Høgskolen i Østfold

Autonomous Learning of Core Skills

Off-line learning of a model-free reinforcement learning agent with sparse reward and goal state

Master's Thesis

Martin Forsberg Lie

Department of Computer Sciences
Østfold University College
Halden

January 16, 2022

Autonomous Learning of Core Skills

Off-line learning of a model-free reinforcement learning agent with sparse reward and goal state

Master's Thesis

Martin Forsberg Lie

Department of Computer Sciences
Østfold University College
Halden

January 16, 2022

Preface

I got my first degree in computer sciences more than 20 years ago, and have been working nearly three decades within the information technology industry, mostly on the business level in the domain of industrial IT. During this time there have been tremendous discoveries within machine learning, and neural networks in particular. I have often got the question on how to optimize some process, and how we can improve the outcome. Entering the field of AI opens up an ocean of possibilities and methods.

As a child, I had a box full of electrical wires that I played with, and I remember advancing to play-sets of different kinds and custom made electronics. I could wait outside the radio shop for its opening and scan all the hardware, tools and supplies on the shelves, picking a few missing components to play with back home. Sparking a blinking light bulb or controlling the speed of a motor was mesmerizing. Later, I entered the computing area with a Commodore 64[1], mostly for programming Basic. When my father bought an IBM XT[2] in 1985, a new world opened up for building applications and connecting my knowledge to actual business use.

One day I asked my father, after seeing a crime investigation drama on television, how they could identify fingerprints from a database *that fast?* Of course, it was fictitious, and doing it in reality is hard, if not close to impossible. *—Isn't this something you should solve?* The flame inside me was lit — and finding digital solutions to problems has followed me since. With the tools and methods available today, a fingerprint-identification system can easily be set up using convolutional neural networks and digitized imagery [97].

With the computing power available at everyone's fingertips, technological use and advance become a matter for society. As we train our algorithms more than explicitly programming them, there are fewer barriers to scaling business ideas or making disruptive concepts. Conversational interfaces have changed how we communicate with businesses, and self-driving cars are at the beginning of their journey towards safer transportation. Our ideas and how we form them is not limited by technology, only by our visions. More devices and businesses are connected to the internet, we have entered a world where *the evil* compete with *the good* on a common, global platform.

This thesis is the result of a long-wanted journey into the world of artificial intelligence and discusses some of the methods and tools to use for building a robot trained on existing data. The robot is not physical but lives inside the computer for making robotized decisions. The robot is not wired, and survive only on the mercy of humans.

Fukuda [22] defines basic, animal intelligence as *the ability to adapt to dynamic environments*. Our *AI robot* starts from scratch, senses the world to create its knowledge map, self-learns which decisions lead to positive results, and acts accordingly. This loop

[1]https://en.wikipedia.org/wiki/Commodore_64
[2]https://en.wikipedia.org/wiki/IBM_Personal_Computer_XT

of *fundamental functions* is shared amongst all living creatures and has been sought for replication by mankind for long.

The thesis is a journey where I invite the reader to join the history, and therefore I consistently use plurals to indicate that *we* are travelling together.

Martin Forsberg Lie
Engelsviken, January 16, 2022

Abstract

This thesis presents a concept and framework for training PPO, SAC and DDPG reinforcement learning agents on historical data logs, exemplified using an industrial spray drier use case. The work consists of four parts: discussing the theoretical foundation of reinforcement learning, describing architecture and realisation of an agent for testing the hypothesis, evaluation of the agent performance in a synthetic simulation setup for an industrial spray drier, and testing in a real-world scenario. A generative model using LSTM autoencoders is trained on historical process data and used as a dynamic simulator for training the agents by careful manipulation of the network state. The results show that it is possible to train a reinforcement learning agent to act towards a new goal set based on the dynamics found in the data set, both in uni- and multivariate action spaces. The historical data logs must exhibit a wide range of dynamics for proper training, which makes the method suitable for unstable- and stochastic processes. This opens up many applications within the domain of process control and optimization, beyond steady-state control.

Keywords: Reinforcement Learning, Actor Critic, Autoencoder, Recurrent Neural Networks, Learning for Control

Acknowledgments

At the start of my master's programme, the Covid-19 pandemic consequently shut down all physical presence at campus and forced us to find digital solutions for meeting, socializing and project work. Thanks to technology, we have found ways to continue life, and the pandemic is still raging when I hand in this thesis.

In these situations, we find comfort in family and friends, and my biggest supporters have been my wife and children during this period, who have patiently let me devote our free time to this work.

I would also like to thank Dr. Steinar Sælid who encouraged me to embark on this programme, my supervisor Dr. Roland Olsson, and Msc. Dag Skjeltorp at my employer Borregaard for discussing ideas and concepts.

Contents

List of Figures

List of Tables

List of Code

Chapter 1

Introduction

The wise are instructed by reason; ordinary minds by experience, the stupid by necessity, and brutes by instinct[1]

We experience the world in linear time, from birth till death. Life unfolds between those milestones, consisting of millions of small decisions, life experiences and episodes of crisis and happiness, mixed with a myriad of ingredients and condiments. We develop knowledge and moral from our cultural proximity, by guidance, experimentation and genes. We become who we are as a combination of society, environment and preconditions.

Unlike computer algorithms, our decisions are executed sequentially and shape whom we become. We cannot always undo our actions but must live with the consequences. A computer program can restart, and retry the same scenario over and over again to learn what matters. This is its strength, but also its culprit. A computer program cannot restart if it transforms sensory information into insecure actions: a self-driving car must not crash just to build experience. Experimental transactions from a stock-market robot will quickly demonstrate its failure when faced with real life. A reward is given to the computer for not crashing or doing the best stock investments. But is a reward-scheme comparable to our evaluation of success? The current state of research within artificial intelligence is to allow the computer to fail in simulated environments based on human-stated priors. Its experience is transferred to the real world when we have enough confidence in its success.

As demonstrated by OpenAI, their GPT-3 algorithm is capable of conversing with humans for extended peridods.[2] The question remains unanswered: is advanced reasoning a kind of pattern recognition in our brains? If so, can synthetic algorithms, like those used for AI, replace human decision-making? Can AI become human, with feelings, passion and moral? Nobel price winner Kazuo Ishiguro describes a world where our perception of the human being is challenged when we consider that we are no longer unique individuals, but the sum of artificial intelligence algorithms.[3]

In this report, a small, staggered stepping-stone is laid on the path to general artificial intelligence. The hope is not to replace the perception of humanity as such, but to establish a framework for assisting decision-making. Can we learn the computer to understand the elements of success, and how will its moral story unfold? Within deep reinforcement learning research, the actor-critic framework is popular for learning from rewards in unknown

[1]Marcus Tullius Cicero, Roman statesman 106 BC-43 BC

[2]https://www.aftenposten.no/kultur/i/563rpK/intervju-med-en-kunstig-intelligens

[3]https://www.nrk.no/kultur/nobelprisvinner-ishiguro_-er-mennesket-mer-enn-summen-av-algoritmer_-1.15394666

systems. The *actor* proposes actions, while the *critic* evaluates its future expected reward. The role of the *critic* is to behave as the grown adult in the room, while the *actor* must learn from its childly experience.

1.1 Background and motivation

In a reinforcement learning problem, the idea of discovering long-term discounted rewards through exploring combinations of action-space rewards leads to a trail of actions with the least penalty or highest reward in the terminal state, as stated by Watkins and Dayan [18]. Watkins [16] introduced the concept of learning from delayed rewards through the *Q-learning-algoritm*. The science of *Q-learning* has been known for some time through the method of temporal differences [15]. Here, prediction of a future state assign rewards for *temporally successive predictions* and establish an *experience buffer* for prediction of future behaviour.

Building the Q-knowledge is done by iteratively exploring all possible actions and rewards, and thereby establishing a set of policies for each state. An *agent* decides on transitions between the states done by observation of responses from an observation space, the *environment* as shown in Figure 1.1. The typical method is to utilize a process simulator where the agent can invoke actions in a safe environment with a performance outmatching a real-life learning process. Synthetic sensory feeds the *Deep Q-network* with virtual responses corresponding to the result of an action transition. The system learns by discounted rewards the best policies for any given state. The trained network is then put in a real-world context using *transfer learning* and executed on real-world systems, which also continues the learning process with new policy updates.

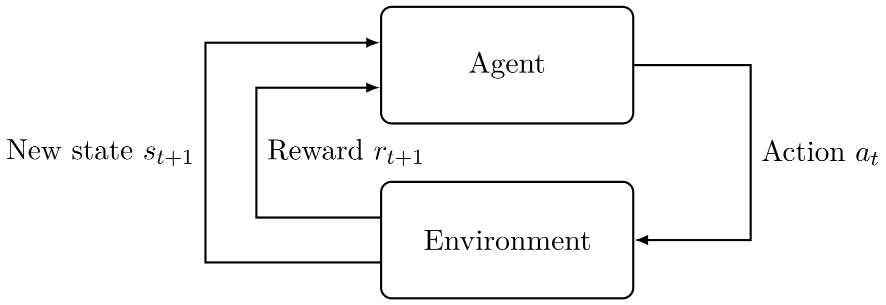


Figure 1.1: Agent and environment interaction

Deep reinforcement learning introduces deep neural networks as function approximators as they efficiently can be trained to interpolate within a trained state-space. The function approximator is computationally efficient and offers many options in its architectural alternatives. There are observered challenges in using reinforcement learning as pointed out by Dulac-Arnold et al. [92]:

1. Training off-line from the fixed logs of an external behaviour policy is complex.

2. Learning on the real system from limited samples is hard when using deep neural networks, which are sample-intensive.

3. High-dimensional continuous state and action spaces involves broad data coverage for training.

Many samples and runs, called *episodes* in RL, are necessary to train the agent, as well as exposing the agent to all facets of the observational space. An interesting aspect is to investigate methods for obscuring data in such a way that enough training samples can be provided from a limited data log set. It is critical for the learning outcome that the dataset is diversified to avoid overfitting the network. Further, the reward-function necessary in an RL context might not always be sufficiently engineered. Can the reward-function be trained? What are the implications of trained reward-functions: will they behave differently given the state of the system?

1.2 Applications

The motivation for entering this field of science is the broad application impact where sequential decision workflows are involved. Implementing this project is relevant for many application areas within many business processes. We can broadly categorize businesses goals as iterative processes on several scenes:

1. Minimize energy consumption
2. Maximize yield
3. Minimize waste
4. Control quality
5. Improve realibility
6. Improve effectiviness
7. Sustainability
8. Reduce emissions
9. Optimize supply chains

Can reinforcement learning be applied as a general optimization concept and trained on historical logs of data? A clear goal is optimized control and higher product yield and quality, with less energy usage and waste. To investigate this hypothesis, we select a delicate industrial process for analysis, further detailed in the Use Case chapter (Chapter 3).

Establishing closed-loop control of either industrial processes or even algorithmic stock traders involves deep knowledge within the respective field. A traditional control model of an industrial process may be solved by ordinary differential equations and Kalman filters, given the correct model. The necessary mathematics may appear intimidating without proper knowledge. Pre-training an RL agent on data and automatically specifying the reward function would make algorithmic control more accessible and increase the efficiency of implementation within the industry. As closed-loop control gains both acceptance and momentum in many business fields, the need for efficient methods for implementation is imperative for being able to deliver new ideas and applications, and improve existing ones.

There is a clear distinction between optimizing existing processes and establishing completely new ones. In the latter case, traditional engineering might be the solution. For existing processes, or processes where comparable data is available, we foresee that optimization and control can be completely done by learning from historical data logs.

1.2.1 Research questions

As earlier mentioned, an RL agent will be trained by a historical dataset, and the reward function estimated based on the data behaviour. The research involves the implementation of a generic agent with several capabilities. The aim is to be able to apply the agent to different industrial processes with minimal engineering effort. For this to succeed, the performance of such an agent should match or surpass existing control strategies. There are economic aspects to why such an implementation is tempting.

The research questions will surface both in the analysis of the problem, the choice of methods, the design and implementation of the project, and, most importantly, it will be revisited in the discussion and conclusion parts of the report. In light of the problem statement, the research will be divided into the following research questions:

RQ 1 Can an RL Agent be efficiently trained on historical data logs?

> **RQ 1.1** Which choice of RL Agent realizations would be a generic solver for the problem?

Here, a process model simulation will be executed by using data from a well-known industrial process, and the resulting data will be recorded. Multiple runs of this dataset will be used for separate training of an RL agent. The trained agent will then replace the process control regulation and the performance of the agent evaluated towards the state-of-the-art regulated process.

