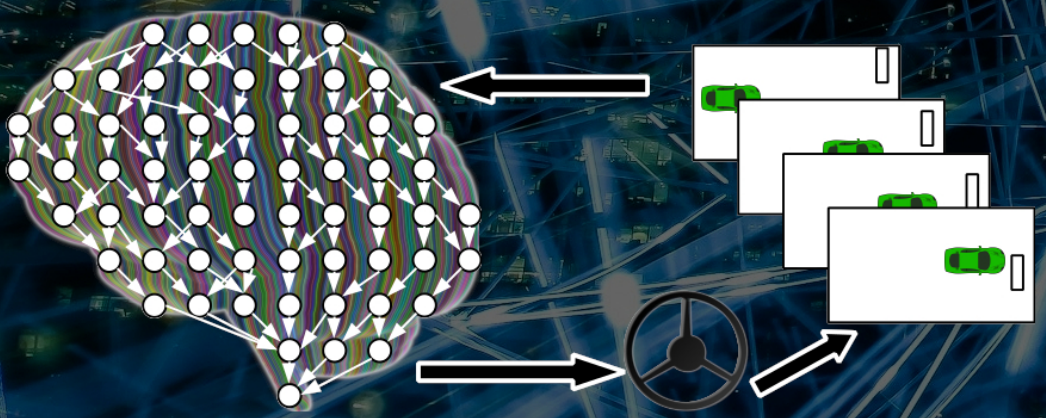
Safe reinforcement learning in long-horizon partially observable environments

B. Kovács

Technische Universiteit Delft



TU Delft

Safe reinforcement learning in long-horizon partially observable environments

by

B. Kovács

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday August 20, 2020 at 10:30 AM.

Student number:	4770552	
Project duration:	November, 2019 – August, 2020	
Thesis committee:	Dr. M. T. J. Spaan,	TU Delft, supervisor
	Dr. J. C. van Gemert,	TU Delft
	Dr. J. Kober,	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

The cover photo was based on
https://www.pexels.com/de-de/foto/abstrakt-bewegung-bewirken-blitz-373543/ and
cliparts from https://openclipart.org/ and http://www.clker.com/.

TU Delft

Preface

You are reading my thesis project, concluding my master studies at Delft University of Technology. In the past nine months, I was conducting research in deep reinforcement learning, an emerging area in artificial intelligence. During this period, my primary goal was to implement a project that has high added value for the machine learning community. I have faced several challenges in both the scientific and engineering parts of my project. Especially, large-scale training of the agents and training for hundreds of thousands of steps proved to be a time-consuming task, during both the hyperparameter tuning phase and the experimental evaluation. Finally, I am happy that I have managed to overcome all difficulties and reach appealing conclusions. I am proud to deliver this report about my work and hope that it will serve as a basis for a scientific publication and my possible Ph.D. studies in the future.

I am thankful to my supervisors Dr. Matthijs Spaan and Qisong Yang, for their guidance through this project with all their advice and support, and to the members of the thesis committee, Dr. Jan van Gemert, and Dr. Jens Kober for providing feedback and evaluating my work. I would also like to thank all the teaching staff of TU Delft for providing high-quality education with exciting lectures, challenging team projects, and many more scientific activities outside class hours. It was pleasant to be part of the international community in the past two years as well as to develop my skills through group projects and teaching assistant activities.

Naturally, this research would not have been possible without the continuous support from my family. Also, I am especially thankful to my girlfriend, Dorina, for keeping me motivated and making the past two years fun and memorable together with our friends, Zsombor, Betti, and Dorka.

Finally, this project could not have been implemented without many open source libraries and online services. Since most of the training procedure was conducted in the cloud ecosystem, I would also thank for the Google and Microsoft for funding my research by making several tools (Google Colab notebooks, Azure ML tools, Google Cloud) and free credits available for educational and research purposes.

B. Kovács
Delft, August 2020

Abstract

Deep reinforcement learning went through an unprecedented development in the last decade, resulting in agents defeating world champion human players in complex board games like go and chess. With few exceptions, deep reinforcement learning research focuses on fully observable environments, while there is slightly less research in the direction of partially observable environments. Though, they are relevant in real-world applications where the physical systems' sensors can read only a limited subset of features required for decision making. In many problems, long-term memory is required for the agent to make optimal decisions, possibly through tens or hundreds of timesteps. In supervised deep learning, in the NLP domain, sequential input processing is done with specialized network architectures like variants of RNNs and attention-based networks.

In this work, we investigate the potential of these advanced sequence-processing architectures in the context of deep reinforcement learning for partially observable environments. Additionally, since partial observability widely appears in the physical world, we take a safe approach by trying to limit high exploration costs and damage to the agent and its environment. First, we augment the soft actor-critic method with constraints on the episodic cost, resulting in an objective function with two Lagrangian multiplier: an entropy temperature and a safety temperature. Then, to support long-horizon, partially observable environments, we use gated recurrent (LSTM, GRU) and self-attention based neural networks for the policy and the estimation of Q-functions. We also study how the design choices and hyperparameters of the self-attention based method affect the performance.

To evaluate the problem in safety-constrained environments with long-term temporal dependencies, we develop a new set of benchmarks with four parameterizable, partially observable simulations. The environments are also parameterizable with the length of the history containing relevant features. Hence, we can observe how different network architectures handle the same problems with varying time horizons. Additionally, we introduce a practical framework for the reproducible evaluation of the methods.

We conclude that both the recurrent and the self-attention based architectures have high application potential in the introduced domains. We confirm that the feedforward network based baseline agent, shows high performance on problems where only a few, or tens of timesteps have to be processed sequentially. The recurrent and self-attention based architectures show their advantage in environments with longer horizons, where the sequence of events play an important role and looking back to a fixed position is not sufficient.

Maintaining safety proves to be problematic when the reward and cost functions show correlation. Additionally, strict cost limits usually lead to a poor policy and no exploration, contributing to higher costs on the long term, for some environments.

We propose further research for architectural changes to scale up the method for more complex environments, and to analyze the method on discrete-action and environments.

Contents

List of Figures

List of Tables

Acronyms

A2C advantage actor critic

A3C asynchronous advantage actor critic

ANN artificial neural network

API long

CMDP constrained Markov decision process

CNN convolutional neural network

CPO constrained policy optimization

DDPG deep deterministic policy gradient

DL deep learning

DNN deep neural network

DQN deep Q network

DRL deep reinforcement learning

DRQN deep recurrent Q-network

FPS first person shooter

GPU graphics processing unit

GRU gated recurrent unit

LSTM long short-term memory

MDP Markov decision process

MLP multilayer perceptron

NLP natural language processing

POMDP partially observable Markov decision process

PPO proximal policy optimization

ReLU rectified linear unit

RL Reinforcement learning

RNN recurrent neural network

SAC soft actor-critic

SGD stochastic gradient descent

SRL safe reinforcement learning

TD temporal-difference

TD3 twin delayed deep deterministic policy gradient

TRPO trust region policy optimization

1 Introduction

Reinforcement learning (RL) (Sutton and Barto, 2018) is an emerging field of artificial intelligence, one of the most actively researched areas nowadays. It is usually treated as the third essential machine learning paradigm besides supervised- and unsupervised learning. In reinforcement learning, a self-learning agent is employed with the goal of discovering how to act optimally in an unknown environment. Agents interact with the environment in discrete steps by taking actions based on the observations. After each step, the environment emits a scalar reward signal. Environments can be episodic, meaning that after reaching a state or a time limit, the environment terminates. The agent's goal is to maximize the return – the sum of the per-step rewards received through an episode.

Besides its huge successes of beating human champions in chess (Silver et al., 2017a) or go (Silver et al., 2017b), reinforcement learning is a natural solution for many real-world problems like closed-loop controlling physical systems or recommending content online. Some people even suggest that reinforcement learning-based techniques are the next milestone on the way to general artificial intelligence (Rocha et al., 2020). While deep reinforcement learning is not yet mature (Ray et al., 2019) it is already extensively used in healthcare (Hochberg et al., 2016), autonomous driving (Shalev-Shwartz et al., 2016) and for trading on financial markets (Nevmyvaka et al., 2006).

The research community focuses on plenty of domains where practical applications of RL can arise. One of the exciting fields is robotics: traditionally, robots are controlled by handcrafted controllers, constructed based on mathematical models using control theory. Reinforcement learning provides a viable alternative to replace classical controllers by policies trained using the toolset of machine learning. Although being a promising approach, to achieve state-of-art results in the field, several challenges need to be tackled, before one can exploit its full potential.

First, robots usually have to make decisions based on high-dimensional and high-resolution sensors like cameras, thermometers, and positional encoders. The used sensors may only provide information with significant delays and noise due to their physical properties and measurement technology. Environments where the observations show similar issues are considered partially observable, contrary to fully observable environments where all relevant state variables are revealed to the agent.

Second, the action space in the robotic domain is challenging to handle due to its possible high dimensionality – for example a, robotic arm may have at least 7 degrees of freedom with the corresponding control signals – and the fact that it is usually continuous – like the voltage applied to a motor.

Moreover, the consequences of actions may appear only on a longer time horizon, which makes it hard to connect an outcome to its determining action. It is also natural that an agent receives only a subset of the important features as observation, rather than the full state of the system during the interaction and may have to take a sequence of observations into account for optimal decision making. It is especially the case in real-world environments because several state variables are not, or not directly measurable contrary to simulations where those are mostly stored in the simulator's memory. For example, the speed of a rotating wheel sometimes can only be measured by differentiating its position, while in labyrinths, several corridors may appear similar. Of course, in the former example of speed estimation, manual preprocessing can significantly make the agent's job easier, but such an algorithm is not trivial to build for more advanced applications like self-driving cars, where the observation space

consists of hundreds of features in every single timestep. In problems where multiple observations need to be processed, the agent usually needs internal memory to find an optimal policy.

Finally, agents in real-world environments usually need to maintain safety during the operation: a robot should never harm equipment or humans, while RL-based trading systems should not unnecessarily risk clients' investments. There are several approaches for safety in reinforcement learning.

For safety-critical environments, where minimal violation of the safety constraints is not permitted, preliminary knowledge is necessary to keep exploration safe. For example, a heuristic can override the agent's action when it tries to act with possible negative consequences.

An alternative method is defining a cost function – similarly to the reward function – and letting the cost-aware agent explore its own environment. Still, in general, depending on the characteristics of the environments, safety constraint violations may appear. It is especially the case at the beginning of the training procedure when the agent has minimal knowledge about the environment.

This work studies safe reinforcement learning methods for long-horizon partially observable environments with continuous high-dimensional observation and action spaces. Complex, continuous environments are usually solved by deep reinforcement learning (DRL) techniques, algorithms using deep neural networks (DNNs) as non-linear function approximators to estimate value functions and policies. Contrary to the fully observable setup where a single observation is sufficient for optimal decision making, in partially observable cases, the agent needs to process a sequence of observations, possibly even full episodes. Acting safely can also be more challenging, due to safety-related features and state variables, possibly missing from the observation, making representation learning necessary. To tackle the challenges introduced by partial observability, previous RL work tends to feed short sequences of observation frames into dense feedforward networks. When the input space consists of high-resolution images, convolutional layers are used beforehand (Mnih et al., 2015).

However, while the approach generally shows good performance on environments with short-term temporal dependencies like Atari games, it is likely that in real-world environments, the approach has only limited usage opportunities due to important features' possible spread across hundreds of timesteps. In the small game environments, only three timesteps need to be processed, and the agent has to look back for a limited time interval. Nonetheless, in real-world environments, certain events can occur at any time, affecting all upcoming timesteps. In this context, the feedforward network would require learning the same features for all possible time intervals. However, achievements in deep learning showed that practical applications of feedforward networks are limited in sequence processing (Sutton and Barto, 2018) and proposed recurrent and attention-based networks for natural language processing (NLP) and its subdomains.

Therefore, intuitively, deep reinforcement learning methods are also likely to benefit using advanced, sequence processing architectures for more complex, partially observable tasks. There are two popular techniques for sequential input processing. Traditionally, recurrent architectures (Sutskever et al., 2014) like long short-term memory (LSTM) (Hochreiter and Schmidhuber, 1997) and gated recurrent unit (GRU) networks Cho et al. (2014) dominated the NLP field. Later, recurrent models augmented with attention mechanisms proved to be more performant. The recurrent architectures were recently replaced by the self-attention-based transformer (Vaswani et al., 2017) and its variants, powering current state-of-art models like GPT-3 (Brown et al., 2020). Some attempts were made to make use of these approaches in RL (Heess et al., 2015), however, there is still a lack of extensive research in the field, contrary to the high potential of the mentioned architectures.

1.1. Problem definition

In this work, we solve safety-constrained reinforcement learning problems in partially observable environments. The environments are made challenging by the fact that a long history of observations contains relevant information about the environment. An agent acts in an environment in discrete time steps. On each step, the agent receives an observation, a reward, and a cost from the environment. Then, it picks an action according to its policy and a set of past observations. The environment executes the action, then the process is repeated until the episode ending in several steps. The episode may end either due to reaching a terminating state or because a time limit is exceeded. The agent's goal is to maximize the (discounted) sum of the reward while sticking to the cost limits provided. The agent is considered to be safe if the expected cost regret for a full episode remains below a given threshold.

1.2. Research questions

We base our research work on our remark about many potential real-world application fields of reinforcement learning. For advanced control tasks (consider a self-driving car or several challenges on the smart-grid), taking a long history of past observations into account is essential. Specifically, for the former example, a car should remember and predict the trajectory of a pedestrian, even if it temporarily gets behind a car. The smart grid controller algorithm also has to focus on trends, besides the actual state. Naturally, safety is also crucial in the context. Our research goal is to examine deep reinforcement learning techniques that could get us closer to achieving a breakthrough for the mentioned problem context.

Based on our goal, we formulate our main question as follows:

How can deep reinforcement learning agents learn safe policies in long-horizon partially observable environments?

We answer the main research question by developing and evaluating deep reinforcement learning agents. Our first sub-research question focuses on the selection of a deep neural network architecture for the problem context.

Which novel deep learning models are the best suited for use with deep reinforcement learning algorithms to process long sequences of observations in partially observable environments?

In our second subquestion the safety-related requirement is considered.

How to make the selected algorithms safer to minimize risks of taking damaging actions?

The final subquestion concerns the environments and research tools required for evaluating our agents' performance.

How to conduct a reproducible evaluation of agents designed for safe reinforcement learning in partially observable environments?

1.3. Contributions

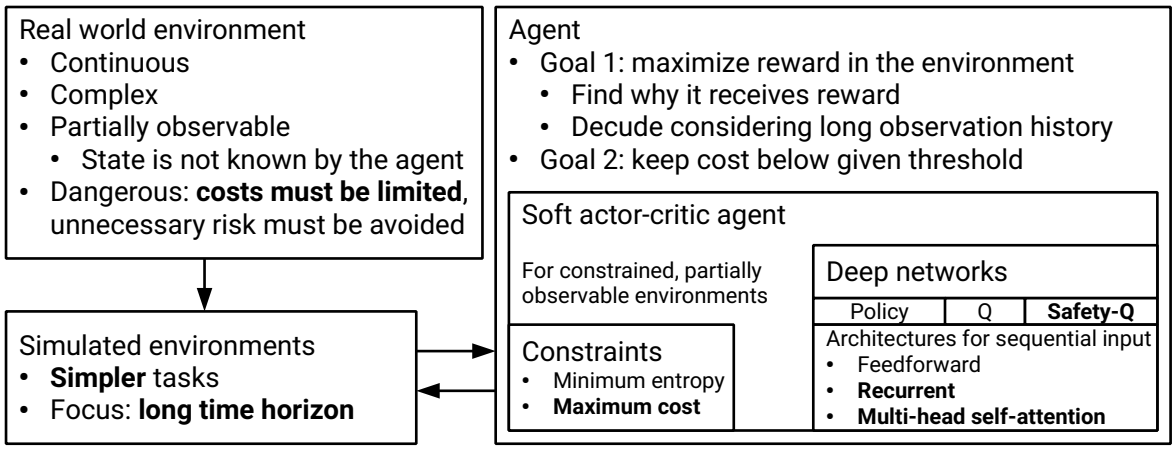


Figure 1.1: Overview of our work

A visual overview of our work is available in figure 1.1. The contributions of our research are threefold.

First, we introduce a novel problem setup in the deep reinforcement learning research. Earlier work was concerned with fully observable environments and partially observable environments where shorter sequences of observations are enough to learn an optimal policy. Contrary, we explicitly focus on long sequences. To gain insights on how the characteristics of an environment affects the performance of agents, we introduce four environments as the part of a new benchmark suite.

To solve our environments we develop deep reinforcement learning agents based on the soft actor-critic method and sequence processing architectures like recurrent networks and self-attention mechanisms. While recurrent networks have already been used in deep reinforcement learning, they were used with simpler algorithms like deep recurrent Q-networks. Additionally, to the best of our knowledge,

self-attention mechanisms only had minimal application in reinforcement learning so far, with no work focusing on its design choices like the number of attention heads and the position encoding options.

In our experimental evaluation we focus on the effects of sequence lengths and conduct an in-depth analysis of the variants of the introduced network architecture.

1.4. Outline

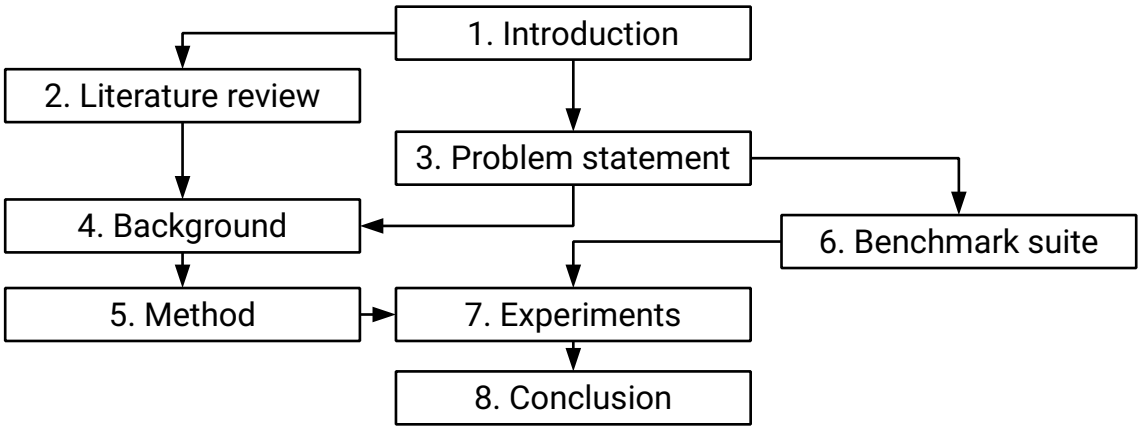


Figure 1.2: Outline

The rest of this work is set up the following way: we formally introduce our problem in chapter 3. Chapter 2 presents the background literature followed by the most the baseline algorithms and approaches this work bases upon, presented in chapter 4. Chapter 5 presents the proposed reinforcement learning method. Chapter 6 continues with the introduction of a new benchmark suite with sequential problems. Then, we present a framework for reproducible experiments and evaluation in section 7.3. In the remaining sections of chapter 7 the empirical evaluation of the proposed methods follows, with some discussion about the outcome of the experiments and analysis of the proposed methods compared to the baseline algorithms from earlier work. Finally, chapter 8 concludes the work and provides outlook for further research opportunities. The appendix provides an overview of the hyperparameters used during the final experiments, as well some details about the practical implementation of the project.

2

Literature review

This chapter introduces the basic definitions, elementary algorithms in the research area of (deep) reinforcement learning, followed by an overview of the latest relevant papers from the literature. In general, the chapter is based on peer-reviewed literature of conference and journal papers and some books. However, there are a few exceptions: since we can observe unprecedented evolution of the field in the past years, with new methods and papers appearing on a daily basis, it is important to consider parallel works and include (possible preprint) papers that appeared after the literature study phase of the project.

While now we give a comprehensive review of previous research, the highlighted papers we directly build our work upon, are presented in chapter 4.

2.1. Reinforcement learning

In Reinforcement learning (RL), a software agent – usually without preliminary domain-specific knowledge – interacts with the environment in discrete steps by taking actions based on the observation and a scalar reward signal received from the environment. The interaction is usually split into episodes, ending either when reaching a final state or after a given number of timesteps. The agent's goal is to maximize the return. The return is defined as the sum of the rewards collected by the agent in each step of an episode. For some infinite horizon (non-episodic) problems, the return is defined as the discounted sum of the reward collected in the future. In practice, we also use discounting in episodic cases.

Reinforcement learning problems are usually modeled as Markov decision processes (MDPs) (Bellman, 1957). MDPs are defined by a quadruple (S, A, T, R), where S is the set of states, A is the set of actions. $T : S \times A \times S \to \mathbb{R}$ is the transition function representing probabilities of reaching a target state s' from a state s by taking an action a'. $R : S \times A \to \mathbb{R}$ is the reward function returning a scalar reward for a state-action pair and it can be stochastic. Also, a discount factor γ is usually used to encourage short-term rewards and make infinite or long horizon problems tractable.

Exploration-exploitation trade-off plays an important role in RL problems. Exploration corresponds to discovering actions that may result in higher rewards, while exploitation means taking actions that are advantageous compared to others based on the current knowledge (Kaelbling et al., 1996).

Researchers proposed various approaches to solve RL problems. We can categorize RL algorithms by several features: first, model-based algorithms build a model of the environment and solve it by executing planning algorithms on the learned model. Contrary, model-free algorithms aim to maximize the long-term reward without building an explicit model. In general, model-based algorithms tend to be more sample efficient. However, model-free algorithms can be used in a broader range of environments where it is not possible or not practical to build a model. From another aspect, methods can be classified as off-policy and on-policy algorithms. The former follows an alternative policy during training compared to the behavior policy used when the goal is to maximize the reward for a single episode. On-policy algorithms take actions during exploration based on the actual policy being learned to maximize the return. Exploration is their natural behavior.

The classical tabular Reinforcement learning algorithms are aimed to be used in small environments

with low-dimensional discrete observation and action spaces. Larger continuous environments can be handled by deep reinforcement learning techniques. Next, we look at the essential methods.

The Bellman equation Bellman (1957) propose the well-known dynamic-programming algorithm based on equation 2.1 which defines the relationship between the value of state s ($v_\pi(s)$) and the values of possible succeeding states s'. $p(s', r|s, a)$ defines the probability of transitioning from state s to s' while receiving reward r. The method can be used to compute the solution for stochastic optimal control problems. The price of optimality and the biggest drawback of the algorithm is the curse of dimensionality, so that the computation demands grows exponentially with the number of states.

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] \text{for all } s \in S \tag{2.1}$$

Q-learning One of the most successful algorithms for solving Reinforcement learning problems is Q-learning (Watkins and Dayan, 1992), which is the base of several current state-of-art deep reinforcement learning methods. It is a model-free temporal-difference (TD) algorithm – learning directly from samples, without building a model – and employs bootstrapping during the learning process – thus, samples can already be used before experiencing the final outcome (return of the full episode). The Q-learning algorithm can be described by equation 2.2 with (S_t, A_t) being the learned action-value function.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{2.2}$$

Then, the agent selects action with the highest value. To explore the environment, we usually use an ϵ-greedy policy: the action with the highest value is selected with probability $1 - \epsilon$, while a random action is taken with a probability of ϵ.

Notably, the algorithms presented above only work in the tabular case, when the state and action space is discrete and sufficiently small so that filling a table for all state-action pairs is tractable concerning the computational memory constraints. An additional drawback of tabular methods is that it may not be advantageous to treat states showing a high level of similarity differently, due to the increasing sample complexity involved. Therefore, for practical problems with larger and or continuous state and or action space, the table representation is usually replaced by function approximators, such as deep neural networks.

2.2. Deep learning

Deep learning (Goodfellow et al., 2016) is an artificial neural network-based machine learning method. The term *deep* comes from the hierarchical structure of the networks. Deep learning models have several layers of neurons built on top of each other. Higher-level layers aim to extract higher-level features based on lower-level features outputted by previous layers. While traditional machine learning algorithms depended on handcrafted feature extractors, deep neural networks learn representation through multiple abstraction levels. Thus, the resulting models can be fed raw input data like pixels for image and video processing or raw audio recordings for speech recognition. The mentioned tasks mainly belong to the class of supervised learning, where the algorithm is presented with examples where the expected outcomes corresponding to the raw input features are known. The model contains (possibly millions of) trainable weights that are adjusted to minimize a loss function based on an optimization objective. In the learning process, we compute the gradient for each weight with respect to the loss and update the weights in small steps in the opposite direction – to minimize the error. Hundreds of thousands to many millions of samples (training set) are needed to train deep networks to provide high accuracy and generalize well.

Training multilayer architectures proved to be a challenging problem. Later, it was discovered that backpropagation and stochastic gradient descent could be used successfully, given the layers represent smooth functions with respect to their inputs and weights. Backpropagation across multiple layers is the direct application of the chain rule. Interestingly, local minima did not prove to be a significant issue when training deep networks, and studies concluded that the final performance of the trained network seemed to be stable across multiple runs (LeCun et al., 2015).

In practice, to fit the networks to the expected outputs, variants stochastic gradient descent (SGD) algorithm, more recently Adam (Kingma and Ba, 2014), Adagrad (Lydia and Francis, 2019) or RMSProp (Tieleman and Hinton, 2012) is used. First, the actual outputs of the network and the losses are computed based on the targets (labeled examples). Then, based on the average gradient of the weights for a set of training samples (batch) and an error (loss), the optimizer algorithm updates the weights of the neural network.

During and after training, the performance is evaluated using a separate test set. Underfitting corresponds to the state where the network is not trained long enough to provide sufficient results, or the parameters or architecture of the network prohibit learning the proper function. Overfitting appears when training on a too-small data set compared to the number of parameters in the network, and the network specializes for the presented examples instead of generalizing.

The layers consume the outputs of the previous layer(s) as input and are usually followed by non-linear activation functions to simplify approximating non-linear functions. Popular choices are $\tanh$, sigmoid, and recently rectified linear unit (ReLU) (Glorot et al., 2011) that showed its superior performance on image classification and sequence processing tasks by accelerating training speed.

For classification, the network predicts the probability distribution for a given input belongs and n classes. In these cases, the head of the network (output) layers is usually a softmax function, defined as

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i \in 1, ..., K \text{ and } \mathbf{z} = (z_1, ..., z_K) \in \mathbb{R}^K. \tag{2.3}$$

In advanced supervised learning tasks in natural language processing (NLP) and image processing, the models usually outperform simpler machine learning techniques by a high margin and reach state-of-art performance (LeCun et al., 2015).

Convolutional neural networks (CNNs) are the prevailing models for processing images or video frames.

2.2.1. Sequence modeling

Recurrent neural networks (RNNs) are architectures designed for sequential tasks like natural language processing (translation, speech recognition) or predicting future data points of series. RNNs are usually capable of generalizing over variable (possibly unseen) sequence lengths, and the length of the produced outputs can differ for input sequences of the same size. These advantageous properties are reached by the fact that RNNs use the network parameterized by the same weights for each timestep. Sequence-to-sequence models usually use encoder-decoder architecture. The encoder generates a hidden state based on the input and the previous hidden state. Then, the decoder generates an output and a hidden based on the previous output and hidden state (Goodfellow et al., 2016). However, RNNs suffer from the vanishing or exploding gradient problem, prohibiting efficient learning for longer sequence lengths.

