Effective House Energy Management Using Reinforcement Learning Technical Documentation

Dawid Czarneta, Jakub Frąckiewicz, Filip Olszewski, Michał Popiel, Julia Szulc

June 4, 2018

Abstract

This is a technical documentation of our university project developed with cooperation and guidance of Rafał Pilarczyk from SAMSUNG. Document contains a short introduction and overview of main concepts, milestones, goals and results of this project. It is followed by a technical part, that contains a setup guide, code and tests overviews, explanation of used algorithms and concepts. Last part provides details about the development process, including useful insights, experiences, further research suggestions and lessons we have learned the hard way.

1 Introduction

The idea of the project is to examine the potential of artificial intelligence in decision-making processes. This potential is based on the growing popularity of systems using Reinforcement Learning as well as successes achieved by them.

1.1 Goals and milestones

We wanted to create a system for a smart house that works properly under any base requirements and conditions and results in visible energy saving. Here we think about the smart house as an agent with the access to the basic devices in the house (currently it is: energy source, light level, window blinds, air conditioning and heater) as well as the inside and outside sensors. The sensors collect the data (temperature, brightness and solar battery level) between the fixed timeframes.

The project milestones was:

- Research activities - the stage consisting in learning about the problem and methods and algorithms solving the problem, theoretical familiarization with the libraries and tools used in the next stages of system creation.
- Planning - the stage in which the concept of the system was created and the initial schedule.
- Implementation of solutions - algorithms selected during planning has been implemented at this stage.
- Improvement of the system operation - through the appropriate selection of parameters, the system has been improved in order to achieve the best results.
- System performance tests - the system has been tested to see if it behaves in the expected manner.

1.2 Results

The project implements the idea of reinforcement learning for decision process using custom learning agent and environment. We created a working system for a smart house that analyzes the outside and inside data collected by sensors and determines the action to perform considering the user desired values of temperature and light and energy cost. As a result the system minimize energy consumption in the house and maximize user's comfort.

Functionalities of the system:

- The system retrieves information collected by external sensors
- The system retrieves information collected by internal sensors
- The system dynamically selects the energy source used by the house
- The system regulates the work of air conditioning
- The system regulates the heating
- The system controls the level of the blind
- The system controls the lighting level

2 Reinforcement Learning

The project is based on one of the machine learning areas called reinforcement learning. The research and development of this field has been rising as of late, because of, among others, the improvements in deep learning methods and constantly increasing computing power that is available to human. Combined with concepts like convolutional neural networks, RL algorithms achieve super-human performance in a variety of tasks. The main field of success is probably the gaming world, including both the traditional games like Chess or Go, and computer games, where, for example, classic Atari games are learned and solved by RL agents taking the input directly from raw pixels.

Reinforcement learning is often explained by comparison to supervised learning, which is usually named the most successful field of Machine Learning. It is based on datasets, which consist of both the observations and the correct labels (answers, classes). For example, the dataset consists of images, and each image has it is correct label as well - a word that describes the object on the image. In Reinforcement Learning, on the other hand, there is no 'supervisor'. Instead, the environment returns a reward signal, which indicates how good or bad the current situation is. The main goal of the agent is to maximize the cumulative reward.

There is a common vocabulary that is used to describe this setup. The **agent** (RL algorithm) has to interact with the **environment**, by performing **actions**. Environment gives the agent information about the **current state**, as well as the **reward signal** for the last time-frame. Agent has to choose the best one from the given set of possible actions. This is usually happening until the environment reaches a **terminal state**. Simulation from the start to terminal state is called an **episode**. A **transition** is defined as a sequence - state, action, reward and next state - and is used as a single 'experience' unit in agent's memory.

To put this into perspective: In our project, the smart house manager is the agent. The simulated world is the environment. Actions are used to control devices such as heating or light. The state is a collection of information about the weather, inside parameters, current devices settings and user requests - everything that is needed for agent to perform rational decisions. The episode can be defined as - for example - one single day, where the terminal state is when the world clock reaches midnight.

2.1 What have we used?

Our agent uses Double DQN with Prioritized Experience Replay. This is a combination of recent developments in the field, which improve the speed, quality and stability of learning, and sometimes are essential for solving particular tasks. The neural network model is a fully connected feedforward network with one hidden layer. It uses ReLU activation function on hidden layer neurons and linear activation function on the output layer. For an explanation of these concepts and more detailed information about the agent, please see the Agent module section (4.2).

