possibility of 361 first moves in Go makes the game more complex than chess with only 20 possible starting moves. The brute-force of analyzing all possible moves as used in IBM's Deep Blue chess gets infeasible for that number of possible actions with an exponentially increasing cost for calculation. Therefore, the team of DeepMind chose an approach in form of a deep neural network (NN). By playing against amateur players and against itself AlphaGo developed an understanding of how humans play and ultimately outplayed them.

3.1 Applications

The use of AI in industry is strongly driven by information technology companies like Google, Microsoft, Apple and Intel (Pan 2016). Google as an example uses Deep Learning to improve their picture search or develop their unmanned ground vehicle. The research in AI is shifting from academia-related research to research which addresses social demands like intelligent cities, medicine, transportation, logistics, manufacturing, as well as driverless automobiles. AI nowadays supports us constantly in everyday life. Google uses AI to sort the emails into different categories and, most importantly, to filter spam emails. Moreover, it recommends search queries based on the first words typed into the search field and then tries to find the best matches for the question (Bradley 2018). The business-focused social media platform LinkedIn uses AI to find best matches of employees to employers, by observing the behavior of applicants and the outcome of hiring processes. Facebook is helping to prevent suicide and to save lives by detecting suicidal thinking patterns and sending resources to help.

In the specific field of building technologies the research in RL started as early as 1997 and gained more interest since 2015 (Wang and Hong 2020). Wang and Hong found in their study that the main focus in building technologies was Heating-Ventilating-Air Conditioning (HVAC) with a 35 % margin of papers released in this topic in 2015. In 2015 Barrett and Lindner introduced a learning thermostat where the desired room temperature is set by the user and the learning thermostat controls the heating or cooling signal with on or off signals by learning the time schedule of the occupants (Barrett and Linder 2015). In comparison, Wei et. al. introduced a system which controls the air flow of the HVAC system with an agent (Wei et al. 2017). An RL-algorithm for the combination of HVAC control and window control was developed by Chen et. al. in 2018 (Chen et al. 2018). The similarity of these approaches is the cost saving potential in comparison to a heuristic control system.

These examples show that ML can be applied to a variety of integral tasks and different learning techniques are necessary to solve these problems.

3.2 Learning Techniques

The different ML techniques can be classified in the four categories of supervised, unsupervised, semi-supervised and reinforcement learning, depending on the required data (Figure 5) (Mohammed et al. 2017). The designation of the data with classified data refers to whether the data have a specific label, e.g. the picture of the dog has the name dog. With unclassified data, where the name of the picture iis not referred to the content.

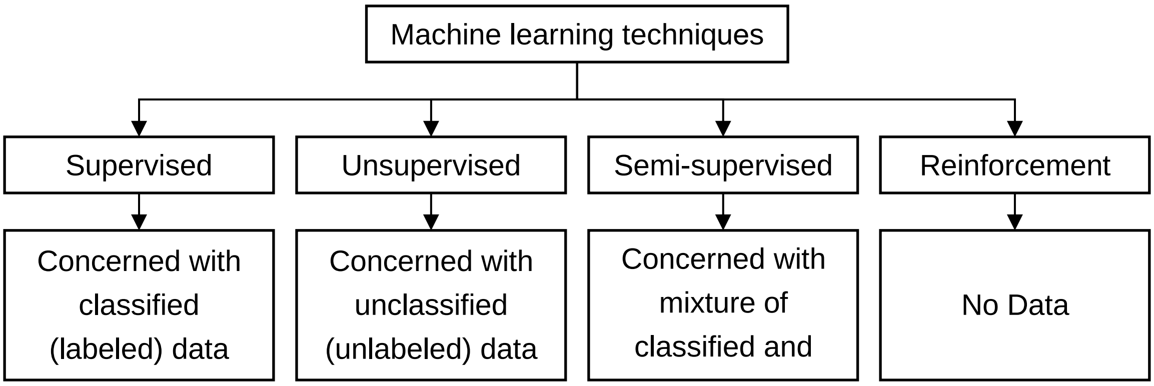

Figure 5: Different machine learning techniques and their required data (modified according to (Mohammed et al. 2017)

Supervised learning

The goal of supervised learning in its basic form is to find a correlation between input data and output data (Brownlee 2019). The two main types of supervised learning are classification and regression. A classification problem could be e.g. a dataset of handwritten digits with pixel data for which the learner should recognize the digits representing numbers from 0 to 9. The regression problem deals with numerical numbers as output, for example house prices could be calculated by given variables that describe the house itself and the neighborhood.

Unsupervised learning

Problems are solved without labelled input data as a reference for learning. In contrast to supervised learning, the model tries to describe or extract relationships in the data. The two main problems it is being used for, are clustering and density estimation which are performed to find patterns in data. Another method where unsupervised learning is used is visualization for graphing or data plotting, as well as projection for reducing the complexity of multidimensional data.

Semi-supervised learning

This technique is a hybrid of supervised- and unsupervised learning where the training dataset contains more unlabeled then labeled data. This method is common for real-world supervised learning problems as in computer vision, natural language processing and automatic speech recognition, due to the lack of training data.

Reinforcement learning

The reinforcement learning technique does not have a dataset available at the start of the training process. In this case, an agent operates in an environment and learns how to operate using feedback and stores the experience. Google's AlphaGo is an example for the most recognized example of reinforcement learning problem. Reinforcement leaning is the technique used in this work and will be discussed in greater detail in the next chapter.

3.3 Reinforcement Learning

The goal of this thesis to control the temperature and illuminance in a room for minimal cost. For the problem at hand, no initial dataset exists prior to the training, making this an obvious candidate for RL.

RL is based on the process by which humans naturally learn (Sutton and Barto 2018). Gaining experience by interacting with our environment is one of the major sources for our knowledge. Figure 6 shows the basic agent-environment setup for RL. The agent operating with the environment selects the actions a_t to take in the current state s_t to reach the next state s_{t+1} and get the reward r_{t+1} as a feedback.

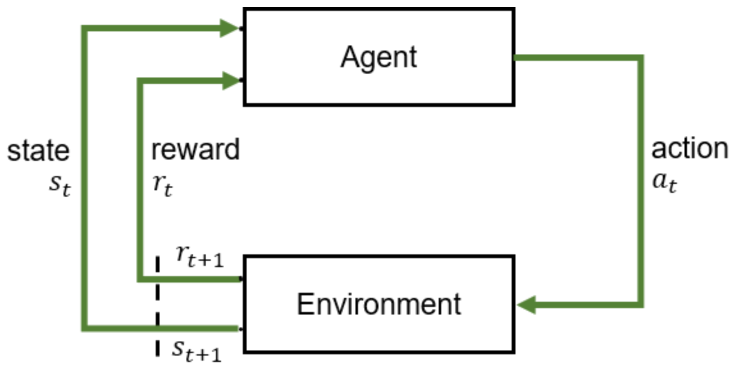

Figure 6: The agent–environment interaction in a Markov decision process. (modified according to (Sutton and Barto 2018, p.48))

The main elements of the RL-system are the policy, the reward function and the value function which are built into the agent and the Environment.

Environment

The environment can be a variety of problems, such as a car or boardgames like chess. In this thesis, the given environment is a thermal room model. The room temperature is the state and output of the environment which should be ensured by the agent. The possible actions for the agent are the energy input by heating or cooling, as well as the control of the shading system. The room reacts to actions taken by the agent and creates an output in form of the next state and the immediate reward.

The reward signal is a single number calculated with a **reward function** in the environment. Its design is crucial for the learning success of the agent, as discussed by Sutton and Barto (Sutton and Barto 2018). The reward in the context of the given room model in this thesis contain the cost of energy and every exceedance of any comfort parameter. How the agent can learn from this reward function is described in chapter 4.1 in greater detail.

Agent

The agent is responsible for selecting actions according to the current state of the environment following a **policy** as the main element of the process. This policy can be a look-up table, a function, or a search policy. Recent algorithms make use of parameterized policy by introducing a NN. The actions can be selected with a stochastic function, with probabilities for each action, or deterministic with the output of the policy being the real value of the action. The optimization goal of the agent in this thesis is to save energy costs while maintaining the needs of the occupants. To achieve an optimal control strategy, the agent tries to maximize a cumulated reward (return) over all viewed timesteps. In its simplest case, this return can be the sum of the rewards (equation 1).

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \quad (1)$$

G_t …… return, cumulated reward
R_{t+i} … reward of timestep
R_T ….. reward of the terminal timestep, last, timestep in the viewed timeperiod

Heating or cooling a building is a continuous task without a terminal state which would lead to an infinite return with the formulation in equation 1. Adding a discount rate γ to future rewards prevents this behavior with equation 2. The initially received reward is worth more than the reward received after the next step.

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \quad (2)$$

G_t … return, cumulated reward
R_{t+i} … reward of timestep
γ …. discounting factor

Value-function

In RL, the two value functions used are called **state-value** and the **action-value** function. The value functions are used to estimate the return, because the rewards for each timestep are not known prior to the state visitation. The value functions are used to train the agent to achieve the optimization goal of maximizing the return.

The **state-value** $v_\pi(s)$ is defined as the total expected reward achievable in the future starting from this state. The state-value noted as $v_\pi(s)$ indicates what the best option for the long run is and takes the next states which are most likely to follow into account. That means that the reward in a specific state can be low, however the value of this state can still be high if the following states can gain a high reward. In the simple maze depicted in Figure 7, the goal is to move from the start in the top left corner to the goal in the bottom right corner. Following the orange line with the highest reward in every single box and summing up the rewards (green numbers), the total return is 120 whereas following the red line by taking the future rewards into account results in the total reward of 155, making it the better option.

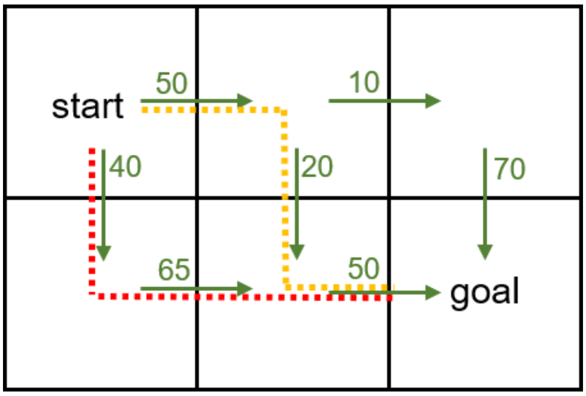

Figure 7: Simple Maze displaying the difference of reward and state-value

This simple example shows that the state-value is crucial for the performance of RL algorithms but is not as straightforward as the reward estimation which is a direct feedback from the environment. The state-value of each state is dependent on the possible actions and the probability these actions are taken following the current policy. Figure 8 shows the state-value and is the visualization of equation 3. Starting from a specific state s the agent can perform any action a. The transition from state s to the next state s' and the immediate reward is expressed as the probability $p(s', r|s, a)$ The reward r is added to the discounted state-value of the next state $v_\pi(s')$. The sum of all possible state-values by taking different actions is then averaged over all possible actions by multiplying the probabilities $\pi(a|s)$ of taking each action a in state s.

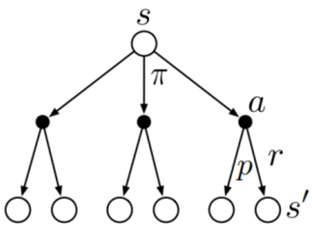

Figure 8: backup diagram for $v_\pi(s)$ (Sutton and Barto 2018, p.59)

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] \quad (3)$$

v_π … state-value
π …. policy
s ….. state
s' …. next state
a ….. action
r …... reward

Like the state-value, the **action-value** $q_\pi(s, a)$ is the estimated return with respect to the state s and the action a. The action value indicates how good it is to take a specific action in a specific state (Figure 9).

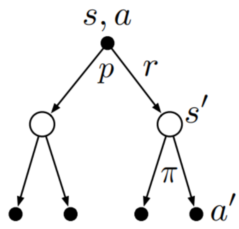

Figure 9: q_π backup diagram (Sutton and Barto 2018, p.61)

For RL two algorithms can be differed. Figure 10 displays an overview of modern RL-Algorithms divisible into model-based- and model-free algorithms (OpenAI 2018). Model-based algorithms do know the model dynamics or learn the dynamics of the environment model with the advantage for planning ahead and seeing what will happen when choosing certain actions. While Google's AlphaZero is model-based with the agent being provided with the ruleset of the game, whereas algorithms like Stochastic Value Gradient learn the model dynamics as part of the learning process. This has the disadvantage that biased models are possibly learnt during the training with the result of sub-optimal performance in the real environment.

Model-free algorithms are separated into the two main approaches of policy optimization and Q-learning. The RL-system either learns policies, action-values (Q-functions) or value functions. As can be seen in Figure 10, the DDPG, TD3 or SAC algorithm are a combination of Policy-Optimization and Q-learning. These policy optimization algorithms are so-called **actor-critic** algorithms and are characterized by using a critic and an actor for training and selecting an action. The state-value of the current step in actor-critic algorithms is calculated with the estimated state-value of the next step and this is added to the actual reward given by the environment. The reward with the estimated state-value of the second step is then called the one-step return $G_{t:t+1}$ which is used to assess the action (Sutton and Barto 2018). The use of the state-value function in this way is called a critic and the function which takes actions is called the actor. Actor-critic algorithms take the one-step return to update and improve the policy.

Other well-known algorithms like PPO or TRPO are pure policy gradient algorithms. Another group of algorithms are Q-learning algorithms, because they learn the Q-value which is in fact the state-value. These algorithms are not feasible for large action spaces because of the discrete actions they use and the necessity to calculate the action-value of all possible actions in the specific state. Policy optimizations can perform in continuous action spaces and are therefore the preferred algorithms for the problem discussed in this work.

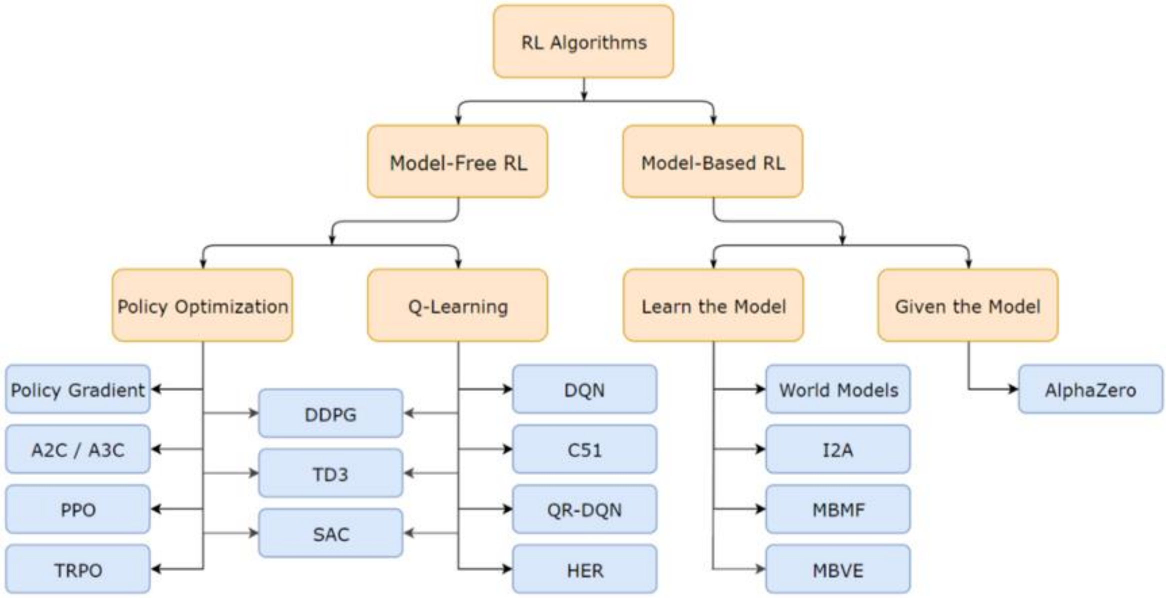

Figure 10: A non-exhaustive, but useful taxonomy of algorithms in modern RL (OpenAI 2018)

The key differences of the Policy-optimization algorithms are described in terms of the action space and policy, the performance measure, and the question if the algorithm is an on- or off-policy algorithm. Off-policy means, that the experience used for training the policy and value functions is not produced by the current policy and can be used multiple times which makes these algorithms more sample efficient. On-policy methods only use the experience from the last episode or steps and compare the new policy with the old one to find out if this episode is better. The performance measure for the comparison is called the advantage of the policy. The key points of the policy optimization algorithms are:

DDPG: Deep Deterministic policy gradient (Lillicrap et al. 2015) (Lillicrap et al. 2019)

- Continuous action spaces with a deterministic policy
- Learns policy and action-value function
- Off-policy

TD3: Twin Delayed Deep Deterministic policy gradient (Fujimoto et al. 2018)

- Continuous action spaces with a deterministic policy
- Learns policy and stabilized action-value function
- Off-policy

SAC: Soft Actor-Critic (Haarnoja et al. 2018)

- Continuous or discrete action spaces with a stochastic policy
- Learns policy and stabilized action-value function
- Off-policy

A2C/A3C: Asynchronous Advantage Critic (Mnih et al. 2016)

- Continuous or discrete action spaces with a stochastic policy

- Advantage function
- On-policy

PPO: Proximal Policy Optimization (Schulman, Wolski, et al. 2017)

- Continuous action spaces with a stochastic policy
- clipped, advantage function
- On-policy

TRPO: Trust Region Policy Optimization (Schulman, Levine, et al. 2017)

- Continuous or discrete action spaces with a stochastic policy
- KL-divergence advantage function
- On-policy

The **policy gradient** algorithms select the actions based on a parameterized policy (e.g. NN) that uses a performance measure (e.g. value function) to update the parameters and improve the performance (Sutton and Barto 2018). The policy is learned based on the gradient of an accumulated reward, as the performance measure with respect to the policy parameters referred to as $J(\theta)$ which can be written as equation 4 with μ as the distribution of the states and π as the policy corresponding to the parameter vector θ.

$$\nabla J(\theta) = \sum_s \mu(s) \sum_a q_\pi(s,a)\, \nabla\pi(a|s,\theta) \tag{4}$$

$\nabla J(\theta)$ … gradient of the performance measure
s ……… state
a ……... action
θ …..….. parameter vector
$\mu(s)$ ..… state distribution
q_π …….. action-value
π …....... policy

The update of policy gradient methods is based on gradient ascent with respect to the current policy parameters θ_t in equation 5.

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)} \tag{5}$$

θ_{t+1} … current parameter vector at timestep t+1
α ……. learning rate (step size of the gradient)
θ_t …… parameter vector at timestep t
∇J …… gradient of performance measure

The policy π selects actions a based on the current state s with the current parameters θ and can be noted as $\pi(a|s,\theta)$.

3.4 Reinforcement Learning in Building Technologies

Wang and Hong have very recently published a review giving a detailed analysis of publications since 1997 in the field of RL in building technologies (Wang and Hong 2020). The algorithms which have been used so far are to 76.6 % based on the number of publications value-based algorithms (Q-learning) which were already excluded for this thesis because of their disability to work in continuous action spaces. Actor-critic algorithms got more popular in recent years with a total share of 15.1 % of all publications. The popularity of actor-critic algorithms is due to the possibility for transfer-learning which means, that a trained behavior from one building can be generalized to other buildings as well. The policy function is suitable for transfer learning because the task of ensuring the room temperature is the same in every building, whereas the mapping from states to actions is not transferable due to different control goals and structures in building technologies.

The methods used to represent the policy and value function shift more and more to NN estimators which were used in all publications in 2019 listed by Wang and Hong. The study concludes that the majority of utilized RL controllers adopted supervisory control which they describe as setpoint control where conventional controllers are still needed to track this setpoint.

Given the analysis by Wang and Hong this thesis will focus on an actor-critic algorithm for developing a RL-controller applicable for transfer learning. Sutton and Barto state, that the advantage of an approximation policy is that it can approach a deterministic policy. Together with the advantages of the policy gradient algorithm the Deep Deterministic Policy Gradient fulfils the approach of a deterministic policy which is described in detail in the section 4.1.

3.5 Neural Networks

The chosen DDPG-algorithm uses NN for the actor to select the actions and the critic to estimate the actor-value. The idea of a NN is based on the functionality of a brain (Ertel 2016). The big step towards an AI with NNs was taken by McCulloch and Pitts in 1943 with the mathematical model of the neuron as a basic switching element for brains. This formulation laid the foundation for the construction of artificial NNs.

The neuron of a brain is comparable with a conductor which get charged by incoming impulses and sends a signal if the voltage exceeds a certain threshold to all connected neurons where the same process is repeated. A neuron can have multiple inputs and outputs and is connected to other neurons (Figure 11).

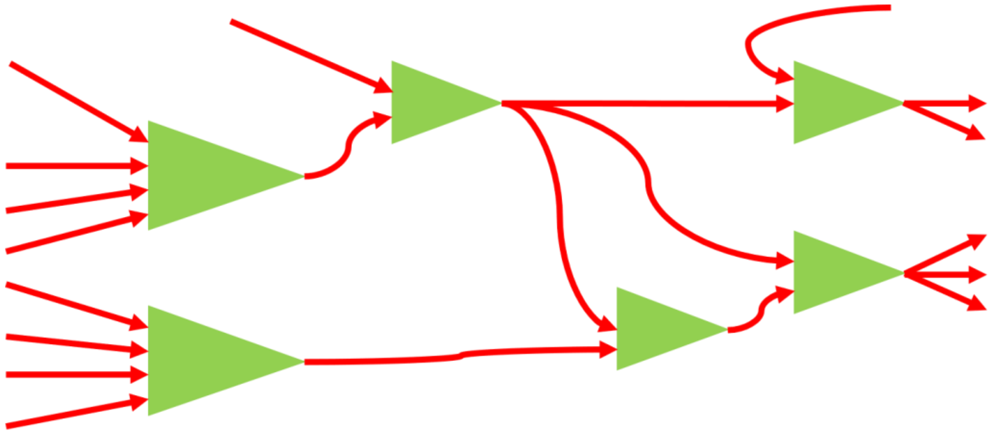

Figure 11: Formal model with neurons and directed connections between them (modified according to (Ertel 2016, p.267))

The mathematical formulation for this process replaces the continuous process of the brain with a discrete time scale and the charging of the activation potential is the sum of the weighted output values with weight ω_{ij} of all input values x_j with an applied activation function f (equation 6 and Figure 12). There are several options for the activation function which are explained in the section 3.5.3.

$$x_i = f\left(\sum_{j=1}^{n} \omega_{ij} x_j\right) \tag{6}$$

x_i ….. output of neuron

f …... activation function

ω_{ij} … weights of the connections

x_j ….. inputs pf neuron

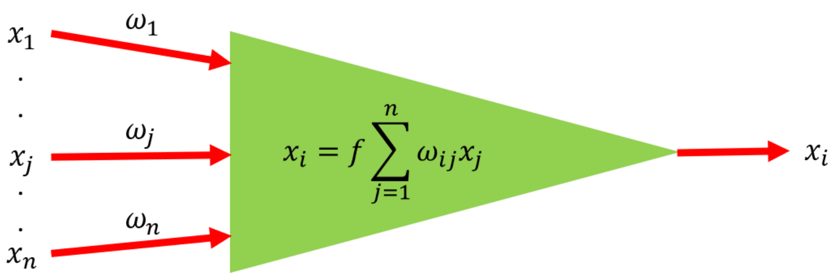

Figure 12: The structure of a formal neuron that applies the activation function f to the weighted sum of all inputs (modified according to (Ertel 2016, p.269))

The most used NN model is the backpropagation algorithm because of its universal applicability for any approximation task. Figure 13 shows a backpropagation network with an input layer, a hidden layer and an output layer. The values x_j^p of the output layer are compared with the values of the targets t_j^p. In the tables to the right in Figure 13 the values of the inputs and outputs and the target values of the NN used in this thesis are shown. The state input is the room temperature and the forecast input is the weather forecast with the outside air temperature, solar radiation, cost of energy and the occupancy of the room. The actor output Q is the heating-or cooling energy input and T_v is the value for the shading system.

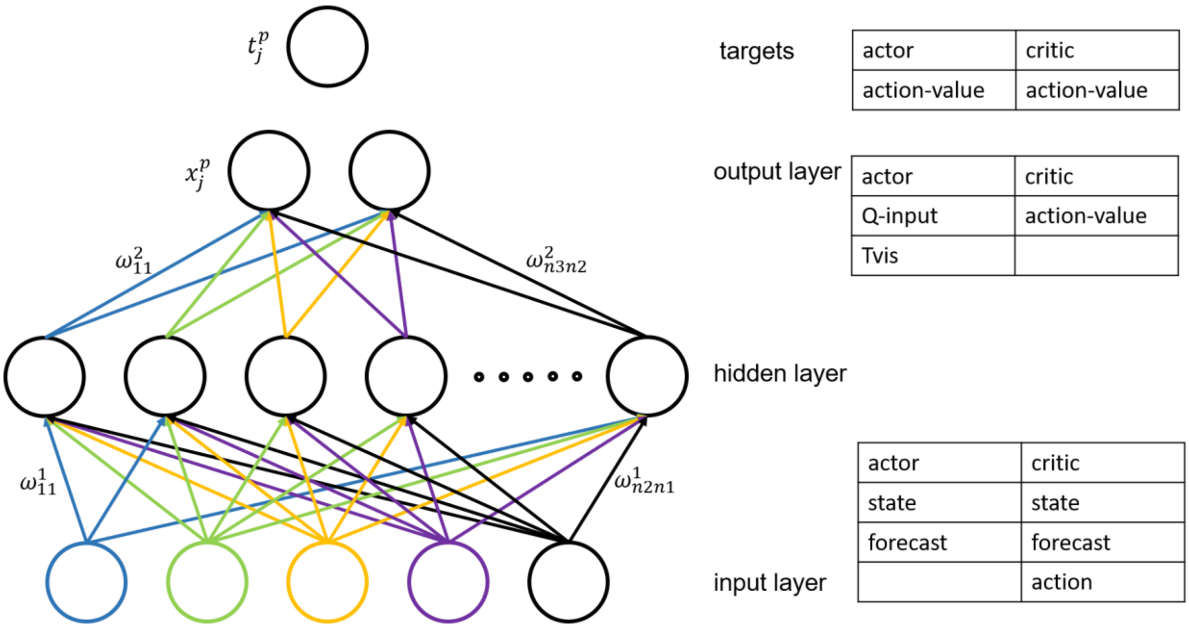

Figure 13: A three-layer backpropagation network with n_1 neurons in the first layer, n_2 neurons in the second and n_3 neurons in the third layer (modified according to (Ertel 2016, p.291))

The target value for the critic network is compared with the value of the output layer and the error is calculated with the preferred function. This error is then used to calculate the negative gradient of the weights and further tune the weights to minimize the error and make accurate estimations of the action-value. The actor network with the actions as an output is not trained to minimize an error and get accurate predictions but trained to minimize the action-value function.

The following section shows two NN architectures based on the backpropagation model for RL which have already proven their usefulness in a wide range of problems. The structure of multi-layer perceptron models and Recurrent NN (RNN) models is described in the following section.

3.5.1 Multi-Layer Perceptron

The multi-layer perceptron network is viewed as the classical NN (Brownlee 2016). The basic structure of this network class is an input layer followed by one or multiple hidden layers and an output layer. Figure 14 shows this structure and displays that the layer size of the layers can vary.

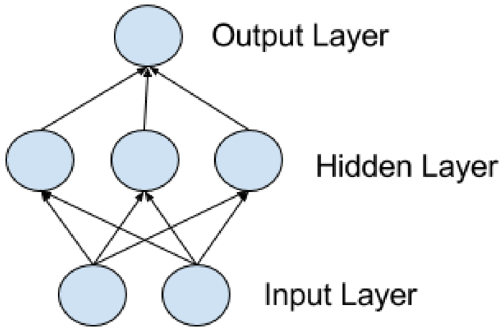

Figure 14: Model of a Simple Network (Brownlee 2016)

The input layer is not constructed with neurons and passes the input to the first hidden layer in the network. The network can have multiple hidden layer which is referred to as Deep Learning. The properties of the output layer as the final layer depends on the problem the NN is used for. The output layer in this thesis has one output neuron for the critic network estimating the action-value function and the actor has two outputs for two actions. The properties and what range of values this neuron can output is depending on the activation function described in 3.5.3.

3.5.2 Recurrent Neural Network

Multi-layer perceptron networks are not able to learn time related dependencies, because they have no knowledge of what happened in the timestep before (Olah 2015). RNN address this issue with loops in the neurons of the RNN layers. A RNN neuron look like the left-hand side of Figure 15 with a loop that allows to use the past information to be used in the current step. The right-hand side shows the unrolled neuron where the output h_0 of timestep zero is passed to the next timestep and is the input together with X_0.

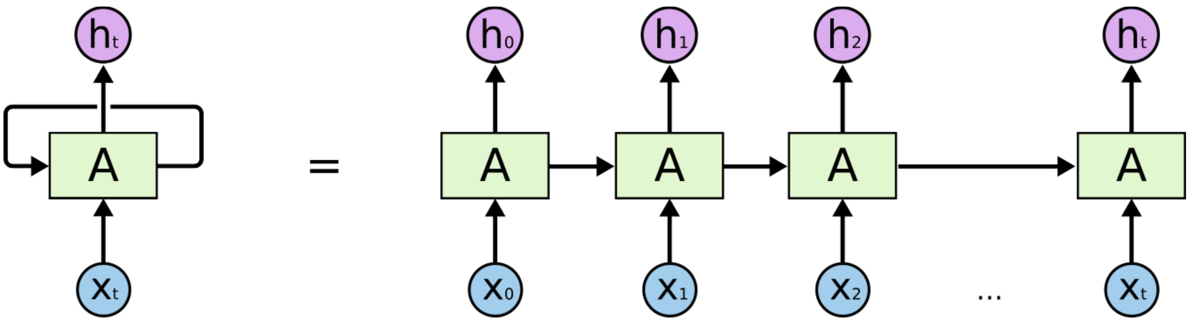

Figure 15: An unrolled recurrent neural network (Olah 2015)

This additional knowledge led to success in speech recognition, language modelling, image captioning or timeseries forecasting. In 2015 Heess et al. used an RNN approach in the DDPG algorithm to conquer problems with partial observable environments like a way sign in a navigation task which is only temporary available (Heess et al. 2015). In the task of room conditioning the interesting value to remember is the past actions and states.

The idea to use an RNN in such a task is to connect previous information (way sign) to the present task (navigation). Unfortunately, basic RNNs have a problem with long term dependencies where not only the information of the last time-step is needed but also the information of a few timesteps back (Olah 2015). Following example by Olah makes this issue clear: I grew up in France …. I speak fluent "?". For a human it is clear, that the missing word is French. The bigger this gap grows it gets more likely for the RNN to fail.

This problem is solved with Long-Short-Term Memory (LSTM) networks which are designed to learn these long-term dependencies. LSTMs were introduced by Hochreiter and Schmidhuber in 1997 (Hochreiter and Schmidhuber 1997). The difference between the RNN and the LSTM is how the information of past timesteps is passed to the next timestep. In RNN the repeating modules responsible for the forward pass of past information is a simple structure with an activation function (Figure 16).

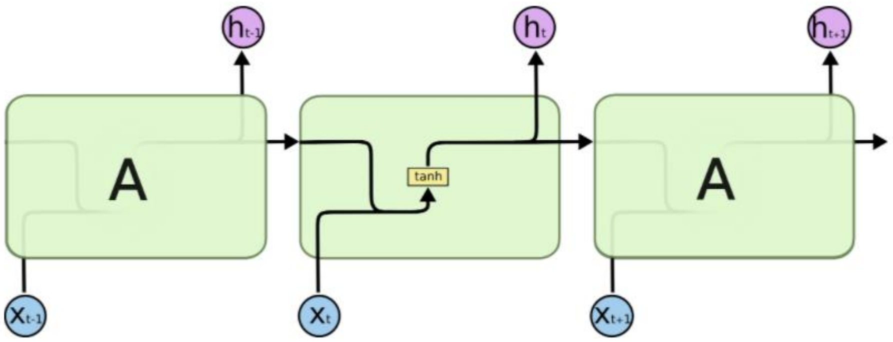

Figure 16: The repeating module in a standard RNN contains a single layer (Olah 2015).

The improved LSTM network layers repeating module is built with four interacting network layers shown in the middle of Figure 17.

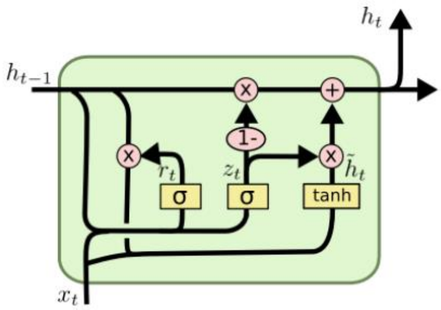

Figure 17: The repeating module in an LSTM contains four interacting layers (Olah 2015).

The architecture of LSTM decides and learns what information to keep of the past information, what information to store as the state of the layer and what information to pass as the output. This is done with the sigmoid (sig) or hyperbolic tangent (tanh) layers called gate layers. The first layer is the forget gate layer which decides what information is thrown away and what to keep followed from the input gate layer which decides which values are updated. Together with the tanh layer the state is updated. The output of the LSTM is a filtered version of the state which is put through a tanh layer to push the values between -1 and 1 multiplied by a sigmoid gate.

3.5.3 Network features

For both presented network architectures the network features like the number of layers, number of neurons of each layer, activation function of the layer and what loss-function should be used to train the network have to be set.

Activation functions

The most common activation functions in NNs are the sig, tanh and variants of rectified linear units (relu) (Ding et al. 2018). Ding et al. analyzed the different activation functions based on their characteristics in NNs.

The **sig function** is the most used activation function because the calculation is easy. The problem with the sigmoid function is that while backpropagating the derivative will reduce to zero around saturation, as shown in Figure 18 and that leads to a vanishing gradient. The gradient vanishes, when more layers with the same activation function are added to a NN (Wang 2019). The weights are not updated effectively which can lead to an inaccurate NN. The output of the sigmoid function is between 0 and 1.

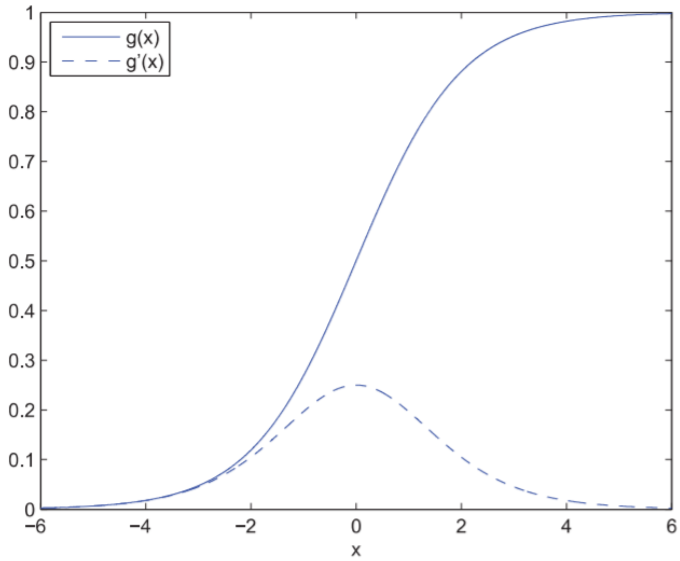

Figure 18: The graphic depiction of Sigmoid function and its derivative (Ding et al. 2018, p.1837).

