Sparse Negative Feedback for Cooperative Inverse Reinforcement Learning

University of Groningen

Vladyslav Tomashpolskyi
s2891654

09 October 2017

Abstract

With the current situation in reinforcement learning field, a lot of attention has been devoted to solving different kinds of problems applying deep learning methods to them. This also includes tasks from a cooperative inverse reinforcement learning(CIRL) subfield - specifically aimed at agents discovering and adjusting to the reward function of a human controlling them.

We test performance of several well-established reinforcement learning algorithms - A3C, ANQL, DDQN, DQN and A3C RNN - on a model devised to represent a CIRL task with the help of negative feedback in both POMDP and MDP settings. We investigate the impact of regularization and historical data on their performance, as well as influence of reward function decomposition on the learning process.

We find that A3C RNN algorithm is capable of reliably learning an expected policy, without falling into a trap of producing an *optimal value policy*. We also find that DDQN algorithm is capable of learning complex policies with the help of regularization.

This research is aimed to uncover further work prospects and investigate the performance of currently available algorithms on the CIRL negative feedback based tasks.

Acknowledgements

I want to thank the crew of Sustainable Buildings, who were understanding and compassionate regarding my undertakings, ready to provide whatever help or support necessary. Specifically, I want to thank Ilche Georgievski, who helped me to distract myself from work and provided necessary relief opportunities, and Angdrija Curganov, my dear friend, who always head something fun to converse about, helping me to keep myself in one piece.

I am grateful to Brian Setz and Azkario Rizky for guidance and provided valuable lessons and feedback. Whenever I was lost or stranded - I could always find some direction with their help, and our brainstorming sessions regarding a number of different issues always were a welcome relief and provided an important contributions to this work.

My supervisor from AI department of University of Groningen prof. dr. Lambert Schomaker, shared his impressive knowledge of the subject with me, as well as his ideas about the direction that this research should undertake. Without him, it would be a completely different paper.

My daily supervisor Prof. dr. Alexander Lazovik taught me a lot of things I did not know I needed learning. His ideas and guidance were instrumental for this thesis, as was his feedback and support. Discussions with dr. Lazovik's always left me eager to work just a little bit more.

My close friend Alexander Pospishnyi, supported me with his faith and unquestionable loyalty during the moments that I needed these qualities the most.

I want to thanks my unexpected friend and a person that put a lot of work into making this paper happen - Alona Tyshaieva. Without her feedback and help this paper would be much harder to read and understand.

I want to thank my family, without whom this whole enterprise would not be possible. My father Alexander Tomashpolskyi, for his continuous support and everlasting readiness to hear out my thoughts and explanations, and to participate in a useful discussion. My mother Marianna Tomashpolska, for her strive to create

homey comfort and provide so much needed support. My sister Sofia Tomashpolska, for her interest and care for my well being.

And, of course, I am deeply grateful to my beloved Lidiia Tarnavska, who gave me strength not to break during the months that this research took. She was always there for me, whenever I needed help and support. She provided essential hope, sometimes opening my eyes, and at other times just carrying me. ”When you can’t run, you crawl, and when you can’t crawl - when you can’t do that... You find someone to carry you.” Thank you.

Contents

List of Abbreviations

A3C Asynchronous Advantage Actor-Critic

ADHDP Action-Driven Heuristic Dynamic Programming

ADP Adaptive Dynamic Programming

ANN Artificial Neural Network

BCP Building Control Problem

BPTT backpropagation-through-time

CEC Constant Error Carousel

CIRL cooperative inverse reinforcement learning

CNN Convolutional Neural Network

DDQN Double Deep Q-Network

DHP Dual Heuristic Dynamic Programming

DP Dynamic Programming

DQN Deep Q-Network

DRQN Deep Recurrent Q-Network

FC fully connected

FNN Feed-forward Neural Network

FUN FeUdal Network

GFV General Value Function

HDP Heuristic Dynamic Programming

HRA Hybrid Reward Architecture

IRL inverse reinforcement learning

LSTM Long Short-Term Memory

MDP Markov Decision Process

MLP Multilayer Perceptron

NN Neural Network

POMDP Partially-Observable Markov Decision Process

relu rectifier nonlinear activation

RL Reinforcement Learning

RNN Recurrent Neural Network

SAM Scaled Memory-Augmented Neural Networks

SGD Stochastic Gradient Descent

SL Supervised Learning

SVG Stochastic Value Gradient

UL Unsupervised Learning

VGL Value Gradient Learning

VL Value Learning

Chapter 1

Introduction

Recent breakthroughs in deep reinforcement learning have sparked a new wave of interest towards the applications of these techniques. Developments in reinforcement learning allow control methods to show human-level operational capabilities. This opens significant prospects for further applications of artificial intelligence methods. Being able to demonstrate robust performance in dynamic Markovian environments, e.g. Atari games (Mnih et al., 2015), deep Q-network signified a new wave of interest towards applications of neural networks for control problems. It led to the development of several derivative algorithms (Wang et al., 2015) and culminated with asynchronous advantage actor-critic, which manages to consistently outperform human subjects(Mnih et al., 2016).

However, applying a neural network agent to control a game, where each consequent action and step has a strict and definite result, and where an end condition is strictly defined, is not the same as controlling an autonomous car on the street or environmental conditions in a building.

One of the concepts that are of interest is the *cooperative inverse reinforcement learning* (Hadfield-Menell, Dragan, Abbeel, & Russell, 2016). This concept introduces the setup, in which the reinforcement learning agents optimize not their intrinsic reward function, but rather strive to align their values with those of end-users. This is useful in many ways, one of which we take as a primary use case for this work. Consider an agent that has the control over environment systems of a home or an office. The necessary requirement for this agent is to optimize the environment so that all the occupants are comfortable. To do that the reinforcement learning agents need to accept the goal of learning and adhering to non-expert human preferences (Fürnkranz, Hüllermeier, Cheng, & Park, 2012; Christiano et al., 2017).

Of course, in a described setting the explicit user feedback may be very scarce. However, if the user actually continues his usual operations, while an agent learns to align his value function with the user's in the background, then the explicit feedback in not as necessary anymore. This way the system only receives required information about user preferences from their usual operational behavior. With this thesis we strive to investigate and answer several questions related to this kind of continuous control with implicit scarce negative feedback in three main areas:

Performance How well do existing reinforcement learning solutions handle cooperative inverse reinforcement learning tasks with respect to scarce negative feedback? Can we achieve a reliable satisfaction of user preferences just from the observations and minimize the negative feedback?

Reward Decomposition Usually automation systems have several different correlated goals: e.g. while making sure that office occupants are comfortable with regard to environment conditions, the tasks of minimizing the consumption of different control systems is also important. What influence does reward decomposition have on the performance of an agent?

Optimization What influence do different learning optimization techniques have on this learning? Does the availability of historical data or regularization provide direct advantage?

To investigate these questions we need a specific use case that presents a cooperative inverse reinforcement learning task, while also providing an environment without strictly defined end state. One of the possible candidate problems comes from the domain of smart environments: a building control problem (BCP)(Georgievski et al., 2017). The main goal of BCP is to satisfy user experiences which represents the cooperative inverse reinforcement learning task. In this thesis we devise a model that corresponds to basic BCP definitions and try to optimally solve it using a set of recent and up-to-date reinforcement learning agents.

Chapter 2

Background

In this chapter applications of neural networks to reinforcement learning in general and control tasks in particular are described. We start with the definition of the environment that constitutes the base of reinforcement learning. In section 2.1 the general assumptions about Markovian environments are covered. However, due to improbability of real-world Markovian environments, its extension - partially-observable Markovian environment - is also described.

The domain of Adaptive Dynamic Programming (ADP) is described in subsection 2.2.1. Different ADP methods make use of Bellman's optimality principle (Bellman, 1957) to solve the *optimal control problem*. One of the main tasks of these methods is to train a value (or value gradient) function approximator.

Neural networks are often used as function approximator subsystems, so a short introduction to neural networks is given in section 2.3. Then different structures of both feed-forward (FNNs) as well as recurrent neural networks (RNNs) are described in subsection 2.3.1.

Given this structural support, we describe several algorithms which were successfully used to solve control problems. We begin with an introduction to aforementioned DQN (subsection 2.4.1) and its derivative algorithms. Their general structure, performance, and drawbacks are discussed. Non-ADP related approaches and techniques used in reinforcement learning are also introduced. These are represented by hierarchical structures, e.g. FeUdal Networks(FUNs).

2.1 Environment

The key problem that the reinforcement learning addresses is to learn from interactions in order to achieve a goal. The learner is usually called an agent, while everything outside of an agent is referred to as *environment*. As the time progresses the agent continually interacts with the environment by selecting and performing actions. Once performed, these actions influence the environment which, in turn, produces the reward and presents the new situation to the agent. (R. Sutton & Barto, 2017)

The environments which produce varying and sometimes seemingly random responses to the action that the agent performs are called *uncertain* environments. The common way to represent such a setting is a Markov Decision Process.

2.1.1 Markov Decision Process

A Markov Decision Process is a discrete time stochastic control process widely used in reinforcement learning tasks. An MDP is a 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ is a finite set of *states*, $\mathcal{A}$ is a finite set of *actions*, $\mathcal{P}$ is a *state transition function* in form of a probability mapping from $\mathcal{S} \times \mathcal{A}$ to $\mathcal{S}$ which defines the success of a transition, $\mathcal{R}$ is a *reward function*, and γ is a discount factor, representing the importance of future rewards over immediate ones.

At each time step t the agent observes the state $s_t \in \mathcal{S}$ and takes an action $a_t \in \mathcal{A}$. This determines the reward $r_t \sim \mathcal{R}(s_t, a_t)$ and the next state $s_{t+1} \sim \mathcal{P}(s_t, a_t)$.

Another intrinsic quality of MDPs is that every given state imparts all the information necessary to act optimally. The state transition function is not dependent on any of the previous states, only on the current s_t and the concrete action a_t performed. This means that in any given state decision can be made based only on the information available from this state and this information alone. This is called the *Markov Property*.

2.1.2 Partially-Observable Markov Decision Process

Most of the real-world situations do not provide the information about the full state of the system in its entirety. Some information is internal to the system while some might be just impossible to get. This inevitably leads to the violation of Markov Property: once the system cannot certainly detect which state it is in, the data present is no longer sufficient to determine optimal policy.

However, this does not mean that MDPs are not applicable to this partially observable framework. The MDP modification known as Partially-Observable Markov Decision Process (POMDP) exists to support environments where *Markov assumption*[1] holds, but the data is not possible to get in full. This allows to acknowledge that the agent's observations about the state are incomplete and provide just pieces of information about true situation.

The POMDP is a 7-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma, \Omega, O)$ which extends MDP with a set of possible observations Ω and a set of conditional observation probabilities O.

Now at each timestep t the agent does not observe the state s_t directly, but rather receives an observation $o_t \in \Omega$, which is generated according to the underlying process state $o_t \sim O(s_t)$ (Jaakkola, Singh, & Jordan, 1995).

2.2 Reinforcement Learning

Modern machine learning is applied to a wide variety of different tasks and thus constitutes a broad field. It is usually broken down into three main subfields (Russell & Norvig, 2002):

- supervised learning (SL) - is mainly concerned with predicting a correct outcome or future and is trained on labeled data with direct feedback. The ImageNet (Russakovsky et al., 2015) is a good example of a dataset that may be used in SL tasks. This subfield is a good example of a traditional use of machine learning in classification problems.

- unsupervised learning (UL) - takes upon itself a task of finding hidden structures and patterns within given datasets. In this subfield feedback is nonexistent. Clustering problems fall into this category of machine learning.

- reinforcement learning (RL) - is concerned with making correct decisions and optimizing agent performance in order to achieve highest possible reward. In layman terms - learn from interactions to achieve a goal. Each action an agent may perform to interact with its environment produces some reward as a consequence, and its optimization is a task that RL needs to address. Unlike in SL, correct input/output pairs are never presented to RL algorithms, neither sub-optimal actions are explicitly corrected (Russell & Norvig, 2002).

[1] assumption that Markov Property holds for the system

RL frequently deals with on-line tasks, so it is a natural fit for optimal control problems. Further in this section we describe main methods of reinforcement learning. Whenever an agent interacts with an environment it is able to observe its changes and receive a reward or punishment from it (see Figure 2.1).

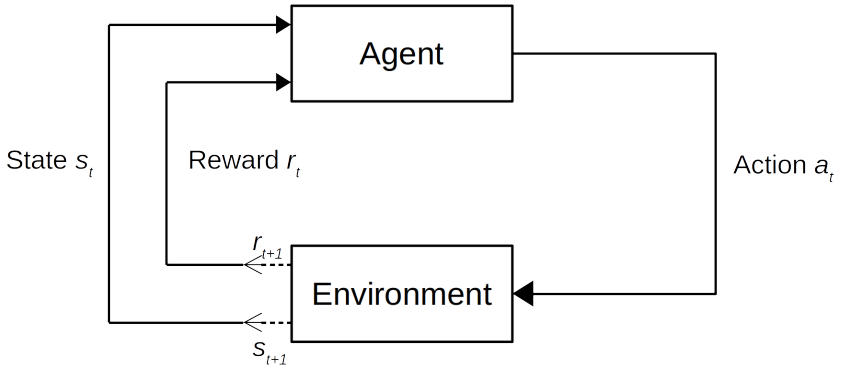


Figure 2.1: RL pipeline: the interaction between an agent and an environment

2.2.1 Adaptive Dynamic Programming

Dynamic programming (DP) is used to solving optimal control problems and thus is applicable to RL tasks. It employs Bellman's principle of optimality (Bellman, 1957): "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." Suppose an agent observes system state vector $s_t \in \mathcal{S}$ at timestep t. It then chooses an action $a_t \in \mathcal{A}$ which then influences the state according to the environment function $s_{t+1} = f(s_t, a_t)$, and implies an immediate scalar cost $r_t = R(s_t, a_t, t)$.

Let us assume that there exists a *value function*[2] J such that:

$$J(s_t, t) = \sum_{k=t}^{\infty} \gamma^{k-t} R(s_t, a_t, t) \tag{2.1}$$

The meaning of the value function is to specify a minimal cost-to-go (or maximum reward) achievable from each particular state in the state space. The main

[2] in the literature frequently interchanged with *cost-to-go* function

task of DP is to choose sequence of actions $a_t, a_{t+1}, ...$ so that the value of J is minimized:

$$J^*(s_t) = \min_{a_t} \left(R(s_t, a_t) + \gamma J^*(s_{t+1}) \right) \tag{2.2}$$

And the optimal action at this point in time is the action a_t^* which achieves the minimal value $J^*(s_t)$:

$$a_t^* = \arg\min_{a_t} \left(R(s_t, a_t) + \gamma J^*(s_{t+1}) \right) \tag{2.3}$$

Equation 2.3 is the principle of optimality for discrete-time systems.

However, running true dynamic programming is often computationally untenable as a result of well-known "curse of dimensionality" (Bellman, 1957; Wang, Zhang, & Liu, 2009). One other drawback of the true DP is the requirement of having already known MDP which is not always possible.

As an extension of the DP, the Adaptive Dynamic Programming (ADP) makes use of function approximators and actor-critic systems (as shown on Figure 2.2) to circumvent the "curse of dimensionality." It proposes an agent that consists of two parts: *actor* or *action network* or *policy function* and *critic* or *approximate value function* (Fairbank & Alonso, 2012).

The action network specifies which action $a = A(s, w_a)$ should be taken based on a given state s and a set of weights w_a. The preferred action is based on minimization of cost-to-go function J, and the task of ADP and RL is to train the actor to select actions that lead to the desired optimization.

However, J is unknown. To circumvent this, a critic network is introduced. It's main goal is to learn a set of critic's internal weight parameters w_c to approximate $\hat{J}(s, w_c) \approx J(s)$. If the critic perfectly approximates the value function, while at the same time actor always chooses its actions according to Equation 2.3, then Bellman's principle of optimality holds and sequences of actions produced will be optimal.

The method of always choosing actions according to Equation 2.3 is called the *greedy policy*, because the best-rated actions are always chosen. However, this may lead to a famous problem of RL - *exploitation* vs *exploration* (Kaelbling, Littman, & Moore, 1996).

In the context of RL exploitation means employing the learned policy and never diverging from the produced trajectories - as that is inefficient according to critic. Nonetheless, this course of actions is prone to falling into the trap of local extrema. On the other hand, exploration, while being less efficient, provides

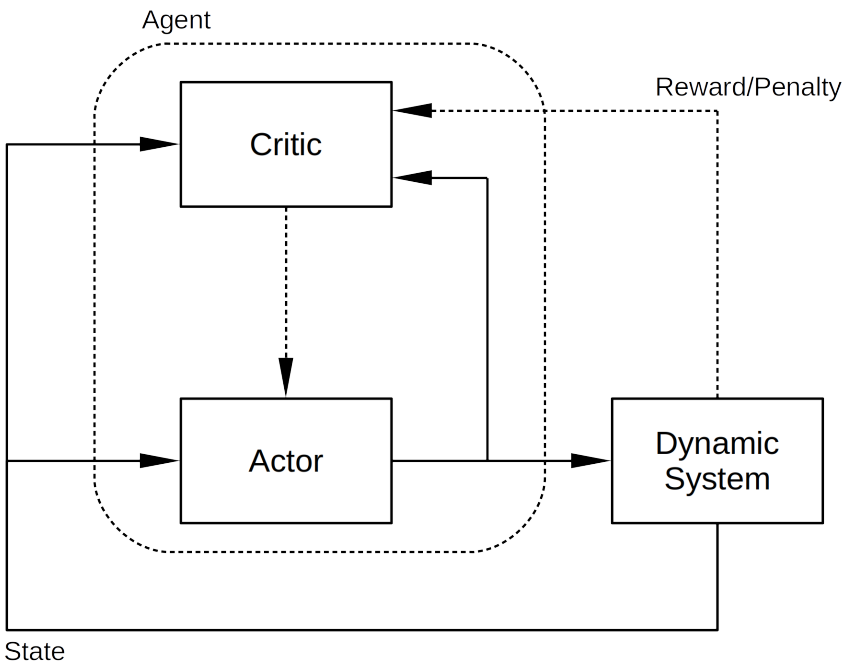


Figure 2.2: Actor-critic agent structure. Adapted from Wang et al. (2009).

more insights about overall shape of policy landscape in the vicinity of chosen trajectory. Striking the balance between exploitation and exploration is an open question, though some solutions exist, e.g. ε-greedy strategy (R. S. Sutton & Barto, 1998).

Value Learning Methods

There are several different approaches to ADP. One of them is called Heuristic Dynamic Programming, also known as *value learning* (VL) (Fairbank & Alonso, 2011). This approach strives for the critic to learn the value function itself, assessing its values along the trajectory and using it to select every consequent action in a *greedy policy* way, abiding by Equation 2.3. Fairbank and Alonso argued that these methods, despite showing some successes, can be slow since Bellman's condition needs meeting over the entirety of the state space. They argue, that even if Bellman's condition is perfectly satisfied along a single trajectory, it is required to be satisfied along the neighboring trajectories as well, otherwise even local optimality cannot be guaranteed. This calls for the exploration techniques as

discussed above.