2.2 PyTorch Framework

To implement the agent and the concepts named above, we have used the PyTorch framework. It is now commonly believed, that this frameworks becomes the go-to Python framework used for Deep Learning, with some essential advantages when compared to the - currently most popular - TensorFlow framework. While this is still a matter of loud discussions, we have chosen the framework more out of curiosity, and because it is often called 'more Pythonic'. We will talk more about our experiences and conclusions about that choice later in the documentation, but the short story is - we are satisfied with it.

We are using PyTorch for a very clear neural network model declaration, ready-to-use optimizers like SGD or Adagrad and loss functions, automatic computation of the gradients of network's parameters, and to perform the Double DQN learning process. It also provides a clear way to use GPUs for the computations.

We have used the version 0.3.1 and it is important to note, that the newest version of the framework (0.4.0 on the day of this documentation's publication) requires some changes to the code.

3 Getting Started

3.1 Setup and Requirements

This project was made on Debian based Linux distributions. We strongly recommend it in similar environment.

Firstly, install Python 3.6.5. You can do it via apt package manager from command line

```
$ sudo apt–get install python3.6
```

or by downloading it from producent's page [1] and performing manual installation.

Secondly, go to main project directory and install required Python libraries:

```
$ pip3 install –r requirements.txt
```

From now on, you should be able to use our project by running different source scripts.

3.2 Usage: Learning and Simulation

Our project can be run in different modes. These can be configured by changing contents of **configuration.json** file, located in main project directory. This directory will be referred to as **TOP** in this section.

3.2.1 Learning mode

This mode can be run by going into source files directory and launching **main.py** with python3 interpreter.

```
$ cd TOP/src
$ python3 main.py
```

[1] `https://www.python.org/downloads/release/python-365/`

In learning mode, Reinforcement Learning Agent model is learning for a given number of episodes. One episode is typically one day in randomly generated environment. Agent is performing action every few minutes, trying to get perfect balance between required conditions (light, temperature) and energy cost; he is also drawing conslusions from his mistakes. As the time passes, Agent is getting better; we can roughly say that in the long term, a longer trained Agent will perform better than the one after shorter training session, with the same training configuration.

After training session, model is typically saved in

```
$ TOP/src/saved_models/
```

directory, within a separate folder with new index (indexes start at 0). Model is saved along with its configuration file, graph presenting learning curve and log with rewards from every step.

Existing agent model can also be loaded into training session. In this case, you will be prompted to insert existing model's index number.

Basic configuration for this mode can be modified by changing contents of configuration.json, a part called **main**. Precise documentation of the whole configuration file can be found later in documentation.

NOTE: Altough basic configuration for main (learning) mode is fairly limited, succesfull learning is built upon all environment / agent settings in configuration file. Default ones are based on our research in order to provide the best and most meaningful learning results.

3.2.2 Simulation mode

This mode can be run by going into source files directory and launching **simulation.py** with python3 interpreter.

```
$ cd TOP/src
$ python3 simulation.py
```

After launching, you will be prompted to insert existing model's number to use in simulation.

NOTE: Simulation will start in full screen, be sure to have only one monitor connected - otherwise, simulation will span across all monitors.

In simulation mode, user is presented to graphical interface representing Reinforcement Learning Agent in action. There is a continuous environment, inside which Agent is trying to make his best decisions. No learning process included.

Key mapping scheme:

key	action
ESC	exit simulation
SPACEBAR	toggle pause / play
-	slow down simulation
=	speed up simulation
z	toggle charts zoom

Widgets

- Weather widget

  Located on the left side of the screen. This widget presents current time and weather informations outside of the house in form of a timer animation and five weather indicators:

  - temperature
  - sun intensity
  - wind intensity
  - clouds intensity
  - rain intensity

- Devices widget

  Located on the right bottom of the screen. This widget presents current house devices settings. Those include:

  - Energy source (icons) and battery level (percentage)
  - Cooling, heating, light and curtains level (leveled bar chart)

- Charts widget

  Located on the right center of the screen. This widget presents last 100 levels of light / degrees of temperature inside the house, along with users required level at given time.

  These two charts can be zoomed by pressing "z" on the keyboard. Zooming centers chart in a way that required level is placed directly at half of the charts height and the visible range is between $-0.1 * max$ and $+0.1 * max$ for maximum light / temperature levels respectively.