Similar to the sigmoid function is the **tanh** with output values between -1 and 1 (Figure 19). The symmetric nature of the function makes it more likely to be used than sigmoid because the average of the layer is close to zero and the NN converges faster. The problem with the vanishing gradient also exists with the tanh activation and is more complicated to calculate what makes the computing of the gradient and the update of the weights more time consuming.

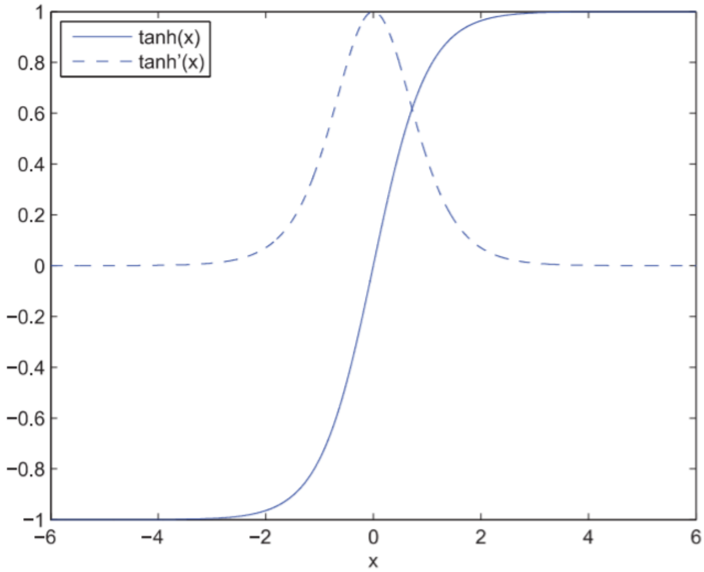

Figure 19: The graphic depiction of hyperbolic tangent function and its derivative (Ding et al. 2018, p.1838)

The **relu** activation and its improvements are currently the most used activation functions in NNs. Values smaller zero which are passed to the activation are always zero and values bigger zero are activated with a linear function (Figure 20). The relu function has advantage of being less computational demanding. With a derivative of 1 the NN converges faster and avoids local optimizations and a vanishing gradient. The disadvantage of relu function is the dying neuron problem. The output of negative values as zero lead to so called dead neurons which will never be activated.

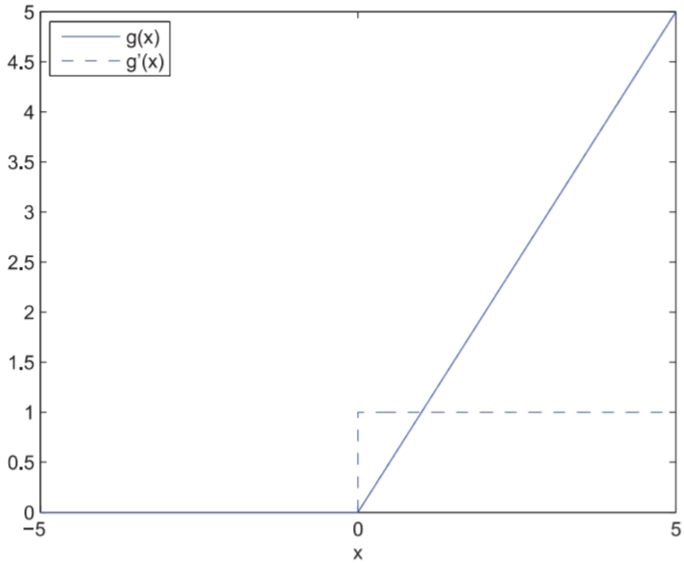

Figure 20: The graphic depiction of ReLU function and its derivatives (Ding et al. 2018, p.1838)

This dying neuron issue can be solved with the **leaky relu (lrelu)** activation function where the negative values of the neuron are not zero and are calculated with a fixed scale for the negative slope, shown in Figure 21. For other activation functions like the **prelu** and the **rrelu**, the negative slope is not fixed but trainable or selected randomly. Ding et. al. tested these activation functions with a classification problem where the NN with the **ReLU** function performed the best.

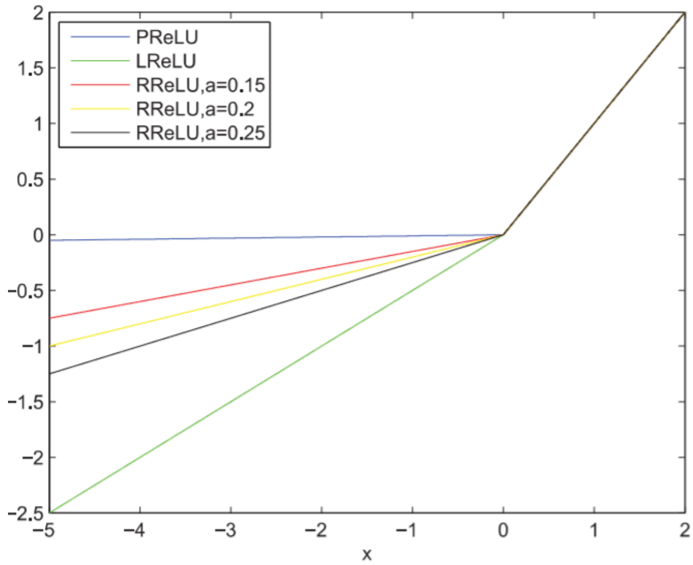

Figure 21: The graphic depiction of LReLU, PReLU and RreLU function (Ding et al. 2018, p.1839)

Loss function

The loss function is the measure of how accurate the model of the NN predicts the target values (Seif 2019). The Mean Squared Error loss used as a default in the DDPG (equation 7) is the right choice when the aim for a NN is to be accurately in the majority of situation.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - q_\pi(s,a))^2 \tag{7}$$

N ……..…. number of samples

y_i ……….. action-value as the target value

$q_\pi(s,a)$ … estimated action-value

4 Results

The basis of the algorithm used to solve the control problem for heating, cooling and controlling of the shading device in an office room is the DDPG, an improvement of the initial Deterministic Policy Gradient by Silver et al which implements a Deep NN, was proposed in 2015 by Lillicrap et al.(Lillicrap et al. 2015) (Lillicrap et al. 2019). Numerous improvements have been made to this RL-algorithm since it was introduced. Improvements considered in this thesis are optimized replay buffer approaches and ways to manage the exploration of the agent using different noise processes.

For a better understanding of the nomenclature in the following chapter and for linking the algorithm to the use case, the state properties and the action space is defined as follows.

state	room temperature
forecast	forecast data for air temperature, solar irradiation, cost of energy and a Boolean variable if the room is occupied or not.
observation	the state and forecast
actions	thermal heating/cooling power and the shading factor
action space	heating/cooling power is bound between -1 and 1 with a scaling factor depending on the room properties shading factor is bound between 0.01 and 0.6.

4.1 Deep Deterministic Policy Gradient (DDPG)

The DDPG is a model-free, off-policy, actor-critic algorithm which can solve problems with high dimensional, continuous action spaces (Lillicrap et al. 2015) (Lillicrap et al. 2019). Lillicrap et al. showed, that DPG is unstable for challenging problems and therefore combined the DPG algorithm with a Deep Q Network algorithm. The advantage of the Deep Q Network algorithm is given by the replay buffer which is replayed in an off-policy way to reduce the correlation between the samples and the use of target networks to reduce the variance of targets while calculating the temporal difference errors for training. The implementation of DDPG follows a straight-forward actor-critic architecture and is therefore easy to implement and to scale to different tasks and network sizes.

The main elements, visualized in Figure 22 of this algorithm are the replay buffer, the environment, the actor network initialized as $\mu(o|\theta^{\mu})$ and the critic network as $Q(o,a|\theta^{Q})$. The weights θ^{Q}, θ^{μ} of both networks are used to initialize the target networks μ', Q' as copies of the actor and critic with the respective weights $\theta^{Q'} \leftarrow \theta^{Q}$, $\theta^{\mu'} \leftarrow \theta^{\mu}$ which are introduced to stabilize training.

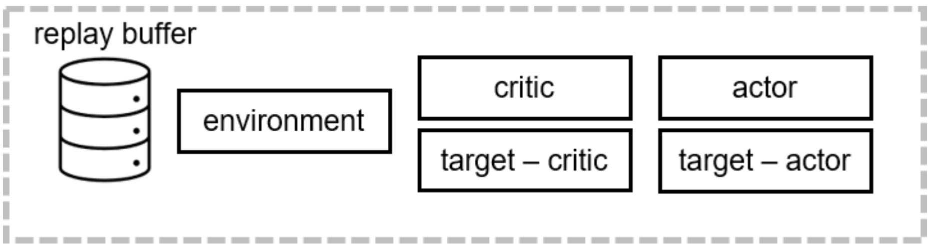

Figure 22: Elements of the DDPG algorithm

The off-policy algorithm explores the action space by selecting an action following the current policy μ in the current observation o_t with an action noise N_t added to the selected action at equation 8. In the DDPG the action noise for exploring the action space can be handled independently of the learning algorithm.

$$a_t = \mu(o_t|\theta^\mu) + N_t \quad (8)$$

a_t … selected actions (Q-input, Tvis)
o_t … observation (state and forecast values)
μ …. deterministic policy (actor)
θ^μ … parameters of the actor
N_t … action noise

The trajectory following the execution of the action is stored with the transition from one state s_t with a forecast f_t and action a_t to the next state s_{t+1} with the next forecast f_{t+1} and the reward r_t for the current timestep as the trajectory$(s_t, f_t, a_t, s_{t+1}, f_{t+1}, r_t)$. Figure 23 shows the process starting from selecting the action until storing the trajectory.

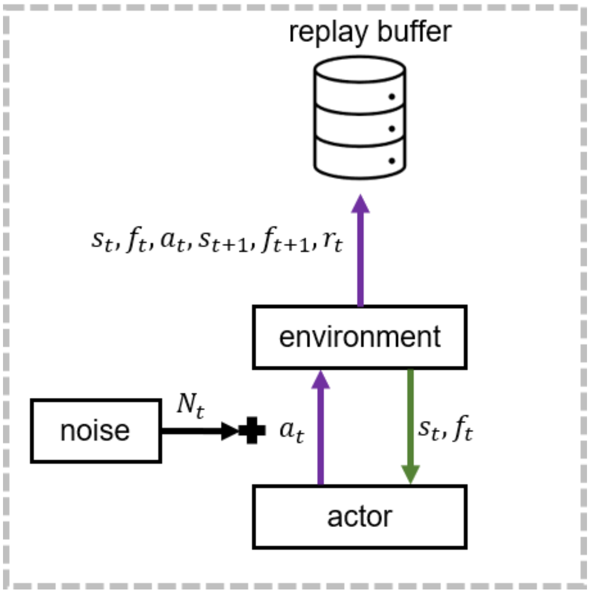

Figure 23: DDPG – agent-environment interaction

The training of the actor-critic networks is executed after each timestep with a minibatch of trajectories, which are sampled randomly from the replay buffer. The training process starts with calculating the action-value with the target networks. With the observation of the timestep t+1 from the sampled minibatch the target actor selects an action and passes it to the target critic to calculate the action-value by adding it to the reward from timestep t. In Figure 24 the green arrows show input data from the replay buffer and the purple arrows are outputs of NNs. Equation 9 depicts the mathematical formulation of the process.

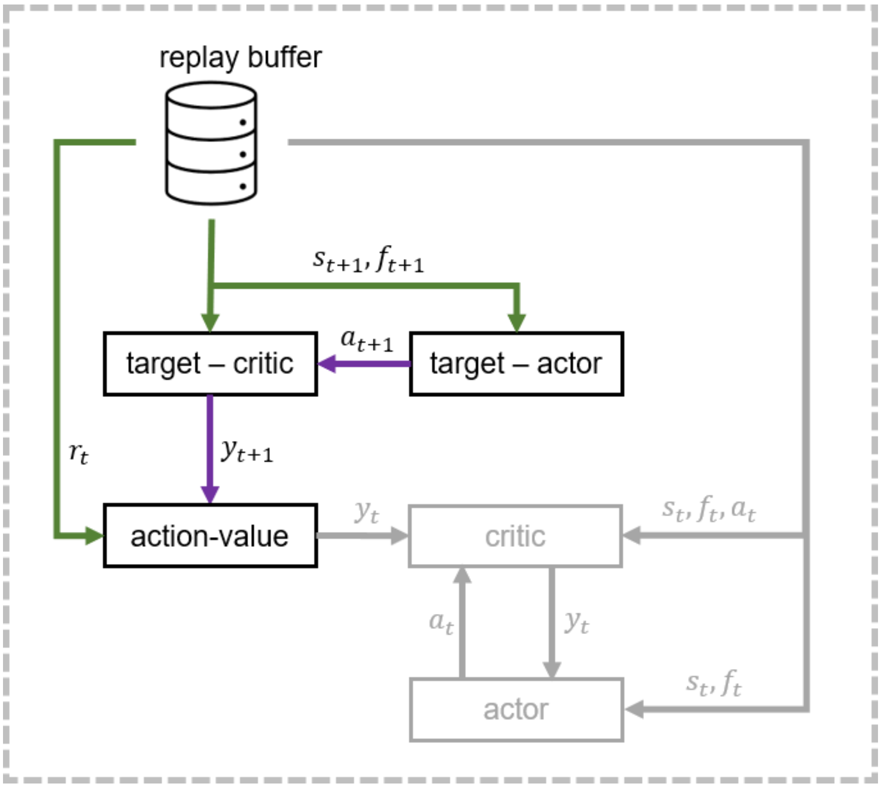

Figure 24: DDPG – calculating the action-values

$$y_t = r_t + \gamma * Q'\left(o_{t+1}, \mu'\left(o_{t+1} | \theta^{\mu'}\right) | \theta^{Q'}\right) \tag{9}$$

y_t …… action-value as target

o_{t+1} … observation of timestep t+1 (state and forecast values)

r_t ….... reward of timestep t

γ ……. discount factor

Q' ….... target critic

$\theta^{Q'}$ …. parameters of target critic

μ' …… target actor

$\theta^{\mu'}$ …. parameters of target actor

The loss L of the critic-network by estimating the action-value is minimized during training of the critic with the mean squared error between the action-value y_t and the approximation of the critic (equation 10).

$$L = \frac{1}{N}\sum_t (y_t - Q(o_t, a_t|\theta^Q))^2 \quad (10)$$

L ….. critic loss

N ….. number of samples

y_t …. action-value as target

o_t …. observation of timestep t (state and forecast values)

a_t …. selected actions (Q-input, Tvis)

Q ….. critic

θ^Q … parameter of critic

The training process presented in Figure 25 illustrates the off-policy training of DDPG. Green arrows are the inputs from the replay buffer, purple arrows are the outputs from the NN and the blue arrows are the values used for backpropagation through the network. The critic is trained with actions selected by an old policy and the action value calculated before. The training of the actor starts with selecting actions with the sampled inputs according to the new policy. The new observation and action inputs are feed into the critic. The objective for optimizing the actor policy is the sampled policy gradient following the updated critic network. The mean value of the estimated actor-value is used to calculate the gradients which are applied to the actor policy. Equation 11 depicts the mathematical formulation of the process.

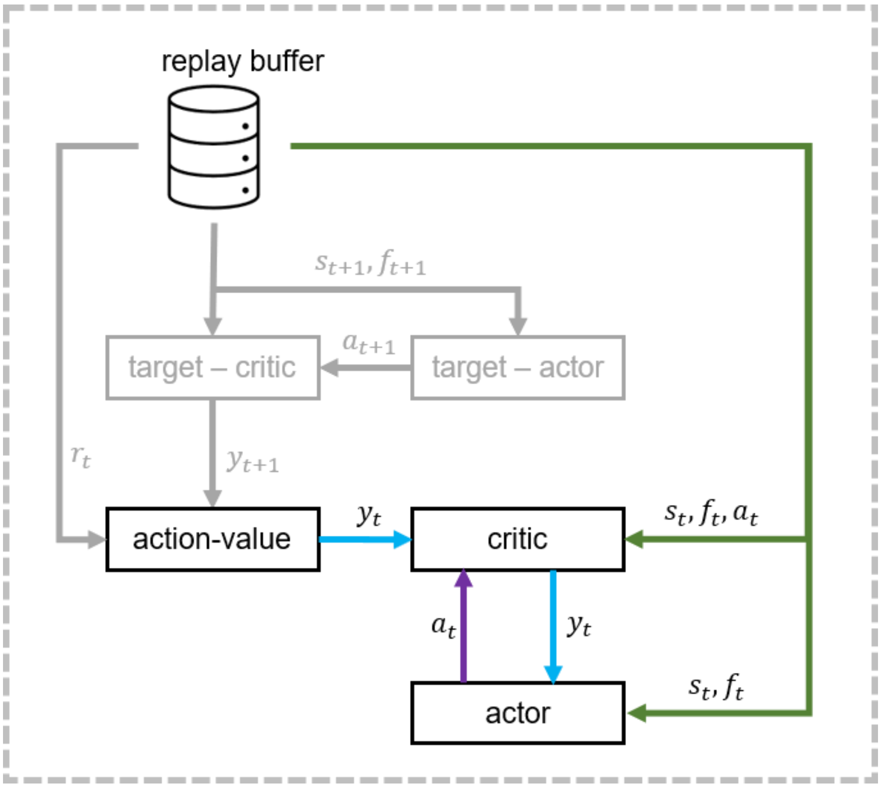

Figure 25: DDPG – training of the critic and actor network

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_t \nabla_\alpha Q(o_t, \theta^Q)|_{o=o_t, a=\mu_{(o_t)}} \nabla_{\theta^\mu}\mu(o_t|\theta^\mu)|_{o_t} \tag{11}$$

$\nabla_{\theta^\mu} J$ … gradient of the performance measure
N ……. Number of samples
o_t …… observation of timestep t (state and forecast values)
Q ……. critic
θ^Q …… parameter of critic
μ …….. deterministic policy (actor)
θ^μ …… parameters of the actor
α …….. learning rate (stepsize of the gradient)

The training of the NN is stabilized with target networks updated with a soft update, which means that the parameter of the actor- and critic network are decreased with the factor τ before copying the parameters to the target networks, calculated with equation 12 for the target critic network and with equation 13 for the target actor network. The disadvantage of this soft update is the slower propagation of the action-value estimation of the critic.

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'} \tag{12}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'} \tag{13}$$

$\theta^{Q'}$ … parameters of the target critic
θ^Q …. parameters of the critic
$\theta^{\mu'}$ … parameters of the target actor
θ^μ …. parameters of the actor
τ …… soft constraint

4.2 Replay Buffer

Experience from the interaction of the agent with the environment is stored in the replay buffer with state, forecast, action, next state, next forecast and the reward. In the original DDPG-algorithm by Lillicrap et al a subset of experiences is randomly sampled from the replay buffer to train the networks (Lillicrap et al. 2019).

Schaul et. al proved that the learning process can be improved by sampling the experiences according to a priority for each experience. **Prioritized Experience Replay (PER)** uses the absolute value of the temporal difference (TD) error (equation 14) of the estimation of the action-value by the critic network (Schaul et al. 2016). The priority of each sample is updated after these experiences are used for training the NN for future training steps. Since new experience do not have a priority and thus, would never be selected for training the priority is set to the clipped maximum priority set by the user.

$$\delta_i = |r_t + \gamma * Q'(o_{t+1}, \mu'(o_{t+1}|\theta^{\mu'})|\theta^{Q'}) - \mathrm{Q}(o_{t-1}, a_{t-1})| \tag{14}$$

δ_i …… TD error
o_{t+1} … observation of timestep t+1 (state and forecast values)
o_{t-1} … observation of timestep t-1 (state and forecast values)
a_{t-1} … actions of timestep t-1
r_t ….... reward of timestep t
γ ……. discount factor
Q' …... target critic
$\theta^{Q'}$ …. parameters of target critic
μ' ..…. target actor
$\theta^{\mu'}$ …. parameters of target actor

In PER, the TD error shrinks slowly, which leads to a frequent replay of experiences with an initial high TD error. This lack of variety in the training data for the NN can lead to over-fitting, meaning that the agent is able to solve the problem in specific states with specific forecasts only. To overcome this issue Schaul et al. introduces a stochastic sampling method, which interpolates between a pure greedy-sampling and random sampling of the experiences with equation 15. The exponent α sets how much prioritization is used with $\alpha = 1$ as the prioritized case with no randomness.

$$P(i) = \frac{\delta_i^\alpha}{\sum_i \delta_i^\alpha} \tag{15}$$

$P(i)$ … priority of sample i
δ_i^α ….. scaled TD error of sample i
α ……. prioritization of randomness

Prioritized sampling introduces a bias in the network because experiences with high priorities are used more often for training. Importance sampling weight (equation 16) is a way to correct the bias. An unbiased sampling is especially important at the end of training, therefore the exponent β sets the amount of correction and increases over time to one. Another benefit of IS weights are the lower magnitudes of the gradients of samples with a high TD error, which enables the use of a higher global step size of the optimizer.

$$\omega_j = \frac{(N * P(i))^{-\beta}}{max_i \omega_i} \tag{16}$$

$P(i)$ … priority of sample
ω_j …... weight of sample
N ...…. batch size
β ….… amount of importance correction

The weight change with IS weights is set according to equation 17.

$$\Delta \leftarrow \Delta + \omega_i * \delta_i * \nabla_\theta Q(o_{i-1}, a_{i-1}) \quad (17)$$

ω_i … importance sampling weight
δ_i … TD error
o_{i-1} … observation of timestep t-1
a_{i-1} … actions of timestep t-1
θ ……. parameter of critic network

Another priority sampling algorithm introduced by Cao et al. in 2019 called the **High-Value Prioritized Experience Replay (HVPER)** builds on PER but combines the action-value and the TD-error for each sample (Cao et al. 2019). The high TD-errors in the first episodes of training do not improve the agent because the optimal policy will not reach these states. The IS weight, as well as the TD error are calculated the same way as in equation 16 and equation 17, respectively. The priority calculation is extended by the variable u_i ,which is updated with $u_i = u_0 * \mu$ every time the experience is used for training.

The priority value for the action-value and the TD-error are often not in the same range. Therefore, these values must be normalized. Cao et al. used the sigmoid function (equation 18) to do so and updates the priorities the action-value and TD error-priority with equation 19 and equation 20.

$$y = \frac{1}{(1 + e^{-x})} \quad (18)$$

$$p_{q_\pi}(i) = sig(q_\pi(o_i, a_i)) \quad (19)$$

$$p_{TD}(i) = sig(|\delta_i|) * 2 - 1 \quad (20)$$

$p_Q(i)$ ….. priority of action-value
$p_{TD}(i)$ … priority of TD-error
q_π …….... action-value
o_i ….....… observation of sample
a_i ……… actions of sample
δ_i …….... TD error

The full calculation of the priority is presented in equation 21. The variable λ shifts the weight of the priorities from the start with a higher weight for the Q-priority until the end with a higher weight of the TD error to speed up the convergence of the NN. The value of u_i declines every time this experience is used, which leads to a smaller priority.

$$p(i) = \lambda * p_{q_\pi}(i) + (1-\lambda) * p_{TD}(i) * u_i \qquad (21)$$

$p(i)$ ……. priority of sample

$p_{q_\pi}(i)$ ….. priority of action-value

$p_{TD}(i)$ …. priority of TD error

λ ……….. Prioritization of action-value/TD-error

u_i ………. scale of priority according to the number of using this sample

The sampling of the experience is a combination of random sampling and priority sampling to reduce the time overhead for updating every priority in the replay buffer with a capacity of up to 10^6 samples. The first step is to randomly select samples with a size of $k * n$ and then select samples via HVPER sampling with a size of n.

These three different approaches for the replay buffer are investigated within the research environment.

4.3 Noise

The noise in RL-algorithms prevents the algorithm to converge to a local optimum and can be applied as an action noise or as a parameter noise (Plappert et al. 2018). In the original DDPG algorithm an Ornstein-Uhlenbeck noise-process is initialized at the start of each episode and added to the selected action (Lillicrap et al. 2019). The Ornstein-Uhlenbeck noise is a temporally correlated noise visualized in Figure 26 by the blue line compared to a Gaussian noise. As discovered by Barth-Maron et al. the correlated noise has no impact on the performance of the algorithm compared to a fixed Gaussian noise. (Barth-Maron et al. 2018).

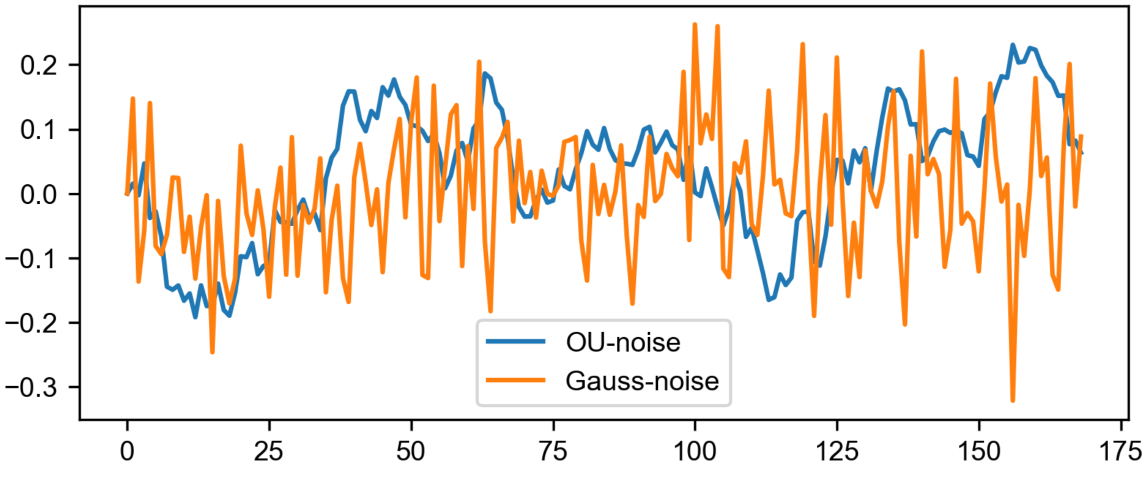

Figure 26: Action-noise process Ornstein-Uhlenbeck and Gaussian

An alternative to action noise is to perturbate the network parameters of the actor (Plappert et al. 2018). Gaussian noise is applied to the parameter vector of the policy network at the beginning of every episode. The action obtained by the policy with action space noise is different with a fixed observation as the input because the noise is independent of the observation. With parameter space noise the obtained action will always be same when passing a fixed observation.

Especially in environments with a sparse reward, means not providing a reward at every timestep, the algorithm with parameter space noise succeeded in the task, whereas the algorithm with action noise failed completely. Scaling the Gaussian noise for the perturbed actor is not as intuitive as scaling the actor noise. Plappert et al. introduced an adaptive noise scaling suitable for all RL-algorithm where the scale over time changes over time with a measure depending on the distance between the actor and the perturbed actor.

4.4 State of the art Controller

PID -controller and MPC can be considered as state of the art controller with MPC (Wang et al. 2017). The simplicity and reliability of PID controllers makes them still widely used, even though MPC has proven to perform better for energy savings and cost savings as Gehbauer et al. demonstrated in their study (Gehbauer et al. 2020).

4.4.1 PID Control

PID controller are a simple form of feedback-controller seen in the control loop displayed in Figure 27 shows the PID controller with the three main elements of P, I and D (Heinrich et al. 2020).

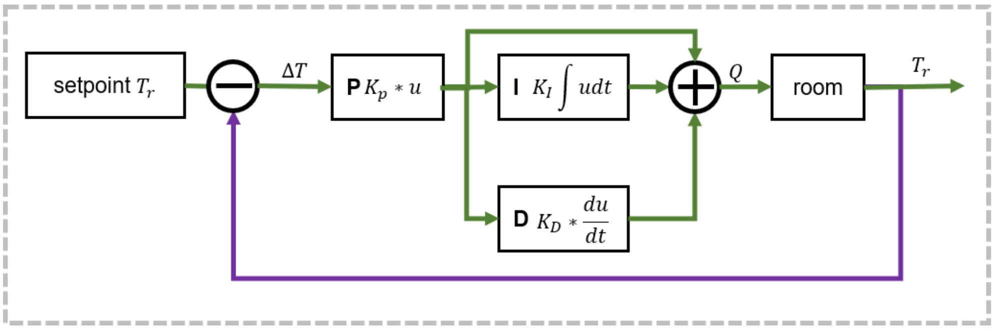

Figure 27: Functional diagram of a PID-controller (modified according to(Heinrich et al. 2020, p.163))

The following icons show the step function after a step for the target value. The step of the target value could be a change of the temperature setpoint, due a time schedule or occupancy sensor. In the control terminology the elements are described with the unit-step response $G(s)$.

Proportional

The P element is a multiplication by a proportional constant with the error between setpoint and the target value (room temperature). This element follows the error without delay (equation 22).

$$G(s) = K_P \tag{22}$$

Integral

With the I element the controller gets more accurate, due to the nature of integration, the control value is not zero if the error is not zero. The target value is reached accurately but the minimization of the error takes longer than with the P element. The unit-step response is given in equation 23.

$$G(s) = \frac{K_I}{s} \tag{23}$$

Derivative

The unit-step response calculated with equation 24 gives the step function of the D element which is an impulse function with a value of zero except at timestep t=0. In combination with a P-element as a PD controller the performance is fast, but the controller is inaccurate, produces high frequent malfunctions.

$$G(s) = K_D * \mathrm{s} \tag{24}$$

The combination of P- and I-element or of P-, I- and D-element is a classic combination for controller as PI-controller or PID-controller. The unit-step response of the PID-controller is specified in equation 25. For a PID controller the equation remains the same, but the derivative constant is set to zero.

$$\begin{gathered} G(s) = K_P\left(1 + \frac{K_I}{K_P}\frac{1}{s} + \frac{K_D}{K_P}s\right) \\ G(s) = K_P\left(1 + \frac{1}{T_I s} + T_D s\right) \end{gathered} \tag{25}$$

$G(s)$ … unit-step function
K_P …... proportional constant
K_I …… integration constant
K_D …... derivative constant
s ……. operator for the derivative by time d/dt
T_I ……. reset time
T_D …… rate time

Setting

The configuration of the PID controller parameters can be done empirically by analyzing the step response and apply the equations 26 of Ziegler and Nichols with the tuning parameters given by the step response in Figure 28.

$$K_P = 0.9 * \frac{T_b}{K_S T_e}$$
$$T_I = 3.3 * T_e \tag{26}$$
$$T_D = 0.5 * T_e$$

T_b …... time constant

T_e …… delay time

K_S …… gain

T_I ……. reset time

T_D …… rate time

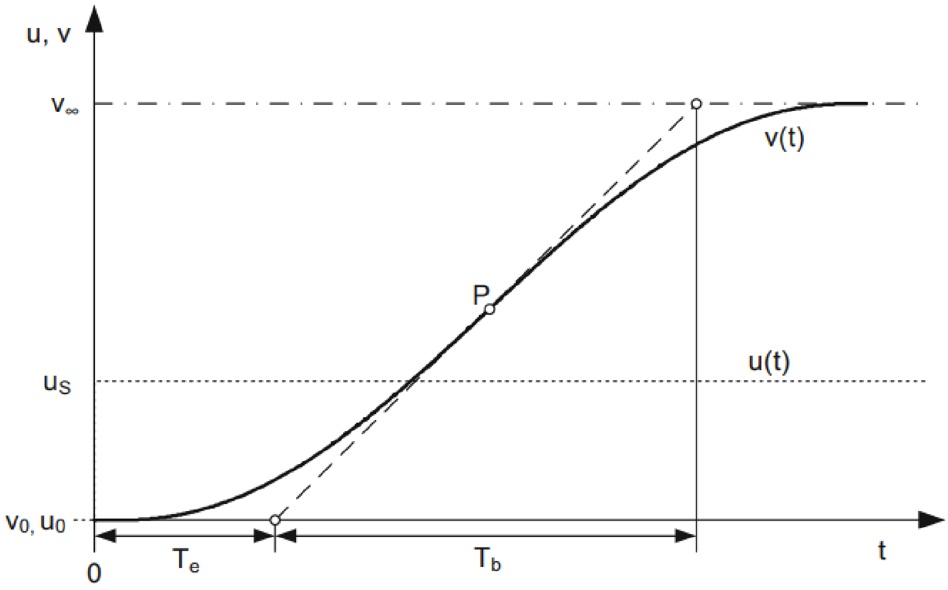

Figure 28: Step response with aperiodic course (Heinrich et al. 2020, p.174)

In this thesis a PI-controller is used to compare it with the developed agent. The parameters of the PI controller in this thesis are:

$K_P = 10{,}000$

$T_I = 30$

$T_D = 0$

4.4.2 Model Predictive Control

In a perfect world, the predictive control model has the knowledge of all relevant information and optimizes its strategy based on this knowledge. The model built in this thesis is a perfect information model and is used to evaluate the agent in the development process. The perfect information model is built with numerical functions and is a twin of the RC-model built in python as the environment of the RL-setup. In Figure 29 the information flow in the model and the constraints and penalties are shown.