Given a selected control strategy, can a viable control simulation that exposes the state-space, be modelled and used for generating data logs for RL training? In a real-world situation, an existing process data log will substitute the simulation, but the agent performance should be evaluated on the same process as the data was generated. It seems not economically viable to expose the agent to a real process within the first exploratory part of the training. After evaluation, confidence can be established for stepping into physical processes.

RQ 2 Can a reward function be approximated and learned without supervision?

> **RQ 2.1** What are the implications of trained reward-functions: will they behave differently given the state of the system?

We investigate the problem of using neural networks as a reward function approximator and training this on data logs. Further, an analysis of the most important features by using techniques from Explainable AI will give valuable insights to evaluate if the trained reward function matches our a priori system knowledge. The term *eXplainable AI* has become a popular topic lately, and Arrieta et al. [104] describes opportunities and challenges regarding trustworthiness in the application of AI models, also called *black-box*-models, where the internals are difficult to test and visualize. Two paths are explained, the *ante-hoc* route where model internals and factors affecting them are attempted described, as well as *post-hoc* explanations, where the model results are interpreted. Lundberg and Lee [72] reviews several frameworks for the latter and proposes the *SHAP-framework* for assigning importance values for all model features for a particular prediction result. *eXplainable AI* will become important for our RL agent to validate its knowledge causality. In evaluating RL frameworks, the reward function would be the main training input for the critic, representing the *moral* of the agent.

1.2.2 Method

To successfully build a deep recurrent network, we must expose our system to as many states and policies as possible. This can be done by simulating the target system for accelerated learning, or expose the system to the real environment as described above. In the latter case, practical implications will most likely render this unacceptable in a running industrial process, due to resource constraints, health- and security implications etc., and to the fact that the learning process is a tedious task where the system needs many iterations of trial-and-error to establish a sensible decision network.

Petsagkourakis, Sandoval and Bradford et al. [110] proposes a multi-step algorithm where a process is controlled using reinforcement learning. They introduce *transfer learning* as a central concept, where the policy gradients are updated based on a step-wise-approach:

1. Step 1, Preliminary off-line learning using a stoichiometric model
2. Step 2-3, Transfer learning by updating policies based on real data
3. Step 4, On-line transfer learning by updating rewards based on real data
4. Step 5, Terminatory policy update on a real system

Several case studies were done and they conclude that it should be possible to train a reinforcement network where the true system dynamics are unknown. They build their work on the assumption that a bioprocess model can be established using hybrid approaches with both data-driven methods and physical models [103], requiring access to historical process data. Rio-Chanona et al. reviewed several methods of the performance for physical and data-driven models for bioprocess simulation [91]. Their experiments show that models in general only can model mechanisms it has been designed or exposed for. Extrapolating to unknown state-spaces cannot be verified. An approach to counter-balance this would be to be sure to expose the learning process to extremely varying process conditions, also those scenarios resulting in faulty or bad quality products. Training in a simulated environment covers this requirement, and one can presumably to some degree perturbate the input data to augment uncovered system states.

The rules of physics must be learned by the agent, but prior knowledge of physical causality could increase the training efficiency and extrapolation properties. Karpatne et al. [70] investigates the possibility to add such priors through the use of adapted loss functions with physical knowledge. They show that better generalization is achieved with physics-based loss functions. In applications where the agent has no physical reality, other constraints may be necessary. This might for example be cases of a stock market trading agent, where the consistency rules might be juridic- and trading-specific laws and regulations.

1.2.3 Hypothesis formulation and testing

An exploratory quantitative evaluation strategy will be chosen to test the success of the different model realizations. Evaluating reinforcement network outcome can be done on the *reward function* result after applying each action in the environment. The difference between real-world observation and our policy result, is the model error, as we infer that the existing process is under control at this stage. Over the course of training, the agent will attempt to minimize this error and increase accuracy towards the existing control. The models should gradually decide on the best long-term strategy to increase the accuracy towards the reward function. Regression accuracy of the models will be measured using standard machine learning metrics, where *Root Mean Squared Error (RMSE)*, *Mean Absolute Error (MAE)* and *Pearson correlation coefficient* (R^2) are widely used. The RMSE penalizes larger model errors as shown in Equation 1.1 due to the squared prediction error. Here, the squared difference between the real value y and our estimation $\hat{y}$ is calculated on i value pairs.

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{1.1}$$

We will also experiment using *Huber loss estimation* as an alternative to *MAE* and *RMSE*.

1.2.4 Experiment modelling

The idea involves training an agent on offline data, but at some point, we reach the need to integrate the algorithm into an online solution. A proper real-time execution environment must be in place, and access to historical and real-time data must be available. The execution model must also be architected, and the proper hyperparameter selection strategy chosen. Step 1-3 will be executed off-process until the agent converges towards plausible accuracy.

Step 4 will be executed by *piggy-backing* a selected process and run the reinforcement in real-time but with no closed-loop control. Step 5 involves proper risk management and mitigation, and need careful termination strategies implemented in the target organization. The risk will be in the categories of health, environment and safety, as well as resource utilization. This report will not implement real time control experiments due to the impact of these considerations.

1.2.5 Architecture

The RL agent will be solved using *Matlab* as computation environment. An architecture for the exploration, evaluation and deployment will be a part of the report.

1.3 Contribution

The main contribution of this thesis are:

- It is demonstrated that a shallow recurrent LSTM network can be trained on historical logs of time-series data in order to reproduce the dynamics of a system

- Given such a trained generative model, it is demonstrated that a reinforcement learning agent can be sufficiently trained in a multivariate scenario and propose actions towards a new goal set
- It is also demonstrated that the reward function of an agent can be trained by training an autoencoder network with a specified class of time-series data

1.4 Report Outline

This section drafts an outline of the report and a foundation for the work progress.

Part 1 will discuss the theoretical foundation of reinforcement learning, and Related work and different actor implementations.

Part 2 will describe an architecture and implementation of an agent for testing the hypothesis.

Part 3 will be an Evaluation of the agent to see if the performance is viable.

Part 4 will focus on testing in real-world scenarios and use the agent to solve real-world tasks in an experimental setting.

Part 5 will be Discussion and elaboration on Research Questions as well as aspects of further work.

The thesis will end with a Conclusion.

Chapter 2

Literature

The image of the world around us, which we carry in our head, is just a model. Nobody in his head imagines all the world, government or country. He has only selected concepts, and relationships between them, and uses those to represent the real system.[1]

2.1 Foundations

In chapter 7 of Alan Turings 1950 article [5], the *learning machine* is discussed referencing a machine that can mimic the human brain. Turing presents the *neutron* that holds *remote ideas*, and given enough disturbance exhibits critical phenomena. Turing's ideas are descriptive of today's deep learning neural networks, abled by developments in silicon performance, programming languages and knowledge sharing in science.

Neural networks have shown their ability to learn from examples, through *supervised learning.* A proficient property is the ability to discover the data representation itself and mask features, not only the mapping to the output. This is what we call *representation learning* and can often result in better model performance [63]. Less pre-processing of the input data is thus necessary. Deep learning solves a central challenge in representation learning where the machine builds different abstraction layers where each neuron layer builds upon knowledge from previous layers, as we go deeper into the network topology. By combining our understanding of physical processes in the brain and construct networks of neurons that are interconnected, we are able to solve many modelling problems through architecting the neural networks accordingly for a broad set of applications [71].

Frank Rosenblatt introduced the theory of *the perceptron* [6] and that learned associations in a *differentiated environment* approaches a better-than-chance classification as the number of learning stimuli increases. A perceptron is an *activation function* with associated weighted inputs, where incoming stimuli fire the perceptron or leave it idle.

Figure 2.1 shows a perceptron with three inputs x_1, x_2, x_3 and one output. Each input has associated weights w_1, w_2, w_3 and is commonly referred to as synapses, the synaptic weight vector $\mathbf{w}$. The activation function in Equation 2.1 outputs a binary representation

[1]Jay Wright Forrester, 1971

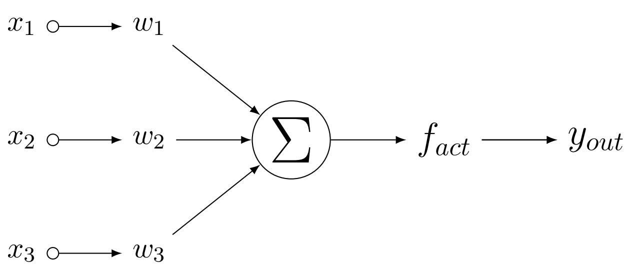


Figure 2.1: A simple neural network architecture

of the weighted sum so that the output is fed to the next layer of perceptrons and their input.

$$\text{output} = \begin{cases} 0 \text{ if} \sum_j w_j x_j \leq \text{threshold} \\ 1 \text{ if} \sum_j w_j x_j > \text{threshold} \end{cases} \tag{2.1}$$

The weights are trained by observing the output with a true sample. The error is iteratively calculated using a backpropagation algorithm. It is now common that the activation of a neuron is a scalar number in the range of $-1, 1$ and that a neuron has a *bias* added to the activation function [63, p. 187]. In order to train our network, we can employ a *stochastic gradient descent* strategy that estimates weight updates during iterations. In backpropagation, we start at the output and step backwards in the network iteratively adjusting the weights and bias with the weighted error derivative, as proposed by [49] showed in Equation 2.2 where C represents a cost-function, the error between the network state and the true state, L is the layer and a is the activation.

$$f(\Delta w) = \Delta C \cdot a^{(L-1)} \cdot \phi'(a^{(L)}) \tag{2.2}$$

The design of the cost function C, activation function ϕ and strategy of gradient descent has been a long-standing focus for the research community, and the selection of the method is tightly dependent on the application.

2.1.1 Markov processes

A Markov decision process is described by a state space S, a possible action-space A, a transition space T and reward function R [16].

$$\mathcal{P} = \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \tag{2.3}$$

The next state s_{t+1} is dependent on an applied action a in state s_t, where an *agent* selects the best action according to some policy. There is a probabilistic relationship between the chosen action and the new state, a relation which is the core of the reinforcement learning problem: a Markovian decision is a process in which the path to the end result is always known based on the immediate observed state. In a non-propelled bullet case, you will be able to calculate its terminal ballistic trajectory by knowing the state at any point (speed, mass and direction). The calculation function can be based on first principle physical models or data-driven by reinforcement learning and neural networks as function approximators. A *Markovian process chain* is one where the history of previous events determine the action and represents a class of algorithms that utilizes a discounted reward system, based on the system horizon being episodic or continuous. The *Markovian property* of the problem space is an important prerequisite for reinforcement learning: the future state is only dependent on the current state.

2.1.2 Temporal differences

A history of machine learning theory is nicely summoned by [15] by describing the problem of learning to predict by *temporal differences.* This field has its roots in the late 1950s starting with Samuel's checker-playing program [7], followed up by various articles in the 1980s. In traditional machine learning, a problem is optimized towards a minimal error residual between a system's true state and the predicted state. This can be described as *supervised learning* where a true state is presented to learn the system probabilities given a set of known inputs. The idea of *TD learning* is to learn from a set of successive predictions in a sequence, as a multi-step prediction problem. Correctness is increased as evidence of the true system is partially revealed. The reasoning behind this is founded on how humans perceive problems and relate to the world, where we often make decisions based on partially available information in a stream of related patterns. The decisions reveal more information, and we learn from these temporal differences. We can model such a multi-step sequence as

$$x_1, x_2, x_3, ..., x_m, z \tag{2.4}$$

where each x_t is an observation vector at time t leading to outcome z. For many such sequences, a learner produces several predictions, each P_t corresponding to the outcome z:

$$P_1, P_2, P_3, ..., P_m \tag{2.5}$$

Each prediction has corresponding weights w with a functional dependence towards x_t, we can then explicitly write the probability function as $P(x_t, w)$. Our problem is now narrowed towards learning w with the *delta rule*, with *backpropagation* or with the *Widrof-Hoff-rule.* Sutton [15] presents the incremental *TD*-procedure, that can be calculated at each sequence time step with the sum of all past values. The generalized $TD(\lambda)$-function introduces a *learning rate*, in which earlier prediction sequences can be weighted differently as opposed to the more recent observations. We see from the function

$$\Delta w_t = \alpha(P_{t+1} - P_t) \sum_{k=1}^{t} \lambda^{t-k} \Delta_w P_k \tag{2.6}$$

that the probability weight is updated by the *temporal difference* of predictions, learning rate α and the sum of earlier weight updates, adjusted by the dynamic learning rate λ.