- Gauges widget

  Located on the right top of the screen. This widget presents current and desired level of light / temperature in form of a gauge chart.

Default speed of simulation can be set in **configuration.json** file, in module "simulation", item "fps".

3.3 Testing

3.3.1 Overview

To make sure that the code is always consistent with the base objectives and works correctly, the unit tests has been implemented. All of them can be found in **src/tests/** directory. Besides the regular unit tests there is also a separate file containing code consistency tests. The agent model can be tested using simulation mode or, with accuracy of a single action, using manual testing mode which is described later.

3.3.2 Coverage test

There is also possibility to run a coverage test using Python Coverage.py tool. If the code coverage needs to be checked the commands need to be invoked as follows:

```
$ cd TOP/src/tests
$ ./get_coverage.sh
```

The output is a .html file that can be found in htmlcov directory.

3.3.3 Manual testing

The project also includes manual testing mode. It can be run by going into source files directory and launching **manual_test.py** with python3 interpreter.

```
$ cd TOP/src
$ python3 manual_test.py
```

This mode is a command-line interface for manual testing of Reinforcement Learning Agent behavior. User can manually test agent-environment connection by deciding which action to make, or can load existing model to be able to use its choice in particular moment. The whole process is done one step at the time.

All actions are made by entering a number specific for a certain command. Commands and numbers are visible all the time on top of the rendered text.

NOTE: This mode was made mainly for testing purposes, and should be treated as additional, not main functionality.

4 Code overview

Below are the summarized descriptions of the fundamental parts of the project and how they are implemented in the code.

4.1 HouseEnergyEnvironment module

The environment where agent is put consists of a few separate classes that depend of each other and allow to create complex working setting that imitates the actual weather and the building.

4.1.1 House

This is where the agent is "placed" and where he operates. This class is responsible for generating the proper reaction after every timeframe based on the agent's action and outside world changes.

After every timeframe it receives the data from outside sensors and updates its parameters. During the update the temperature and light level are recalculated, energy cost increases and the energy stored in photovoltaics changes depending on the outside light and current energy source at home.

The House class contains a few methods used to calculate present environmental conditions. All of them implement the mechanisms that are meant to be as similar to the real world and its physics as possible. This way we could place the agent in the lifelike scenery and mimic its use in real life.

The vital part of the House is reward-calculating method. It is called after every timeframe. The important thing about the reward is the fact that it is always a negative value - that is why it can be treated as a "penalty". The agent's goal is to learn to minimize this penalty as much as possible. The reward formula is simply a weighted sum of penalties (temperature, light, energy cost) plus additional action penalty (for pointless or illogical actions like for example trying to switch powersource for the one that is already used).

4.1.2 World

This class represents the model of the outside world. Its main purpose is to compute time and weather. Each time the method step is called, the world adds fixed timeframe value to the current time, updates weather and sends information about the changes to the listeners.

The weather changes are computed using the random values as well as fixed dependencies between specific components. Most of the weights and biases connected to weather were determined experimentally. The main goal was to provide natural-looking weather with varied rain periods, clouds, occasional storms etc.In order to achieve this, in some places a modified version of Gillbert-Elliotchannel model was used to simulate constant periods of given weather phenomena.

4.1.3 Environment

Environment is indispensable part of the project. It connects smaller parts of the simulation like House and World into one working model. The class contains methods used to communicate with the agent as well as the ones that create human-readable information about current state. Basing on the received information

from the lower components, Environment is in charge of notifying their listeners about the events such as another step or finishing the episode.

4.2 Agent module

The agent module is where all the reinforcement learning mechanisms are implemented and used. To get the idea of how to use it and how it works, we split the module into small components and explain each one by one.

4.2.1 Neural Network

The neural network model used in learning and choosing the actions is modeled as the **Net** class. It is fairly simple thanks to PyTorch and the code pretty much speaks for itself. The forward function defines the forward pass.