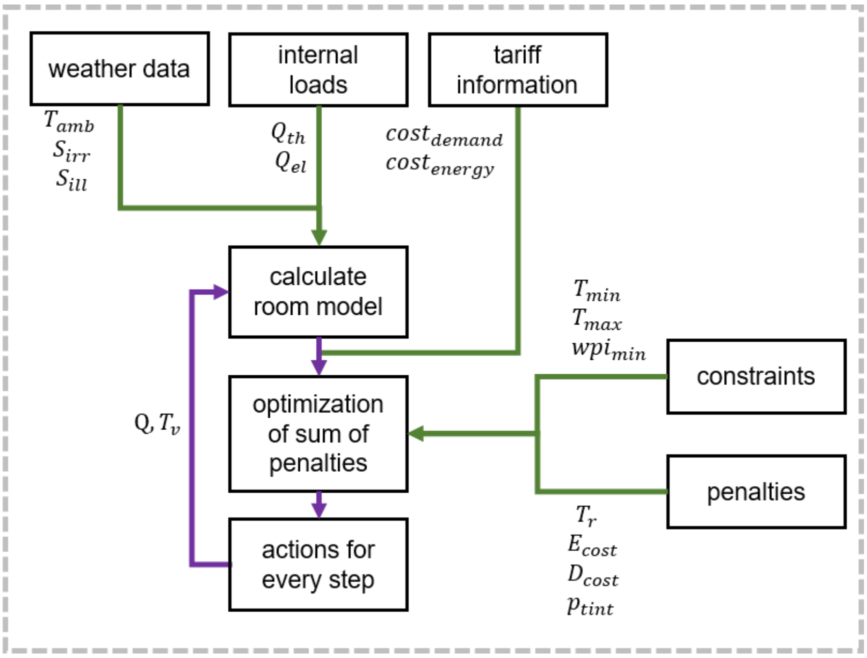

Figure 29: Information flow in the perfect knowledge MPC model

T_{amb} ……….. outside air temperature
S_{irr} …………. solar irradiation on the tilted window
S_{ill} ………..... global horizontal illuminance
Q_{th} …………. thermal internal load
Q_{el} …………. electrical internal load
$cost_{energy}$ …. tariff information energy costs
$cost_{demand}$ ... tariff information demand costs
T_r …………… room temperature
E_{cost} ……….. energy costs
D_{cost} ……….. demand costs
p_{tint} ……….. penalty for tinting the window
T_{min} ……..… minimum room temperature
T_{max} ……..… maximum room temperature
wpi_{min} …….. minimum workplace illuminance
Q …………… energy input
T_v ………….. visibility through EC-window

4.5 Room Model

For this thesis a medium office building, based on a study conducted by the National Renewable Energy Laboratory is the basis of the building properties used for developing the agent (Deru et al. 2011). The reference building has the form parameters of a medium office building which corresponds to a mass or steel construction. These parameters are summarized in Table 1.

Table 1: Reference Building Form Assignments (Deru et al. 2011, p.19)

Floor Area		Aspect Ratio	No. of Floors	Floor-to-Floor Height		Floor-to-ceiling Height		Glazing Fraction
ft^2	m^2			ft	m	ft	m	
53,628	4,982	1.5	3	13	3.96	9	2.74	0.33

The energy relevant specifications of medium office buildings are shown in Table 2.

Table 2: U-Value by Reference Building Vintage - Standard 90.1-2004 (Deru et al. 2011, p.26)

	$Btu/h*ft^2*°F$	W/m^2*K
Roof	0.034	0.1936
Wall	0.580	3.294
Window	1.22	6.927

The single office room controlled in this thesis (Figure 30 in green) has an area of 14 m^2 and a window with a size of 5.2 m^2 which corresponds to a typical window to wall ratio according to a study conducted by the U.S. Department of Energy of 33 % (Deru et al. 2011).

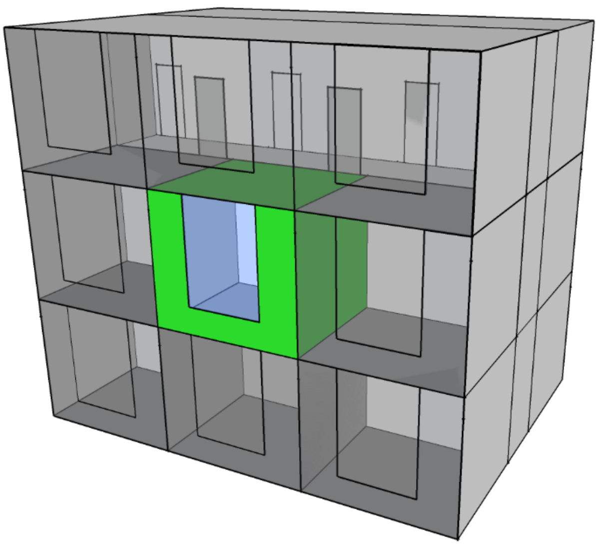

Figure 30: Room model (green)

The resistance value (R-value) is calculated applying the U-values and the respective wall- and window area. The room has no heat loss through ceiling, floor or inside walls. The total capacity of the room is calculated by taking the air properties at 20 °C and by calculating the effective thermal mass of the walls, floor and ceiling following the standard EN-ISO 13786 with the calculation tool developed by HTflux (Rüdisser 2018). The specifications of the room regarding the building envelope are stated in Table 3.

Table 3: specification of the room model

area	14 m^2 (150 ft^2)
height	3.95 m (13.12 ft)
window area	5.2 m^2
exterior wall area	10.6 m^2
U-value wall	3.294 W/m^2K (1.22 $Btu/h{\cdot}ft^2$ ºF)
U-value window	6.923 W/m^2K (1.22 $Btu/h{\cdot}ft^2$ ºF)
R-value room	0.014 K/W
C Room	2205 kJ/K

The HVAC system is modelled with a fixed coefficient of performance with 3.5 for cooling and 1 for heating.

4.5.1 Electrochromic Window

The shading device controlled by the agent is integrated in the glazing of the window as an Electrochromic Window (EC-window). EC-windows are coated with a switchable nanometer-thick ($1x10^{-9}$ m) thin-film which tint can be reversibly changed by applying a small direct current voltage (Lee et al. 2006). The thin film is formed with the following layers:

1. transparent conductor
2. active electrochromic
3. counter-electrode
4. ion-conducting electrolyte

When a bipolar potential is applied to the outside layer (transparent conductor) where lithium ions migrate across the ion-conducting layer from the counter electrode layer to the electrochromic layer. The EC-window is tinted to a Prussian Blue and can be reversed to a clear state by reversing the potential. The window only needs power while changing its tint state and remains unchanged until a voltage is applied. In Figure 31 the principle of an EC-window is shown for the clear and colored state.

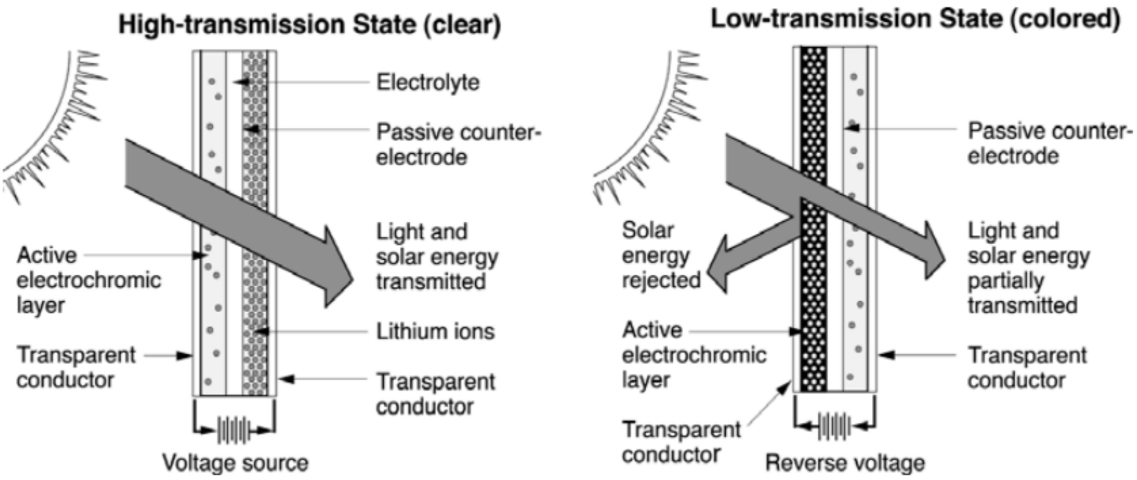

Figure 31: Diagram of a typical tungsten-oxide electrochromic coating (Lee et al. 2006, p.6)

The window can be controlled by changing the visibility transmittance (Tv) in a range of Tv = 0.6 - 0.01. Consequently, the solar heat gain coefficient (SHGC) changes accordingly ranging from SHGC = 0.48 - 0.09. EC-windows are considered to have the potential for real time optimization in buildings regarding the total energy-and demand costs, the stress on the power grid and occupant comfort due to an undistorted view to the sky. In Figure 32 the EC-window is shown installed in an office building in Sacramento. CA (Fernandes et al. 2018).

Figure 32: Each window pane had three sub-zones that could be independently controlled (Fernandes et al. 2018, p.14)

The three independent subpanels of the glass enable a better glare control. The Subpanels can be tinted in four discreet states with the glazing properties for the EC-windows used in this study shown in Table 4.

Table 4: Name and visible transmittance of the four tint levels. (Fernandes et al. 2018, p.15)

<table>
<tr><th rowspan="2">Tint name</th><th rowspan="2">Visible transmittance [%]</th><th rowspan="2">Solar transmittance [%]</th><th rowspan="2">SHGC [-]</th><th colspan="2">U-value</th></tr>
<tr><th>[W/m²K]</th><th>[BTU/ft²F]</th></tr>
<tr><td>Clear</td><td>60</td><td>33</td><td>0.42</td><td rowspan="4">1.816</td><td rowspan="4">0.32</td></tr>
<tr><td>Light tint</td><td>18</td><td>7</td><td>0.16</td></tr>
<tr><td>Medium tint</td><td>6</td><td>2</td><td>0.12</td></tr>
<tr><td>Full tint</td><td>1</td><td>0.4</td><td>0.1</td></tr>
</table>

The dependency of Tv to SHGC is shown in Figure 33 as a linear and a quadratic function. The SHGC is calculated after taking the action Tv to calculate the solar heat gain. The linear function is chosen to calculate SHGC because the quadratic function would slow the simulation down and has no further advantage over the linear function. The action taken by the agent is continuous and can be any number between 0.6 and 0.01.

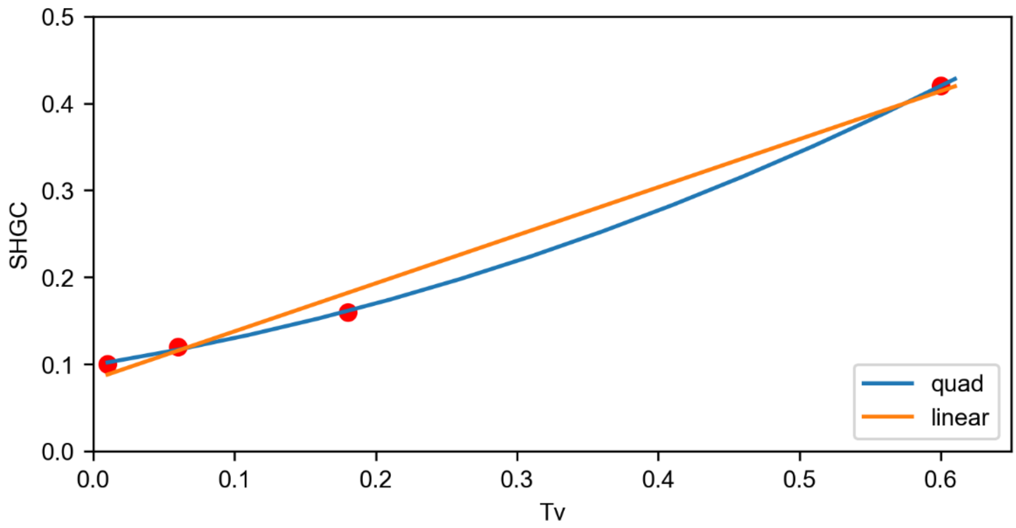

Figure 33: EC-window properties

4.5.2 Solar Position and Radiation

The determination of the solar position and thus the calculation of the incident radiation on the window is necessary for the calculation of the room model. The global horizontal irradiance (GHI), the diffuse horizontal irradiance (DHI) and the direct normal irradiance (DNI) together with the geographical position and the time zone are needed as inputs for the calculation. Starting with the calculation of the real location time t_{WOZ} (equation 29) and the hour-angle ω (equation 30) (Duffie and Beckman 2013). Equation 27 describes the time the earth traveled on the orbit so far this year in degrees and is used in the equation of time (equation 28) which describes the variable length of the days in the year.

$$B = \frac{360}{365} * (N - 1) \tag{27}$$

B … travelled distance in degree
N … day of year

$$E = 229{,}2 * (0{,}000075 + 0{,}001868 * cos\, B - 0{,}032077 * sin\, B - 0{,}014615 * cos\, 2B - 0{,}04089 * sin\, 2B) \tag{28}$$

E … equation of time
B … distance in degree of the earth on the earth orbit

The real location time is referenced to the standard meridian of the timezone and the latitude of the location. With E the elliptic orbit of the earth is also included in the equation 29.

$$t_{WOZ} = t_{LZ} - DST + \frac{\phi_{Bz} - \phi}{15} + E * \frac{1h}{60min} \tag{29}$$

t_{WOZ} … real location time
t_{LZ} …... local time
DST …. daylight saving time
ϕ_{Bz} …. standard meridian
ϕ …..… latitude
E ……. equation of time

The hour angle is referenced to the real location time and is negative before noon and positive in the afternoon.

$$\omega = (t_{WOZ} - 12) * 15 \tag{30}$$

ω …. hour angle
t_{LZ} ... real local time

The orbit of the sun and thus also the position of the sun can be described over several angles, some of them are shown in Figure 34 and described further on.

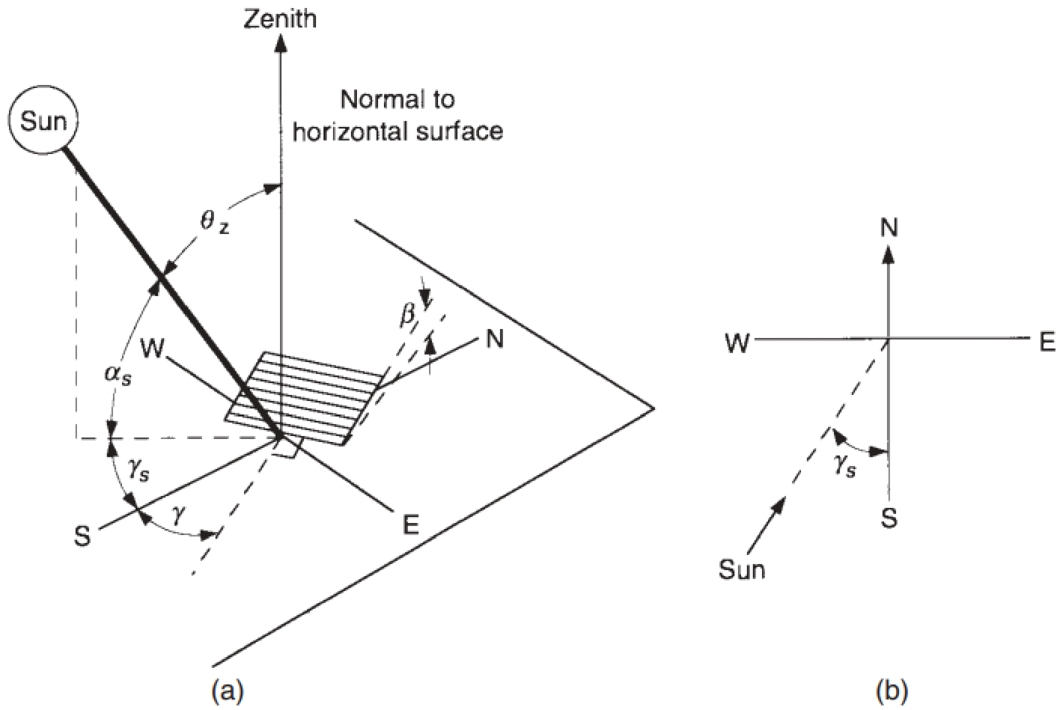

Figure 34: (a) Zenith angle, slope, surface azimuth angle and solar azimuth angle for a tilted surface. (b) Plan view showing solar azimuth angle (Duffie and Beckman 2013, p.13)

Another angle, not shown in Figure 34 is the declination of the earth which varies between -23° and 23° as seen in Figure 35 and can be described with the approximation by Cooper in equation 31 (Duffie and Beckman 2013). The declination is the angle between the sun at solar noon and a plane on the equator.

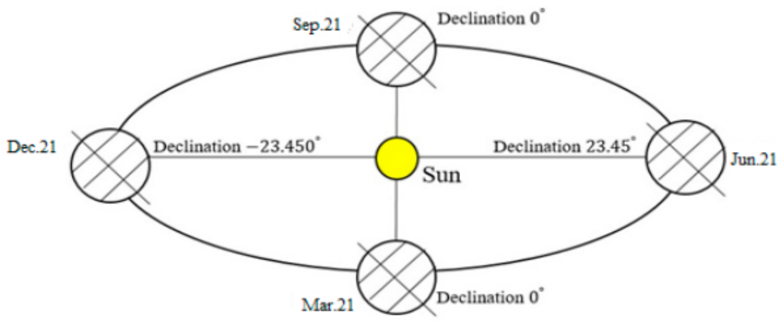

Figure 35: Maximum and minimum value of declination angle (Mousavi Maleki et al. 2017, p.2)

$$\delta = 23{,}45 * sin\left(360 * \frac{284 + N}{365}\right) \qquad (31)$$

δ … declination

N … day of year

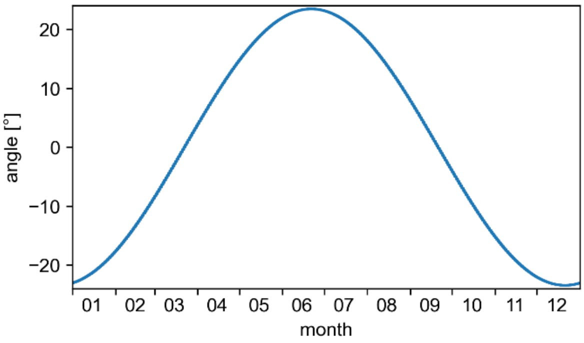

Figure 36: Declination angle in Oakland, CA

The zenith – angle θ_z shown in Figure 34 is a function of the declination δ, latitude φ as well as, the hour angle ω.

$$cos(\theta_z) = cos(\phi) * cos(\delta) * cos\ (\ \omega) + sin(\phi) * sin(\delta) \tag{32}$$

θ_z … zenith angle
ϕ …. latitude
δ …. declination
ω …. hour angle

The azimuth angle γ_s is related to south and varies between -180° and 180° which represents before noon and after noon.

$$\gamma_S = sign(\omega) \left| arccos\left(\frac{\cos(\theta_Z) * \sin(\phi) - \sin\ (\delta)}{\sin(\theta_Z) * \cos\ (\phi)}\right) \right| \tag{33}$$

γ_S … azimuth angle
θ_Z … zenith angle
ϕ …. latitude
δ …. declination
ω …. hour angle

With the calculated angles the angle of incidence θ_{Di} can be calculated according to equation34.

$$cos\ \theta_{Di} = cos(\theta_Z) * cos(\beta) + \ sin(\theta_Z) * sin(\beta) * cos\ (\gamma_s - \gamma) \tag{34}$$

θ_{Di} … angle of incidence
γ_S … azimuth angle
θ_Z … zenith angle
ϕ …. latitude
δ …. declination
ω …. hour angle
γ …. surface azimuth angle
β …. slope of the surface (window 90 °)

The GHI is a product of DHI and the DNI dependent on the zenith angle.

$$GHI = DHI + DNI * cos(\theta_Z) \tag{35}$$

GHI … global horizontal irradiation
DHI … diffuse horizontal irradiation
DNI … direct normal irradiation
θ_Z …... azimuth angle

The product of equation 36 is the DNI on the tilted surface, calculated with the angle of incidence.

$$DNI_T = DNI * \ cos\,\theta_{Di} \tag{36}$$

DNI_T … direct normal irradiation on the surface (window)
DNI ….. direct normal irradiation
θ_{Di} … angle of incidence

The total irradiation on the tilted surface, calculated with equation 37 is the sum of the DNI on the tilted surface, the DHI depending on the angle of the surface in respect to the sky and the GHI depending on the angle of the surface in respect to the ground and the value for ground reflection.

$$I_T = DNI_T + DHI * \left(\frac{1 + cos(\beta)}{2}\right) + \rho_B * GHI * \left(\frac{1 - cos(\beta)}{2}\right) \tag{37}$$

I_T …….. total irradiation on the tilted surface
DNI_T … direct notmal irradiance on the tilted surface
DHI … diffuse horizontal irradiation
GHI … global horizontal irradiation
β …. slope of the surface (window 90 °)
ρ_B ……. reflectance of the ground (albedo)

Figure 37 summarizes the calculated solar angles and displays the total solar irradiation on a window oriented to the south for Oakland, CA with a longitude of -122.22 and latitude 37.72 for a window with an orientation with 0° off south and the slope with 90° of the window. The figure shows the calculated values for January 1st and August 1st with the weather data from 2019.

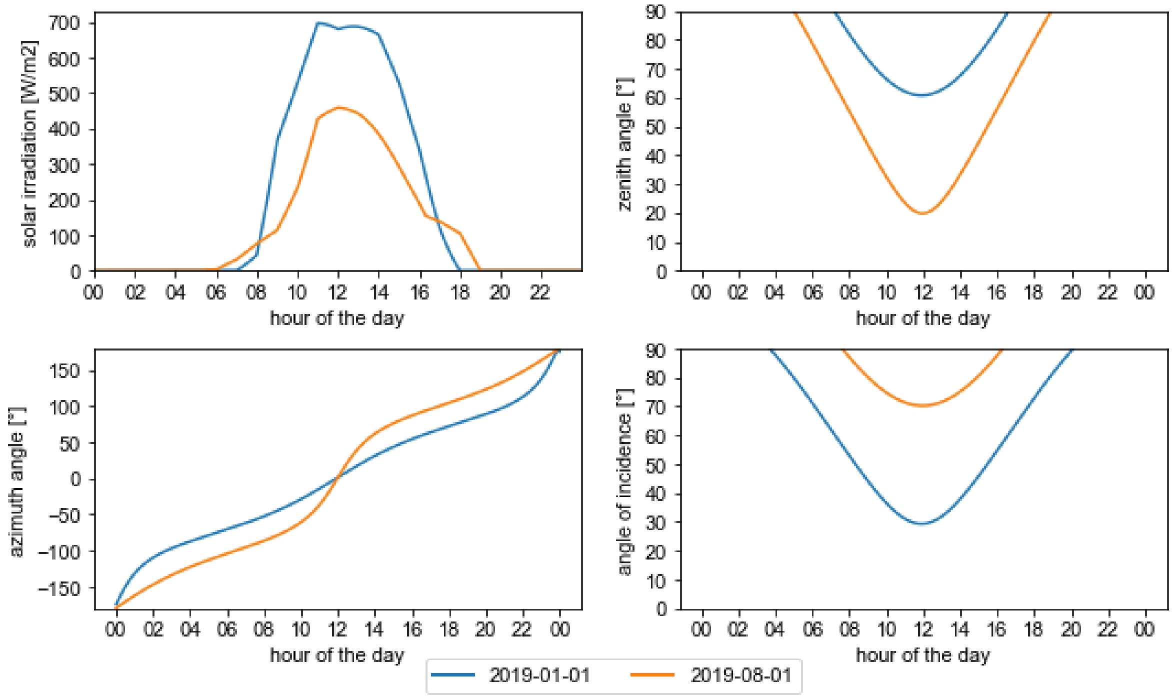

Figure 37: Solar angles and solar irradiation on tilted surface

4.5.3 Electricity Market in California

The master thesis focuses on cost savings for electricity demand. Therefore, the tariff structure of the electricity market in California for economic criteria are analyzed. The focus in this thesis is on the electricity market in Berkeley, Alameda County. This area of California is in the electric utility area of Pacific Gas and Electricity (PG&E) (CEC 2020). The electric power industry is deregulated since 1992 when the U.S. Congress passed the Energy Policy Act and opened the transmission networks to independent energy producers and dissolved the natural monopole of electric utilities (State of California 2018). Due to an energy crisis in 2001 the costumer choice has a limited availability. Customers can enter a lottery system if they intend to choose their energy service provider and opt out of from PG&E as the default energy provider in the city of Berkeley.

For commercial customers PG&E offers two rate options with time-of-use (TOU) or peak day pricing (PG&E 2020b). With the PDP rate plans the customer gets discounted electricity rates in the summer in exchange of higher priced peak periods during peak events from 2-6 p.m., which occur during the summer months on the hottest days of the year. PGE&E proposes the TOU rates with “Maximize your savings with time-of-use rates”. Since the thesis focuses on reducing the electricity bill the electricity rate is chosen from the TOU plans portfolio. The representative electricity rate is the PG&E E-19 tariff with a winter and summer period with different time schedules and energy prices (PG&E 2020a).

Table 5: E-19 definition of time periods, energy- and demand-costs

SUMMER	May 1st	October 31st	Energy cost [$/kWh]	Demand cost [$/kW]
Peak	12:00 p.m. - 06:00 p.m.	workdays	0.16225	19.63
Partial peak	08:30 a.m. - 12:00 p.m. 6:00 p.m. to 09:30 p.m.	weekdays weekdays	0.11734	5.37
Off-peak	09:30 p.m. - 08:30 a.m. 24 hours	weekdays weekends and holidays	0.08846	0.00
WINTER	November 1st	April 30th		
Partial peak	08:30 a.m. - 09:30 p.m.	workdays	0.11127	0.18
Off-peak	09:30 p.m.- 08:30 a.m. 24 hours	weekdays weekends and holidays	0.09559	0.00
Base rate	All year			17.63

The maximum demand is averaged over 15-minute intervals and is calculated and charged monthly. For the demand calculation PG&E uses the maximum demand for each period multiplied with the corresponding costs. The base rate is multiplied with the maximum demand in the month. The bill for the demand costs of one month in the summer could look like Table 6.

Table 6: Example for the demand cost calculation

	Demand [kW]	Demand cost [$]
Peak	0.75	14.7225
Partial peak	1.12	6.0144
Off-peak	0.64	0.00
Base rate	1.12	19.7456
Total demand cost		40.4825

The energy costs are calculated according to the energy consumption every hour corresponding to the TOU-tariff. Figure 38 shows the four different cases occurring in a year.

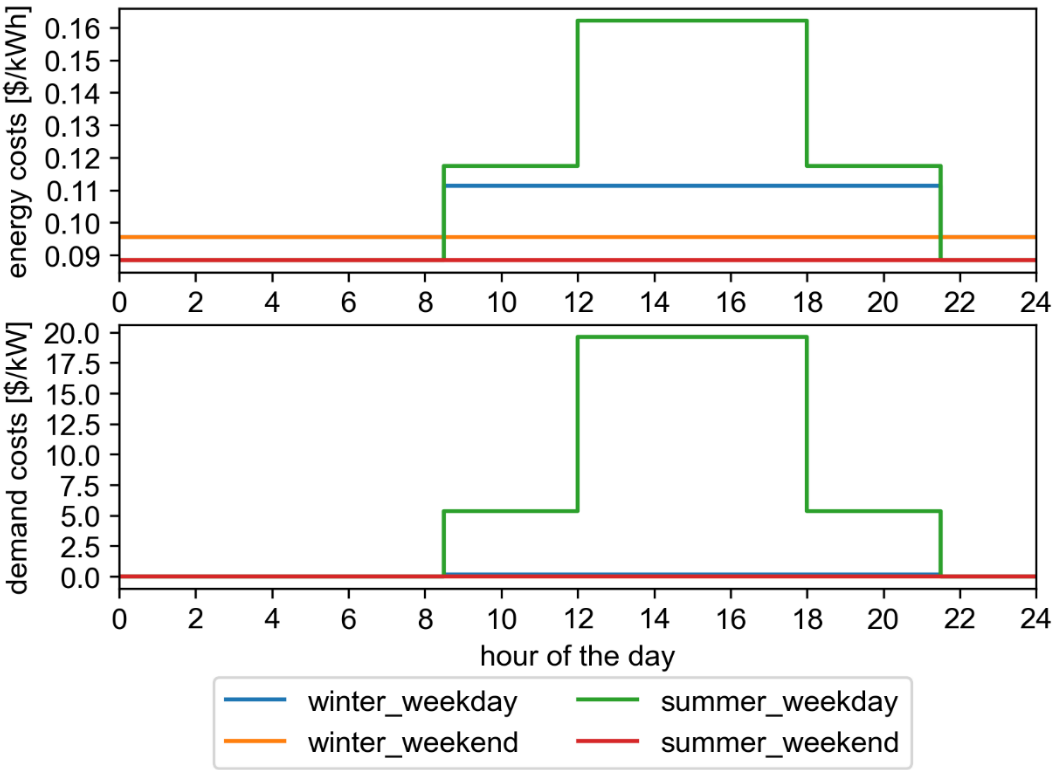

Figure 38: E-19 tariff with time dependent energy- and demand costs (PG&E 2020a)

4.6 RL-Setup

The task of the agent is defined as follows:

Ensure the room temperature within the boundaries of 21 – 24 °C while the room is occupied and 15,5 – 26,5 °C while the room is empty. The workplace illuminance (WPI) should be at least 350 lx for office work. The goal hereby is, to lower the total costs for energy and demand while the constraints are met. The agent can control the shading system by setting the visibility with a linear dependency to the applied current and the heating – and cooling system by controlling the thermal power distributed to the room.

4.6.1 Environment

In the environment in the RL-setup the thermal model and the reward function are defined and calculated. The environment for the development of the agent is a simplified resistance and capacitance (RC) model.

Room- Model

The equation for the RC-model is given with equation 38 and considers the outside air temperature, the room temperature of the previous timestep and the current room temperature. The solar heat gain on the tilted surface I_T is calculated with equation 37 as described in chapter 4.5.2.

$$\mathrm{Q} = \frac{\left(T_{r(t)} - T_{amb}\right)}{R_r} + C_r * \left(T_{r(t)} - T_{r(t-1)}\right) + I_T * SHGC - Q_{int} \tag{38}$$

Q ………. heating- or cooling energy (action of agent)
I_T …...… solar irradiation on the window glazing
$SHGC$ …. solar heat gain coefficient
Q_{int} …… internal loads (people, power consumers, artificial lights)
$T_{r(t)}$ …... current room temperature
$T_{r(t-1)}$ … room temperature of last timestep
T_{amb} ….. outside air temperature
C_r ……... capacitance of the room
R_r ……... thermal resistance of the room

For the second task of the agent to ensure the WPI the illuminance in the room has to be calculated. A detailed calculation of the WPI using raytracing is a computer intensive work. In this thesis the goal is to develop an agent and a raytracing calculation exceeds the scope. The Building Research Establishment on behalf of the Department for Communities and Local Government of the United Kingdom developed an analyzing tool for the energy consumption of buildings (BRE 2015). Building Research Establishment calculates the average daylight factor with total window area and the area of all surfaces in the room (equation 39).

$$\mathrm{DF} = 45 * \frac{A_w * T_v}{0.76 * A_{surf}} \tag{39}$$

DF …… average daylight factor
A_w ……. window area
A_{surf} … area of all room surfaces (ceiling, floor, walls and windows)
T_v …….. visibility (action of the agent)

The daylight factor per definition is the ratio between global horizontal illuminance and the average illuminance in the room. Therefore, by calculating the daylight factor with equation 39 the available illuminance in the room is calculated by multiplying the global horizontal illuminance with the daylight factor.

The internal loads are the sum of artificial light, power consumers and the people in the room. The artificial light ensures the minimum level of WPI, therefore the only signal the agent gets for the tint status is the energy consumption of the artificial light. The power consumers per workplace are assumed to be 10.78 W/m^2 with 10 % of standby energy consumption (Deru et al. 2011). The thermal internal load per workplace, equivalent to one person is 100 W The time schedule for the power consumers and people's presence on weekdays is 07:00 am to 06:00 pm and no occupancy on the weekends is assumed.

Reward

The reward (equation 40) is calculated with the total costs for energy and demand for all energy consumers as the optimization goal. Furthermore, a penalty for exceeding the room temperature boundaries and a penalty for tinting the EC-window with no solar radiation are included in the calculation. The demand is charged monthly, therefore the costs per month are scaled to represent the ratio between one hour of energy costs and the monthly demand costs. The penalty for the room temperature is limited to a maximum value of two to prevent the reward deviate too much from the optimal policy especially at the beginning of the training process. The tint penalty is one when the visibility is set to a lower level than 99 % of the maximum visibility level which means no tinting.

$$r = -\frac{|E_{cost}|}{maxE_{cost}} - \frac{|D_{cost}|}{maxD_{cost}} * scale_D - max\left(\left|T_{r(t)} - T_{const}\right|, 2\right) - p_{tint} \qquad (40)$$

r ……..…… reward

E_{cost} ……… energy costs

$E_{\max\_cost}$ … maximum energy costs

D_{cost} ……… demand costs

$D_{\max\_cost}$ … maximum demand costs

$scale_D$ ……. scale factor for the demand costs

$T_{r(t)}$ …….... current room temperature

T_{const} …..… temperature boundary (min, max)

p_{tint} …….... penalty for tinting the window

4.6.2 Development

The development of the agent includes the selection of the algorithm, the network architecture with its input values and the replay buffer to improve the agent. The action space for the energy input Q is set with a maximum specific heat- and cooling power of 100 W/m^2 and the visibility T_v with the properties shown in 4.5.1 with action boundaries of 0.01 to 0.6. The observation of the agent contains the state of the room and a forecast including the outside air temperature, solar radiation, costs of energy and the occupancy of the room.

The training process of the agent runs within episodes with a length of one day and a total of 3,000 episodes. For each episode, the start day is selected randomly from the weather dataset and a random start state (room temperature) is selected within the temperature boundaries. The agent is trained with the weather data set of Oakland Intl AP 724930, distributed by EnergyPlus (EnergyPlus 2019).

The basic setup of the agent is based on the experimental setup of the DDPG with a multi-layer perceptron network with 2 hidden layers with a layer size of 400 and 300 respectively (Lillicrap et al. 2019). The structure of both NNs is the same, with the difference that the action is added to the critic network after the first hidden layer. The activation function for the hidden layers is the relu function and for the output layer of the critic a linear function is applied. For the actor, the activation function for the output of Q is tanh with values between -1 and 1 and the output for T_v with a relu function with a maximum value of 1 is utilized. The learning rate was chosen to be 10^{-4} for the actor and 10^{-3} regarding the critic to ensure, that the critic converges faster than the actor. The soft update for the target networks is set to 0.001.

Furthermore, the magnitude of the input values is a critical parameter for the NN. This in regards, all input values are normalized between -1 and 1.

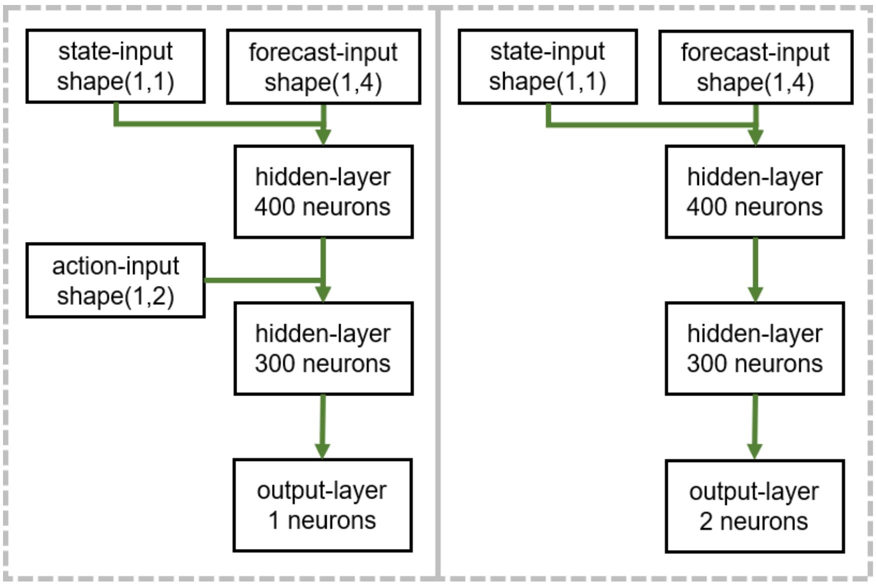

Figure 39: NN architecture; critic left and actor on the right with the shape of the input vectors

The default action noise in the DDPG algorithm is the Ornstein-Uhlenbeck process, which is initialized for both actions with a different scale, due to the varying action spaces with 0.15 for Q and 0.1 for T_v. 64 samples for each training step are selected randomly from the replay buffer.