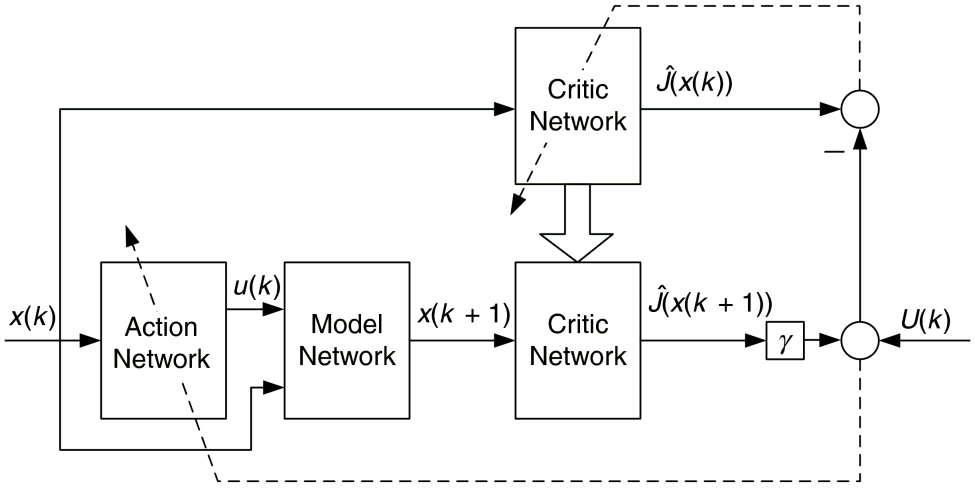


Figure 2.3: Heuristic Dynamic Programming structure. Adapted from Wang et al. (2009).

The general structure of HDP systems is shown on Figure 2.3. It includes action and critic networks (on the figure two critics actually share weights as indicated by the large arrow), as well as the implementation of the model network (necessary to learn the environment function[3]) and backpropagation paths.

The training process for critic lies in minimization of

$$\|E_h\| = \sum_{s_t} E_h(s_t) = \frac{1}{2}\sum_{s_t}[\hat{J}(s_t) - A(s_t) - \gamma\hat{J}(s_{t+1})]^2 \tag{2.4}$$

Once $\|E_h\| = 0$ for all s_t, then the value function is learned, meaning that $\hat{J}(s_t) = A(s_t) + \gamma\hat{J}(s_{t+1})$ and hence Equation 2.2 is satisfied. This, however, leads to the problem of convergence for VL methods. The training of critic's value function $\hat{J}$ depends on learned policy $A(s_t)$, which, in turn, depends on critic's value function again. This means that learning one might lead to deprecating another (Fairbank & Alonso, 2011).

[3]Not entirely necessary, as proven by Si and Wang (2001)

One of the most well-known HDP modifications is Action-Driven HDP (ADHDP) which also provides action value as an input for the critic network. Famous Q-learning algorithm is an example of ADHDP system (Wang et al., 2009).

Value Gradient Learning Methods

Another important approach to ADP is called Dual Heuristic Dynamic Programming, also known as *value gradient learning*. It's structure is similar to that of HDP systems, with one significant difference. The function being learned by critic is not the value function $\hat{J}$ anymore, but rather its gradient $\frac{\partial \hat{J}(s_t)}{\partial x(s_t)}$. The critic network's training is more complicated than that in HDP since we need to take into account all relevant pathways of backpropagation (Wang et al., 2009).

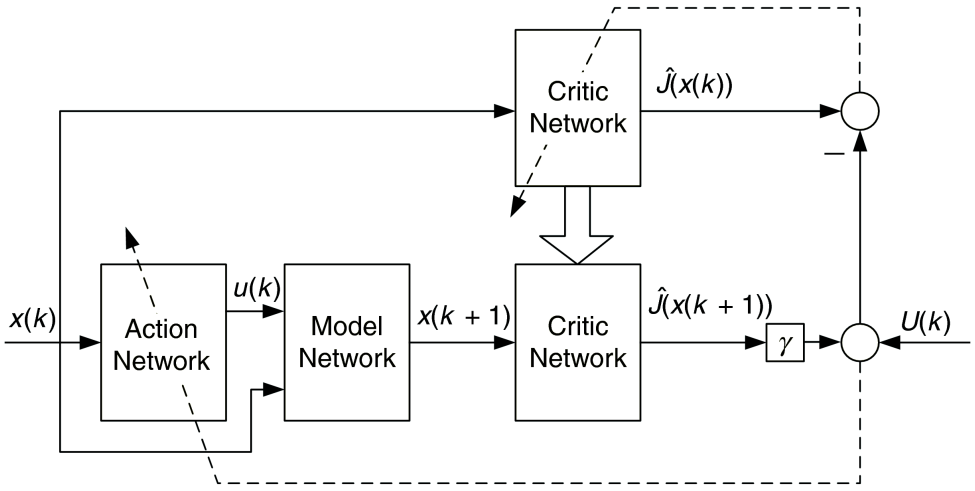


Figure 2.4: Dual Heuristic Dynamic Programming structure. Adapted from Wang et al. (2009).

Unlike with VL methods, it is not necessary to learn the value gradient along all neighboring trajectories - it is enough to learn it along a single trajectory (Fairbank & Alonso, 2011). This gives VGL methods a significant efficiency gain. Provided that the VGL methods makes progress learning value gradient along a single trajectory, it automatically bends towards local optimality, without the need of stochasticity in the policy. This means that exploration is performed

without any extra resources and hence leads from "exploration *or* exploitation" RL problem to "exploration *and* exploitation".

The training process for DHP critic lies in minimization of

$$\|E_D\| = \sum_{s_t} E_D(s_t) = \frac{1}{2} \sum_{s_t} \left[\frac{\partial \hat{J}(s_t)}{\partial x(s_t)} - \frac{\partial A(s_t)}{\partial x(s_t)} - \gamma \frac{\partial \hat{J}(s_{t+1})}{\partial x(s_t)} \right]^2 \tag{2.5}$$

Once $\|E_D\| = 0$ for all s_t, then the value function is learned, meaning that

$$\frac{\partial \hat{J}(s_t)}{\partial x(s_t)} = \frac{\partial A(s_t)}{\partial x(s_t)} + \gamma \frac{\partial J(s_{t+1})}{\partial x(s_t)} \tag{2.6}$$

Once Equation 2.6 holds, the value gradient function is learned.

Both VL and VGL methods require a perfectly learned critic (value function or value gradient function) on the entirety of state space to achieve global optimality. Once the critic is perfectly learned Bellman's Condition will ensure global optimality.

2.3 Neural Networks

As seen in previous section, function approximation systems are necessary for ADP to work properly. One of the widely used techniques for function approximation is the use of artificial neural networks (ANNs). In this section the basics concepts of the neural networks are covered.

One of the early models of ANNs is a famous perceptron algorithm described by Rosenblatt (1958). It is a simple model that is capable to learn a linearly separable rule and is widely used in supervised learning problems as a binary classifier for linearly separable datasets. The idea is simple: the perceptron is a unit that consists of several inputs x, a set of corresponding weights w and a bias b that represents an activation threshold. Then a step function defines the activation of the perceptron:

$$f(x) = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 1 \end{cases} \tag{2.7}$$

However, this model is hard to teach. A small change in w will lead either to no change at all or to a very significant change in the output of the perceptron. This problem may be solved by the introduction of a type of neurons know as *sigmoid* or *logistical* neuron (Nielsen, 2015). It operates is a similar manner to

the previously described perceptron, but uses a smooth activation function instead of a step function:

$$f(x) = g(w \cdot x + b) \tag{2.8}$$

One example of such a smooth activation function is sigmoid (hence the name of this neuron type):

$$g(z) = \frac{1}{1 + e^{-z}} \tag{2.9}$$

Unless stated otherwise, all the neurons mentioned int this thesis are considered to be logistical with sigmoid as an activation function.

Learning

Now that the general algorithm for a neuron is defined, it needs to learn from data. Learning is an ability of the program to improve its performance with regard to some task based on encountered experience with regard to some performance measure (Goodfellow, Bengio, & Courville, 2016). For neural networks this is powered by one important algorithm - *stochastic gradient descent* (SGD) (R. Sutton & Barto, 2017).

SGD algorithm uses well known gradient descent method to optimize the loss function value with regard to the weight vector. The main assumption of SGD is that gradient is an expectation, and it can be approximately estimated using a small set of samples. The idea of this modification is to use a *minibatch* $\mathcal{B} = \{x_1, x_2, \ldots, x_{m'}\}$ of examples on each step. The size of the minibatch m' is usually held fixed as the training set size grows. The estimate of the gradient is then:

$$G = \frac{1}{m'} \nabla_w \sum_{i=1}^{m'} L(x, y|w) \tag{2.10}$$

where L - is the loss function being minimized.

To apply SGD to deep neural networks another important concept needs to be introduced. In earlier days the deep neural networks were almost intractable to train - until the backpropagation algorithm was introduced (Nielsen, 2015). This algorithm allows to assess errors on every layer of a multi-layered network, and hence to get its gradient for the purposes of changing the weight vector.

2.3.1 Architecture

There are different types of ANN architectures. One of the features that may be used to distinguish between different classes of neural networks is the existence of *feedback* loops within the graph structure of the network. Models which feature feedback loops that feed some outputs within the network back into itself are often called recurrent neural networks (RNNs), while strictly acyclic architectures are referred to as feedforward neural networks (FNNs). These names come from the informational flows within the network: FNNs featuring direct computation of output value y from input x via some possible intermediate transformations f, while RNNs also reuse the hidden layer outputs as inputs for these same transformations.

Neural networks are called *networks* because they may be represented as compositions of several different functions. It may be said that these functions are chained, e.g. $y = f(g(z(x)))$. This kind of chaining naturally suits FNNs with the function composition represented as an acyclic directed graph. In this case it is said that z is the first layer, g is the second and so on (Goodfellow et al., 2016).

The short description of the layer types that are of interest to this thesis is given in this section.

Fully connected layers

Often regarded as basic neural networks, fully connected (FC) layers[4] feature connections from every input to every single neuron in this layer. This powerful structure is regarded as an enhancement of the basic idea of a perceptron, and several layers of this type are able to distinguish even non-linearly separable data (Cybenko, 1989).

FC layers often feature a number of units with sigmoid activation function, hence being logistic neurons. To approximate non-linear data, the linear approximation models may be applied to the non-linear transform of inputs (e.g. via a well-known kernel trick (Hofmann, Schölkopf, & Smola, 2007)), that may itself be learned by the model with the help of non-linear activation functions.

A multilayer perceptron (MLP) architecture is a prominent example of usage of fully connected layers in neural network architecture. The MLP consist of several FC layers, and is proven to successfully classify non-linearly separable data.

[4] also referred to as *dense*

Due to the specificity of their architecture, FC layers usually feature large quantities of parameters because each connection has a weight associated with it, so the overall number of parameters is $N_{fc} = \|x\| \cdot q_{fc}$, where $\|x\|$ is the number of inputs, and q_{fc} is the number of units in this layer.

Convolutional layers

FC layers are a powerful tool that may be used for approximation of different kinds of functions. However, they do not take into consideration any spatial relations between incoming data. To account for that convolutional neural networks (CNNs) were introduced. Based on the organization of animal visual cortex, this type of architecture binds each individual neuron only to a small region of the input sequence - be it a part of timeseries data, a region of a 2D image or other grid-like input entity.

That region is called a *local receptive field* for a neuron, and is one of the main features applied in CNNs. Consider an image 25x25 pixels, with each pixel being an input for a CNN. Then, if each neuron has a receptive field of 5x5 pixels and we set a *stride*[5] to 1 pixel, then we end up with 21x21 grid of convolutional neurons. An example of a local receptive field is shown in Figure 2.5.

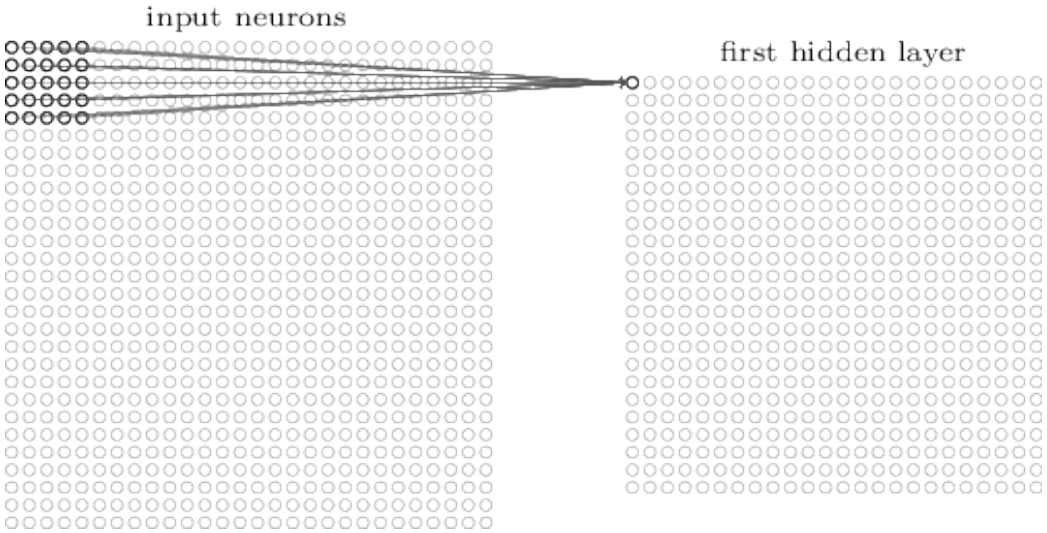


Figure 2.5: Local receptive field. Adapted from Nielsen (2015)

[5] step between receptive fields

Another important idea for CNNs is weight sharing. For each of these 21x21 neurons same weights are used. This 5x5 weight matrix is called a *filter*. This way, the output of each neuron of the first hidden layer is:

$$z_{i,j} = g(\sum_{k=0}^{4}\sum_{l=0}^{4} x_{i+k,j+l} \cdot w_{k,l} + b_{i,j}) \tag{2.11}$$

where $z_{i,j}$ is the output of the (i,j) hidden layer neuron, $w_{k,l}$ are the filter weights, $x_{i+k,j+l}$ is an input value for this neuron and $b_{i,j}$ is its bias.

There may exist a number of different filters, each having its own weight matrix. This allows to find similar shift-invariant patterns in the input data.

The combination of the filter and the mapping from the input layer to the hidden layer is often called a *feature map*. Because of the ability to accommodate different filters, it is also possible to have multiple feature maps available.

Long Short-Term Memory

Until this point we covered FNNs. These structures are powerful, but they lack the necessary support for sequential computations. FNNs do not preserve any state information, and that limits their usefulness. To circumvent this, the capability to store and reuse previously learned information was added with the usage of RNNs. This type of networks provides the necessary framework for storage of and learning not only from the data itself, but also from the sequence of inputs.

One of the most prominent RNN architectures is called *Long Short-Term Memory* (LSTM). Introduction of LSTM helped to resolve a severe problem that was crucial to development of RNNs: vanishing gradient problem. Because of the constant reuse of weights in RNN structures, gradients flowing "backward-in-time" tend to either *explode* or *vanish* because their temporal evolution exponentially depends on the size of the weights. However, LSTM introduces a new structure for neurons, that enforces *constant* gradient flow through every unit (Hochreiter & Schmidhuber, 1997). This structure allows to superimpose new experiences thus saving them inside the cell.

The architecutre of LSTM cell is shown in Figure 2.6. It consists of a linear memory cell, an input gate, an output gate and a forget gate.

The memory cell ensures the constant error flow (so called *constant error carousel* (CEC)) that handles the problem of vanishing and exploding gradients. The input gate protects the contents of the cell from being influenced by irrelevant experiences. The output gate protects other units from influence of currently

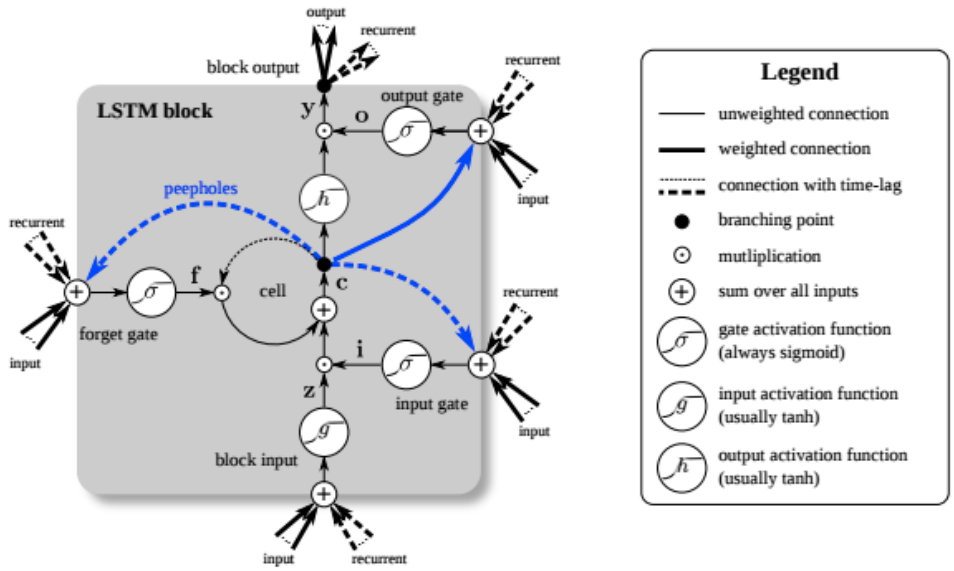


Figure 2.6: LSTM cell architecture. Image courtesy of *A Beginner's Guide to Recurrent Networks and LSTMs* (2017)

irrelevant error stored inside this cell. The forget gate (Gers, Schmidhuber, & Cummins, 1999) facilitates forgetting data that is no longer necessary or harmful. Each of the gates has its own set of weights, and learning those is the goal when training LSTM layers.

LSTM is capable of handling long and short time lags, handle noise for longer lag problems, and is *local in space and time*. This means that the update complexity does neither depend on the size of the network (*local in space*) nor on input sequence length (*local in time*). In this LSTM is unlike full backpropagation-through-time (BPTT), another popular algorithm for RNNs, which expands recurrent units into a sequence of connected feed-forward neurons depending on the input sequence length

Other

There are multiple ways to create a layer inside a neural network. Some of them are applicable only to RNNs, being an enhancement, an update or a based on already presented structures like BPTT(Werbos, 1988) or SAM (Rae et al., 2016)), while some are extensions or additions to FNNs (max-pooling layers (Weng,

Ahuja, & Huang, 1992)). Structures like autoencoder (Ballard, 1987) are used in unsupervised learning to represent data in different ways for use inside the network.