This difficulty was overcome by gated architectures. These specialized RNNs like LSTM (Hochreiter and Schmidhuber, 1997) and GRU (Cho et al., 2014) networks have been developed to maintain longer-term dependencies in traditional sequence processing problems like machine translation. However, while many state-of-art performing models used RNN-based architectures for long, recent advancements showed their limitations for long sequences, due to their fixed-sized hidden state representation (Pascanu et al., 2013). It is also known that the training procedure is hardly parallelizable for efficient processing on GPUs, since the internal states depend on previous ones and can only be computed sequentially. Moreover, the batch sizes are limited due to the high memory requirements for maintaining data over long sequences (Vaswani et al., 2017).

Vaswani et al. (2017) proposed the Transformer model to address these shortcomings of RNNs. In the novel model, temporal dependencies are being handled by attention mechanisms instead of a recurrent network. The model is based on self-attention mechanisms establishing the representation of a sequence by computing interaction between the elements in the sequence itself. Similarly to RNNs, the transformer also follows an encoder-decoder architecture. However, contrary to RNNs, where a sample consists of a whole input and output sequence, the transformer maps a series of inputs and (masked) sequence of previous outputs to the next element of the output sequence.

The transformer model was initially designed for machine translation tasks, so it assumes a series of words as source and target. The sentences are preprocessed with a tokenizer mapping them to

a sequence of integers according to a word-to-number dictionary. Next, the tokenized are embedded into low-dimensional vectors, where the words of similar meaning are represented as similar vectors. Since, contrary to RNNs, the model does not have any information about the position of words in the sentence, a positional encoding is added to the input vectors. The authors propose two solutions: in the papers, sinusoidal functions are used, but it is mentioned that learned positional encoding is also possible. Then, the encoder, consisting of N similar blocks, processes the input. The first multi-head attention sub-layer is summed with the residually connected input. The second sub-layer of the encoder consists of a feedforward network and similarly summed with its input. Both sub-layers are layer-normalized after the addition.

The decoder has a similar structure with an extra sub-layer with masked multi-head attention. The masking filters out the future elements of the sequence, so only the valid elements can be attended. The decoder is followed by a dense layer with the size of the output dictionary and softmax activation. Thus, its outputs correspond to the probability of a word from the target language. One of the key innovation points is using scaled dot-product attention that is efficiently computable using matrix operations, and the scaling makes the softmax function more reactive by pushing the values to a range where its gradient is higher. Additionally, multi-head attention makes it possible to focus on several parts of the input by executing more attention functions in parallel. To keep the mode computationally tractable, the dimensions of the heads are reduced with different projections to the key, query and value vectors.

The model was trained with the Adam optimizer (Kingma and Ba, 2014) with a dynamic learning rate consisting of a warm-up period followed by slow decay. To regularize the model, a dropout of rate 0.1 is added after each sub-layer and to the sums of embedding and positional encodings in both the encoder and the decoder. The model showed superior performance on standardized machine translation datasets compared to the state-of-art approaches of the time, with significantly less training cost and time.

The recent state-of-art models in the NLP research field are dominated by Transformer-based architectures like the BERT (Devlin et al., 2018), Transformer-XL (Dai et al., 2019) and GPT-3 (Brown et al., 2020).

2.3. Deep reinforcement learning

Traditional RL algorithms rely on learning a model or value and transition functions for each state or state-action pair of the environment. However, this approach quickly becomes intractable for large state space in real-world problems. The intractability comes from multiple sources: for example, in environments with continuous and or high-dimensional observation spaces creating a state-action table is not more feasible due to memory and time constraints. In fact, if all distinct states are considered separately, the whole state space must be traversed at least once to ensure full exploration. For continuous input spaces – like measurements coming from high-resolution sensors as raw camera frames – and continuous action spaces – robot control problems with analog reference signals for actuators – or their combinations, it would require infinite tables and exploration time. Also, we can assume in most environments that similar states will lead to similar outcomes. Thus with proper generalization, it is enough to visit only a representative subset of states. Quantization is proven to be useful for models with limited features. However, the exponential growth of the state space with feature number limits its usage to simple problems with small dimensionality – we call this phenomenon the curse of dimensionality.

To handle large input spaces, function approximators can be employed instead of tables. Advances in deep learning promise to fill the role of the function approximators by a deep neural network (DNN), leading to the approach of deep reinforcement learning (Arulkumaran et al., 2017). DRL algorithms are usually based on traditional RL algorithms like Q-learning or Sarsa, but use deep artificial neural networks for function approximation and possibly feature detection (if applicable for the input space). However, using DNNs in RL is challenging. Tsitsiklis and Van Roy (1997) experimentally illustrated the problem of possible divergence for non-linear function approximators in RL context. Also, the application of deep models raises theoretical issues like the violation of the independence of the training samples and the continuously changing targets that depend on the network itself.

Deep reinforcement learning is currently dominated by model-free algorithms. Value-based methods aim to learn a value function and establish a policy based on the learned action-value function, for example, by picking the highest value action in a given state.

A major breakthrough in DRL was presented by Mnih et al. (2013) (Mnih et al., 2015). Based on Q-learning, the authors introduce deep Q networks (DQNs) – a model-free, off-policy method – to solve environments with larger input spaces, where using traditional Q-tables would not be tractable due to the complexity of the curse of dimensionality. Deep neural networks are used for both feature extraction from raw sensory input (high-resolution images) and approximation of the action-value function (Q-function). The resulting agents show superhuman performance on several Atari games, while not using any handcrafted game-specific information and being trained with the same hyperparameters. The input frames from the Atari simulators are preprocessed by cropping the relevant region and conversion to grayscale. Since a single frame usually does not reveal the full state of the environment – movements of objects cannot be estimated – the Atari environments can mostly be considered partially observable. To deal with this issue, four stacked frames (84x84) are fed into the network as input. According to Hausknecht and Stone (2015), all games investigated in the experiments become fully observable in this case. The introduced DQN architecture is built up as follows: the inputs are fed to convolutional layers for object detection, followed by a densely connected layer with 256 neurons and rectifier activation function. There is a single neuron for each action in the output layer corresponding to the estimated Q-value of the state-action pair. An assumption in the training process of deep neural networks is that the training samples are independent and identically distributed. In the reinforcement learning setting, this assumption does not hold since a single agent exploring the environment provides experiences only about a small of states, so the training samples tend to correlate. To overcome the issue, the authors introduce an experience replay buffer collecting state, action, reward, next state tuples. The network is trained in every step based on a randomly selected set samples from the buffer. The replay buffer stabilizes the learning and makes more sample efficient by re-using past experiences. However, it adds an extra hyperparameter, the size of the replay buffer. After filling the buffer, the older experiences are removed. Schaul et al. (2015) also showed that prioritizing experiences in sampling can improve the convergence speed of the network. Another issue is that the target values depend on the weights of the network – contrary to supervised learning problems with fixed targets. To reduce the correlation Mnih et al. (2013) use a separate target network, which is updated after a fixed number of steps based on the weights of the trained network.

Although employing experience replay and target networks makes it possible to use deep neural networks as function approximators, approximation errors like overestimation bias and temporal difference error accumulation persist with DQNs. Fujimoto et al. (2018) shows that double deep Q networks are less effective in the actor-critic setting to solve these issues. They propose the twin delayed deep deterministic policy gradient (TD3) method that takes a minimum value predicted by two critic networks making overestimation slightly less likely. *Delayed* corresponds to the delay introduced in updating the actor. Finally, action smoothing is applied by adding noise to the generated actions, so single approximation errors do not take the learning in a wrong direction for a long term.

The generally known issue of overfitting in deep learning raises the question if it is also a problem in the RL context. Compared to supervised learning, where the set of examples is usually split into a training and a validation set, it is less trivial to investigate overfitting for RL models. Zhang et al. (2018) conducted experiments on a maze environment and conclude that large state-of-art models like A3C may fail to generalize properly. Agents acting in deterministic environments are more vulnerable to overfitting since, in this case, optimizing for open-loop action sequences may result in optimal policies (Kearns et al., 1999). It is also mentioned that there is a lack of protocols that are able to detect overfitting. Nevertheless, it would be crucial to develop such methods to ensure the safety of RL-agents acting in real-world environments.

2.3.1. Policy gradient methods

Policy-based methods aim to directly learn a parameterized policy function. In deep reinforcement learning, we use a deep neural network as the policy, with raw observations as its inputs. The policy function is optimized in the direction of improvement with respect to a given performance measure. The advantages of policy-based methods are the capability of learning optimal policies in environments, where estimating the value functions is not tractable. Additionally, policy gradient methods learn stochastic policies naturally. Given a discrete action space with n actions, the softmax function is applied to the n-neuron output layer, providing probabilities for taking each action. Thus a large variety of policies can be reached, ranging from near-deterministic ones to stochastic policies with similar probabilities for several actions. In the case of continuous action spaces, the head of the network usu-

ally represents parameters of probability distributions (for example, a normal distribution with a mean or standard deviation). Then, taking an action corresponds to sampling from the probability distribution with the predicted parameters.

It is important that there are stronger convergence guarantees for policy gradient methods compared to value-based methods, since the policy changes smoothly, while in the latter value-based case, an arbitrarily small change to an action value can lead to taking an entirely different path if its value becomes minimally higher.

However, policy gradient methods tend to be less sample efficient since the experiences are not incorporated in the long term as the policy is continuously changing. (Sutton and Barto, 2018)

Policy gradient methods can still use value-functions. However, contrary to value-based methods, they are not directly used for action selection.

Next, the basics of policy optimization are introduced. Let $\pi_\theta(a|s) = Pr\{A_t = a|S_t = s, \theta_t = \theta\}$ be the probability of taking action a in the t-th timestep, given s is the state of the environment and the policy is parameterized by $\theta \in \mathbb{R}^d$ (where d is the number of the parameters of the policy). Then, we aim to find an optimal θ that maximizes the performance of the policy for a given performance measure $J : \theta \to \mathbb{R}$.

The policy is updated using gradient ascent: $\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t)$, where $\nabla J(\theta_t)$ is approximated based on a batch of random samples.

Policy gradient theorem In order to update the policy in the right direction, the performance gradient needs to be estimated. However, changing the policy effects the state distribution, which is an unknown and encapsulated in the environment. Thus, it also affects which actions are taken, and the reward obtained. Nevertheless, the policy gradient theorem (equation 2.4, for episodic case), provides a way to update the policy parameters without knowing the mentioned state distribution.

$$\nabla J(\theta) \propto \sum_{s \in S} \mu(s) \sum_{a \in A} q_\pi(s, a) \nabla \pi(a|s, \theta) \tag{2.4}$$

Hereby, we work with a standard MDP with the set of states S, the set of actions A, and the parameters of the policy θ. μ is the on-policy distribution under policy π and $\nabla J(\theta)$ is the gradient of the policy parameters θ with respect to the performance measure J, thus exactly what we looked for.

2.3.2. Actor-critic methods

Actor-critic algorithms (Konda and Tsitsiklis, 2000) combine value- and policy-based methods. The function learned by the critic is used as a baseline to improve the actor. Notable actor-critic algorithms using deep networks as function approximators are (asynchronous) advantage actor-critic by Mnih et al. (2016) and deep deterministic policy gradient (DDPG) by Lillicrap et al. (2015).

Sample efficient actor-critic with experience replay (ACER) by Wang et al. (2016) is an off-policy actor-critic algorithm for RL problems for both continuous and discrete action spaces. ACER was developed aiming a universal algorithm that has the same performance as the deep Q networks on discrete action spaces but can also handle continuous action spaces more sample efficiently compared to A3C.

Schulman et al. (2015) present trust region policy optimization (TRPO), a theoretical method that guarantees improvement when training policies based on nonlinear function approximators, like deep neural networks. The proposed surrogate objective is the following:

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right] \tag{2.5}$$

The algorithm – adapted to practical use using approximations – show a solid performance in several robotic control domains.

Achiam et al. (2017) present constrained policy optimization (CPO), an extension of TRPO for CMDPs to incorporate safety constraints during training and convergence.

2.3.3. Deep reinforcement learning in continuous action spaces

The introduced deep reinforcement learning algorithms like the discussed DQN and A3C are designed for environments with discrete action spaces. However, several domains, especially in the physi-

cal world (like robotics), have continuous action space. Similar to high-dimensional continuous input spaces, discretization does not provide a general solution for environments accepting a float vector as an action.

Lillicrap et al. (2015) present deep deterministic policy gradient (DDPG), an off-policy learning algorithm that optimizes the actor-network. It uses a deterministic policy mapping, possibly high-dimensional continuous states to a float vector corresponding to the action, using an actor-critic approach. Similarly to DQNs, target networks are employed to stabilize learning (although slowing down the training procedure), and the experiences are collected in replay buffers and reused during the training procedure. The critic is trained based on Bellman-updates, similarly to Mnih et al. (2015). Some environments provide an observation space with measures of several units and numerical values of different scales. To improve training performance, batch normalization (Ioffe and Szegedy, 2015) is used on the state input when learning from low-dimensional observation spaces. Obviously, when learning from images, it is not necessary since the values of pixels are in a bounded range. Exploration – one of the most challenging part when dealing with continuous action space – is guaranteed by an exploration policy adding noise to the action provided by the actor-network. To guarantee exploration, the authors used an Ornstein-Uhlenbeck process to sample noise as it fits the characteristics of inertial control problems. We have to note that the noise function should be chosen considering the characteristics of the environment the agent acts in. The approach was tested on challenging environments backed by the physics engine MuJoCo using both low-dimensional state description and stacked images as input.

2.3.4. Proximal policy optimization

Schulman et al. (2017) proposes proximal policy optimization (PPO) algorithm. Its later variant uses the clipped surrogate objective in equation 2.6, to discouraging undesired large changes of $r_t(\theta)$ during the training process, with θ the parameters of the policy, r_t the probability ration between the new and old policies, ϵ threshold and $\hat{A}_t$, the advantage.

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right] \tag{2.6}$$

Alternatively, an adaptive KL divergence penalty is proposed leading to the following surrogate loss:

$$L^{KLPEN}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t - \beta \text{KL}\left[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)\right] \right] \tag{2.7}$$

where β is computed on each update by using the β value from the last iteration:

$$\beta = \begin{cases} \frac{\beta}{2} & \text{if } d < d_{targ}/1.5 \\ \beta \cdot 2 & \text{if } d > d_{targ} \cdot 1.5 \end{cases} \tag{2.8}$$

$$d = \hat{\mathbb{E}}_t \left[\text{KL}\left[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)\right] \right] \tag{2.9}$$

The clipped surrogate objective is proven to outperform the KL-divergence based on the benchmark set investigated in the paper.

The algorithm can be implemented in an actor-critic framework with layer sharing between the policy and value function. In that case, the proposed loss function L^{CLIP} is extended with the loss of the value function (L_t^{VF}: mean squared error) and optionally an entropy bonus to encourage exploration. Then, the surrogate loss is defined using c_1 and c_2 weight coefficients:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}\left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S\left[\pi_\theta\right](s_t) \right] \tag{2.10}$$

Proximal policy optimization (with clipping) outperformed TRPO and advantage actor-critic algorithms in nearly all continuous control environments on the MuJoCo benchmark suite. On the Atari domains (discrete action space) PPO outperformed the ACER algorithm in the metric ”average reward over all of the training”, but showed slightly worse performance when only comparing rewards for the last 100 episodes. Both PPO and ACER algorithms significantly outperformed A2C.

2.3.5. Soft actor-critic

Haarnoja et al. (2018a) introduce the soft actor-critic (SAC), an off-policy actor-critic algorithm for continuous action spaces. Since soft actor-critic (SAC) plays an important role in this work, it is more extensively introduced in section 4.1.

2.3.6. Deep reinforcement learning in partially observable environments

Handling long-term temporal dependencies proved to be challenging with deep neural networks. The field was dominated by recurrent neural network based solutions like LSTMs (Hochreiter and Schmidhuber, 1997) and GRUs (Cho et al., 2014) for long, though it was known that these approaches are limited in modeling long-term dependencies by the single internal state vector needs to store contextual information. Several DRL algorithms employ the discussed solutions to handle partial observability by encoding information about previous states in the internal state of the recurrent network.

Heess et al. (2015) apply the DDPG algorithm on partially observable environments by replacing a feedforward layer with an LSTM layer directly after the input, to process the input observations.

Hausknecht and Stone (2015) introduce deep recurrent Q-learning, attempting to handle partially observable environments without making them fully observable by stacking observations as input. The authors propose an architecture where the first fully connected layer after the convolutional layers is replaced by a fully connected LSTM layer. Then, a fully connected layer outputs the Q-values predicted for each possible action in the discrete action space. Similarly to DQNs, an experience replay buffer is used to train the network with decorrelating samples. However, backpropagation for recurrent neural networks requires several input timesteps to be stored in memory during the training process. Two similar methods have been proposed for selecting experience from the buffer during the training process of the network. Both of them use a target Q-network that is kept fixed for a single batch. Firstly, bootstrapped sequential updates select a random episode from the replay memory and use the whole episode from beginning to end for training. In this case, the hidden state of the RNN can clearly be carried forward from step to step. The alternative method called bootstrapped random updates picks a random point from the selected episode and train the network based on a single backward iteration, while the initial hidden state of the RNN is zeroed before the update making harder to learn long-term dependencies. The latter method has the advantage of fully random sampling (satisfying the assumption made for training DQNs) while handling the hidden state is more straightforward in the former case. Based on experiments, the paper concludes that both strategies can work but use the method starting in a random state for further experiments. Experiments were conducted using a game developed for this specific purpose by making the environment partially observable by blanketing input frames with a probability of 0.5. The recurrent variant of the network was implemented with a single frame as input, meaning that velocities can not be detected by convolutional layers but only with the LSTM layer. It was trained with backpropagation through 10 last frames. The LSTM based implementation was compared to the standard DQN receiving 4 frames and the DQN receiving 10 last frames as input – so having access to the same input sequence as the recurrent neural network-based. The presented results show the significant advantage of the recurrent architecture on the flickering pong game, while concluding that overall the architectures provide similar performances on other Atari environments. Discussing the computational tractability, the authors conclude that the backward pass for LSTMs when unrolling for multiple time steps significantly slows down the training process, making the problem possibly intractable.

Lample and Chaplot (2017) scales up the problem and attempts to solve a more realistic environment, a first person shooter (FPS) game in Lample and Chaplot (2017). Two agents have been trained to handle different scenarios in the game: exploring the environment to discover enemies and fighting them. The model is based on the one presented in Hausknecht and Stone (2015): convolutional layers followed by a fully connected LSTM and one more dense layer before the output (one neuron for each action). Initial experiments showed that the agent was not able to detect enemies based solely on the input frames thus it was never or constantly firing. Therefore, the authors also extend the observation space during training with a boolean array representing whether some game features (enemy, health pack, weapon, etc.) are present on the screen. This information was not available at test time. Also the LSTM itself had no direct access to this information: the presence of the game features were incorporated as target values, and only shared the convolutional layers with the LSTM. Overall, the solution significantly improved the performance of the agent and solved the previously introduced problem of constant firing. An attempt to extend the pure DQN (without recurrence) with the game features was also made, however since in this setting multiple frames need to be fed in as inputs, while the game features should only be predicted for the last frame, the authors observed a significant drop in performance. Summarizing the results, the final agent consisted of 2 separate networks, a DQN without the extra game feature information for exploration, and a DRQN for the action scenarios. To boost training speed, a frame-skip technique was used, similar to the one in Mnih et al. (2015): the agent

only observers every n-th frame, and all actions are performed n times consecutively. For this method, the trade-off between training speed and performance must be considered, since the agent may not be able to perform accurate movements this way. During training, only the states with longer history are backpropagated to eliminate inaccuracies at the beginning of sequences caused by the reinitialization of the hidden state of the LSTM. Experimental results show that the agent is able to outperform human players. However we have to note that human reaction time and targeting accuracy gives unfair advantages to the agent. Comparing agent performance with different training parameters, we can conclude that adding game features to the DRQN saves several hours of training. The number of LSTM updates also affects agent performance: increasing the number of updates makes samples correlated while convergence to a high-performing policy is less likely with only a small number of backpropagation steps.

Transformer networks and self-attention in reinforcement learning Considering the drawbacks of RNN observed in the supervised deep learning setup, Fang et al. (2019) introduce the transformer model Vaswani et al. (2017) in RL context by using its adoption to add memory to a deep Q network constructed to perform advanced robotic tasks. The model receives a high-resolution image of the environment, the pose of the agent, and the last action as an observation. The first maj0or block of the architecture, the scene memory stores embeddings of observations for all timesteps of an episode, so it is possible to maintain dependencies over nearly arbitrary sequence lengths. However, its storage requirement grows linearly with episode length. Embeddings allow storing full episodes without violating memory constraints of devices executing the policy. The policy network receives the observation and the scene memory as input. The content of the scene memory is fed to the encoder, processing all elements in the context of the others with attention mechanisms. Then, the decoder receives the output of the encoder and the current observation and outputs Q-values for all actions. Applying softmax function to this output results in a stochastic policy used by the agent that also guarantees exploration. The model proposed by Vaswani et al. (2017) has a quadratic computational complexity with respect to the number of observation embeddings in the scene memory. Since this is undesired for longer sequences, a memory factorization procedure is used to reduce the complexity to linear by using attention over a compressed memory. The training process involves two stages: first, the embedding is trained separately. Then, the weights of the embedding network are frozen and the rest of the network is trained while keeping only the embedded observations instead of the original full frames. The algorithm outperforms the compared LSTM implementations in three simulated robotic tasks (roaming, coverage, search).

Shen et al. (2019) combine the asynchronous advantage actor critic (A3C) algorithm (Mnih et al., 2016) with self-attention in a network that receives a high-resolution image as input, applies convolutional layers, and feeds the convolved output to a multi-head attention mechanism. There is a residual connection between the input and output of the multi-head attention. The output of the multi-head attention module is fed into a fully connected layer, followed by an output for the policy and the value (thus all but the output layers are shared between the actor and the critic). The model uses ReLu activation functions at the outputs of the convolutional and dense layers with RMSProp (Tieleman and Hinton, 2012) optimizer. Tests were conducted on parts of the StarCraft learning environment, and the agent showed state-of-art performance on one minigame.

2.3.7. Deep reinforcement learning with parallel agents

As an alternative to creating batches of uncorrelated samples by random sampling from the experience buffer, Mnih et al. (2016) presents asynchronous versions of widely used algorithms in Mnih et al. (2016). Their novel idea is gathering samples from agents running in parallel, distinct instances of the environment. The approach makes several on-policy algorithms usable in the deep reinforcement learning context and makes the already dominating off-policy algorithms more robust. It is shown that the parallelized approaches can outperform earlier GPU-optimized architectures on multi-core CPUs for several Atari game domains. In general, multiple agents collect samples in separate instances of the environment running on a single machine. To obtain less correlated samples, agents can run different exploration policies. Asynchronous one-step Q-learning computes the gradient of the Q-learning loss in each environment separately, step-by-step, based on the target model. Gradients are accumulated to mini-batches before updating the model, so distinct learners do not eliminate each other's update. An ϵ-greedy exploration policy with ϵ values periodically changed by random sampling from a

set of possible values proved to improve the robustness of the learning algorithm. As an alternative, the target values can be updated based on the Sarsa approach, leading to the algorithm asynchronous one-step Sarsa. Finally, the asynchronous advantage actor critic (A3C) algorithm is presented. An LSTM based agent was also trained beside the standard feedforward version of A3C in a 3D maze environment. The algorithms were evaluated on a wide variety of domains: Atari 2600 games, continuous action control problems, and 3D environments. The experiments showed that all approaches reached significant speedup – at least proportional to the additional resources accessed – compared to their single-threaded implementations. Moreover, a superlinear speedup can be observed at one-step Q-learning and Sarsa algorithms, probably caused by the positive effect of separate threads on the bias of the methods.

2.4. Safe reinforcement learning

Reinforcement learning has already succeeded in a wide variety of domains ranging from video games to board to games. However, most of its famous applications are in environments where obtaining samples has negligible cost (for example they can be simulated). The results promise successes in other domains like robotics, where the introduction of new technologies (like self-driving cars) requires extremely complex controllers that usually cannot be created by hand. Also, it is suspected that Reinforcement learning can be competitive – and with time it could outperform – for traditional control tasks that usually use simple handcrafted algorithms. Still, real-world deployment opportunities are limited, since agents with low-quality policies during (the initial phases of the) exploration process may take risks that are unacceptable due to damage it may cause. Therefore, a minimum quality should be guaranteed before the execution of a policy in the real world.

Safety itself has a wide variety of interpretations in the reinforcement learning context: hard safety constraints are usually dominant in environments where suboptimal decisions may result in irreversible consequences like directly harming people or devices. In applications with soft safety constraints, some risk can be taken in the hope of long-term improvement, keeping an eye on a budget constraint for an episode or a minimal reward that the agent is expected to deliver during a period of an episode, possibly with a given probability. The goal of safe reinforcement learning is to tackle these issues by analyzing and improving algorithms to consider the safety aspects of the problems, thus avoiding risky situation or keeping the exploration costs within a specific budget. Modifying the optimization criterion can incorporate safety features like the variance of the return. Alternatively, exploration algorithms can also be adapted based on externally obtained knowledge of the domain or the experiences of the agent (García and Fernández, 2015).

Some work focuses on transfer learning Taylor and Stone (2009) which is based on the assumption that learning agents should be able to generalize between different but somewhat related tasks and agents mastering a rewarding behavior in a specific environment will be able to exploit their knowledge in solving other problems as well. For example, models can be trained in simulated environments (safety constraints may be violated) and then deployed with pre-trained policies to real-world. While simulations help to tackle the issue in several cases, it is not always possible to properly simulate all environments where Reinforcement learning would potential be applied, either due to the computational complexity it would require, or the simple lack of the information to build a model (like user interaction domains). Therefore, it is important to make exploration in real-world environments possible.

Robotic domains – a promising application field of deep reinforcement learning – are especially challenging due to their large dimensional observation space (possibly high-resolution images and unprocessed sensor data) and multi-dimensional continuous action space. Peters and Schaal (2008) propose the usage of policy gradient methods in the context, due to their natural support of continuous domains and advantageous convergence properties. Papini et al. (2018) introduces a method for safe exploration with policy gradient methods, aiming to find a balance between safety and exploration for environments with soft safety constraints where conservative approaches may suffer from long-term losses due to their slow learning process caused by the over-limiting exploration. They consider a continuous MDP to be solved by a Gaussian-policy $\pi_\theta(a|s) = \mathcal{N}(\mu_\theta(s), \sigma_\theta^2)$ parameterized by θ. The state space is assumed to be bounded respectively to each feature. The policy is optimized through gradient ascent on a performance measure $J(\theta)$.

$$\theta^{t+1} = \theta^t + \alpha \cdot \nabla_\theta J(\theta^t) \tag{2.11}$$

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau p_0}\left[\nabla_\theta \log p_\theta(\tau)\mathcal{R}(\tau)\right] \tag{2.12}$$

Instead of traditional gradient ascent, the greedy coordinate ascent is used, since Papini et al. (2017) concluded its superior performance. $\nabla_\theta J$ is approximated based on the experienced trajectories. To ensure the safety of a policy update (i.e., with a specified probability the performance of the new policy is minus the performance of the old policy is higher than a given threshold), the step size α is computed considering the volume of the action space, the discount factor, and the current variance. The variance is updated, keeping required exploration in mind since a purely greedy update would tend to decrease variance the policy's behavior changes towards exploitation.