The following figures represent the agent after the training process for one test week starting from August 1st or January 1st. The figures are structured as follows:

- The weather data is represented in the first graph including:
  - The solar radiation with (GHI, DHI, DNI)
  - The outside air temperature (T-out) on the right y-axis

- The second graph shows the themal power of the HVAC system whether its heating or cooling.
- The third graph is the tint state of the EC-window
- The fourth graph shows the room temperature and the temperature boundaries with the setpoints for the time the room is occupied and not.

In the first training run the agent is limited to one action with a fixed T_v to 0.6 to proof if they can succeed. After the training run the agent with the basic settings of the DDPG algorithm is able to maintain the room temperature most of the time but has problems in the morning hours when the minimal room temperature increases from 15.5 °C to 21 °C (Figure 40).

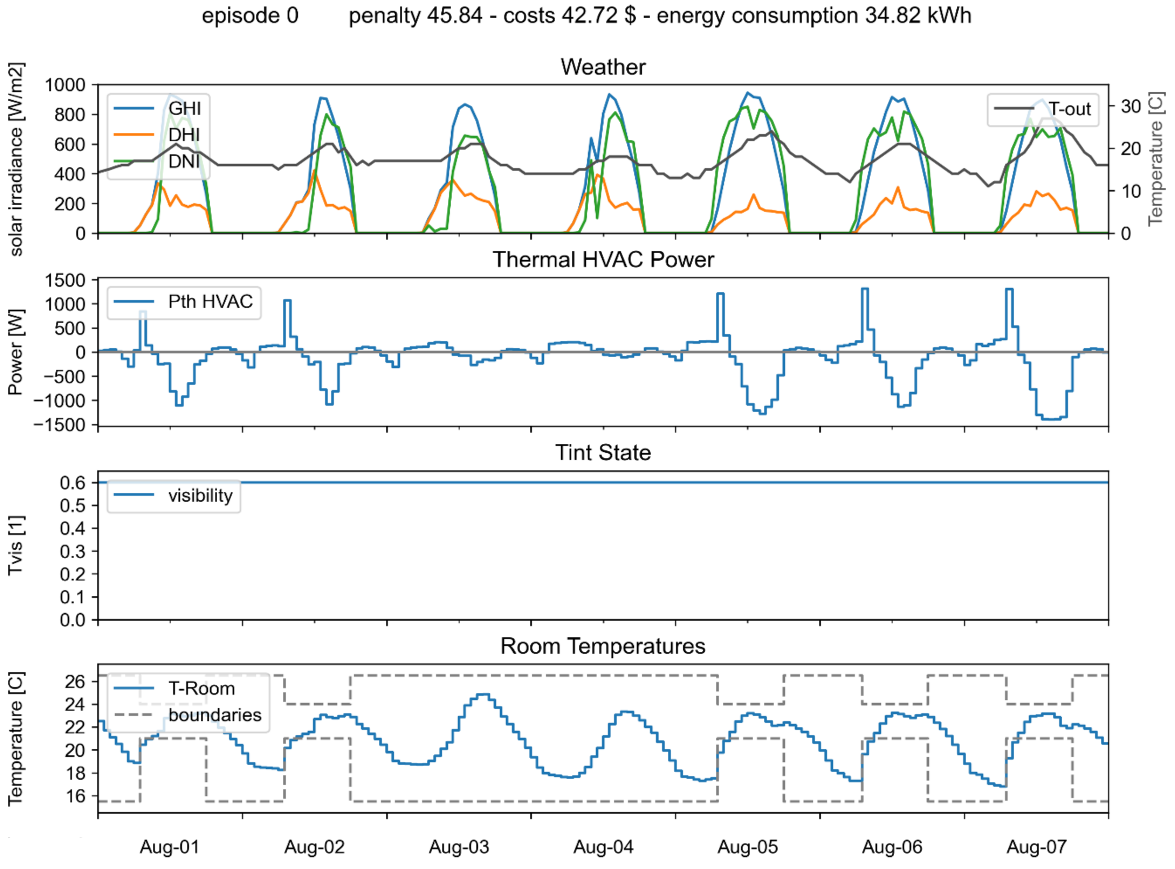

Figure 40: First training result of a week starting on August 1st with HVAC control and a fixed Tint state

Moving on with the development the agent must control both possible actions. With the same setup as before the agent does not succeed in its task (Figure 41). The agent is not eager to heat the building, even though the temperature constraints are not met and only does that on the weekends.

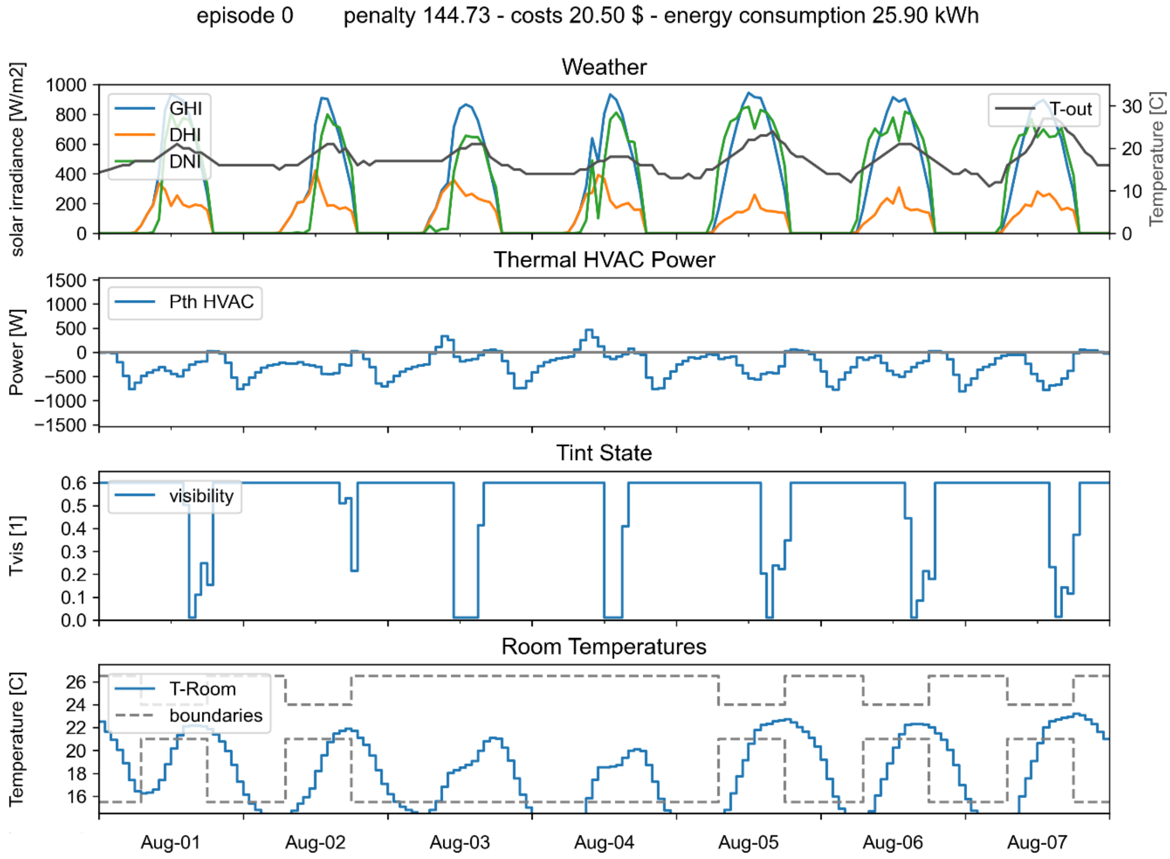

Figure 41: Training result of a week starting on August 1st with HVAC and Tint state control

Investigating the reason, why the agent failed, the loss as an accuracy measurement of the critic was analyzed. Figure 42 indicates that the critic loss increases over time but stabilizes at the end of the run.

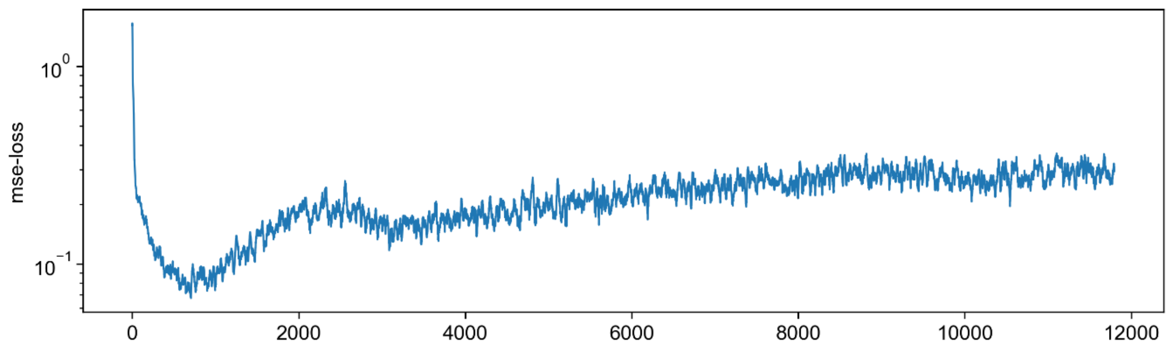

Figure 42: Critic loss of a week starting on August 1st with HVAC and Tint state control

To get a more accurate critic which is leading to a more successful agent, the NN architecture of Fujimoto et. al. proposed in the TD3 algorithm in 2018 is implemented with the critic architecture including the action in the same layer as the state-and forecast input (Figure 43) while the rest of the network remains unchanged (Fujimoto et al. 2018).

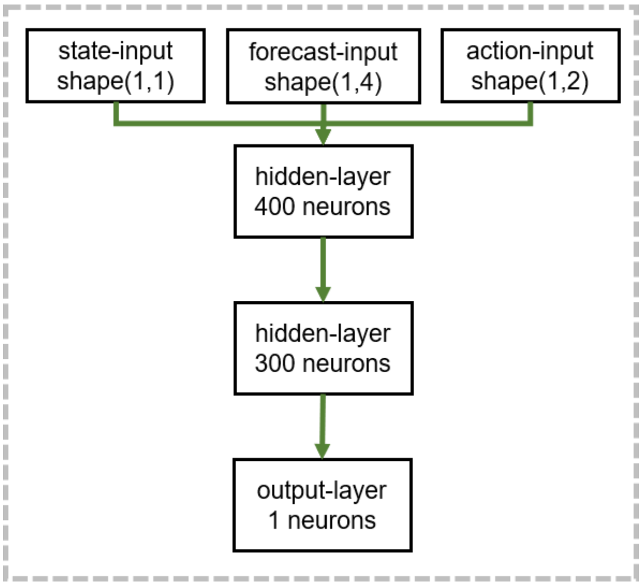

Figure 43: New critic network based on the TD3 algorithm with the shape of the input vectors

After another training run with the new critic network, the critic loss is lower (Figure 44), which indicates that the critic is more accurate and stabilizes after 3,800 training steps.

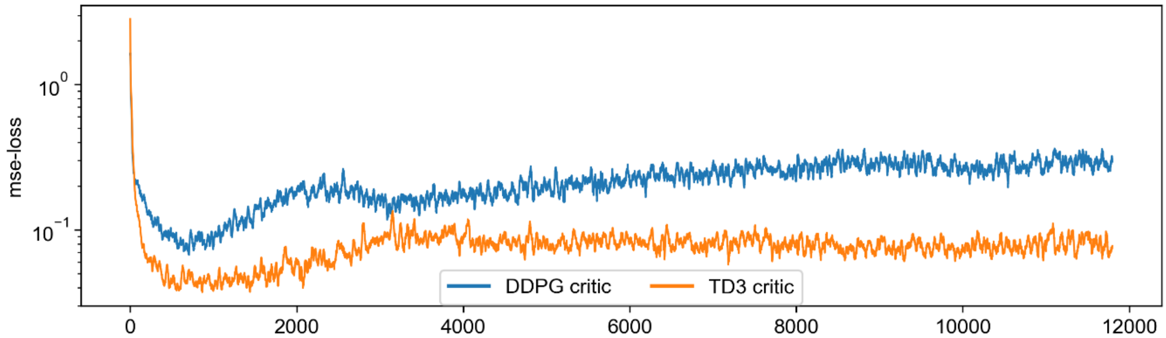

Figure 44: Comparison of critic loss between DDPG critic and TD3 critic architecture

The agent is successful, regarding the temperature constraints by controlling the energy input but fails to control the tint state Figure 45. Following the reward function, the agent should select the brightest tint state during the night to avoid getting penalized for tinting the window. The agent does not find the correct way to handle the reward function. The behavior of the agent, regarding the HVAC system is not energy saving by any means. The agent heats the room starting in the night until the room temperature gets close to the upper boundary and then starts cooling the building Figure 45. The positive thing out of this analysis is, that the agent recognizes the constraints.

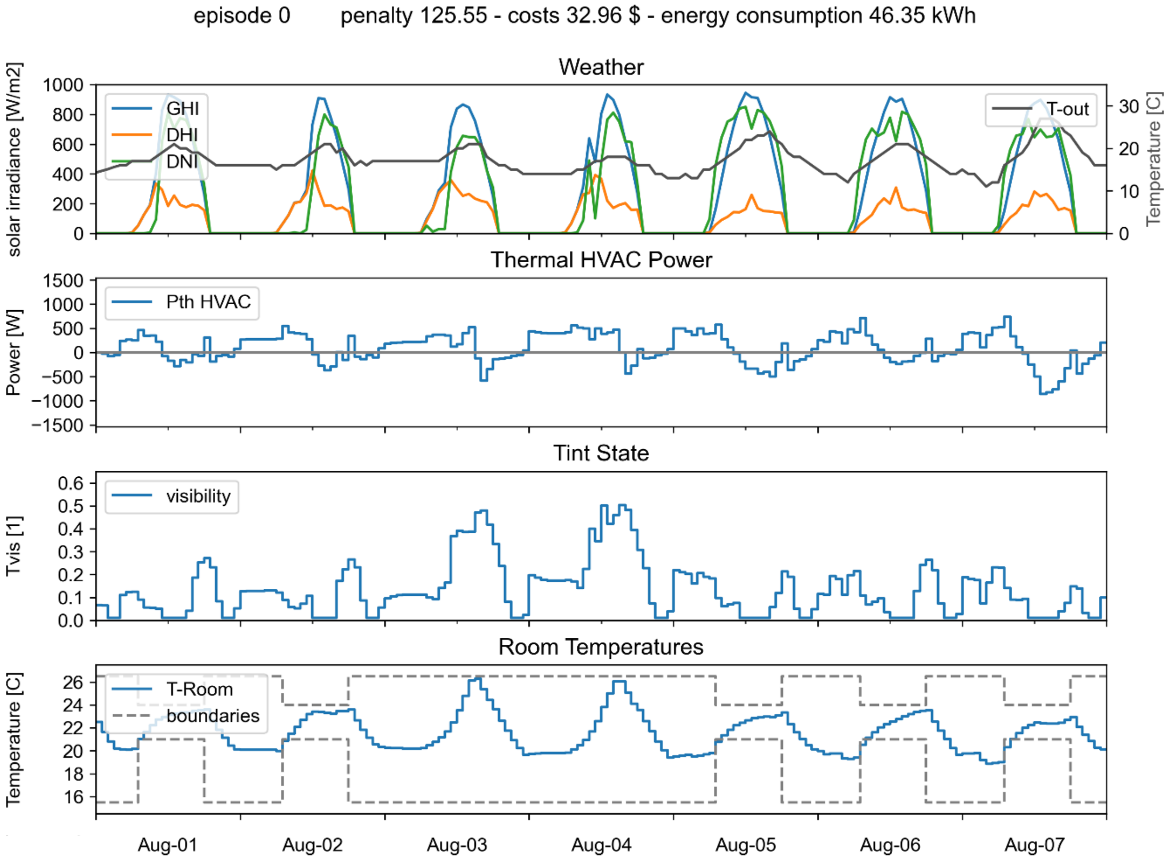

Figure 45: Training result with the new critic of a week starting on August 1st with HVAC and Tint state control

A reason for this behavior may be the lack of forecast data. With forecast data, the agent should be able to change its behavior based on future data. The training process should guide the agent, which timestep is the most important to learn a control strategy that optimizes the energy costs and keep the room temperature within the boundaries. Due to the low capacitance of the room the long-term dependency is low and thus, leads to the decision of four hours of forecast. The network architecture remains the same, but the forecast input now has 16 values for the four-hour forecast (Figure 46).

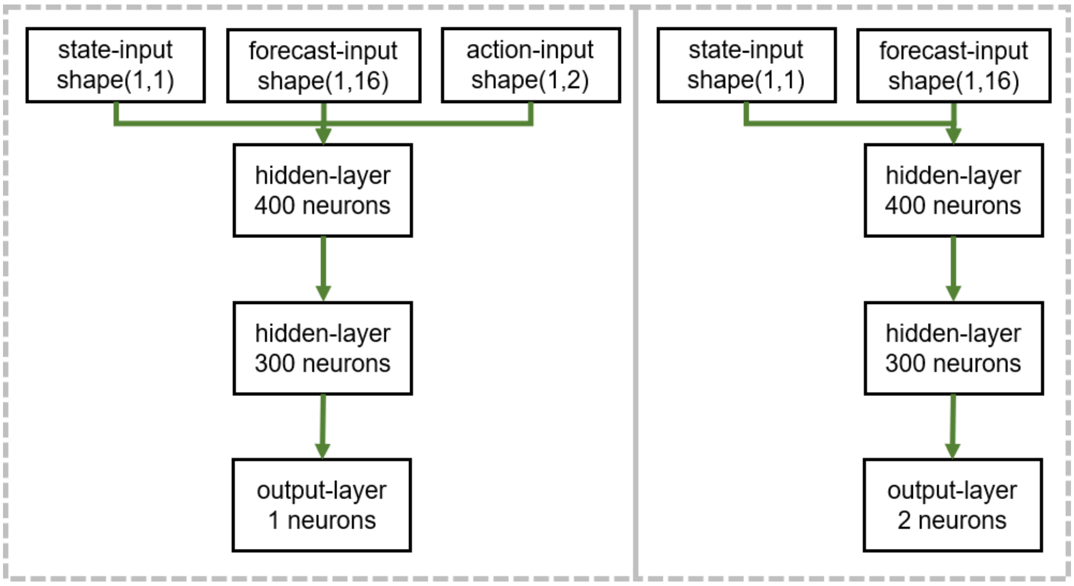

Figure 46: NN setup with 4 forecast hours with the shape of the input vectors with the critic on the left and actor on the right

The critic loss, of the first training run with multiple hours of forecast is lower, than with only current values of the forecast inputs.

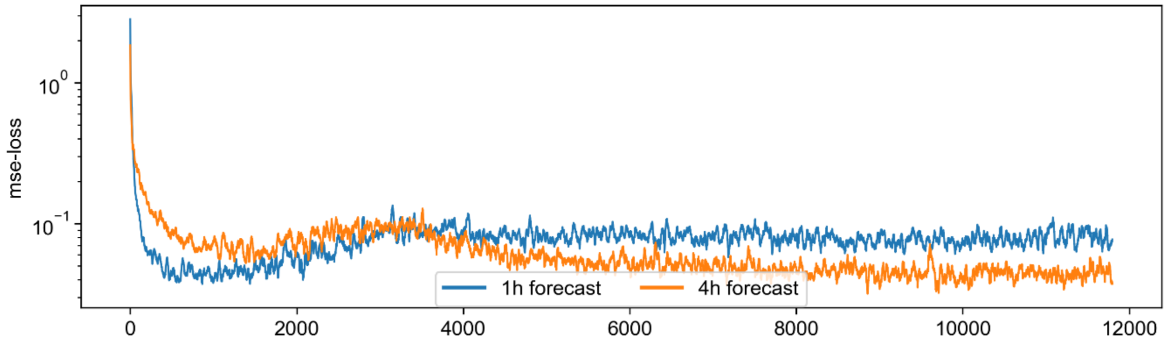

Figure 47: Comparison of critic loss between 1h and 4h forecast

It is not clear why the agent fails in the control task since the loss of the four hour forecast is lower (Figure 47). The agent failed to maintain the room temperature before noon but performs better in terms of penalty and the taken actions do not fluctuate as much as with one hour of forecast, as can be seen in Figure 48.

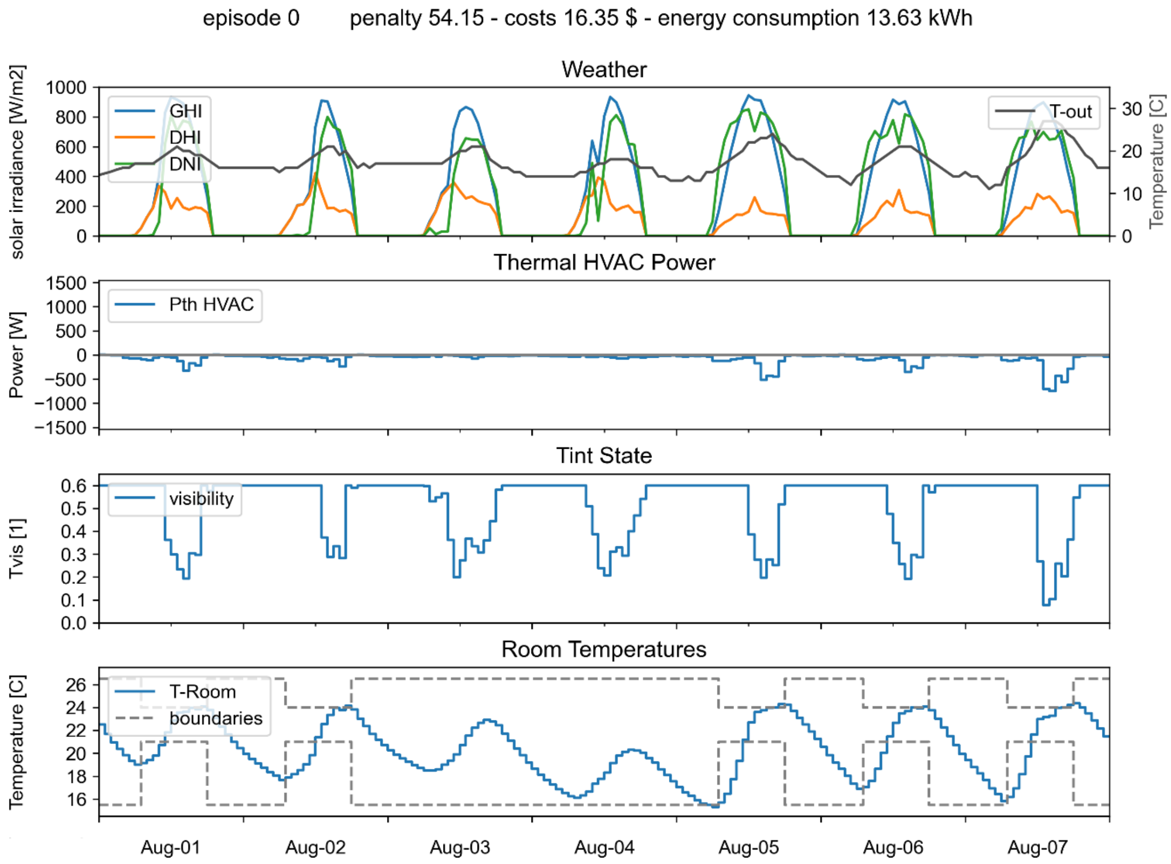

Figure 48: Training run with four forecast hours of a week starting on August 1st with HVAC and Tint state control

The solution in this case is not obvious, therefore a gridsearch is performed, where the network configuration in terms of neurons per layer and number of hidden layers is tested in all possible combinations of one or two hidden layers and a layer size of 300 to 600 neurons with a step size of 100.The best results of the gridsearch in Table 7 compared with the reference of the best run so far, show that the critic with two hidden layers tends to be more accurate and the penalty for the test run with a larger first layer than the second layer is lower.

Table 7: Gridsearch results of best NN configurations

jobID	hidden layer	layer size 1	layer size 2	critic loss	test Aug 1st	test Jan 1st
Ref	1	400	300	0.026	54.15	164.58
30	2	500	300	0.101	32.81	71.78
07	1	600	400	0.491	35.19	102.41
27	2	400	300	0.245	35.42	81.78
06	1	600	300	0.232	37.83	141.14
25	1	600	400	0.277	39.42	110.48

The results of the gridsearch in Figure 49 indicate, that long term dependencies were not taken into account by the agent and thus, tend to react too slow on changes of outside air temperatures. Herein, all agents manage to keep the room temperature within the boundaries with a similar behavior. The best EC-window control was achieved by the agent with jobID 07 which keeps a brighter tint state of the window from August 3rd to August 4th to keep the room temperature higher compared to the other agents.

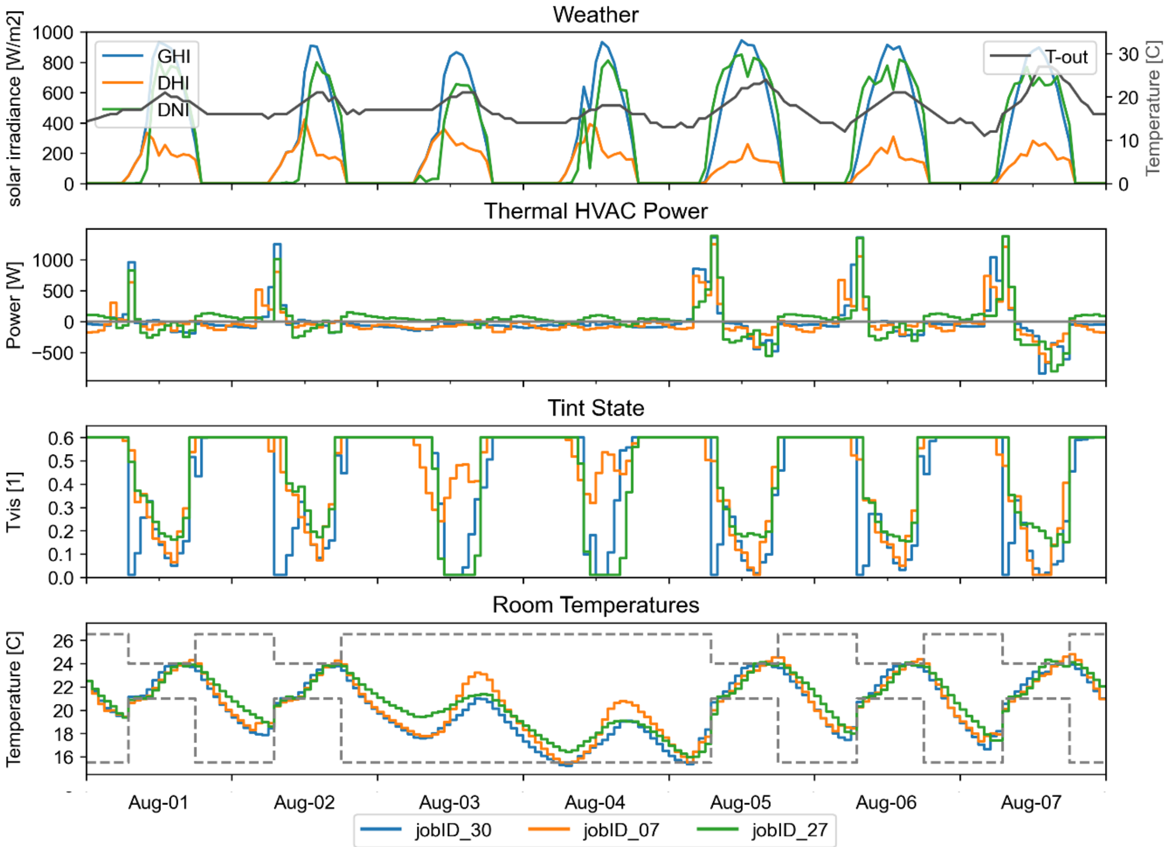

Figure 49: Comparison of best gridsearch results of a week starting on August 1st with HVAC and Tint state control

The results of the performance in terms of the costs, penalty and the maximum peak load, of the best results of the gridsearch, are summarized in Figure 50. The agent with jobID 07 has the lowest peak load with 1.41 kW with a small difference to the other two runs which have a peak load of 1.54 kW and 1.58 kW. Based on these performances and general policy of the agent during the test week, the next improvement step is conducted with all three agents.

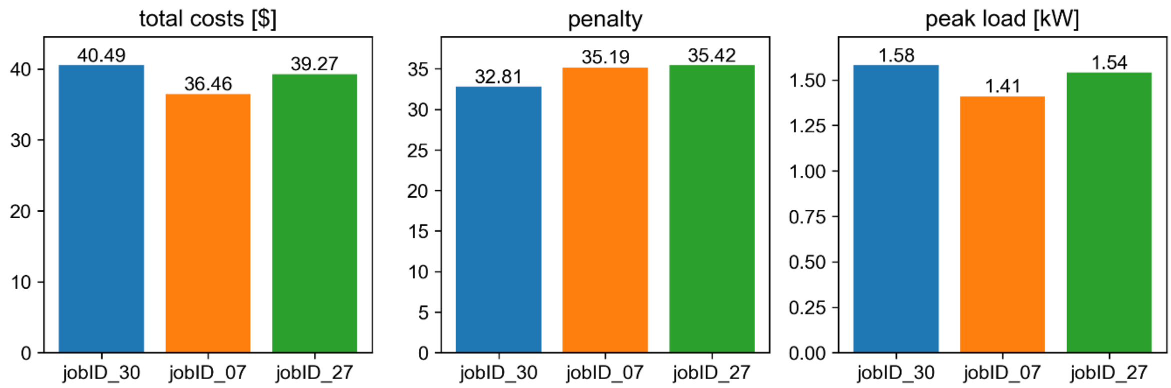

Figure 50: Performance measure of the best gridsearch results

As already described the four-hour forecast does not lead the agent to a successful strategy and increasing the forecast to eight hours does not contribute to an improvement either. A possible way to overcome the issue of a too short dependence, is to use the N-step reward for calculating the action-values with the critic. A variation of the DDPG algorithm was proposed in 2018 called the Distributed Distributional Deterministic Policy Gradient which outperforms the DDPG algorithm (Barth-Maron et al. 2018). The use of the N-step reward had the greatest influence on their performance and was most successful with a length of 5 steps. The N-step reward is the sum of discounted rewards of a fixed length and is calculated with equation 41.

$$Y_t = \left(\sum_{n=0}^{N-1} \gamma^n\, r_{t+n}\right) + \gamma^N * Q'\left(o_{N+1}, \mu'\left(o_{N+1}|\theta^{\mu'}\right)|\theta^{Q'}\right) \tag{41}$$

Y_t ……. action-value as target
N ……. length of N-step reward
n …….. step in N-step reward
γ^n …… discount factor
o_{N+1} … observation of timestep t+1 (state and forecast values)
r_{t+n} …. reward of timestep t
Q' ……. target critic
$\theta^{Q'}$ ….. parameters of target critic
μ' ……. target actor
$\theta^{\mu'}$ ….. parameters of target actor

The calculation of a three N-step reward looks like following example with the transition from the start step with time t to the time step t+2 as the last step.

$$Y_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2}$$

The trajectories with the N-step reward are stored with the observation of the current timestep (s_t, f_t) with the timestep after the N-step reward (s_{t+N+1}, f_{t+N+1}) as $(s_t, f_t, a_t, s_{t+N+1}, f_{t+N+1}, Y_t)$. The trajectories are stored for every timestep, to gather as many trajectories as possible for the training process and not have any gaps in the stored data.

The gridsearch for the best fitting N-step reward will be run with a possible N-steps of 2, 3, 4, 5. The best results indicate, that the optimal length of the N-step for this problem is four steps of the run jobID 27, by taking both, the test week starting on August 1st and the week starting on January 1st into account. However, the critic loss for the run is higher, than of the run with 2 N-steps because a longer N-step reward makes it hard for the critic to estimate the action-value. The critic has no further information, of the next states and which actions are taken to reach the current state.

Table 8: Gridsearch results of best N-step reward

jobID	hidden layer	layer size 1	layer size 2	N-step	critic loss	test Aug 1st	test Jan 1st
Ref	2	500	300	1	0.101	32.81	71.78
30_2	2	500	300	2	0.108	33.15	106.79
27_4	2	400	300	4	0.431	39.04	86.60
07_2	1	600	400	2	0.283	41.92	129.78

With the N-step reward system, the agent uses the forecast data to his advantage and precools or preheats the room displayed in Figure 51. The maximum peak demand can be lowered by all agents compared to the 1 step reward used in the basic DDPG algorithm. The agent with jobID 07_2 has promising behavior for tinting the EC-window and the oscillation of the thermal HVAC power on the weekend is the lowest but fails to keep the room temperature in the boundaries. This agent reacts always a bit slower than the two others. A non-optimal behavior, regarding the tinting of the EC-window is seen by the agent with jobID 30_2. This agent tints the window on the weekend what leads to lower room temperatures and a higher demand to heat up the building. The N-step length of 4 of the agent with jobID 27_4 leads to a farsighted behavior and a similar good tinting behavior as the agent with jobID 07_2.