An attempt was made to combine different layer types and architectures to create a single universal model (Kaiser et al., 2017). Furthermore, a number of different approaches used neural networks as generic function approximators for reinforcement learning, some of which are covered in the next section.

2.4 Reinforcement Learning with Neural Networks

The field of reinforcement learning experiences the reinvigoration since the introduction and success of Deep Q-Learning (Mnih et al., 2015). Application of deep learning algorithms to RL and therefore control problems achieve significant successes even outperforming humans in some tasks. This led to the rise of interest in deep learning for RL and to the new developments and enhancements that are useful in the context of control problems. In the following section we describe popular applications of neural networks to the reinforcement learning.

2.4.1 Deep Q-Network

In this approach a deep convolutional neural network is used to approximate the Q-value and function as a critic-network within the ADHDP system. Instead of *cost-to-go* function J, Q-learning makes use of an optimal reward function Q. These are fundamentally the same, with the only difference being the optimization method: where J needed to be minimized, Q needs to be maximized.

The structure of Deep Q-Network (DQN) is as follows: input layer (84x84x4 nodes), 1st convolutional layer (32 filters of 8x8 nodes with stride 4, rectifier nonlinear activation (relu)), 2nd convolutional layer (64 filters, 4x4, stride 2, relu), 3rd convolutional layer (64 filters, 3x3, stride 1, relu), fully connected layer (512 units) and an output fully connected layer (18 units). The cardinality of the last layer is defined by the amount of possible actions that may be performed in a system, so the network produces an action-value for each possible action (Mnih et al., 2015).

In this setup convolutional layers allow the network to perceive visual information - an image from a game screen - as an input. This serves as an important addition to the vanilla Q-learning algorithm. Another essential innovation is the

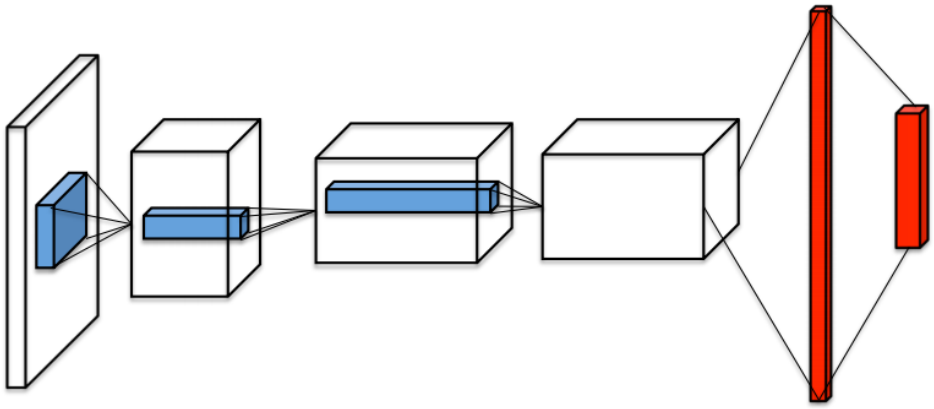


Figure 2.7: DQN architecture. Image courtesy of Wang et al. (2015).

usage of experience replay technique. Instead of learning from immediate experiences the network "remembered" them, storing a 4-tuple (s_t, a_t, r_t, s_{t+1}) as a memory fragment. Each state features a sequence of four images providing valuable data about the order of events. Then, after an action on each state is performed, a batch of random memory fragments are retrieved from the buffer of recent memories and used to train the Q-network.

Third valuable extension of the Q-network comes from the usage of a separate "target" network during training phase. Due to the shift of the weights intrinsic to the network being trained, the usage of a more stable "target" network with fixed weights helps to mitigate the possible divergence.

DQN-derivatives

Multiple modifications of simple DQN architecture followed. Two of the most prominent ones are: Double DQN(van Hasselt, Guez, & Silver, 2015) and Dueling DQN(Wang et al., 2015).

Double DQN modifies the initial concept by decoupling action choice from policy evaluation. Instead of using online weights of the policy network to choose next action, the "target" network is used to do that.

$$Y_t^{DQN} = R_{t+1} + \gamma \max_a Q(S_{t+1}, a; w_t^-) \tag{2.12}$$

$$Y_t^{DDQN} = R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; w_t); w_t^-) \tag{2.13}$$

While the update for DQN uses Equation 2.12, making action choice entirely dependent on the target network weights w_t^-, to evaluate action value, DDQN employs the separation, using greedy policy to pick an action, but evaluating it against target network anyway as seen in Equation 2.13. This allows to get rid of reported overestimations suffered by DQN systems (van Hasselt et al., 2015).

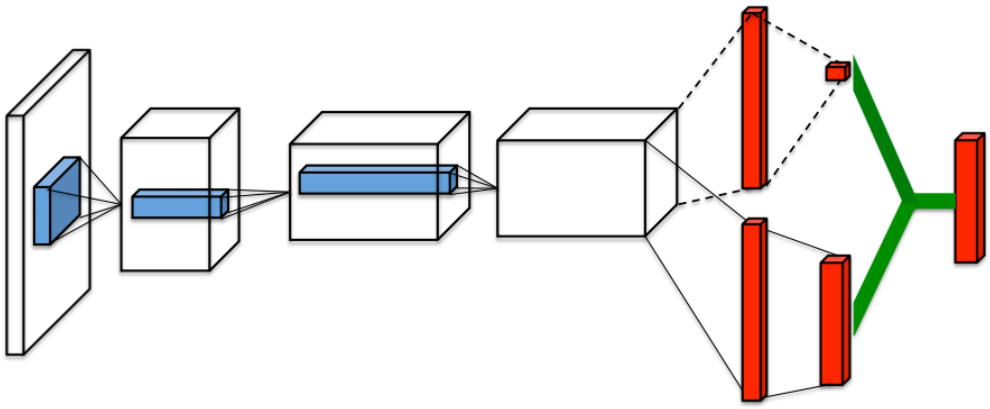


Figure 2.8: Dueling DQN architecture. Image courtesy of Wang et al. (2015).

Another notable modification of DQN is Dueling DQN. It makes use of the notion that Q-value $Q(s,a)$ is actually a composition:

$$Q(s,a) = V(s) + A(a) \tag{2.14}$$

where $V(s)$ is a state value ("how good is it to be in the state s") and $A(a)$ is action advantage ("how good is it to perform action a"). Dueling DQN architecture decouples state value and action advantage learning by breaking a penultimate network layer into two parts, only combining them back into Q-value at the final layer. Each stream is then learned independently, having their own weight vectors. This allows the dueling Q-network to update state value, influencing all actions in that state, not just a single preferred one. The existence of separate action stream allows for more robust Q-values with regard to noise (Wang et al., 2015).

2.4.2 Deep Recurrent Q-Network

DQN demonstrated an admirable performance on a number of different Atari games. However, it has its own shortcomings: it relies on a fully-observable game screen, has a limited amount of memory and requires a stack of several images

as an input. These shortcomings were addressed with the introduction of Deep Recurrent Q-Network (DRQN).

The architecture of this network is highly similar to that of vanilla DQN with two major differences: the penultimate fully-connected layer is replaced by LSTM, and the input will now take only one image instead of the sequence of four.

The introduction of the recurrent LSTM layer allows DRQN to perform in POMDP setting instead of a fully-observable MDP, as well as to cut input size (Hausknecht & Stone, 2015).

2.4.3 Asynchronous Advantage Actor-Critic

The next iteration of research following DQN successes concentrated on improving and flashing out the method. However, an issue of performance and efficiency, it took multiple days of training on GPU units to achieve a human-comparable performance for DQN (Mnih et al., 2016). The next logical step is to go parallel.

Asynchronous advantage actor-critic (A3C) algorithm presented by Mnih et al. combines several ideas into a powerful RL framework. The structure of A3C (as shown in Figure 2.9) supposes the existence of a global network that is being trained and a number of workers that execute training. In the beginning of each epoch every worker copies the state of the global network and then independently learns in its own environment, performing actions based on the current policy and accumulating losses according the RL principles. Once the epoch is about to end, the gradient of these losses is computed by every worker and presented for integration back into the global network. Thus, the actual learning happens simultaneously and in several different environments.

A3C makes use of actor-critic structure, not presenting a singular Q-value output, but rather producing a policy (actor) and a state value (critic) as a general output, perfectly corresponding to HDP structure presented in Section 2.2.1. Another notable addition is the use of policy entropy as a regularization term for policy training. Intuitively this emphasizes exploration, giving higher values to policies with multiple similar-valued actions and discouraging convergence to suboptimal deterministic policies.

A3C demonstrated robustness and good performance, even when trained for lesser periods of time, performing almost six times better then DQN after only 4 days training on CPU (Mnih et al., 2016).

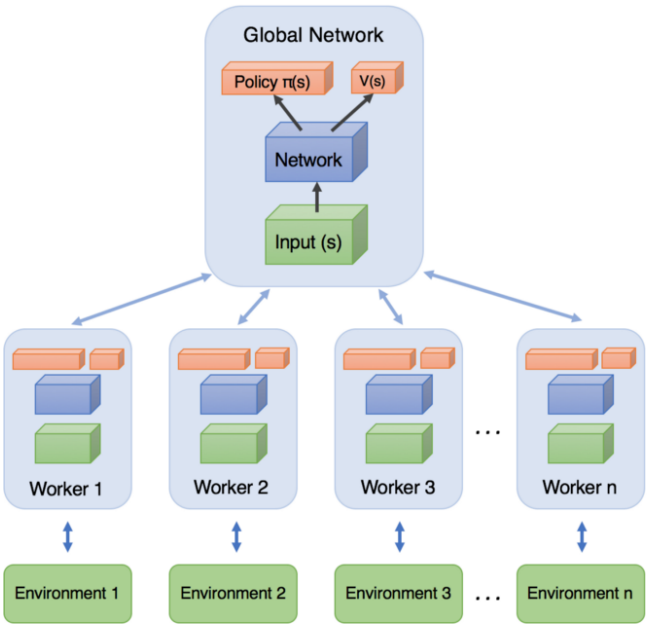


Figure 2.9: A3C architecture. Adapted from Arthur Juliani (2016)

2.4.4 Hybrid Reward Architecture

Most of previously introduced methods employed mostly "one reward" technique and did not make use of any domain-knowledge to try and reduce dimensionality of the state space. Nevertheless, this does not imply the impossibility of such attempts.

Another asynchronous approach is suggested by the authors of Hybrid Reward Architecture (HRA). They propose to decompose the reward function of the environment into n different reward functions, assigning each of them to a separate RL agent. However, because lower-level layers of the networks may be shared by multiple agents, it is rather said that this is a single agent with multiple *heads*, each producing action-values for the current state under a specific reward function (as shown in Figure 2.10)

HRA demonstrated a significant performance boost compared to DQN while tasked with a complex control problem. Provided with a reward decomposition into low-dimensional general value functions (GFVs), HRA manages to cut a state space size from 10^{77} positions to mere $1.4 * 10^7$ for a particular problem (Ms.

Figure 2.10: HRA architecture. Image courtesy of (van Seijen et al., 2017)

Pacman Atari game). HRA showed a notable performance improvement over both DQN and A3C in this context (van Seijen et al., 2017).

One disadvantage of HRA is that it usually requires some effort (and, possibly, domain knowledge) to provide the reward decomposition, while DQN learns from pixels. This makes HRA a more powerful, but less generic tool.

2.4.5 Stochastic Value Gradient

Most of the previously mentioned methods fall into the category of value-learning algorithms, as Q-learning, that is used as a base for all of them, is a value-learning algorithm. VL methods have an intuitive appeal and are fairly straight-forward to understand and implement. However, as discussed in section 2.2.1, value-gradient learning is sometimes more advantageous and efficient then VL.

The exploration of the application of concepts similar to ones used in DQN to VGL was performed by Heess et al. (2015). They introduce Stochastic Value Gradient algorithm, addressing two VGL limitations: 1) previous value-gradient algorithms could only learn deterministic policies and 2) learning environment, policy and value functions from agent-environment interactions only. Stochastic Value Gradient (SVG) framework allowed to learn a stochastic policy in model-based, as well as model-free methods in continuous domain.

Learning stochastic policy has several advantages over deterministic policies: it allows for on-policy exploration, it can be beneficial for partially-observable environments, and it allows for a more principled experience replay techniques (because probabilities are assigned to off-policy trajectories).

2.4.6 Feudal Networks

Another novel approach was used by Vezhnevets et al. (2017) with their FeUdal Nework(FUN) architecture. They proposed to employ hierarchical reinforcement learning, with several agents training in parallel.

Specifically, FUN combines two agents: Manager and Worker. Both are represented as RNN heads on top of a common lower-level CNN with a fully-connected layer (almost as authors of HRA proposed), but function in a slightly different manner. Manager operates on a lower time frequency than the Worker does, and it's main task is not to control the Worker per se, but rather set a goal vector, state that it will try to achieve. At the same time Worker determines primitive actions based on currently set goal and the output of the policy it trained. In such cases Manager is said to produce *transition policy* that governs general state transitions, while Worker handles primitive actions to handle successive states.

Manager's policy usually generates a much larger output vector (frequently accounting for a concrete state within the state-space) than Worker's one (only producing as an output a value from the much smaller action-space). FUNs show significant progress compared to simpler RL NN structures providing a framework that supports an individual sub-goal specifications and thus separation of complex behaviors into simpler ones (Vezhnevets et al., 2017).

2.5 Related Work

2.5.1 Inverse Reinforcement Learning

The question of *inverse reinforcement learning*, and specifically its cooperative variation, is directly related to the topic of this thesis. The basic idea behind cooperative inverse reinforcement learning (CIRL) is to align the agent's reward function with the same of the user (Hadfield-Menell et al., 2016). Proposed framework tries to align these using one assumption - whenever a reward is received, it is received by both agent and user identically. For this thesis, however, we use the reward functions that are different for both user and agent, and so, despite having POMDP in the core and cooperative learning as a goal CIRL is not entirely applicable to our use case.

Another approach to supplying the system with explicit expert feedback was proposed by Akrour, Schoenauer, Sebag, and Souplet. They described the "Programming by Feedback" system that learns from the preference judgments given by the expert and provided its proof of concept implementation. One of the key

features offered by this system is an ability to handle unavoidable user incompetence and survive mistakes during learning phase. What is different in our approach is that our system does not learn form the user feedback judging sets of different solutions, but rather consequentially learns from user's actions.

2.5.2 Preference Learning

As we are investigating systems with implicit user feedback, we are bound to be influenced by the user's preferences. An entire subfield of reinforcement learning emerged to deal specifically with this kind of tasks - preference-based reinforcement learning. First steps into this field where performed with the introduction of qualitative preference learning system (Fürnkranz et al., 2012). Its main objective is to augment the conventional RL algorithms with qualitative data from the expert, thus providing additional learning data and making use of preference learning techniques such as label ranking in the process. One of the recent examples of enhancement and refinement of previously mentioned system also comes in a form of a deep learning toolkit for preference learning (Christiano et al., 2017).

2.5.3 Building Control Problem

For this thesis we chose the Building Control Problem (Georgievski et al., 2017) as a basic use case. Therefore we consider it necessary to give a brief introduction into this field as well.

The Building Control Problem (BCP) addresses the necessity to minimize resource consumption while keeping comfort levels uncompromisable. With the large amount of different types and ranges of data it is intractable to analytically predict a model for this kind of environment.

Previous research shows the possible applications of negative feedback systems in real-world building control environments Setz, Nizamic, Lazovik, and Aiello. The described system optimized the sleep timeout of idle workstations. The idea was the following: sleep timeout should be short enough for the computers to shut down during non-working hours, while being long enough not to bother people using workstations during office hours. It demonstrated that negative feedback might be instrumental in achieving two goals simultaneously: energy efficiency and user satisfaction.

Automated machine learning orchestrator systems for environment control has been successfully developed before: Google's DeepMind used a neural network

based system to control environment and power consumption inside Google data-centers (Jim Gao, 2014), reducing the total consumption by up to 40%. Application of machine learning is effective in this case due to abundance of monitoring data and complexity of interactions and feedback loops involved.

Nonetheless, none of the previously conducted machine learning researches in the BCP domain took into account the CIRL task. Cooperatively learning user's preferences from his behavior, while still considering consumption has not been done before. Therefore, this task is eligible as a use case scenario for this thesis.

Chapter 3

Problem Description

This chapter covers in detail the design of the model that will be used throughout this thesis. In the first section we discuss the selected scenario and the reasons behind this selection. In the second section we describe the agents that will be evaluated and the reasons for their selection. In the third section the description of the MDP model that will be used for reinforcement learning is provided, while the second section contains data about specific RL algorithms used, as well as hyperparameter values used to configure the underlying neural networks.

3.1 The Problem

The main topic we intend to explore in this thesis is the performance of different reinforcement learning algorithms under the conditions of scarce negative feedback with regard to continuous control tasks. Hence, the research questions we strive to answer may be formulated in a following fashion:

1. How well do different reinforcement learning algorithms perform of a continuous control task with negative feedback?
2. Does the availability of historical data or regularization provide direct advantage?
3. What effect does reward function decomposition have on the system performance?

To investigate these questions we need a strictly defined model. For that purpose we take the BCP scenario, which is covered in detail in section 3.3. In the

next section we describe agents that are evaluated to answer the research questions postulated.

3.2 Agents

In this section we describe agents that were picked for evaluation, discuss the reasoning behind this, inner architecture and expected performance. General descriptions of the algorithms used may be found in Chapter 2.

As the foundation we use *RL4J* library from the *deeplearning4j* project. It provides implementations of all necessary algorithms and defines an interface that may be used to introduce a custom MDP for the algorithms to learn from and interact with. *RL4J* is used in conjunction with the *Scala* JVM language, that provides a flexible multiparadigm approach to model creation and data processing.

3.2.1 Agent Selection

While defining what algorithms should be used in this thesis we set up two main criteria. First, the algorithm should have wide reception and use neural network in its core. Second, the implementation of the algorithm should be readily available for use in any environment.

Name	Asynchronous	Recursive	Based on
A3C	+	-	A3C
A3C RNN	+	+	A3C
ANQL	+	-	Q-learning
DQN	-	-	Q-learning
DDQN	-	-	Double Q-learning

Table 3.1: Overview of the algorithms

The overview of the selected agents is presented in Table 3.1. We selected A3C algorithm as a modern industry standard in reinforcement learning field, and its recurrent variation A3C RNN to consider RNNs as well as FNNs in our system. DQN and its modification DDQN were selected due to their importance to the field: DQN is the first successful deep RL algorithm, while DDQN is its most prominent modification(van Hasselt et al., 2015). ANQL was chosen to test the asynchronous modification of DQN.

A3C

As a de-facto industry standard[1], *asynchronous advantage actor-critic* algorithm (Mnih et al., 2016) is now widely used as a basic reinforcement learning agent.