4.2.2 Agent class

The main class of the module is the Agent class and it is essential to know how to use this class. Most of the code is much more easier to read and understand with beginner knowledge in the RL field. Here are the descriptions of the central methods of the class:

- *reset()* - method used to reinitialize the agent. It is called from the constructor, but you can call it separately if there is a need.
- *run()* - most essential method, called to perform a full episode on the environment - with learning. This method resets the environment, and perform a loop in which the agent chooses an action, receives feedback, performs *reward clipping*, updates statistics, saves the observed transition into the memory, calls the training function and then, in the end, slightly *updates the target network*. Reward clipping is one of the normalization methods and we are using it because of the commonly accepted suggestions to do so. The updates to the target networks are another concept that is used to stabilize the learning process and is explained in more detail in the next section.
- *learn()* - this is the method that performs training of agent's q-network. It is a single training iteration, which samples a batch of transitions from memory and, based on Double DQN approach, performs the training on given batch. Training is explained in the next section.
- *get_next_action_greedy()* - returns next action given a state with use of the target network using a greedy policy. Greedy means that agent return the action with the biggest q-function estimate. This function should be used if an outside object wants to know the agent's action for the given state.
- *get_episode_stats()* - method that implement the idea of collecting statistics about agent's actions, which is currently not used extensively. We are using these to register counts of each action to allow external object to analyze it.

4.2.3 Double DQN

4.2.4 Memory class

One of the concepts that is essential for Deep Q Networks to work is the **experience replay**. Agent, instead of learning from sequence of observed, correlated transitions, saves them to a memory buffer and samples random transitions into a batch of uncorrelated ones. This improves stability of learning. It is still unclear what should be the perfect memory size, but it is proved that it matters. Another part of research tries to whether how to improve the sampling, and the Prioritized Experience Replay concept, where the transition is more likely to be chosen based on 'how wrong the network was about it', is reported to be a big improvement across many tasks.

4.2.5 AgentUtils class

This class provides methods for saving and loading the models of networks and configuration parameters. We have separated it from the Agent class for readability and file-based operations separate from the learning functionality of Agent class.

4.3 Configuration file

The project allows user to adjust values of some of the key parameters such as training settings or environment specifications. The whole list can be found in **configuration.json** file in the main project directory. All of the available parameters are listed below along with their types and brief descriptions.

boldparameter name	type	description
training_episodes	integer	Number of episodes in training process.
save_experiment	boolean	Determines whether the model has to be saved after the experiment or not.
print_stats	boolean	Decides whether the current statistics has to be printed after each episode or not. The statistics are current most common action and how different are the current light and temperature values compared to their expected rate.
make_total_reward_plot	boolean	Determines whether the plot depicting the reward changes should be generated at the end or not.
load_agent_model	boolean	Decides whether the ready model should be loaded beforehand or not.
fps	integer	Number of frames per second in the simulation.
hidden_layer_size	integer	Size of the neural network hidden layer.
memory_alpha	float	Agent memory factor that determines how much prioritization is used (0 is uniform case)
memory_beta	float	Agent memory parameter used in weighted importance sampling.
memory_beta_increment	float	The value by which the memory_beta is gradually incremented during memory sampling.
memory_epsilon	float	Agent memory constant that prevents the transition from having zero-valued priority
memory_size	integer	Size of the agent memory.
double_dqn	boolean	Determines whether DQN or Double DQN has to be used.
gamma	float	The discount factor in learning algorithm.
epsilon	float	Controls the exploration rate in the learning process (probability of choosing random action).
epsilon_decay	float	The value by which the epsilon is gradually decremented.
epsilon_min	float	The minimum value of epsilon.
batch_size	integer	Size of transition batch retrieved from agent memory.
learning_rate	float	Value of learning rate in learning process.
training_freq	integer	The frequency with which the target network is slightly updated towards the q-network weights.
target_network_update_freq	integer	The frequency with which the target network is fully updated.
reward_clip	integer	Value of parameter used in the reward clipping.
sgd_momentum	float	Parameter used for stochastic gradient descent algorithm with momentum.

q_to_target_ratio	float	Value of which the target-network is updated during the learning process.
timestep_in_minutes	integer	Number of minutes in each world step.
day_start	integer	Number of minutes after midnight when the daytime begins.
day_end	integer	Number of minutes after midnight when the nighttime begins.
devices_power	integers	Power of the devices in the house (watt).
temperature_w_in_reward	float	Weight of the temperature penalty in reward calculation.
light_w_in_reward	float	Weight of the light penalty in reward calculation.
cost_w_in_reward	float	Weight of the cost of the energy penalty in reward calculation.
max_pv_absorption	integer	Maximum absorption of photovoltaics.
day_grid_cost	float	Energy price during the daytime.
night_grid_cost	float	Energy price during the nighttime.
house_light_factor	float	Determines how much of outside light is in the house.
house_isolation_factor	float	Parameter used in outdoor-to-indoor temperature calculation.
battery_max	integer	Amount of energy in fully charged photovoltaics.
influence_per_min	float	Describes how large is the change of device setting when action is executed
stats	floats	Maximum differences between measured stats and expected. These constants tell which temperature and light level are described as "perfect" and "ok" compared to the expected values.