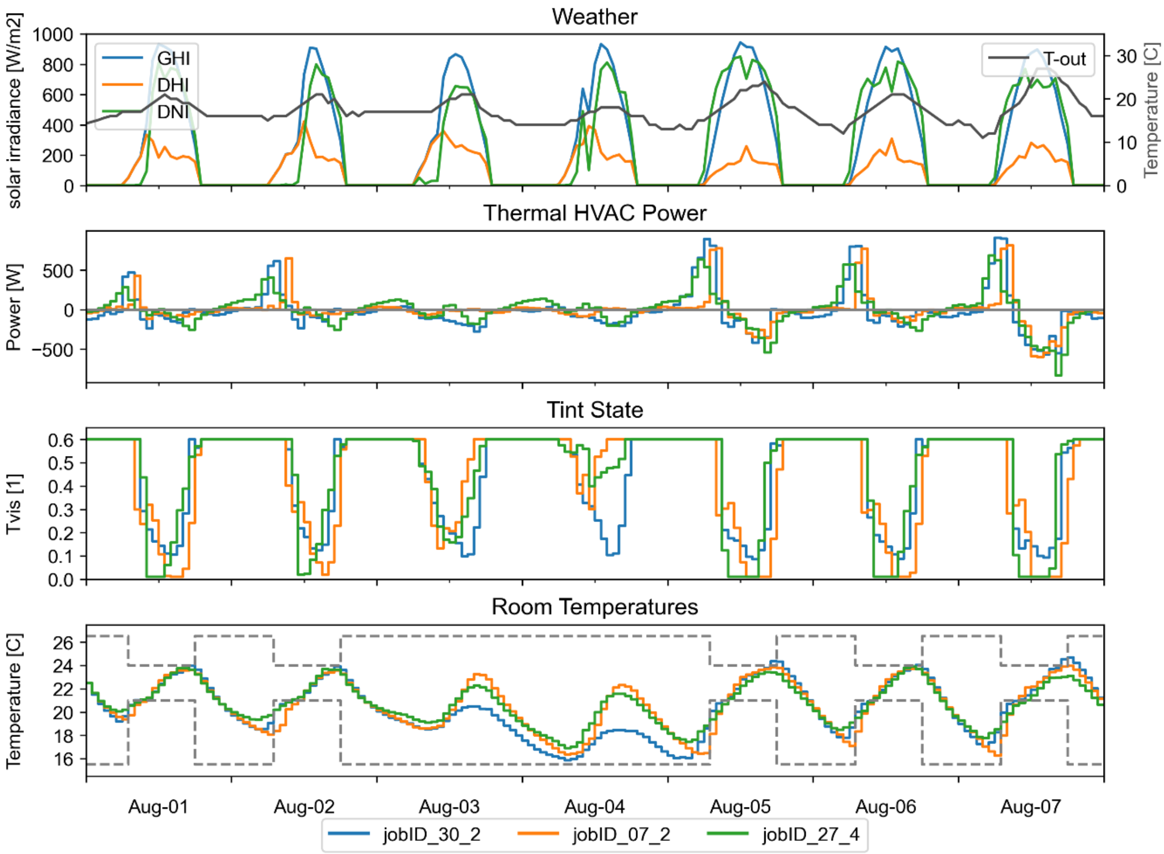

Figure 51: Comparison of gridsearch results for different N-step rewards starting on August 1st

The performance measures are compared in Figure 52 and show that the agent with jobID 27_4 with a N-step length of four has the highest cost saving potential and has the lowest impact on the power grid.

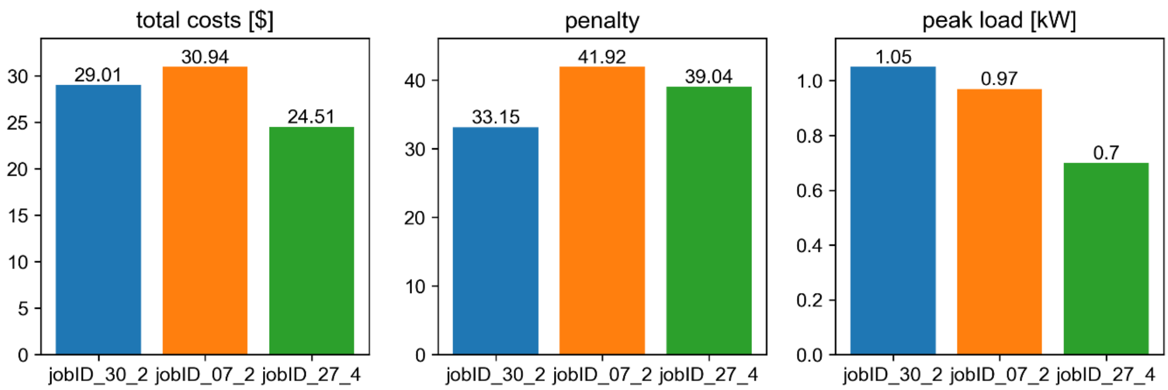

Figure 52: Performance measure of the best gridsearch results for different N-step rewards

The improvement introduced in chapter 4.1 with the replay buffer, noise process and the activation function of the hidden layers are applied in the final run with the agent with jobID 27 as the most promising. Following options are possible for the improvements:

- Activation function
    - Rectified Linear Unit – relu
    - Leaky Rectified Linear Unit – lrelu
- Replay Buffer
    - Uniform
    - Prioritized Experience Replay – PER
    - High-Value Prioritized Experience Replay – HVPER
- Noise process
    - Ornstein Uhlenbeck noise – OU
    - Gaussian noise – Gauss
    - Parameter noise – Param

The network architecture and size of the hidden layers from Figure 46 with a N-step length of 4 proves itself by keeping the room temperature within the boundaries with the best combinations of improvements shown in Table 9. Both versions of an improved replay buffer with priority sampling led to increased accuracy of the critic network. However, this does not automatically lead to a better performing actor. The new configurations are not as good in the test week starting in August as the best agent so far but perform better in the winter. The prioritized replay buffer led to an agent that is more generic, meaning it works not only in cooling mode, but also in heating mode. The combinations with the relu activation functions perform better, as also shown by Ding et al. (Ding et al. 2018). The noise process does not show any differences in performance, but as Barth-Maron also stated is, that the complexity of the Ornstein-Uhlenbeck noise is not benefiting the training compared to the simpler Gaussian noise. 3,000 episodes with a length of 24 steps/hours are not enough to train an agent to its optimum. The agent with jobID 07 is the most generic when comparing both the summer and winter performance.

Table 9: Gridsearch results of the best improvements to the agent

jobID	activation function	replay buffer	noise process	critic loss	test Aug 1st	test Jan 1st
Ref	relu	Uniform	OU	0.431	39.04	86.60
07	relu	HVPER	Gauss	0.047	42.60	78.16
03	relu	PER	OU	0.022	54.91	80.91
06	relu	HVPER	OU	0.015	54.91	83.66

The timeseries comparison of the agents in Figure 53 show the similar behavior for the HVAC system. Especially jobID 07 and 03 show the similar peak demand whereas, the agent with jobID 06 cools with a higher power in the afternoon, which is not so relevant for the total reward and the the demand costs since the COP for cooling is 3.5, which is visible in the lower graph in Figure 53. In this lower graph the sum of all electric power consumers is displayed and shows that the agent with jobID 07 saves the most energy especially visible on the weekend on August 3rd and August 4th. Clearly better is the tint behavior of the agent with jobID 07 with almost no tint on the weekend, whereas the other agents behave almost the same as on weekdays. The tint behavior is even better without the recent improvements for agent 03 and 06.

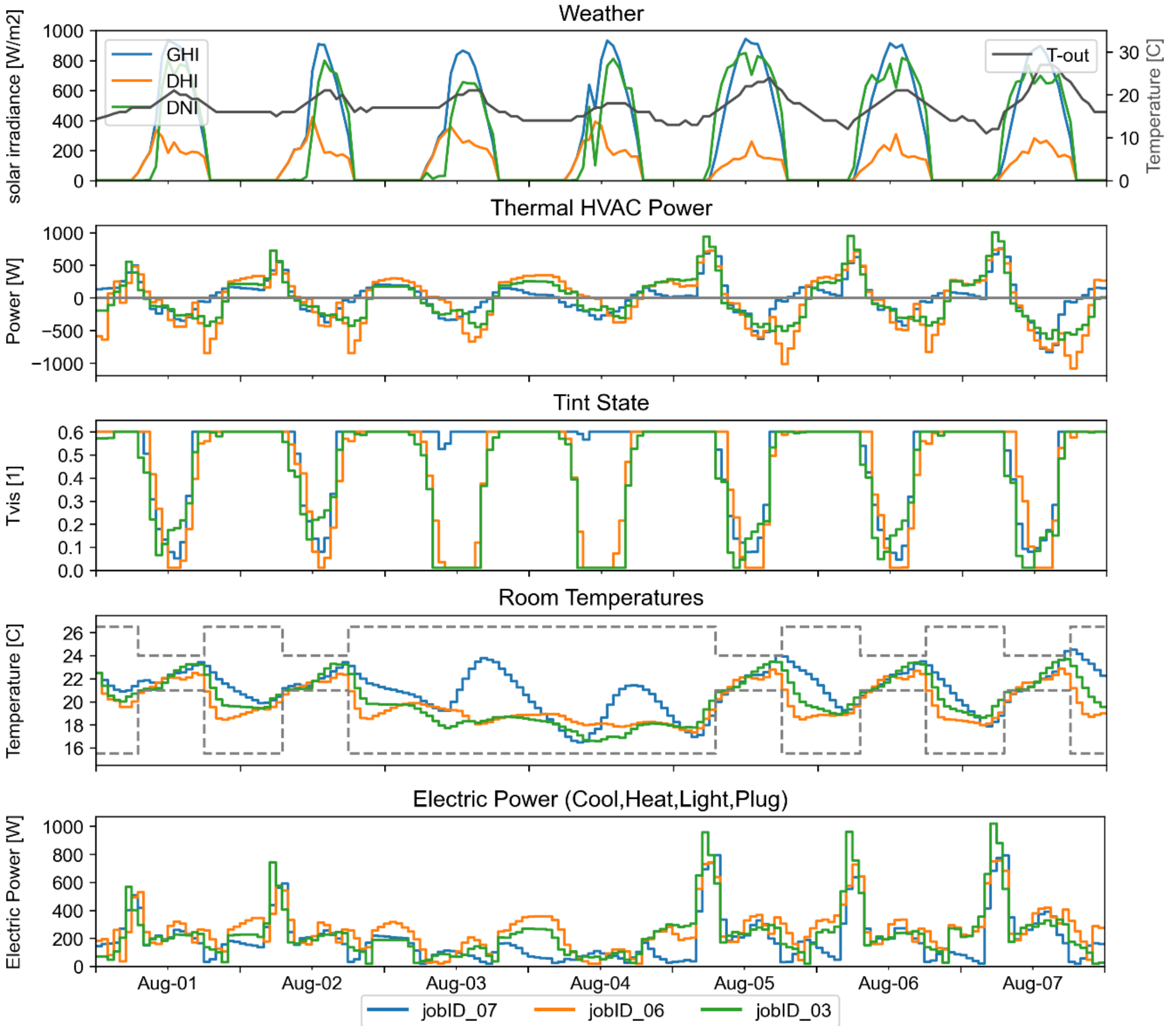

Figure 53: Comparison of gridsearch results for the improvements starting on August 1st

.The performance measures in Figure 54 declare the agent with jobID 07 as the best agent regarding the total costs and penalty. The peak load is 3.8 % higher as of the agent with jobID 06 and 29.1 % lower as the peak load of the agent with jobID 03.

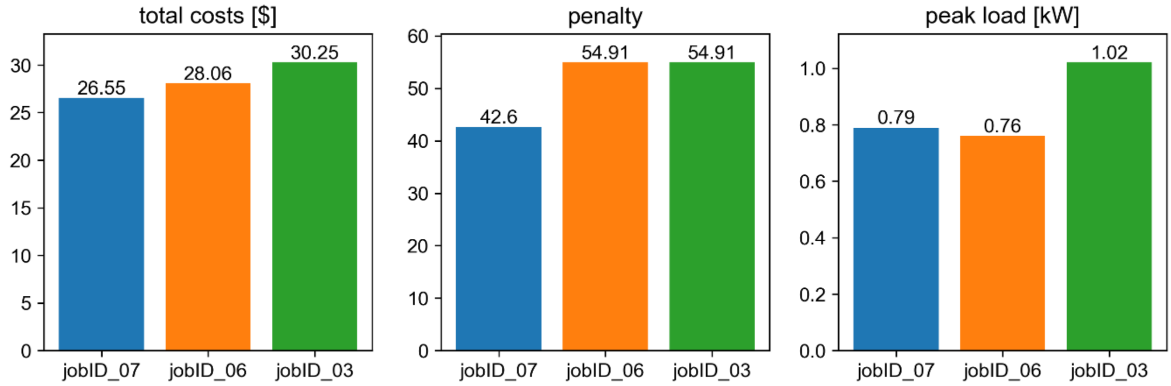

Figure 54: Performance measure of the best gridsearch results for the improvements

With the N-step reward, introduced for the multi-layer perceptron network long term dependencies are taken into account but the agent misses knowledge of the steps taken after the initial step. Only the initial observation and action and the final observation are stored in the replay buffer. An algorithm developed for time series dependent problems published by Google DeepMind is the Recurrent Deterministic Policy Gradient (RDPG)with a LSTM network for both the actor and the critic (Heess et al. 2015). For this algorithm, the entire history of steps as $(o_1, a_1, o_2, a_2, \dots a_{t-1,} o_t)$ is used for selecting actions with the deterministic policy μ. The critic network therefore is initialized as $Q(h, a|\theta^Q)$ and the actor as $\mu(h|\theta^\mu)$. Same as in the DDPG algorithm noise is added to the selected actions to explore the continuous action space. The value function introduced in chapter 4.1 DDPG stays unchanged and is calculated for every step.

The NN architecture is based on the experimental setup of Song et al. published in 2019. In their paper, the inputs for the NNs were based on a pixels and numerical inputs. Since that is not the case for this thesis, the layers dedicated to the pictures are not used. The adapted architecture is shown in Figure 55 with the critic and actor network. The inputs for the critic are the observation-and action history and for the actor only the observation history is passed. The forecast in the observation is passed with the current values.

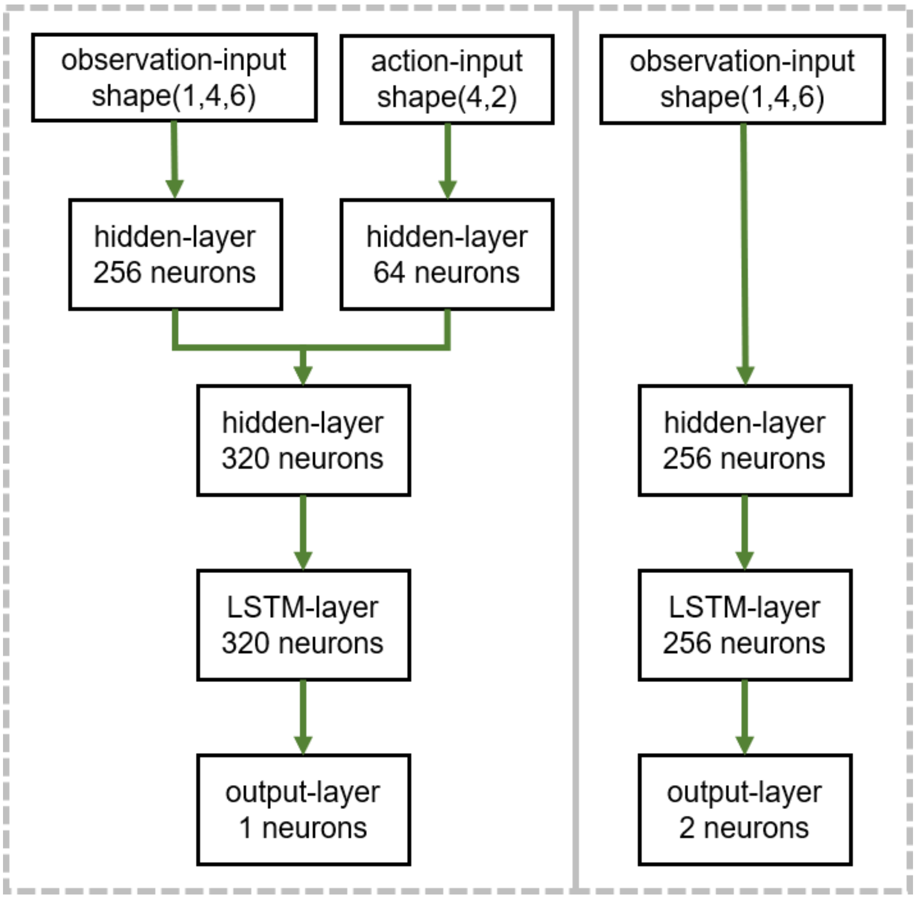

Figure 55: NN setup for the RDPG with the shape of the input vectors vectors with the critic on the left and actor on the right

With the described setup and the beforehand selected improvements for the activation function, the replay buffer and the noise process, the agent with the RDPG algorithm is successful in keeping the room temperature between the boundaries. The taken actions for the EC-windows, however, are not beneficial for cost saving. The lack of forecast information also leads to high peak loads for heating and cooling.

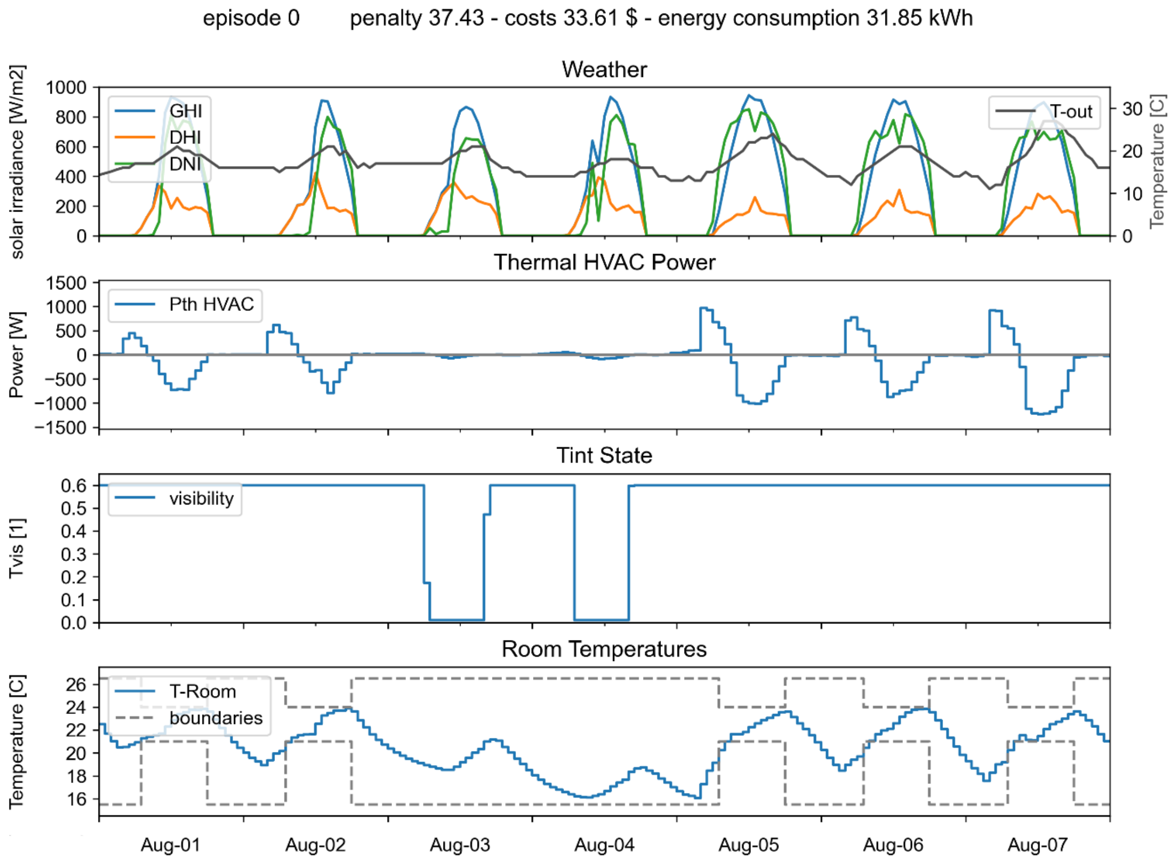

Figure 56: Training run with the LSTM network of a week starting on August 1st

The same approach as with the DDPG algorithm of four forecast hours as inputs does not lead to any improvements but leads to a failure of the agent.

Therefore, the latest DDPG agent is the best performing agent and is compared with the PI-controller and the MPC in Figure 57. The agent has not the same foresight, as the MPC but can decrease the maximum peak compared to the PI-controller. During the high-priced period, the agent reduces the load to save operation costs. The agent, as it is clearly visible is fluctuating around zero between heating and cooling on the weekend where it is not necessary according to the MPC. The control of the EC-window is close to the MPC, which could be seen as the perfect behavior. The fifth chart shows the WPI where it is visible, that the MPC, as well as the agent control the EC-window to minimize the energy consumption for lighting. The artificial light would brighten the room to exactly 350 lx.

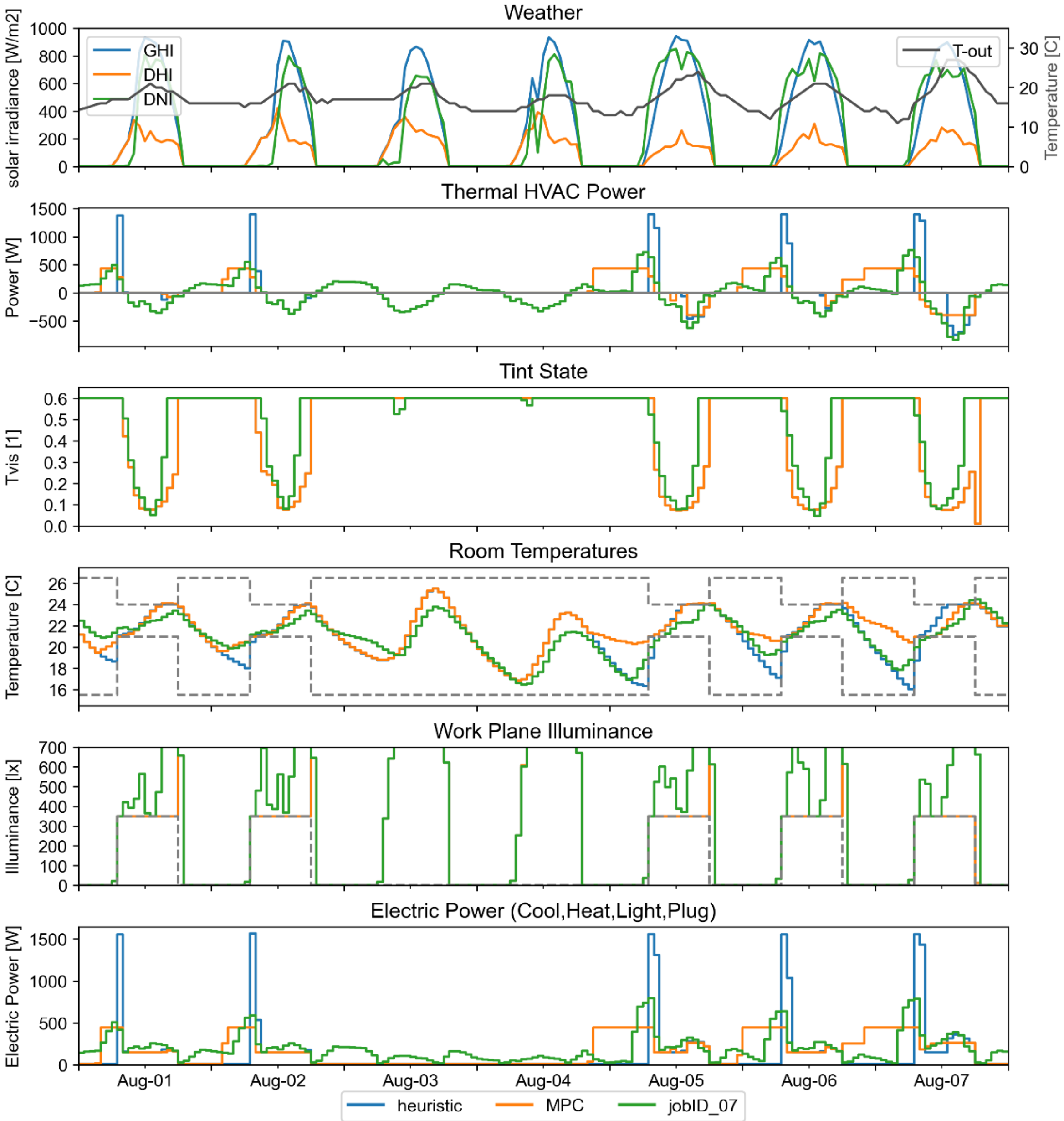

Figure 57: Results of the final RL agent compared to the PI controller and MPC of a week starting on August 1st

The MPC as a perfect information model precools or preheats the room, which leads to a 43.04 % lower peak load (Figure 58) compared to the agent with jobID 07. The PI controller has peak loads of 1,57 kW which is 98.7 % higher than the peak load of the agent. The MPC has a total energy consumption of 28.65 kWh which led to costs for demand and energy of 17.49 $. With the PI controller the required room conditions need 22.28 kWh but, because of the higher peak load the demand and energy cost 38.04 $. The agent controls the HVAC system and EC-window in a way, that it consumes 32.14 kWh, which costs 26.55 $ in total.

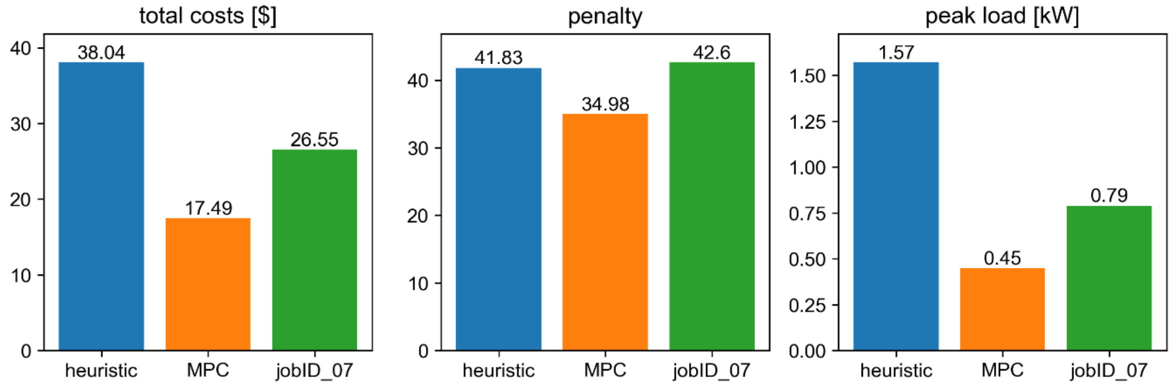

Figure 58: Performance measure compared between PI-controller, MPC and best agent

5 Discussion and Outlook

The aim of this thesis was the implementation of a machine learning agent which strives to minimize the total operation costs of a room, while ensuring the comfort parameters for the occupants. One of the main tasks was the question which Reinforcement Learning (RL) methodology would be best suited for the control of building technology to further reduce total energy costs compared to state-of-the-art controllers and MPC controllers.

The agent, in this thesis was developed for the heating and cooling control of an office building, as well as its shading system with input values for the weather-forecast, occupancy and TOU-tariff. The latter is the most crucial factor for a cost-effective control system. The TOU-tariff as a main input value for the agent enables the power grid operator to actively manage the energy load of the building by changing energy costs for a short period of time. To keep the operation costs low the agent/controller must react to the changes. This possibility for the power grid operator will help to increase the share of renewable energy systems, without the necessity to reinforce the power grid. The advantage of the agent for building owners are the significantly lower total operation costs compared to state-of-the-art PI-controller. The main reason is the agent minimizing the maximum peak load and energy consumption during high priced periods with the agent learning a control strategy to keep costs low. With a controller that takes the total energy costs into account, including for the HVAC-system, as well as for the artificial light and all equipment, the illuminance level of the room remains unknown to the agent and is not required for training or operating. Compared to the MPC, the agent is not as farsighted and has a higher defined peak demand. However, the peak is building up over several timesteps which makes it easier for the power grid operator to predict upcoming peak demands. Whereas the MPC has a stable power level, for heating or cooling, with a sharp increase of power. The agent is not able to outperform the MPC in terms of operation costs but manages to control the temperature with the HVAC system and the EC-window with a similar behavior and performance as the MPC.

The agent’s actions taken are never zero, but rather oscillate on the weekend where the room temperature would stay within the set boundaries even if no actions were taken by the agent. A solution for this problem could be a hierarchical agent setup. An agent would for example set the goal for the room temperature and the illuminance level and the underlying agent would try to take actions to reach these goals while an additional threshold would prevent the agent from performing unnecessary actions. Another important incident to consider is the change of the tariff or the tariff structure by the electricity utilities. It is important to recalibrate the normalization of the TOU tariff for the NN input to ensure a successful behavior with the new tariff.

The agent could be trained as a generalized agent for multiple weather zones and RC models prior to deploying it to a real building. Therefore, the performance in the real building would be acceptable and the chance for training a really good performing agent is higher, due to the lower risk of a biased agent. Therefore, the training time in different buildings could be decreased. Feedback loops for occupants could be integrated in the reward function which would lead to an agent, that fits the occupants comfort expectations. This thesis shows the high potential of machine learning in building controls for multiple actions and constraints.

Bibliography

Alpaydin, E., 2010, Introduction to machine learning. 2nd ed. London, England, The MIT Press

Barrett, E., Linder, S., 2015, Autonomous HVAC Control, A Reinforcement Learning Approach. in: A. Bifet, M. May, B. Zadrozny, R. Gavalda, D. Pedreschi, F. Bonchi, J. Cardoso, & M. Spiliopoulou (eds.), Machine Learning and Knowledge Discovery in Databases. Lecture Notes in Computer Science. Cham, Springer International Publishing; 3–19

Barth-Maron, G., Hoffman, M.W., Budden, D., Dabney, W., Horgan, D., TB, D., Muldal, A., Heess, N., Lillicrap, T., 2018, Distributed Distributional Deterministic Policy Gradients. arXiv:1804.08617 [cs, stat]; http://arxiv.org/abs/1804.08617; 14.7.2020

Bradley, R., 2018, 16 Examples of Artificial Intelligence (AI) in Your Everyday Life | The Manifest.; https://themanifest.com/development/16-examples-artificial-intelligence-ai-your-everyday-life; 19.11.2020

BRE, 2015, A Technical Manual for SBEM.; https://www.uk-ncm.org.uk/filelibrary/SBEM-Technical-Manual_v5.2.g_20Nov15.pdf; 26.11.2020

Brownlee, J., 2019, 14 Different Types of Learning in Machine Learning. Machine Learning Mastery; https://machinelearningmastery.com/types-of-learning-in-machine-learning/; 18.8.2020

Brownlee, J., 2016, Crash Course On Multi-Layer Perceptron Neural Networks. Machine Learning Mastery; https://machinelearningmastery.com/neural-networks-crash-course/; 23.11.2020

Cao, X., Wan, H., Lin, Y., Han, S., 2019, High-Value Prioritized Experience Replay for Off-Policy Reinforcement Learning. in: 2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI). Portland, OR, USA; 1510–1514

CEC, 2020, Electric Utility Service Area.; https://cecgis-caenergy.opendata.arcgis.com/pages/pdf-maps; 26.11.2020

Chen, Y., Norford, L.K., Samuelson, H.W., Malkawi, A., 2018, Optimal control of HVAC and window systems for natural ventilation through reinforcement learning. Energy and Buildings, Volume 169, Number; 195–205

DeepMind, 2016, AlphaGo: The story so far. Deepmind; /research/case-studies/alphago-the-story-so-far; 13.7.2020

Deru, M., Field, K., Studer, D., Benne, K., Griffith, B., Torcellini, P., Liu, B., Halverson, M., Winiarski, D., Rosenberg, M., Yazdanian, M., Huang, J., Crawley, D., 2011, U.S. Department of Energy Commercial Reference Building Models of the National Building Stock.

Ding, B., Qian, H., Zhou, J., 2018, Activation functions and their characteristics in deep neural networks. in: 2018 Chinese Control And Decision Conference (CCDC). Shenyang; 1836–1841

Duffie, J.A., Beckman, W.A., 2013, Solar Engineering of Thermal Processes. fourth Edition. Madison, John Wiley & Sons

EnergyPlus, 2019, Weather Data by Location. EnergyPlus; https://energyplus.net/weather-location/north_and_central_america_wmo_region_4/USA/CA/USA_CA_Oakland.Intl.AP.724930_TMY3; 28.11.2020

Ertel, W., 2016, Neuronale Netze. in: Grundkurs Künstliche Intelligenz. Wiesbaden, Springer Fachmedien Wiesbaden; 265–311

Fernandes, L.L., Lee, E.S., Dickerhoff, D., Thanachareonkit, A., Wang, T., Gehbauer, C., 2018, Electrochromic Window Demonstration at the John E. Moss Federal Building, 650 Capitol Mall, Sacramento, California.