The basic stochastic gradient descent variation of the algorithm is utilized. While running A3C, 8 worker threads are used. Training occurs in three different modes: normal mode (2880000 iterations, 7200 in an epoch, 50 epochs), medium mode (4000000 iterations, 7200 in an epoch, 69 epochs) and long mode (5760000 iterations, 7200 in an epoch, 100 epochs).

We expect this agent to perform faster then the other agents, while learning an optimal set of actuators that corresponds to minimal necessary set for occupant's satisfaction, but failing to account for time sequences and thus "forgetting" to minimize consumption during out-of-office hours.

A3C RNN

Same algorithm as provided in the previous section with one major adjustment: the output layer of the neural network is an LSTM layer instead of a fully-connected one. Otherwize network and algorithm parameters are the same.

A3C RNN is expected to learn significantly longer than A3C, as LSTM needs to perform recursive backpropagation updates. However, it is expected that this agent will learn to better account for time sequencing, and thus outperform A3C reward-wise.

ANQL

The *Asynchronous N-step Q-Learning* is an asynchronous edition of a famous Q-learning algorithm (Mnih et al., 2016). It makes use of ε-*greedy* policy, essentially artificially introducing a degree of exploration to the basic Q-learning approach.

We set the amount of steps necessary to go through the exploration phase $t_\varepsilon = 1000$, and minimal degree of exploration used throughout the learning process $\varepsilon = 0.1$. Hard target Q-network updates happen every 100 iterations. ANQL is run using 8 worker threads. We use the same training modes as were used for A3C and A3C RNN.

We expect this algorithm to demonstrate significant performance increase with respect to DQN, but perform slightly worse then A3C. However, due to ANQL employing ε-greedy policy, we expect it to diverge more often then A3C.

[1] As seen in OpenAI default agent implementation (*OpenAI universe-starter-agent*, 2017)

DQN

When first introduced *deep Q-Network* revolutionized the reinforcement learning field, showcasing capabilities of the deep neural networks in reinforcement learning (Mnih et al., 2015). It uses the experience replay approach, using the stored "memory" to train the network instead of the current action-observation pair.

We configure DQN training modes as follows: normal mode (360000 iterations, 7200 in an epoch, 50 epochs), medium mode (540000 iterations, 7200 in an epoch, 75 epochs) and long mode (720000 iterations, 7200 in an epoch, 100 epochs). We set the size of stored history to 360000, and batch size to 32. Hard target Q-network updates happen every 500 iterations. As DQN also uses ε-greedy policy, we set its parameters to $\varepsilon = 0.1$ and $t_\varepsilon = 1000$.

DQN is expected to learn quite longer then A3C or ANQL, but its derived policy is expected to demonstrate comparable performance. Moreover, due to the fact that DQN learns in batches of random memories, it is expected to diverge less often then both A3C and ANQL.

DDQN

The *Double Deep Q-Network* algorithm is a refinement and improvements over DQN (van Hasselt et al., 2015). It is configured in exactly the same way DQN is, all differences are exclusively internal.

We expect DDQN to learn faster and in a more stable way then DQN does. We still expect A3C RNN to outperform it, because both DQN and DDQN do not provide any specific ways to account for the event order.

We use the similar hyperparameters and neural network designs for all of the described agents, so the effectiveness and performance of the algorithms may be assessed without regard for the underlying architecture. All networks are composed of 5 fully connected layers of 16 nodes each. The learning rate is set to $\eta = 0.0001$, and future reward factor $\gamma = 0.99$.

3.3 Environment

We define the environment for this study from the standpoint of a concrete use case for reinforcement learning algorithms in continuous control setting. This use case should be easy to convert to both MDP and POMDP models, as well as provide a number of ways to control it and to extract information necessary for training of a continuous control system.

The *continuous control* system here is defined as a system that operates with a small time step and does not have a proper terminal state, e.g. heating control in an office is necessary every minute of every day since the moment that this location was created and until the time that the office is no longer in use. Despite the fact that the terminal condition here exists in a form of "office would not have any more occupants", this is the state that is absolutely independent from the actions that the controlling solution may undertake.

One evident use case of a continuous control system is the solution to the well-known building coordination problem, which sets two targets: user comfort and energy savings (Georgievski et al., 2017; Setz et al., 2016). The solution to building coordination problem has to produce a control policy that minimizes actuator energy consumption while maximizing user comfort (i.e. minimizing the amount of negative feedback provided by user). It should operate given following elements:

- a set of environment parameters to control
- a set of actuators and sensors to influence and report the environment parameters values
- negative feedback sensors that report user attitude towards current conditions

Further in this paper we will use an MDP based on this concept: a location containing a set of actuators that allow us to reliably influence the environment while requiring certain amount of power to function, a set of sensors that report reliable information about the current environment, and a person that occupies a room and has a personal set of preferences with regard to environment parameters.

3.3.1 Environment Parameters

The environment parameters are a set of variables that are intrinsic to the system and cannot be controlled directly, but are easy to observe.

Definition 1. The *environment parameters set* is a tuple of non-controllable variables $E = (\mathcal{T}, \rho, I, \mathcal{E}_\nu)$, where $\mathcal{T}$ - is the temperature of the location, ρ - humidity, I - illumination and $\mathcal{E}_\nu$ - the amount of CO_2 contained in the air.

Definition 2. The *default value* $d_\mathcal{P}$ of environment parameter $\mathcal{P}$ is the value that this parameter assumes once no external influences are present.

This default value is set and does not change with time. Moreover, whenever an external influence changes the current parameter value, the environment will strive to return to the default value, given no further support for the change is provided. It does so using a specific default environment parameter behavior.

Definition 3. The *behavior* of environment parameter $\mathcal{P}$ is a function $G_{\mathcal{P}}(V_{\mathcal{P}}, d_{\mathcal{P}})$, where $V_{\mathcal{P}}$ is the current value of the parameter $\mathcal{P}$ and $d_{\mathcal{P}}$ is its default value, which outputs the parameter change during the next time step. We define several methods of parameter behavior:

- *immediate* - change occurs fully in one moment. Switching the desk lamp on and off is an example of the immediate change - the illumination level shift is instantaneous.
- *logarithmic* - the closer current value is to the target the slower the change occurs.
- *linear* - every tick the value changes for a specific delta, reaching the target in a steady manner.
- *stochastic* - every tick the value changes towards the target by a random step.

We assume that every change that happens to the specific parameter is supplied with a concrete behavior in mind. This way, if every non-controllable parameter $\mathcal{P}$ is associated with a tuple $(d_{\mathcal{P}}, G_{\mathcal{P}})$, where $d_{\mathcal{P}}$ is a default parameter value and $G_{\mathcal{P}}$ is the behavior of this parameter, then we have a way to specify default environment conditions, and the way to achieve these values given that external influence is not exerted anymore.

3.3.2 Negative Feedback

Negative feedback constitutes one of the main points of interest for this work, thus another important function of the model is to provide the networks with negative feedback values. In our setting the negative feedback represents the general measure of performance that is given to the system by its user. It may be perceived as "a general measure of happiness".

Definition 4. The *negative feedback provider* function $f_n(S)$, where S is a tuple representing current state of the MDP, is a classifier function that assigns to S class "1" if the negative feedback needs to be obtained and "0" otherwise.

The main purpose of the negative feedback function is to assess the current state and classify it as either "acceptable" or "unacceptable" and thus generate a negative feedback value if the environment is judged as being "unacceptable". We assign "acceptable" class to the negative feedback provider output of "0", and "unacceptable" to "1".

As was noted before, encompassing negative feedback into classification and other machine learning tasks is a complex endeavor (Muller, Muller, Marchand-Maillet, Pun, & Squire, 2000). Even in real world environments, specifically in cognitive experiments performed on mice, a large amount of negative feedback may lead to fear conditioning and cause helplessness in a learner (Landgraf, Long, Der-Avakian, Streets, & Welsh, 2015). In reinforcement learning setting this may lead to the model developing similar "fear" of action due to negative reward that might follow.

To apply the concept of negative feedback provider to our office-specific use-case, we introduce the notion of an office occupant. We model it as an entity, which has an associated set of preferred environmental parameters as well as a specific *presence profile*.

Definition 5. *Presence profile* is a function $P(t,o)$ that represents the probability that the occupant o is present at the office at time t.

The negative provider function for our usecase, (also known as *occupant utility function*) may be represented as:

$$f_n(S) = \begin{cases} 1, & \text{if } g(E_s, Pr) != P, \\ 0, & \text{if } g(E_s, Pr) = P. \end{cases} \tag{3.1}$$

where E_s is the tuple of current values of non-controllable environment parameters, and Pr - the tuple of preferred values of these parameters, specific to this concrete instance of a negative feedback provider, function g is an *action function*. This implementation of negative feedback provider yields 0 as if environment is "acceptable" and 1, if it is judged as being "unacceptable".

Preferences and Actions

In our usecase the office occupant may be able to change the environment to satisfy its preferred conditions. To do that, it must know what action to take - whether any of the environment parameters needs changing or not.

Definition 6. *Action function* $g(E_s, Pr, \mathcal{A})$ is a function that assesses the current state environment E_s with regard to preferences Pr and outputs a target $a \in \mathcal{A}$ that needs to be achieved in order for environment to be in line with the preferred conditions.

The set $\mathcal{A}$ consists of $n+1$ targets, where n is a number of non-controllable parameters. Each entity has a "*change*" target $Ch(E)$ associated with it. For the situation when all parameters are within expected ranges, the "*patience*" target P is included. It represents the correspondence between current and preferred environment parameters. For the building control scenario we set $\mathcal{A} = (P, Ch(T), Ch(\rho), Ch(I), Ch(E_v))$, including the patience target as well as a change target associated with every environmental variable among possible targets.

Definition 7. *Parameter deviation* for variable E with regard to preference Pr is expressed as: $D(E, Pr) = \frac{(E-Pr)}{Pr}$.

We define several action functions based on the following assumption: there has to be a strict function that always requires the change once any parameter has any non-zero deviation, there has to be a function that does not produce a change target regardless of available parameter deviations, there has to be a function that outputs a change target following a specific rule, and a function behaves stochastically. Thus, these are the following:

- *conservative* - pick the target with the largest deviation from the set ($\forall e \in E$: if $D(E_s(e), Pr(e)) \neq 0$ then $Ch(e)$, otherwize P)
- *enduring* - always output target P, disregarding any parameter deviations.
- *normal* - pick the target with the largest deviation from the set ($\forall e \in E$: if $D(E_s(e), Pr(e)) > Tr$ then $Ch(e)$, otherwize P), where Tr is a static *action threshold value*.
- *stochastic* - pick random target from the set ($\forall e \in E$: if $D(E_s(e), Pr(e)) > z_e$ then $Ch(e)$, otherwize P), where z_e is a random number

The experiments were performed under the assumption that the modeled system has an action function that conforms to one of the defined types.

Time and Presence

Another important feature of the office environment control use case is the unequal importance of different features at different moments in time. For example, in a real world situation, if the person that occupies the office is not present at work, all available actuators shall be turned off to preserve energy and minimize consumption, while if the person is present, the priorities shift, and the system should comply with the occupant's preferences regarding their immediate environment.

This means that time of day and presence play an important role in the proposed MDP. For our model we use a discrete time scale with a small step. As the default values we use a full time scale of one day and a time step of one minute, therefore we have 1440 time steps to make one full cycle. The information about current occupation of the office is provided via presence profiles.

As was shown, the presence profile provides useful data for learning and making predictions - and therefore controlling the user environment. It may be computed as a probability function over an extended period of time, e.g. using the motion detection and PC usage data for the same weekday over several month (Setz et al., 2016). In this work we use the following types of presence profiles:

- *always present* - $P_{ap}(t) = 1$. This represents the occupant being present at the location at all times.
- *empty* - $P_e(t) = 0$. This represents the fact that location is never used.
- *present during working hours* - $P_{wh}(t) = \begin{cases} 1, & \text{if } t > t_{start} \ \& \ t < t_{end}, \\ 0, & \text{otherwize.} \end{cases}$. This represents that the location is always occupied during office hours.
- *stochastic presence during working hours* - $P_{swh}(t) = \begin{cases} z_t, & \text{if } t > t_{start} \ \& \ t < t_{end}, \\ 0, & \text{otherwize.} \end{cases}$ where $z_t \in [0,1]$ is a random number. This represents that the location may be occupied during office hours, however an occupant is constantly leaving the location.
- *normal presence during the day* - $P_n(t) = f(t|\mu,\sigma^2)$, bound to interval $[0,1]$, where f is a probability density function of the normal distribution, and μ and σ^2 are distribution parameters that may be specified by an expert. This represents that the occupant may start his presence at the location and end it at flexible hours.

3.3.3 Location

For our use case the environment is represented as an occupied location of some kind: an office, a room, a social corner, etc. The location usually has an owner, which takes the role of the negative feedback provider, a default environment setting and a set of actuators which allow to indirectly influence the non-controllable variables. For example, if the occupant wants to influence the temperature in its office, and there is a heating unit installed to be used precisely for these purpose, then the occupant shall turn this heating unit on to change the environment so it better corresponds to its preferences.

We model the surrounding environment as a tuple $E_0 = (\mathcal{T}_0, \rho_0, I_0, \mathcal{E}_{v0})$ of setpoints, and as mentioned in subsection 3.3.1, every entity has a parameter behavior associated with it. Therefore, if no constant change is present, the entity value will gravitate towards the default value. For example, if $I_0 = 1000$ lux, and the behavior type assigned to illumination is "immediate", then, if all the lighting devices in the location are turned off, the current illumination value will be 1000 lux.

Actuators

As described above, to influence the non-controllable environment variables in the building control use case, every location has a set of actuators associated with it. This concept is a necessary mediator between the control unit, regardless whether it is a trained network or an occupant action function, and the surrounding environment.

Definition 8. An actuator is a tuple $ac = (E_{ac}, \mathcal{V}, \mathcal{C}, \mathcal{A})$, where the "delta" tuple $E_{ac} = (\mathcal{T}_{ac}, \rho_{ac}, I_{ac}, \mathcal{E}_{vac})$ contains information about maximal possible change that this actuator can impose and support, the velocity tuple $\mathcal{V} = (\mathcal{V}_T, \mathcal{V}_\rho, \mathcal{V}_I, \mathcal{V}_v)$ contains information about parameter behavior under influence from this actuator, $C \in \mathcal{R}$ is a consumption cost value, $A \in [0, 100]$ is the activity level of this actuator.

There is a linear dependency between the activity level of the actuator, how much power it consumes and how much influence it exerts upon the surrounding environment. This denotes the possibility of gradual change of actuator activity instead of just providing a two state "on-off" action space.

Chapter 4

Evaluation

Once the environment setup is complete, and the choice of agents is made, we try to answer the postulated in section 3.1 research questions. To address these questions we devise a number of experiments. Each of these investigates one particular aspect of the specified research questions. In the following sections of this chapter we provide descriptions and results for these experiments. We also state their relevance to the research questions formulated and asked in this chapter.

4.1 Methodology

To perform any experiment in RL setting we have to go through several steps. First, we need to define the model that we will be using. This includes two main parameters: definition of the reward function, and definition of the agent's input parameters. These are specific to every experiment and vary depending on what specific kind of model the agents are evaluated against in this experiment. For example, set of input parameters for MPD and POMDP setting will vary, as partial observability requires some parameters to be inaccessible to the agents.

The next step is to train every agent against the selected model. We ran every agent for a preset number of iterations with the goal to optimize the reward function and accumulate the highest reward possible. During this time the exploration happens, and the optimizations actually make impact: e.g. ε-greedy policy is executed, and L_2-regularization influences the neural network inner weights.

As the last step we perform the evaluation of the trained agent against the testing model. Specifically, we run the agent ten times, taking note of the accumulated result at the end of each model run. The average of these ten results is the

accumulated reward value that represents the overall performance of the agent on this particular scenario.

4.2 MDP setting performance evaluation

Setup In this experiment we investigate the performance of different reinforcement learning agents. We assume that the generic model uses normal action function and strict presence profile as described in previous chapter.

As input parameters the agents receive: current accumulated consumption value, current presence probability, time, actuator activation, negative feedback values and a set of all environment parameter deviations.

We use the reward function of a following form:

$$\begin{aligned} R(s|M) = &- (k_c - P(s|M)) \cdot C_s \\ &- P(s|M) \cdot D(E_s, Pr(M)) \\ &- k_{nf} * f_n(s) \end{aligned} \tag{4.1}$$

where

$$\begin{aligned} D(E_s, Pr(M)) = D(T_s, Pr_t(M)) + D(\rho, Pr_\rho(M)) + \\ D(I, Pr_I(M)) + D(E_{\nu s}, Pr_\nu(M)) \end{aligned} \tag{4.2}$$

is the sum of parameter deviations, C_s is total consumption at state s, k_c is the consumption importance coefficient, k_{nf} is the negative feedback value. We set $k_c = 1.0$, and $k_{nf} = 6$.

This form of a reward function is used to account for several different reward components. The term $-(k_c - P(s|M)) \cdot C_s$ corresponds to the importance of consumption during the moments with lower presence profile values. Depending on the value of consumption importance coefficient, it may be disregarded whatsoever during time intervals when the presence is definite, or included in full extent if absence is evident. Term $-P(s|M) \cdot D(E_s, Pr(M))$ is responsible for taking user preferences into account and providing necessary information for full state observability. The last term corresponds to the effect that negative feedback has towards the composite reward.

As the plots accumulate the negative reward - similar to accumulating penalty - we use the inverted scale for figures which present the accumulated reward score. Higher value means worse performance.

Relevance and expectations The results of this experiment provide the necessary general outlook on RL agent performance with regard to the given task. We expect A3C RNN agent to perform better then the others, being able to the learn from the sequences of observations. Asynchronous A3C and ANQL agents are expected to perform worse then A3C RNN, but still better then DDQN and DQN, as the last two do not have any access to historical data and are only able to learn sequentially.

Results As can be seen from the learning reward accumulation plots presented in Figure 4.1, only some algorithms were able to complete the training without divergence. ANQL (Figure 4.1b) diverged toward the end of the training, and A3C RNN (Figure 4.1e) has started diverging as well. Both DQN and DDQN showed a reward spike in the second half of their learning period.