Dalal et al. (2018) present a safety-layer based approach for environments with continuous state and action space. They formulate safety constraints on safety signals in constrained Markov decision processs (CMDPs) (Altman, 1999), corresponding to observable (measured) values of physical processes. Assuming the environment is a physical system with dynamics dominated by processes describable with first-order differential equation systems, the authors exploit that state values change continuously and suggest that effect of performing actions can be observed after a relatively short interval. To ensure safety from the beginning, a safety layer is trained based on data logs before deployment, assuming linear behavior of the system (proportional to performed actions). The layer is trained by fitting a linear function directly, so hyperparameter tuning is not required for this step. Before the training process, data logs are generated considering real-world circumstances, where usually human controllers or handcrafted algorithms act until reaching a critical region where an emergency policy is automatically activated. The agent is placed in the environment randomly and performs random action until constraint violation. Then, the experiences are used to train the safety layer. Before performing an action that would predictably result in the violation of a constraint, the safety layer modifies the action (aiming to minimize the distance between the original and modified action) to keep the system within the safe region. The approach is tested on top of a DDPG model and shows superior discounted returned, compared to the unmodified DDPG algorithm, as well as negligible constraint validations.

Chow et al. (2018) propose a Lyapunov method with safety guarantees to solve RL problems modeled by CMDP. CMDPs can be solved the Lagrangian method by introducing penalties on constraint violation in the reward function and using usual RL methods. Drawbacks of Lagrangian method based algorithms numerical stability issues caused by saddle points and lack of strict safety guarantees. Chow et al. (2018) method extends existing general RL methods to incorporate safety constraints during exploration using an algorithm that formulates a Lyapunov function using a linear programming algorithm from the safety constraints. In CMDPs, safety is defined as a cumulative constraint cost for an arbitrary trajectory, so it does not solely belong to individual states. A policy is considered safe it the cumulative cost constraint is within a given budget.

Chow et al. (2019) extend the method is extended to continuous action spaces and applied with DDPG and PPO algorithms.

Ray et al. (2019) propose a standardization of safety specifications. First, the reward should still relate to task completion and be independent from safety. Second, they propose safety specifications to be encoded as constraints, using CMDPs introduced by Altman (1999). To standardize benchmarking, they develop Safety Gym[1] and propose to evaluate agents based on the the following metrics: performance and constraint satisfaction of the final policy, safety (constraint violation) costs during the training procedure. In CMDPs, the optimal policy is defined by

$$\pi^* = \arg\max_{\pi \in \Pi_C} J_r(\pi) \tag{2.13}$$

with $\Pi_C = \{\pi : J_{c_i}(\pi) \leq d_i, i = 1, ..., k\}$, the feasible set of policies and $J_r(\pi)$, $J_{c_i}(\pi)$ the reward based objective function and the cost-based constraint functions, respectively. d_i is a safety threshold (hyperparameter).

2.4.1. Safe RL in partially observable environments

Bagnell and Schneider (2001) present a POMDP model of a helicopter control problem. They employ policy search limited to a class of controllers. The approach is motivated by the fact that exact planning in POMDPs is computationally intractable, previous research in robotics that showed the near-optimal behavior of simple controllers and that controllers with limited complexity are already regularized. To control the helicopter, they set up a PD controller like neural network decoupled for the roll and pitch axis (10 parameters, 5 per axis) which shows linear behavior around the equilibrium. The hidden

[1] https://github.com/openai/safety-gym

layer helps to efficiently tackle use cases where a non-linear control signal is required. The controller is optimized with the Bayesian Stationary Performance criterion and is tested on multiple simulators before it is deployed to an unmanned aircraft.

3

Problem statement

We assume an environment modeled by a partially observable Markov decision process (POMDP) where the observations received and actions taken from the start of the episode are sufficient to determine the actual state of the environment (thus MDP assumptions hold when the agent gets a long sequence of frames). The POMDP is formally defined as a 6 tuple (S, A, T, R, C, Ω, O). Similarly to a Markov decision process (MDP),

- S is the set of states,
- A is the set of actions,
- $T : S \times A \times S \to \mathbb{R}$ represent the transition probabilities from a state by taking an action to a target state,
- $R : S \times A \to [r_{min}, r_{max}]$ is the reward function.
- Constraint satisfaction evaluated based on $C : S \times A \to [c_{min}, c_{max}]$, the cost function.
- γ conventionally represents the discount factor for infinite horizon problems.

However, in the case of POMDPs, the agent has no direct access to the state. Therefore, we add

- Ω: the set of observations,
- $O : S \times \Omega \to \mathbb{R}$ the conditional observation probabilities.

Figure 3.1 presents the setup. The agent interacts with the environment in discrete timesteps. In each timestep t, it receives an observation $\omega \in \Omega$ based on the actual state $s \in S$ of the environment and O. Next, it performs an action $a \in A$, receives a reward r according to R and proceeds to the next timestep. Upon reaching a terminal state, the agent starts a new episode beginning in an arbitrary state in a new instance of the environment.

The goal of the agent is to maximize the expected return (equation 3.1) from each state s in time step t such that the safety constraint violation costs remain below a given threshold (equation 3.2):

$$\max_{\pi \in \Pi} \mathbb{E}_{\rho_\pi} \left[\sum_{t=0}^{T} r(\mathbf{s}_t, \mathbf{a}_t) \right], \tag{3.1}$$

$$\mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \rho_\pi} \left[\sum_{t=0}^{T} c(\mathbf{s}_t, \mathbf{a}_t) \right] \leq d \qquad \forall t. \tag{3.2}$$

We consider environments with a possible multi-dimensional, continuous, bounded observation and action space.

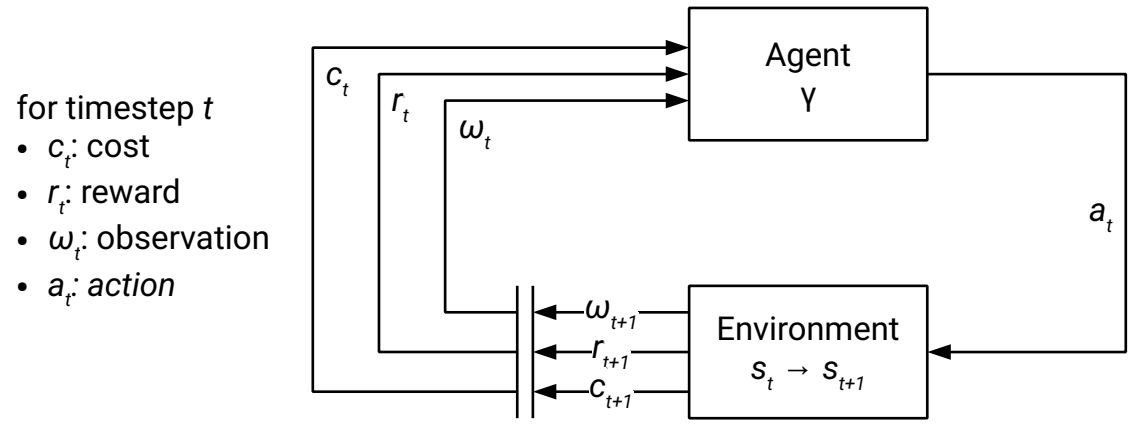


Figure 3.1: Partially observable problem setup with safety violation cost. The environment presents an observation ω_t, a reward r_t and a safety constraint violation cost c_t on each timestep t to the agent. Next, the agent performs an action $a_t \in A$ in the partial observable environment. As a result, the state of the environment changes from s_t to s_{t+1}. The whole procedure is repeated until the end of the episode is reached.

In POMDPs, a single observation may correspond to multiple states of the environment. Therefore, using traditional Reinforcement learning (RL) algorithms designed with MDP assumptions (state = observation) can give arbitrary bad solutions. Therefore, we feed a long sequence of frames as input to the network and assume that to be enough for optimal decision making.

4

Problem context

This chapter provides a more extensive introduction to the previous work that is a direct base of the proposed framework.

As previously introduced in section 3, we consider a partially observable environment with multi-dimensional, continuous, bounded state and action spaces and a bounded reward.

The proposed method is based on soft actor-critic, a recent work of Haarnoja et al. (2018b). This off-policy algorithm has proven to be sample efficient and more stable compared to previous state-of-art methods, which makes it a good candidate for safe reinforcement learning. In fact, the algorithm was already applied in such environment Ha et al. (2020).

4.1. Soft actor-critic

Soft actor-critic can be classified as a maximum entropy reinforcement learning method.

A maximum-entropy RL algorithm trains a stochastic policy that acts as random as possible by optimizing for a combination of long-term return and the entropy of the actor. Table 4.1 presents the augmentation of the objective for the maximum entropy case. The entropy term is introduced with a temperature coefficient α denoting the relative importance compared to the reward (where it is unit). Similarly to the classical RL framework, introducing a discount factor of γ makes optimization possible for infinite horizon problems. The maximum-entropy opens up further opportunities when compared to the conventional objective. It encourages more explanation near the already promising paths, and it is able to capture multiple optimal. Moreover, experiments showed improved learning speed compared to the traditional approach.

$\sum_t \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\rho_\pi}[r(\mathbf{s}_t,\mathbf{a}_t)]$	$\sum_t \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\rho_\pi}[r(\mathbf{s}_t,\mathbf{a}_t) + \alpha\mathcal{H}(\pi(\cdot\|\mathbf{s}_t))]$
Standard RL	Maximum-entropy RL (Ziebart, 2010)

Table 4.1: Objectives in Reinforcement learning. The maximum-entropy goal generalizes the standard goal with an entropy term, with a relative importance α.

The authors derive the soft actor-critic algorithm from soft policy iteration, which is proven to converge to an optimal policy.

Soft policy iteration consists of policy evaluation and policy improvement steps, applied iteratively. The former, evaluation step calculates the value of the policy with respect to the maximum-entropy objective defined in equation 4.1, using a modified Bellman backup operator $\mathcal{T}^\pi$ and the soft value function $V(\mathbf{s}_t)$.

$$\mathcal{T}^\pi Q(\mathbf{s}_t,\mathbf{a}_t) \triangleq r(\mathbf{s}_t,\mathbf{a}_t) + \gamma\mathbb{E}_{\mathbf{s}_{t+1}\sim p}[V(\mathbf{s}_{t+1})], \tag{4.1}$$

$$V_{\mathbf{s}_t} = \mathbb{E}_{\mathbf{a}_t\sim\pi}\left[Q(\mathbf{s}_t,\mathbf{a}_t) - \alpha\log\pi(\mathbf{a}_t|\mathbf{s}_t)\right] \tag{4.2}$$

With equations 4.1 and 4.2 the soft Q-function can be computed for an arbitrary policy by applying T^{π} iteratively to $Q^0 : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$: $Q^{k+1} = T^{\pi} Q^k$ and Q^k is the desired function when $k \rightarrow \inf$ (Haarnoja et al., 2018b).

The policy improvement step updates the policy in a way that guarantees performance improvement (equation 4.3).

$$\pi_{new} = \arg\min_{\pi' \in \Pi} \mathrm{D}_{KL}\left(\pi'(\cdot|\mathbf{s}_t) || \frac{\exp\left(\frac{1}{\alpha} Q^{\pi_{\text{old}}}(\mathbf{s}_t, \cdot)\right)}{Z^{\pi_{\text{old}}}(\mathbf{s}_t)} \right) \tag{4.3}$$

The soft policy iteration algorithm is proved to converge to the optimal policy. However, practically it can only applied in tabular cases.

To make the algorithm suitable for continuous state and action spaces, approximations are needed, resulting in the soft actor-critic algorithm. Both the policy and Q-functions are replaced with function approximators, and since running to convergence is intractable, the algorithm iterates between optimizing the Q function and the policy using stochastic gradient descent.

We have to note that there more alternatives have been presented as soft actor-critic algorithm. Hereby, we consider the one in Haarnoja et al. (2018b) that has no separate value function but soft Q-functions $Q_\theta(\mathbf{s}_t, \mathbf{a}_t)$ and a policy $\pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$ with trainable parameters θ and ϕ, respectively.

The objective J_π (in table 4.1) considers the entropy term $\mathcal{H}$, with a relative importance (temperature) of α.

For the soft Q function, the goal is to minimize the Bellman residual, defined in 4.4. The network is updated according to equation 4.5.

$$J_Q(\theta) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \mathcal{D}} \left[\frac{1}{2} \left(Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - \left(r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \, \mathbb{E}_{\mathbf{s}_{t+1} \sim p} \left[V_{\overline{\theta}}(\mathbf{s}_{t+1}) \right] \right) \right)^2 \right] \tag{4.4}$$

$$\hat{\nabla}_\theta J_Q(\theta) = \nabla_\theta Q_\theta(\mathbf{a}_t, \mathbf{s}_t) \left(Q_\theta(\mathbf{s}_t, \mathbf{a}_t) - \left(r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \left(Q_{\overline{\theta}}(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - \alpha \log\left(\pi_\phi\left(\mathbf{a}_{t+1}|\mathbf{s}_{t+1}\right)\right) \right) \right) \right) \tag{4.5}$$

In equations 4.4, 4.5, $\overline{\theta}$ corresponds to the target Q function which is known to stabilize convergence (Mnih et al., 2015). The target network is a moving average of the actual networks. Additionally, two separate Q networks (and corresponding target networks) have been used to avoid overestimation bias.

The policy is trained to minimize:

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}} \left[\mathbb{E}_{\mathbf{a}_t \sim \pi_\phi} \left[\alpha \log\left(\pi_\phi\left(\mathbf{a}_t|\mathbf{s}_t\right)\right) - Q_\theta(\mathbf{s}_t, \mathbf{a}_t) \right] \right] \tag{4.6}$$

The temperature α can either be a constant or be tuned automatically. The latter is advantageous because an optimal constant value depends on the environment and needs manual tuning for each one. The proposed method adaptively changes the entropy term by decreasing its value where there is less uncertainty about the most advantageous behavior, while encouraging more exploration in other regions. Haarnoja et al. (2018b) introduce a practical method for adjusting the temperature, based on the reformulation of the maximum entropy problem as a constrained optimization problem:

$$\max_{\pi_{0:T}} \mathbb{E}_{\rho_\pi} \left[\sum_{t=0}^{T} r(\mathbf{s}_t, \mathbf{a}_t) \right], \tag{4.7}$$

$$\text{such that } \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \rho_\pi} \left[-\log(\pi_t(\mathbf{a}_t|\mathbf{s}_t)) \right] \geq \mathcal{H} \quad \forall t. \tag{4.8}$$

An upper constraint on the entropy was not introduced, because optimal policies for MDPs tend to be deterministic (Haarnoja et al. (2018b), also verified on figure 7.5). Then, the objective is rewritten as constrained iterated maximization,

$$\max_{\pi_0} \left(\mathbb{E}\left[r(\mathrm{s}_0, \mathrm{a}_0) \right] + \max_{\pi_1} \left(\mathbb{E}\left[... \right] + \max_{\pi_T} \mathbb{E}\left[r(\mathrm{s}_T, \mathrm{a}_T) \right] \right) \right), \tag{4.9}$$

$$\text{such that } \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \rho_\pi} \left[-\log(\pi_t(\mathbf{a}_t|\mathbf{s}_t)) \right] \geq \mathcal{H} \quad \forall t. \tag{4.10}$$

The authors evaluate the approach against following state-of-art deep reinforcement learning algorithms: deep deterministic policy gradient (Lillicrap et al., 2015), proximal policy optimization (Schulman et al., 2017), twin delayed deep deterministic policy gradient and soft Q learning (Haarnoja et al., 2017) algorithms. Experiments in the MuJoCo benchmarks of OpenAI gym (Brockman et al., 2016) concluded that SAC performs similarly to the mentioned reference algorithms. In general, it outperforms the DDPG algorithm by a high margin on all environments and solves environments where the DDPG fails. Regarding the PPO algorithm, the difference in the final policy is less significant. However, SAC requires slightly less samples and learns faster. The latter observation can be supported by the fact that SAC is an off-policy algorithm, thus it can reuse the collected samples contrary to PPO which is on-policy and needs enormous batch sizes to be stable. Interestingly, SAC also shows to outperform the concurrently developed TD3 algorithm, which even fails to converge on some benchmarks.

4.1.1. Real-world training of a walking robot

Ha et al. (2020) present practical application of the SAC algorithm by training a four-legged robot in a real-world environment. Since the experiments are executed directly in the real world it is crucial to guarantee that the procedure takes place without human intervention as much as possible. To that and, a fallback controller is deployed to the robot that can make it stand up if it falls. Additionally, multiple policy networks (for all directions of movement) are being trained by alternating between the policy to train to keep the robot near the center of the training area. To avoid continuous falling of the robot, the SAC algorithm is modified to incorporate constraints. Thus, the problem is formulated as a CMDP as follows:

$$\max_{\pi \in \Pi} \mathbb{E}_{\tau \sim \rho_\pi} \left[\sum_{t=0}^{T} r(\mathbf{s}_t, \mathbf{a}_t) \right]$$
$$\text{s.t. } \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \rho_\pi} [f_s(\mathbf{s}_t, \mathbf{a}_t)] \geq 0, \ \forall t \tag{4.11}$$

The safety constraint, f_s is defined as

$$f_s(\mathbf{s}_t, \mathbf{a}_t) = \min(\hat{p} - |p_t|, \hat{r} - |r_t|) \tag{4.12}$$

where $\hat{p}$ and $\hat{r}$ correspond to the physical tilt limits of the servo motors and p_t and r_t are the actual values. The problem is rewritten to the following form with a Lagrangian multiplier λ:

$$\mathcal{L}(\pi, \lambda) = \mathbb{E}_{\tau \sim \rho_\pi} \left[\sum_{t=0}^{T} r(\mathbf{s}_t, \mathbf{a}_t) + \lambda f_s(\mathbf{s}_t, \mathbf{a}_t) \right]. \tag{4.13}$$

Then, the policy and the Lagrangian multiplier is optimized by alternating with the dual gradient method.

4.2. Constrained optimization using the Lagrange multiplier method

Constrained optimization problems can usually be solved by the Lagrange-multiplier method (Bertsekas, 1982). In our work, the constraints will the take form of inequalities, thus we focus on that case. Let

$$f(x_1, x_2, ..., x_n) \tag{4.14}$$

be the n-variable function to be minimized. Constraints are given in the form of m inequalities:

$$c_i(x_1, x_2, ..., x_n) \leq 0, \qquad i \in \{1, 2, ..., m\}, \tag{4.15}$$

resulting in the following Lagrange function:

$$\mathcal{L} = f(x_1, x_2, ..., x_n) + \sum_{i=1}^{m} \lambda_i \cdot c_i(x_1, x_2, ..., x_n) \tag{4.16}$$

with the Karush-Kuhn-Tucker conditions

$$\frac{\partial L}{\partial x_i} = 0, \qquad \forall i \in \{1, 2, ..., n\}, \tag{4.17}$$

$$c_i(x_1, x_2, ..., x_n) \leq 0, \qquad \forall i \in \{1, 2, ..., m\}, \tag{4.18}$$

$$\lambda_i \geq 0, \qquad \forall i \in \{1, 2, ..., m\}, \tag{4.19}$$

$$\lambda_i \cdot c_i(x_1, x_2, ..., x_n) = 0, \qquad \forall i \in \{1, 2, ..., m\}. \tag{4.20}$$

Maximization problems can be solved with the method by negating the target function and applying minimization.

4.3. Constrained Reinforcement learning

Ray et al. (2019) introduced Safety Gym, a benchmark suite for safe reinforcement learning, along with recommendations on standardization of measuring scientific progress in the field.

4.3.1. Formulation

The authors formulate the problems to be solved in the constrained Reinforcement learning framework. The goal is to find the optimal policy π^* such that:

$$\pi^* = \arg\max_{\pi \in \Pi_C} J_r(\pi) \tag{4.21}$$

where Π_C is the set of constraint-satisfying policies and $J_r(\pi)$ is the general (reward-based) objective function. The set of feasible policies in terms of constraint satisfaction is defined according to the CMDP framework (Altman, 1999):

$$\Pi_C = \{\pi : J_{C_i}(\pi) \leq d_i, i = 1, ..., k\} \tag{4.22}$$

The constraint functions are generalized for long or infinite horizon problems similarly to the reward function, so an J_{C_i} represents an average metric or an per-episode expected cost. An upper limit d_i is specified for each cost constraint with the exploration budget needed to be kept below that limit. Notably, the constraint functions and the reward function are independent of each other, and the reward is kept with its original purpose. Thus, it solely represents task fulfillment.

Based on previous work, it is suggested that a constant trade-off parameter defining the importance of the unsafe penalty cannot be chosen before execution. Additionally, if improperly selected, the agent may fail to learn anything or, depending on the environment, it may learn unsafer behavior compared to an unconstrained agent. Constrained RL is recommended to overcome these difficulties.

Finally, the authors note that the key difference between multi-objective and constrained RL is that in the latter, if the costs are below the specified threshold, further cost decrease is not a requirement. Contrary, in multi-objective RL no such limit exists, and higher returns in any objective are always preferred.

4.3.2. Standardizing benchmarking

The second goal of the paper by Ray et al. (2019) is to provide a framework that makes the outcomes of experiments from multiple authors comparable. Besides providing safety aware reinforcement learning environments across multiple difficulties, methods have been proposed on the way of evaluating performance.

The authors provide a highly-customizable framework based on the MuJoCo physics engine (Todorov et al., 2012) with pre-defined challenges. The interface is compatible with OpenAI gym (Brockman et al., 2016) with a safety signal provided in the info dictionary.

Evaluation criteria The authors use an aggregate cost function and define the optimization problem as provided in equation 4.24.

$$\max_{\pi_\theta} = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} r_t\right] \tag{4.23}$$

$$\text{s.t. } \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} c_t\right] \leq d \tag{4.24}$$

During the training process the following measures are recorded:

- $J_r(\theta)$: the average episodic return
- $J_c(\theta)$: the average episodic cost sum
- ρ_c the average cost over the entire training procedure

The cost rate for a training run is preferred over the sum cost because this way, it is easier to compare multiple training runs with different episode lengths. Still, the cost rate has several drawbacks. For example, it is not possible to differentiate between training runs with a constantly low cost and training runs with an oscillating, possibly many times very high cost, which is less advantageous.

Agents and training runs are compared as follows: firstly, a constraint satisfying agent is always preferred to an agent that fails to meet the constraints. Second, an agent is considered to outperform another if it is strictly better in either in cost or return, and at least as good in the other measure.

To compare algorithms across multiple environments and or random seeds, the normalized return and normalized constraint violation metrics have been computed based on the values from the training runs.

4.4. Recurrent networks

Recurrent neural networks (RNNs) are designed for sequential input processing. The generalization to unseen or variable sequence lengths and parameter sharing across multiple timesteps are their most important features, making them prevailing in sequential tasks. Advanced RNN architectures represent state of the art for many natural language processing problems, although recently, they have been mostly overtaken by attention-based methods.

long short-term memory (LSTM), which is the number one option for processing long sequences along with gated recurrent unit (GRU) and their variants. Notably, GRUs tend to be computationally cheaper. The GRU and LSTM cells are presented in table 4.2.

4.5. Self-attention

Recently, the natural language processing field is dominated by transformer-based architectures, based on self-attention mechanisms. Attention is a mapping of 3 vectors – called query, key and value – to an output vector. Scaled dot-product attention is defined by equation 4.25. There is also additive attention and dot-product attention, which devises only by the scaling factor from the scaled dot-product attention. The scaling factor is required to keep the values in a range where the gradient of the softmax value does not vanish.

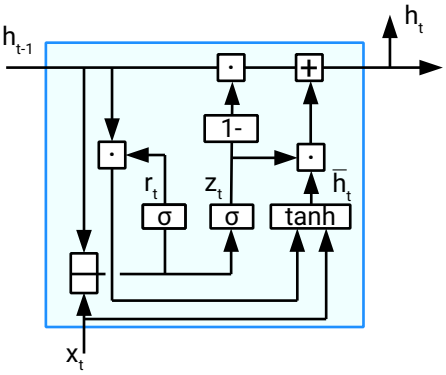
$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{4.25}$$

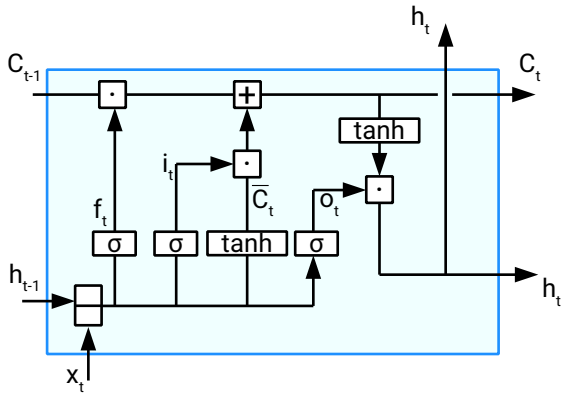
Its advantage is efficient computation using matrix multiplication operations, while the scaling factor makes it less prone to the vanishing gradient problem of the softmax function for large values.

Vaswani et al. (2017) introduced self-attention, an attention mechanism where an input sequence represents the queries (Q), keys (K) and values (V):

$$Q = K = V. \tag{4.26}$$

Thus, self-attention computes a representation of the input sequence. In equation 4.27 $W_i^{\{Q,K,V,O\}}$ are trainable projection matrices for the query, key, value and output vector.





GRU (gated recurrent unit)	LSTM (long short-term memory)
$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$ $r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$ $\overline{h}_t = \tanh(W \cdot [r_t \cdot h_{t-1}, x_t])$ $h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \overline{h}_t$	$\overline{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$ $C_t = f_t \cdot C_{t-1} + i_t \cdot \overline{C}_t$ $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$ $i_t = \sigma \cdot (W_o [h_{t-1}, x_t] + b_o)$ $h_t = o_t \cdot \tanh(C_t)$

Table 4.2: The GRU and the LSTM cells

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{4.27}$$

The purpose of multi-head attention is that multiple attention heads can focus on different positions and representation subspaces in the input sequence. The heads are concatenated afterwards:

$$\text{MultiHeadAttention}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O. \tag{4.28}$$

In the model presented by Vaswani et al. (2017), the projection matrices transform the input vectors into new vectors of lower dimensionality, so that the computational complexity of the multi-head attention is comparable to single head attention with the original query, key and value vectors.

In general, self-attention is invariant to the order of elements in the input sequence. Therefore, in order to use position information in the model, a positional encoding needs to be added to the input sequence.

5

Safe reinforcement learning framework for partially observable environments

In this chapter, we present our method for constrained deep reinforcement learning in partially observable environments.

First, we discuss the selection criteria of the base algorithm we build upon. To minimize exploration costs, a relatively sample-efficient algorithm was selected. From the class of model-free algorithms, on-policy algorithms are generally less sample efficient compared to the off-policy alternatives, since new samples need to be collected for each training iteration. Thus, an off-policy method is preferable. Still, Reinforcement learning is a trial and error method, so the complete avoidance of risk is not possible unless we have preliminary knowledge about the environment to explore. An additional advantage of the off-policy method is that the policy and Q-networks can theoretically be pre-trained on experiences collected in the past by running the system on alternative controllers. Tuning hyperparameters of the deep models and the algorithm may also result in increased costs. Therefore methods having fewer hyperparameters or providing automatic tuning opportunities are prioritized when meeting other necessary criteria. The soft actor-critic (Haarnoja et al., 2018a) showed to be a promising candidate, since it is off-policy, shows stable convergence (Haarnoja et al., 2018b) and already proved in safety constrained environments (Ha et al., 2020).