The whole idea of learning by temporal differences is that it allows us to predict the next state given a sequence of preceding states. In this way, we are not only using the current state to predict the next, which could lead to a less favourable next state but instead use our earlier observations and experiences to select states that might be intermediate towards our terminal state. The $TD(\lambda)$-procedure tries to use the information from the sequence predictions, while traditional machine learning methods simply ignore them. In such a system, there will only be one prediction given the current state, if the history is ignored.

It is tempting to draw a relation towards Bayes' theorem that states that the conditional probability of an event is based on knowledge of other related events. Mathematically, this is described as

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.7}$$

where the conditional probability of A given event B is described using already known relations of events A and B. In a frequentist interpretation, $P(A)$ and $P(B)$ is the *likelihood* of the event given available data. In a dice game, we might initially say that there is a probability of $\frac{1}{6}$ for each side of the dice, which is our initial estimate of the maximum likelihood of the event. Learning the agent becomes a task of presenting available data, and update our belief system's probability distribution given new events.

The TD-learning process follows the principles of statistical gradient descent, pushing the predictions in the direction of the steepest descent. Further, we may be interested in the cumulative score when predicting the outcome of a gameplay, and the process can be used intermediary, where system re-evaluation can be done at each time step. The predicted score, cost or reward is cumulative, and thus we can calculate the remaining cumulative cost anywhere in the observation sequence given an episodic model. The cumulative cost prediction is the *value* of the observation sequence, hinting towards a separate *critic* part of an actor. The strength of the TD-learning algorithm is its computing efficiency and memory utilization, not needing access to previous observation and prediction sequences. Sutton [15] links to related research, the most prominent is backpropagation in connectionist networks [12] which decides which part of a network to change. An error derivative combines the descent and weight update.

Further, we can map the episodic learning to a continuous learning case by discounting the predictions:

$$z_t = \sum_{k=0}^{\infty} \gamma c_{t+k+1} \tag{2.8}$$

where γ is a discount parameter. This way, we can predict with a parameterizable horizon in the infinity time scale where we can balance the need for long-term or shorter-term predictions.

Williams [19] argues that the search behaviour of a learning function should include some randomness in the input-output state to accommodate *exploratory* behaviour. The prediction function can be implemented in a neural network where each unit is considered a learning agent that predicts the reinforcement signal r. In an immediate-reinforcement learning problem, r is learnt at each time step, as opposed delayed-reinforcement where r is learnt at the end of a sequence. The network is trained for each time step, a situation applicable to real-time scenarios, however, we will later see that immediate training can unnecessarily skew the training towards the last events. Williams [19] describes a simple actor-critic algorithm where the actor at each time step predicts the output given the input, and the reinforcement value r of the input/output pair is predicted by the critic. Such a prediction can be a function of time or given by the sequence history.

The network weights are updated using r towards a baseline difference b:

$$\Delta w = \alpha(r - b)e \tag{2.9}$$

with α being the learning rate and e a *characteristic eligibility* of w. Each weight w is locally updated to reflect the globally predicted reinforcement signal r. Selecting a proper baseline is done towards minimizing the variance of the weight changes over time. This gradient-based approach integrates well with backpropagation and serves as a starting point for other reinforcement learning algorithms.

2.1.3 Delayed reinforcement learning

Tesauro [17] describes the delayed reinforcement learning paradigm where the learner passively observes a temporal sequence of input states that leads to a reinforcement signal r. This method is used to learn a system from scratch with no prior knowledge of the optimal control strategy. By observing the temporal differences with the $TD(\lambda)$-algorithm, actions that lead to the highest terminal reinforcement signal is learned. By terminal, we mean that there is no supervised signal telling the success of each transient state during a sequence. The temporal observed actions might therefore be suboptimal, depending on the actual *player performance* in the system. Tesauro [17] describes practical considerations of implementing the $TD(\lambda)$-algorithm to complex real-world problems, bringing the field of reinforcement learning forward by applying TD-learning to the game of backgammon.

The problem is very well suited for reinforcement learning due to several appealing features. Backgammon is non-deterministic, as opposed to chess and checkers, due to a stochastic dice roll that controls the game. Several strategies can be employed for blocking opponents, hitting them and resetting their pieces, but depending on game state, different tactics are usually employed. At the start of the game, blocking configurations are engaged, as opposed to the end of the game were racing the pieces to the goal is more important. Further, branching ratios are high, meaning that there is no single *best way* or action state. Tesauro [17] employs both deep- and shallow fully connected neural networks, and demonstrate that the system has better performance than competing solutions at the time of writing the article.

The network is trained with a sequence of board positions $x_1, x_2, x_3, ..., x_m$ and a final reward signal z. In a control scenario, the network is presented with new stochastic dice rolls, enforcing an exploration strategy of the state space. This way, new strategies and improved evaluations can be discovered [20].

However, the $TD(\lambda)$-algorithm does not address prediction-control tasks and Tesauro [17] proposes to train a separate controller network for this purpose. Convergence of the prediction and control networks towards a global optimum can be an issue in non-linear problems like this. *Volatility* in stochastic networks may also influence weight updates, as the noise should not exceed the baseline variance. Employing a learning rate schedule and tuning the λ-parameter is important, as well as considerations for avoiding overfitting by tuning the number of nodes and layers in the network, what we can call the *network fidelity.*

Input volatility and network fidelity must be balanced in complex problems as the network might only see x_t and the succeeding state once. If x_{t+1} and x_t is unrelated due to high volatility, the network might not be a good predictor. One way to overcome this is to iterate on state x_t multiple times and assure that all states are visited during training. There is a direct relation between these considerations and system performance. Neural networks are known to interpolate well but fails at extrapolating. In these cases, physical modelling rules can be of importance, depending on the problem being solved.

2.1.4 Q-learning

A true Markovian approach may in some cases be attractive, depending on the application. In systems with finite state spaces, where there is no dependency on earlier observations or actions, a learning process can be based on delayed rewards only. The *Q-learning* is such a method described by Watkins [16]. In a *Q-table*, the future expected reward for each possible state-action pair is trained by exploring all possible states and noting the final reward in the terminal state. This trial-and-error approach is called *exploring* the environment and is typically the strategy we use during training the *Q-table* and learning the environment. Training each state-action-pair can be done with a simple *one-step-Q-learning*, where the $Q(s, a)$ is trained by a simple iterative update of Q:

$$Q(x, a) = (1 - \alpha)Q(x, a) + \alpha r \tag{2.10}$$

where α is a learning rate parameter. At some point, we will select the action with the highest future expected reward in each state. This is called *exploitation* of the environment where the agent suggests experimental actions. It is easy to see that balancing exploration and exploitation is a matter of performance and security. We would optimally find a strategy that balances these issues so that our *Q-table* is kept valid. Watkins [16] note that exploration of the environment should be done when it adds value, is cheap and the time used for exploration is short compared to the time we will use the behaviour.

2.2 Rediscovery

Playing games has a long history of being a demonstrator for reinforcement learning. Gameplay consists of a *goal*, *a player* and *the game with rules*, and the interaction between the player and the game to reach a specific target. Since computer games are excellent non-destructive opponents (the *real world state* is never manipulated and stays intact), training reinforcement agents within such an environment is tractable. A property of computer programs is the ability to duplicate and parallelize, and speeding up beyond real-time, thereby accelerating training. Further, games have often *rounds* where the player takes action based on evaluation of a state. The action manipulates the state, initiates environment feedback, and then the player can take his next draw based on the new state. The gameplay metaphor fits perfectly for the evaluation of reinforcement learning.

2.2.1 Deep Q-learning

Mnih et al. [51] of Deepmind[2] demonstrates how a deep convolutional neural network can learn control policies from complex environments. Advances in deep learning and the introduction of convolutional layers in neural networks have made it possible to represent high fidelity sensory data efficiently. Mnih et al. [51] demonstrates the use of raw pixel inputs from an Atari game emulator, as illustrated in Figure 2.2, to learn a deep neural network with very little pre-processing. In fact, no specific feature engineering was done except reducing the dimension space by resizing and cropping image input and colour channels from the raw pixels outputted from the game emulator, in order to reduce computation time and network volatility. The same neural network architecture was used to train an agent to play multiple games, some outperforming human players.

[2]https://deepmind.com/

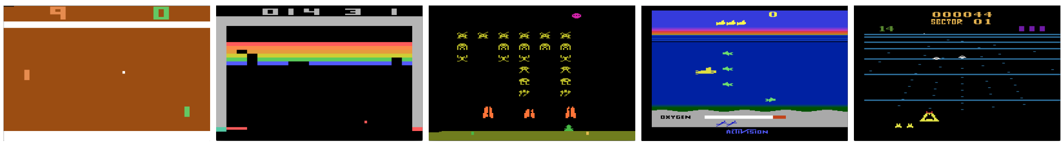


Figure 2.2: Screenshots from five Atari 2600 Games
Figure from [51]

There are nearly 20 years of research history between the initial backgammon *TD-learning* and the Atari *Deep Q Network* by Deepmind. However, as the authors point out, the relative advances have not been as expected. The backgammon gameplay seemed to be the most promising attempt at reinforcement learning, probably due to its stochastic nature and dice-roll, enabling efficient environment exploration. Attempts with chess and checkers have not seen similar successive developments.

Deepmind was able to demonstrate that the principles of the *Q-learning* can be used to calculate a *Q-network*, a neural network used as a function approximator instead of the *Q-table*. The neural network was built with one input layer, two convolutional layers and a fully connected layer. The output layer was organized with one node for each possible action in the game under test, saving computation of the *Q-value* to one network pass only. This would not be suitable for continuous action spaces, although the use case presented a finite action space for each game under test. This was the only difference between the game plays. The input layer was the pre-processed raw pixels from the emulator without any additional feature engineering.

However, learning from a sequence of consecutive samples would skew the network in the direction of the most recent training. Deepmind introduced *experience replay* in the learning process which should break those correlations and reduce variance. A fixed history of state-action-values is kept in a cyclic memory, and a random selection from this memory is applied whenever the network is trained on new samples. New samples would also be inserted into the memory and used for training later.

The reward function followed the game score but was clipped to the set $[-1, 0, 1]$, indicating negative, neutral or positive game score development. This was done so that the same learning rate parameter could be used across all gameplay training, but it could affect the system's ability to counter the different magnitude of score changes. Last, a frame-skipping strategy was chosen to further limit the data presented to the network. Depending on the game, only every 3rd or 4th frame was used as input. We will later observe that this is a common technique for training, Dota 2 by OpenAI [98] uses the frame-skipping technique so that the system can perform its evaluation and policy-calculation in real-time, enabling game plays with human opponents.

The performance of the *DQN-network* is impressive, and an update on the matter was presented in *Nature* in 2015 with results of several other games trained under the same conceptual framework. Out of 49 tested games, the methods show better-than-human performance in 29 of the games [60]. Further, the authors demonstrated that training a separate target network affects the results.

2.2.2 Value networks

Silver et al. [67] continued to work on mastering the game of *Go*, a task long time seen as an ultimate goal for AI gameplay. They introduced *value networks* that evaluates the current game state that outputs the probability of winning based on the board positions in their project *AlphaGo*. Further, a *policy network* outputs action probability distributions based on the state. Combined with a tree search algorithm, a look-ahead search algorithm is used to evaluate the actions with the highest state value, where the nodes store the action value *Q(s,a)*, the visit count *N(s,a)* and prior probability *P(s,a)*. On each game iteration, a game simulation occurs where an action is selected as to maximize the action value and a bonus u being the proportion between P and N:

$$a_t = argmax(Q(s_t, a) + u(s_t, a)) \tag{2.11}$$

On each pass, the game is simulated to its terminal state using the tree search, starting from the root node. After each simulation iteration i, the search tree is updated with *Q, N* and *P*. $u(s_t, a)$ decays after many visits to a node, encouraging exploration of the environment. When reaching a leaf node, it can be expanded by adding the probabilities from a different policy network trained using supervised learning on human gameplay. The game is simulated using a fast rollout policy network combined with a value network, and the tree is updated on each simulation iteration with the updated values reaching the terminal state with the desired outcome. This combination is controlled with a mixing parameter λ, indicating which method has the most impact, value networks or rollout policy. The most visited action from the root indicates the move with the highest probability of winning and is chosen for updating the environment in that round.