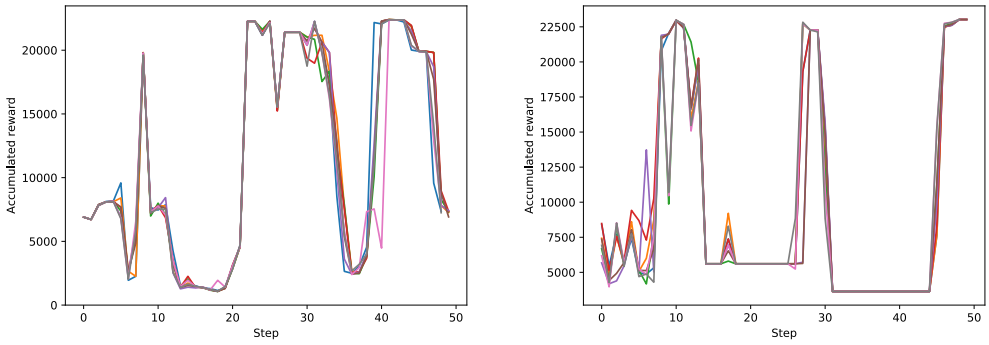
In Table 4.1 the results of the testing simulation run are given. Surprisingly, the best result was demonstrated by DQN algorithm. The evaluating simulation process is shown on Figure 4.2. A3C managed to find a policy that successfully discerned between moments of presence and non-presence, but did not manage to completely satisfy the occupant utility provider, still frequently going over negative feedback threshold. The ANQL agent completely ignored the negative feedback. DQN also learned a policy that was resembling that of A3C agent, succesfully learning the threshold value as well, while A3C RNN and DDQN both learned and maintained an optimal value throughout the testing run. We call it "optimal value policy" throughout this research paper.

Algorithm	Reward
A3C	-2134.39
A3C RNN	-5600.56
ANQL	-10998.53
DDQN	-2167.65
DQN	-1851.76

Table 4.1: Testing rewards for experiment 1

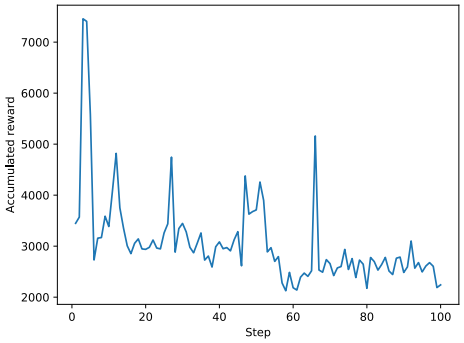
4.3 L_2 influence on learning

Setup Basic setup of this experiment is the same as the setup for experiment 1 with one major difference. In this experiment the L_2 regularization term is included into the agent's internal architecture. We set L_2 value to 0.001. Regularization modeling is achieved via the means provided by `dl4j` library.

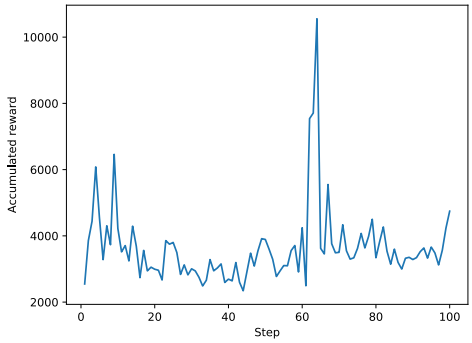


(a) A3C

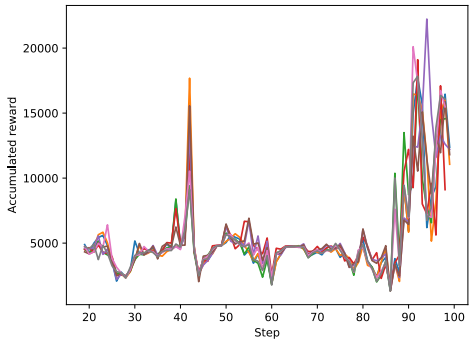
(b) ANQL



(c) DDQN

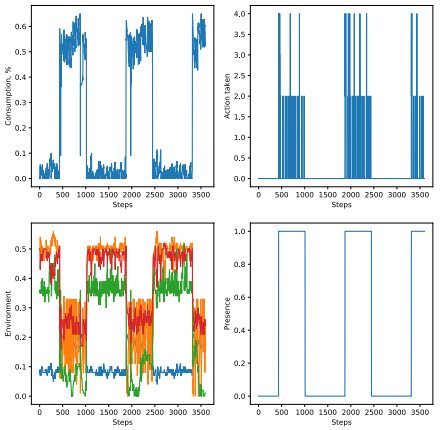


(d) DQN

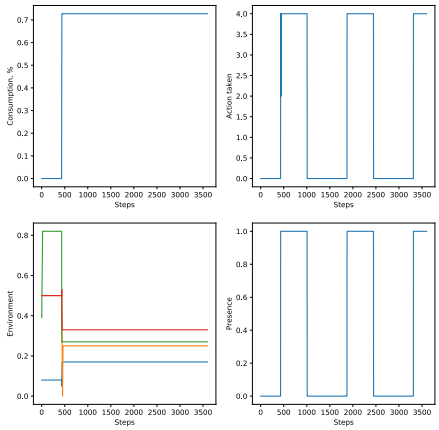


(e) A3C RNN

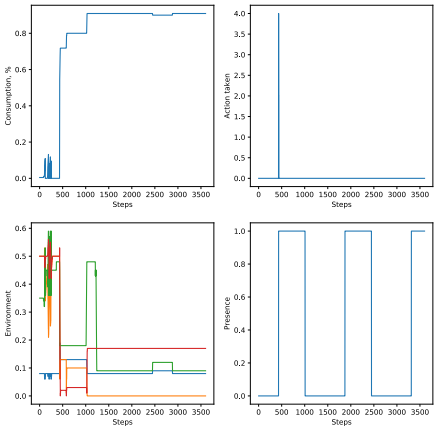
Figure 4.1: Accumulated reward over epochs with regard to strict presence profile and normal negative feedback utility function



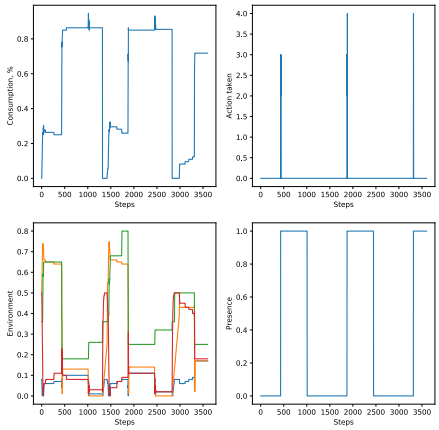
(a) A3C



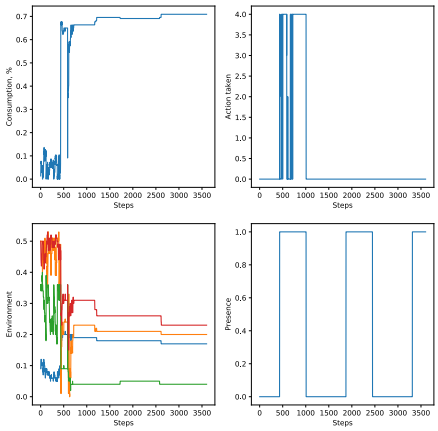
(b) ANQL



(c) DDQN



(d) DQN



(e) A3C RNN

Figure 4.2: Consumption, feedback action, environment conditions and presence with respect to the current time step during the simulation run

Relevance and expectations This experiment is meant to investigate the influence of regularization and thus weight decay techniques on the performance of our agents. It directly answers one of the stated research questions. We expect agents to generally perform better, due to the expectation that regularization will help to deal with previously observed divergence on the later stages of the experiments.

Results The accumulated reward and simulated run plots may be found in the appendix in Figure A.1 and Figure A.2. The quantitative value of accumulated rewards is also provided in Table 4.2.

Algorithm	Reward
A3C	-10998.53
A3C RNN	-1230.15
ANQL	-1767.69
DDQN	-10708.74
DQN	-1860.43

Table 4.2: Testing rewards with L_2 regularization

During this experiment A3C diverged nearing the end of the training phase. However, ANQL converged and ended up being second best agent, outperformed only by A3C RNN algorithm. DQN demonstated performance comparable to that during experiment 1, while DDQN diverged during the testing phase. Out of all agents only A3C RNN did not learn a constant-value policy.

4.4 Influence of historical data

Setup Basic setup of this experiment is the same as the setup for experiment 1 with one major difference. Instead of only supplying the latest observation as an input for the agent, we supply four latest observations. Thus, the length of the input layer increases 4-fold.

Algorithm	Reward
A3C	-10998.53
A3C RNN	-1409.95
ANQL	-2751.31
DDQN	-4487.80
DQN	-2337.14

Table 4.3: Testing rewards with historical data

Relevance and expectations The purpose of this experiment is to investigate influence of available historical data on learning process. This was used previously in learning tasks to help agents play Atari games - it was proven to be a useful addition for non-recursive agents to provide them with data about consequential parameter change (Mnih et al., 2015; Hausknecht & Stone, 2015).

We expect the non-RNN algorithms to benefit from the availability of historical data.

Results The results for this experiment are presented in Table 4.3 and in Figure A.3 and Figure A.4. During this experiment only A3C agent diverged. Unlike during previous experiments, ANQL learned a presence policy, while DQN demonstrated a result similar to that of experiment 1. A3C RNN agent surprisingly still performed better then all the others, both learning a presence profile and demonstrating highest accumulated reward.

Unexpectedly, the introduction of historical data did not benefit the training process. In both cases, however, both DDQN and DQN learned an expected behavior from the input data. Also unexpectedly A3C RNN was not able to cope with this task, falling into the "optimal value" behavior in the historical data case, while acting against negative feedback in the regular case. Both A3C and ANQL agents diverged during training.

4.5 Incomplete data performance

Setup In this experiment we investigate the performance of the agents in a POMDP setting instead of MDP. We disable their access to variables intrinsic to the occupant, and only report consumption, presence, time, actuator activation and negative feedback values.

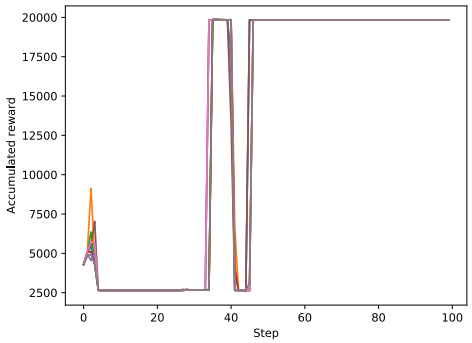
We use the reward function of the following form:

$$\begin{aligned} R(s|M) = &- (k_c - P(s|M)) \cdot C_s \\ &- k_{nf} * f_n(s) \end{aligned} \tag{4.3}$$

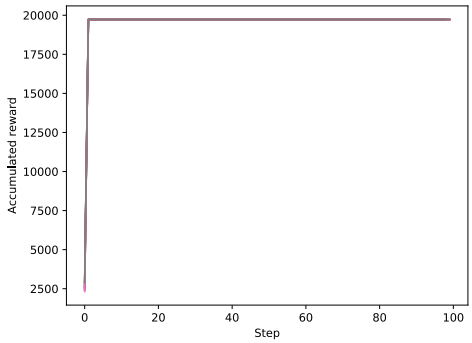
The terms included are the same as in Equation 4.1, with one exception: the term that corresponded to internal preference parameters deviation is not included. As inputs these values are not provided anymore as well.

For comparison we also provide the same experiment performed with historical data included. As in previous experiment, it contains data from 4 observations as inputs for the network.

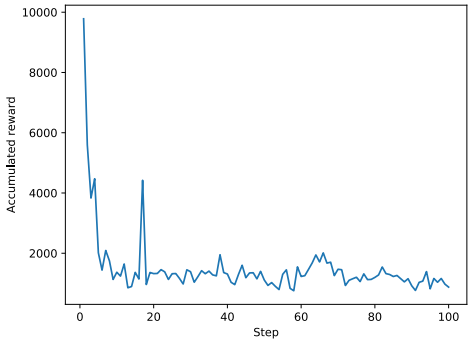
Relevance and expectations This experiment strives to examine the agent performance in partially observable setting with incomplete information. This is specifically the environment that RNNs are designed to handle, so we expect A3C RNN to show the best performance.



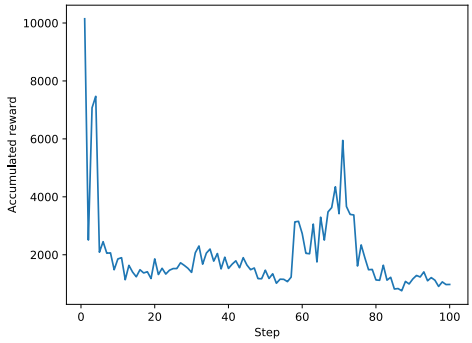
(a) A3C



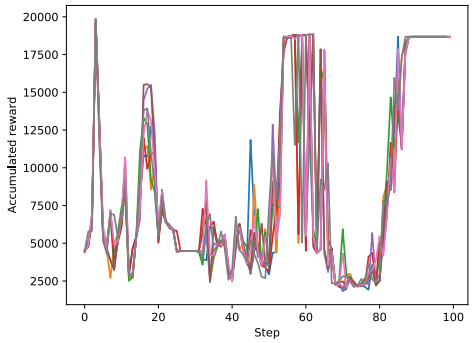
(b) ANQL



(c) DDQN

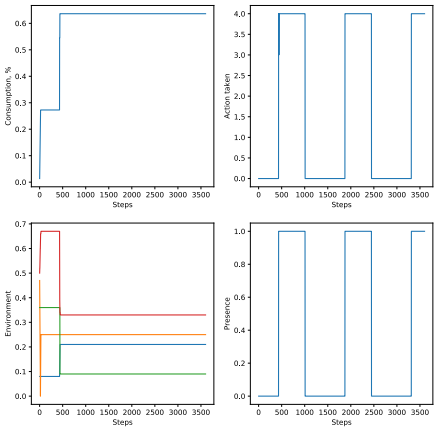


(d) DQN

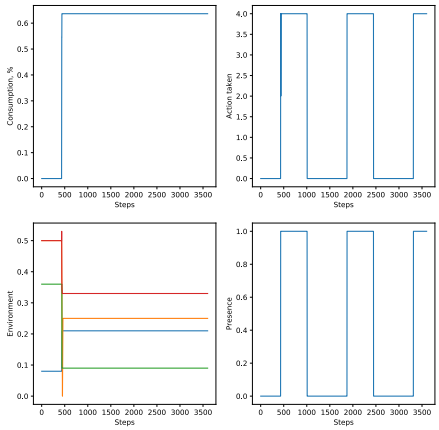


(e) A3C RNN

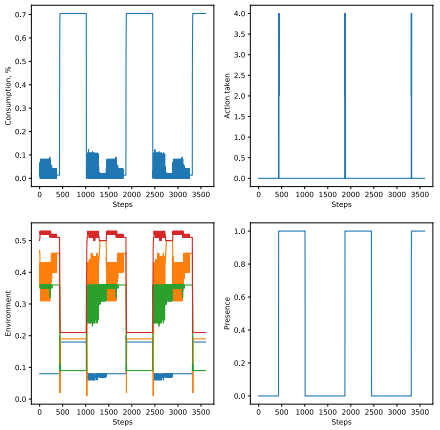
Figure 4.3: Accumulated reward over epochs with regard to strict presence profile and normal negative feedback utility function in POMDP setting



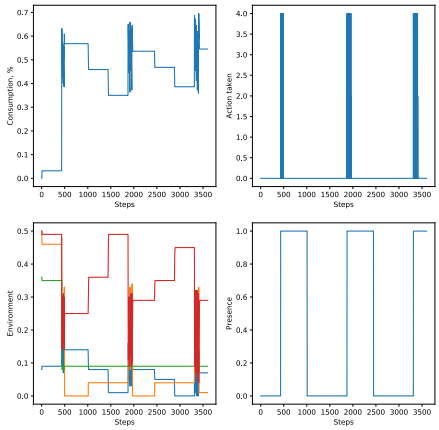
(a) A3C



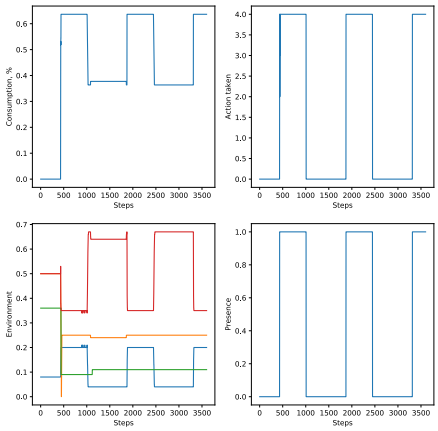
(b) ANQL



(c) DDQN



(d) DQN



(e) A3C RNN

Figure 4.4: Consumption, feedback action, environment conditions and presence with respect to the current time step during the simulation run in POMDP setting

Regular		Historical	
Algorithm	Reward	Algorithm	Reward
A3C	-9844.14	A3C	-9728.91
A3C RNN	-9262.27	A3C RNN	-9262.27
ANQL	-9728.91	ANQL	-9728.91
DDQN	-193.03	DDQN	-305.54
DQN	-872.53	DQN	-944.11

Table 4.4: Testing rewards for POMDP setting

Results The simulated rewards for a regular POMDP task are available in Table 4.4. In Figure 4.3 the training process accumulated rewards are shown, and the results of a simulated run are provided in Figure 4.4. We also provide results for the same experiment in historical setting in Figure A.5 and Figure A.6.

An interesting observation lies in the fact that A3C RNN, DDQN and DQN agents managed to better learn to distinguish between targets in POMDP setting then in MDP setting. However, both asynchronous agents A3C and ANQL only learned an optimal value policy.

4.6 Reward function decomposition

Setup We use the basic setup as described in experiment 1. However, we make some changes to the reward computation. We test the agents with different presence profiles - "always present" and "empty". This means that the reward calculations from Equation 4.1 is changed to R_1 for the "always present" profile and R_0 for "empty".

$$R_0(s|M) = -C_s - k_{nf} * f_n(s) \tag{4.4}$$
$$R_1(s|M) = -D(E_s, Pr(M)) - k_{nf} * f_n(s) \tag{4.5}$$

We simulate agent trainings for both MPD and POMDP setting with all the changes as previously described. This means that POMDP simulations do not have data about internal preferences as input data. Thus, in POMDP setting with R_1, the learning will only be based on the negative feedback component.

Relevance and expectations In this experiment we test the ability of our agents to learn to optimize for less complex reward functions in our setting. This is

done to determine whether there is a difference between agent learning from a composite reward function with different goals at specific environment states, and simpler rewards that only account for a specific goal at any given point in time.

We expect all the agents to be able to learn from Equation 4.4, and keep all actuators shut down for the duration of the testing run, both in MDP and POMDP setting. We also do not expect agents to experience any difficulties while learning the occupant preferences, though only in MDP setting. We expect POMDP to still be easier to learn for A3C RNN, and non-RNN algorithms may even diverge.

Results The results for this experiment are provided on Figure A.7, Figure A.8, Figure A.9, Figure A.10. The accumulated during the testing simulation rewards are shown in Table 4.5.