Given the partially observable problem setup, we need to process multiple observations for optimal decision making. Three deep neural network classes have been selected for this purpose. Our first choice is feedforward network with dense linear layers, each followed by an activation like ReLu. Currently, feedforward networks are the number one option for DRL problems, therefore using them as a baseline is sensible. Additionally, two sequence processing models, namely RNN and self-attention based networks have been selected due to their successful applications in supervised deep learning, especially natural language processing.

To accommodate safety requirements, the soft actor-critic algorithm needs an augmentation, since it was designed to solve unconstrained Reinforcement learning problems. A popular method for constrained optimization that already proved its usefulness in previous safe reinforcement learning methods is converting the problem to its Lagrangian. Then, the policy and the Lagrangian multiplier can be optimized by dual gradient descent (Paternain et al., 2019). Theoretically, the method is suitable to ensure that exploration stays within the budget limit with high probability given a number of assumptions hold.

Next, the proposed method's details will follow, first by introducing the deep neural network architectures. Later, we continue with the constrained soft actor-critic algorithm.

5.1. Sequence processing for partially observable environments

To tackle partial observability of the environments, three network architectures are being considered. The first option is the dense feedforward network. Although they can theoretically approximate arbitrary functions, in natural language processing and other domains where sequential inputs dominate, they tend to be impractical, since they may need to learn the representation of a single features for each

distinct timestep. Therefore, we also investigate architectures specialized for sequence processing, namely recurrent neural networks and self-attention layers.

Sequence length In a partially observable environments, we usually have to rely on multiple observations from the past. Sequence length is defined as the number of observations from the history, that the agent has access to. In general, the last N frames can be fed to the network, but in special cases, it can be beneficial to use another way of sampling to make the modified observation space smaller and still provide information to the agent from a long horizon. The recurrent neural network and the self-attention based architectures are designed for processing sequential inputs, and naturally handle multiple observation in our case. These networks receive an input of shape $(N \times L \times H)$ where N is the batch size, L is the maximal sequence length and H is the number of input features. For the dense-feedforward architecture, the two dimensions are flattened into a single dimension, and it receives an input of size $(N \times L \cdot H)$.

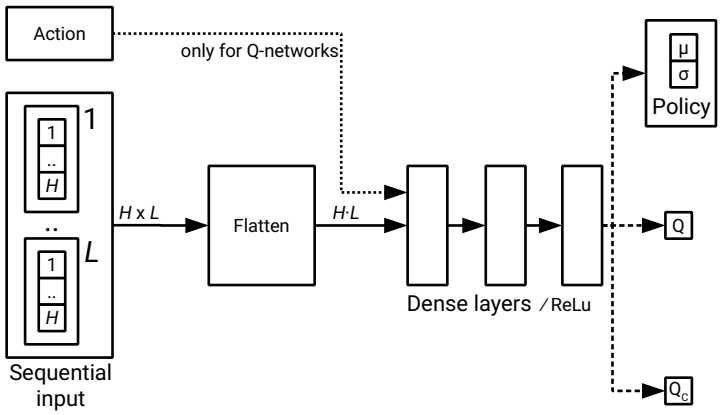


Figure 5.1: Basic multilayer-perceptron (MLP) network with dense layers

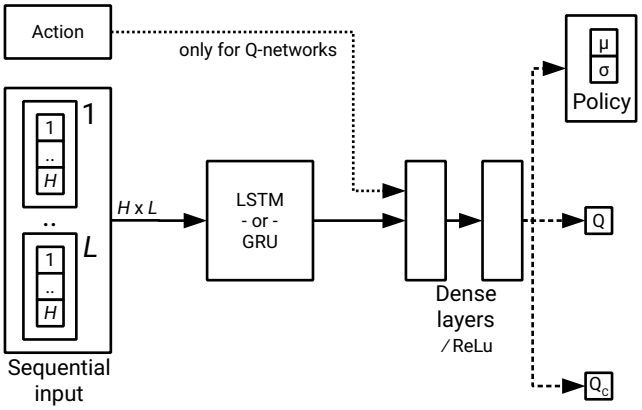


Figure 5.2: Recurrent (LSTM / GRU) based network

Networks We use similar network architecture for the actor (policy), critic (Q) and cost-critic (cost-Q) networks, since they all need process the same observation sequence. Naturally, the network heads differ for the policy and the value networks. For the actor, a normal distribution with a mean and variance is parameterized, while the value networks have a single float output. Additionally, for the Q-network, the recurrent and self-attention based architectures process only the state history, the action is concatenated after the sequence processing layers. Notably, there is no layer sharing between the

policy and Q-networks (the dashed lines in the figures about the network architecture only represent the possible options that our implemented for the networks), because some preliminary experimental analysis showed no advantage of the approach.

5.1.1. Feedforward network with dense layers

The architecture employs a sequence of dense layers followed by ReLU activation function. Naturally, it receives (a batch of) multiple observation frames concatenated after each other. Theoretically, fully connected networks can approximate arbitrary functions, so as the required policy and Q-functions given there are enough layers and neurons. To be successful in partially observable context, the feedforward model is augmented with extra layers compared to a well-performing model on the fully observable problem. The number of additional layers highly depends on the characteristics of the problem and the partial observability. For example, in simpler cases where the velocity is emitted from the observation, but the position is still present, an extra layer can be sufficient to compute the velocity in the first layer and then in the remaining layers build a model comparable to the fully observable variant. Still, we mainly use this architecture as a baseline, as many famous papers used a similar approach with alternative algorithms to solve partially observable environments.

5.1.2. Recurrent Neural Network (LSTM or GRU-based)

We also investigate recurrent network-based solutions. Hereby, the first dense layer is replaced with the an LSTM or a GRU layer. They are intended to process the sequential input and encode a state based on the observations received. The output vector from the last time step is fed into a feedforward network, ending in the required heads for either the policy or the value functions. We still make use of the ReLU activation after each hidden dense layer in the feedforward part (figure 5.2).

5.1.3. Self-attention-based network

In natural language processing, self-attention based architectures are taking over the role of recurrent neural networks. Intuitively, these novel models can also be beneficial in reinforcement learning, especially for image processing or in the partially observable context. In our work, the latter use case is investigated. Similarly to our recurrent model, the self-attention layer replaces the first dense layer of the feedforward model, so it directly processes the input observations.

Observation embedding In our experiments, we deal with low dimensional state-space representation. Thus, an embedding for the observation is not necessary. Therefore, in the networks, we use wide multi-head attention, so the dimensionality of the projected vectors used as input for a single self-attention head is equal to the input features (H). Contrary, for a high dimensional (for example image) input space, or after applying convolutional layers, the size of the input vector can still be relatively large. Therefore, to keep the possibly large number of self-attention blocks tractable, scaling down the input vector with the projections can be considered.

Positional embedding The self-attention mechanisms are known to be permutation invariant with respect to the order of elements in the input sequence. Therefore, an additional positional embedding must be employed to ensure that the network can distinguish the order of frames. Otherwise it would be impossible to predict the moving direction of a car, for example. Vaswani et al. (2017) used trigonometric functions for positional encoding. In chapter 7, we compare the performance of architectures with learned a fixed (sinusoidal) position embeddings.

The network architecture is presented in figure 5.3. The input consists of a sequence of observation with dimensionality $L \times H$ with H the number of input features and L the input sequence length. We have to note that at the beginning of the episodes in the first $L-1$ steps, the sequence of observations is actually shorter. In natural language processing the practice is usually using masking on the input sequences. However, in our case, it is not necessary. We use an alternative solution to fill up the remaining values with zeros. First, since episodes are usually longer than the sequence length, the number of samples involved in reduced sequence length is limited. Additionally, the 0 values can provide extra useful information for the network, so it knows that it is at the beginning of the episode. The position embedding matrix has the same dimensionality as the input matrix. Consequently, they can be summed together by element-wise addition.

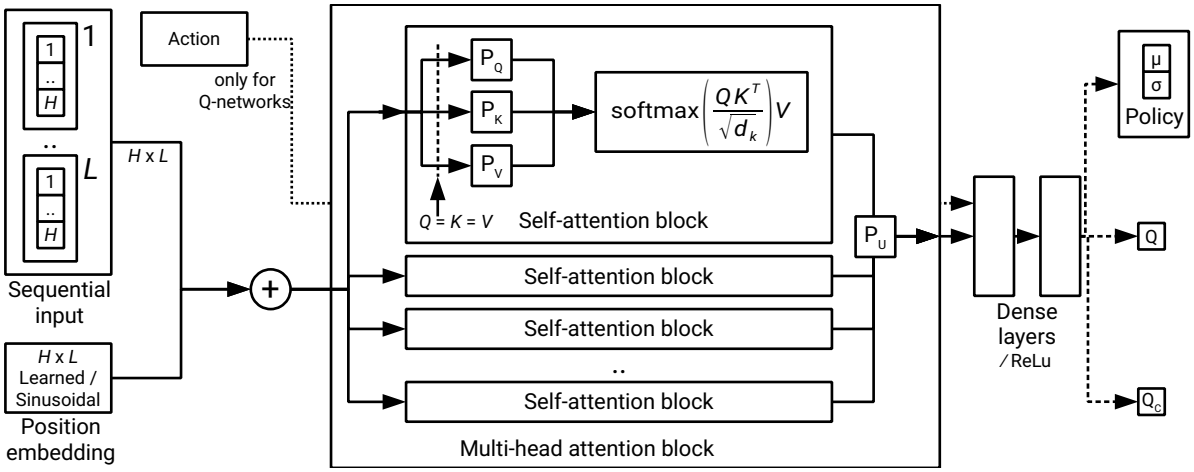


Figure 5.3: Architecture of the self-attention based network. The input sequence is embedded with a learned position embedding before the self-attention layer. The linear layers are followed by a ReLu activation function. The policy and Q-networks use the same underlying architecture except the network heads, however they do not share weights.

In the next step, the multi-head scaled dot-product self-attention is applied. We use *wide* self-attention. Hence, each attention head's dimension will have the same size as the (embedded) observation vector. This is in contrast to the NLP domain, where the input vector is usually scaled down to a dimension l, where $l = d/n$ and d is the input dimension and n is the number of attention heads. In our case using wide self-attention does not lead to computational intractability, since the state representation consists only of relatively few values. Again, we note that for environments with raw image frames as input, the output dimensions of the convolutional layers can be high dimensional. In this latter case, the multi-head attention can possibly work with downscaled inputs. First, the per time step inputs of dimensionality H are projected into an $H \times C$ query, key, value matrices where C is the count of attention heads with a linear projection of parameters P_Q, P_K, and P_V.

The multiple heads provide the opportunity to focus on different parts of the input sequence in parallel. Notably, too few heads lead to faster training, but less accuracy, why too many heads slow down, or even make the training process diverge, due to overfitting.

Residual connection in self-attention based network The original transformer model (Vaswani et al., 2017), where self-attention mechanisms were initially used, has residual connections around the attention layer to preserve useful features from the input or lower layers. Similarly, in our case, a residual link was added from the sequence of input observation around the multi-head attention layer, concatenated to the input of the first linear layer. However, in the later phase of our research, it was omitted during the hyperparameter tuning since it did not improve the performance.

5.2. Maximum-entropy reinforcement learning with safety constraints

Now, we present the augmentation of the soft actor-critic algorithm with a constraint on costs.

We formulate our maximum-entropy constrained reinforcement learning problems as

$$\max_{\pi \in \Pi} \mathbb{E}_{\rho_\pi} \left[\sum_{t=0}^{T} r(\mathbf{s}_t, \mathbf{a}_t) \right], \tag{5.1}$$

$$\text{such that } \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \rho_\pi} \left[\log(\pi_t(\mathbf{a}_t | \mathbf{s}_t)) \right] \leq -\mathcal{H} \qquad \forall t, \tag{5.2}$$

$$\mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \rho_\pi} \left[\sum_{t=0}^{T} c(\mathbf{s}_t, \mathbf{a}_t) \right] \leq d \qquad \forall t. \tag{5.3}$$

Hereby, equation 5.1 corresponds to the goal of maximizing the episodic return, with T the length of the episode and $r(\mathbf{s}_t, \mathbf{a}_t)$ the reward function. Equation 5.2 constrains the entropy of the soft actor-

critic algorithm according to the maximum-entropy reinforcement learning framework (section 4.1). The safety constraint is represented by equation 5.3, and c is the aggregate cost function, thus the linear combination of the possibly multiple cost functions. d is the cost limit, and $\mathcal{H}$ is the minimum required entropy of the policy. While in POMDPs the entropy of on optimal policy can be arbitrarily large, the policy tends to be deterministic in environments where MDP assumptions apply. In our case, the environment is partially observable, but taking sequences of observations as input makes it considered fully observable. Therefore, we expect the policy's entropy to decrease with learning (Haarnoja et al., 2018b). Therefore, similarly to the original soft actor-critic, no upper bound constraint is needed on the entropy (verified on figure 7.5).

We have to emphasize the usage of more permissive safety constraints and their implications. The agent has no preliminary knowledge about the environment, so predicting the outcomes of actions based on a model is not possible. Therefore, during the exploration, strict safety constraints cannot be fulfilled. To keep the problem tractable, instead of strict constraints, we constrain the expected cost below a given threshold, allowing violations as long as the average safety-related performance of the agent is sufficient. Still, it is likely that in the initial phases of the exploration, when the cost-Q function is inaccurate, many violations may occur.

The problem in equation 5.1 - 5.3 can be rewritten to its Lagrangian

$$\mathcal{L}(\pi, \alpha, \sigma) = \mathbb{E}_{\rho_\pi}\left[\sum_{t=0}^{T} r(\mathbf{s}_t, \mathbf{a}_t)\right] - \alpha \cdot \left(\mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t)\sim\rho_\pi}\left[\log(\pi_t(\mathbf{a}_t|\mathbf{s}_t))\right] + \mathcal{H}\right) - \sigma \cdot \left(\mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t)\sim\rho_\pi}\left[\sum_{t=0}^{T} c(\mathbf{s}_t, \mathbf{a}_t)\right] - d\right)$$

with the dual variables $\alpha \geq 0$ and $\sigma \geq 0$, the importance of the entropy and safety, respectively.

Based on the Lagrangian in equation 5.4, the dual function is defined as

$$\mathcal{D}(\alpha, \sigma) = \max_{\pi \in \Pi} \mathcal{L}(\pi, \alpha, \sigma), \tag{5.4}$$

and provides an upper bound on the optimal objective value in equation 5.1. The tightest upper bound can be found by minimizing the dual function (equation 5.5).

$$D^* = \min_{\alpha \geq 0, \sigma \geq 0} \mathcal{D}(\alpha, \sigma) \tag{5.5}$$

Thus, we reach a min-max optimization problem

$$D^* = \min_{\alpha \geq 0, \sigma \geq 0} \max_{\pi \in \Pi} \mathcal{L}(\pi, \alpha, \sigma) \tag{5.6}$$

and use the dual gradient descent (Boyd and Vandenberghe, 2004) to update the primal θ – the parameters of the policy network – and the dual variables α and σ. In the practical implementation, instead of solving for optimality, which is obviously intractable in the introduced RL context, we use approximation. The primal and dual variables are updated iteratively, based on batches of samples and stochastic gradient descent.

With λ_α and λ_σ the learning rates for the update of entropy and safety importance, respectively, the dual variables are updated according to the loss functions in equations 5.7 and 5.8.

$$L(\alpha) = -\log(\alpha) \cdot (\log(\pi_t(\mathbf{a}_t|\mathbf{s}_t)) + \mathcal{H}) \tag{5.7}$$

$$L(\sigma) = \sigma \cdot (d - Q_c(\mathbf{s}_t, \mathbf{a}_t)) \tag{5.8}$$

The Q-networks simply learn the Lagrangian function value, incorporating the reward, cost and entropy values for a batch of samples, and the next state-action pair's value from the target network for bootstrapping. The cost-Q-networks aim to learn the discounted cost for a state-action pair (Q_C), so it can be learned similarly to the reward for a non-constrained RL problem.

The policy is optimized with respect to the Lagrangian (equation 5.4).

$$L(\pi) = -Q(\mathbf{s}_t, \mathbf{a}_t) + \alpha \cdot \log(\pi_t(\mathbf{a}_t|\mathbf{s}_t)) + \sigma \cdot Q_C(\mathbf{s}_t, \mathbf{a}_t) \tag{5.9}$$

Finally, we reach an update similar to the one for the soft actor-critic (Haarnoja et al., 2018b) with the additional constraint on safety and the second dual variable to optimize. Algorithm 1 presents the pseudocode of our method.

For practical reasons, it is easier consider a safety bonus instead of a safety violation cost. We get the former by the negation of the latter. After negation, similarly to the reward, the larger value is more favorable, meaning that the same code can be used to train the cost-related Q-networks as for training the original Q-networks, including the parts of the algorithm that aim to avoid overestimation.

Algorithm 1 Safety Aware Soft Actor-Critic (adapted from Haarnoja et al. (2018b))

Input: θ_1, θ_2, ϕ, Σ_1, Σ_2 *Initial parameters*

$\bar{\theta}_1 \leftarrow \theta_1$, $\bar{\theta}_2 \leftarrow \theta_2$, $\bar{\Sigma}_1 \leftarrow \Sigma_1$, $\bar{\Sigma}_2 \leftarrow \Sigma_2$ *Initialize target network weights*

$\mathcal{M} \leftarrow \emptyset$ *Initialize an empty replay pool*

for each iteration **do**

  for each environment step **do**

    $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$ *Sample action from the policy*

    $\mathrm{s}_{t+1} \sim p(\mathrm{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ *Sample transition from the environment*

    $\mathcal{M} \leftarrow \mathcal{M} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathrm{s}_{t+1}, c(\mathbf{s}_t, \mathbf{a}_t))\}$ *Store the transition in the replay pool*

  end for

  for each gradient step **do**

    sample experience from replay pool $\mathcal{M}$

    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$ *Update the Q-function parameters*

    $\theta_i \leftarrow \Sigma_i - \lambda_S \hat{\nabla}_{\Sigma_i} J_S(\Sigma_i)$ for $i \in \{1, 2\}$ *Update the safety Q-function parameters*

    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$ *Update policy weights (primal)*

    $\alpha \leftarrow \alpha + \lambda_\alpha(\log(\pi_t(\mathbf{a}_t|\mathbf{s}_t)) + \mathcal{H})$ *Adjust entropy temperature (dual 1)*

    $\sigma \leftarrow \sigma + \lambda_\sigma(Q_c(\mathbf{s}_t, \mathbf{a}_t) - d)$ *Adjust safety importance (dual 2)*

    $\bar{\theta}_i \leftarrow \tau\theta_i + (1 - \tau)\bar{\theta}_i$ for $i \in \{1, 2\}$ *Update target network weights*

    $\bar{\Sigma}_i \leftarrow \tau\Sigma_i + (1 - \tau)\bar{\Sigma}_i$ for $i \in \{1, 2\}$ *Update safety target network weights*

  end for

end for

Output: θ_1, θ_2, Σ_1, Σ_2, ϕ, α, σ *Optimized parameters*

6

Benchmark set of partially observable environments with safety constraints

While several popular benchmarks sets are available for fully observable environments modeled by MDPs, there is a lack of standardized suites for POMDP-modeled environments. By searching with popular web search engines, the results mostly provide wrapper frameworks[1] around the OpenAI gym (Brockman et al., 2016) suite or are too simple and have only discrete state and action spaces [2].

Still, in the initial phases of this research, a similar approach is taken to make results comparable to the state-of-art in the fully observable setup. Initial experiments were conducted with the `Pendulum-v0` and `HalfCheetah-v2` environments, as presented in chapter 7.

However, our problem statement and research goals require alternative environments. First, it became clear that partially observable environments are significantly harder to solve. Even for the mentioned simple problems, it required significantly more time to train, which places the problems to the limits of computation tractability, making experimenting with novel architectures and several hyper-parameters hard. Furthermore, running large batches of reproducible experiments takes significantly longer time (multiple days) than desired. Second, the current benchmarks sets have significantly different focus compared to this work. The environments in OpenAI gym (Brockman et al., 2016) and similar suites are fully observable with respect to their observation space – or can be treated as fully observable by neglecting minor details. Therefore, when our wrapper removes the velocity component in the half cheetah environment, it can be easily reconstructed from 2 frames. Although estimating the velocity from the position is theoretically relatively simple, it proves to be challenging in practice (section 7.5). Additional changes, like removing more features or adding random noise, can be made to reach an even more challenging environment, but the general goal of the problems remain the same. Therefore they are not suitable for experiments where the focus is on sequence processing.

To address the mentioned issues, we develop a new benchmark suite focusing on learning sequence representation. It provides environments with various characteristics changing the observation space and the number of observations that need to be processed in parallel to act optimally in the environment. The framework also provides a cost, similarly to the Safety gym in (Ray et al., 2019), and simple heuristics for each problem to collect experience to pre-train our policies on. Additionally, the provided simple problem set corresponds to the use cases introduced in the problem statement. Thus a heuristic corresponding to a middle-performance policy is provided for all problems to make the pre-training of agents possible, based on experiences obtained by legacy controllers. Finally, the suite significantly shortens development time with some problems only complex in terms of sequence processing.

[1] `https://github.com/stweigand/gym-pomdp-wrappers`
[2] `https://github.com/d3sm0/gym_pomdp`

6.1. Requirements for the environments

Before the development of the benchmark suite, we defined several requirements to match our problem statement and speed up the development process of the algorithms.

- **To optimally solve the environment, the agent needs to understand long-term dependencies**: several real-world problems require establishing dependencies through a long time horizon. From the earlier work in deep learning, it is known that processing long sequences is challenging, therefore it is interesting to explore the opportunities in RL.
- **The safety-related goals are clearly separable from the general goal**: While modifying the reward function to incorporate safety-related goals makes the problem solvable by general RL algorithms, more sophisticated optimization is only possible the general goal and the safety requirements a treated separately. This way, at the beginning of the exploration the agent can be allowed to take more unsafe actions to find a rewarding policy. Later the importance of safety can be increased, so the final policy will be less costly.
- **Support pre-training**: In safety critical environments, random exploration is rarely tolerated to due its costs risk. Therefore, we should bundle a policy in the benchmark set, providing an acceptable solution for the environments.
- **Variable sequence lengths**: To evaluate how the methods scale for longer sequences, there is a need for environments with similar complexity level, but changeable time horizon. Therefore the environments should be parameterizable with respect to the sequence length.
- **Standardized action space**: The action space can have arbitrary dimensions, but a meaningful action should always be between -1 and 1. The environments should crop the provided action vector accordingly.
- **Only free, open source libraries to build upon**: Closed source and commercial framework make both short- and long-term reproducibility problematic. Additionally, parallelization across multiple machines is not possible with single-user licenses. Therefore, only free software should be used during the implementation.

6.2. Environments

For a start, four simple environments have been developed some with multiple complexity levels. The project will be available on *GitHub*, with installation instructions using *Python*'s package manager *pip*.

All environments have several instances with multiple sequence lengths pre-registered. After installing the package, the following import makes them available:

```
import pogym
```

Alternatively, it is possible to register an environment with a custom sequence length or other parameters:

```
from gym.envs.registration import register

register(
  id=f'PO{envName}{L}-v0',
  entry_point='pogym.env.{envName.lower()}:PO{envName}',
  max_episode_steps=1000,  # set maximum steps per episode
  kwargs=dict(seq_length=L)  # set sequence length
)
```

Next, the introduction of the implemented environments will follow.

6.2.1. Canon

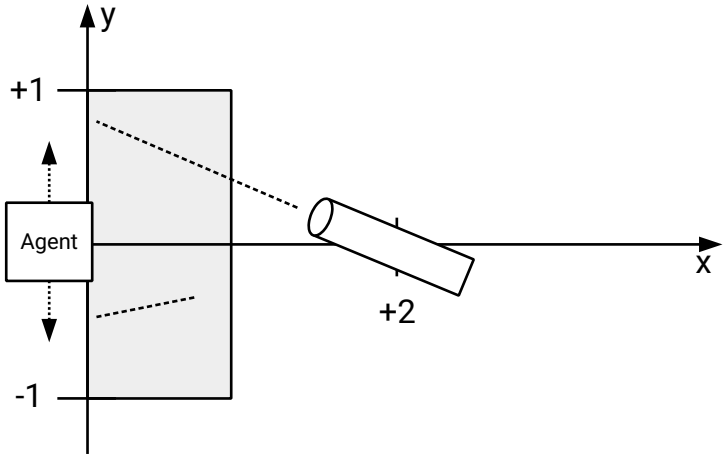


Figure 6.1: The canon environment

A canon is firing balls in the direction of the agent periodically. The ball is visible for the agent until it reaches the gray zone. A ball is fired when the previous one disappears from the observation because of entering the gray zone and going invisible. The agent has to catch as many balls as possible. There is no dense reward provided. The direction and velocity of the balls is determined randomly, the only condition is that the ball has to reach the y axis in maximum L steps. L is a parameter of the environment and it defines the time horizon of the problem. The larger L is, the more observations are needed for optimal decision making.

The environment can be easily modified to add more extra canons with a different reward for catching its ball.

Parameters

- L: time horizon of the tasks, maximum number of steps for a ball to reach the y axis.

Observation space

- x position of the ball between the canon and the gray zone
- y position of the ball between the canon and the gray zone
- y position of the agent

Action space

- v_y: the y direction velocity of the agent / step (max: 1, internally scaled down to 0.1)

Reward The agent receives a reward of 10 for each caught ball. There is no denser reward in the environment, therefore a longer training time is expected.

Cost A penalty is applied when the agent tries to move over 1 or below -1 on the y axis.

Pre-configured environments

- `POCanon`L where L, the sequence length can take the following values: 16, 20, 24, 32, 40, 48, 64, 128

6.2.2. Sail

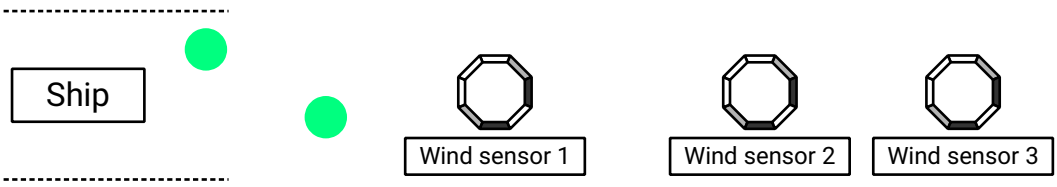


Figure 6.2: The sail environments.

The goal of the sail environment is to keep an under-actuated ship between the dashed lines. There are three wind sensors in front of the ships, sending real-time data (x and y velocity of the air in their position). The power of the wind gusts linear increase until reaching a peak value, followed by a linear decrease to zero. The wind always blows from the direction of the sensors (x dimension), while in y direction it can blow either up or downwards. The magnitude varies in both dimensions. Reward is received for collecting objects flowing in the direction of the ship. Only the ship is affected by the wind, the sensors and objects remain in place. The problem is continuing, however, after a 1000 timesteps it is terminated.

Observation space The observation space contains

- the location of the ship on the y axis,
- the x and y direction values from the 3 wind sensors,
- the relative position of the nearest two objects compared to the ship.

Action space $a \in [-1, 1]$ moves the ship along the y axis.

Reward and cost To facilitate learning, the environment provides both sparse and dense reward. 100 points are awarded for collecting an object. 10 points are deducted if an object is not collected and leaves the screen. The dense reward is received for getting closer to an object and is negative when the agent moves away from it.
The cost increases when the ship touches the sides of the canal.