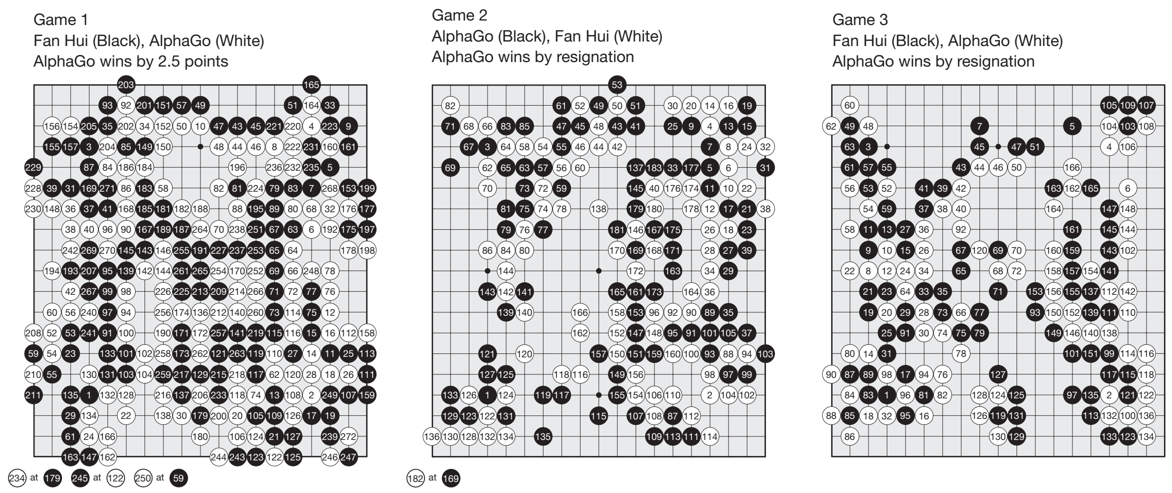

Game 1
Fan Hui (Black), AlphaGo (White)
AlphaGo wins by 2.5 points
234 at 179 245 at 122 250 at 59
Game 2
AlphaGo (Black), Fan Hui (White)
AlphaGo wins by resignation
182 at 169
Game 3
Fan Hui (Black), AlphaGo (White)
AlphaGo wins by resignation

Figure 2.3: AlphaGo game play
Figure from [67]

This *Monte Carlo tree search* method combined with value- and policy networks were able to beat a European Go champion in five games and achieved a winning rate of 99.8% against other Go programs. The authors also evaluated different mixing parameters λ, and they found a balancing 0.5 had the best performance, however, using value networks only ($\lambda = 0$) with no rollout policy network, AlphaGo exceeded the performance of all other Go algorithms. This indicates that a value network implementation is a viable method to evaluate states.

The authors introduced the *AlphaGoZero* in 2017 [74], where the algorithm did not use pre-trained networks of human gameplay but learned the strategy itself without any guidance. Only the game rules were known, and the system is trained by random self-play only, using the historic- and current board positions as the input state. There is no separate policy- and value networks, only one single neural network that outputs the action probabilities and the state value. The gameplay uses this single network for look-ahead search and updates the network based on the error between the actual winner and the predicted winner. The implementation is efficient and defeats other implementations of Go. In January 2017 it defeated human professional players 60-0 in online games [74].

Central to this pure reinforcement learning strategy is how to improve the policy network. A traditional approach is to alternate between policy evaluation and policy improvement as to estimate the value function based on outcomes from different gameplays [112]. A greedy approach is to only select actions as to maximize the value function using the error residual. This approach is useful in environments with large state spaces as demonstrated in *AlphaGoZero*.

2.2.3 Memory networks

A group of scientists from OpenAI demonstrated the use of deep reinforcement learning with recurrent networks on a large-scale gaming platform, the Dota 2 [98]. This game presents challenges of long time horizons, complex environments and occlusion of state information by only being able to partially observe the environment. The complexity is increased by high dimension state- and observation spaces, where the reinforcement agent controlled a team of several players, forming groups when the gameplay rewarded such strategy. In early phases of the game, solo achievements were more valuable than cooperative manoeuvres, ending in pure group collaboration in the end phases of the game. The agent was trained with a single-layer LSTM recurrent memory network with 4096 nodes for several months of self gameplay, for a total of 159 million network parameters. The network outputs a fully connected layer with action space probabilities and state value.

During training, the team updated parts of their algorithms, and to preserve the training weights they invented a method called *surgery*, in which parts of the affected network is replaced by the updated algorithms to avoid restarting the training. This method performed well and was evaluated by complete retraining on the final algorithm by evaluating the differences with the networks that were iteratively undergoing surgery. The system utilizes one network instance per game *hero*, the character controlled by the agent, and use frame-clipping to reduce the observational space and hence computation time. The policy was trained based on a variation of the *advantage actor critic*, using *Proximal Policy Optimization* algorithm. *Experience buffers* was used for training, but also for distributing training across computers and GPUs.

However, the reward function was hand-made by humans, forcing the agent to learn strategies that humans believe are *best choices*, like killing enemies and gaining resources. OpenAI also engineered the reward function such that it forced the agent to change its play strategy during playtime, to accommodate for group formation towards the end of the game through a dynamic *team spirit* that was dependent on game time. Different

Figure 2.4: Game play from Dota 2
Figure from [98]

rewards were given based on the human setup, like rewarding different *towers concurred* and spending *gold* on resources. The rewards differed whether the resources were bought individually or as a team. In sum, the reward function heavily determines game progress, and it could be argued that the agent shows less self-intelligence when the reward function is heavily engineered by humans. At least, we can say that the agent exhibits tactical intelligence, but not strategic intelligence. The authors also reflect that investigating other reward schemes is interesting future work.

One of the interesting approaches to reinforcement learning is to estimate the reward function, which is one of the research questions in this thesis.

2.3 Physical control

Bringing the success of gameplay to the physical world enables us to broaden the menu of available control strategies when considering a control problem. If we could let the computer learn how the physical world works, it could potentially drive wider adoption of data-driven methods within control engineering. There are situations where other methods are better suited, i.e. in fields where less sensory data is present, or where you cannot express the system state adequately. Here lies the culprit of data-driven methods: to use them, we need data. In the case of reinforcement learning using neural networks, we need a lot of training data to drive the network adaption and avoid gradient valleys.

2.3.1 Simulated environment

Within *Deep Q-learning*, Mnih et al. [51] and OpenAI [98] demonstrated control in complex environments. Their method using convolutional neural networks and memory networks to play computer games was however limited to a predefined discrete action space. Applying DQN to the continuous action space is not so straight forward as the method relies on maximizing each action-value pair, and iterating within the continuous domain makes each step an optimization problem itself, which is computationally impractical. One solution is to discretize the action space to such a level that does not break the system's ability to learn the underlying system dynamics, however, this may be inadequate within environments of higher action space complexity.

Lillicrap et al. [59] proposes a model-free, off-policy actor-critic method that is able to learn within high-dimensional continuous action spaces so that $a_t \in \mathbb{R}^N$. The work is based on extending the deterministic policy gradient *(DPG)* by Silver et al. [54] to what the authors call the *Deep DPG*. The work challenges the algorithm on several robotic control problems using two different observational spaces: one low-dimensional space based on sensory inputs like joint positions and cartesian coordinates, and a pure camera-based input using the pixel values directly. Training such algorithms is done in a virtual environment that can simulate physical dynamics and react on the actions proposed by the RL agent. Instead of learning the action-value directly, the states are mapped to a probability distribution over the actions $\pi : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{A})$.

An interesting discovery is that the same network architecture and hyperparameters could be used on different control tasks. Further, they also observed that the agent in some cases found policies that out-performed the dynamics engine itself. The algorithm makes use of the *experience replay memory* technique whereby samples are stored for later use for random training, and also *batch normalization* where samples in training on mini-batches are normalized to unit mean and variance. The reasoning here is that the network should slowly work towards convergence and minimize covariance shift, as well as accepting observations of different magnitudes. Another interesting approach is that the *target network* is also updated slowly by weighting the updates, and not directly replaced as done by Mnih et al. [51]. The stability gained outweighed the negative consequences of slow learning.

Exploration vs. exploitation of environments must be balanced, and the topic must be addressed when applying reinforcement learning. Exploration means that the agent is presented observations never or seldom seen before, to learn wider action policies. A popular method is the *epsilon greedy policy*, in which we randomly select an action in ϵ of the iterations to explore new parts of the environment. This is viable in discrete action space implementations but cannot be directly applied here. Lillicrap et al. [59] makes use of a noise function $\mathcal{N}$ added to the policy, so that

$$\mu' = \mu(s_t|\theta_t^\mu) + \mathcal{N} \tag{2.12}$$

$\mathcal{N}$ could be any noise function suited for the application; Lillicrap et al. [59] selected the Ornstein-Uhlenbeck function [1] to generate temporally correlated noise. Another way to explore is to present more samples to the network [63, p. 233] by data augmentation where random data disturbance is introduced. For pixels or image data, this can be done by e.g., zooming, rotating, occluding, de-focusing etc. [99]. Other ways are to introduce random disturbance is in the form of homoscedastic noise [110], Del Rio-Chanona et al. proposes noise $\pm 3\%$ the standard deviation [91], or by synthetically re-create a signal using Generative Adversarial Networks [68].

2.3.2 Transfer to the physical environment

Haarnoja et al. [81] demonstrates the use of reinforcement learning within the physical domain by creating a learning algorithm for a multi-legged walkable robot. They argue that as opposed to simulated environments, updates and hyperparameter searches in the real world cannot be done extensively. This would impose the danger of damaging the physical equipment through such trial-and-error policies. By applying the maximum entropy method, they show that minimal training is necessary to achieve reasonable performance early in the training, without utilizing simulated environments.

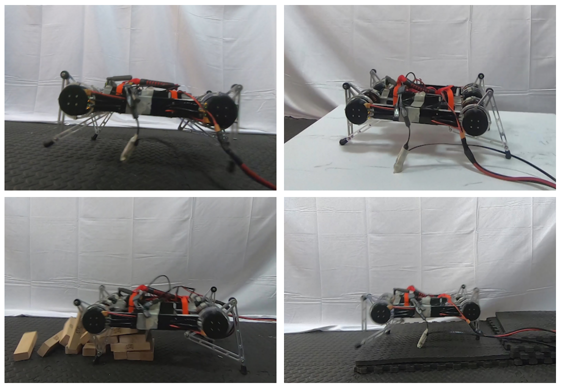


Figure 2.5: Illustration of a walking gait learned in the real world
Figure from [81]

They propose the *soft actor-critic* and removes the manually tuned *temperature parameter* α which is used to balance exploration of the environment with environment exploitation in maximum entropy algorithms. They also show that minimal hyperparameter tuning is necessary to reach stable performance across several control tasks. In this task, the robot executes continuously with no defined terminal state, except where humans must intervene due to physical constraints and safety. Hence, the given reward is *discounted* by a factor γ so that the sum of future expected rewards are finite:

$$R_t = \sum_{k=t}^{\infty} \gamma^{k-t} r_k(s_k, a_k) \tag{2.13}$$

The γ influences the control horizon where a smaller factor makes the algorithm more short-sighted than a larger factor.

Maximum entropy RL is robust and sample efficient, and the authors demonstrate that it can be insensitive to hyperparameters by dynamically tuning the temperature parameter α by Langrangian relaxation. Optimal entropy relies on the magnitude of the reward and policy, which develops during training to be more and more efficient, and again would influence the need to update the temperature parameter across tasks and trained policies. The authors add α to the value function, and periodically optimize this parameter during training to achieve an efficient physical control demonstration without the use of a simulated environment.

To the other extent, there are also examples of systems where all training is done in a simulator before being transferred and executed on a real system. This is an appealing approach where safe operation of equipment is critical. OpenAI [99] used a robotic hand to solve the Rubik's cube based on RL, computer vision and convolutional neural networks by the concept of a *Deep Q Network*. Central to the concept is *environment randomization* where changes and occlusions in the simulated environment introduce randomness in the training, and hence situations that are slightly new to the system. This would further help to generalize the trained network to the real world. A virtual simulator was set up replicating the robotic hand model and robotic environment, using three virtual cameras of the Rubik's cube as input to a *Deep Q Network* to estimate the cube's and finger's positions. All fingertips of the robot had a special pulsating light-emitting diode system attached so that the cartesian coordinates of all fingertips could be tracked by altering the LED pulse individually for each finger, via the same camera system. Further, the robot joint positions were also recorded.

The reward system was highly specialized towards the task; distance between current state and goal state and goal-reaching. Dropping the cube gave negative rewards. OpenAI used 1024 LSTM memory blocks in two separate networks, one for the policy, and one for the value network, where the value network was only trained in simulation as showed in Figure 2.6. There was also a separate observation network based on ResNet50, where cube positions were trained via supervised learning and inputted as real values to the RL network.