Algorithm	R_1 POMDP	R_1 MDP	R_0 POMDP	R_0 MDP
A3C	-21600.0	-24777.75	-108.62	*0.0*
A3C RNN	-1042.2	*-1713.86*	-118.72	-135.14
ANQL	-21600.0	-24777.75	*0.0*	*0.0*
DDQN	*-30.0*	-1791.14	*0.0*	*0.0*
DQN	*-30.0*	-2083.70	-252.14	*0.0*

Table 4.5: Testing rewards for decomposed reward functions R_1 and R_0

From the simulation data we make following observations: DDQN algorithm generally showed best results, consistently managing to learn policies corresponding to the current goal. DQN showed comparable performance, while both A3C and ANQL agents unexpectedly failed to learn any of the occupant preferences in "always present" test.

A3C RNN consistently showed convergence, oscillating near the negative feedback trigger threshold, or near zero activation values. It was the only agent that did not leave all the actuators inactive during R_0 MDP experiment (Figure A.7e), while showing best performance in a significantly harder to learn R_1 POMDP case (Figure A.10e).

Chapter 5

Discussion and conclusions

In this chapter we discuss the experiment results given previously. We try to explain and interpret them keeping in mind initial research questions, and note downsides and advantages of the model used throughout the experiments. We also discuss future work prospects and cover missing bits and pieces.

5.1 Discussion

One of the recurring trends evident from the simulation plots is the tendency towards learning a specific optimal value, which results into what we call an *optimal value policy*. We interpret this occurrences as agents learning that a specific set of actuations satisfies user preferences and leads to the minimal amount of negative feedback.

However, this leads to agents learning a specific consumption value and assuming that it is an optimal one, e.g. Figure A.2a, Figure A.2c. Nevertheless, same approach works for another set of agents, e.g. Figure A.2b, Figure A.2d. This behavior constitutes agents finding a plateau on the shape of the reward function - and never moving away from it, as evident by their reward accumulation plots, as shown in Figure A.1a and Figure A.11b. This leads to the divergence of some agents.

Interesting finding lies in the fact that asynchronous agents are more prone to learning optimal value policy. As a counterpoint to this behavior stands performance of DQN and DDQN agents - these manage to learn consistent policies: out of all experiments, DDQN produced a solution prone to large amount of negative feedback only once not managing to take L_2 regularization into account properly,

while DQN failed only on an experiment that featured a more complex presence profile with partial-observability (Figure A.11d). This corresponded to our expectations - DQN did not have proper toolkit to account for consequential timeseries data.

A3C RNN agent also supplies us with interesting observations. First of all, it still manages to perform fairly well with regard to reward function accumulation, but, as evident from Figure A.11e, Figure A.2e, Figure 4.2e it also learns to switch goals depending on presence value. Surprisingly, it was coping better in partially-observable setup then in a full MDP. RNNs are designed to handle POMDP setting better, but we expected the A3C RNN agent to still accumulate higher reward in the regular MDP setting. One possible explanation to this occurrence is the applicability of ”curse of dimensionality”. During MDP experiments the general learning space was larger, that during the POMDP experiments.

This leads us to our next set of observations. During the POMDP experiment, asynchronous agents again fell into the optimal value policy trap, while DDQN achieved the best performance, learning to avoid negative feedback almost entirely. From Figure 4.4c, however, we assume that it learned exactly what set of sensors should be enabled at each specific moment in time. During the same experiment DQN also learned to avoid negative feedback, however, failing to disable power consumption during moments when negative feedback was impossible. At the same time A3C RNN agent learned the presence profile shape, but failed to abide by provider's preferences. This might be due to the lack of exact understanding of preference margins - these were stripped in the POMDP context. At the same time DDQN and DQN did manage to attribute the reward gain to proper actions.

During the same experiment DDQN and DQN agents showed a generally better understanding of the context of the goal. This is unexpected, because less information was provided during this experiment. This may be due to the ”curse of dimensionality”: once agents had to learn in a less dimensional space, the probability of finding local optimum increased, as opposed to the situation with higher number of dimensions. During MDP experiment DDQN only learned an optimal value policy, which roughly corresponds to locating a plateau state.

To further investigate this phenomena, we tried to teach to the agents the constituent part of the reward function. In the MDP setting, all agents handled the task well, with a notable exclusion of A3C RNN - due to its nature and ”learning memory” it was oscillating near the 0 consumption line for the duration of its testing, as can be seen from Figure A.7e. In the POMDP setting only DDQN and ANQL agents learned to keep the consumption at zero point, with other algo-

rithms showing oscillations similar to those of A3C RNN in previous experiment.

In search for learning optimization we provided our agents with some additional data regarding the sequence of states it experienced. This was done to soften the influence of time separation of action and the consequent reward. It proven useful - only A3C agent still learned an optimal value policies, all other agents managed to find a proper way of avoiding negative feedback, while not sacrificing the consumption levels. This can be best seen from Figure A.4. It is evident from these plots, that both A3C RNN and ANQL were again oscillating near preference threshold - thus still invoking negative feedback reactions. DQN adapted to the availability of historical data the best, while A3C RNN again showed results that were closest to the corresponding presence profile.

Another point we addressed in this thesis was the introduction of weight decay in form of L_2 regularization was another point addressed. We expected that weight decay would help certain agents to stop diverging by reaching a plateau state, and that is indeed what happened - only A3C and DDQN agents diverged with L_2 in place. However, this lead to all other agents apart from A3C RNN learning an optimal value policy, probably because weight decay did not provide enough momentum for them to reach more optimal solutions. A3C RNN learned to resemble the required user preference while not sacrificing consumption, as seen from Figure A.2e. It learned user preference, but does not entirely avoid negative feedback - again oscillating around the threshold value, thus triggering negative feedback provider to act occasionally.

5.1.1 Limitations

We used implementation of the RL algorithms as provided by `dl4j` library. Due to this limitation, any errors or specifics of this library may have impacted the performance of the agents. As a concrete example - A3C implementation was prone to frequent failures, not being able to finish the full learning cycle. This fact is an important point to be addressed in future work.

5.2 Future Work

Multiples aspects of this thesis may be investigated in more detail. As the main point of this work was to give a brief overview of RL with applications of negative feedback learning and performance comparison of different RL agents, some topics still remain unexplored.

One of the possible research prospects is an attempt to exclude the influence of the chosen RL algorithm implementation on the results of this thesis, specifically allowing to circumvent the main limitation of this thesis. This may either be done by testing other available agent implementations, or creating an altogether new one.

In this thesis we did not go over all possible combinations of action function types and presence profiles, as described in chapter 3. This might provide better insights into the performance of RL agents and allow to test them against more unpredictable environments, e.g. stochastic action function.

We tested our agents on a simulation with a number of different assumptions. Specifically, in section 3.3, we defined several types of available action functions, parameter behaviors and presence profiles. These assumptions do not necessarily hold in a real-world scenarios. Therefore, one of possible further research opportunities lies in deployment and learning of the agents in the real environment. For example, presence profiles may be gathered and derived from the information provided by a presence sensor installed in an office. The system may then proceed with operating actuators, e.g. heating valves, AC units or light switches.

Another notable agent that is missing form our comparison is an Hybrid Reward Architecture (van Seijen et al., 2017). This agent should be very well suited for this kind of tasks, being able to learn from different reward functions. As shown in section 4.6, agents are generally capable of learning simple reward functions. This may prove very beneficial to HRA agent. This agent was not included into the test due to two reasons. First, it is very recent and thus there is no good implementation readily available. Second, HRA is not a single neural network, but rather an internal ensemble of several disjoint networks. In this thesis we only tested integral agents.

5.3 Conclusion

In this thesis we have investigated several problems regarding neural network agent performance on cooperative inverse reinforcement learning tasks. We were interested in three topics:

Performance Our experiments show that agents are prone to finding an *optimal value policy* in both MDP and POMDP setting. This way they avoid most of the negative feedback, but fail to accomplish other goals set to the agents. On the other hand A3C RNN and DQN agents showed good adaptability and potential of

learning a complex reward function. A3C RNN performed even better in POMPD setting compared to a fully-observable environment.

Reward Decomposition From experiments made with the decomposed reward functions it is evident that the agents adapt better to learning a simple reward function. It is not surprising, but serves as a good basis for further investigations based on HRA architecture, or agent ensemble learning.

Optimization We investigated influence of L_2 regularization and availability of historical data on the learning process. These do not lead to radical increase in efficiency, though with availability of these optimization components agents tend to learn to switch between expected goals better.

As for the use case that we have been studying, results are inconclusive. In most of our experiments we were able to successfully control the environment with respect to avoiding negative feedback, while keeping consumption optimal. DQN and A3C RNN were able to master the task in different conditions. However, even these agents diverged in some experiments, so we cannot claim that they were able to consistently control the environment.

There are still multiple open research question in the field of cooperative inverse reinforcement learning. In this work we analyzed general performance of most notable RL algorithms in this setting, both including full and partial observability, and impact of optimization techniques on their performance. There is still much work to be done to reliably solve these problems.

Appendix A

Extra experiment results

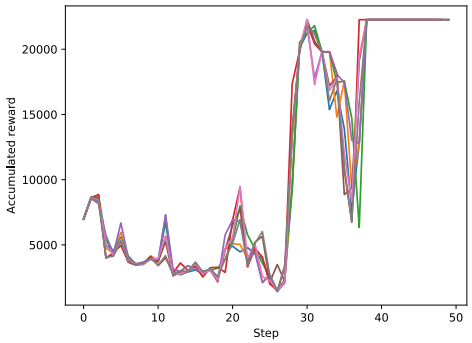
In this appendix we include some experiment results that were not included in the main text.

A.1 Performance on more complex presence profiles

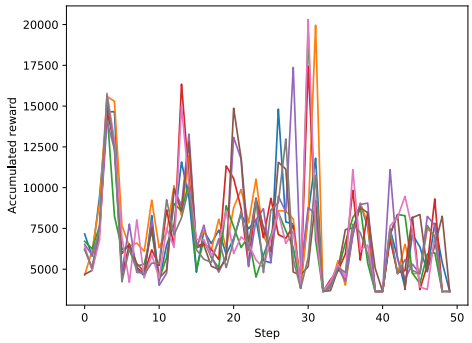
Setup Basic setup of this experiment is the same as the setup for MDP performance experiment with one major difference - we use ”normal presence during the day” presence profile instead of ”present during working hours”.
Relevance and expectations Expectations and relevance are similar to those of MDP and POMDP experiments.
Results The results are aggregated in Figure A.11 and Figure A.12. The accumulated reward over simulation runs are provided in Table A.1. From the table and plots it is evident that A3C RNN managed to learn the presence profile in POMDP setting, while all other agents learned an ”optimal value” policy.

MDP		POMDP	
Algorithm	Reward	Algorithm	Reward
A3C	-19394.62	A3C	-16938.12
A3C RNN	-19394.62	A3C RNN	-954.97
ANQL	-19394.62	ANQL	-534.03
DDQN	-2303.14	DDQN	-449.94
DQN	-2253.31	DQN	-1443.10

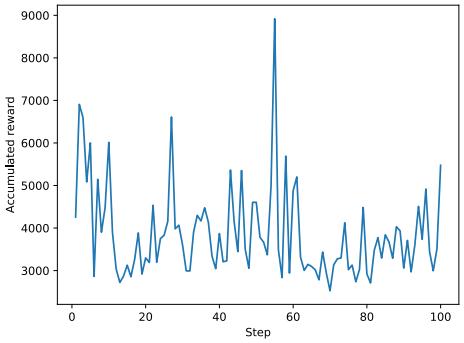
Table A.1: Testing rewards for POMDP setting



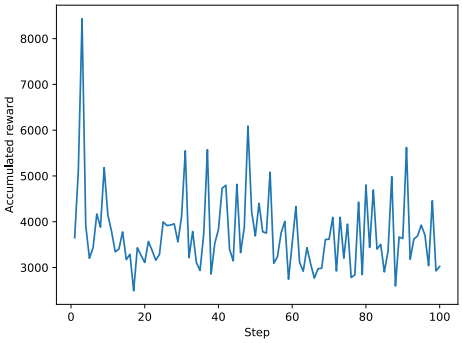
(a) A3C



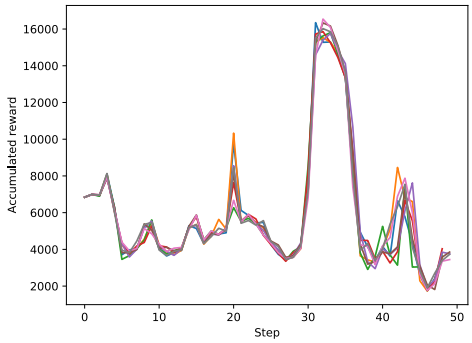
(b) ANQL



(c) DDQN

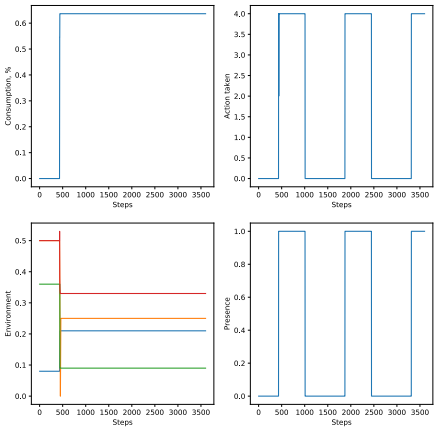


(d) DQN

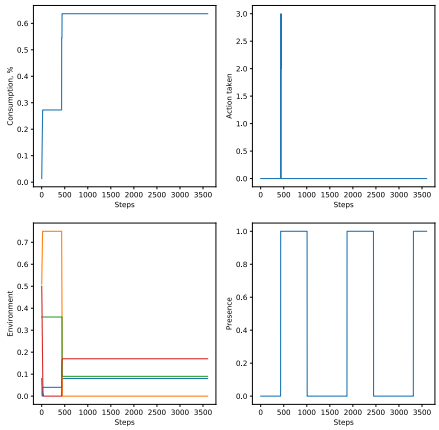


(e) A3C RNN

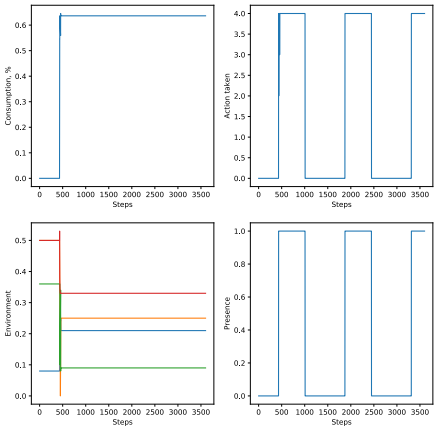
Figure A.1: Accumulated reward over epochs with regard to strict presence profile and normal negative feedback utility function using L_2 regularization



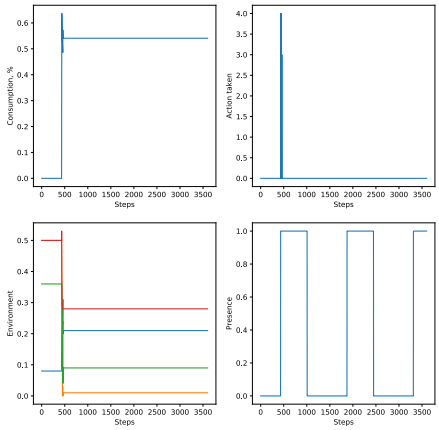
(a) A3C



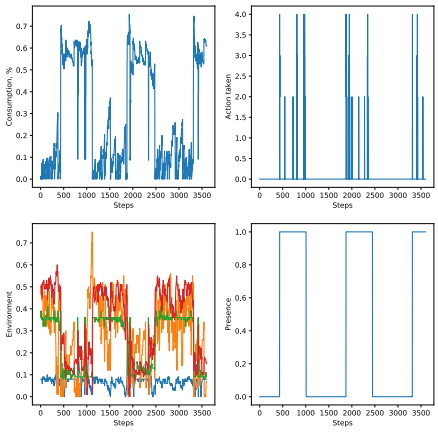
(b) ANQL



(c) DDQN

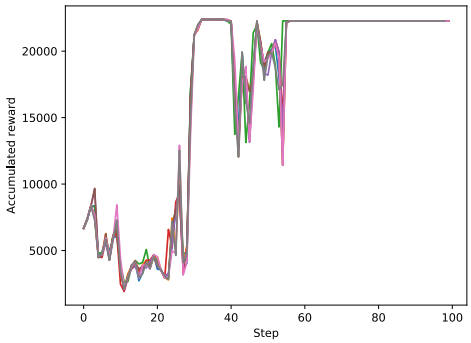


(d) DQN

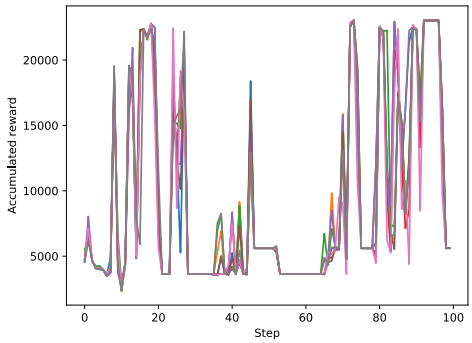


(e) A3C RNN

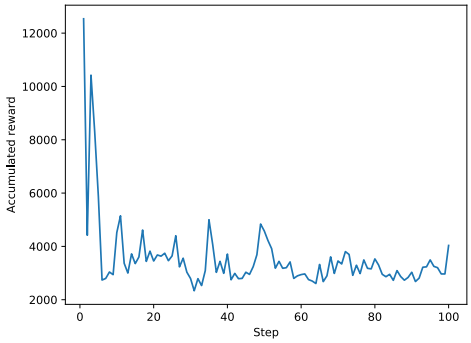
Figure A.2: Consumption, feedback action, environment conditions and presence with respect to the current time step during the simulation run using L_2 regularization