Discussion The goal and the safety related constraints are clearly separated in this environment. The agent needs to be smart in order to keep away from the borders when high wind is approaching, even though a rewarding object may be missed. The sampling of the simulation is relatively high, so long-horizon thinking is necessary, since predicting wind gusts accurately is only possible from multiple observations.

6.2.3. Growing flowers

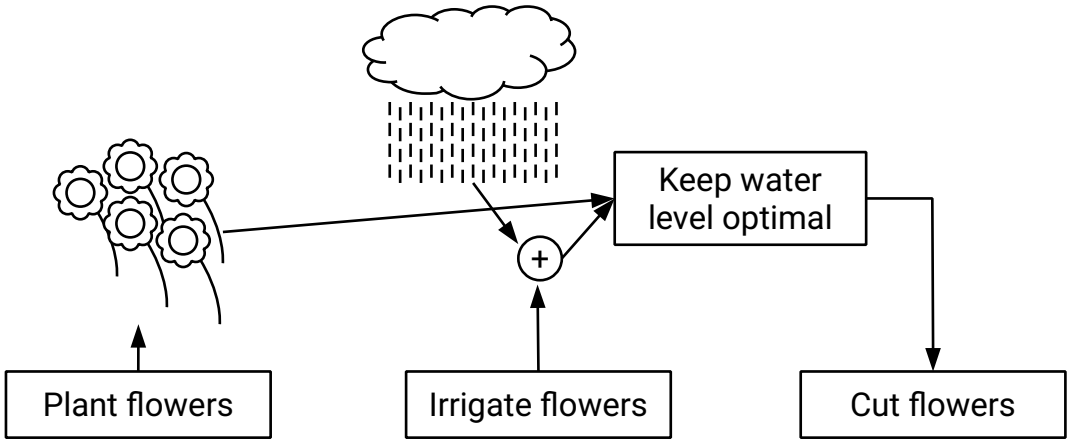


Figure 6.3: The flowers environments.

The environment simulates a flower-growing farm. The goal is to maximize the reward received for grown as much flowers as possible, while keeping the costs of tap water below a given threshold. Incoming rain irrigates the flowers automatically, and the collected rain water can also be used for irrigation later, without incurring any costs. The environment is parameterized by the sequence length L of the problem. The length of the episode also equals to L.

Observation space The observation space contains the following features:

- the number of living flowers,
- the amount of water in the rain water collector tank,
- the actual precipitation,
- the performed action.

Action space The action space has three components,

- flowers to be planted (in the first half of the episode),
- flowers to be cut (in the first half of the episode),
- water to be used for irrigation.

Reward and cost There is a dense reward for planting and growing the flowers. High amount of extra reward is given for cutting mature flowers.

The cost is the price of tap water used for irrigation. As long as there is rain water in the reservoir, the irrigation has no cost.

Levels There are two levels of the environment available. On the first level, only the maturity of flowers is counted in the reward, while the advanced version also considers the amount of water the flowers receive.

6.2.4. MoveNd

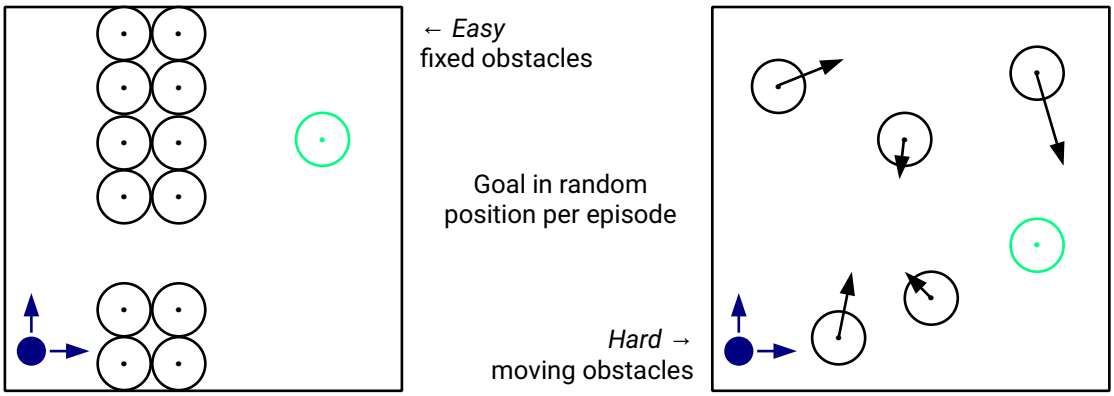


Figure 6.4: The move environments.

An agent and m moving obstacles are placed in a continuous n-dimensional coordinate system bounded by $[0, 1]$ on both axes. The episode ends after 100 timesteps or when the agent gets closer than 0.001 to the goal. In the easy version, the obstacles are at a fixed position, while in the hard version, the obstacles are moving.

Parameters The environment is highly customizable. The following parameters can be used to set it up:

- The number of dimensions.
- Goal position
  - moving with constant speed
  - moving in constant direction with periodically varying speed
- Cost
  - Obstacle-based: getting closer to an obstacle is costly.
  - Velocity-based: changing the velocity of the agent too quickly violates safety constraints.
- Stopping option: the goal stops periodically and the agent should only move to the goal position when it is not moving.
- Inverted action option: every i-th action has inverted effect.

Observation space The observation space consists of the x and y coordinates of the agents and the obstacles, as well as the goal coordinates where the agent should navigate.

Action space The action corresponds to acceleration or deceleration.

Reward and cost The reward in positive when the agent gets close to the goal, is negative when it moves away and is proportional to the distance change in the distance. Reaching the goal gives a one-time reward of 100 and terminates the episode.

The cost is the sum of the distances from all obstacles that are closer to the agent than 0.02.

6.3. Wrapper environments

To obtain further insights on the performance of the algorithm it is important to evaluate against state-of-art algorithms on popular benchmark suites. As mentioned, for partially observable environments there is a lack of widely used test sets. Therefore, the only remaining option is to adapt environments by making them partially observable by hiding features from the observation space and adding a cost function to be used with safety-aware constrained Reinforcement learning algorithms.

6.3.1. `OpenAI gym / Pendulum-v0`

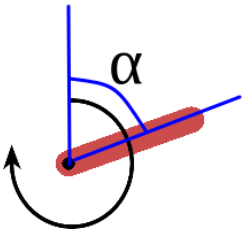


Figure 6.5: Visualization of the pendulum environment

Definition	Visible in fully observable setup	Visible in partially observable setup
$\sin(\alpha)$	yes	yes
$\cos(\alpha)$	yes	yes
$\dot{\alpha}$	yes	**no**
action performed	**no**	yes

Table 6.1: The observation space of the pendulum environment.

The environment is included in one of the most popular benchmark suites, the *OpenAI gym*. In general, this environment is not challanging anymore for state-of-art Reinforcement learning methods. Being one of the simplest continuous environments, a simple dense network solves the problem in some thousand steps and 10 minutes on a middle-grade GPU.

Due to its simplicity, adding a reasonable cost function was not possible, therefore we only the environment for unconstrained tests.

6.3.2. `OpenAI gym / HalfCheetah-v2`

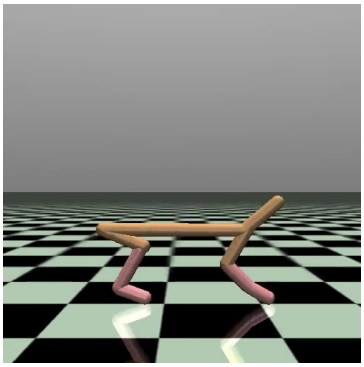


Figure 6.6: Visualization of the half cheetah environment

Identifier	Type	Unit	Visible in fully observable setup	Visible in partially observable setup
rootx	slider	position (m)	yes	yes
rootz	slider	position (m)	yes	yes
rooty	hinge	angle (rad)	yes	yes
bthigh	hinge	angle (rad)	yes	yes
bshin	hinge	angle (rad)	yes	yes
bfoot	hinge	angle (rad)	yes	yes
fthigh	hinge	angle (rad)	yes	yes
fshin	hinge	angle (rad)	yes	yes
ffoot	hinge	angle (rad)	yes	yes
rootx	slider	velocity (m/s)	yes	**no**
rootz	slider	velocity (m/s)	yes	**no**
rooty	hinge	angular velocity (rad/s)	yes	**no**
bthigh	hinge	angular velocity (rad/s)	yes	**no**
bshin	hinge	angular velocity (rad/s)	yes	**no**
bfoot	hinge	angular velocity (rad/s)	yes	**no**
fthigh	hinge	angular velocity (rad/s)	yes	**no**
fshin	hinge	angular velocity (rad/s)	yes	**no**
ffoot	hinge	angular velocity (rad/s)	yes	**no**

Table 6.2: State space of the fully and partially observable variants of the half cheetah environment

To test the method in a simulated robotic environment, an additional wrapper was created for the half cheetah environment from the MuJoCo (Todorov et al., 2012) benchmarks of OpenAI gym (Brockman et al., 2016) . In general, the velocity components are removed from the observation space in the partially observable case, which reduces the number of features from 18 to 9 (table 6.2).

The action space and the reward function remain unchanged. A cost function is added to limit the velocity component `rootx`.

7

Experimental evaluation

The proposed methods have been evaluated on the introduced benchmark set. Since characteristics of the environments (reward shaping, goal, dimensionality of observation and action spaces) differ, we expect the performance of the agents with different network architectures to vary heavily across multiple domains. Additionally, the evaluation can show whether there is a universal method that is capable of learning acceptable policies for a wider range of problems.

We start by introducing the metrics used during the evaluation. Then, the experiments are split into four main parts. First, the results from the fully and partially observable variant of the environments in OpenAI gym (Brockman et al., 2016) are compared. Then, we investigate the satisfaction of safety constraints in partially observable environments. Next, the proposed network architectures are evaluated, with extra focus on the variants of the self-attention based architecture. Finally, we discuss how pretraining with experiences obtained from the execution of alternative controllers affects the learning process.

7.1. Evaluation metrics

The performance metrics are defined as follows:

1. cumulative return (R) and cumulative cost (Z) regret during the training procedure,

$$R = \sum_{e=0}^{E} \sum_{t=0}^{T_e} r(t) \tag{7.1}$$

$$Z = \sum_{e=0}^{E} \sum_{t=0}^{T_e} c(t) \tag{7.2}$$

   where E is the number of episodes the training lasted for, and T_e is length of episode e,

2. average episodic return and cost regret of the mature policies, tested with 100 test runs on the final policy,

3. wall clock time of the training.

The first metric helps us gaining insight into cumulative training costs and rewards. If these measures remain inside acceptable boundaries, the algorithm can be sufficient to be used in real safety-constrained environments, given the appropriate values of the hyperparameters can be predicted beforehand.

In some environments, the length of the episodes may vary highly. Thus, per-step measures may represent the performance of the agents more accurately. The charts with epochs on the x axis present these per-step values since the number of steps in an epoch is fixed. However, the test episode-based charts do not take the length of the episode into account, only the return received and the cost of the episode. These two approaches may provide different results for environments where the reward is

awarded for fulfilling the goal while there is no or insignificant reduction for taking a step. However, in environments, where accomplishing the single goal earlier is more valuable, there is usually a per-step penalty in the reward function.

7.2. General setup

The presented set of experiments were executed on the Azure Cloud with the virtual machine configuration NC6 (with NVIDIA K80 GPU) running Microsoft's Ubuntu-based gen1 deep learning image and the experimental framework introduced in chapter 7.3. Other tests were run on Google Cloud VM with NVIDIA V100 GPUs and Google's Deep Learning VM. Hyperparameter tuning was conducted on the development computer or Google Colab.

Each configuration was executed with three seed values to reach sufficient confidence in the outcomes of the experiments.

The same machine and circumstances have been used for the final evaluation of each environment, with no extra computational load at the time of execution, to make the computational cost of the different architectures measurable.

Initially, many variants and hyperparameters were tested and presented. However, due to training time limitations, further experiments, for example, the constrained variants of the problems, only involved the highest performing variant and hyperparameter combinations. When investigating the network architectures, many charts and tables present results from unconstrained agents, since the unconstrained agent can be trained significantly faster due to the lack of the safety network.

To keep visualizations interpretable, we use abbreviations for the network architectures. MLP stands for the feedforward agent, and SA for the self-attention based network. The latter is also parameterized with the number of heads in the multi-head attention and the type of positional encoding, which is *fixed* in the case of sinusoidal encoding and *learned* otherwise. Naturally, GRU and LSTM stand for the gated recurrent unit and long short-term memory based architectures, respectively.

For some visualization, the RAWGraphs (Mauri et al., 2017) library has been used.

7.3. Evaluation framework for gym-based environments

Reproducible experiments play an important role in deep reinforcement learning research (Henderson et al., 2017). Sharing source codes of implementations of models used for experimental evaluation in papers becomes the case increasingly. Still, accurate reproduction of results usually remains challenging, mainly due to technical questions: both popular benchmark suites and machine learning frameworks are continually evolving, usually raising compatibility issues with obsolete runtimes. Even months after there release of the source codes, it may become impossible to execute the same program on an up-to-date system. Finally, the evaluation of the completed experiments is also problematic, as collecting and plotting data can be a time-consuming procedure.

While there are several popular standardized benchmark suites, there are less universal frameworks to help the tedious process of training and evaluating larger batches of executions. This chapter presents the co-product of the research project, a solution providing practical tools to accelerate reproducible research and evaluation for reinforcement learning problems.

The framework consists of three main components. The first is a Python library, providing a wrapper for environments with the `OpenAI gym` (Brockman et al., 2016) interface that automates logging and data collection. The second part is a web application for data collection, analysis and parallelization across multiple machines. Finally, baseline Docker[1] images help setting up containers to be deployed across multiple computational nodes for a fast and reproducible training process.

7.3.1. Python library

The first building block of the evaluation framework is a wrapper for environments with the *OpenAI Gym* interface. The environment creator takes a *Gym* environment identifier and the length of a training epoch as input and provides generator functions that can generate arbitrary logged instances of the specified environment. The wrapper collects information like a reward, episode length, and elapsed time from the agent's interaction in each step. The data is collected per epoch, training episode, and test episode. Test episodes are handled separately because people may want to use a different policy

[1] `https://docker.io/`

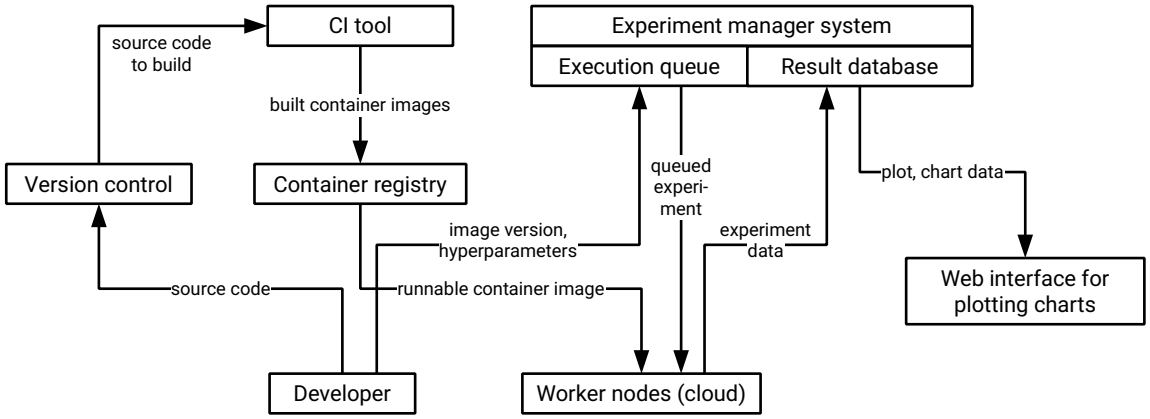


Figure 7.1: Components and data flow in the evaluation system. The developer pushes the new version of the code to the version control system and the hyperparameters and image version for the experiments in the queue. The version control system triggers a build on the continuous integration (CI) tool building a frozen image with the dependencies of the application. Next, the image is pushed to the container registry, from where the worker node can fetch it.

there, for example, a greedy policy instead of an ϵ-greedy policy for value-based methods. Custom data collection – for example, the safety violation costs in our case – is also possible. At the end of a training run, the data collected per episode or epoch is uploaded to a centralized database. Multiple epochs and episodes are stored as an *execution* instance along with the hyperparameters and other properties that make a specific experiment identifiable.

7.3.2. Web application

The web application is separated into a backend and a frontend. The backend provides an long (API) to upload and query experiments. The frontend provides a list of experiments with sorting and filtering capabilities.

The results are stored in a PostgreSQL database. Custom per-episode and per-run properties can be added. Standard features like the obtained reward and the name of the environment are compulsory to define.

Charts can be generated from the web-based user interface, with the automatic summation of multiple executions, if a given filtering condition fits multiple runs.

7.3.3. Process overview

The development and deployment process is explained in figure 7.1 and consists of the following main stages.

1. The developer implements the algorithm and tests it locally.
2. The library is integrated with the algorithm by replacing the environment instantiation with the one from our library. Additional options can also be set at this stage. Optionally, local test runs can also be logged to make experimental analysis faster.
3. Once the codes and testable hyperparameters are finalized, the developer pushes the source codes to a version control system and builds an image with the CI module.
4. Then, the image identifier and additional data about the experiments (hyperparameters, random seeds) are registered on the web.
5. The (possible multiple) runner(s) pull the image and run the experiments with given hyperparameters.
6. Once some instances are finished, the results can be evaluated online, or converted to a LaTeXchart.

Thus, the whole pipeline is repeatable from execution to report generation.

7.3.4. Summary

A proof-of-concept version of the framework was implemented during this project. In the later stages, improvements and new features have been continuously added. Besides providing reproducibility, the application considerably increases the development experience. It allows immediate evaluation of experimental results directly after execution, automates data aggregation across large number of experiments and enables writing re-usable code to (re-)generate charts for LaTeX reports.

All charts and tables in this report have been generated using the prototype and directly imported in the document. The experiments are automatically reproducible by running the corresponding script.

Next, we analyze our experimental results.

7.4. Implementation of the method

Our method was implemented in PyTorch, based on open source projects `pytorch-soft-actor-critic`[2], `former`[3], and OpenAI gym (Brockman et al., 2016) .

7.5. Comparing fully and partially observable environments

We evaluate the performance of the agents on the wrapped environments to compare the performance of our agents with earlier state-of-art results.

7.5.1. The pendulum environment

First, we compare the agents driven by the feedforward networks on the `Pendulum-v0` environment from OpenAI gym (Brockman et al., 2016) and its partially observable wrapper, `POPendulum-v0`. For the latter, contrary to the fully observable setup, the agent has to compute the velocity feature, which is essential for learning a policy of acceptable quality. Therefore, we expect the training to take more epochs in the partially observable environment.

The episodic return for the test episodes (no exploration, deterministic actions) is presented in figure 7.2. The performance of the final policy shows comparable results to the official leaderboard [4] for both the fully and partially observable case. Thus, we can conclude that the policy converges to optimality, and the agents in the partially observable environments are capable of similar performance, after a possibly longer training procedure. However, they do require more complex network architectures: our experiments showed that the dense network with two hidden layers did not converge to the optimal solution during the observed training epochs, while after adding an extra layer, the agent could easily solve the task.

While the training procedure proves to be longer for the partially observable variant, the linear network solves the problems easily with the extra layer added. This is evident from the fact that the omitted components can be estimated from two observations, so advanced sequence processing is not necessary. These preliminary results show the significance of our benchmark suite: for the wrapped environments, there is no need to process a longer history. Therefore, the feedforward network maintains high performance, and the usage of sequence-processing architectures is not necessary.

7.5.2. The half cheetah environment

Next, we focus on the `HalfCheetah-v2` environment, which is a more complex simulation from the robotic domain. It has an observation space with 18 features, where 9 of those are (angular) positions, and the remaining 9 are (angular) velocities. Since the velocity components can be estimated from the positions, they are truncated from the observation space in the partially observable variant. The cost function limits the x direction velocity to 200, which is closely related to the reward. Therefore the safety constraint will prohibit a high reward.

The experimental results in figure 7.3. show that the agent in the partially observable environment gets stuck in a local optimum – with both the feedforward and GRU network architectures – while for the fully observable, it slowly converges to the state of art result [5]. We can also observe that the GRU agent shows more stable behavior across multiple seeds compared to the feedforward.

[2] `https://github.com/pranz24/pytorch-soft-actor-critic`
[3] `https://github.com/pbloem/former`
[4] `https://github.com/openai/gym/wiki/Leaderboard`
[5] `https://www.endtoend.ai/envs/gym/mujoco/half-cheetah/`

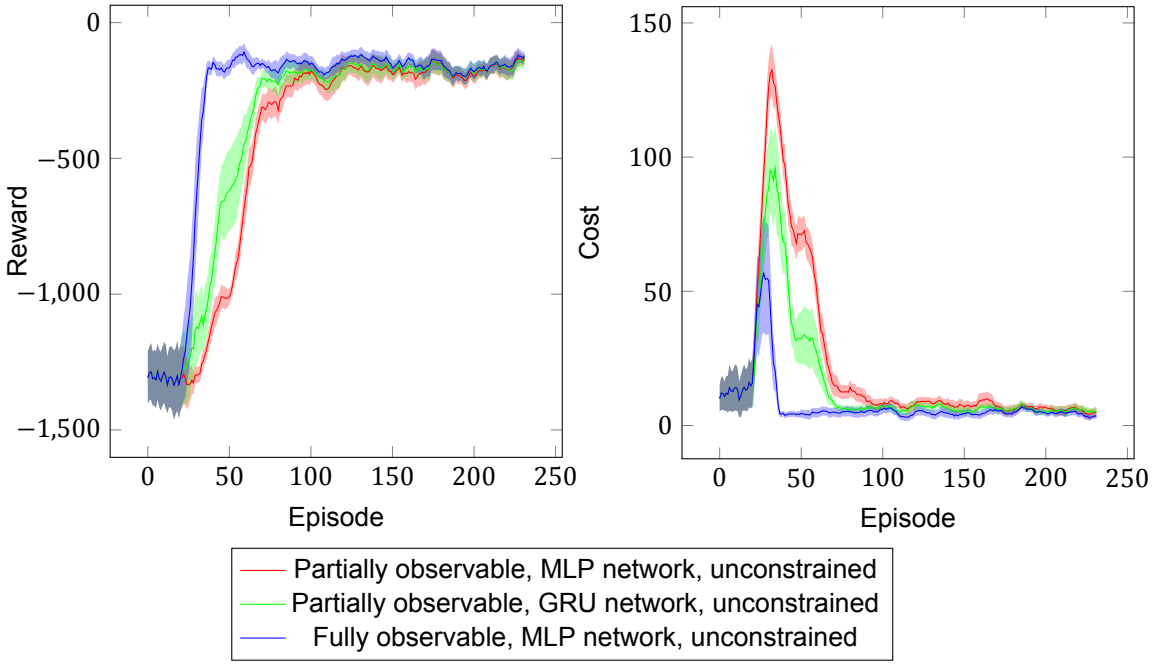


Figure 7.2: Comparing agents on partially and fully observable unconstrained pendulum environments.

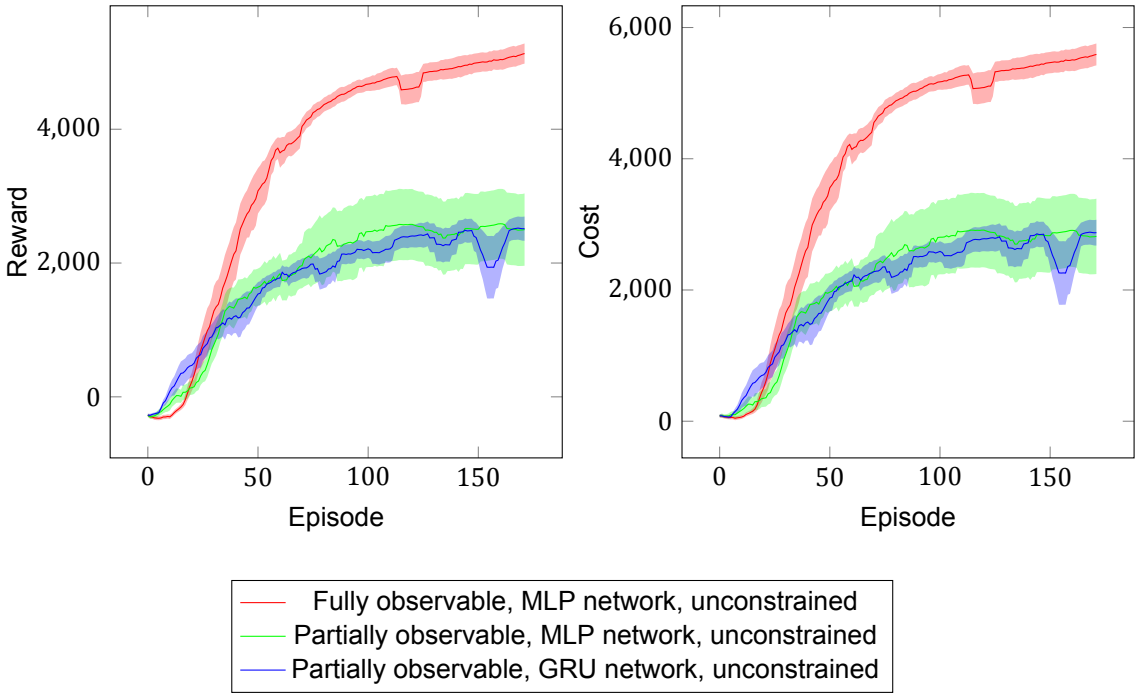


Figure 7.3: Unconstrained agents on the partially and fully observable `HalfCheetah-v2` environments

Constraints in the half cheetah environment By constraining the maximum velocity to 1500 (figure 7.4), all agents remain close to the threshold, however many violations occur, especially for the fully observable agent, where the unconstrained cases see a significantly higher reward. To understand the issue, we look at how the Lagrangian multipliers α and σ change during the training procedure.

The entropy and safety temperature values of a randomly picked training run are presented in figure 7.5. We can observe the importance of safety to increase when the cost constraints are violated. Then, the costs are plunging due to the increased σ value. Thus, we observe an unstable behavior as it cannot maintain a balance between collecting reward and safety.