Fujimoto, S., van Hoof, H., Meger, D., 2018, Addressing Function Approximation Error in Actor-Critic Methods. arXiv:1802.09477 [cs, stat]; https://arxiv.org/pdf/1802.09477.pdf; 3.10.2020

Gehbauer, C., Blum, D.H., Wang, T., Lee, E.S., 2020, An assessment of the load modifying potential of model predictive controlled dynamic facades within the California context. Energy and Buildings, Volume 210, Number; 109762

Google Inc., 2019, Keras: the Python deep learning API.; https://keras.io/; 13.11.2020

Haarnoja, T., Zhou, A., Abbeel, P., Levine, S., 2018, Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. arXiv:1801.01290 [cs, stat]; http://arxiv.org/abs/1801.01290; 21.11.2020

Haenlein, M., Kaplan, A., 2019, A Brief History of Artificial Intelligence: On the Past, Present, and Future of Artificial Intelligence. California Management Review, Volume 61, Number 4; 5–14

Hale, J., 2019, Deep Learning Framework Power Scores 2018. Medium; https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a; 23.7.2020

Hale, J., 2020, Which Deep Learning Framework is Growing Fastest? Medium; https://towardsdatascience.com/which-deep-learning-framework-is-growing-fastest-3f77f14aa318; 23.7.2020

Heess, N., Hunt, J.J., Lillicrap, T.P., Silver, D., 2015, Memory-based control with recurrent neural networks. arXiv:1512.04455 [cs]; http://arxiv.org/abs/1512.04455; 14.7.2020

Heinrich, B., Linke, P., Glöckler, M., 2020, Grundlagen Automatisierung: Erfassen - Steuern - Regeln. 3rd ed. Springer Vieweg

Hochreiter, S., Schmidhuber, J., 1997, Long Short-Term Memory.; http://www.bioinf.jku.at/publications/older/2604.pdf

Hong, T., Wang, Z., Luo, X., Zhang, W., 2020, State-of-the-art on research and applications of machine learning in the building life cycle. Energy and Buildings, Volume 212, Number; 109831

Lee, E.S., Selkowitz, S., Clear, R., DiBartolomeo, D., Klems, J., Fernandes, L.L., Ward, G., Inkarojrit, V., Yazdanian, M., 2006, A Design Guide for Early-Market Electrochromic Windows. California Energy Commission, PIER. 500-01-023. LBNL-59950

Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D., 2015, Continuous control with deep reinforcement learning. arXiv:1509.02971 [cs, stat]; https://arxiv.org/pdf/1509.02971v1.pdf; 4.12.2020

Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D., 2019, Continuous control with deep reinforcement learning. arXiv:1509.02971 [cs, stat]; https://arxiv.org/pdf/1509.02971v6.pdf; 20.5.2020

Mitchell, T.M., 1997, Machine Learning. New York, McGraw-Hill

Mnih, V., Badia, A.P., Mirza, M., Graves, A., Harley, T., Lillicrap, T.P., Silver, D., Kavukcuoglu, K., 2016, Asynchronous Methods for Deep Reinforcement Learning. arXiv:1602.01783v2 [cs.LG]; https://arxiv.org/pdf/1602.01783v2.pdf; 10.12.2020

Mohammed, M., Khan, M.B., Bashier, E.B.M., 2017, Machine Learning: Algorithms and Applications. 0 ed. Boca Raton : CRC Press, 2017., CRC Press

Mousavi Maleki, S., Hizam, H., Gomes, C., 2017, Estimation of Hourly, Daily and Monthly Global Solar Radiation on Inclined Surfaces: Models Re-Visited. Energies, Volume 10, Number 1; 134

Olah, C., 2015, Understanding LSTM Networks -- colah’s blog.; http://colah.github.io/posts/2015-08-Understanding-LSTMs/; 23.11.2020

Open Data Science, 2019, What is TensorFlow? Medium; https://medium.com/@ODSC/what-is-tensorflow-13200525e852; 11.11.2020

OpenAI, 2018, Part 2: Kinds of RL Algorithms — Spinning Up documentation.; https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#a-taxonomy-of-rl-algorithms; 14.7.2020

Pan, Y., 2016, Heading toward Artificial Intelligence 2.0. Engineering, Volume 2, Number 4; 409–413

PG&E, 2020a, Electric Schedule E-19 Medium General Demand-Metered TOU Service, effective April 19,2020.; https://www.pge.com/tariffs/assets/pdf/tariffbook/ELEC_SCHEDS_E-19.pdf; 2.10.2020

PG&E, 2020b, PG&E, Pacific Gas and Electric - Gas and power company for California.; https://www.pge.com/; 26.11.2020

Pierre, C., 2020, PYPL PopularitY of Programming Language index.; http://pypl.github.io/PYPL.html; 23.7.2020

Plappert, M., Houthooft, R., Dhariwal, P., Sidor, S., Chen, R.Y., Chen, X., Asfour, T., Abbeel, P., Andrychowicz, M., 2018, Parameter Space Noise for Exploration. arXiv:1706.01905 [cs, stat]; http://arxiv.org/abs/1706.01905; 20.5.2020

Python Software Foundation, 2020, Python. Python.org; https://www.python.org/; 12.5.2020

Rüdisser, D., 2018, A brief guide and free tool for the calculation of the thermal mass of building components (according to ISO 13786).; http://rgdoi.net/10.13140/RG.2.2.18312.72967; 26.11.2020

Sandia National Laboratories, 2019, Pyomo. Pyomo; http://www.pyomo.org; 13.11.2020

Schaul, T., Quan, J., Antonoglou, I., Silver, D., 2016, Prioritized Experience Replay. arXiv:1511.05952 [cs]; https://arxiv.org/pdf/1511.05952.pdf; 14.5.2020

Schulman, J., Levine, S., Moritz, P., Jordan, M.I., Abbeel, P., 2017, Trust Region Policy Optimization. arXiv:1502.05477 [cs]; http://arxiv.org/abs/1502.05477; 21.11.2020

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O., 2017, Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs]; https://arxiv.org/pdf/1707.06347.pdf; 21.11.2020

Seif, G., 2019, Understanding the 3 most common loss functions for Machine Learning Regression. Medium; https://towardsdatascience.com/understanding-the-3-most-common-loss-functions-for-machine-learning-regression-23e0ef3e14d3; 24.11.2020

State of California, 2018, California Costumer Choice.; https://www.cpuc.ca.gov/uploadedFiles/CPUC_Public_Website/Content/Utilities_and_Industries/Energy_-_Electricity_and_Natural_Gas/Cal%20Customer%20Choice%20Report%208-7-18%20rm.pdf; 22.9.2020

Sutton, R.S., Barto, A.G., 2018, Reinforcement learning: an introduction. Second edition. Cambridge, Massachusetts, The MIT Press

Turing, A.M., 1950, I.—COMPUTING MACHINERY AND INTELLIGENCE. Mind, Volume LIX, Number 236; 433–460

U.S. Energy Information Administration, 2020, U.S. energy facts explained - consumption and production - U.S. Energy Information Administration (EIA).; https://www.eia.gov/energyexplained/us-energy-facts/; 9.11.2020

Wang, C.-F., 2019, The Vanishing Gradient Problem. Medium; https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484; 24.11.2020

Wang, Y., Kuckelkorn, J., Liu, Y., 2017, A state of art review on methodologies for control strategies in low energy buildings in the period from 2006 to 2016. Energy and Buildings, Volume 147, Number; 27–40

Wang, Z., Hong, T., 2020, Reinforcement learning for building controls: The opportunities and challenges. Applied Energy, Volume 269, Number; 115036

Wei, T., Wang, Y., Zhu, Q., 2017, Deep Reinforcement Learning for Building HVAC Control. in: Proceedings of the 54th Annual Design Automation Conference 2017. DAC ’17: The 54th Annual Design Automation Conference 2017. Austin TX USA, ACM; 1–6

List of Figures

List of Tables

List of Abbreviations

AI	Artificial Intelligence
DDPG	Deep Deterministic Policy Gradient
DHI	Diffuse Horizontal Irradiance
DL	Deep Learning
DNI	Direct Normal Irradiance
GHI	Global Horizontal Irradiance
HVAC	Heating Ventilating Air Conditioning
HVPER	High-Value Prioritized Experience Replay
LBNL	Lawrence Berkeley National Laboratory
lrelu	Leaky Rectified Linear Unit
LSTM	Long-Short Term Memory
ML	Machine Learning
MPC	Model Predictive Control
NN	Neural Network
PER	Prioritized Experience Replay
PG&E	Pacific Gas and Electricity
PI	Proportional-Integra
PID	Proportional-Integral-Derivative
RC	Resistance and Capacitance
RDPG	Recurrent Deterministic Policy Gradient
relu	Rectified Linear Unit
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SHGC	Solar Heat Gain Coefficient
sig	Sigmoid
tanh	Hyperbolic tangent

TD	Temporal Difference
TOU	Time-Of-Use
Tv	Visibility Transmittance
WPI	Workplace Illuminance

Appendix A: Setting

Parameter setting

```python
import numpy as np
import os

def get_parameter(l_wall, U_wall, U_window, A_room, h_room, A_window, step_size, model, \
                  hvac_control, radiance, max_power):
    rho_air = 1.1894 # Density, in kg/m3
    cp_air = 1.0086 # Air Specific heat capacity, in kJ/kgK
    R_wall = (U_wall * (l_wall * h_room - A_window))
    R_window = (U_window * A_window)
    C_int_wall = ((l_wall + A_room/l_wall)*2*h_room - l_wall*h_room)*8.184 # Drywall Capacitance in kJ/K
    C_int_floor_ceiling = 2*A_room * 43.7545#[kJ/K] light concrete floor and ceiling
    C_Air = A_room * h_room * rho_air * cp_air #[kJ/K]
    C_ext_wall = l_wall*h_room * 35.284 #[kJ/K]
    C_room = (C_int_wall + C_Air + C_ext_wall + C_int_floor_ceiling)
    operatingsys = 'windows' if os.name == 'nt' else 'linux'

    weather_columns = ['weaCelHei','weaCloTim','weaHDifHor','weaHDirNor',
                       'weaHGloHor','weaHHorIR','weaNOpa','weaNTot',
                       'weaPAtm','weaRelHum','weaSolTim','weaSolZen',
                       'weaTBlaSky','weaTDewPoi','weaTDryBul',
                       'weaTWetBul','weaWinDir','weaWinSpe']
    parameter = {}

    # Inputs
    parameter['inputs'] = {}
    parameter['inputs']['labels'] = ['Q_hvac', 'Tvis', 'start_time', 'T_out', 'S_irr', 'S_ill', 'Q_int_th', 'Q_int_el',
                                  'Occ', 'C_energy', 'C_demand', 't_min', 't_max', 'wpi_min'] + weather_columns

    # input data setting
    parameter['input_data'] = {}
    parameter['input_data']['step_size'] = step_size # hour

    # Window
    tints = np.array([[0.42,0.6],[0.16,0.18],[0.12,0.06],[0.1,0.01]]) # [shgc, Tvis]
    coeff = np.poly1d(np.polyfit(tints[:,1], tints[:,0], 1))
    parameter['window'] = {}
    parameter['window']['area'] = A_window # Window area, in m2
    parameter['window']['coeff_a'] = coeff[0] # Window tint fit funciton
    parameter['window']['coeff_b'] = coeff[1] # Window tint fit funciton

    # Room configuraiton
    parameter['zone'] = {}
    parameter['zone']['area'] = A_room # Room area, in m2
    parameter['zone']['height'] = h_room # Room height, in m
    parameter['zone']['length'] = l_wall # Room length, in m
    parameter['zone']['surface_area'] = ((parameter['zone']['length'] + parameter['zone']['area']\
                                     / parameter['zone']['length']) * parameter['zone']['height']\
                                     + parameter['zone']['area']) * 2
    parameter['zone']['t_init'] = None # Initial room temperature, in K (None = random)
    parameter['zone']['t_init_min'] = 21 + 273.15 # Minimal initial room temperate; Minimum room temperature when occupied, in K
    parameter['zone']['t_init_max'] = 24 + 273.15 # Maximal initial room temperate; Maximum room temperature when occupied, in K
    parameter['zone']['eff_lights'] = 5 * A_room / 500 # 5 W/m2 => W/lux ==> LPD of 0.5 W/ft2
    parameter['zone']['eff_heat'] = 1 # Efficieny of heating
    parameter['zone']['eff_cool'] = 1/3.5 # Efficiency of cooling
    parameter['zone']['int_th_load'] = 100 * np.round(parameter['zone']['area']/18.58) # NREL --> 1 person on 18.58 m2
    parameter['zone']['int_el_load'] = 10.76*parameter['zone']['area'] # --> 10.76 W/m2
    parameter['zone']['office_hours'] = [[7,18]]
```

```python
    if model == 'RC':
        parameter['zone']['control_hvac'] = False
    else:
        parameter['zone']['control_hvac'] = hvac_control # Flag to control HVAC system

    # Constraints
    parameter['constraints'] = {}
    parameter['constraints']['max_t_penalty'] = 2 # Maximal reward penalty
    parameter['constraints']['night_tint_penalty'] = 1 # penalty for tinting the windows during the night
    parameter['constraints']['cool_max'] = max_power * A_room # Maximal cooling power, in W
    parameter['constraints']['heat_max'] = max_power * A_room # Maximal heating power, in W
    parameter['constraints']['wpi_min'] = 350 # Minimum work place illuminance (wpi) when occupied, lux
    parameter['constraints']['t_min'] = 15.5 + 273.15 # Minimum temperature when not occupied, K
    parameter['constraints']['t_max'] = 26.5 + 273.15 # Maximum temperature when not occupied, K

    # demand charge properties
    parameter['demand_charge'] = {}
    parameter['demand_charge']['period'] = 24 * 30.436875 # demand is charged every month, in h
    parameter['demand_charge']['base_charge'] = 17.63 # demand is charged every month, in h

    parameter['tariff'] = {}
    parameter['tariff']['periods'] = {}
    # month
    parameter['tariff']['periods']['summer'] = {}
    parameter['tariff']['periods']['winter'] = {}
    parameter['tariff']['periods']['summer']['month'] = [5,6,7,8,9,10]
    parameter['tariff']['periods']['winter']['month'] = [1,2,3,4,11,12]
    # day of week
    parameter['tariff']['dayofweek'] = {}
    parameter['tariff']['dayofweek']['weekday'] = [0,1,2,3,4]
    parameter['tariff']['dayofweek']['weekend'] = [5,6]
    # hour of day
    parameter['tariff']['periods']['summer']['s_peak'] = [[12,6+12]]
    parameter['tariff']['periods']['summer']['s_part_peak'] = [[8.5,12],[6+12,9.5+12]]
    parameter['tariff']['periods']['summer']['s_off_peak'] = [[0,8.5],[9.5+12,12+12]]
    parameter['tariff']['periods']['winter']['w_part_peak'] = [[8.5,9.5+12]]
    parameter['tariff']['periods']['winter']['w_off_peak'] = [[0,8.5],[9.5+12,12+12]]
    # energy rate
    parameter['tariff']['C_energy'] = {}
    parameter['tariff']['C_energy']['s_peak'] = 0.16225
    parameter['tariff']['C_energy']['s_part_peak'] = 0.11734
    parameter['tariff']['C_energy']['s_off_peak'] = 0.08846
    parameter['tariff']['C_energy']['w_part_peak'] = 0.11127
    parameter['tariff']['C_energy']['w_off_peak'] = 0.09559
    # demand rate
    parameter['tariff']['C_demand'] = {}
    parameter['tariff']['C_demand']['s_peak'] = 19.63
    parameter['tariff']['C_demand']['s_part_peak'] = 5.37
    parameter['tariff']['C_demand']['s_off_peak'] = 0.0
    parameter['tariff']['C_demand']['w_part_peak'] = 0.18
    parameter['tariff']['C_demand']['w_off_peak'] = 0.0
    parameter['tariff']['C_demand']['base_rate'] = 17.63

    # parameters for RL_Agent
    parameter['agent']={}
    parameter['agent']['stepsize'] = parameter['input_data']['step_size'] # hour
    parameter['agent']['Agent'] = 'DDPG' # DDPG, RDPG
    parameter['agent']['dtype'] = 'float32'
    parameter['agent']['actions'] = 0 # 0 = Q_hvac+Tvis; 1 = Q_hvac; 2 = Tvis
```

```python
    # set the hyperparameters for the neural network
    parameter['agent']['NN'] = {}
    parameter['agent']['NN']['network_architecture'] = 'MLP' # MLP, LSTM
    parameter['agent']['NN']['hidden_layers'] = 2
    parameter['agent']['NN']['layer_size_1'] = 400
    parameter['agent']['NN']['layer_size_2'] = 300
    parameter['agent']['NN']['activation'] = 'relu'
    parameter['agent']['NN']['tow'] = 0.001
    parameter['agent']['NN']['discount_factor'] = 0.99
    parameter['agent']['NN']['demand_charge_scale'] = 1
    parameter['agent']['NN']['act_learning_rate'] = 0.0001
    parameter['agent']['NN']['crit_learning_rate'] = 0.001

    parameter['agent']['setting'] = {}
    parameter['agent']['setting']['n_step'] = 4
    parameter['agent']['setting']['forecast_hours']  = 4
    parameter['agent']['setting']['training_days'] = 1
    parameter['agent']['setting']['test_days'] = 7
    parameter['agent']['setting']['noise_process'] = 'Gauss_noise' # OU_noise, Gauss_noise, param_noise
    parameter['agent']['setting']['forecast_col'] = ['weaTDryBul', 'S_irr', 'TOU_tariff', 'occupancy']
    parameter['agent']['setting']['episodes'] = 3000
    parameter['agent']['setting']['episodes_with_noise'] = 0.5
    parameter['agent']['setting']['exploration_episodes'] = 50

    parameter['agent']['replay_buffer'] = {}
    parameter['agent']['replay_buffer']['Buffer'] = 'HVPER' # Uniform, PER, HVPER
    parameter['agent']['replay_buffer']['max_buffer_size'] = int(1e9)
    parameter['agent']['replay_buffer']['batch_size'] = 512

    parameter['agent']['scaling'] = {}
    parameter['agent']['scaling']['forecast_norm_data'] = np.array([[0.+273.15,0.,-max(parameter['tariff']['C_energy'].values()),0.],
                                                                    [34.+273.15,1000.,max(parameter['tariff']['C_energy'].values()),2.1]])
    parameter['agent']['scaling']['state_norm_data'] = np.array([[parameter['constraints']['t_min']],[parameter['constraints']['t_max']]])
    parameter['agent']['scaling']['actions_norm_data'] = np.array([[-parameter['constraints']['heat_max'],-tints[:,1].max()],[parameter['constraints']['heat_max'] ,tints[:,1].max()]])
    parameter['agent']['scaling']['noise_scale'] = parameter['constraints']['heat_max']*0.15, 0.1 #if action_noise 2 dim, if param noise 1 dim
    parameter['agent']['scaling']['reward_0'] = 24 * 30.436875 / parameter['agent']['setting']['n_step']

    # set flags
    parameter['agent']['flags'] = {}
    parameter['agent']['flags']['valuefunction'] = True
    parameter['agent']['flags']['plot'] = True
    parameter['agent']['flags']['print'] = True
    parameter['agent']['flags']['hp_tune'] = False
    parameter['agent']['flags']['load_model'] = False
    parameter['agent']['flags']['save_models'] = True

    weather_config = {}
    weather_config['start_time'] = '2019-01-01 00:00:00'
    weather_config['stepsize'] = parameter['input_data']['step_size'] * 60*60 #set hourly timestep in parameter
    weather_config['weather_dir'] = r'resources\weather\\'
    weather_config['weather_columns'] = ['weaCelHei','weaCloTim','weaHDifHor','weaHDirNor',
                                         'weaHGloHor','weaHHorIR','weaNOpa','weaNTot',
                                         'weaPAtm','weaRelHum','weaSolTim','weaSolZen',
```

```
                                         'weaTBlaSky','weaTDewPoi','weaTDryBul',
                                         'weaTWetBul','weaWinDir','weaWinSpe']

    return parameter, weather_config
```

Appendix B: Execution

Training

```python
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
from tf_agents.environments import tf_py_environment

from environment import Room
from input_data import get_weather_files
from input_data_handler import RL_results_handler, files_handler
from parameter_handler import get_parameter
from RL_Agent import agent

# set room parameters and select model
l_wall = 4 # Length of wall, in m
U_wall = 3.294 # U-value wall, in W/m2K
U_window = 6.923 # U-value window, in W/m2K
A_room = 14 # Room area, in m2
h_room = 3.95 # Room height, in m
A_window = l_wall * h_room * 0.33 # Window area, in m2
step_size = 60/60 # hour
model = 'RC' #RC = simple python
hvac_control = False # if True modelica controls the heating and cooling
radiance = False # if True, solar heat gain is calculated with radiance
max_power = 100 # specific power input W/m2
# get all parameters necessary, based on the settings
parameter, weather_config = get_parameter(l_wall, U_wall, U_window, A_room, h_room, \
                          A_window, step_size, model, hvac_control, radiance, max_power)

# parameters for RL_Agent
parameter['agent']['Agent'] = 'DDPG' # DDPG, RDPG(LSTM)
parameter['agent']['actions'] = 0 # 0 = Q_hvac&Tvis; 1 = Q_hvac; 2 = Tvis

# set the hyperparameters for the neural network
parameter['agent']['NN']['network_architecture'] = 'MLP' # MLP, LSTM
parameter['agent']['NN']['hidden_layers'] = 2
parameter['agent']['NN']['layer_size_1'] = 400
parameter['agent']['NN']['layer_size_2'] = 300
parameter['agent']['NN']['activation'] = 'relu' # relu, leakyrelu
parameter['agent']['NN']['demand_charge_scale'] = 1

parameter['agent']['setting']['n_step'] = 4
parameter['agent']['setting']['forecast_hours'] = 4
parameter['agent']['setting']['training_days'] = 1
parameter['agent']['setting']['test_days'] = 7
parameter['agent']['setting']['noise_process'] = 'Gauss_noise' # OU_noise, Gauss_noise, param_noise
parameter['agent']['setting']['forecast_col'] = ['weaTDryBul', 'S_irr', 'TOU_tariff', 'occupancy']
parameter['agent']['setting']['episodes'] = 5000
parameter['agent']['setting']['episodes_with_noise'] = 0.5
parameter['agent']['setting']['exploration_episodes'] = 50

parameter['agent']['replay_buffer']['Buffer'] = 'HVPER' # Uniform, PER, HVPER
parameter['agent']['replay_buffer']['batch_size'] = 512

# set flags
parameter['agent']['flags']['plot'] = True
parameter['agent']['flags']['print'] = True
parameter['agent']['flags']['hp_tune'] = False
parameter['agent']['flags']['load_model'] = False
parameter['agent']['flags']['save_models'] = True

''' enter the name of the country or abbrevation of state or city as a list for the weather
   file as string ('rand' = load all files in directory)'''

weather_config['location'] = ['Oakland']
weather_config['weather_path'] = get_weather_files(weather_config)

# if file is not valid for this directory, select a new one
```

```python
while not weather_config['weather_path']:
    print('location not available in this path')
    weather_config['location'] = [input('enter new location: ')]
    weather_config['weather_path'] = get_weather_files(weather_config)
print(weather_config['weather_path'])

job_id = 0

# train a new network/test pretrained network
mode = input("enter train or test: ")
# load all input data files/ weather data files
input_files = files_handler(weather_config, parameter)
if mode == 'train':
    train_results = RL_results_handler()
    test_results = RL_results_handler()
    env = tf_py_environment.TFPyEnvironment(Room(parameter))
    Agent = agent(env, job_id, parameter, input_files, train_results, test_results)
    train_results, test_results = Agent.train()
elif mode == 'test':
    parameter['agent']['flags']['load_model'] = True
    parameter['agent']['flags']['save_models'] = False
    parameter['zone']['t_init'] = 22.5+273.15
    test_results = RL_results_handler()
    env = tf_py_environment.TFPyEnvironment(Room(parameter))
    Agent = agent(env, job_id, parameter, input_files, None, test_results)
    testing = 'select' #input('"select" for new location or "rand" for random
    initialized location: ') #rand to choose randomly from weather data set and set
    random date, select when setting new location and date
    if testing == 'select':
        weather_config['location'] = ['Oakland']#[input('Enter City for the location
        file: ')]# select city for weather file, as string (None = random)
        weather_config['weather_path'] = get_weather_files(weather_config)
        while not weather_config['weather_path']:
            print('location not available in this path')
            weather_config['location'] = [input('enter new location: ')]
            weather_config['weather_path'] = get_weather_files(weather_config)
            print('What date (month,day) should be the first day of the episode?')
        date = [int(input('month: ')), int(input('day: '))]
    else: date = [None,None]

    data = files_handler(weather_config, parameter).input_files['file0']
    training_days = int(input('how many days should be tested? '))
    test = Agent.test(testing ,data, training_days, date[0], date[1])
```

Gridsearch

```python
import copy
import multiprocessing as mp
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import pandas as pd
from sklearn.model_selection import ParameterGrid
import tensorflow as tf
from tf_agents.environments import tf_py_environment
from environment import Room

def ddpg_worker(job):
    from environment import Room
    from input_data import get_weather_files
    from input_data_handler import files_handler
    from RL_Agent import agent

    job_id = job[0]
    params = job[1]

    weather_config = {}
    weather_config['start_time'] = '2019-01-01 00:00:00'
    weather_config['stepsize'] = params['input_data']['step_size'] * 60*60 #set hourly timestep in parameter
    weather_config['weather_dir'] = r'resources\weather\\'
    weather_config['location'] = ['Oakland'] # enter the name of the city as a list for the weather file, as string (None = random)
    weather_config['weather_path'] = get_weather_files(weather_config)
    weather_config['weather_columns'] = ['weaCelHei','weaCloTim','weaHDifHor','weaHDirNor',
                                         'weaHGloHor','weaHHorIR','weaNOpa','weaNTot',
                                         'weaPAtm','weaRelHum','weaSolTim','weaSolZen',
                                         'weaTBlaSky','weaTDewPoi','weaTDryBul',
                                         'weaTWetBul','weaWinDir','weaWinSpe']

    input_files = files_handler(weather_config, params)
    ## load the environment and wrap it to tf_environment
    envpy = Room(params)
    env = tf_py_environment.TFPyEnvironment(envpy)

    Agent = agent(env, job_id, params, input_files, None, None)

    return Agent.train()

if __name__ == '__main__':
    from parameter_handler import get_parameter
    # set environment settings and get all parameters
    l_wall = 4 # Length of wall, in m
    U_wall = 3.293 # U-value wall, in W/m2K
    U_window = 6.923 # U-value window, in W/m2K
    A_room = 14 # Room area, in m2
    h_room = 3.95 # Room height, in m
    A_window = l_wall * h_room * 0.33 # Window area, in m2
    step_size = 60/60 # size of timesteps hour
    model = 'RC' #RC = simple function
    max_power = 100 # specific heating/cooling power input W/m2
    parameter,_ = get_parameter(l_wall, U_wall, U_window, A_room, h_room, \
                                A_window, step_size, model, hvac_control, radiance, max_power)

    # Make grid
    # all_possible settings are listed below
    agent_algorithm = list(['DDPG', 'RDPG'])
    agent_architecture = list(['MLP', 'LSTM'])
    hidden_layers = list([2])
    layer_1 = list([400,500,600])
```

```
    layer_2 = list([300,400,500])
    n_step = list([1,2,3,4])
    forecast_hours = list([1,2,3,4])
    noise_process = list(['OU_noise', 'Gauss_noise', 'param_noise'])
    buffer = list(['Uniform', 'PER', 'HVPER'])
    activation = list(['relu', 'leakyrelu'])
    batch_size = list([512])
    parameterset = dict(agent_algorithm=agent_algorithm, agent_architecture=agent_architecture,\
                        hidden_layers=hidden_layers, layer_1=layer_1, layer_2=layer_2,\
                        forecast_hours=forecast_hours, n_step=n_step, noise_process=noise_process,\
                        buffer=buffer, activation=activation, batch_size=batch_size)

    params = list(ParameterGrid([parameterset]))
    jobs = copy.deepcopy(params)
    # remove all not suitable or impossible configurations
    for dict_p in params:
        if dict_p['agent_algorithm'] == 'RDPG' and dict_p['agent_architecture'] != 'LSTM':
            jobs.remove(dict_p)
            continue
        if dict_p['agent_algorithm'] == 'DDPG' and dict_p['agent_architecture'] == 'LSTM':
            jobs.remove(dict_p)
            continue
        if dict_p['agent_algorithm'] == 'DDPG' and dict_p['forecast_hours'] == 1:
            jobs.remove(dict_p)
            continue
    parameter_list = []

    for params in jobs:
        job_params = {}
        job_params['inputs'] = copy.deepcopy(parameter['inputs'])
        job_params['input_data'] = copy.deepcopy(parameter['input_data'])
        job_params['window'] = copy.deepcopy(parameter['window'])
        job_params['zone'] = copy.deepcopy(parameter['zone'])
        job_params['constraints'] = copy.deepcopy(parameter['constraints'] )
        job_params['somodel'] = copy.deepcopy(parameter['somodel'])
        job_params['model'] = copy.deepcopy(parameter['model'])
        job_params['tariff'] = copy.deepcopy(parameter['tariff'])
        job_params['agent'] = copy.deepcopy(parameter['agent'])
        job_params['agent']['flags']['plot'] = False
        job_params['agent']['flags']['print'] = False
        job_params['agent']['flags']['hp_tune'] = True
        job_params['agent']['flags']['load_model'] = False
        job_params['agent']['flags']['save_models'] = True
        job_params['agent']['Agent'] = params['agent_algorithm']
        job_params['agent']['NN']['network_architecture'] = params['agent_architecture']
        job_params['agent']['NN']['hidden_layers'] = params['hidden_layers']
        job_params['agent']['NN']['layer_size_1'] = params['layer_1']
        job_params['agent']['NN']['layer_size_2'] = params['layer_2']
        job_params['agent']['NN']['activation'] = params['activation']
        job_params['agent']['setting']['forecast_hours'] = params['forecast_hours']
        job_params['agent']['setting']['noise_process'] = params['noise_process']
        job_params['agent']['setting']['n_step'] = params['n_step']
        job_params['agent']['setting']['training_days'] = 1
        job_params['agent']['setting']['episodes'] = 3000
        job_params['agent']['setting']['episodes_with_noise'] = 0.5
        job_params['agent']['setting']['exploration_episodes'] = 50
        job_params['agent']['replay_buffer']['Buffer'] = params['buffer']
        job_params['agent']['replay_buffer']['batch_size'] = params['batch_size']

        parameter_list.append(job_params)
    #dict with all jobs in a list
    jobs = list(zip(list(range(len(parameter_list))), parameter_list))
```

```
    # Run all jobs in parallel
    pool = mp.Pool(mp.cpu_count()-1)
    physical_devices = tf.config.list_physical_devices()
    print(physical_devices)
    data = pool.map(ddpg_worker, jobs)
    pool.close()

    # Convert to pandas
    data = pd.DataFrame(data)
    data.to_csv('logs/job_results.csv')
```

Anhang C: RL-Setup

Agent

```python
import calendar
from datetime import datetime, timedelta
import json
import numpy as np
import random
import tensorflow as tf
from tensorflow.keras.layers.experimental.preprocessing import Normalization
import tf_agents
import time
print('tensorflow', tf.__version__)
print('tf_agents', tf_agents.__version__)

from AC_NN import AC_network
from input_data_handler import RL_results_handler
from plot import episodeplot, runplot
from ReplayBuffer import Uniform, PER, HVPER

class agent():
    def __init__(self, env, job_id, parameter, input_files, train_results, test_results):

        ''' agent in the RL framework
            Inputs: env .............. initialized environment (tf_Py_environment)
                    job_id ........... only important for hyperparameter tuning
                    parameter ........ parameters set in parameter_handler.py and Main.py
                    input_files ...... all selected weather files in a dict as input_handlers
                    train_results .... Results handler for training data
                    test_results ..... Results handler for test data
        '''
        self.parameter = parameter
        self.env = env

        self.input_files = input_files

        if parameter['agent']['flags']['hp_tune']:
            tf.config.threading.set_inter_op_parallelism_threads(1)
            tf.config.threading.set_intra_op_parallelism_threads(1)
            self.job_id = job_id
            self.train_results = RL_results_handler()
            self.test_results = RL_results_handler()
        else:
            self.train_results = train_results
            self.test_results = test_results

        self.T_train = int(parameter['agent']['setting']['training_days']* 24 / \
                     parameter['agent']['stepsize']) # length of train episode
        self.T_test = int(parameter['agent']['setting']['test_days']*24 / \
                     parameter['agent']['stepsize']) # length of train episode
        self.update_freq = int(6) # update frequency of neural networks
        self.n_Step = parameter['agent']['setting']['n_step']
        self.dflt_dtype = parameter['agent']['dtype']

        # decrease noise scale to 0 after episodes with noise
        self.noise_decrease = 1-(1/(parameter['agent']['setting']['episodes'] * \

                                parameter['agent']['setting']['episodes_with_noise']))
        # select noise scale for action noise shape(action_dim) or param noise shape(1)
        if parameter['agent']['setting']['noise_process'] == 'param_noise':
            self.noise_scale = np.array(0.6)
            self.init_noise_scale = np.array(0.6)
        else:
            self.noise_scale = np.array(parameter['agent']['scaling']['noise_scale'])
            self.init_noise_scale = np.array(parameter['agent']['scaling']['noise_scale'])