This is the configuration we used to generate the final agent and make it learn:

```
{
    "main": {
        "training_episodes"          : 10000,
        "save_experiment"            : true,
        "print_stats"                : true,
        "make_total_reward_plot"     : true,
        "load_agent_model"           : false
    },
    "simulation": {
        "fps"                        : 16
    },
    "agent": {
        "memory_alpha"               : 0.6,
        "memory_epsilon"             : 0.01,
        "memory_beta"                : 0.4,
        "memory_beta_increment"      : 0.001,
        "hidden_layer_size"          : 80,
        "memory_size"                : 10000,
        "double_dqn"                 : true,
        "gamma"                      : 0.9,
        "epsilon"                    : 1,
        "epsilon_decay"              : 0.999,
        "epsilon_min"                : 0.1,
        "batch_size"                 : 16,
        "learning_rate"              : 0.0001,
        "training_freq"              : 4,
        "target_network_update_freq" : 500,
        "reward_clip"                : -2,
        "sgd_momentum"               : 0.9,
```

```
        "q_to_target_ratio"          : 0.1
    },
    "env": {
        "timestep_in_minutes"        : 1,
        "day_start"                  : 420,
        "day_end"                    : 1080,
        "devices_power": {
            "air_conditioner": 1500,
            "heater": 3000,
            "light": 720
        },

        "temperature_w_in_reward"    : 0.35,
        "light_w_in_reward"          : 0.82,
        "cost_w_in_reward"           : 0.024,
        "max_pv_absorption"          : 5,
        "day_grid_cost"              : 0.5,
        "night_grid_cost"            : 0.3,
        "house_light_factor"         : 0.0075,
        "house_isolation_factor"     : 0.996,
        "battery_max"                : 14000,
        "influence_per_min"          : 0.2,
        "stats": {
            "temp_ok_diff": 2,
            "temp_perfect_diff": 0.5,
            "light_ok_diff": 0.15,
            "light_perfect_diff": 0.05
        }
    }
}
```

5 Accuracy measures and tempo of learning

5.1 Parameters impact on the agent learning

Once the final version of the neural network has been implemented, we have calibrated all its parameters in such a way that the agent's learning was as efficient as possible. The word "efficient" is understood as the most accurate teaching of all the functions performed by the agent.

Parameters were selected by repeatedly learning the network again and again, each time with different parameter values. So to test a particular set of values, we had to repeat the entire learning process of the agent. Due to the inability to perform these processes on a strong machine or computing cloud using the GPU calculations, we could not carry out many tests.

Parameters that affect the learning of the network are included in the configuration json file. Belong to them the fields inside the label "agent" and the fields from the label "env"; weight of temperature, light and energy consumption.

Parameters descriptions:

- Hidden layer size specifies the number of neurons in the inner layer of the neural network. The more neurons, the more states of the agent's science can be determined. If the value is too small, it can be shown that the agent will not be able to learn all the required functions. However, when this value will be too much, then created redundant neurons will slow down the propagation process inside the network, and thus the learning process.

- The memory size specifies the amount of recent Agent transitions batches. A priority is given for each written transitions to the memory. The bigger the penalty caused the transition, the higher the priority is set for it. Then, when selecting another action, the Agent searches the memory and selects transition from it. The greater the priority given to a transition, the more likely it is that the agent

chooses it. Thus, the Agent can focus more on these elements during the learning, which generate a larger penalty, that is, those who have learned less.

- Double dqn determines whether during the training of the agent we use the second network in accordance with the double q learning algorithm. Using this algorithm makes learning more stable, as opposed to using only one network, where its values are burdened with larger noises.

- The gamma parameter determines to what extent the reward for a new change is important to us. The smaller the parameter, the more we rely on the temporary reward, the one we get from the current stage, and less on the previous stage.

- In its basic setting, the decision maker takes and action, and gets a reward from the environment, and the environment changes its state. Then the decision maker senses the state of the environment, takes an action, gets a reward, and so on so forth. The state transitions are probabilistic and depend solely on the actual state and the action taken by the decision maker. The reward obtained by the decision maker depends on the action taken, and on both the original and the new state of the environment.

