Effective House Energy Management Using Reinforcement Learning Technical Documentation

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

June 3, 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

1.1 Goals and milestones

1.2 Results

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

3.2 Learning and Simulation

3.3 Testing

4 Code overview

4.1 HouseEnergyEnvironment module

4.2 Agent module

4.3 Configuration file

5 Accuracy measures and tempo of learning

6 Development Process

6.1 Chronology

6.2 What Has Failed

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

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.

-

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/