```

```
        if parameter['agent']['flags']['print']:
            print('update frequency ',self.update_freq, ' steps')

        # NN parameters
        self.Agent = parameter['agent']['Agent']
        if self.Agent == 'RDPG':
            self.parameter['agent']['NN']['network_architecture'] = 'LSTM'

        # set dimension of NN output
        self.action_dim = env._action_spec.shape[0]

        # get max action set in environment
        self.action_bound_range = env.action_spec().maximum
        self.state_dim = env.reset()[3].shape[0]

        # shape for forecast data
        if self.parameter['agent']['NN']['network_architecture'] == 'CNN':
            self.forecast_dim = (parameter['agent']['setting']['forecast_hours'],
                                 len(parameter['agent']['setting']['forecast_col']))
        else:
            self.forecast_dim = (parameter['agent']['setting']['forecast_hours']\
                                 * len(parameter['agent']['setting']['forecast_col']))

        # normalization layers prior to NN input
        self.norm_state_layer = Normalization()
        self.norm_state_layer.adapt(parameter['agent']['scaling']['state_norm_data'])
        self.norm_forecast_layer = Normalization()
        self.norm_forecast_layer.adapt(parameter['agent']['scaling']['forecast_norm_data'])
        self.norm_action_layer = Normalization()
        if parameter['agent']['actions'] == 0:
            self.norm_action_layer.adapt(parameter['agent']['scaling']['actions_norm_data'])
        elif parameter['agent']['actions'] == 1:
            self.norm_action_layer.adapt(self.parameter['agent']['scaling']['actions_norm_data']\
                                         [:,0].reshape(2,self.action_dim))
        else:
            self.norm_action_layer.adapt(self.parameter['agent']['scaling']['actions_norm_data']\
                                         [:,1].reshape(2,self.action_dim))

        # initialize replay buffer according to replay
        if parameter['agent']['replay_buffer']['Buffer'] == 'PER':
            self.buffer = PER(parameter['agent']['replay_buffer']['max_buffer_size'], parameter['agent']['dtype'], self.forecast_dim)
        elif parameter['agent']['replay_buffer']['Buffer'] == 'HVPER':
            self.buffer = HVPER(parameter['agent']['replay_buffer']['max_buffer_size'], parameter['agent']['dtype'], self.forecast_dim)
        else:
            self.buffer = Uniform(parameter['agent']['replay_buffer']['max_buffer_size'], parameter['agent']['dtype'], self.forecast_dim)

        # importance sampling for prioritized Replay Buffer (PER and HVPER)
        if isinstance(self.buffer, Uniform):
            self.importance_sampling = False
        else:
            self.importance_sampling = True
            self.beta = 0.5 # beta corrects the prioritized sampling probability and changes linear over time to 1
            self.beta_increase = 1 + (1/(parameter['agent']['setting']['episodes'] * self.T_train) * self.update_freq / 2)
```

```python

        # initialize actor and critic network
        self.agent_network = AC_network(self.state_dim, self.forecast_dim, self.action_dim, self.action_bound_range,\
                                        self.norm_forecast_layer, self.norm_state_layer, self.norm_action_layer, parameter)
        # load saved network model
        if parameter['agent']['flags']['load_model'] == True:
            self.agent_network.load_model()
        else:
            self.agent_network.build_actor(self.parameter['agent']['NN']['network_architecture'])
            self.agent_network.build_critic(self.parameter['agent']['NN']['network_architecture'])

        if parameter['agent']['flags']['print']:
            self.agent_network.actor.summary()
            self.agent_network.critic.summary()

    def test(self, testing, data, episode_length, month, day):
        ''' testing of the Agent
            inputs  testing......... selected or random location
                    data............. file_handler for input data files
                    episode_length... how many test days
                    month............ month of test day
                    day.............. start day of testing period
        '''
        if testing == 'rand':
            if self.parameter['agent']['flags']['hp_tune']:
                test_episodes = 1
            else:
                test_episodes = 10
        else:
            test_episodes = 1
            self.T_test = episode_length * 24
        for episode in range(test_episodes):
            if testing == 'rand':
                if len(self.input_files.input_files) > 1 or episode == 0:
                    self.inputs_data, self.parameter['somodel'], year = \
                            self.input_files.select_file(self.parameter['somodel'])

                # randomly select the start date for this episode
                # for hp_tuning all jobs teste with same start day
                if self.parameter['agent']['flags']['hp_tune']:
                    month = 8
                    day = 1
                else:
                    month = random.choice(np.arange(1,13))
                    day = random.choice(np.arange(1,calendar.monthrange(year,month)[1]+1))
                if month >= 12:
                    month = min(month,12)
                    day = min(day, calendar.monthrange(year,month)[1]-self.parameter['agent']['setting']['test_days']-1)
                start_time = datetime(year, month, day,0,0)
                save_figure = False
            else:
                self.inputs_data, self.parameter['somodel'], year = \
                        self.input_files.choose_file(data,self.parameter['somodel'])
                if month >= 12:
                    month = min(month,12)
                    day = min(day, calendar.monthrange(year,month)[1]-episode_length-1)
                start_time = datetime(year, month, day,0,0)
                save_figure = True
            current_time = start_time
```

```
        # Append mapping to fmu inputs (for Buildings Library only)
        if not 'RCmodel' in self.parameter['model']['fmu_path']:
            weather_offset = len(self.inputs_data.cols_inputs) + len(self.inputs_data.cols_data)
            for i, c in enumerate(self.inputs_data.cols_weather):
                #parameter['model']['inputs_map'][c] = weather_offset + i
                self.parameter['model']['inputs_map'][c] = c

        add_noise = False
        # reset the environment to start setting
        state_t = np.array(self.env.reset()[3]).reshape(1,1)
        # get the init forecast
        forecast_t = self.inputs_data.get_forecast(current_time)
        n_step_forecast = np.zeros(shape=(forecast_t.shape))
        n_step_state = np.zeros(shape=(1,1))
        for t in range(self.T_test):
            if self.Agent == 'RDPG':
                if n_step_state.shape[0] == 1:
                    action_t = self.agent_network.take_action_lstm(state_t[0],forecast_t, add_noise)
                elif 1 < n_step_state.shape[0] < 4 or t < 4:
                    action_t = self.agent_network.take_action_lstm(n_step_state[1:],n_step_forecast[1:],add_noise)
                else:
                    action_t = self.agent_network.take_action_lstm(n_step_state,n_step_forecast,add_noise)
            else:
                action_t = self.agent_network.take_action(state_t, forecast_t, add_noise)
            if np.isnan(action_t).any():
                action_t = np.array([[0,0]])
            if self.parameter['agent']['actions'] == 1:
                Q = action_t[0]
                Tvis = np.array([0.6])
            elif self.parameter['agent']['actions'] == 2:
                Q = np.array([0.])
                Tvis = action_t[0]
            else:
                Q = action_t[0][0]
                Tvis = action_t[0][1]

            env_input = self.inputs_data.get_inputs(Q, Tvis, current_time, start_time)

            _, rwrd_t, _, state_t_pls_n = self.env.step(env_input)
            self.test_results.append_res(self.env, 0, episode, current_time)
            current_time += timedelta(hours=self.parameter['input_data']['step_size'])

            # set the next forecast
            forecast_t_pls_n = self.inputs_data.get_forecast(current_time)

            n_step_state = np.append(n_step_state,state_t, axis=0)
            n_step_forecast = np.append(n_step_forecast,forecast_t, axis=0)

            if (t+1) >= self.n_Step:
                # store the experience into the buffer for updating the critic network
                n_step_state = np.delete(n_step_state,0, axis=0)
                n_step_forecast = np.delete(n_step_forecast,0, axis=0)

            state_t = state_t_pls_n
            forecast_t = forecast_t_pls_n

            if (t+1) % self.T_test == 0:
```

```python
                    rr = self.test_results.data['reward_0'].iloc[-self.T_test:].sum()+\
                         self.test_results.data['reward_1'].iloc[-self.T_test:].sum()
                    if self.parameter['agent']['flags']['print']:
                        print('Episode %d : Total Penalty = %f' % (episode+1, rr))
                    if self.parameter['agent']['flags']['plot']:
                        episodeplot(self.parameter, self.test_results.data.iloc[-int(self.T_test):], save = save_figure, fig_name = str(self.agent_network.AC_name)+str(start_time).split(' ')[0])
        return self.test_results

    def train(self):
        ''' main training function of the Agent'''
        time_start = time.time()
        experience_cnt = 0
        break_out = False
        logs = 0
        rand = True
        add_noise = True
        distance = 0.6

        for episode in range(self.parameter['agent']['setting']['episodes']):
            # reset the environment to start setting
            if len(self.input_files.input_files) > 1 or episode == 0:
                self.inputs_data, self.parameter['somodel'], year = \
                        self.input_files.select_file(self.parameter['somodel'])

            # randomly select the start date for this episode
            month = random.choice(np.arange(1,13))
            day = random.choice(np.arange(1,calendar.monthrange(year,month)[1]+1))
            if month >= 12:
                month = min(month,12)
                day = min(day, calendar.monthrange(year,month)[1]-self.parameter['agent']['setting']['training_days']-1)
            start_time = datetime(year, month, day,0,0)
            current_time = start_time

            # Append mapping to fmu inputs (for Buildings Library only)
            if not 'RCmodel' in self.parameter['model']['fmu_path']:
                weather_offset = len(self.inputs_data.cols_inputs) + len(self.inputs_data.cols_data)
                for i, c in enumerate(self.inputs_data.cols_weather):
                    #parameter['model']['inputs_map'][c] = weather_offset + i
                    self.parameter['model']['inputs_map'][c] = c

            # reset the environment to the startvalues
            state_t = np.array(self.env.reset()[3]).reshape(1,1)

            # get the init forecast
            forecast_t = self.inputs_data.get_forecast(current_time)

            # init add OUA-noise or Guassian noise process at the start of every episode or
            # add param noise to the actor network
            if add_noise:
                if self.parameter['agent']['setting']['noise_process'] == 'param_noise':
                    self.agent_network.param_noise_process.calc_scale(distance)
                    self.agent_network.parameter_noise_handling()
                else:
                    self.noise_scale = self.noise_scale * self.noise_decrease
                    self.agent_network.action_noise_handler(self.noise_scale)

            # initialize n_step buffer
            n_step_state = np.zeros(shape=(1,1))
            n_step_forecast = np.zeros(shape=(forecast_t.shape))
```

```
n_step_forecast_t_pls_n = np.zeros(shape=(forecast_t.shape))
n_step_actions = np.zeros(shape=(1,self.action_dim))
n_step_rwrd_t = np.zeros(shape=(1,1))
n_step_demand = np.zeros(shape=(1,1))
n_step_state_t_pls_n = np.zeros(shape=(1,1))

# start of the episode
for t in range(self.T_train):
    if self.Agent == 'RDPG':
        if n_step_state.shape[0] == 1:
            action_t = self.agent_network.take_action_lstm(state_t[0],forecast_t, add_noise)
        elif 1 < n_step_state.shape[0] < 4 or t < 4:
            action_t = self.agent_network.take_action_lstm(n_step_state[1:],n_step_forecast[1:],add_noise)
        else:
            action_t = self.agent_network.take_action_lstm(n_step_state,n_step_forecast,add_noise)
    else:
        action_t = self.agent_network.take_action(state_t, forecast_t, add_noise)
    if np.isnan(action_t).any():
        break
    if self.parameter['agent']['actions'] == 1:
        Q = action_t[0]
        Tvis = np.array([0.6])
    elif self.parameter['agent']['actions'] == 2:
        Q = np.array([0.])
        Tvis = action_t[0]
    else:
        Q = action_t[0][0]
        Tvis = action_t[0][1]

    # get input data for environment
    env_input = self.inputs_data.get_inputs(Q, Tvis, current_time, start_time)
    # make step and get results
    _, rwrd_t, _, state_t_pls_n = self.env.step(env_input)
    # append all results to dataframe
    self.train_results.append_res(self.env, logs, episode, current_time)
    logs=0
    current_time += timedelta(hours=self.parameter['input_data']['step_size'])

    # set the next forecast
    forecast_t_pls_n = self.inputs_data.get_forecast(current_time)

    # store the step in the n_step_memory for calculation of the n_step_reward
    n_step_state = np.append(n_step_state,state_t, axis=0)
    n_step_forecast = np.append(n_step_forecast,forecast_t, axis=0)
    n_step_forecast_t_pls_n = np.append(n_step_forecast_t_pls_n,forecast_t_pls_n,axis=0)
    n_step_actions = np.append(n_step_actions, action_t,axis=0)
    n_step_rwrd_t = np.append(n_step_rwrd_t,np.array([rwrd_t[0][0]]))
    n_step_demand = np.append(n_step_demand,np.array([rwrd_t[0][1]]))
    n_step_state_t_pls_n = np.append(n_step_state_t_pls_n,state_t_pls_n, axis=0)

    # Calculation of the n_step_reward
    if (t+1) >= self.n_Step:
        # store the experience into the buffer for updating the critic network
        n_step_state = np.delete(n_step_state,0, axis=0)
        n_step_forecast = np.delete(n_step_forecast,0, axis=0)
        n_step_actions = np.delete(n_step_actions,0,axis=0)
```

```python
            if self.Agent != 'RDPG':
                n_step_action = np.array([n_step_actions[0]])
            else:
                n_step_action = n_step_actions
            n_step_rwrd_t = np.delete(n_step_rwrd_t,0, axis=0)
            n_step_demand = np.delete(n_step_demand,0, axis=0)
            n_step_state_t_pls_n = np.delete(n_step_state_t_pls_n,0, axis=0)
            n_step_forecast_t_pls_n = np.delete(n_step_forecast_t_pls_n,0, axis=0)
            reward = 0.

            # calculate n_step reward and set how n_step arrays are stored for diff algorithms
            if self.Agent != 'RDPG':
                for j, (rwrd_j) in enumerate(n_step_rwrd_t):
                    reward += rwrd_j * self.parameter['agent']['NN']['discount_factor']**(j+1)
                demand_charge = n_step_demand[-1]
                forecast = n_step_forecast[0]
                next_forecast = n_step_forecast_t_pls_n[-1]
            else:
                reward = n_step_rwrd_t + n_step_demand[-1]
                reward = reward.reshape(self.n_Step,1)
                demand_charge = n_step_demand
                forecast = n_step_forecast
                next_forecast = n_step_forecast_t_pls_n

            self.buffer.add_experience(n_step_state, forecast, n_step_action, reward, demand_charge,
                                       n_step_state_t_pls_n, next_forecast)

        # TRAINING AND UPDATING THE NETWORKS
        if not rand and experience_cnt % self.update_freq == 0:
            if isinstance(self.buffer,Uniform):
                states_batch, forecast_batch, actions_batch, rewards_batch, demand_batch, next_states_batch, next_forecast_batch = self.buffer.sample_batch(self.parameter['agent']['replay_buffer']['batch_size'])
                indices = None
                importance_weight = np.array([1])
            else:
                self.beta = min(self.beta * self.beta_increase,1)
                states_batch, forecast_batch, actions_batch, rewards_batch, demand_batch, next_states_batch, next_forecast_batch, indices, importance_weight = self.buffer.sample_batch(self.parameter['agent']['replay_buffer']['batch_size'], self.beta)
            num_samples = min(len(states_batch),self.parameter['agent']['replay_buffer']['batch_size'])

            # normalize all sampled batches
            states_batch = self.norm_state_layer(states_batch)
            forecast_batch = self.norm_forecast_layer(forecast_batch)
            actions_batch = self.norm_action_layer(actions_batch)
            next_states_batch = self.norm_state_layer(next_states_batch)
            next_forecast_batch = self.norm_forecast_layer(next_forecast_batch)

            if self.parameter['agent']['NN']['network_architecture'] != 'CNN' and self.Agent !='RDPG':
                forecast_batch = tf.reshape(forecast_batch,(num_samples, self.forecast_dim))
                next_forecast_batch = tf.reshape(next_forecast_batch,(num_samples, self.forecast_dim))
                actions_batch = tf.reshape(actions_batch,(num_samples,self.action_dim))
```

```
                # shapes of sampled batches
                # states_batch .......... shape(num_samples, n_Step, 1)
                # forecast_batch ........ shape(num_samples, forecast_dim)
                # actions_batch ......... shape(num_samples, actions_dim)
                # rewards_batch ......... shape(num_samples, 1)
                # demand_batch .......... shape(num_samples, 1)
                # next_states_batch ..... shape(num_samples, n_Step, 1)
                # next_forecast_batch ... shape(num_samples, forecast_dim)

                if self.Agent != 'RDPG':
                    try:
                        logs, td_error =
                        self.agent_network.train_critic_network(num_samples,states
                        _batch, forecast_batch, actions_batch,
                                                    rewards_batch, demand_batch,
                                                    next_states_batch,
                                                    next_forecast_batch,
                                                    indices, importance_weight)
                        self.agent_network.train_actor_network(num_samples,
                        states_batch, forecast_batch)
                    except:
                        break_out = True
                        break
                    if self.parameter['agent']['setting']['noise_process'] ==
                    'param_noise' and add_noise == True:
                        actions =
                        self.agent_network.actor([states_batch[:,0],forecast_batch
                        ])
                        actions[0] = actions[0]/self.action_bound_range
                        p_actions =
                        self.agent_network.perturbed_actor([states_batch[:,0],fore
                        cast_batch])
                        p_actions[0] = p_actions[0]/self.action_bound_range
                        actions =
                        np.dstack(actions).reshape(num_samples,self.action_dim)
                        p_actions =
                        np.dstack(p_actions).reshape(num_samples,self.action_dim)
                        distance = tf.math.sqrt(1/self.action_dim *
                        tf.reduce_mean(tf.math.square(actions-p_actions)))
                else:
                    forecast_batch = tf.reshape(forecast_batch,(num_samples,
                    self.n_Step, self.forecast_dim))
                    next_forecast_batch =
                    tf.reshape(next_forecast_batch,(num_samples, self.n_Step,
                    self.forecast_dim))
                    hist_t = np.concatenate((states_batch,forecast_batch),axis=2)
                    hist_t_1 =
                    np.concatenate((next_states_batch,next_forecast_batch),axis=2)
                    logs, td_error =
                    self.agent_network.train_critic_lstm(num_samples, hist_t,
                    actions_batch, rewards_batch,
                                                hist_t_1, indices,
                                                importance_weight)
                    self.agent_network.train_actor_lstm(num_samples, hist_t,
                    actions_batch)

                    if self.parameter['agent']['setting']['noise_process'] ==
                    'param_noise' and add_noise == True:
                        actions = self.agent_network.actor([hist_t])
                        actions[0] = actions[0]/self.action_bound_range
                        p_actions = self.agent_network.perturbed_actor([hist_t])
                        p_actions[0] = p_actions[0]/self.action_bound_range
                        actions =
                        np.dstack(actions).reshape(num_samples,self.action_dim)
                        p_actions =
                        np.dstack(p_actions).reshape(num_samples,self.action_dim)
                        distance = tf.math.sqrt(1/self.action_dim *
```

```
                              tf.reduce_mean(tf.math.square(actions-p_actions)))

                        self.buffer.batch_update(indices, td_error)

                    experience_cnt += 1
                    if episode >= self.parameter['agent']['setting']['exploration_episodes']:
                        rand = False
                    if episode == self.parameter['agent']['setting']['episodes_with_noise'] *\
                                self.parameter['agent']['setting']['episodes']:
                        add_noise = False

                    state_t = state_t_pls_n
                    forecast_t = forecast_t_pls_n

                ### Plot of 1 episode
                    if  (t+1) % self.T_train == 0:
                        rr = self.train_results.data['reward_0'].iloc[-self.T_train:].sum()+\
                             self.train_results.data['reward_1'].iloc[-self.T_train:].sum()

                        if self.parameter['agent']['flags']['print']:
                            print('Episode %d : Total Penalty = %f' % (episode, rr))
                        if self.parameter['agent']['flags']['plot'] and episode > self.parameter['agent']['setting']['episodes'] -15:
                            episodeplot(self.parameter, self.train_results.data.iloc[-int(self.T_train):])

                    if experience_cnt % (10*self.T_train) == 0 and self.parameter['agent']['flags']['save_models']:
                        self.agent_network.save_model()
                if break_out:
                    break
        # Plot of 1 run
            if self.parameter['agent']['flags']['plot']:
                runplot(self.train_results.data, save = True, fig_name = str(self.agent_network.AC_name))

            self.test('rand',0,0,0,0)
            time_end = time.time()
            # hyperparameter tuning results
            if self.parameter['agent']['flags']['hp_tune']:
                filename = str('logs/' + str(self.job_id) + '_' +
                              datetime.now().strftime("%Y_%m_%d-%I_%M_%S") + '.json')
                run_data = {'train_data': self.train_results.data.to_json(),'test_data': self.test_results.data.to_json()}
                mean_critic_loss = self.train_results.data['critic_loss']
                mean_critic_loss = mean_critic_loss.drop(mean_critic_loss[mean_critic_loss == 0].index).values
                episode_reward = self.train_results.data[['episode','reward_0']].groupby(['episode']).sum().values +\
                                 self.train_results.data[['episode','reward_1']].groupby(['episode']).sum().values
                res = {}
                res['job_id'] = self.job_id
                res['critic_loss'] = mean_critic_loss[-1]
                res['mean_loss'] = mean_critic_loss[-100:].mean()
                res['reward'] = episode_reward[-1].item()
                res['mean_reward'] = episode_reward[-30:].mean()
                res['test_reward'] = self.test_results.data['reward_0'].iloc[-self.T_test:].sum()+\
                                     self.test_results.data['reward_1'].iloc[-self.T_test:
```

```
                                 ].sum()
            res['test_energy_use [kWh]'] =
            abs(self.test_results.data['grid_import'].iloc[-self.T_test:]).sum()/1e3*\
                                        self.parameter['input_data']['step_size']
            res['episodes'] = self.parameter['agent']['setting']['episodes']
            res['n_step'] = self.parameter['agent']['setting']['n_step']
            res['forecast_hours'] =
            self.parameter['agent']['setting']['forecast_hours']
            res['batch_size'] = self.parameter['agent']['replay_buffer']['batch_size']
            res['episodes_with_noise'] =
            self.parameter['agent']['setting']['episodes_with_noise']
            res['exploration_episodes'] =
            self.parameter['agent']['setting']['exploration_episodes']
            res['Agent'] = self.parameter['agent']['Agent']
            res['network'] = self.parameter['agent']['NN']['network_architecture']
            res['replay'] = self.parameter['agent']['replay_buffer']['Buffer']
            res['noise'] = self.parameter['agent']['setting']['noise_process']
            res['num_layer_1'] = self.parameter['agent']['NN']['layer_size_1']
            res['num_layer_2'] = self.parameter['agent']['NN']['layer_size_2']
            res['hidden_layers'] = self.parameter['agent']['NN']['hidden_layers']
            res['activation'] = self.parameter['agent']['NN']['activation']
            res['act_learn'] = self.parameter['agent']['NN']['act_learning_rate']
            res['crit_learn'] = self.parameter['agent']['NN']['crit_learning_rate']
            res['noise_scale'] = self.parameter['agent']['scaling']['noise_scale']
            res['discount_factor'] = self.parameter['agent']['NN']['discount_factor']
            res['demand_charge_scale'] =
            self.parameter['agent']['NN']['demand_charge_scale']
            res['value_function'] = self.parameter['agent']['flags']['valuefunction']
            res['duration'] = str(timedelta(seconds=time_end - time_start))
            res['resname'] = filename

            with open(filename,'a') as json_file:
                json_file.write(json.dumps(run_data))

            return res

        return self.train_results, self.test_results
```

Environment:

```python
import pandas as pd
import sys
import random
import numpy as np
from tf_agents.environments import py_environment
from tf_agents.specs import array_spec
from tf_agents.trajectories import time_step as ts

class Room(py_environment.PyEnvironment):
    '''
    Training environment of a thermal zone (room) for controls development and evaluation.
    '''
    def __init__(self, parameter):

        self.parameter = parameter
        self.tvis_to_shgc = np.poly1d([self.parameter['window']['coeff_b'],
                                       self.parameter['window']['coeff_a']])
        # initiate the array for the calculation of the demand charge with [grid_import,C_demand]
        self.demand_charge_period = np.zeros(shape=(1,2))
        self.max_energy_cost = self.parameter['constraints']['heat_max']/1e3 * \
                        max(parameter['tariff']['C_energy'].values())
        self.max_demand_cost = self.parameter['constraints']['heat_max']/1e3 * \
                        max(parameter['tariff']['C_demand'].values())

        self.reward = 0
        self.demand_costs_calc = np.zeros(shape=(1,2))
        if self.parameter['agent']['actions'] == 0:
            action_dim = 2
        else:
            action_dim = 1
        self._action_spec = array_spec.BoundedArraySpec(shape=(action_dim,),
            dtype=np.float64, minimum=-self.parameter['constraints']['cool_max'],
            maximum=self.parameter['constraints']['heat_max'], name='action')
        self._observation_spec = array_spec.BoundedArraySpec(shape=(1,),
            dtype=np.float64, name='observation')

    def action_spec(self):
        return self._action_spec

    def observation_spec(self):
        return self._observation_spec

    def get_info(self):
        '''function returns variables calculated in the environment'''
        return self.data.index, self.data.values

    def heat_balance(self, inputs, T_room):
        ''' heat balance model '''

        Q_thermal =  inputs['Q_int_th'] + inputs['Q_int_el'] + inputs['P_lights']
        if not self.parameter['zone']['control_hvac']:
            Q_thermal += inputs['Q_hvac']
        if self.parameter['model']['include_solargains']:
            Q_thermal += inputs['Q_solar']

            T_in = ((Q_thermal * self.parameter['input_data']['step_size'])+ \
                    inputs['T_out'] * 1/self.parameter['model']['param']['R1']+\
                    T_room * self.parameter['model']['param']['C1'] / 3600)/\
                    (1/self.parameter['model']['param']['R1'] + \
                    self.parameter['model']['param']['C1'] / 3600)
            Q_hvac = 0
        return T_in, Q_hvac

    def calc_illuminance(self, solar_Illumination, Tvis):
        '''calculation of the average illuminance in the room using the daylight factor from

```

```
            https://www.uk-ncm.org.uk/filelibrary/SBEM-Technical-Manual_v5.2.g_20Nov15.
            pdf'''
        return (solar_Illumination * Tvis * (45*self.parameter['zone']['area']) / \
                (self.parameter['zone']['surface_area']*0.76))/100

    def calculate_solar_power(self, inputs):
        '''calculate the solar power through the window with a simple calculation
        with the
           SHGC, or include the radiance calculation'''
        outputs = {}
        outputs['shgc'] = self.tvis_to_shgc(inputs['Tvis'])
        outputs['Q_solar'] = inputs['S_irr'] * self.parameter['window']['area'] * \
            outputs['shgc']
        tvis_to_state = pd.Series(self.parameter['somodel']['tvis_to_state'])
        outputs['tint'] =
        tvis_to_state.iloc[tvis_to_state.index.get_loc(inputs['Tvis'],
            method='nearest')]
        outputs['uWin'] = 1 - outputs['tint'] / tvis_to_state.max()
        if self.parameter['somodel']['use_radiance']:
            if not self.radiance:
                self.radiance = \

                    self.radiance_handler.Radiance(self.parameter['somodel']['config_f
                    ile'],
                    regenerate=self.parameter['somodel']['regenerate_matrices'],
                    orient=self.parameter['somodel']['orientation'],
                    location=self.parameter['somodel']['location'],
                    filestruct=self.parameter['somodel']['filestruct'],
                    use_gendaymtx=False)
            weather = pd.DataFrame({k:[v] for k,v in inputs.items() if
            k.startswith('wea') or k == 'start_time'})
            weather.index = [pd.to_datetime('2020-01-01') +
            pd.DateOffset(seconds=ix) for ix in weather['start_time']]

            # Rough approximation of solar heat gain
            outputs['Q_solar_radiance'] = sum(radiance_outputs[2:7]) +
            radiance_outputs[8] + 0 * radiance_outputs[7]
        else:
            outputs['daylight'] = self.calc_illuminance(inputs['S_ill'],
            inputs['Tvis']) # lux
        return outputs

    def calculate_lighting_power(self, inputs):
        '''calculate the necessary lighting power to meet the wpi-constraints'''
        if inputs['daylight'] > inputs['wpi_min']:
            P_light = 0
            Ill_light = 0
        else:
            P_light = (inputs['wpi_min'] - inputs['daylight']) *
            self.parameter['zone']['eff_lights']
            Ill_light = inputs['wpi_min'] - inputs['daylight']
        return P_light, Ill_light

    def calculate_demand_charge(self,inputs):
        '''calculate the demand charge for the n_step periods
           demand costs of TOU-tariff are calculated for every demand period in
           the n_step range. Average of the costs is taken for further calculation
           of the demand charge in this period
           '''
        demand_cost = 0
        self.demand_charge_period = np.append(self.demand_charge_period,\
            (np.array([[inputs['grid_import']/1e3,inputs['C_demand']]])),axis=0)
        if self.demand_charge_period.shape[0] >
        self.parameter['agent']['setting']['n_step']:
            # delete the first row in the array to have the latest steps with length
            (n_step)
            self.demand_charge_period = np.delete(self.demand_charge_period ,0,
            axis=0)
            # take the maximum grid_import of each unique period and calculate the
```

```
            resulting demand costs
            periods_power = np.split(self.demand_charge_period[:,0], np.sort(np.unique(self.demand_charge_period[:,1], return_index = True)[1]))[1:]
            periods_cost = np.unique(self.demand_charge_period[:,1])
            for i in range(periods_cost.shape[0]):
                demand_cost += np.max(periods_power[i]) * periods_cost[i]
            demand_cost /= periods_cost.shape[0]
            # add the base demand charge with the maximum grid_import of the latest steps with length (n_step)
            demand_cost += np.max(self.demand_charge_period[:,0]) * self.parameter['tariff']['C_demand']['base_rate']
        return demand_cost

    def _step(self, inputs):
        # Parse inputs
        data = pd.Series(inputs, index=self.parameter['inputs']['labels'])

        # Calculate solar gains and daylighting in room
        data = data.append(pd.Series(self.calculate_solar_power(data)))

        # Calculate lighting requirement in room
        data['P_lights'],data['Ill_light'] = self.calculate_lighting_power(data)

        # Calcualte heat balance
        data['T_now'], data['Q_hvac_env'] = self.heat_balance(data, self.state[0])
        self.state[0] = data['T_now']

        # Electricity balance
        if self.parameter['zone']['control_hvac']:
            data['Q_hvac'] = data['Q_hvac_env']
        data['P_hvac'] = (data['Q_hvac'] * self.parameter['zone']['eff_heat']) if data['Q_hvac'] > 0 else \
            (abs(data['Q_hvac']) * self.parameter['zone']['eff_cool'])
        data['grid_import'] = data['P_hvac'] + data['P_lights'] + data['Q_int_el']

       # Cost calculation
        data['energy_cost'] = data['grid_import'] * self.parameter['input_data']['step_size'] * data['C_energy'] / 1e3
        data['demand_cost'] = self.calculate_demand_charge(data)

        # Temperature constraint
        if data['T_now'] < data['t_min']:
            data['Penalty_T_room'] = min(abs(data['t_min'] - data['T_now']),
                                     self.parameter['constraints']['max_t_penalty'])
        elif data['T_now'] > data['t_max']:
            data['Penalty_T_room'] = min(abs(data['T_now'] - data['t_max']),
                                     self.parameter['constraints']['max_t_penalty'])
        else:
            data['Penalty_T_room'] = 0

         # penalty for tint status in the night
        if data['S_ill'] == 0 and data['Tvis'] < 0.59:
            data['Penalty_tint'] = self.parameter['constraints']['night_tint_penalty']
        else:
            data['Penalty_tint'] = 0

        data['reward_0'] = data['energy_cost']/self.max_energy_cost + (data['Penalty_T_room'] + data['Penalty_tint'])
        data['reward_1'] = data['demand_cost']/self.max_energy_cost/self.parameter['agent']['scaling']['reward_0']

        reward = np.array([data['reward_0']*-1, data['reward_1']*-1])

        self.data = data
        return ts.transition(self.state, reward = reward , discount = 0.0)

    def _reset(self):
```

```
        '''reset is called at the start of the episode'''
        if self.parameter['zone']['t_init']:
            self.state = np.array([self.parameter['zone']['t_init']])
        else:
            self.state = 
            np.array([random.uniform(self.parameter['zone']['t_init_min'],
                                                      self.parameter['zone']['t_init_max']
                                                      )])
        self.fmu_loaded = False
        self.demand_costs_calc = np.zeros(shape=(1,2))
        return ts.restart(self.state)
```

Actor Critic Network

```
from datetime import datetime
import numpy as np
import os
import pandas as pd
import sys
import tensorflow as tf
tf.compat.v1.logging.set_verbosity('ERROR')
import tensorflow.keras
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.layers import Dense, ReLU, LeakyReLU, Input, concatenate, 
BatchNormalization
from tensorflow.keras.layers import Conv1D, Flatten, Multiply, Add
from tensorflow.keras.layers import LSTM
from tensorflow.keras.losses import MeanSquaredError
from tensorflow.keras import Model
from tensorflow.keras.models import load_model
tf.keras.backend.set_floatx('float32')

from Noise import OU_Noise, Gauss_Noise, Param_Noise, add_OU_noise, add_Gauss_noise

class _actor_MLP():
    def __init__(self, state_dim, forecast_dim, action_bound_range, parameter):
        self.state_dim = state_dim
        self.action_bound_range = tf.constant(action_bound_range, shape=(1,), 
        dtype='float64')
        self.forecast_dim = forecast_dim
        self.num_ly1 = parameter['NN']['layer_size_1']
        self.num_ly2 = parameter['NN']['layer_size_2']
        self.hidden_layers = parameter['NN']['hidden_layers']
        self.normalize_a = tf.constant(0.6-0.01, shape=(1,), dtype='float64')
        self.normalize_b = tf.constant(0.01, shape=(1,), dtype='float64')
        if parameter['NN']['activation'] == 'relu':
            self.activation = ReLU
        elif parameter['NN']['activation'] == 'leakyrelu':
            self.activation = LeakyReLU

        self.action = parameter['actions']

```

```
    def model(self):

        state = Input(shape=(self.state_dim,), name='state_input')
        forecast = Input(shape=(self.forecast_dim,), name='forecast_input')

        sf = concatenate([state, forecast])

        sf = Dense(self.num_ly1, bias_initializer = 'zeros', name='param_noise')(sf)
        sf = self.activation()(sf)
        sf = BatchNormalization()(sf)
        for i in range(self.hidden_layers):
            sf = Dense(self.num_ly2, bias_initializer = 'zeros', name =
            'hidden'+str(i+1))(sf)
            sf = self.activation()(sf)
            sf = BatchNormalization()(sf)

        if self.action != 0:
            if self.action == 1:
                action = Dense(1, activation='tanh')(sf)
                action = Multiply(name = 'Q')([action, self.action_bound_range])
            elif self.action == 2:
                action = Dense(1)(sf)
                action = ReLU(max_value=1)(action)
                action = Multiply()([action, self.normalize_a])
                action = Add(name = 'Tvis')([action,self.normalize_b])
            return Model(inputs=[state, forecast], outputs=[action], name='actor')
        else:
            action1 = Dense(1, activation='tanh')(sf)
            action1 = Multiply(name = 'Q')([action1, self.action_bound_range])

            action2 = Dense(1)(sf)
            action2 = ReLU()(action2)
            action2 = Multiply()([action2, self.normalize_a])
            action2 = Add(name = 'Tvis')([action2,self.normalize_b])

            return Model(inputs=[state, forecast], outputs=[action1, action2],
            name='actor')

class _critic_MLP():
    def __init__(self, state_dim, forecast_dim, action_dim, n_Step, parameter):
        self.state_dim = state_dim
        self.action_dim = action_dim
        self.forecast_dim = forecast_dim
        self.num_ly1 = parameter['NN']['layer_size_1']
        self.num_ly2 = parameter['NN']['layer_size_2']
        self.hidden_layers = parameter['NN']['hidden_layers']
        if parameter['NN']['activation'] == 'relu':
            self.activation = ReLU
        elif parameter['NN']['activation'] == 'leakyrelu':
            self.activation = LeakyReLU

    def model(self):
        state = Input(shape=(self.state_dim,), name='state_input')
        forecast = Input(shape=(self.forecast_dim,), name='forecast_input')
        action = Input(shape=(self.action_dim,), name='action_input')

        sfa = concatenate([state, forecast,action])

        sfa = Dense(self.num_ly1, bias_initializer = 'zeros')(sfa)
        sfa = self.activation()(sfa)
        sfa = BatchNormalization()(sfa)

        for i in range(self.hidden_layers):
            sfa = Dense(self.num_ly2, bias_initializer = 'zeros')(sfa)
            sfa = self.activation()(sfa)
            sfa = BatchNormalization()(sfa)

        value = Dense(1, activation='linear', name = 'value')(sfa)
        return Model(inputs=[state, forecast, action], outputs=value, name='critic')
```

```
class _actor_LSTM():
    def __init__(self, state_dim, forecast_dim, action_bound_range, parameter):
        self.observation_dim = (None,forecast_dim+state_dim)
        self.action_bound_range = tf.constant(action_bound_range, shape=(1,),
        dtype='float64')
        self.num_ly1 = parameter['NN']['layer_size_1']
        self.num_ly2 = parameter['NN']['layer_size_2']
        self.hidden_layers = parameter['NN']['hidden_layers']
        self.normalize_a = tf.constant(0.6-0.01, shape=(1,), dtype='float64')
        self.normalize_b = tf.constant(0.01, shape=(1,), dtype='float64')
        if parameter['NN']['activation'] == 'relu':
            self.activation = ReLU
        elif parameter['NN']['activation'] == 'leakyrelu':
            self.activation = LeakyReLU

        self.action = parameter['actions']

    def model(self):
        observation = Input(shape=(self.observation_dim), name='state_input')

        x = Dense(256, bias_initializer = 'zeros', name='param_noise')(observation)
        x = self.activation()(x)
        x = BatchNormalization()(x)

        x = LSTM(256)(x)

        if self.action != 0:
            if self.action == 1:
                action = Dense(1, activation='tanh')(x)
                action = Multiply(name = 'Q')([action, self.action_bound_range])
            elif self.action == 2:
                action = Dense(1)(x)
                action = ReLU(max_value=1)(action)
                action = Multiply()([action, self.normalize_a])
                action = Add(name = 'Tvis')([action,self.normalize_b])
            return Model(inputs=observation, outputs=[action], name='actor')

        else:
            action1 = Dense(1, activation='tanh')(x)
            action1 = Multiply(name = 'Q')([action1, self.action_bound_range])

            action2 = Dense(1)(x)
            action2 = ReLU(max_value=1)(action2)
            action2 = Multiply()([action2, self.normalize_a])
            action2 = Add(name = 'Tvis')([action2,self.normalize_b])

            return Model(inputs=observation, outputs=[action1, action2],
            name='actor')