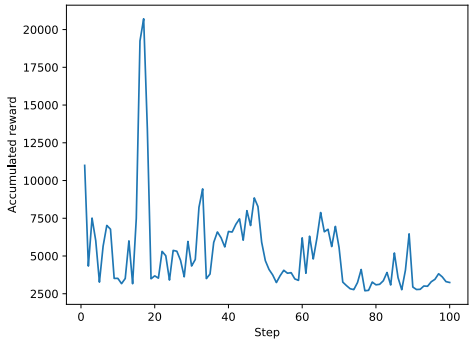
(a) A3C



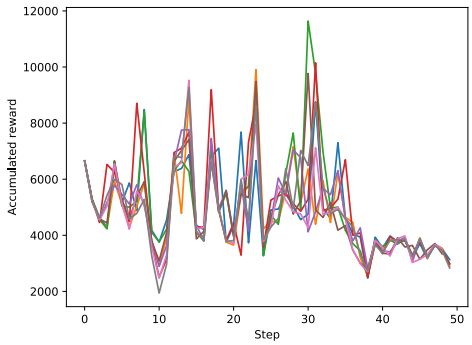
(b) ANQL



(c) DDQN

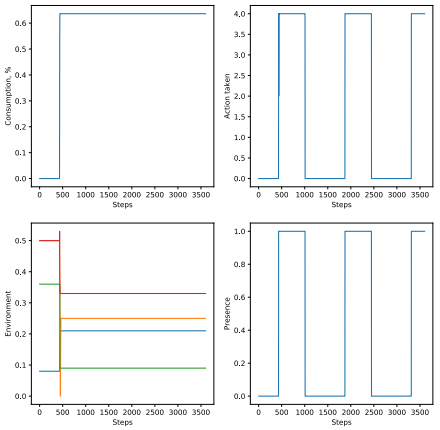


(d) DQN

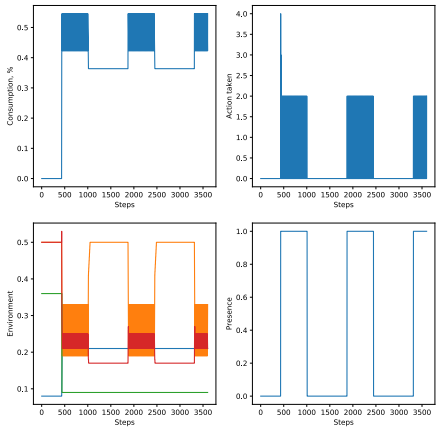


(e) A3C RNN

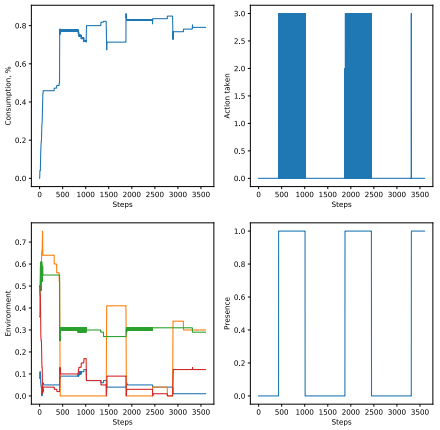
Figure A.3: Accumulated reward over epochs with regard to strict presence profile and normal negative feedback utility function using historical data



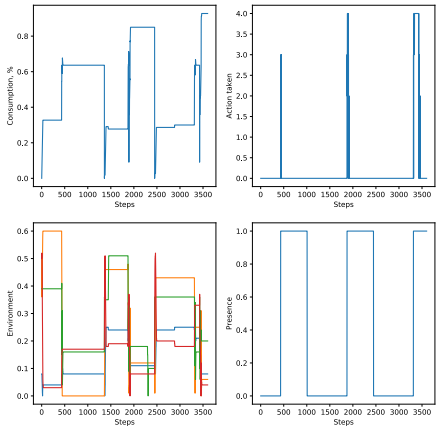
(a) A3C



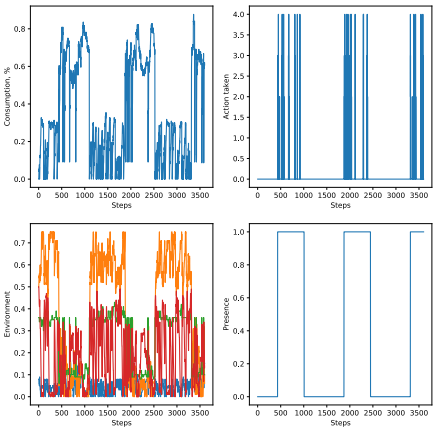
(b) ANQL



(c) DDQN

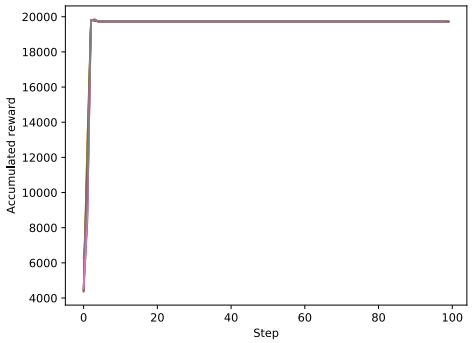


(d) DQN

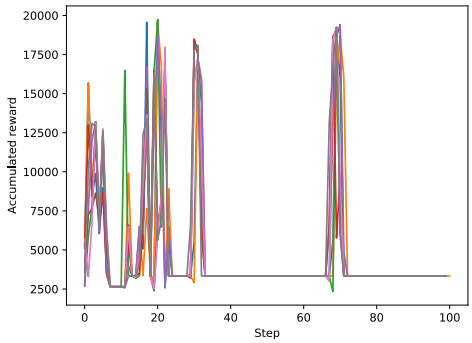


(e) A3C RNN

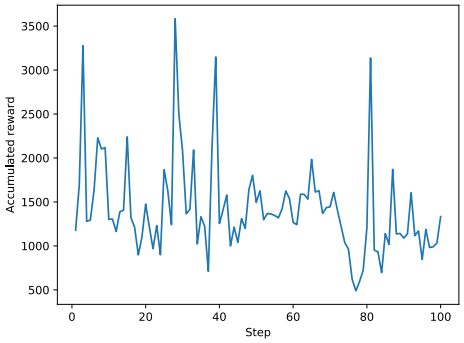
Figure A.4: Consumption, feedback action, environment conditions and presence with respect to the current time step during the simulation run using historical data



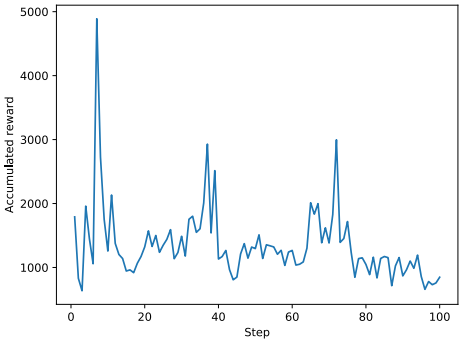
(a) A3C



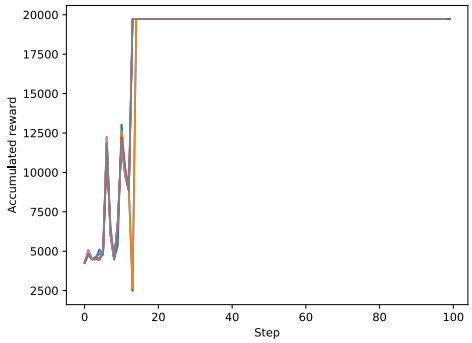
(b) ANQL



(c) DDQN

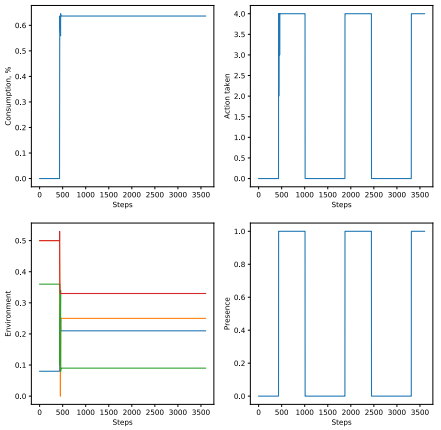


(d) DQN

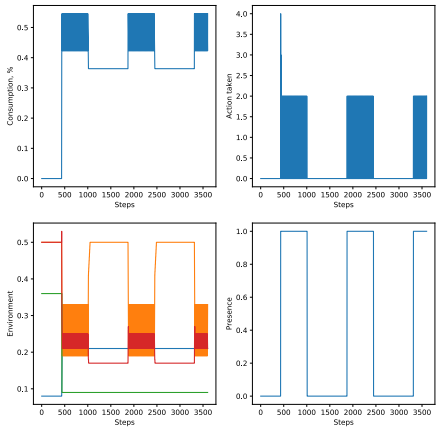


(e) A3C RNN

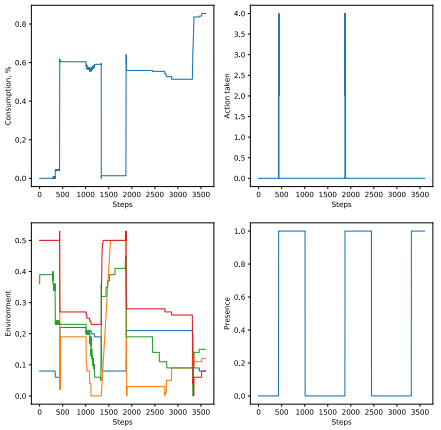
Figure A.5: Accumulated reward over epochs with regard to strict presence profile and normal negative feedback utility function in POMDP setting



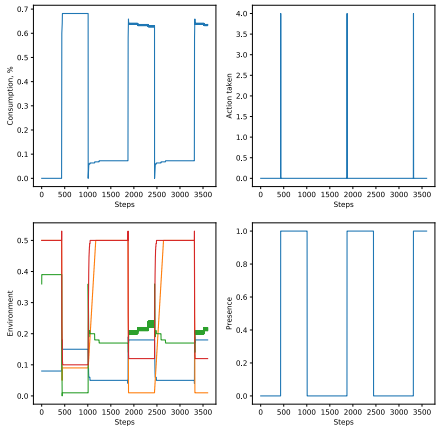
(a) A3C



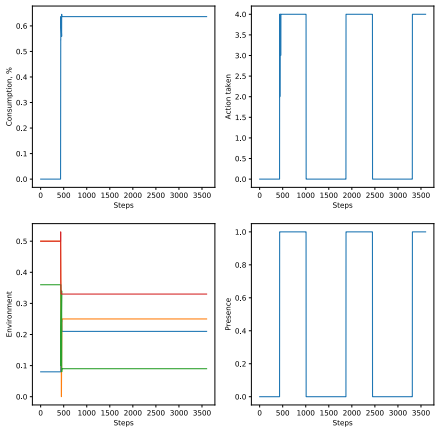
(b) ANQL



(c) DDQN

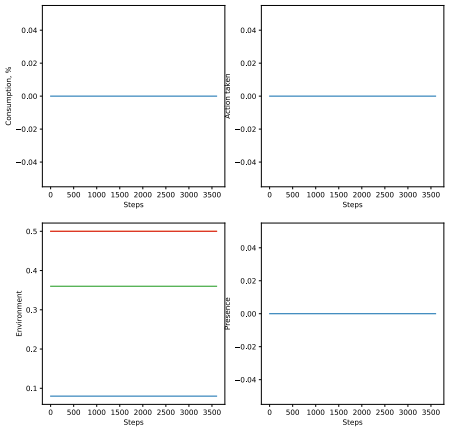


(d) DQN

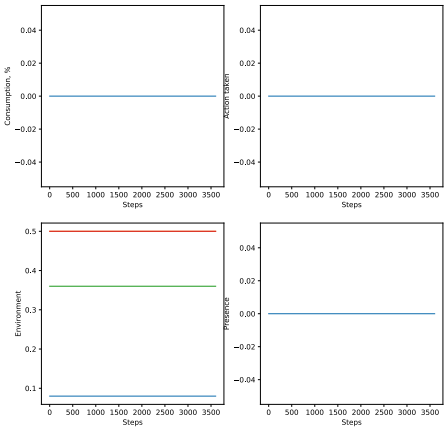


(e) A3C RNN

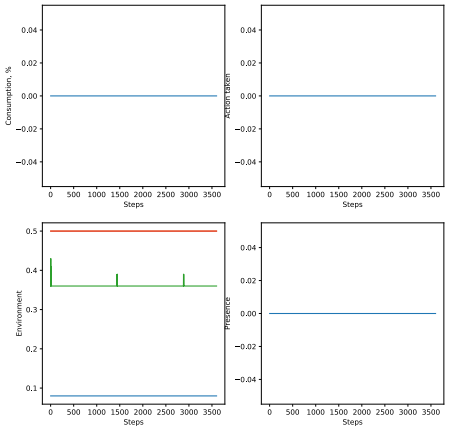
Figure A.6: Consumption, feedback action, environment conditions and presence with respect to the current time step during the simulation run in POMDP setting



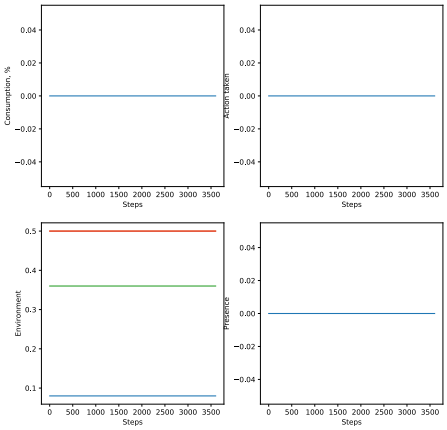
(a) A3C



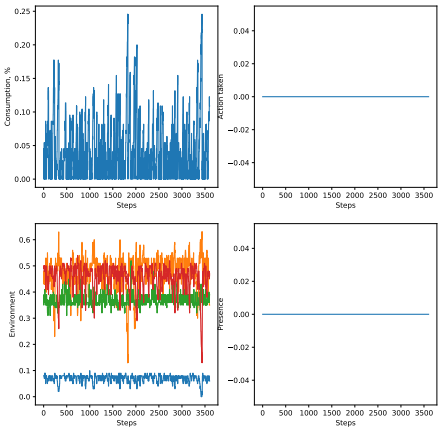
(b) ANQL



(c) DDQN

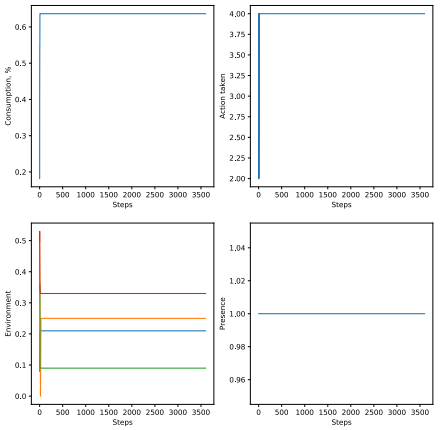


(d) DQN

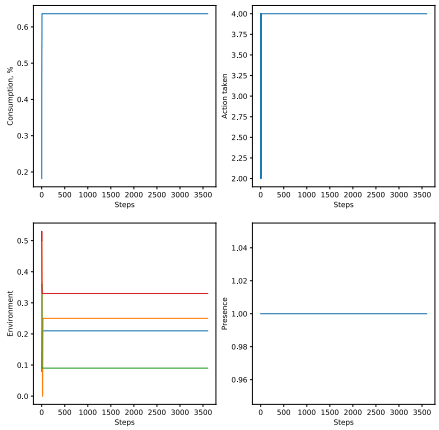


(e) A3C RNN

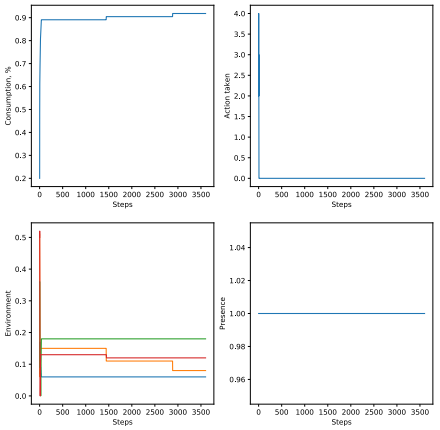
Figure A.7: Consumption, feedback action, environment conditions and presence with respect to consumption only reward function



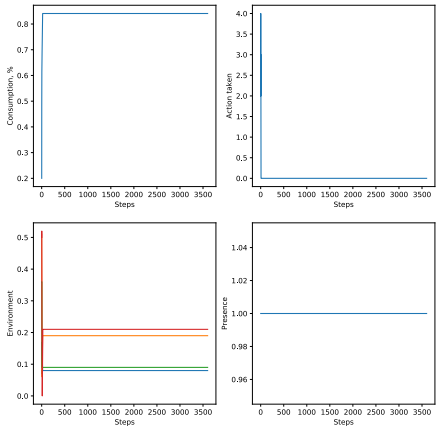
(a) A3C



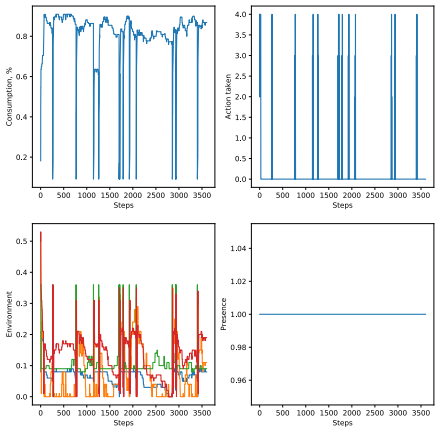
(b) ANQL



(c) DDQN

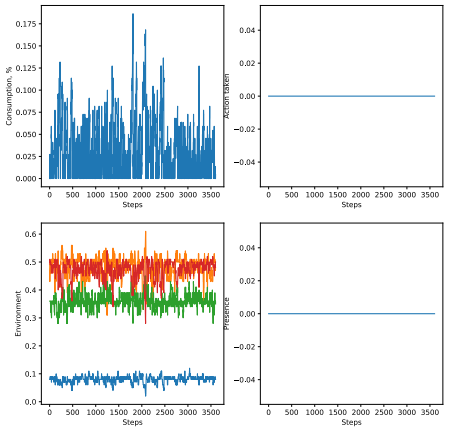


(d) DQN

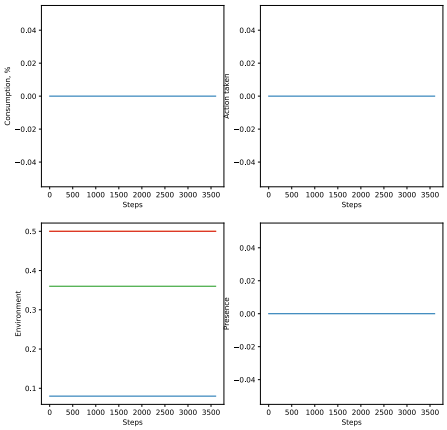


(e) A3C RNN

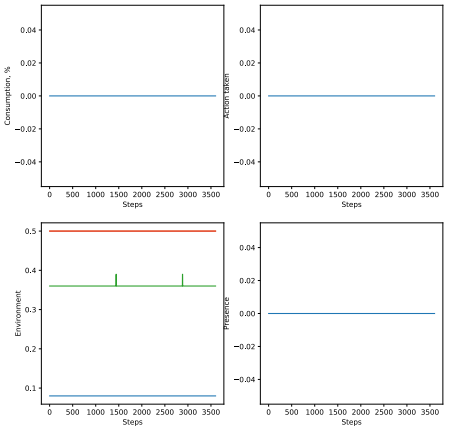
Figure A.8: Consumption, feedback action, environment conditions and presence with respect to preference satisfaction only reward function