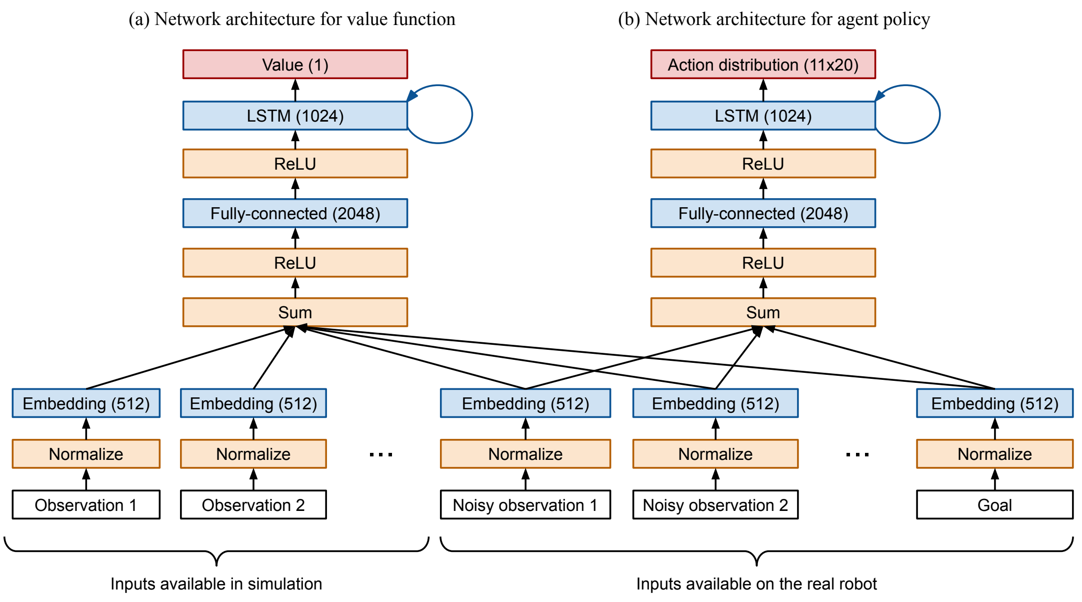

(a) Network architecture for value function
(b) Network architecture for agent policy
Value (1)
Action distribution (11x20)
LSTM (1024)
ReLU
Fully-connected (2048)
ReLU
Sum
Embedding (512)
Normalize
Observation 1
Observation 2
Noisy observation 1
Noisy observation 2
Goal
...
Inputs available in simulation
Inputs available on the real robot

Figure 2.6: Neural network architecture for (a) value network and (b) policy network
Figure from [99]

The combination of pixel- and sensory input make out a complex observation space where the network needs to observe as many perturbations as possible. In the simulator, parameters to the physics engine, vision and observation modules were randomized according to a *Automatic Domain Randomization* formula. When training was reaching convergence, the system gradually extended the parameters so that training could commence in a broader environment. The degree of domain randomization was dynamically adjusted according to the entropy of the parameter distribution. This method allows for no manually engineered randomizations and hence more efficient system implementation.

2.4 Bioprocess modelling

Advances in reinforcement learning raise the question of whether combining first principle dynamic models and modern machine learning concepts can be applied to modelling and control, and whether these models can be explained and tested for significance in the real world. Reinforcement learning has shown to handle nonlinear stochastic control problems well and is an attractive implementation alternative [110].

Del Rio-Chanona et al. [91] presents methods and experiments for improving bioprocess modelling based on traditional machine learning and dynamic models. The authors conducted bacterial laboratory experiments and studied the variation and equality between dynamic kinetic models, artificial neural networks and Gaussian process models. The intention was to compare control strategies for algae- and bacterial wastewater treatment, for the sake of efficiency and safety of bioprocesses. They also address the challenge of building data-driven models from scarce datasets and present a method for augmenting time series by linear interpolation.

Traditionally, kinetic models have been used for modelling bioprocesses and require extensive model studies. Kinetic models are basically differential equations that model cell growth, substrate uptake and end-product. Primarily, most time is used on the construction of such models and parameter optimization, to be applied to a specific process. These models need less data for training than machine learning methods but underlying knowledge of the chemical processes are needed, and the authors report that typical model construction times are several weeks, as opposed to machine learning methods where construction can be done in a few days. This, however, requires extensive and continuous sensory data for training, often not available, or at least limited. The article also refers to several studies where machine learning methods have been applied, but the problem of scarce datasets in bioprocesses limits the usefulness.

Laboratory experiments were conducted by the authors to study the effect of numerical prediction accuracy of glucose, nitrate, phosphate and biomass by the different models. The article shows that neural networks tend to diverge in prediction compared with traditional methods and data observations during the longitudinal progression of the experiments. They used a fully connected neural network without taking the sequence of observations into account, this would probably influence the results such that false conclusions are made. The authors point out that while machine learning methods need interpolated training datasets, sometimes kinetic models are the only choice, e.g. during process scale-up where no historical data exists. They claim machine learning methods can not replace traditional methods in every aspect, and so those two paradigms serve different purposes.

Applying reinforcement learning in such a context is particularly interesting since the long-term drift of such models could possibly be reduced, as well as the potentially positive economic aspects.

2.4.1 Process optimization on economic factors

Powell et al. [111] investigates the performance of their novel algorithm for process optimization using reinforcement learning. In their work, economic incentives are included in the model that directly controls a continuously stirred reactor process. As such, different external resources may have a differing cost depending on time: when power prices are low, more heat can be used in exchange for reactants, and vice versa. Economic factors can include power cost, but also commodity prices or other ambient conditions. Extending RL beyond steady-state control still uses many of the aforementioned methods and techniques. The article proposes that RL can be used in the real-time optimization domain, extending from the more traditional regulatory applications that use first-principle models. In cases where the plant model is unknown, the process is highly uncertain or where skilled manpower is unavailable, a data-driven methodology for optimization should be possible. They show that reinforcement learning is a viable alternative to non-linear programming methods in optimization tasks, although more work is needed to fully compete with them.

The idea is that the model is learnt from contextual data, environmental data and plant inputs. The model will respond with an optimal learnt policy in a continuous action space. The authors use the actor-critic method utilizing a deep neural network, and separates the value- and policy network into two different and standalone networks, where the value network is trained before the policy network, on process data alone (see Figure 2.7). Training of the policy network uses a static version of the value network to train discounted future rewards for a particular state-action pair of the policy network. The value network is trained using traditional backpropagation of the error derivate using a gradient-based solver. This is done on the plant's process data with corresponding decision variables from a historic recording, or from simulated data, as was done by the authors. The value network is trained on all available data at once, decreasing the risk of entering local minima as can happen when iterating on small datasets consisting of recent data only.

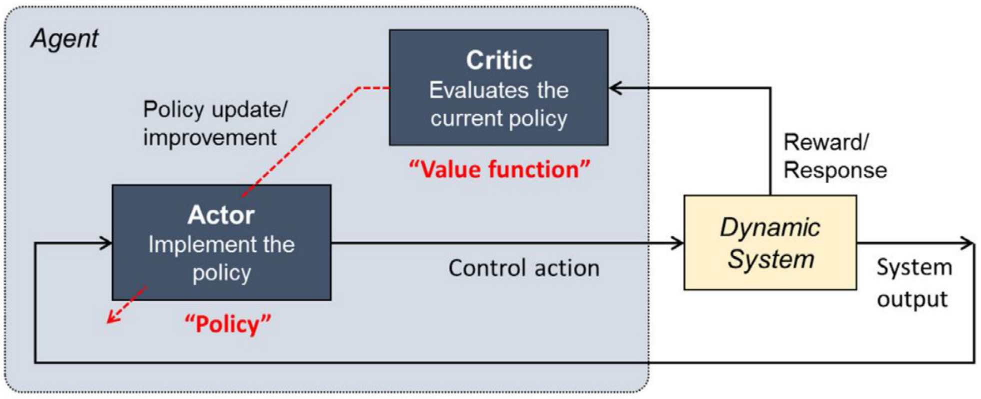


Figure 2.7: Actor-Critic RL design architecture
Figure from [111]

Solving the policy network function need careful consideration: the goal is to maximise the reward and an error-minimizing algorithm is undesirable. Instead, the authors train the policy network to select actions that maximise the reward using a non-gradient-based particle swarm optimization algorithm implemented in *Matlab*. The reward function was engineered according to the optimization goal: maximizing profit subject to environmental conditions. This could lead to actions that are undesirable from a normal plant operation view, so a *penalty term* was added so that actions were kept within a secure process envelope. Having these two goals as part of the reward function is an interesting feature to investigate in this thesis.

From a simulation perspective, as soon as the networks converge, they can be kept constant. However, in a real-time application, plant drift or unknown conditions must be catered to avoid a mismatch between the model and the actual plant. As such, the networks should be updated as new process data becomes available to compensate for such plant drift during operation, either in real-time or via asynchronous tasks on separate computing resources. The penalty term of the reward function should secure that the actions space does not over-compensate for such drift. The work shows that static learned policies can be used for real-time execution whereas updating them could be separated and run in parallel.

Without any knowledge of the system, the actor-critic method was able to learn strategies so that performance of the system was increased. This alone demonstrates the effectiveness of reinforcement learning as an alternative to non-linear programming in control- and optimization tasks.

2.5 Model predictive control

Mathematical models can serve as a basis for control algorithms, as described earlier. A theoretical model of the application domain is constructed to predict the future given the inputted actions. Feedback to the model with the state response would then be used to update the model's perception of the real world to make better predictions. The model would allow a selection of optimal control actions if it is able to describe the application model. In practice, modelling every aspect is not possible due to system drift, instrumentation noise and unknown system relations, so a feedback signal is used to correct the model accordingly for such deviations, to some extent. This *error signal* e_t is the measured deviation to some system *set point* of the monitored and controllable signal, $e_t = m_{setpoint} - m_{measured}$.

Many of today's control applications utilizes Propotional Integral Derivative (PID) control. PID is widely used in both the industrial and medical domain for basic closed-loop feedback control [38]. The basic idea is that deviations to the controlled quantity is corrected by the degree of error and the sum of past deviations. The input is then adjusted proportionally to the target deviation given a sum of three formulas:

$$A_t = K_p e_t + K_i \int e_t dt + K_d \frac{de_t}{dt} \tag{2.14}$$

where e_t is the *error signal* and the gain coefficients K_p, K_i and K_d are tunable parameters controlling the weight of each individual formula. Some systems utilize only the proportional part, while others a combination of two or all parts. This simple system is widely used due to its simplicity and performance.

A PID controller is *reactive* in that it will directly respond to the system state, while other methods allow for *predictive* control and hence falls under the umbrella *Model Predictive Control.* The modern research area picked up speed after World War II and is an attractive strategy for complex systems [13].

For MPC, the concept of the PID-controller is extended to include [106]:

1. A dynamic simulation model that can predict the future state given a sequence of control actions.

2. A mathematical solver that can optimize the actions given the control response from the simulation model.

3. Boundaries of acceptable action dynamics and states
4. Optimization goal that the solver should achieve

An algorithm will iteratively calculate the future states on each time step t, the *receding horizon*. The mathematical solver will adjust its parameters to regressively focus on the states that would achieve the optimization goal and recalculate the receding horizon prediction whenever new state observations are available and apply the first action to the environment. The response feedback is then re-evaluated in this iterative fashion.

Identification of the system model is historically exercised by identifying ordinary differential equations (ODEs) describing a dynamical system, often relying on first principle physical equations. Identifying the model involves data analysis and data wrangling, and a deep theoretical understanding of the application domain. In some cases, the staff is not available for such modelling, or the system complexity makes it economically unattractive. However, the model could also utilize data-driven methods by using neural networks [106].

In a dynamical system, next state is dependent on the current state, i.e. there is a Markovian relationship within the states of a sequence. Chen et al. [77] observes that residual neural networks models this behaviour where the output of a layer is the sum of the layer activation itself and the input:

$$h_{t+1} = h_t + f(h_t, \theta_t) \tag{2.15}$$

where $t \in \{0...T\}$ is residual block number and f is a function learned by layers inside the block. By rewriting to

$$\frac{dh_t}{dt} = f(h_t, t, \theta) \tag{2.16}$$

the hidden dynamics of f can be solved with an ODE solver, replacing the network to a *continuous network* where the function parameters θ are trained using gradient methods. The authors claim there are many benefits of learning this function, such as computation efficiency and constant memory utilization. Honchar [93] demonstrated applications of this technique using standard non-linear functions, and showed that a multilayer *hyperbolic tangens* network as ODE was able to learn a dynamic problem quite well using this method.