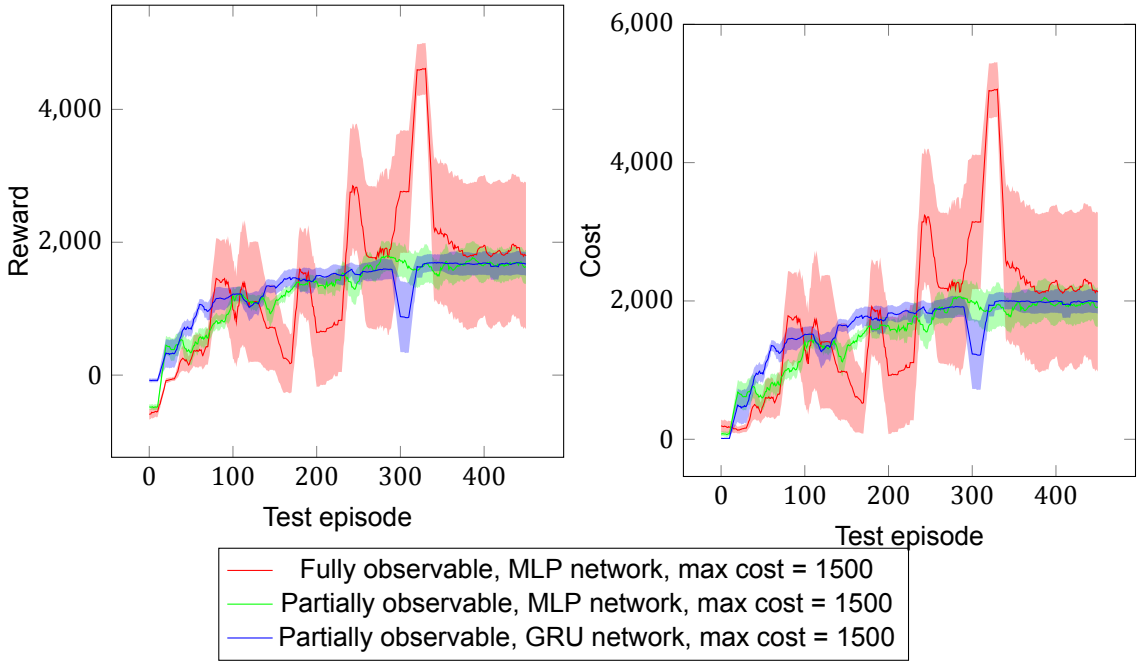
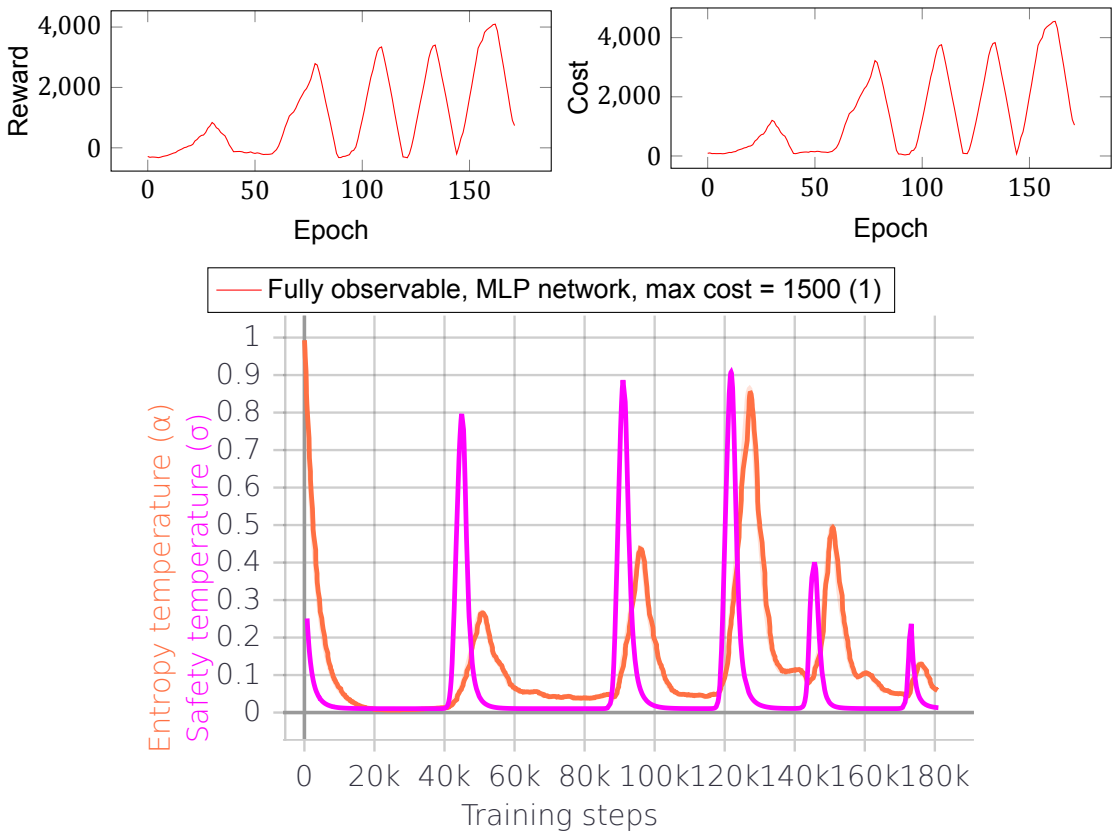


Figure 7.4: Constrained agents the partially and fully observable `HalfCheetah-v2` environments

For the partially observable variant, in the unconstrained case, we have seen the return converging to a suboptimal value. Interestingly, the partially observable agent remains more stable, possibly because the tested combination of hyperparameters is more optimal for that case.

A possible workaround for the issue could be the tuning of the learning rate for σ, the importance of safety.

Concluding our comparison of agents acting in fully- and partially observable environments, we can observe that partial observability makes the problems more challenging, even when two frames provide enough information for full observability. Therefore, investigating longer horizon environments and multiple network architectures proves to be necessary.



(a) The entropy entropy and safety temperatures (α, σ) as the function of the training steps

Figure 7.5: A training run for the safety constrained agent on the HalfCheetah environment

7.6. Comparing network architectures

In the previous section, the pendulum and half cheetah environments have been tested with the feedforward agent in the fully observable case, and with both the recurrent (GRU) and the feedforward agents in the partially observable case. Since the estimation of velocity components are possible based on two observations, the recurrent network showed no advantage over the feedforward on the half cheetah environment. The performance in terms of reward was similar to the feedforward agent, but the GRU based showed lower variance (figure 7.3).

Next, we visit our newly introduced environments, where corresponding to our problem statement – long sequences of observations need to be processed. Most environments are parameterizable with respect to the sequence length, while they are diverse in other characteristics like the density of the reward function, and the shape of observation and action spaces.

7.6.1. Canon environment

First, the agents' performance is analyzed on the variants of the canon environment with multiple sequence lengths. Besides partial observability, the sparsity of the reward also poses a challenge in the environment, requiring the agent to look back for a long horizon to understand why it received the reward. We present average return and costs for sequence lengths of 16, 24 and 40 on the unconstrained `POCanon-v0` environment in table 7.1. The self-attention based architectures use fixed (sinusoidal) positional encoding in this case. The first two columns represent the average per-episode return and cost collected training procedure, respectively. Notably, the test data is based on runs with a deterministic agent. However, during the training, the action is drawn from a normal distribution, and a minimum entropy for the policy network is maintained. Naturally, the higher reward and the lower cost is preferred.

As a baseline (100% performance), we use the feedforward (MLP) network and present relative performances in the tables.

For the shortest sequence length (16), the feedforward network shows convincing performance. The self attention-based agents with more heads (over 10) show similar results. However, the gated recurrent unit network-based agent shows significantly worse performance compared to both alternative architectures on this relatively short sequence length.

Network	Tr. return	Test return
GRU	-23.22%	-27.16%
MLP	+0.00%	+0.00%
SA(4)	-10.49%	-18.12%
SA(8)	-5.28%	-9.20%
SA(10)	+10.13%	+1.20%
SA(12)	+3.36%	-4.38%
SA(16)	-0.46%	-5.10%
SA(20)	+4.36%	-4.40%

(a) `POCanon16-v0`: canon environment with sequence length of 16

Network	Tr. return	Test return
GRU	-1.53%	-3.17%
MLP	+0.00%	+0.00%
SA(4)	-8.13%	-10.99%
SA(8)	-5.03%	-4.27%
SA(10)	-0.72%	+1.91%
SA(12)	-0.62%	+0.31%
SA(16)	+5.88%	+4.79%
SA(20)	-11.45%	-8.50%

(b) `POCanon24-v0`: canon environment with sequence length of 24

Network	Tr. return	Test return
GRU	+27.39%	+69.45%
MLP	+0.00%	+0.00%
SA(4)	-9.58%	-2.19%
SA(8)	-10.14%	-0.94%
SA(10)	+4.94%	+22.96%
SA(12)	-4.40%	+9.73%
SA(16)	-0.54%	+8.19%
SA(20)	-2.81%	+4.20%
SA(28)	+1.19%	+22.40%
SA(32)	-2.36%	+12.18%

(c) `POCanon40-v0`: canon environment with sequence length of 40

Table 7.1: Average return and cost for agents with multiple network architectures on the variants of the `POCanon-v0` environment.

For $L = 24$, the GRU based network performs nearly as high as the MLP with the 16-headed self-attention agent dominating on this sequence length.
Finally, we benchmark a sequence length of 40. In this case, the GRU-based agent significantly outperforms both alternatives. Additionally, with at least than 10 heads, the self-attention based agents also outperform the MLP based agent on the test episodes, while collecting a similar amount of return during training.
Notably, we did not conduct experiments on longer sequences due to extra-long training times. Since the environment has a sparse reward function, training is relatively slow, so each cell in the table 7.1 is based on three training runs for 400 000 steps. A single run takes several hours for the recurrent and larger self-attention based agents even for the sequence length of 40.

7.6.2. Flower environment

Next, we continue evaluating the network architectures on the easy flower environment (figure 7.6 and table 7.2). The environment has a dense reward function, making the agents easier to learn the optimal policy. For the self-attention based network, using learned positional embedding proved to be fruitful. We investigate four different sequence lengths for the environment: 20, 40, and the extreme long 160 and 320.

For shorter sequences, many agents get stuck in local optima. The GRU architecture shows the most convincing performance, while the feedforward network has the worst performance of all. Next, we focus on extra long sequences: 160 and 320. In this case, the clear dominance of the self-attention based network can be observed with all of its variants converging to a high-quality policy quickly. The feedforward agent also shows acceptable performance. However, for these sequence lengths, the recurrent, GRU based network converges slowly and does not show convincing performance at all, compared to the alternatives. Considering execution time for the longest sequence (320), the training of the feedforward agent took about 800 seconds, while it was about 2100 seconds for the two-headed self-attention agent and more than 8000 seconds for the recurrent agent.

Moving on to the hard version, still with the unconstrained agent (figure 7.7). We see similar trends compared to the easy version. The feedforward network still fails to converge, and while the self-attention based network outperforms the GRU on the longer sequence.

Concluding our observations of the flower environment, the recurrent network is the best option for short sequences, contrary to the extra-long sequences where the self-attention based network should be the primary choice.

Network	Tr. return	Test return
GRU	+36.21%	+90.32%
MLP	+0.00%	+0.00%
SA(2, ”fixed”)	-32.39%	+10.14%
SA(2, ”learned”)	+24.63%	+31.30%
SA(4, ”fixed”)	-45.13%	-23.08%
SA(4, ”learned”)	+11.00%	+57.62%
SA(8, ”fixed”)	-51.41%	-91.66%
SA(8, ”learned”)	-44.78%	-13.36%

(a) Sequence length = 20

Network	Tr. return	Test return
GRU	+120.54%	+508.97%
MLP	+0.00%	+0.00%
SA(2, ”fixed”)	-2.25%	+107.17%
SA(2, ”learned”)	+72.42%	+332.95%
SA(4, ”fixed”)	+35.28%	+137.44%
SA(4, ”learned”)	+32.72%	+213.92%
SA(8, ”fixed”)	-30.46%	-62.35%
SA(8, ”learned”)	+39.92%	+246.31%

(b) Sequence length = 40

Network	Tr. return	Test return
GRU	-21.16%	-4.79%
MLP	+0.00%	+0.00%
SA(2, ”fixed”)	+2.88%	-6.52%
SA(2, ”learned”)	+6.78%	+3.26%
SA(4, ”fixed”)	+8.54%	+4.73%
SA(4, ”learned”)	+2.43%	-0.00%
SA(8, ”fixed”)	+6.34%	+3.50%

(c) Sequence length = 160

Network	Tr. return	Test return
GRU	-31.11%	-20.75%
MLP	+0.00%	+0.00%
SA(2, ”fixed”)	+23.55%	+5.90%
SA(2, ”learned”)	+29.90%	+7.52%
SA(4, ”fixed”)	+14.63%	+3.96%
SA(4, ”learned”)	+21.50%	+5.85%
SA(8, ”fixed”)	+11.49%	-12.20%

(d) Sequence length = 320

Table 7.2: Training and test return on flower environment

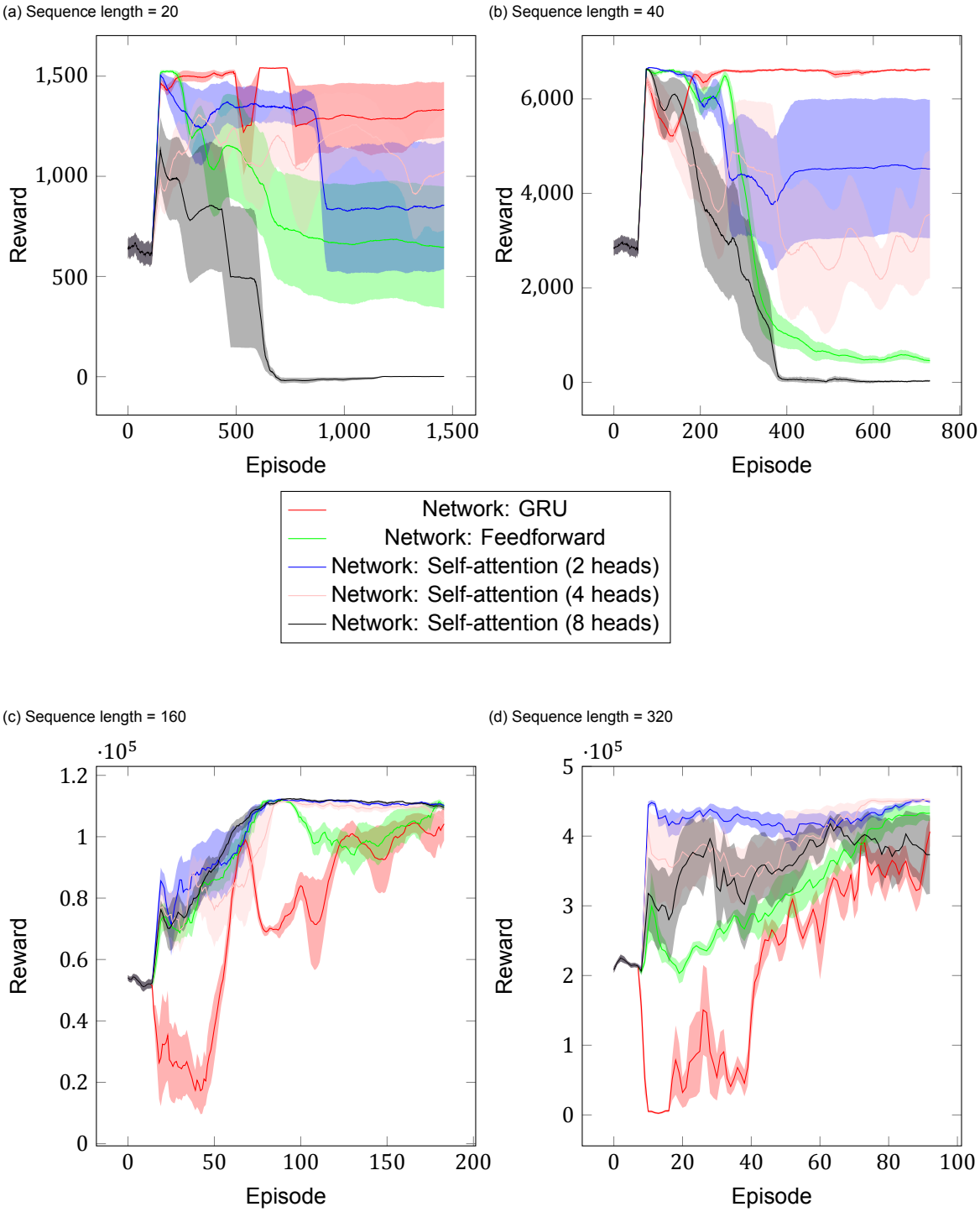


Figure 7.6: Analyzing network architectures for unconstrained agents on the flower environment

7.6.3. Sail environment

Looking at the unconstrained agents in the sail environment (figure A.3), only minor differences can be observed across multiple architectures. All agents converge to a policy of similar quality. The only noticeable difference is that the GRU-based agent does it faster, resulting in a higher return across the training procedure.

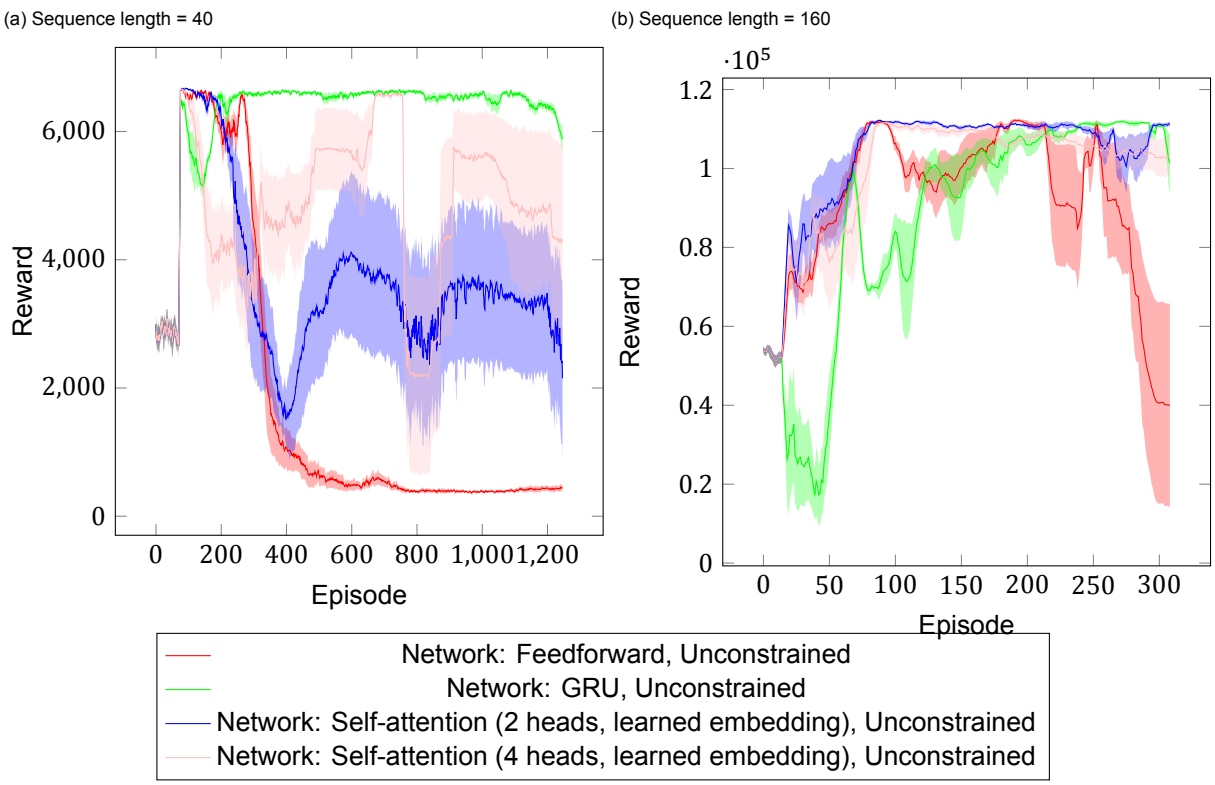


Figure 7.7: Various agent on the unconstrained flower environment (hard level)

7.6.4. Move environment

In the move environment, with the sequence length of five (figure 7.10), the one and two-headed self-attention based agent show slightly higher performance on the two-dimensional tests compared to the average. On the eight-dimensional tests run, a larger attention-based network proves to be more optimal. Still, when starting with random exploration, the feedforward network-based agent seems to show similar performance and considering its simplicity, it should be the first pick on the environment.

7.7. Comparing recurrent architectures

During the preliminary analysis, there was no significant difference between the LSTM and GRU-based agents in terms of performance. Usually, the GRU-based agent slightly outperformed the LSTM based (figure 7.10). Since the GRU layers are computationally more efficient, in many experiments, we opt for it as the only recurrent architecture, due to their extremely high training costs caused by the back-propagation through long sequences.

7.8. Comparing the self-attention based network

In this section, we focus on the variants of the self-attention based network and analyze how the choice of the positional encoding and the number of attention heads affect the agent's performance.

7.8.1. Attention heads

The most important hyperparameter of the multi-head attention network is the number of the heads, which allows the model to attend on multiple representation subspaces in multiple positions of the input sequence. Thus, we expect more attention heads showing higher performance on longer sequences. Revisiting table 7.1, we conclude that the number of heads significantly affects performance. For the shorter sequence length, 16, the ten-headed agent performs the highest, while for longer sequences, a higher number of heads shown to be more performant. The four-, eight-, and twenty-headed networks show poor performance on the benchmark, possibly due to issues related to under- or overfitting.

In figure 7.10 where the multi-head attention based agents are evaluated on the `POMove-v0` en-

vironment. Clearly, the one- and two-headed headed network-based agents show the highest performance. We can explain this observation by the fact that the sequence length is relatively short, and the environment itself is simple, with only a few input features for each timestep. Thus a larger network unnecessarily slows down the training progress.

We conclude that the number of attention heads has a significant impact on the outcome of the training procedure. Therefore, this hyperparameter also requires tuning before training in a real-world safety-constrained environment.

7.8.2. Position embedding

During the preliminary tuning procedure, multiple environments were trained with both fixed and learned embeddings. On the sail, canon, and move environments, it showed to outperform the learned implementation. Therefore, on these benchmarks, the final evaluation was conducted with fixed embeddings. An exception was the flower environment, which is a non-continuing problem with fixed-length episodes. In this case, the agent with learned embeddings seemed to surpass the agent with fixed (sinusoidal) embeddings (table 7.2).

7.9. Safety

After an introductory safety analysis in 7.5.2, we revisit the comparison of the unconstrained and constrained agents. In figure 7.8, we present an unconstrained and two constrained runs on the `POCanon16-v0` environment. Notably, on the environment, a cost of 0 is possible with an optimal policy. Therefore we expect that the constrained agent will show the same performance as the unconstrained. We can observe that the unconstrained agent continuously violates the safety constraints resulting in a cost higher than zero, even when it already converges to the optimal return. However, if we set the maximum cost to 20, the constraints are not violated after the initial exploration phase anymore, while the performance is only slightly affected, and the cost stays far below the limit set. Still, when we maximize the per episodic cost in 1, the agent refuses to explore the environment, after a significant peak in the costs initially. Also, the return of the agent does not converge to the optimal value anymore. By looking at the values in table 7.3, we observe that the cost slightly drops for the agent with a cost limit of 20, compared to the unconstrained one. Theoretically, a cost of 1 would also be accessible if the agent would be allowed to do some initial exploration. For the agent with the maximum cost of 40, we can observe costs similar to the unconstrained version. However, it clearly stays under the defined threshold.

As we have seen on both the half cheetah and cannon environments, constrained agents, the safety requirements may prohibit proper exploration of the environment, leading to the lower return or no convergence. Additionally, depending on the environment, the training and test costs can also increase. However, it is not necessarily the case, because an agent can relatively quickly learn that if it does not start the car, it will not crash it. If there is a higher cost budget for the exploration, which is usually the case, since deep reinforcement learning is a trial-error method, some workarounds are possible. To resolve this issue, we propose that the cost limit should be infinity at the beginning of the exploration. Then, as the policy shows the signs of convergence, it can gradually be decreased to the desired value. This way, the agent can behave safely in production, while it can sufficiently explore its environment.

Next, we visit the flower environment with our constrained agents (figure 7.9). To see how safe exploration works in a real long-horizon environment, we choose the sequence length to be 160. We can observe clear differences between the behavior of the constrained and the unconstrained implementation. In the case of the unconstrained agents, the reward rockets at the beginning, after the random exploration phase. However, for the safe agents, the reward plunges to 0, and the agent only starts to

Maximum cost	Tr. return	Tr. cost	Test return	Test cost
∞ (unconstrained)	7.0713e+01	1.9763e+00	9.9956e+01	2.3737e+00
1	4.4155e+01	1.4140e+00	5.7575e+01	3.2258e-02
20	5.1800e+01	1.4540e+00	7.3899e+01	2.0684e+00
40	6.7985e+01	1.6225e+00	1.0214e+02	3.3621e+00

Table 7.3: Average per-episode metrics after random exploration on the `POCanon16-v0` environment by the feedforward agent

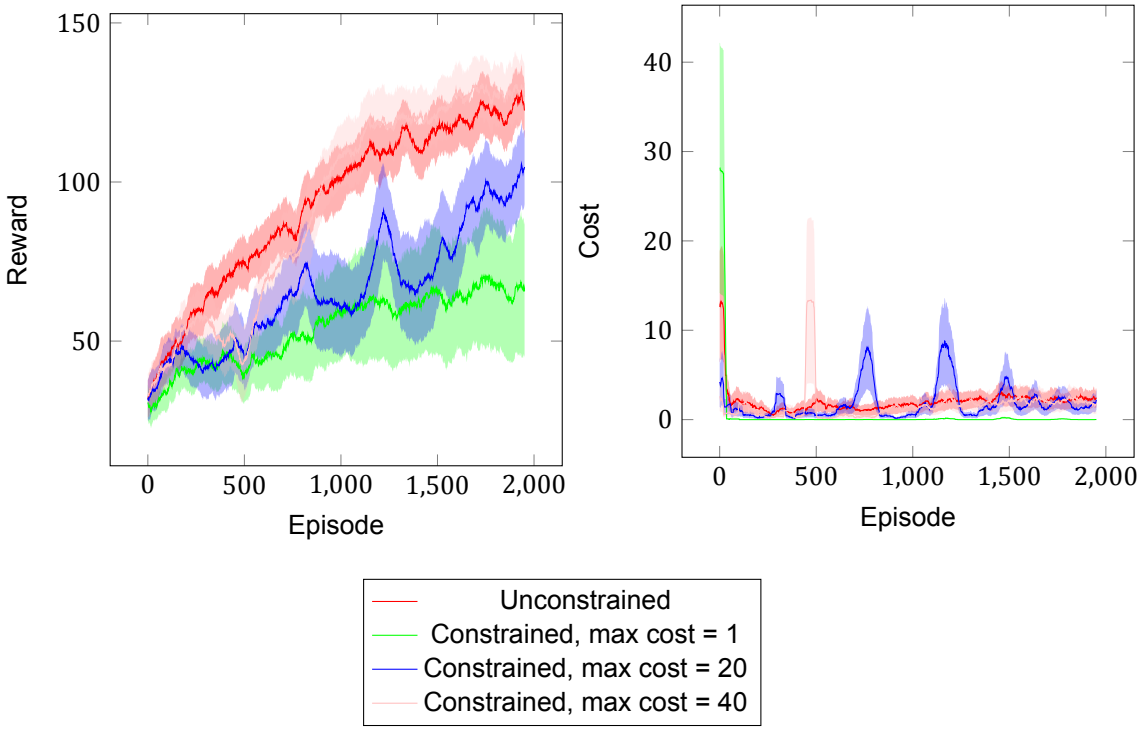


Figure 7.8: Constrained and unconstrained feedforward agent on the `POCanon16-v0` environment

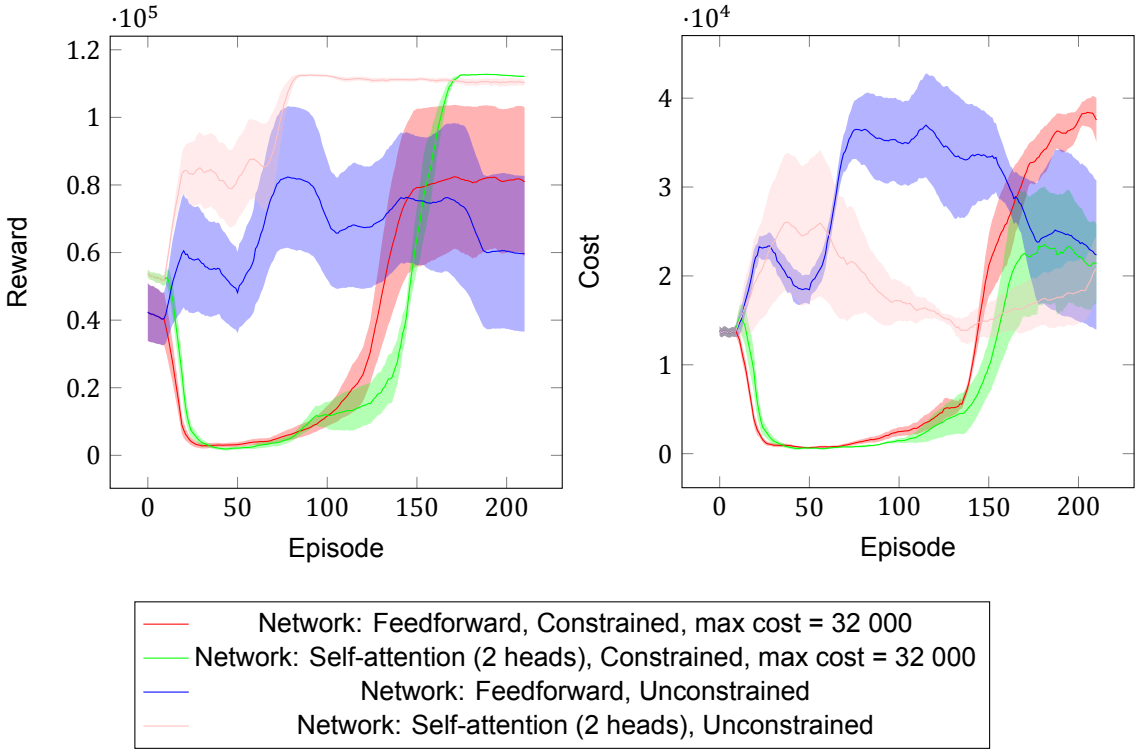


Figure 7.9: Analyzing safety on the flower environment (sequence length = 160)

explore later carefully. For the self attention-based network, it finds a policy with similar performance compared to the unconstrained edition but keeps the reward below the given threshold.