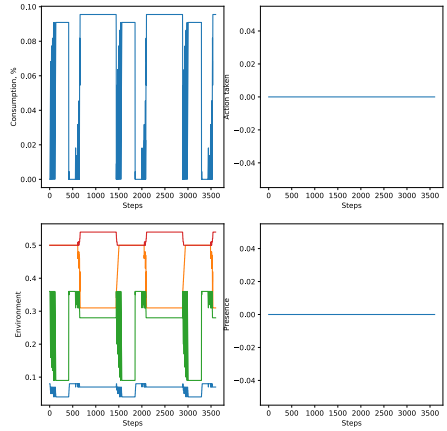
(a) A3C



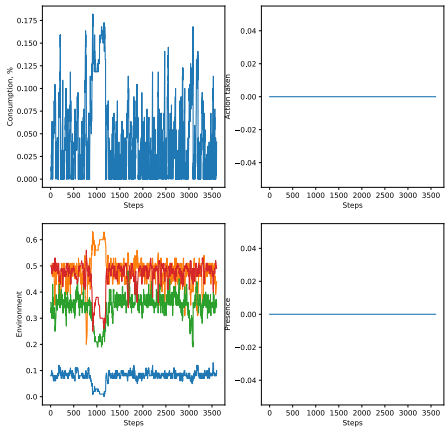
(b) ANQL



(c) DDQN

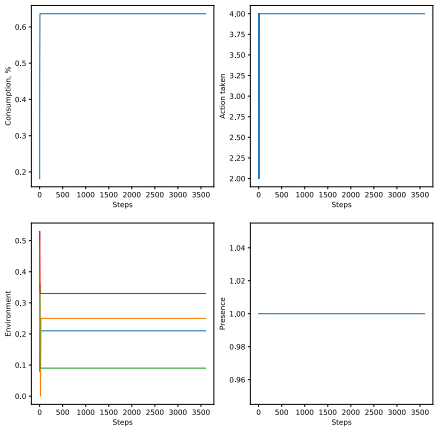


(d) DQN

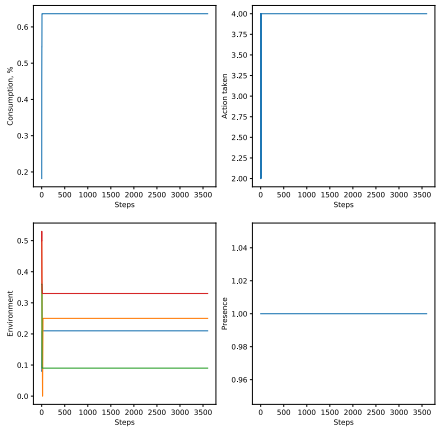


(e) A3C RNN

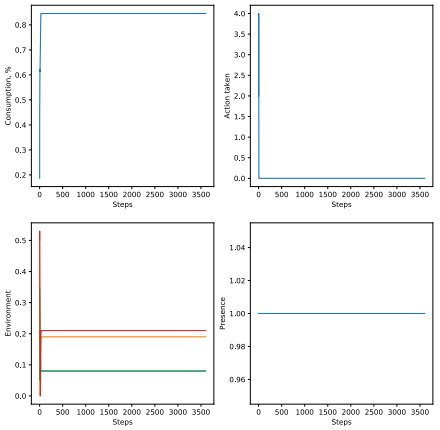
Figure A.9: Consumption, feedback action, environment conditions and presence with respect to consumption only reward function in POMDP setting



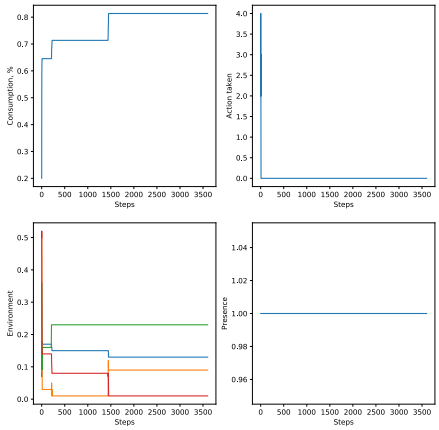
(a) A3C



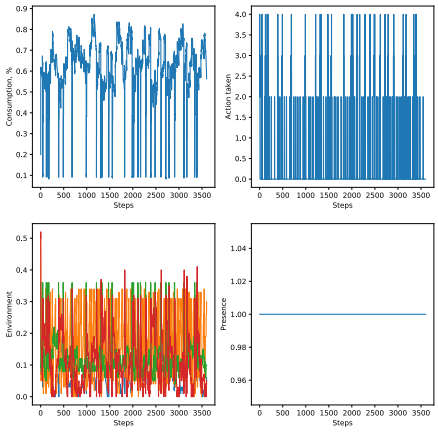
(b) ANQL



(c) DDQN

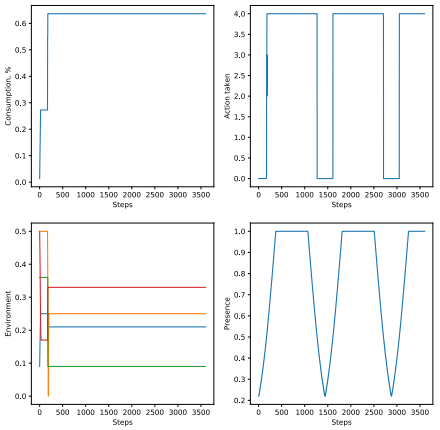


(d) DQN

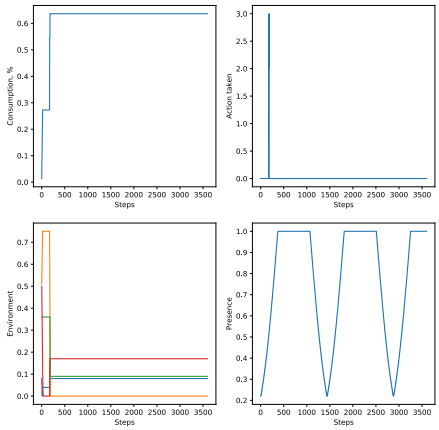


(e) A3C RNN

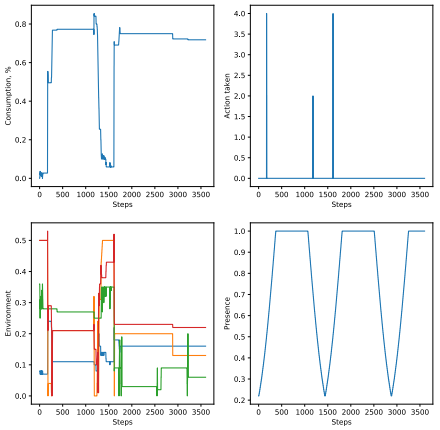
Figure A.10: Consumption, feedback action, environment conditions and presence with respect to preference satisfaction only reward function in POMDP setting



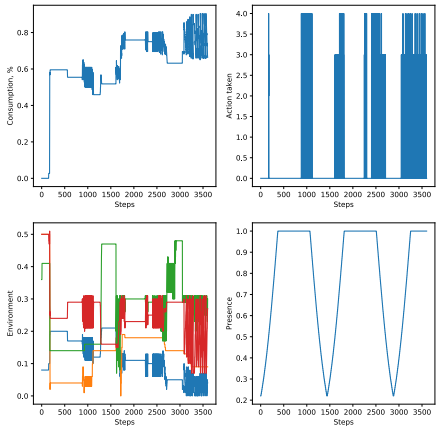
(a) A3C



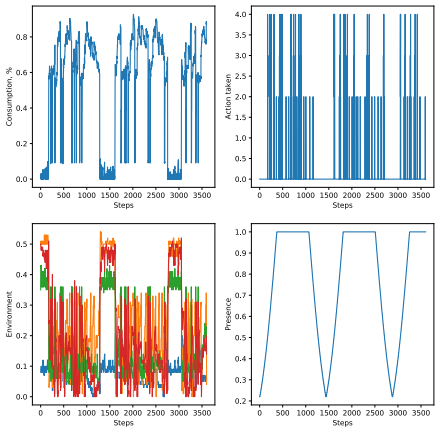
(b) ANQL



(c) DDQN

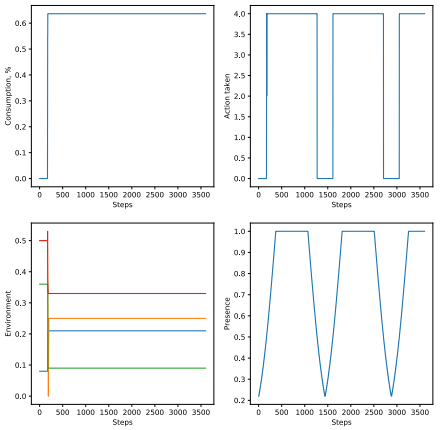


(d) DQN

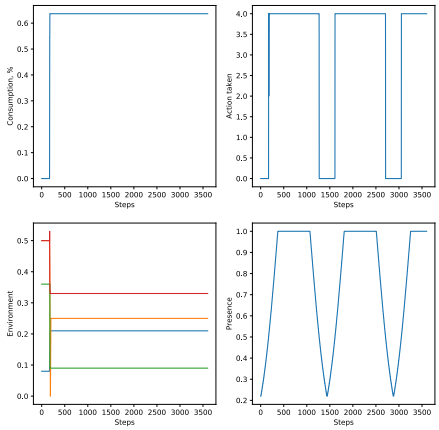


(e) A3C RNN

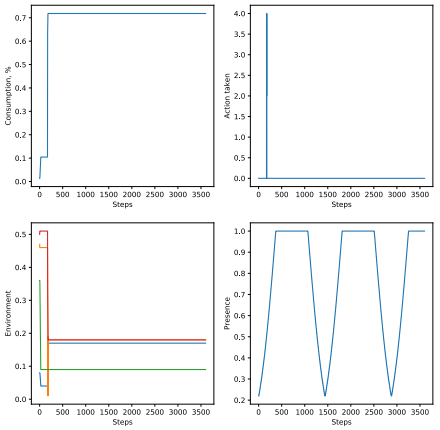
Figure A.11: Consumption, feedback action, environment conditions and presence with respect to normal presence profile in POMDP setting



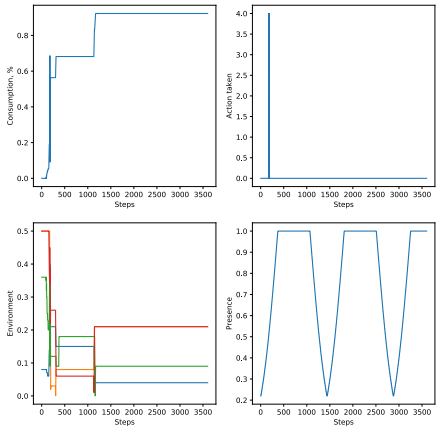
(a) A3C



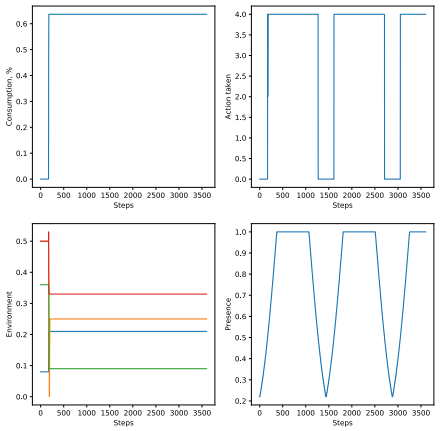
(b) ANQL



(c) DDQN



(d) DQN



(e) A3C RNN

Figure A.12: Consumption, feedback action, environment conditions and presence with respect to normal presence profile

References

Akrour, R., Schoenauer, M., Sebag, M., & Souplet, J.-C. (2014, 06). Programming by feedback. , *4*.

Arthur Juliani. (2016). *Simple Reinforcement Learning with Tensorflow Part 4: Deep Q-Networks and Beyond.* https://medium.com/@awjuliani/simple-reinforcement-learning-with-tensorflow-part-4-deep-q-networks-and-beyond-8438a3e2b8df. ([Online; accessed 04-July-2017])

Ballard, D. H. (1987). Modular learning in neural networks. In *Proceedings of the sixth national conference on artificial intelligence - volume 1* (pp. 279–284). AAAI Press. Retrieved from http://dl.acm.org/citation.cfm?id=1863696.1863746

A Beginner's Guide to Recurrent Networks and LSTMs. (2017). https://deeplearning4j.org/lstm.html. ([Online; accessed 04-July-2017])

Bellman, R. (1957). *Dynamic programming*.

Christiano, P., Leike, J., Brown, T. B., Martic, M., Legg, S., & Amodei, D. (2017, June). Deep reinforcement learning from human preferences. *ArXiv e-prints*.

Cybenko, G. (1989, Dec 01). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, *2*(4), 303–314. Retrieved from https://doi.org/10.1007/BF02551274 doi: 10.1007/BF02551274

Fairbank, M., & Alonso, E. (2011, January). The Local Optimality of Reinforcement Learning by Value Gradients, and its Relationship to Policy Gradient Learning. *ArXiv e-prints*.

Fairbank, M., & Alonso, E. (2012, June). Value-gradient learning. In *The 2012 international joint conference on neural networks (ijcnn)* (p. 1-8). doi: 10.1109/IJCNN.2012.6252791

Fürnkranz, J., Hüllermeier, E., Cheng, W., & Park, S.-H. (2012, Oct 01). Preference-based reinforcement learning: a formal framework and a policy iteration algorithm. *Machine Learning*, *89*(1), 123–156. Retrieved from https://

doi.org/10.1007/s10994-012-5313-8 doi: 10.1007/s10994-012-5313-8

Georgievski, I., Nguyen, T. A., Nizamic, F., Setz, B., Lazovik, A., & Aiello, M. (2017). Planning meets activity recognition: Service coordination for intelligent buildings. *Pervasive and Mobile Computing*. Retrieved from http://www.sciencedirect.com/science/article/pii/S1574119217301050 doi: http://doi.org/10.1016/j.pmcj.2017.02.008

Gers, F. A., Schmidhuber, J., & Cummins, F. (1999). Learning to forget: Continual prediction with lstm. *Neural Computation*, *12*, 2451–2471.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press. (http://www.deeplearningbook.org)

Hadfield-Menell, D., Dragan, A. D., Abbeel, P., & Russell, S. J. (2016). Cooperative inverse reinforcement learning. *CoRR*, *abs/1606.03137*. Retrieved from http://arxiv.org/abs/1606.03137

Hausknecht, M., & Stone, P. (2015, July). Deep Recurrent Q-Learning for Partially Observable MDPs. *ArXiv e-prints*.

Heess, N., Wayne, G., Silver, D., Lillicrap, T., Tassa, Y., & Erez, T. (2015, October). Learning Continuous Control Policies by Stochastic Value Gradients. *ArXiv e-prints*.

Hochreiter, S., & Schmidhuber, J. (1997, November). Long short-term memory. *Neural Comput.*, *9*(8), 1735–1780. Retrieved from http://dx.doi.org/10.1162/neco.1997.9.8.1735 doi: 10.1162/neco.1997.9.8.1735

Hofmann, T., Schölkopf, B., & Smola, A. J. (2007, January). Kernel methods in machine learning. *ArXiv Mathematics e-prints*.

Jaakkola, T., Singh, S. P., & Jordan, M. I. (1995). Reinforcement learning algorithm for partially observable markov decision problems. In *Advances in neural information processing systems 7* (pp. 345–352). MIT Press.

Jim Gao. (2014). *Machine learning applications for data center optimization.*

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *CoRR*, *cs.AI/9605103*. Retrieved from http://arxiv.org/abs/cs.AI/9605103

Kaiser, L., Gomez, A. N., Shazeer, N., Vaswani, A., Parmar, N., Jones, L., & Uszkoreit, J. (2017, June). One Model To Learn Them All. *ArXiv e-prints*.

Landgraf, D., Long, J., Der-Avakian, A., Streets, M., & Welsh, D. K. (2015, 04). Dissociation of learned helplessness and fear conditioning in mice: A mouse model of depression. *PLOS ONE*, *10*(4), 1-17. Retrieved from https://doi.org/10.1371/journal.pone.0125892 doi: 10.1371/journal.pone.0125892

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G.,

... Hassabis, D. (2015, Feb 26). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529-533. Retrieved from http://dx.doi.org/10.1038/nature14236 (Letter)

Mnih, V., Puigdomènech Badia, A., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., ... Kavukcuoglu, K. (2016, February). Asynchronous Methods for Deep Reinforcement Learning. *ArXiv e-prints*.

Muller, H., Muller, W., Marchand-Maillet, S., Pun, T., & Squire, D. M. (2000). Strategies for positive and negative relevance feedback in image retrieval. In *Proceedings 15th international conference on pattern recognition. icpr-2000* (Vol. 1, p. 1043-1046 vol.1). doi: 10.1109/ICPR.2000.905650

Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press.

OpenAI universe-starter-agent. (2017). https://github.com/openai/universe-starter-agent. ([Online; accessed 12-September-2017])

Rae, J. W., Hunt, J. J., Harley, T., Danihelka, I., Senior, A., Wayne, G., ... Lillicrap, T. P. (2016, October). Scaling Memory-Augmented Neural Networks with Sparse Reads and Writes. *ArXiv e-prints*.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65–386.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., ... Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, *115*(3), 211-252. doi: 10.1007/s11263-015-0816-y

Russell, S. J., & Norvig, P. (2002). *Artificial intelligence: A modern approach (2nd edition)*. Prentice Hall. Hardcover.

Setz, B., Nizamic, F., Lazovik, A., & Aiello, M. (2016). Power management of personal computers based on user behaviour. In *International conference on smart cities and green ict systems* (pp. 409–416).

Si, J., & Wang, Y. (2001, 3). On-line learning control by association and reinforcement. *IEEE Transactions on Neural Networks and Learning Systems*, *12*(2), 264–276. doi: 10.1109/72.914523

Sutton, R., & Barto, A. (2017). *Reinforcement learning: An introduction.*

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning i: Introduction.*

van Hasselt, H., Guez, A., & Silver, D. (2015, September). Deep Reinforcement Learning with Double Q-learning. *ArXiv e-prints*.

van Seijen, H., Fatemi, M., Romoff, J., Laroche, R., Barnes, T., & Tsang, J. (2017, June). Hybrid Reward Architecture for Reinforcement Learning. *ArXiv e-prints*.

Vezhnevets, A., Osindero, S., Schaul, T., Heess, N., Jaderberg, M., Silver, D., & Kavukcuoglu, K. (2017, March). FeUdal Networks for Hierarchical Reinforcement Learning. *ArXiv e-prints*.

Wang, F. Y., Zhang, H., & Liu, D. (2009, May). Adaptive dynamic programming: An introduction. *IEEE Computational Intelligence Magazine*, *4*(2), 39-47. doi: 10.1109/MCI.2009.932261

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., & de Freitas, N. (2015, November). Dueling Network Architectures for Deep Reinforcement Learning. *ArXiv e-prints*.

Weng, J., Ahuja, N., & Huang, T. S. (1992, Jun). Cresceptron: a self-organizing neural network which grows adaptively. In *[proceedings 1992] ijcnn international joint conference on neural networks* (Vol. 1, p. 576-581 vol.1). doi: 10.1109/IJCNN.1992.287150

Werbos, P. J. (1988). Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, *1*(4), 339 - 356. Retrieved from `http://www.sciencedirect.com/science/article/pii/089360808890007X` doi: http://dx.doi.org/10.1016/0893-6080(88)90007-X