Tuor et al. [115] demonstrated the use of a neural network as a replacement of ODEs, where a system model can be learned by a sparse amount of data. The learnt ODEs are represented as fully connected layers using a rectified linear unit as activation function. They demonstrate that such a network can have stability guarantees by constraining the eigenvalues of the weights of the layers. The result is a network able to generalize physically consistent ODEs. Look and Kandemir [95] propose a Bayesian version of neural ODEs where prediction accuracy can be well established.

With relation to reinforcement learning, the MPC *receding horizon* prediction is trained within the *critic* network. That is, the future discounted reward for an action sequence is continuously updated as changes to both the network parameters and function approximation is done as new data is trained towards a reward signal. As such, reinforcement learning has the potential to succeed MPC as it is lightweight, data-driven and computationally effective and being able to constantly adjust for drift and system dynamics given that it is properly trained. By focusing on limiting the function dynamics to physically legal actions, the security of such a data-driven method should be achievable.

2.6 Agent training strategies

Drawing lines back to the early days of *TD-learning*, the research community has seen tremendous developments within reinforcement learning. *Deep Actor Critic*-methods are now some of the most advanced approximators in this field, involving deep neural networks in many combinations and utilizing many extensions. A non-exhaustive overview and relations of some of the current algorithms are shown in Figure 2.8, but many more exists. The *Soft Actor Critic* is an algorithm of many reasons applicable to the real world, as earlier described.

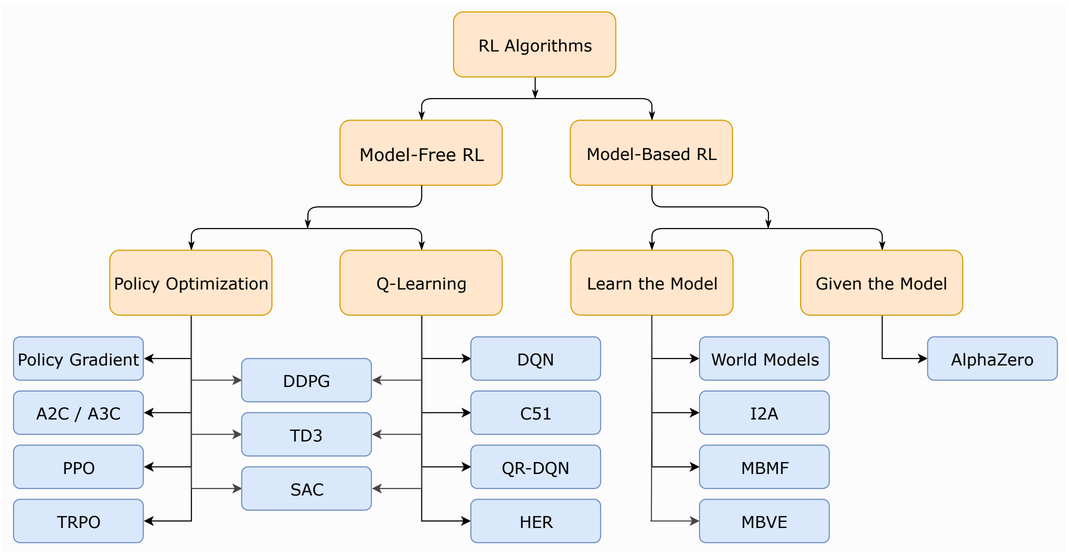


Figure 2.8: A non-exhaustive, but useful taxonomy of algorithms in modern reinforcement learning

Figure from [87]

Even though the combinations of possible methods and parameters are endless and somewhat intimidating, a template for actor-critic-algortihms from Bhatnagar, Sutton, Ghavamzadeh and Lee [37] is still valid under general circumstances. Algorithm 2.6 describes a framework for an agent: *input data, calculate action, observe reward, update value- and policy network*. Variations on the algorithm differ between methods, but still, the general template holds.

2.6.1 Representation learning

Neural networks architectures are roughly divided into *generative* and *discriminative* models. Fawaz et al. [78] reviews the state of research within this area for time series classification and regression tasks. A *generative* model tries to find a representation of the data prior to training. Such networks estimate the output variables through learning the probabilistic distribution of a set of latent variables. These are often used for data filtering and reconstruction tasks in multivariate problems, an example is the *Auto-Encoder* type of neural networks. *Discriminative* models directly learn the mapping between the input and output. For classification tasks, the output would be a probability distribution over the

Algorithm 1 A Template for actor-critic algorithms

```
    Input:
        Randomized parameterized policy $\pi^\theta$
        Value function feature vector $f_s$
    Initialization:
        Policy parameters $\theta = \theta_0$
        Value function weight vector $v = v_0$
        Step sizes $\alpha = \alpha_0, \beta = \beta_0, \xi = c\alpha_0$
        Initial state $s_0$
    for $t = 0, 1, 2, 3...$ do
        Execution:
            Draw action $a_t \sim \pi^{\theta_t}(s_t, a_t)$
            Observe next state $s_{t+1} \sim P(s_t, a_t, s_{t+1}$
            Observe reward $r_{t+1}$
        Average Reward Update
        TD Error
        Critic Update
        Actor Update
  end for
  return Policy and value function parameters $\theta, v$
```

class variables of the dataset, in a regression problem, the output is a continuous variable [78].

Convolutional Neural Networks (CNNs) is a special kind of architecture for handling image recognition and classification. CNNs are very similar to fully connected networks but assume input data that has a spatial relationship. This could be 1-dimensional data in the form of a time series, 2D images and also 3D spatial volumes like Magnetic Resonance images. A convolution operation is performed on the input data transforming a matrix to a more abstract representation given parameterized filter matrices, learned by backpropagation.

A deep network of multiple convolution operations has the ability to learn specific abstract details of the input and is analogous to how the visual cortex in our brain is thought to be architectured. The mammalian visual system is in one model described by two cell types: *S*imple and *C*omplex [8]. The *S*-cells identify basic shapes, and the *C*-cells combine a larger receptive field without being sensitive to shape positions. The neocortex is thought to store this information hierarchically, and the concept of the *neocognitron* used this analogy to recognise patterns after learning [10].

CNNs are a successor of this research, appearing first in the LeNet-5 CNN [23], describing a result of merging two sets of information, the input and a filter matrix and sliding this over the input. At every such location, a matrix multiplication with the filter is performed which results in a feature map for that location. The filter is designed to react on image features like edges, horizontal and vertical lines etc., but research has shown that using several randomly initialized filter matrices can produce better results than hand-designed filters, especially when followed by a pooling layer after an activation layer [45]. The filter matrices are the learned parameters in a convolutional layer, and we can apply several such filters per layer. Randomizing the first weights is a way of letting each filter travel in different descents. Pooling is effective to make further processing invariant to small changes in the input and gathers statistics of neighbouring activations.

Today, CNNs are superior within the image classification domain, but the methods can also be applied to multivariate time series: given a univariate time series $X = [x_1, x_2, ..., x_t]$ where $x \in \mathbb{R}$, a multi-dimensional time series X consists of m different univariate time series in a matrix: $X = [X_1, X_2, ..., X_M]$ as illustrated in Figure 2.9. An image is also formed as a matrix, often with several colour *channels* in addition representing as n matrices per image.

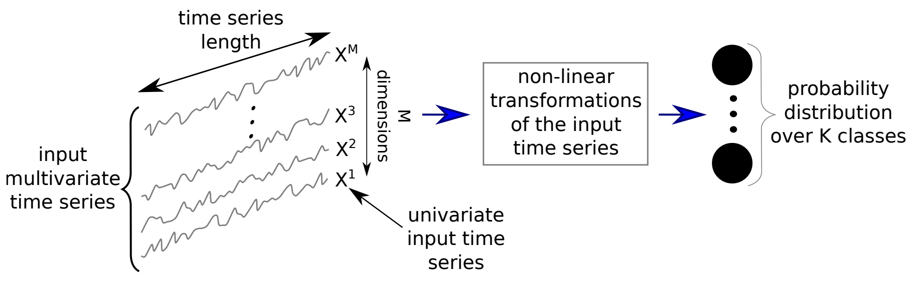

time series length
X^M
X^3
X^2
X^1
M dimensions
input multivariate time series
univariate input time series
non-linear transformations of the input time series
probability distribution over K classes

Figure 2.9: Multivariate time series
Figure from [78]

Networks based on CNNs are divided into a feature extraction part consisting of several layers of convolution, and a classification/regression part built on fully connected neurons. Applying convolution filters on a time series results in a new representation with dimensions equal to the number of filters used. Learning multiple discriminative features is achieved by stacking several layers of convolution. Applying a global pooling aggregation before the classification layers, reduces the number of learnable parameters and the risk of overfitting to the data [78]. Adding *Dropout layers* is another strategy to avoid overfitting. Dropout randomly disconnects neurons and is a simple way to introduce regularization [48], [55]. Such networks play well on all input data formed as matrices and serve as a basic discriminative neural network.

Adding a global pooling layer would also enable us to utilize *class activation maps* that explains a model's decision for a particular prediction [78]. The CAM method would identify which parts of the time series matrix is responsible for the prediction and hence reduce the black-box-effect of neural networks.

The sliding filter needs to be properly sized and is shown to be an important hyper-parameter known as the *kernel size.* Tang et al. [114] demonstrates the relation between larger kernel sizes and their ability to capture more correlations. However, oversized kernels might have useless features that must be trained to zero, thus requiring more training data. Under-sized kernels have less frequency resolution and do not capture important data features. Randomly initializing the filters leads to noise in the data that must be trained, but is not necessarily so that the training data exhibits features that train all filters. The authors argue that time-series data has multiple salient signals that require multiple kernel sizes to capture them. They propose that the kernel size should be part of the learning process simply by adding several convolution operations with different kernel sizes on each convolutional layer, visualised in Figure 2.10. The stochastic gradient descent will simply assign weights to the kernels that best represent the data during backpropagation, and the author's experiments confirm the performance. We also see that they have added a global pooling layer for the benefit of Class Activation Mapping and explainability of the model.

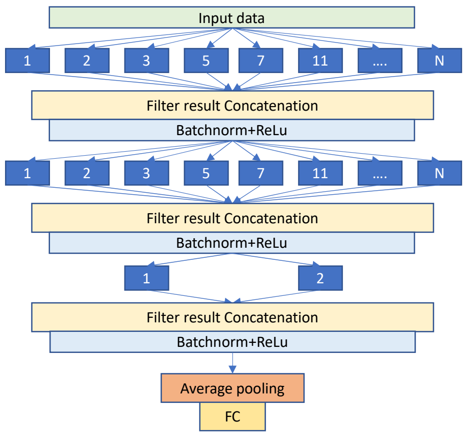


Figure 2.10: Different kernel sizes (blue box) within each layer
Figure from [114]

Recurrent Neural Networks (RNN) represents another form of networks with *memory*, but is seldom used for classification, according to Fawaz et al. [78] due to the problem of vanishing gradients on training on long time series. Further, the computational resources needed are greater, making RNNs a complex alternative. RNNs has seen success in forecasting time series, although another variant, the long-short-term memory (LSTM) networks have proven better accuracy [101].

2.6.2 Partial observability and physical consistency

Hausknecht and Stone [58] demonstrated that adding recurrency to a convolutional network-based reinforcement agent would benefit the agent's possibility to act on partially visible states. They argue that a single observation is not enough to estimate the state of all agents, which in case would make them non-Markovian. Such agents should not be limited to using the current input data frame only. By replacing the fully connected layers in the classification parts of a network with long-short-term memory blocks (LSTM), the agent becomes a *Partially-Observable Markov Decision Process*.

Using LSTM as the classification layer is extensively demonstrated by the architecture of Dota 2 by OpenAI [98]. In *Solving Rubik's cube with a robotic hand* [99], the visual model and policy network is separated, where the visual model uses CNNs and the policy network is LSTM-based.

Hausknecht and Stone [58] observe that performance of pure DQN agents decline when given incomplete state observations, but by adding recurrency, the agent is better able to capture the underlying system dynamics. However, their experiments show that adding recurrency has no systematic benefit over stacking input frames to the convolutional network as separate image channels, where each channel represents a moment in time (x_{t-2}, x_{t-1}, x_t) rather than colours (r, g, b). We must remember that the authors use gameplay with virtual screens as frame inputs to the agent. Stacking the frames leave a trail of history, and in the case of games, physical properties of the play can be deducted from this information, as illustrated in Figure 2.11.