```

```
class _critic_LSTM():
    def __init__(self, state_dim, forecast_dim, action_dim, n_Step, parameter):
        self.observation_dim = (n_Step,forecast_dim+state_dim)
        self.action_dim = (n_Step,action_dim)
        self.num_ly1 = parameter['NN']['layer_size_1']
        self.num_ly2 = parameter['NN']['layer_size_2']
        self.hidden_layers = parameter['NN']['hidden_layers']
        if parameter['NN']['activation'] == 'relu':
            self.activation = ReLU
        elif parameter['NN']['activation'] == 'leakyrelu':
            self.activation = LeakyReLU

    def model(self):
        observation = Input(shape=(self.observation_dim), name='obs_input')
        action = Input(shape=(self.action_dim), name='action_input')

        observation_i = Dense(256)(observation)
        observation_i = self.activation()(observation_i)
        observation_i = BatchNormalization()(observation_i)

        action_i = Dense(64)(action)
        action_i = self.activation()(action_i)
        action_i = BatchNormalization()(action_i)

        x = concatenate([observation_i, action_i], name='concat')

        x = Dense(320)(x)
        x = self.activation()(x)
        x = BatchNormalization()(x)

        x = LSTM(320, return_sequences = True)(x)

        value = Dense(1, activation='linear',name = 'value')(x)

        return Model(inputs=[observation,action], outputs=value, name='critic')

class AC_network():
    def __init__(self, state_dim, forecast_dim, action_dim, action_bound_range, \
                 forecast_norm_layer, state_norm_layer, action_norm_layer, parameter):
        ''' setup and init Actor-Critic networks and'''
        self.parameter = parameter
        self.state_dim = state_dim
        self.forecast_dim = forecast_dim
        self.action_dim = action_dim
        self.n_Step = self.parameter['agent']['setting']['n_step']
        self.action_bound_range = action_bound_range

        self.norm_forecast_layer = forecast_norm_layer
        self.norm_state_layer = state_norm_layer
        self.norm_action_layer = action_norm_layer
```

```
        self.actor_network_types = {'MLP':_actor_MLP, 'CNN':_actor_CNN, 'LSTM':_actor_LSTM}
        self.critic_network_types = {'MLP':_critic_MLP, 'CNN':_critic_CNN, 'LSTM':_critic_LSTM}

        # set optimizer for network updates
        self.actor_opt = Adam(self.parameter['agent']['NN']['act_learning_rate'])
        self.critic_opt = Adam(self.parameter['agent']['NN']['crit_learning_rate'])
        self.loss_function = MeanSquaredError()

        self.lamb = 1. # initial weight for Q-value priority which changes over time to 0
        self.lamb_decrease = 1-(1/(self.parameter['agent']['setting']['episodes'] * int(parameter['agent']['setting']['training_days']* 24 / \
                          parameter['agent']['stepsize'])) * 6)

        self.AC_name = datetime.now().strftime("%Y_%m_%d_%I_%M_%S") + '-' + str(self.parameter['agent']['Agent'])+'_'\
                        + str(self.parameter['agent']['NN']['network_architecture'])+'_' \
                        + 'action' + str(self.parameter['agent']['actions'])+'_'\
                        + 'layer' + str(self.parameter['agent']['NN']['layer_size_1'])+'_'\
                        + str(self.parameter['agent']['NN']['layer_size_2'])+'_'\
                        + str(self.parameter['agent']['NN']['hidden_layers'])+'_'\
                        + str(self.parameter['agent']['NN']['activation'])+'_'\
                        + str(self.parameter['agent']['setting']['n_step'])+'_'\
                        + 'forecast' + str(self.parameter['agent']['setting']['forecast_hours'])+'_'\
                        + str(self.parameter['agent']['replay_buffer']['Buffer'])+'_'\
                        + 'batch' + str(self.parameter['agent']['replay_buffer']['batch_size'])+'_'\
                        + str(self.parameter['agent']['setting']['noise_process'])+'_'\
                        + 'valuefct' + str(self.parameter['agent']['flags']['valuefunction'])+'_'\
                        + 'demand_scale' + str(self.parameter['agent']['NN']['demand_charge_scale'])

        if os.path.exists('actor/' + self.AC_name + '.h5') == True and self.parameter['agent']['flags']['save_models'] == True:
            job = input('Do you want to overwrite the existing network? (y/n) ')
            if job == 'y':
                pass
            elif job == 'n':
                sys.exit()

    def build_actor(self, architecture):
        ''' build and save new actor network'''
        self.actor = self.actor_network_types[architecture]\
                (self.state_dim, self.forecast_dim, self.action_bound_range, self.parameter['agent']).model()
        self.actor_target = self.actor_network_types[architecture]\
                (self.state_dim, self.forecast_dim, self.action_bound_range, self.parameter['agent']).model()
        self.actor_target.set_weights(self.actor.get_weights())

        if self.parameter['agent']['setting']['noise_process'] == 'param_noise':
            self.perturbed_actor = self.actor_network_types[architecture]\
                (self.state_dim, self.forecast_dim, self.action_bound_range, self.parameter['agent']).model()
            self.perturbed_actor.set_weights(self.actor.get_weights())
            actor_weights = self.actor.get_layer(name='param_noise').get_weights()
            n_weights = actor_weights[0].size
            layer_shape = actor_weights[0].shape
```

```
            self.param_noise_process = Param_Noise(n_weights, layer_shape,
            self.action_dim)
        self.actor.compile(optimizer = self.actor_opt)

        if self.parameter['agent']['flags']['save_models']:
            self.actor.save('actor/' + self.AC_name + '.h5')
            del self.actor
            self.actor = load_model('actor/' + self.AC_name + '.h5')

    def build_critic(self, architecture):
        ''' bulid and save new critic network'''
        self.critic = self.critic_network_types[architecture]\
                (self.state_dim, self.forecast_dim, self.action_dim, self.n_Step,
                self.parameter['agent']).model()
        self.critic_target = self.critic_network_types[architecture]\
                (self.state_dim, self.forecast_dim, self.action_dim, self.n_Step,
                self.parameter['agent']).model()
        self.critic_target.set_weights(self.critic.get_weights())
        self.critic.compile(optimizer = self.critic_opt)

        if self.parameter['agent']['flags']['save_models']:
            self.critic.save('critic/' + self.AC_name + '.h5')
            del self.critic
            self.critic = load_model('critic/' + self.AC_name + '.h5')

    def parameter_noise_handling(self):
        ''' takes perturbed-actor NN initialized in this class and
        changes the weights by adding with gaussian noise '''
        actor_weights = self.actor.get_layer(name='param_noise').get_weights()
        noisy_weights = actor_weights.copy()
        noise = self.param_noise_process.perturb_actor()
        noisy_weights[0] = noisy_weights[0] + noise

        self.perturbed_actor.get_layer(name='param_noise').set_weights(noisy_weights)


    def action_noise_handler(self,noise_scale):
        ''' init noise process for current episode'''
        if self.parameter['agent']['setting']['noise_process'] == 'OU_noise':
            self.noise_Q = OU_Noise(mu = np.zeros(1), sigma = noise_scale[0], theta
            = 0.2, dt=1e-1)
            self.noise_Tvis = OU_Noise(mu = np.zeros(1), sigma = noise_scale[1],
            theta = 0.2, dt=1e-2)
        else:
            self.noise_Q = Gauss_Noise(noise_scale[0])
            self.noise_Tvis = Gauss_Noise(noise_scale[1])

    def save_model(self):
        ''' save actor and critic in directories with initiated newtwork name'''
        self.actor.save('actor/' + self.AC_name + '.h5')
        self.critic.save('critic/' + self.AC_name + '.h5')

    def load_model(self):
        ''' load pretrained model by giving the filename '''
        self.AC_name = input('filename:')
        try:
            self.actor = load_model('actor/' + self.AC_name +'.h5')
            self.actor_target = load_model('actor/' + self.AC_name +'.h5')
            self.critic = load_model('critic/' + self.AC_name +'.h5')
            self.critic_target = load_model('critic/' + self.AC_name +'.h5')
            print('actor-critic model successfully loaded')
        except:
            print('network not available')

            job = input('Network not available! continue with new network (y/n)? ')
            if job == 'y':
                pass
            else:
                sys.exit()
```

```
    def take_action(self, state_t, forecast_t, noise):
        ''' Input for selecting an action
                state_t...... current state (Room Temperature) (array(shape=1,1))
                forecast_t... forecast (array(shape=forecast_dim))
                noise........ flag indicating if noise should be added during exploration (boolean)
            output
                actn......... seleted action (Energy Input, tint state) (array(shape=1,2))'''
        # normalization for NN input
        state = self.norm_state_layer(state_t)
        forecast = self.norm_forecast_layer(forecast_t)

        if self.parameter['agent']['NN']['network_architecture'] == 'MLP':
            forecast = tf.reshape(forecast,(1,self.forecast_dim))
        if self.parameter['agent']['setting']['noise_process'] == 'param_noise' and noise:
            action = self.perturbed_actor([state, forecast])
        else:
            action = self.actor([state, forecast])

        if self.parameter['agent']['actions'] == 0:
            Q = action[0][0]
            Tvis = action[1][0]
        elif self.parameter['agent']['actions'] == 1:
            Q = action[0]
            Tvis = 0.6
        else:
            Q = 0
            Tvis = action[0]

        if noise:
            if self.parameter['agent']['setting']['noise_process'] == 'OU_noise':
                Q, Tvis = add_OU_noise(self.noise_Q, Q, self.noise_Tvis, Tvis)
            elif self.parameter['agent']['setting']['noise_process'] == 'Gauss_noise':
                Q, Tvis = add_Gauss_noise(self.noise_Q, Q, self.noise_Tvis, Tvis)
        # clip the values after adding the action noise to min and max values
        if self.parameter['agent']['actions'] == 0:
            Q = tf.clip_by_value(action[0], -self.action_bound_range, self.action_bound_range)[0].numpy()
            Tvis = tf.clip_by_value(action[1],0.01,0.6)[0].numpy()
            actn = np.array([Q,Tvis])
        elif self.parameter['agent']['actions'] == 1:
            Q = tf.clip_by_value(action[0], -self.action_bound_range, self.action_bound_range)[0].numpy()
            actn = np.array([Q])
        elif self.parameter['agent']['actions'] == 2:
            Tvis = tf.clip_by_value(action[0],0.01,0.6)[0].numpy()
            actn = np.array([Tvis])
        actn = actn.reshape(1,self.action_dim)
        return actn

    def take_action_lstm(self, state_history_t, forecast_history_t, noise):
        ''' Input for selecting an action
                state_history_t...... state history (Room Temperature) (array(shape=n_step,1,1))
                forecast_history_t... forecast history (array(shape=n_step,forecast_dim))
                noise................ flag indicating if noise should be added during exploration (boolean)
            output
                actn................. seleted action (Energy Input, tint state) (array(shape=1,2))'''
        # normalization for NN input
        state_history = self.norm_state_layer(state_history_t)
        forecast_history = self.norm_forecast_layer(forecast_history_t)

        history =
```

```python
        np.concatenate((state_history,tf.reshape(forecast_history,(state_history.shape[0],self.forecast_dim))),axis=1)\
                    .reshape(1,state_history.shape[0],self.forecast_dim+self.state_dim)

        if self.parameter['agent']['setting']['noise_process'] == 'param_noise' and noise:
            action = self.perturbed_actor(history)
        else:
            action = self.actor(history)

        if self.parameter['agent']['actions'] == 0:
            Q = action[0][0]
            Tvis = action[1][0]
        elif self.parameter['agent']['actions'] == 1:
            Q = action[0]
            Tvis = 0.6
        else:
            Q = 0
            Tvis = action[0]

        if noise:
            if self.parameter['agent']['setting']['noise_process'] == 'OU_noise':
                Q, Tvis = add_OU_noise(self.noise_Q, Q, self.noise_Tvis, Tvis)
            elif self.parameter['agent']['setting']['noise_process'] == 'Gauss_noise':
                Q, Tvis = add_Gauss_noise(self.noise_Q, Q, self.noise_Tvis, Tvis)
        # clip the values after adding the action noise to min and max values
        if self.parameter['agent']['actions'] == 0:
            Q = tf.clip_by_value(action[0], -self.action_bound_range, self.action_bound_range)[0].numpy()
            Tvis = tf.clip_by_value(action[1],0.01,0.6)[0].numpy()
            actn = np.array([Q,Tvis])
        elif self.parameter['agent']['actions'] == 1:
            Q = tf.clip_by_value(action[0], -self.action_bound_range, self.action_bound_range)[0].numpy()
            actn = np.array([Q])
        elif self.parameter['agent']['actions'] == 2:
            Tvis = tf.clip_by_value(action[0],0.01,0.6)[0].numpy()
            actn = np.array([Tvis])
        actn = actn.reshape(1,self.action_dim)
        return actn

    def train_critic_network(self,num_samples, states_batch, forecast_batch, actions_batch, rewards_batch, demand_batch, next_states_batch, next_forecast_batch, indices, importance_weight):
        ''' off-policy training of the critic network with stored experience
            DDPG by (Lillicrap et.al.)

            after training update the priority of the sampled expirience '''

        if self.parameter['agent']['flags']['valuefunction']:
            target_actions = self.actor_target([next_states_batch[:,self.parameter['agent']['setting']['n_step']-1], next_forecast_batch])
            target_actions = np.dstack(target_actions).reshape(num_samples,self.action_dim)
            target_actions = self.norm_action_layer(target_actions)
            target_critic_value = self.critic_target([next_states_batch[:,self.parameter['agent']['setting']['n_step']-1], next_forecast_batch, target_actions])

            y_i = rewards_batch + demand_batch
            y_i = np.reshape(y_i,(num_samples,1))
            for i in range(num_samples):
                y_i[i] = y_i[i] + self.parameter['agent']['NN']['discount_factor']**self.parameter['agent']['setting']['n_step'] * target_critic_value[i]
            with tf.GradientTape(watch_accessed_variables=False) as tape:
```

```python
                tape.watch(self.critic.trainable_variables)
                critic_value = self.critic([states_batch[:,0], forecast_batch, actions_batch])
                critic_loss = self.loss_function(y_i,critic_value,sample_weight=importance_weight)
            critic_grad = tape.gradient(critic_loss, self.critic.trainable_variables)
            self.critic_opt.apply_gradients(zip(critic_grad, self.critic.trainable_variables))
        else:
            y_i = rewards_batch + demand_batch
            y_i = tf.reshape(y_i,(num_samples,1))
            with tf.GradientTape(watch_accessed_variables=False) as tape:
                tape.watch(self.critic.trainable_variables)
                critic_value = self.critic([states_batch[:,0], forecast_batch, actions_batch])
                critic_loss = self.loss_function(y_i,critic_value,sample_weight=importance_weight)
            critic_grad = tape.gradient(critic_loss, self.critic.trainable_variables)
            self.critic_opt.apply_gradients(zip(critic_grad, self.critic.trainable_variables))

        if self.parameter['agent']['replay_buffer']['Buffer'] == 'PER':
            priority = tf.abs(y_i - self.critic([states_batch[:,0], forecast_batch, actions_batch])[0])
        elif self.parameter['agent']['replay_buffer']['Buffer'] == 'HVPER':
            td_priority = tf.math.sigmoid(np.abs(y_i - self.critic([states_batch[:,0], forecast_batch, actions_batch])[0])) * 2 -1
            q_priority = tf.cast(tf.math.sigmoid(y_i),dtype=tf.float32)
            self.lamb = max(self.lamb * self.lamb_decrease,0)
            priority = self.lamb * q_priority + (1-self.lamb) * td_priority
        else:
            priority = 0

        if self.parameter['agent']['flags']['print'] :
            print('critic_loss:\t', critic_loss.numpy())

        return pd.Series(critic_loss.numpy()), priority

    def train_actor_network(self,num_samples, states_batch, forecast_batch):
        '''the loss function for the actor is the  negative value function
        (critic network) as we want to maximize this value'''

        with tf.GradientTape(watch_accessed_variables=False) as tape:
            tape.watch(self.actor.trainable_variables)
            actions = self.actor([states_batch[:,0], forecast_batch])
            actions = tf.reshape(tf.concat(actions, -1), (num_samples, self.action_dim))
            actions = self.norm_action_layer(actions)
            critic_value = self.critic([states_batch[:,0], forecast_batch, actions])
            actor_loss = -tf.math.reduce_mean(critic_value)
        actor_grad = tape.gradient(actor_loss, self.actor.trainable_variables)
        self.actor_opt.apply_gradients(zip(actor_grad, self.actor.trainable_variables))

        if self.parameter['agent']['flags']['print']:
            print('actor_loss:\t', actor_loss.numpy())

        self.update_target(self.critic_target, self.critic, self.parameter['agent']['NN']['tow'])
        self.update_target(self.actor_target, self.actor, self.parameter['agent']['NN']['tow'])

    def train_critic_lstm(self,num_samples, hist_t, actions_batch, rewards_batch, hist_t_1, indices, importance_weight):
        ''' off-policy training of the critic network with stored experience
            RDPG by (Hess et.al.)
```

```python

            after training update the priority of the sampled expirience '''

        target_actions = self.actor_target([hist_t_1])
        target_actions = tf.reshape(tf.stack(target_actions,axis=self.action_dim),(num_samples,1,self.action_dim))
        target_actions = self.norm_action_layer(target_actions)
        target_actions = tf.concat((actions_batch[:,1:,:],target_actions),axis=1)
        target_critic_value = self.critic_target([hist_t_1, target_actions])

        y_i = rewards_batch
        for i in range(num_samples):
            y_i[i] = y_i[i] + self.parameter['agent']['NN']['discount_factor'] * target_critic_value[i]

        with tf.GradientTape(watch_accessed_variables=False) as tape:
            tape.watch(self.critic.trainable_variables)
            critic_value = self.critic([hist_t, actions_batch])
            critic_loss = self.loss_function(y_i,critic_value,sample_weight=importance_weight)
        critic_grad = tape.gradient(critic_loss, self.critic.trainable_variables)
        self.critic_opt.apply_gradients(zip(critic_grad, self.critic.trainable_variables))

        if self.parameter['agent']['replay_buffer']['Buffer'] == 'PER':
            priority = np.abs(y_i - self.critic([hist_t, actions_batch])[0])
        elif self.parameter['agent']['replay_buffer']['Buffer'] == 'HVPER':
            td_priority = tf.math.sigmoid(np.abs(y_i - self.critic([hist_t, actions_batch])[0])) * 2 -1
            q_priority = tf.math.sigmoid(critic_value)
            self.lamb = max(self.lamb * self.lamb_decrease,0)
            priority = self.lamb * q_priority + (1-self.lamb) * td_priority
        else:
            priority = 0

        if self.parameter['agent']['flags']['print']:
            print('critic_loss:\t', critic_loss.numpy())

        return pd.Series(critic_loss.numpy()), priority

    def train_actor_lstm(self,num_samples, hist_t, actions_batch):
        '''the loss function for the actor is the  negative value function
        (critic network) as we want to maximize this value'''

        with tf.GradientTape(watch_accessed_variables=False) as tape:
            tape.watch(self.actor.trainable_variables)
            actions = tf.reshape(tf.stack(self.actor(hist_t),axis=self.action_dim),(num_samples,1,self.action_dim))
            actions = self.norm_action_layer(actions)
            actions = tf.concat((actions_batch[:,0:self.n_Step-1],actions),axis=1)
            critic_value = self.critic([hist_t, actions])
            actor_loss = -tf.math.reduce_mean(critic_value)
        actor_grad = tape.gradient(actor_loss, self.actor.trainable_variables)
        self.actor_opt.apply_gradients(zip(actor_grad, self.actor.trainable_variables))

        if self.parameter['agent']['flags']['print']:
            print('actor_loss:\t', actor_loss.numpy())

        self.update_target(self.critic_target, self.critic, self.parameter['agent']['NN']['tow'])
        self.update_target(self.actor_target, self.actor, self.parameter['agent']['NN']['tow'])

    def update_target(self, target, online, tow):
        ''' soft update of target networks'''
```

```
        init_weights = online.get_weights()
        update_weights = target.get_weights()
        weights = []
        for i in tf.range(len(init_weights)):
            weights.append(tow * init_weights[i] + (1 - tow) * update_weights[i])
        target.set_weights(weights)
        return target
```

Replay Buffer

```
from collections import deque
import numpy as np
import random
from tensorflow import math

class Uniform:
    def __init__(self, max_buffer_size, dflt_dtype, forecast_dim):
        self.buffer = deque(maxlen=max_buffer_size)
        self.dflt_dtype = dflt_dtype
        self.forecast_dim = forecast_dim

    def add_experience(self, state, forecast, action, reward, demand, next_state,
    next_forecast):
        self.buffer.append([state, forecast, action, reward, demand, next_state,
        next_forecast])

    def batch_update(self, indices, priorities):
        pass

    def sample_batch(self, batch_size):
        num_samples = min(len(self.buffer),batch_size)
        replay_buffer = np.array(random.sample(self.buffer, num_samples))
        arr = np.array(replay_buffer)
        states_batch = np.stack(arr[:, 0])
        forecast_batch = np.stack(arr[:, 1])
        actions_batch = np.stack(arr[:, 2])
        rewards_batch = np.stack(arr[:, 3])
        demand_batch = np.stack(arr[:, 4])
        next_states_batch = np.stack(arr[:, 5])
        next_forecast_batch = np.stack(arr[:, 6])

        return states_batch, forecast_batch, actions_batch, rewards_batch,
        demand_batch, next_states_batch, next_forecast_batch
```

```
class PER:
    def __init__(self, max_buffer_size, dflt_dtype, forecast_dim):
        self.max_buffer_size = max_buffer_size
        self.buffer = deque(maxlen=max_buffer_size)
        self.priorities = deque(maxlen=max_buffer_size)
        self.indexes = deque(maxlen=max_buffer_size)
        self.dflt_dtype = dflt_dtype
        self.forecast_dim = forecast_dim
        self.absolute_error_upper = 10.
        self.alpha = 0.7

    def add_experience(self, state, forecast, action, reward, demand, next_state, next_forecast):
        self.buffer.append([state, forecast, action, reward, demand, next_state, next_forecast])
        if self.priorities:
            max_priority = np.max(self.priorities)
        else:
            max_priority = self.absolute_error_upper
        if max_priority == 0:
            max_priority = self.absolute_error_upper

        self.priorities.append(max_priority)

        ln = len(self.buffer)
        if ln < self.max_buffer_size : self.indexes.append(ln)

    def batch_update(self,indices, priorities):
        clipped_errors = np.minimum(priorities, self.absolute_error_upper)
        ps = math.pow(clipped_errors, self.alpha)
        for indx, priority in zip(indices, ps):
            self.priorities[indx-1] = priority

    def sample_batch(self, batch_size, beta):
        num_samples = min(len(self.buffer),batch_size)
        indices = random.choices(self.indexes,weights=self.priorities, k = num_samples)

        importance_weight = np.array([(self.priorities[indx-1] * num_samples)**-beta for indx in indices])
        importance_weight = importance_weight / max(importance_weight)

        replay_buffer = [self.buffer[indx-1] for indx in indices]
        arr = np.array(replay_buffer)
        states_batch = np.stack(arr[:, 0])
        forecast_batch = np.stack(arr[:, 1])
        actions_batch = np.stack(arr[:, 2])
        rewards_batch = np.stack(arr[:, 3])
        demand_batch = np.stack(arr[:, 4])
        next_states_batch = np.stack(arr[:, 5])
        next_forecast_batch = np.stack(arr[:, 6])

        return states_batch, forecast_batch, actions_batch, rewards_batch, demand_batch, next_states_batch, next_forecast_batch, indices, importance_weight
```

```python
class HVPER:
    def __init__(self, max_buffer_size, dflt_dtype, forecast_dim):
        self.max_buffer_size = max_buffer_size
        self.buffer = deque(maxlen=max_buffer_size)
        self.priorities = deque(maxlen=max_buffer_size)
        self.usage = deque(maxlen=max_buffer_size)
        self.indexes = deque(maxlen=max_buffer_size)
        self.dflt_dtype = dflt_dtype
        self.forecast_dim = forecast_dim
        self.n_k = 5

    def add_experience(self, state, forecast, action, reward, demand, next_state, next_forecast):
        self.buffer.append([state, forecast, action, reward, demand, next_state, next_forecast])
        self.priorities.append(1)
        self.usage.append(1)
        ln = len(self.buffer)
        if ln < self.max_buffer_size : self.indexes.append(ln)

    def batch_update(self,indices, priorities):
        for indx, priority in zip(indices, priorities):
            self.usage[indx-1] += 1
            self.priorities[indx-1] = priority * (1*math.pow(0.95,self.usage[indx-1]))

    def sample_batch(self, batch_size, beta):
        num_samples = min(len(self.buffer),batch_size*self.n_k)
        n_k = min(self.n_k, int(num_samples/batch_size))
        if n_k == 0:
            n_k = 1
        num_samples = min(len(self.buffer),batch_size)
        nk_indices = random.sample(self.indexes,k = num_samples * n_k)
        nk_priorities = np.array([self.priorities[indx-1] for indx in nk_indices])
        indices = random.choices(nk_indices,weights=nk_priorities, k = num_samples)

        importance_weight = np.array([(self.priorities[indx-1] * num_samples)**-beta for indx in indices])
        importance_weight = importance_weight / max(importance_weight)

        replay_buffer = [self.buffer[indx-1] for indx in indices]
        arr = np.array(replay_buffer)
        states_batch = np.stack(arr[:, 0])
        forecast_batch = np.stack(arr[:, 1])
        actions_batch = np.stack(arr[:, 2])
        rewards_batch = np.stack(arr[:, 3])
        demand_batch = np.stack(arr[:, 4])
        next_states_batch = np.stack(arr[:, 5])
        next_forecast_batch = np.stack(arr[:, 6])

        return states_batch, forecast_batch, actions_batch, rewards_batch, demand_batch, next_states_batch, next_forecast_batch, indices, importance_weight
```

Noise

```python
import numpy as np
import tensorflow as tf
import tensorflow_probability as tfp

class OU_Noise(object):
    '''OU noise process'''
    def __init__(self, mu, sigma = 0.15, theta = 0.2, dt=1e-1, x0 = None):
        self.theta = theta
        self.mu = mu
        self.dt = dt
        self.sigma = sigma
        self.x0 = x0
        self.reset()

    def __call__(self):
        x = self.x_prev + self.theta*(self.mu-self.x_prev)*self.dt +
        self.sigma*np.sqrt(self.dt)*np.random.normal(size=self.mu.shape)
        self.x_prev = x
        return x

    def reset(self):
        self.x_prev = self.x0 if self.x0 is not None else np.zeros_like(self.mu)

class Param_Noise(): #Gaussian Noise θe = θ + N (0, σ**2 I) from
https://arxiv.org/pdf/1706.01905.pdf
    ''' Parameter noise as gaussian normal distribution
        Input is the shape of the layer weights of the NN
        Output is a gaussian distribution in the shape of the weights'''
    def __init__(self, size, shape, action_dim):
        self.shape = shape
        self.scale = 0.6
        self.size = size
        self.action_dim = action_dim
        self.alpha = 1.01

    def calc_scale(self, distance):
        if distance < self.scale:
            self.scale = self.alpha * self.scale
        else:
            self.scale = 1/self.alpha * self.scale

    def perturb_actor(self):
        paramnoise = np.random.normal(loc = 0, scale = self.scale, size = self.size)
        paramnoise = paramnoise.reshape(self.shape)
        return paramnoise

def Gauss_Noise(scale): #Gaussian Noise θe = θ + N (0, σ**2 I) from
https://arxiv.org/pdf/1706.01905.pdf
    ''' action noise as gaussian normal distribution
        Output is a gaussian distribution in the shape of the weights'''
    gaussnoise = tfp.distributions.Normal(loc = 0, scale = scale)
    return gaussnoise

def add_OU_noise(noise_Q, Q, noise_Tvis, Tvis):
    ''' Input:  Q ...... Energy Input selected by actor
                Tvis ... facade/tvis selected by actor
                noise .. current initiated noise
        Output: Q ...... Energy Input + noise
                Tvis ... facade/tvis + noise'''
    noise_Q = noise_Q()
    noise_Tvis = noise_Tvis()
    Q = Q + noise_Q
    Tvis = Tvis + noise_Tvis
    return Q, Tvis

def add_Gauss_noise(noise_Q, Q, noise_Tvis, Tvis):
    ''' Input:  Q ...... Energy Input selected by actor
                Tvis ... facade/tvis selected by actor
                noise .. current initiated noise
```

```
            Output: Q ...... Energy Input + noise
                    Tvis ... facade/tvis + noise'''
        noise_Q = tf.cast(noise_Q.sample(),dtype = 'float32')
        noise_Tvis = tf.cast(noise_Tvis.sample(),dtype='float32')
        Q = Q + noise_Q
        Tvis = Tvis + noise_Tvis
        return Q, Tvis
```

Input calculations

```python
import copy
import numpy as np
import os
import pandas as pd

def calc_tilted_surface(data, parameter, ground_reflection = 0.2):
    '''calculate the irradiance and illuminance on the tilted window with the weather data'''
    params = copy.deepcopy(parameter)
    params['location']['longitude'] *=-1
    params['location']['timezone'] *=-1
    calc = pd.DataFrame(index=data.index)

    window_tilt = np.radians(90)
    window_orientation = np.radians(params['orientation'])
    sin_lat = np.sin(np.radians(params['location']['latitude']))
    cos_lat = np.cos(np.radians(params['location']['latitude']))

    calc['declination'] = 23.45*(np.sin(np.radians(360/365*(284+data.index.dayofyear.values))))
    calc['sin_dec'] = np.sin(np.radians(calc['declination']))
    calc['cos_dec'] = np.cos(np.radians(calc['declination']))

    calc['B'] = np.radians(360/365*(data.index.dayofyear.values-1))
    calc['E'] = (229.18*(0.000075+0.001868*np.cos(calc['B'])-0.032077*np.sin(calc['B'])-0.014615*
                 np.cos(2*calc['B'])-0.04089*np.sin(2*calc['B'])))
    calc['solar_time'] = ((data.index.hour.values)*60 + data.index.minute.values)/60 + (params['location']['timezone']-
                               params['location']['longitude'])/15 + calc['E']/60

    calc['omega'] = np.radians((calc['solar_time']-12)*15)
    calc['cos_omega'] = np.cos(calc['omega'])
    calc.loc[calc['omega'] < 0, 'sign_omega'] = -1
    calc.loc[calc['omega'] >= 0, 'sign_omega'] = 1

    calc['cos_tetaz'] = calc['cos_dec'] *calc['cos_omega']*cos_lat + calc['sin_dec']*sin_lat
    calc['tetaz'] = np.arccos(calc['cos_tetaz'])
    calc['sin_tetaz'] = np.sin(calc['tetaz'])

    calc['gamma_s'] = ((calc['cos_tetaz']*sin_lat-calc['sin_dec'])/calc['sin_tetaz']/cos_lat).astype('float32')
    calc['gamma_s'] = np.arccos(calc['gamma_s'])*calc['sign_omega']

    calc['cos_teta_1'] = (calc['cos_tetaz']*np.cos(window_tilt)+calc['sin_tetaz']*np.sin(window_tilt)*
                  np.cos(calc['gamma_s']-window_orientation))

    calc['Rb'] = calc['cos_teta_1']*calc['cos_teta_1'].ge(0)

    calc['S_irr'] = (data['weaHDirNor'] * calc['Rb'] + data['weaHDifHor'] * ((1+np.cos(window_tilt))/2) +
                data['weaHGloHor'] * ground_reflection * ((1-np.cos(window_tilt))/2))
    calc['S_ill'] = (data['Direct normal illuminance in lux during minutes preceding the indicated time']
                         * calc['Rb'] + data['Diffuse horizontal illuminance in lux  during minutes preceding the indicated time'] * ((1+np.cos(window_tilt))/2) +
                data['Averaged global horizontal illuminance in lux during minutes preceding the indicated time'] * ground_reflection * ((1-np.cos(window_tilt))/2))
    return calc[['S_irr', 'S_ill']]
```

```python
def set_tariff(index, parameter):
    ''' set the TOU costs for energy and demand for every timestep in the input/weather data file'''

    tariff_map = pd.DataFrame(index = index)
    for k,v in parameter['dayofweek'].items():
        tariff_map.loc[(tariff_map.index.weekday.isin(v)),'dayofweek'] = k
    for k,v in parameter['periods'].items():
        tariff_map.loc[(tariff_map.index.month.isin(v['month'])),'season'] = k
        for k1,v1 in sorted(v.items())[1:]:
            for i in range(len(v1)):
                tariff_map.loc[(tariff_map['season'] == k) & (tariff_map['dayofweek']=='weekday')&((tariff_map.index.hour + tariff_map.index.minute/60) >= v1[i][0])&\
                            ((tariff_map.index.hour + tariff_map.index.minute/60) < v1[i][1]),'TOU_period'] = k1
                tariff_map.loc[(tariff_map['season'] == k) & (tariff_map['dayofweek'] == 'weekend') & ('off' in k1),'TOU_period'] = k1

    tariff_map['C_energy'] = tariff_map['TOU_period'].replace(parameter['C_energy'])
    tariff_map['C_demand'] = tariff_map['TOU_period'].replace(parameter['C_demand'])

    return tariff_map[['TOU_period', 'C_energy','C_demand']]

def set_internal_loads(index, parameter):
    ''' set the electric and thermal internal load according to the schedule set in
        paremter_handler.py '''
    internal_loads = pd.DataFrame(index = index)
    for v in parameter['zone']['office_hours']:

        internal_loads.loc[(internal_loads.index.weekday.isin(parameter['tariff']['dayofweek']['weekday']))&\
                            ((internal_loads.index.hour + internal_loads.index.minute/60) >= v[0])&\
                             ((internal_loads.index.hour + internal_loads.index.minute/60) < v[1]),'occupancy'] = 1
    internal_loads.loc[(internal_loads['occupancy'].isna() == True),'occupancy'] = 0 # not occupied

    internal_loads['internal_th_loads'] = internal_loads['occupancy'] * parameter['zone']['int_th_load']
    internal_loads['internal_el_loads'] = internal_loads['occupancy'] * parameter['zone']['int_el_load']
    internal_loads.loc[internal_loads['internal_el_loads'] == 0, 'internal_el_loads'] = 0.1 * internal_loads['internal_el_loads'].max()

    return internal_loads[['occupancy','internal_th_loads','internal_el_loads']]

def get_weather_files(parameter):
    ''' search the directory for files matching the string input and add to the path'''
    weather_parameter = copy.deepcopy(parameter)

    weather_files = list()
    weather_path = list()
    for file in os.listdir(weather_parameter['weather_dir']):
        if file.endswith('.mos'):
            weather_files.append(weather_parameter['weather_dir']+file)

    if 'rand' in weather_parameter['location']:
        return weather_files
    else:
        for file in weather_files:
            for loc in weather_parameter['location']:
                if loc in file :
                    weather_path.append(file)
        return weather_path
```