We conclude that safer learning requires sacrifices in terms of return. In critical cases, when the cost limits are strict, or the cost function correlates with the reward, obtaining a stable, constraint satisfying

is challenging and requires careful tuning of the agent's parameters like the cost threshold and the learning rate of the safety temperature σ.

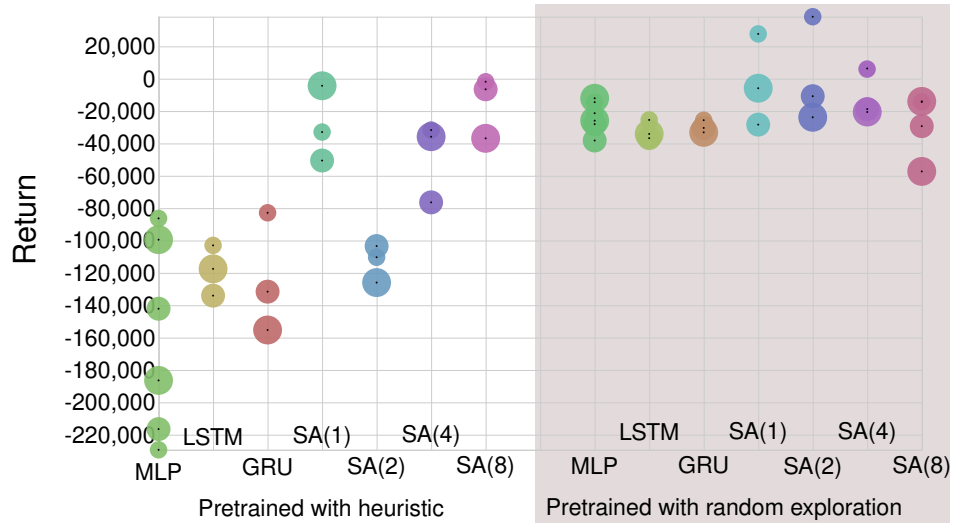
7.10. Pre-training with experiences obtained by legacy controller

In this section, we analyze whether replacing initial random exploration has a positive effect on the learning procedure. The motivation of the approach is the following: in real-world environments, where random exploration is prohibited, making use of a pre-existing controller to give hints may help to significantly increase training return.

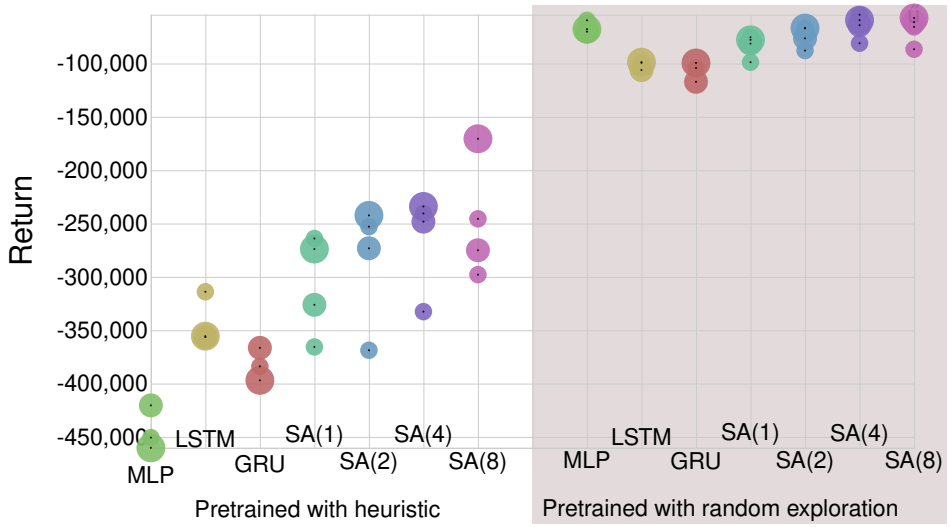
We consider the 2- and 8-dimensional move environments: in the initial phase of the training procedure, the agent collects samples with a heuristic, instead of the random policy. We perform multiple training runs with 10000, 20000, and 30000 steps of pre-training for each network - pre-train policy combination. The environment has the stop parameter set to true, so the agent only gets a reward when it moves over the steady-state goal.

The experimental results are presented in figures 7.10 (a) and (b), for the 2 and 8 dimensional environments, respectively. The corresponding data is also available in tables A.6 and A.7. Contrary to the rest of our report, we also present full return and costs (including the pre-training procedure) in tables A.4 and A.5.

Overall, the agents seem to benefit more from pre-training with random policy at the beginning. If we do not consider the random exploration itself, the return of the exploration process is significantly higher, compared to the agent pre-trained with the heuristic. However, if random samples are unavailable, a significant amount of return is lost with random exploration.



(a) 2-dimensional



(b) 8-dimensional

Figure 7.10: Performance of agents in the move environment. The radius of the points is proportional the length of pretraining.

7.11. Computational complexity

In this section, we discuss the tractability, training time, and computational complexity of the introduced problems and architectures.

As we already concluded in section 7.5, by analyzing the half cheetah environment, partial observability imposes extra difficulty for the agent, even when two consecutive observations contain all necessary information. It becomes the case increasingly for our newly introduced benchmark suite because longer sequence lengths imply that a more careful selection of the input features is necessary. Depending on the environment, some features may need to be extracted from a single, usually the last observation, while for other crucial features, representation learning through the whole sequence may be necessary. Importantly, the mentioned environment has a sparse reward function, so an untrained agent receives a reward only 5-10 times each episode. Still, it is more than one could expect in a real-world environment with no dense reward, where learning without initial demonstrations can be intractable. With a well-tuned hyperparameter combination, the canon environment has been trained for 400k steps (about 6 hours on a high-end GPU) with some agents still showing some improvement. Therefore, a scalability analysis is essential to verify whether the proposed methods can serve in more complex environments. To gain further insights, we run separate complexity tests on a gamer notebook with an 8-core CPU, and a mid-class general-purpose (not deep learning optimized) GPU (NVIDIA GeForce GTX 960M).

First, regarding safety, the constrained agents train additional cost networks, resulting in an about 66% higher training time compared to the unconstrained version, where only the policy and Q networks are trained. This component only adds a fixed overhead to the training time and is irrespective of the sequence length and the number of features.

Comparing the feedforward, recurrent, and self-attention based architectures, we can draw more conclusions from their applicability for large-scale problems. Figure 7.11 presents the training times for the canon environments a time horizon of 2, 4, 8, 16, and 32, and all investigated network architectures. Testing time was not taken into account. We can observe that the simplicity of the feedforward network makes it computationally less expensive. Only slight fluctuations are present in the training time, without any correlation with the sequence length.

On the other hand, both the self-attention and the recurrent architectures show significantly higher training times on longer horizon problems. We can observe the runtime of both the training time of both the LSTM- and GRU-based architectures doubles with doubling sequence length. The self-attention based architecture shows similar behavior, but with a lower growth rate. However, it is known that with optimized hardware, more parallelization is possible compared to the recurrent architectures, and there are also attempts to implement more efficient self-attention variants (Wang et al., 2020) with linear complexity. The training times also grow linearly with the number of heads in the multi-head attention module.

For larger models, the number of trainable weights also has to be considered, since it has to be kept in the device (usually GPU or TPU) memory for the whole training procedure. Therefore we compare the number of learned parameters for our network architectures with various sequence lengths. Similarly to our experiments, the feedforward (MLP) architectures have $k = 3$ dense layers, while the first layer in the recurrent and self-attention based architectures is the sequence processing layer, followed by $k - 1 = 2$ layers. Since the recurrent network architectures use the same weight vector across multiple time steps, the sequence length has no effect on their parameter count. The GRU network has fewer gates compared to the LSTM. Therefore it also requires fewer parameters. Still, the recurrent architectures have about twice as many parameters as the dense and self-attention based network, event for the sequence length of 70. For the dense network, the number of parameters increases considerably, since the number of parameters in the first layer grows with the larger inputs, the flattened sequence observations. Similarly, the input projection has more parameters in the case of the self-attention based network and longer sequences. However, the number of attention heads only has minimal effect on the parameter count.

Concluding our observations, it is clear that the network selection procedure has to take computational and memory constraints into account. Feedforward networks tend to be the primary, computationally cheap option for environments where the sequence process architectures only slightly improve the policy's quality. When keeping the number of heads constant, self-attention based methods have more favorable scaling properties in terms of computational complexity. However, they need more memory for longer sequences, contrary to recurrent networks. The differences between LSTMs and GRU net-

works are only significant in terms of memory consumption. Interestingly the GRU-based agent was only slightly faster to train. In general, optimal selection between sequence processing architectures can only be made when the environment's characteristics are known.

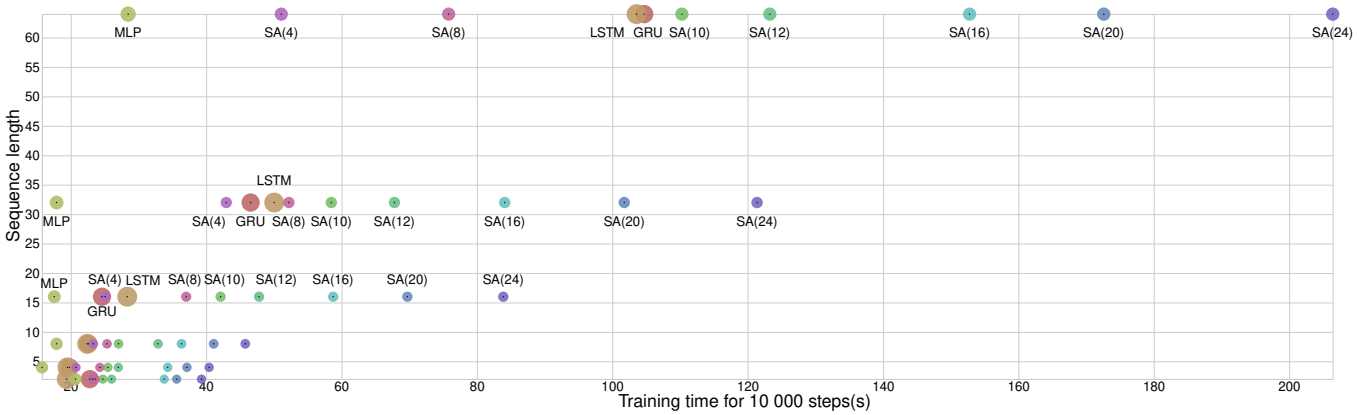


Figure 7.11: Training time for 10000 steps versus sequence length.

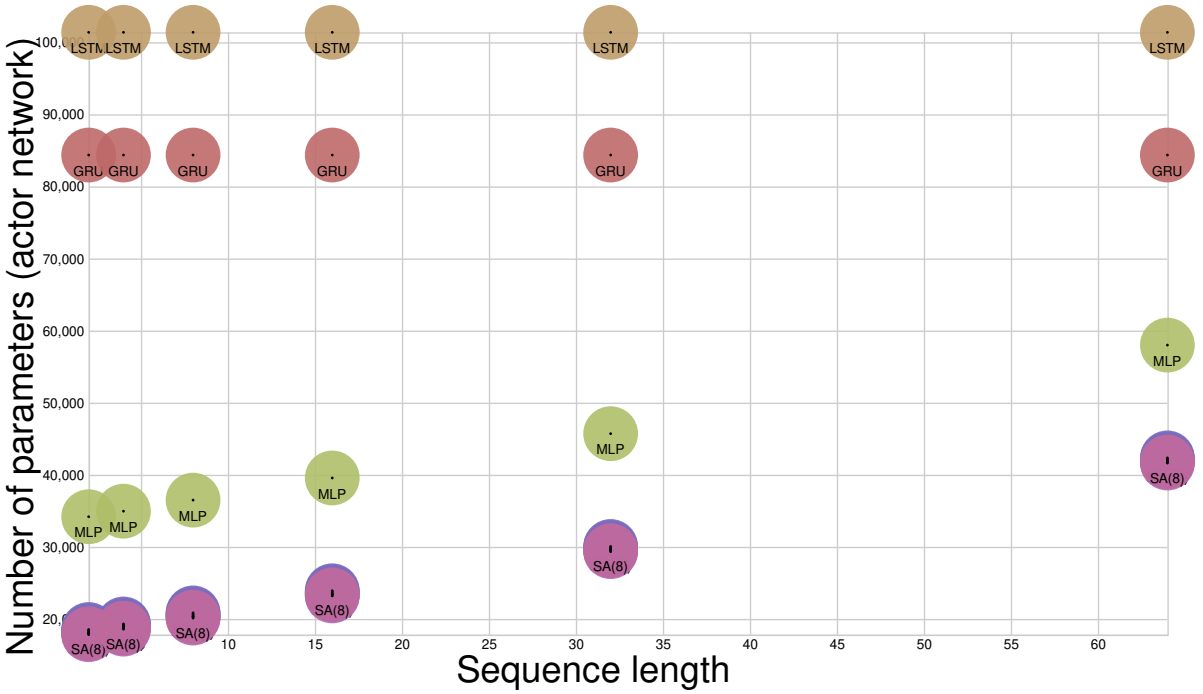


Figure 7.12: Parameter count versus sequence length.

8

Conclusion

8.1. Summary

In this work, we introduced a novel, partially observable reinforcement learning problem context: safety-constrained, continuous environments, where agents have to learn representations of relevant features from a long sequence of observations. A wide range of real-world tasks can be modeled by our problem formulation, including robotic challenges and subsystems of self-driving cars.

Our research questions were defined as follows.

How can deep reinforcement learning agents learn safe policies in long-horizon partially observable environments?

1. *Which novel deep learning models are the best suited for use with deep reinforcement learning algorithms to process long sequences of observations in partially observable environments?*
2. *How to make the selected algorithms safer to minimize risks of taking damaging actions?*
3. *How to conduct a reproducible evaluation of agents designed for safe reinforcement learning in partially observable environments?*

Next, we answer our research questions, starting with sub-research question three.

Our problem statement proved to be different compared to earlier research. Therefore, we introduced a benchmark suite constructed for evaluating methods developed around our research goal. We present four safety-constrained partially observable reinforcement learning environments, where optimally solving the problems requires an understanding of long-term dependencies across tens or hundreds of observations. The time horizon of three environments can be changed by parameterization. Using our benchmark set, we can evaluate problems with similar goals and complexity but varying time horizons. Thus, in-depth research can be conducted on agents designed for handling problems that require the encoding of long sequences of observations. We have also added wrapper environments, to make results comparable to state-of-art results of the OpenAI gym (Brockman et al., 2016) benchmark suite. Experiments showed that agents in the partially observable variants of the environments could be capable of learning policies of similar quality, possibly after more exploration, and adaption in the network architecture.

Now, we are moving towards sub-research question one. We analyzed multiple network architectures, classified into three categories.

First, we looked at feedforward networks. Although they are not engineered for processing sequential input, they are the most widely used architectures for policy and value function estimation in most current state-of-art deep reinforcement learning algorithms. As in previous literature concerning video games, we observe that the feedforward network-based agents show high performance on problems with shorter sequences, or where the agents have to look back to a fixed step in the past. In more

complex, longer horizon environments, their performance plunges compared to other architectures. Still, feedforward agents are computationally more efficient, and on long sequences, the training only takes a fragment of time used to train the sequence processing architectures.

Next, we visited recurrent architectures, namely the long short-term memory (LSTM) and the gated recurrent unit (GRU) networks. Only slight differences are noticeable, both in terms of performance and computational tractability between the two agents. On short sequences, they provide no or minimal advantage compared to the feedforward architecture. We noticed the architectures to perform well on mid-long sequences, providing the highest returns on environments with sequence lengths of about 20 and 40. However, they hardly scale to very long sequences. With a fixed hidden size, we observe a huge drop in performance, although it happens on much longer sequences than in the case of the feedforward network. Increasing the number of hidden units is an option; however, due to the extremely high computational complexity and problems with parallelization, it is not an option after a while.

Finally, we analyzed the self-attention based architectures. We tested multiple options for position embedding: the fixed, sinusoidal function-based, and the learned. Their applicability varies across environments. In some cases, we observed that more attention heads are required for longer sequences, and the parameter has to be carefully tuned to maximize performance. On extremely long sequences, the agents with the multi-head self-attention based network dominated, outperforming both the recurrent and feedforward agents. The idea of a residual connection was dropped in the initial phase of development, and revisiting it in a later phase of research, it did not improve on the performance.

Concerning sub-research question two, the implemented Lagrangian safety approach was verified to fulfill constraints in partially observable environments, when safety constraints did not contradict the reward. The tuning of hyperparameters like the learning rate of the safety temperature also proved decisive. However, partly due to the nature of model-free deep reinforcement learning, we cannot state that the algorithm would be entirely safe. Constraints may limit exploration, even when a high-performing policy is reachable within the cost budget. If we get lower quality policies, caused by safer exploration, it usually leads to significantly lower training return. Long-term cost can also increase due to the suboptimality of the policy, but it depends on the environment. Still, the behavior of the agent with safety constraints mostly depends on the environment's characteristics.

Exploring further approaches for safety should definitely be considered in the future.

We conclude our work by answering the main research question. With the presented network architectures, we succeeded in training agents in long-horizon, partially observable environments. In problems where the return and cost can be balanced reasonably, the constrained agents proved to act more safely compared to the unconstrained ones.

8.2. Drawbacks of the method

Overall, the experimental evaluation showed that all three architectures are equally sensitive to hyperparameters. However, it is also the case for other state-of-art reinforcement learning methods, as it was concluded in the initial phases of the project. In our case, it is an advantage that both the safety and entropy temperatures are optimized on-the-fly and do not require tuning for a different environment.

Defining the reward and cost functions also remain in issue, because experiments showed that a wrongly chosen reward function could be as harmful as suboptimal hyperparameters. Thus, for more complicated problems, a well-designed dense reward function remains important.

For hyperparameter tuning, simulators may provide solutions, when the simulated environment only negligibly differs from the real world. Additionally, simulations may help to test pre-trained policies before they are executed in the physical systems.

A completely different issue is related to safety. While the safety feature showed promising results for some environments, in some cases, it proved to be ineffective, and it even made the exploration more costly by making the policies failing to converge to the optimal solution. We could observe this behavior even though the constraints did not contradict the reward-related goal. Therefore, a more robust safety approach could significantly improve the algorithm.

As long as the hyperparameter-related issues persist, the possibilities of real-world deployment are limited, since the cost of hyperparameter tuning outweighs the costs of the final training procedure of the agent.

8.3. Future work

While we aimed for a comprehensive study covering multiple architectures and experimental evaluation of multiple environments, several interesting points have not been covered. This section presents future research opportunities based on our work.

8.3.1. Experimental evaluation

First, we propose topics related to practical applications and further experiments.

Scaling up for larger environments In this work, the presented methods were evaluated based on small environments that are relatively easy to solve in a fully observable setup, and the challenge lies in the partial observability and proper encoding of the state based on a (long) sequences of observations. This decision was driven by the fact that the complexity of more advanced benchmark suites requires significantly more computation power than it was available. In state-of-art NLP models, advanced versions of the transformer model use tens of layers with more than one billion parameters (Radford et al., 2019) to produce state-of-art results. However, it also requires high-end training infrastructure with state-of-art GPUs or TPUs to keep the problem computationally tractable. However, in our case, large scale experiments were not conducted, due to the constraints in computational resources and the fact that the introduced simple environments already required 400 thousand steps until convergence on the canon environment. Therefore, it would be interesting to experiment on large-scale robotic simulations, where both general and partial-observability related complexity is high, and using large, multilayer models pay off.

Real-world experiments Partial observability and safety usually appear naturally in real-world environments. Our experiments showed a prevalence of self-attention based agents in environments with many input features and when the networks were pre-trained using experience from legacy policies. While self-attention based architectures are computationally more expensive than basic networks for dense layers, in physical systems, the neural network training costs can negligible compared to operating the plant to be controlled. Therefore, further experiments on complex, high-dimensional physical systems can definitely form a subject of further research.

8.3.2. Method

Besides more experimenting, architectural changes can also improve performance, thus worth further exploring.

Improving the self-attention based architecture The transformer model and its variants employ many multi-head attention modules after each other, with dense layers in-between. To allow higher abstraction levels in representation learning, a similar solution could be used on more complex environments.

Network architectures While we have seen the advantages of self-attention based and recurrent network architectures, some experiments showed that the simplicity of dense networks can lead to faster learning. Still, recurrent and self-attention based architectures showed advantages on longer sequence length. However, the computational complexity of the sequence processing architectures can be prohibitive for real-world use cases with thousands of timesteps to be processed. Therefore we propose to study computationally cheaper sequence processing architectures, like the linear self-attention mechanism proposed by Wang et al. (2020).

Safety While the Lagrangian safety approach proved to be efficient in some of our application cases, there are multiple novel approaches in the latest literature with more promising characteristics and performance. Since the current method proved to cause unstable learning in multiple cases, we propose the implementation of an alternative safety method before deployment to real-world environments.

Discrete action space Partial observability and long-range dependencies may occur in tasks with discrete action space. For example, a 2D labyrinth environment can be a challenging context. The

agent moves in four directions, and it has to collect treasures in a specific order before leaving the maze. In this case, both the walls and the already passed checkpoints must be remembered.

Storing encoded state in the replay buffer In the current approach, we store a sequence of observations in the state field of the replay buffer. While it is memory-wise tractable with our benchmark suite, where the agent receives relatively few features, in real-world scenarios where agents have to work with raw high-resolution data (like image frames), storage capacity problems can occur. As an alternative, storing embedded observations can save a significant amount of memory, but important encoding features before training is challenging.

Comparison with alternative algorithms Deep recurrent Q-network was designed for discrete action spaces. Therefore they are not applicable in the current context. However, our method could be compared to a recurrent network-based DDPG or a variant of the A3C algorithm.

Improving the batch reinforcement learning technique In the current implementation, we only train from experiences obtained by alternative policies at the beginning. The experimental evaluation showed that in many cases, replacing the initial exploration by training on experiences with previous policies makes the training process more costly. Therefore, it can be considered to mix experience from the random agent and a well-performing policy at the beginning.

Additionally, since the policies can get stuck in local minima, especially due to safety constraints, demonstrating optimal behavior may help in the later stages of the training too.

We could use replay buffer sharing, where the replay buffer is split into two parts, and one is filled with experience form the legacy controllers while the rest is filled with the on-policy samples. Alternatively, we can alternate between training from two separate buffers by selecting the off-policy buffer to train for a single epoch after periods of training.

A

Appendix

In the appendix, we provide supplementary material, including network hyperparameters and additional tables used in the experimental analysis section.

A.1. Hyperparameters

This section lists the hyperparameters of the experiments we presented in chapter 7. Table A.1 presents the common hyperparameters, applied to all experiments. Some hyperparameter values were customized for each environment to optimize performance and shorten the training times, where it was possible. The custom values are presented in table A.2.

Note that the temperatures α and σ are learned, based on the constraint on the entropy and max cost. The minimum entropy is defined as $|\mathcal{A}|$, the dimensions of the action space (Haarnoja et al., 2018a), while the maximum cost is given for each constrained experiment and not treated as a hyperparameter.