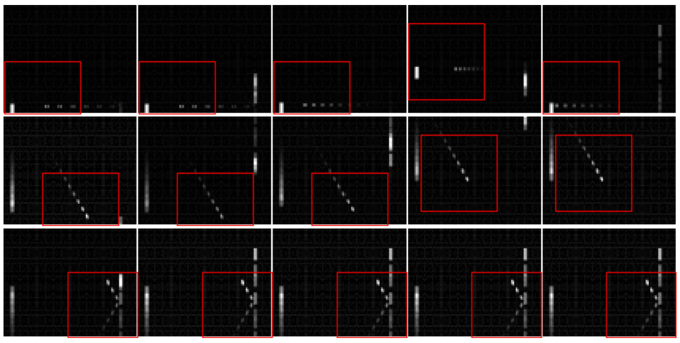


Figure 2.11: Stacking images leaves trails of history for physical model identification Figure from [58]

The added information from the time-sequenced frames would indicate that the agent can learn the physical dynamics of the underlying system. Karpatne et al. [70] demonstrated that neural networks benefit from implementing physical consistency rules in the loss function of the stochastic gradient descent algorithm. They argue that sparse datasets would benefit from constraining the learning rules to be physically consistent. There is no guarantee that a standard loss function

$$Loss(\hat{Y}, Y) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2 \tag{2.17}$$

would be physically consistent. By adding a physical model to the loss term, the authors were able to demonstrate that a network modelling inland water temperatures would converge and provide more detailed results than currently available methods. The physical model would generate a negative or positive number according to the input, and during inference a standard rectified linear unit would negate data that do not conform to the physical model, thus pushing the neural network to behave *physically consistent*.

2.6.3 Pre-training and imitation learning

The effect of randomness in the initialization of neural networks is well established within supervised learning [45], [63], but Fawaz et al. [78] demonstrated a significant decrease in accuracy on a classification task given *bad* randomisation. There might be several explanations to this effect, like entering into different local minima for each training session or being exposed for vanishing gradients for long time series. In recent years, adding rectified linear units (ReLUs) as activation functions has shown to be effective. Using multiple convolutional layers instead of many fully connected layers is also useful, however as a general rule, training deep networks with random weight initialisation provide worse results than shallow networks [101]. One approach to succumb to the challenge of entering local minima due to *bad* randomisation, is pre-training the network weights with a generative model.

Sagheer and Kotb [101] propose a method for pre-training a network using an LSTM-based stacked autoencoder (LSTM-SAE). Stochastic gradient descent has a difficulty of learning long-term dependencies, but the authors developed a deep-LSTM memory network, capable of capturing these long-term dependencies in a time series forecasting problem [100].

When training a machine learning model, our aim is that the model capture only relevant information from the data presented and filters noise, bad samples and unknown data. Using neural networks, many data samples are necessary to push the weights in the correct directions. When the network complexity increases due to a increase in number of trainable parameters, training is demanding as compared to traditional machine learning algorithms. Further, there is no convergence guarantee for the network.

Sagheer and Kotb [101] train an autoencoder to learn the representation of the data in an unsupervised manner in a method they call *greedy layer-wise pre-training*, visualised in Figure 2.12. They successively add layers to the stack during training, to increase the search space while preserving the already collected knowledge in the upper layers. Initializing the upper layers first might push the weights in the direction of high-level abstractions of the data, an idea that has been proposed earlier by others and investigated by the authors. As more layers are added, better detail is achieved.

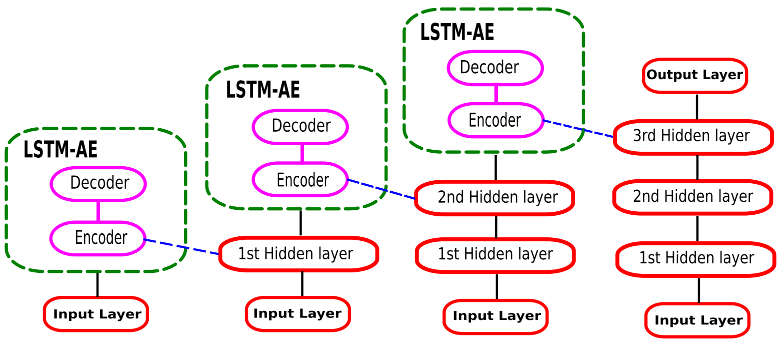


Figure 2.12: Greedy layer-wise pre-training
Figure from [101]

When the *pre-training phase* ends, the result is encoder-layers of our network with learned weights for the data representation that we disconnect from the autoencoder-architecture. An output layer is then added for the classification or regression problem at hand, and the *fine tuning phase* starts. In this phase, standard supervised learning is executed to push the classification layer weights towards the problem solution. The authors were able to demonstrate improved performance and faster convergence than other models. They also conclude that this method is suitable for situations of correlated multivariate input.

Using a layer-wise pre-training on time series could also alleviate the problem of sparse data by pushing the model in the correct *direction* earlier in the training process.

Another related method of interest is described by Ho and Ermon [64]: *Generative Adversarial Imitation Learning*. Their framework claims to directly extract policy from data through generative adversarial networks, training a network **G** to confuse a discriminative classifier **D**. When network **D** cannot see the difference between **G** and real data (given a reasonable threshold term), successful cloning of expert behaviour is met. They tested the algorithm on OpenAI testbench environments and showed that the method was able to perform close to expert's behaviour on several physical environment tests.

2.6.4 Latent space training

Ha and Schmidhuber [80] proposes to separate agent models in three parts: vision **V**, memory **M** and controller **C** in order to effectively train the networks for the parts they actually focus on. They argue that the vision model could be trained in an unsupervised fashion using autoencoder convolutional neural networks to represent the complex input frames in an abstract, compressed world without convergence towards a supervised output. This model could be trained separate from the task, meaning that excessive computing power for representing the temporal and spatial properties of the world can be done once, and reused for several tasks. Continued training of the vision can be done over time for adding new representations. While the vision model compresses the moment, the *memory model* abstract the time axis, abling prediction over what happens in the future by remembering through recurrent neural networks, or LSTMs.

The last controller layers are deliberately made small and shallow, to keep the complexity of the problem reside in the **V** and **M** models. This means the controller can be retrained for several tasks in an effective fashion. Figure 2.13 show this architecture in relation to what the authors call a *World Model*.

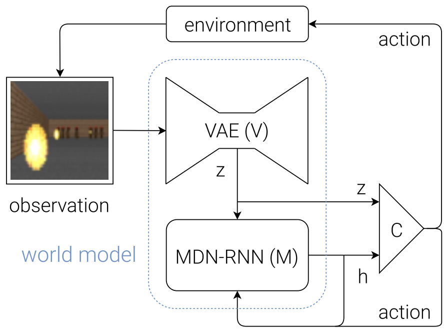


Figure 2.13: Vision **V**, memory **M** and controller **C** networks forms the World Model
Figure from [80]

The latent space model in the autoencoder of **V** enables us to separate the *real world* from a *simulated world*. In effect, a controller can be trained on a simulated world and act accordingly when transferred to the real world, which makes this an interesting architecture for cases where execution on real actuators is limited, costly or not preferred due to health or safety. The authors were able to demonstrate superior performance in OpenAI environments.

Robine et al. [113] further extends this idea by increasing the latent space into the multidimensional domain to preserve world representations, using a model-free Proximal Policy Optimization. They demonstrate superior performance in an Atari gameplay context. Another change is the ability to predict the next latent variable based on the previous latent variable and action, independent of the observation. The reward should not be dependent on the next latent variable as well. Further, the policy is trained on the latent variables.

2.6.5 Regularization

According to Thodoroff et al. [89], reinforcement learning in high-dimensional domains suffers from instability due to high variance. There are many sources for this variance, like randomness in data collection, state of initial parameters, the complexity of the learner, hyperparameters and of course sparsity in the reward signal. Regularization is one way to increase generalisation. An example is to introduce data augmentation on the signal input, i.e. adding noise of some distribution. However, the authors suggest smoothing the reward trajectory through temporal regularization. An important assumption is that a state is dependent on the previous state in a sequence, that is, it occurs after a known state. A time series is such a sequence where each value is somehow connected to the predecessor and falls under the umbrella of being a Markov Decision Process.

The goal is to find the policy that maximizes future expected return. The method is simply to smooth the value estimate of a state with estimates occurring earlier in the sequence. The smoothing function could use standard methodology from time series prediction, like exponential smoothing, ARMA etc. The authors demonstrate that this method reduces variance and helps the learning process. They implemented the algorithm in a high-dimensional setting using an actor-critic trained by proximal policy optimization (PPO), exponentially smoothing the target of the critic by temporal regularization, controlled by a decay factor λ as a hyperparameter. This improves the performance when testing on a suite of Atari games, reduces variance and increase robustness. The authors claim the method is particularly appealing for real-world applications where agents might not observe all states during training, i.e. the data is sparse.

Instead of assuming that rewards are similar for spatially close states, the authors turn the RL problem around and assume that the rewards are similar for states *visited closely in time*. This could prove useful for the exploration of new states.

Antoniou et al. [68] presents a more traditional route to regularization by training a generative adversarial network to produce data. The model learns to produce data that are generalised within the same class, exhibiting similar features as expected from a missing value. The point is to generate more data for training to avoid overfitting to a small sample space. The generator network **G** produces samples, and a network **D** tries to discriminate the generated samples from real samples according to the *Wasserstein distance*. The authors conclude that the method improves the performance of classifiers and is applicable in low-data settings.

2.6.6 Experience replay

Experience replay is a method to increase the available samples during the training of a reinforcement learning agent, as we have seen several examples of earlier. Typically, a circular buffer with a defined capacity keeps the n latest transitions in memory or persistent storage. When training, random samples from this buffer are mixed with new transitions and is used to avoid overfitting the network, avoiding a skew towards the last visited states. The buffer increases the available sample size when training the RL network, and is a way to avoid the network to *forget* earlier knowledge. Fedus et al. [105] review several aspects of this method and studied the effects of different replay buffer parameters towards a DQN network.

It is worthwhile to draw a line between this method and neuroscience. According to O'Neill et al. [42], experience replay is suggested to explain hippocampus-dependent memory formation in the brain. This happens in two phases, slightly analogous to our usage of experience replay within reinforcement learning. First, memories are encoded during full wakefulness, that is, added to our buffer. In phase two, sharp wave/ripple events are fired in the hippocampus, initiating the start of our training. Here, the memories are either transferred to long-term storage or used for strengthening associations in the brain. This likely happens during sleep, according to emerging evidence. Several parts of the brain are active during this time, coordinated by the hippocampus.

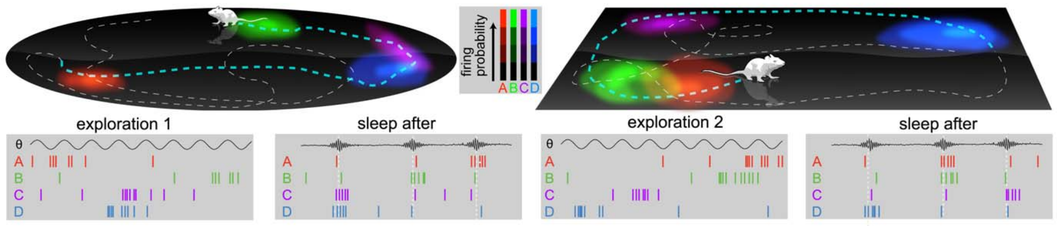


Figure 2.14: Reactivation patterns during sleep after hippocampal reactivation
Figure from [42]

Three important factors determine the experience replay performance, according to Fedus et al. [105]. The *replay capacity* is the total number of transactions stored in the buffer, the *age* can be expressed in the number of steps since the transition was inserted, and the *replay ratio* is the relative frequency between transitions from the buffer and the environment used for training. Their experiments show that increasing replay capacity increases performance, although keeping the replay ratio fixed has varying improvements. They also discovered that using n-step returns on increased capacity buffers is a critical factor for taking advantage of larger buffers. They tested the extreme where an RL agent was trained by offline data only, without any environment to interact with, just the recording. A setting of $n > 3$ showed very positive results. This discovery would enable us to architecture agents where training is done separately from the inference platform.