- Epsilon specifies the probability of selecting a random action by an agent in a given episode. Because the learning process consists in maximizing the reward function, it can result in getting stuck inside a one from many locals minimum. So epsilon provides that there is a probability when the agent will choose a random action that will make it jump out of this minimum. In accordance with the papers for learning to proceed correctly, the chance of selecting a random action should be decreased with the duration of the study, so that more often perform actions in accordance with the acquired knowledge, rather than try to perform random actions again.

With each next episode the actual value of epsilon is reduced by the formula:

epsilon = epsilon * epsilon decay, where epsilon decay belongs to (0,1).

The minimum chance to select a random action is determined by the value of epsilon min.

- Batch size defines number of samples that going to be propagated through the network. The smaller the batch size, the faster the values propagate over the network and the estimate of the gradient is less accurate.
- Learning rate determines the Agent's learning speed. The larger the model values, the greater the gradient value is added to the model parameters. Choosing the value of the learning rate depends on the type of the problem. However, according to the papers, this value should be very small.
- Freq training defines what episodes we actually do Agent training. According to the reviewed papers, it turns out that when the actual training will not take place during the episode, the agent learning process will be more stable. It is better to do the learning in one episode, and in the next few make decisions based on this earlier learning, without further learning, until the next number of episodes determined by the training freq parameter. Furthermore, the Agent's training with a specific frequency, we accelerate the overall process of learning the agent.
- Target network update freq specifies the number of episodes after which the target network is fully updated. We ignore the Q to the target ratio parameter and the whole target network refreshes by changing the values of its parameters to the value of q network parameters.
- Reward cliping is a parameter that is the lower limit of the reward received by the agent during learning. When an agent chooses his actions, he receives penalties for bad decisions. Sometimes these penalties can be very large, so when they exceed the critical value set by reward cliping, they are cut off to that particular minimum value, which says that "it is already so bad, that it can not be worse".
- Sgd momentum value specifies how the previous update parameter affects the new, just performed update. Is a response to the disadvantages of the standard descent gradient. Momentum proposes the following tweak to a gradient descent. It gives a gradient descent a short-term memory, so the improved descent gradient more effectively finds the function optima. According to the papers, in order for the momentum to bring the greatest benefit, it should have values from the 0.99 to 0.999 range.
- Q to the target ratio is the parameter determining the importance of changes in the target network when we combine it with the q network. This subroutine determines how big part of the q network parameter values should be summed with the remaining value of the target network parameters.
- The weight of temperature, light and the cost of consumed energy are some of the values set by the user. They say about whether he prefers to have, lower bills at the end of the month, with less accuracy in maintaining the set parameters, or maybe he does not, and he prefers more accurate holding set temperature or light, with a minimum bigger energy consumption. If we significantly increase the cost of energy consumption, it may turn out that the agent determines that it is better not to keep the temperature set by the user at all, because the more it will enjoy lower energy costs than freezing at home.

6 Development Process

It is not a secret, that this project is more of a research than it is about making a production-ready system. While the most important research is concluded in the next section, we want to write couple of notes about our development process, to make it easier for new contributors to understand how the project was constructed. The most important tasks that were done are listed chronologically, and we also share some of the things that failed during the development process.

6.1 Chronology

Chronology of system preparation:

- Defining the organizational dates of such team meetings, talks with the project supervisor - 28/02/2018
- Design the wireframe of application - 13/03/2018
- The design of the reward function received by the agent - 13/03/2018
- Creation of the class diagram - 15/03/2018
- Implementation of actions possible to be performed by the agent - 19/03/2018
- Implementation of the reward function - 20/03/2018
- Agent class wireframe design - 25/03/2018
- Preparation and implementation of internal dependencies - 24/03/2018
- Preparation of a list of conventions on the source code of the application - 4.04.2018
- Implementation of the basic version of the agent - 18/04/2018
- Creation of weather dependencies in the environment - 18/04/2018
- Documentation on the algorithms used in the agent - 19/04/2018
- Manual tests of the environment - 20/04/2018
- Saving and loading agent's neural network models - 25/04/2018
- Analysis and tests of cloud solutions used to speed up calculations - 8/05/2018
- Implementation of the console user interface - 8/05/2018
- Implementation of Double DQN - 8/05/2018
- Creation of configuration files for store Agent parameters - 14/05/2018
- Design and functional requirements of the GUI - 4.05.2018
- GUI implementation - 21/05/2018
- Presentation of the system 11/06/2018

6.2 What did not work out

As a young team of developers, we are very happy to work on an exciting field of research, but we have to admit that some of the things about developing a solid, consistent code base for a team project turned out to be very challenging. Additionally, working with Reinforcement Learning agents and environments can be very tricky. In this section we are going to name some of the things that did not work out at all, including both our failures and concepts that have not improved the RL agent.