Parameter	Value	Note
Replay buffer size	Unlimited	All experiences are kept
γ	0.99	Discount factor
τ	0.005	Target smoothing coeff.
Batch size	256	
Epoch length	10000	
Evaluation episodes after each epoch	20	With deterministic agent
Evaluation episodes after training	100	With deterministic agent

Table A.1: Hyperparameters applicable for all experiments

Parameter	Pendulum	POPendulum	Half cheetah	Flower	MoveNd	Sail	Note
Size of hidden layers	64	64	256	128	32, 128	32	both for MLP and recurrent layers
Count of MLP layers (FF network)	2	3	2	3	3	3	
Count of MLP layers (SA, recurrent networks)	-	2	1	2	2	2	
Learning rate for policy and Q-functions	0.001	0.001	0.001	0.001	0.001	0.0005	
Pre-training steps	6000	6000	10000	3000	10000 20000 30000	4000	
Training steps	48000	48000	180000	50000	40000 50000 60000	30000	

Table A.2: Hyperparameters values for the tested environments

A.2. Supporting material for experimental evaluation

A.2.1. Sail environment

Network	Max. cost	Tr. return	Tr. cost	Test return	Test cost
GRU	∞	-1.2281e+02	3.7708e+02	-1.0719e+02	1.4441e+02
	500	-1.5925e+02	1.2196e+03	-1.7206e+02	1.8891e+03
	2000	-1.4491e+02	5.6828e+02	-1.6541e+02	1.4000e+03
	5000	-1.3778e+02	4.4035e+02	-1.2764e+02	7.5354e+01
MLP	∞	-1.3677e+02	2.1311e+03	-1.2404e+02	6.6215e+01
	500	-1.4028e+02	2.1585e+02	-1.3796e+02	2.6958e+01
	2000	-1.3916e+02	2.2208e+02	-1.3863e+02	4.0421e+01
	5000	-1.4024e+02	2.7301e+02	-1.4333e+02	2.8340e+02
SA(12, "fixed")	∞	-1.2640e+02	4.7951e+02	-1.1149e+02	1.1155e+02
SA(16, "fixed")	∞	-1.2943e+02	5.3267e+02	-1.1766e+02	1.0981e+02
SA(1, "fixed")	∞	-1.4023e+02	7.9312e+02	-1.3168e+02	2.2177e+02
SA(2, "fixed")	∞	-1.3588e+02	1.1427e+03	-1.0283e+02	2.0099e+02
	500	-1.4497e+02	6.1271e+02	-1.4147e+02	3.1121e+02
	2000	-1.5250e+02	9.4839e+02	-1.5909e+02	1.0810e+03
	5000	-1.3986e+02	3.4527e+02	-1.3455e+02	3.2456e+02
SA(4, "fixed")	∞	-1.3117e+02	9.0629e+02	-1.1220e+02	1.6098e+02
SA(8, "fixed")	∞	-1.3069e+02	9.3723e+02	-1.1343e+02	1.4111e+02
	500	-1.8797e+02	3.0616e+03	-2.2168e+02	5.8235e+03
	2000	-1.4675e+02	6.6422e+02	-1.3612e+02	9.5386e+01
	5000	-1.4075e+02	3.3319e+02	-1.3673e+02	1.5349e+02

(a) Sequence length = 20

Network	Max. cost	Tr. return	Tr. cost	Test return	Test cost
GRU	∞	-1.1947e+02	4.9860e+02	-1.0663e+02	3.8302e+02
	500	-1.8216e+02	6.0038e+03	-1.7145e+02	1.9092e+03
	2000	-1.6723e+02	1.9929e+03	-1.5705e+02	9.1221e+02
	5000	-1.7251e+02	2.5992e+03	-1.5038e+02	9.6581e+02
MLP	∞	-1.3568e+02	6.6343e+02	-1.2889e+02	2.5469e+02
	500	-1.4015e+02	3.6891e+02	-1.4731e+02	5.3026e+02
	2000	-1.3998e+02	3.2467e+02	-1.4024e+02	2.5047e+02
	5000	-1.4466e+02	5.7133e+02	-1.4315e+02	4.7993e+02
SA(12, "fixed")	∞	-1.3756e+02	7.2933e+02	-1.2438e+02	2.4582e+02
SA(16, "fixed")	∞	-1.3465e+02	5.3031e+02	-1.2525e+02	2.1479e+02
SA(1, "fixed")	∞	-1.5283e+02	1.8283e+03	-1.4378e+02	6.6107e+02
SA(20, "fixed")	∞	-1.3348e+02	1.0120e+03	-1.2244e+02	2.1875e+02
SA(2, "fixed")	∞	-1.4727e+02	1.3280e+03	-1.3038e+02	2.5456e+02
	500	-1.6630e+02	1.8966e+03	-1.4681e+02	4.7253e+02
	2000	-1.7230e+02	2.3882e+03	-2.0826e+02	4.6083e+03
	5000	-1.5869e+02	1.2123e+03	-1.8655e+02	3.6353e+03
SA(3, "fixed")	∞	-1.4192e+02	1.6004e+03	-1.2943e+02	3.1312e+02
SA(4, "fixed")	∞	-1.4313e+02	8.3036e+02	-1.2766e+02	2.9752e+02
SA(5, "fixed")	∞	-1.4051e+02	1.4262e+03	-1.3051e+02	5.2873e+02
SA(8, "fixed")	∞	-1.3517e+02	7.7064e+02	-1.2321e+02	4.3121e+02
	500	-1.4774e+02	6.7096e+02	-1.5812e+02	1.0673e+03
	2000	-1.4836e+02	6.7662e+02	-1.4242e+02	4.3495e+02
	5000	-1.5802e+02	1.1849e+03	-1.7945e+02	2.7072e+03

(b) Sequence length = 30

Network	Max. cost	Tr. return	Tr. cost	Test return	Test cost
GRU	∞	-1.4244e+02	1.5017e+03	-1.2713e+02	1.0447e+03
	500	-1.7675e+02	3.4218e+03	-1.7636e+02	2.2832e+03
	2000	-1.6077e+02	1.6046e+03	-1.6727e+02	3.7667e+03
	5000	-1.7445e+02	3.0412e+03	-1.6695e+02	1.9035e+03
MLP	∞	-1.5320e+02	1.6825e+03	-1.4345e+02	9.0544e+02
	500	-1.5599e+02	1.3019e+03	-1.5234e+02	8.9940e+02
	2000	-1.5687e+02	1.3611e+03	-1.5628e+02	1.0786e+03
	5000	-1.5619e+02	1.2946e+03	-1.5697e+02	9.4109e+02
SA(12, "fixed")	∞	-1.5096e+02	1.6610e+03	-1.3780e+02	9.8865e+02
SA(16, "fixed")	∞	-1.5661e+02	2.0107e+03	-1.4261e+02	9.3721e+02
SA(1, "fixed")	∞	-1.8601e+02	7.7902e+03	-1.6123e+02	1.6863e+03
SA(20, "fixed")	∞	-1.5497e+02	2.3089e+03	-1.3760e+02	9.5491e+02
SA(2, "fixed")	∞	-1.6832e+02	3.5195e+03	-1.4677e+02	1.1182e+03
	500	-2.1413e+02	1.6729e+04	-2.1909e+02	1.0898e+04
	2000	-1.5951e+02	2.0304e+03	-1.5962e+02	1.1802e+03
	5000	-2.2449e+02	7.4829e+04	-2.2539e+02	7.8792e+04
SA(3, "fixed")	∞	-1.7647e+02	1.1614e+04	-1.4640e+02	9.0482e+02
SA(4, "fixed")	∞	-1.6932e+02	3.8428e+03	-1.4798e+02	9.1565e+02
SA(5, "fixed")	∞	-1.6918e+02	8.7313e+03	-1.4501e+02	8.8220e+02
SA(8, "fixed")	∞	-1.5270e+02	2.3883e+03	-1.3600e+02	9.0719e+02
	500	-1.7598e+02	3.0834e+03	-1.8794e+02	3.9074e+03
	2000	-2.1509e+02	5.1004e+04	-1.7406e+02	2.4780e+03
	5000	-2.0594e+02	1.0382e+04	-2.1479e+02	1.0296e+04

(c) Sequence length = 50

Table A.3: Training and test return on the sail environment

A.2.2. Flower environment

Network	Start steps	Pretrain policy	FF layers	Constr.	Tr. return	Tr. cost	Test return	Test cost
GRU	10000	heuristic	32	0.0	-6.8776e+04	1.8951e+04	-4.5394e+03	7.6501e+02
	20000	heuristic	32	0.0	-6.1862e+04	1.8977e+04	-6.1306e+03	7.8620e+02
	30000	heuristic	32	0.0	-5.2610e+04	1.8998e+04	-7.6627e+03	7.9186e+02
LSTM	10000	heuristic	32	0.0	-8.4557e+04	1.8988e+04	-4.3415e+03	7.4470e+02
	20000	heuristic	32	0.0	-6.8314e+04	1.8984e+04	-5.4151e+03	7.6290e+02
	30000	heuristic	32	0.0	-6.4127e+04	1.9029e+04	-7.0321e+03	7.8208e+02
MLP	10000	heuristic	32	0.0	-1.1046e+05	1.9007e+04	-6.2531e+03	7.5309e+02
			128	0.0	-1.0407e+05	1.9165e+04	-5.2370e+03	7.8302e+02
	20000	heuristic	32	0.0	-7.6614e+04	1.9005e+04	-8.0395e+03	7.5475e+02
			128	0.0	-7.8623e+04	1.9118e+04	-7.8350e+03	7.8738e+02
	30000	heuristic	32	0.0	-5.8387e+04	1.9041e+04	-6.6957e+03	7.5731e+02
			128	0.0	-5.4499e+04	1.9131e+04	-8.0406e+03	7.9485e+02
SA(1, "fixed")	10000	heuristic	32	0.0	-1.0835e+05	1.8926e+04	-4.9739e+03	5.6301e+02
	20000	heuristic	32	0.0	-4.9830e+04	1.8838e+04	-4.2309e+03	5.3147e+02
	30000	heuristic	32	0.0	-4.5505e+04	1.8921e+04	-5.8656e+03	5.6030e+02
SA(2, "fixed")	10000	heuristic	32	0.0	-4.1665e+04	1.8807e+04	-1.5788e+03	4.0508e+02
	20000	heuristic	32	0.0	-2.9583e+04	1.8750e+04	-2.6111e+03	5.0362e+02
	30000	heuristic	32	0.0	-3.2019e+03	1.8724e+04	-2.6467e+03	4.3040e+02
SA(4, "fixed")	10000	heuristic	32	0.0	-5.1363e+04	1.8670e+04	-2.4170e+03	4.5060e+02
	20000	heuristic	32	0.0	-2.1551e+04	1.8724e+04	-3.3727e+03	5.0195e+02
	30000	heuristic	32	0.0	-1.0566e+03	1.8617e+04	-3.3339e+03	4.9270e+02
SA(8, "fixed")	10000	heuristic	32	0.0	-4.7760e+04	1.8664e+04	-1.8103e+03	3.2168e+02
	20000	heuristic	32	0.0	-2.3992e+04	1.8737e+04	-2.5142e+03	4.2474e+02
	30000	heuristic	32	0.0	-6.2922e+03	1.8772e+04	-3.7177e+03	4.9618e+02

Table A.4: Agents on the 2-dimensional move environment. The training return **includes the reward** of samples collected off-policy way (by the heuristics or random exploration).

Network	Start steps	Pretrain policy	FF layers	Constr.	Tr. return	Tr. cost	Test return	Test cost
GRU	10000	heuristic	128	0.0	-2.7125e+05	7.7856e+04	-1.4919e+04	3.1863e+03
	20000	heuristic	128	0.0	-1.8550e+05	7.7595e+04	-1.5345e+04	3.1864e+03
	30000	heuristic	128	0.0	-1.5410e+05	7.7520e+04	-1.4379e+04	3.1871e+03
LSTM	10000	heuristic	128	0.0	-2.6602e+05	7.7775e+04	-1.4027e+04	3.1843e+03
	20000	heuristic	128	0.0	-1.9218e+05	7.7557e+04	-1.5237e+04	3.1844e+03
	30000	heuristic	128	0.0	-1.5311e+05	7.7420e+04	-1.5900e+04	3.1841e+03
MLP	10000	heuristic	128	0.0	-3.5421e+05	7.7846e+04	-2.0295e+04	3.1885e+03
	20000	heuristic	128	0.0	-2.2293e+05	7.7443e+04	-1.7584e+04	3.1880e+03
	30000	heuristic	128	0.0	-1.6914e+05	7.7241e+04	-1.8403e+04	3.1881e+03
SA(1, "fixed")	10000	heuristic	32	0.0	-2.0653e+05	7.7378e+04	-1.1248e+04	3.1865e+03
			128	0.0	-2.7010e+05	7.7611e+04	-1.6098e+04	3.1883e+03
	20000	heuristic	128	0.0	-1.7909e+05	7.7328e+04	-1.5505e+04	3.1876e+03
	30000	heuristic	128	0.0	-1.1628e+05	7.7150e+04	-1.4894e+04	3.1871e+03
SA(2, "fixed")	10000	heuristic	32	0.0	-1.9793e+05	7.7401e+04	-1.2301e+04	3.1844e+03
			128	0.0	-2.5585e+05	7.7552e+04	-1.5298e+04	3.1867e+03
	20000	heuristic	128	0.0	-1.6622e+05	7.7292e+04	-1.5151e+04	3.1852e+03
	30000	heuristic	128	0.0	-9.9393e+04	7.7114e+04	-1.3588e+04	3.1846e+03
SA(4, "fixed")	10000	heuristic	32	0.0	-1.9010e+05	7.7401e+04	-1.5162e+04	3.1839e+03
			128	0.0	-2.6098e+05	7.7534e+04	-1.5126e+04	3.1841e+03
	20000	heuristic	128	0.0	-1.3888e+05	7.7239e+04	-1.3097e+04	3.1822e+03
	30000	heuristic	128	0.0	-1.0366e+05	7.7092e+04	-1.3289e+04	3.1817e+03
SA(8, "fixed")	10000	heuristic	32	0.0	-1.8912e+05	7.7367e+04	-1.2299e+04	3.1778e+03
			128	0.0	-2.3046e+05	7.7471e+04	-1.3120e+04	3.1804e+03
	20000	heuristic	128	0.0	-1.4113e+05	7.7226e+04	-1.3198e+04	3.1787e+03
	30000	heuristic	128	0.0	-7.9245e+04	7.7071e+04	-1.1862e+04	3.1788e+03

Table A.5: Agents on the 8-dimensional move environment. The training return **includes the reward** of samples collected off-policy way (by the heuristics or random exploration).

Network	Start steps	Pretrain policy	FF layers	Constr.	Tr. return	Tr. cost	Test return	Test cost
GRU	10000	heuristic	32	0.0	-9.7888e+04	1.8932e+04	-3.3215e+03	7.5024e+02
	20000	heuristic	32	0.0	-1.3238e+05	1.9011e+04	-4.7636e+03	7.8021e+02
	30000	heuristic	32	0.0	-1.6172e+05	1.9093e+04	-6.7287e+03	7.8982e+02
LSTM	10000	heuristic	32	0.0	-1.1151e+05	1.8998e+04	-2.8492e+03	7.1932e+02
	20000	heuristic	32	0.0	-1.4227e+05	1.9033e+04	-3.1847e+03	7.4281e+02
	30000	heuristic	32	0.0	-1.8410e+05	1.9122e+04	-4.8895e+03	7.7341e+02
MLP	10000	heuristic	32	0.0	-1.4787e+05	1.8998e+04	-4.4253e+03	7.3920e+02
			128	0.0	-1.3072e+05	1.9260e+04	-3.9485e+03	7.7783e+02
	20000	heuristic	32	0.0	-1.5036e+05	1.9070e+04	-6.9355e+03	7.3044e+02
			128	0.0	-1.6341e+05	1.9285e+04	-7.1456e+03	7.7970e+02
	30000	heuristic	32	0.0	-1.7102e+05	1.9223e+04	-4.3134e+03	7.3204e+02
			128	0.0	-1.6135e+05	1.9425e+04	-7.2642e+03	7.9443e+02
SA(1, "fixed")	10000	heuristic	32	0.0	-1.3360e+05	1.8828e+04	-3.4142e+03	4.8361e+02
	20000	heuristic	32	0.0	-9.4466e+04	1.8666e+04	-1.4854e+03	4.0221e+02
	30000	heuristic	32	0.0	-1.4145e+05	1.8887e+04	-2.9915e+03	4.2974e+02
SA(2, "fixed")	10000	heuristic	32	0.0	-5.3336e+04	1.8712e+04	-2.6110e+02	2.6911e+02
	20000	heuristic	32	0.0	-5.8064e+04	1.8479e+04	-5.8986e+02	3.4684e+02
	30000	heuristic	32	0.0	-5.6847e+04	1.8457e+04	-1.2580e+02	2.2486e+02
SA(4, "fixed")	10000	heuristic	32	0.0	-4.0919e+04	1.8404e+04	-1.2504e+03	3.5453e+02
	20000	heuristic	32	0.0	-6.1924e+04	1.8486e+04	-8.2360e+02	3.6366e+02
	30000	heuristic	32	0.0	-3.7786e+04	1.8158e+04	-4.5052e+02	3.1321e+02
SA(8, "fixed")	10000	heuristic	32	0.0	-3.7580e+04	1.8401e+04	3.6072e+01	1.5639e+02
	20000	heuristic	32	0.0	-5.6390e+04	1.8540e+04	-1.1863e+02	2.5953e+02
	30000	heuristic	32	0.0	-7.1761e+04	1.8665e+04	-1.5314e+03	2.9347e+02

Table A.6: Agents on the 2-dimensional move environment. The training return **does not include** the reward of samples collected off-policy way (by the heuristics or random exploration).

Network	Start steps	Pretrain policy	FF layers	Constr.	Tr. return	Tr. cost	Test return	Test cost
GRU	10000	heuristic	128	0.0	-3.4367e+05	7.8188e+04	-1.3931e+04	3.1857e+03
	20000	heuristic	128	0.0	-3.4361e+05	7.8336e+04	-1.4640e+04	3.1859e+03
	30000	heuristic	128	0.0	-3.5542e+05	7.8526e+04	-1.2595e+04	3.1867e+03
LSTM	10000	heuristic	128	0.0	-3.4500e+05	7.8069e+04	-1.2275e+04	3.1831e+03
	20000	heuristic	128	0.0	-3.3115e+05	7.8132e+04	-1.3126e+04	3.1830e+03
	30000	heuristic	128	0.0	-3.3835e+05	7.8175e+04	-1.3325e+04	3.1824e+03
MLP	10000	heuristic	128	0.0	-4.7516e+05	7.8222e+04	-1.9327e+04	3.1884e+03
	20000	heuristic	128	0.0	-3.9003e+05	7.7964e+04	-1.5740e+04	3.1878e+03
	30000	heuristic	128	0.0	-3.9436e+05	7.7876e+04	-1.6706e+04	3.1882e+03
SA(1, "fixed")	10000	heuristic	32	0.0	-2.6399e+05	7.7520e+04	-9.1367e+03	3.1862e+03
			128	0.0	-3.6308e+05	7.7943e+04	-1.3766e+04	3.1882e+03
	20000	heuristic	128	0.0	-3.1647e+05	7.7813e+04	-1.2207e+04	3.1873e+03
	30000	heuristic	128	0.0	-2.6995e+05	7.7693e+04	-1.0846e+04	3.1869e+03
SA(2, "fixed")	10000	heuristic	32	0.0	-2.5292e+05	7.7598e+04	-1.0003e+04	3.1830e+03
			128	0.0	-3.3811e+05	7.7872e+04	-1.2923e+04	3.1865e+03
	20000	heuristic	128	0.0	-3.0243e+05	7.7768e+04	-1.1907e+04	3.1847e+03
	30000	heuristic	128	0.0	-2.3318e+05	7.7608e+04	-9.1135e+03	3.1842e+03
SA(4, "fixed")	10000	heuristic	32	0.0	-2.4056e+05	7.7599e+04	-1.4894e+04	3.1844e+03
			128	0.0	-3.3931e+05	7.7865e+04	-1.2948e+04	3.1831e+03
	20000	heuristic	128	0.0	-2.4386e+05	7.7641e+04	-9.1518e+03	3.1811e+03
	30000	heuristic	128	0.0	-2.3820e+05	7.7572e+04	-8.3870e+03	3.1805e+03
SA(8, "fixed")	10000	heuristic	32	0.0	-2.4557e+05	7.7563e+04	-1.0125e+04	3.1769e+03
			128	0.0	-3.0210e+05	7.7741e+04	-1.0471e+04	3.1788e+03
	20000	heuristic	128	0.0	-2.4951e+05	7.7630e+04	-9.9974e+03	3.1768e+03
	30000	heuristic	128	0.0	-1.9065e+05	7.7522e+04	-7.4970e+03	3.1783e+03

Table A.7: Agents on the 8-dimensional move environment. The training return **does not include** the reward of samples collected off-policy way (by the heuristics or random exploration).

Bibliography

Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. Constrained policy optimization. *CoRR*, abs/1705.10528, 2017. URL http://arxiv.org/abs/1705.10528.

E. Altman. *Constrained Markov Decision Processes*. Chapman and Hall, 1999.

K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, Nov 2017. ISSN 1558-0792. doi: 10.1109/MSP.2017.2743240.

J. A. Bagnell and J. G. Schneider. Autonomous helicopter control using reinforcement learning policy search methods. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, volume 2, pages 1615–1620 vol.2, May 2001. doi: 10.1109/ROBOT.2001.932842.

Richard Bellman. A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684, 1957. ISSN 0022-2518.

Dimitri P. Bertsekas. Chapter 3 - the method of multipliers for inequality constrained and nondifferentiable optimization problems. In Dimitri P. Bertsekas, editor, *Constrained Optimization and Lagrange Multiplier Methods*, pages 158 – 178. Academic Press, 1982. ISBN 978-0-12-093480-5. doi: https://doi.org/10.1016/B978-0-12-093480-5.50007-6. URL http://www.sciencedirect.com/science/article/pii/B9780120934805500076.

Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, USA, 2004. ISBN 0521833787.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016. URL http://arxiv.org/abs/1606.01540.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, and Dario Amodei. Language models are few-shot learners, 05 2020.

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014. URL http://arxiv.org/abs/1406.1078.

Yinlam Chow, Ofir Nachum, Edgar Duenez-Guzman, and Mohammad Ghavamzadeh. A lyapunov-based approach to safe reinforcement learning. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 8092–8101. Curran Associates, Inc., 2018. URL http://papers.nips.cc/paper/8032-a-lyapunov-based-approach-to-safe-reinforcement-learning.pdf.

Yinlam Chow, Ofir Nachum, Aleksandra Faust, Edgar Duenez Guzman, and Mohammad Ghavamzadeh. Lyapunov-based safe policy optimization for continuous control. 2019. URL https://arxiv.org/abs/1901.10031.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *CoRR*, abs/1901.02860, 2019. URL http://arxiv.org/abs/1901.02860.

Gal Dalal, Krishnamurthy Dvijotham, Matej Vecerík, Todd Hester, Cosmin Paduraru, and Yuval Tassa. Safe exploration in continuous action spaces. *CoRR*, abs/1801.08757, 2018. URL http://arxiv.org/abs/1801.08757.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL http://arxiv.org/abs/1810.04805.

Kuan Fang, Alexander Toshev, Li Fei-Fei, and Silvio Savarese. Scene memory transformer for embodied agents in long-horizon tasks. *CoRR*, abs/1903.03878, 2019. URL http://arxiv.org/abs/1903.03878.

Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477, 2018. URL http://arxiv.org/abs/1802.09477.

J. García and F. Fernández. A comprehensive survey on safe reinforcement learning. 16:1437–1480, 08 2015.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR. URL http://proceedings.mlr.press/v15/glorot11a.html.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Sehoon Ha, Peng Xu, Zhenyu Tan, Sergey Levine, and Jie Tan. Learning to walk in the real world with minimal human effort, 02 2020.

Tuomas Haarnoja, Haoran Tang, Pieter Abbeel, and Sergey Levine. Reinforcement learning with deep energy-based policies. *CoRR*, abs/1702.08165, 2017. URL http://arxiv.org/abs/1702.08165.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018a. URL http://arxiv.org/abs/1801.01290.

Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications. *CoRR*, abs/1812.05905, 2018b. URL http://arxiv.org/abs/1812.05905.

Matthew J. Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527, 2015. URL http://arxiv.org/abs/1507.06527.

Nicolas Heess, Jonathan J Hunt, Timothy P Lillicrap, and David Silver. Memory-based control with recurrent neural networks. *arXiv preprint arXiv:1512.04455*, 2015.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. *CoRR*, abs/1709.06560, 2017. URL http://arxiv.org/abs/1709.06560.

Irit Hochberg, Guy Feraru, Mark Kozdoba, Shie Mannor, Moshe Tennenholtz, and Elad Yom-Tov. Encouraging physical activity in patients with diabetes through automatic personalized feedback via reinforcement learning improves glycemic control. *Diabetes Care*, 39(4):e59–e60, 2016. ISSN 0149-5992. doi: 10.2337/dc15-2340. URL https://care.diabetesjournals.org/content/39/4/e59.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 448–456. JMLR.org, 2015. URL http://dl.acm.org/citation.cfm?id=3045118.3045167.

Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *CoRR*, cs.AI/9605103, 1996. URL http://arxiv.org/abs/cs.AI/9605103.

Michael Kearns, Yishay Mansour, and Andrew Y. Ng. Approximate planning in large pomdps via reusable trajectories. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, page 1001–1007, Cambridge, MA, USA, 1999. MIT Press.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, pages 2140–2146. AAAI Press, 2017. URL http://dl.acm.org/citation.cfm?id=3298483.3298548.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015. doi: 10.1038/nature14539. URL https://doi.org/10.1038/nature14539.

Timothy Lillicrap, Jonathan Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, 09 2015.

Agnes Lydia and Sagayaraj Francis. Adagrad - an optimizer for stochastic gradient descent. Volume 6:566–568, 05 2019.

Michele Mauri, Tommaso Elli, Giorgio Caviglia, Giorgio Uboldi, and Matteo Azzi. Rawgraphs: A visualisation platform to create open outputs. In *Proceedings of the 12th Biannual Conference on Italian SIGCHI Chapter*, CHItaly '17, pages 28:1–28:5, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5237-6. doi: 10.1145/3125571.3125585. URL http://doi.acm.org/10.1145/3125571.3125585.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015. doi: 10.1038/nature14236.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016. URL http://arxiv.org/abs/1602.01783.

Yuriy Nevmyvaka, Yi Feng, and Michael Kearns. Reinforcement learning for optimized trade execution. In *Proceedings of the 23rd international conference on Machine learning*, pages 673–680. ACM, 2006.

Matteo Papini, Matteo Pirotta, and Marcello Restelli. Adaptive batch size for safe policy gradients. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 3594–3603, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

Matteo Papini, Andrea Battistello, Marcello Restelli, and A. Battistello. Safely exploring policy gradient. 2018.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III–1310–III–1318. JMLR.org, 2013. URL http://dl.acm.org/citation.cfm?id=3042817.3043083.

S. Paternain, M. Calvo-Fullana, L. F. O. Chamon, and A. Ribeiro. Learning safe policies via primal-dual methods. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, pages 6491–6497, 2019.

Jan Peters and Stefan Schaal. Natural actor-critic. *Neurocomputing*, 71(7):1180 – 1190, 2008. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2007.11.026. URL http://www.sciencedirect.com/science/article/pii/S0925231208000532. Progress in Modeling, Theory, and Application of Computational Intelligenc.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

Alex Ray, Joshua Achiam, and Dario Amodei. Benchmarking Safe Exploration in Deep Reinforcement Learning. 2019.

Filipe Rocha, Vítor Costa, and Luís Reis. *From Reinforcement Learning Towards Artificial General Intelligence*, pages 401–413. 06 2020. ISBN 978-3-030-45690-0. doi: 10.1007/978-3-030-45691-7_37.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015. URL http://arxiv.org/abs/1502.05477.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. Safe, multi-agent, reinforcement learning for autonomous driving. *arXiv preprint arXiv:1610.03295*, 2016.

Xiangxiang Shen, Chuanhuan Yin, and Xinwen Hou. Self-attention for deep reinforcement learning. In *Proceedings of the 2019 4th International Conference on Mathematics and Artificial Intelligence*, ICMAI 2019, pages 71–75, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6258-0. doi: 10.1145/3325730.3325743. URL http://doi.acm.org/10.1145/3325730.3325743.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017a. URL http://arxiv.org/abs/1712.01815.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 10 2017b. doi: 10.1038/nature24270.

I Sutskever, O Vinyals, and QV Le. Sequence to sequence learning with neural networks. *Advances in NIPS*, 2014.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018. ISBN 0262039249.

Matthew E. Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *J. Mach. Learn. Res.*, 10:1633–1685, December 2009. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=1577069.1755839.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, Oct 2012. doi: 10.1109/IROS.2012.6386109.

J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, May 1997. ISSN 2334-3303. doi: 10.1109/9.580874.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity, 2020.

Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Rémi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. *CoRR*, abs/1611.01224, 2016. URL http://arxiv.org/abs/1611.01224.

Christopher Watkins and Peter Dayan. Technical note: Q-learning. *Machine Learning*, 8:279–292, 05 1992. doi: 10.1007/BF00992698.

Chiyuan Zhang, Oriol Vinyals, Rémi Munos, and Samy Bengio. A study on overfitting in deep reinforcement learning. *ArXiv*, abs/1804.06893, 2018.

Brian D. Ziebart. *Modeling Purposeful Adaptive Behavior with the Principle of Maximum Causal Entropy*. PhD thesis, USA, 2010.