- The first thing that we have failed in, is to maintain a full scalability of the environment. While there are some mechanisms that make it easier to add a feature to the simulator, like automatic action-methods detection, we feel that with more of an 'enterprise' approach to code architecture or more abstraction would make it even easier. The simulation and manual testing modes are designed more or less for the current state shape. The good news might be that the agent's class does not need any code tweaks during a state shape change.

- We have not tried to compare the Q-Learning method with the other well known approach called Policy Gradient methods. These methods try to directly estimate the policy instead of Q function values, and, considering the characteristics of our environment, this could work even better. It might be harder to estimate a pretty complicated function, but fairly easy to find out about rules like 'turn heating on when it is too cold'. We list these methods as one of the possible further research suggestions.
- One of the research fields in reinforcement learning is to allow development of RL environments to be easier by using a sparse reward function. This means that the agent receives a 0 signal, when the goal is achieved, and -1 otherwise. We have tried out this approach, using function similar to that described, and it drastically decreased performance of the agent. There are some recent improvements in the field, probably the loudest was the Hindsight Experience Replay concept, which was proven to, in some cases, condition the solving of a task, in which the reward function is sparse. We have not tried to implement this feature.
- Another thing that made the learning process harder was the way we modeled the reward function at first. Both quadratic functions for penalties and concept of 'acceptance intervals' for light and temperature differences made the learning harder, as the resulting Q function, that our model had to estimate, got much more complicated that it needed to be. The lesson here was to keep the reward function simple, and to not introduce any constructs that might be rational to a human, but do no good for the agent.
- While this might not be a proper failure, we are aware that the project - or this documentation - lacks statistics about the experiments, including results that would suggest the impact of certain parameters to quality of learning. We explain this by very broad space of possible configurations and limited computing power available during the development. It is also worth to come back to the project's main ideas and goals - as these were not strictly about extensive research on how to make the learning the fastest, but more to check the overall potential of reinforcement learning methods.

7 Conclusions

While working on the project, we learned a lot of things in the field of reinforcement learning, which is part of the wider section of artificial intelligence. We had the opportunity to learn how to work with the PyTorch computing package, which was created not only to work on artificial intelligence, but it can also be used for complex matrix calculations, which have nothing to do with machine learning. Lot of this things we learned during the previous research and others during the implementation. In addition to the items related to reinforcement learning, we also learned how to collaborate in a team and plan progress in the project, sharing its implementation into smaller parts, which we solved in accordance with the kanban system.

7.1 Further Development

Based on what we have not tried out yet and what have we read during the project development, we would like to list some of the potential improvements and further research directories.

- First, the alpha version of the house simulation treated as a single room, with only one sensor. Thus, in subsequent versions, it would be possible to give a greater reality by introducing further sensors and rooms, and then model the heat flow between the rooms.
- We could add more factors that an agent would deal with, such as humidity or pollution.

8 Bibliography and Useful Sources

8.1 Reinforcement Learning - Getting Started

- Richard S. Sutton and Andrew G. Barto. 1998. Introduction to Reinforcement Learning. MIT Press, Cambridge, MA, USA.
- Reinforcement Learning Course by David Silver - http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html

8.2 Reinforcement Learning - Papers on improvements to basic methods

- Playing Atari with Deep Reinforcement Learning - https://www.cs.toronto.edu/ vmnih/docs/dqn.pdf
- Deep Reinforcement Learning with Double Q-learning - https://arxiv.org/abs/1509.06461
- Prioritized Experience Replay - https://arxiv.org/abs/1511.05952

8.3 Reinforcement Learning - Useful Sources

- https://github.com/dennybritz/reinforcement-learning
- Rainbow: Combining Improvements in Deep Reinforcement Learning - https://arxiv.org/abs/1710.02298
- https://jaromiru.com/2016/11/07/lets-make-a-dqn-double-learning-and-prioritized-experience-replay/

8.4 PyTorch

- https://pytorch.org/tutorials/