2.6.7 Inverse reinforcement learning

Reinforcement learning is about finding a policy $\pi : S \rightarrow \Delta_A$ that maximizes the future expected reward for each state-action-pair $(s, a) \in S \cdot A$. *Inverse Reinforcement Learning* (IRL) on the other hand, is about learning the reward function given system dynamics and expert behaviour. One example is the reward function for autonomous driving that should capture the desired behaviour of the driver. Such behaviour could be stopping at red lights, avoiding pedestrians and one-way roads etc. [76]. Engineering these rules would require us to identify a list of every such situation, as well as each rule's importance towards other rules. One example of a very detailed reward function is the one engineered by the Open AI Dota 2 team.

Within IRL, a policy or sequence of actions embedding some expert behaviour is used to find a reward function that explains this behaviour. We note that the reward function completely captures the optimal set of policies in a system. An important assumption is that the expert acts optimally so that we can estimate the function that led to this behaviour. This is our first attempt at estimating an actor's *own actions*, guided by expert advice. We might be philosophical at this point and warn about the consequences of learning *bad* behaviour. How can we alleviate a situation where an agent acts against human actions?

Ramachandran and Amir [33] considers the problem from a Bayesian perspective where a probability distribution over the reward space is estimated. Expert behaviour is seen as the posterior *evidence* that is used to update the prior using a modified Markov Chain Monte Carlo (MCMC) algorithm. They claim that this method allows imperfect pieces of expert advice, as well as multiple expert inputs. However, they assume that an agent **X** always choose actions that maximize the future reward, meaning that all actions are *perfect* in the sense of being less exploratory. Further, they assume a *stationary policy* meaning that any policy for action a is independent of the previous sequence of actions. This would imply that the expert always *exploit* his or her knowledge without exploring new paths in the environment since it is assumed a fully learned expert. However, Bayesian statistics and the Central Limit theorem assumes normal distributivity, and thus we would accept some expert *noise* in that respect. The method makes the following assumption:

$$Pr_X(O_X|\mathbf{R}) = Pr_X((s_1,a_1)|R)Pr_X((s_2,a_2)|R)...Pr_X((s_k,a_k)|R) \tag{2.18}$$

where the probability distribution of expert behavior $O_X = (s_1,a_1),(s_2,a_2),(s_k,a_k)$ given a reward function $\mathbf{R}$ is the individual reward function distributions for each state-action pair, multiplied through the chain rule of probability [123]. Each product in the chain is independent, so for each state-action-pair we would estimate the probability of (s_k,a_k) given $\mathbf{R}$. Finding the distribution of $\mathbf{R}$ would mean we want to find the distribution that maximizes the value of (s_k,a_k). This is an optimization problem with calculation of priors depending on the application. The authors explain *reward learning* and *apprenticeship learning* as methods for such and demonstrated implementations of gameplay using the framework.

Lopes et al. [39] uses the IRL-perspective in a sense of *active learning.* The authors argue that their algorithm query an expert when needed since a policy is unlikely to be completely specified beforehand, instead of learning from static state samples. The algorithm actively learns from expert samples when needed, that is when informative decisions for the policy needs to be taken. The problem space of learning the policy on sampling only is complex, meaning that the system will likely calculate multiple optimal reward functions and multiple possible policies for one reward function. In comparison to MCMC, which is an *uniform sampling approach*, the active learning algorithm could potentially use the most recent learnt expert advice for a *local control problem* not experienced in the general sense.

The agent is given a set of expert demonstrations $\mathbf{D} = (s_t,a_t)$ and updates an estimate of possible reward functions $P(r|\mathbf{D})$. When the probability distribution's standard deviation is above some threshold, the agent becomes unsure about the policy and can query an expert for the correct action. Their method reduces the number of samples necessary to learn the system and they argue that the Markov Chain Monte Carlo-algorithm (MCMC) from Ramachandran and Amir [33] becomes computationally expensive in large dimensional spaces. The method was demonstrated on several problems describing a Markov Decision Problem consisting of a continuous state space and discrete action space. A defined reward function using some relation to the problem domain where the parameters are estimated by the algorithm should be a better choice than just defining $r : \mathbf{X} \to [0;1]$, i.e. a sparse reward space. In the latter case, the authors indicate that the problem may be too complex for active learning due to the computational cost.

A real-world example of IRL active learning is given by Ziebart et al. [36]. They employ *maximum entropy* to the IRL problem originating from the stem that expert behaviour has noise and that a system has difficulties of differencing policies when such noise correlates to several policies. By maximizing the entropy, the uncertainty of expert behaviour is reduced. They demonstrate the method in a *driver route modelling* problem including route preferences and possible destinations. Destinations for a driver are modelled using *Bayes theorem.*

We could relate this problem to Florensa et al. [69] where agent goals are generated during the course of agent experience. The idea is that an external adversarial network generates tasks that the agent has the knowledge to achieve, slightly making each new task more complex. When a task is complete, a new slightly more difficult task is generated. The authors claim this method allows for sparse reward methods, i.e. where the reward is either 0 or 1.

2.6.8 Soft Actor Critic

As described by Haarnoja et al. [81], a *Soft Actor Critic (SAC)* algorithm has shown sample-efficient attributes and might have less sensitivity to hyper-parameters. SAC utilizes maximum entropy for choosing exploratory actions that maximize the reward, with minimum pre-learning. Haarnoja et al. [82], [83] describes an *Off-Policy Maximum Deep Reinforcement Stochastic Actor* with continuous state and action spaces and three distinct characteristics: separate value- and policy networks, off-policy formulation and entropy maximation. Instead of just acting on *future discounted rewards* as shown in Equation 2.8, SAC utilizes *maximum entropy* in order to act as randomly as possible while still completing the tasks.

Let $J(\pi)$ be the *value function* of a policy, then

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)\sim p_\pi}[r(s_t, a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))] \tag{2.19}$$

where the *temperature parameter* α controls the relative importance of the entropy term against the reward which must be tuned. Further, gradient update of the policy is done off-policy utilizing the history of available data from a replay buffer. This gradient update means slower learning of the agent than just replacing the target network as done earlier by Mnih et al. [51]. The soft policy update alternates between policy evaluation and policy improvement as described by Sutton and Barto [112]. The stochasticity of the policy and value update prevents premature convergence and encourages exploration due to the maximum entropy objective. The authors were able to demonstrate an agent that quickly learned the underlying dynamical model based on actions in the real world.

Chapter 3

Use Case

3.1 Introduction to spray drying

To investigate the main research questions, a delicate industrial drying process is selected for analysis due to its widespread use and available scientific material. Drying material substances is widely applied in all types of industries, and there are many technologies implemented to either reduce moisture content or completely dry out water from a product liquid stream. Using spray drying technology, a liquid material can be nearly completely dried to a single material substance in the form of solid particles. This process poses an interesting problem since the control of the spray drier process equipment directly affects the end-product quality, and that there is an optimization potential on other factors as well. Still, the operation of a spray drier involves some manual adjustments and knowledge for optimum operation.

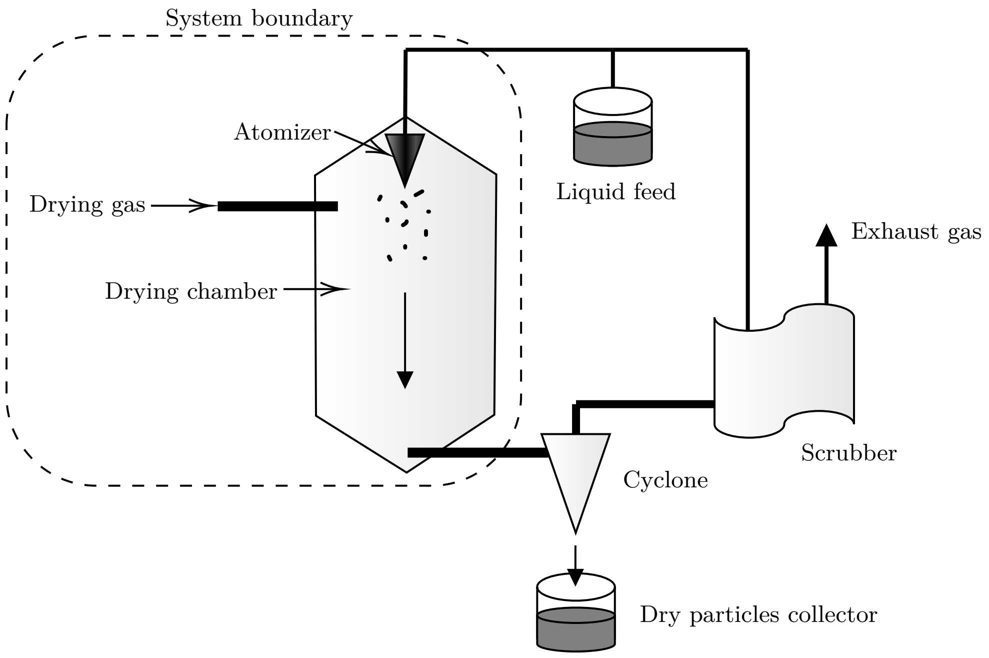


Figure 3.1: A typical spray drier setup

Spray driers are used in the production of food and feed, detergents, polymers, pharmaceuticals and within the pulping industry to convert liquid lignin to powder. Reducing product moisture content means efficient logistics and transport, as well as potentially longer storage periods.

By definition, *spray drying is the transformation of feed from a fluid state into a dried form by spraying the feed into a hot drying medium* [40]. The technology dates back to the 1860s and has of course undergone massive development since then, however, the underlying principle is the same today. The first patent dates back to 1872 by Samuel Percy, for the production of milk powder by spray drying [75].

Figure 3.1 give a schematic overview of a typical spray drier process and is described in several sources ([40], [88], [11]): a liquid product feed is discharged into the atomizer, a critical component that converts the substance into very fine particles. This is done either using high-pressure nozzles or a spinning disc, where the aim is to increase the surface area of the liquid through the liquid disintegration phenomenon [88]. The selection of the design of the atomizer is carefully done according to the type of material to deplete.

When the liquid spray enters the drying chamber, gravity pulls the particles downwards and turbulent gas flows in the form of heated air either with- or counter-flow the product stream. This phase is called the *particle formation phase* [88] where the large droplet area contributes to efficient energy exchange. As the particle heat is increased due to hot-air contact, moisture is exchanged due to the evaporation of water inside the particles. At the bottom of the chamber, moist hot air and dry powder exit through a powder separation system using vacuumized cyclones. Particles are accepted at the bottom of the cyclones, whereas the reject stream is the moist and hot air that can be released as emission gases.

The emission gas is most often cleaned through wet scrubbers in order to minimize the escaped particle content and odours released into the atmosphere. The collected particles can be re-introduced to the liquid product flow and will as such blend with the liquid and re-agglomerate.

Hot air is used both as a drying medium and for transport of the product flow. For the efficient operation of the spray drier, a delicate balance between airflow, air temperature, feed flow and -temperature as well as atomizer speed and cyclone vacuum level, must be maintained. Even moisture of the incoming airflow will influence the process, as the moisture difference between the airflow and product particles directly affects the timing of the drying process. If the incoming air is moist (as in the summertime when hot weather increases the air absolute humidity potential), the heat difference between air and product must be increased to assure a proper level of evaporation. Chamber size must also be designed such that enough drying time is exerted for the product in question. The feed flow viscosity and feed temperature also influences the process and is of likewise importance.

Further, the process operation also dictates end-product qualities, mainly the end-product moisture content, particle size and density. The optimum operation window varies in time, given the large operational envelope and degrees of freedom.

3.2 Controlling the process

Spray drying is still preferred over other methods mainly due to its continuous and efficient single-step method that shows great adaptibility [88]. Further, the end-product does not need any additional processing and the process is excellent for heat-sensitive materials.

By empiric observation, some guiding relationships of the process parameters exist. Santos et al. [88, p. 16] give an excellent overview over the inherently interconnected relationships of some major process parameters in Table 3.1.

Parameter	T_{out}	$Droplet_{size}$	h_{out}	Efficiency
Drying air flow rate	↑↑		↓↓	↑↑
Air humidity	↑		↑↑	↓
Inlet temperature	↑↑↑		↓↓	↑
Atomizing air flow	↓	↓↓↓		
Feed rate	↓↓	↑	↑↑	
Solid concentration in feed	↑↑	↑↑↑	↓	↑↑

Table 3.1: Relationships between spray drying parameters
By increasing the parameter in the first column, the rate of increase/decrease in the related parameter is shown by corresponding arrows Source: [88, p. 16]

These process parameters relate to the outlet temperature, particle size, final product moisture and overall efficiency of the spray drier. Some parameters can be controlled through engineering, but some is a consequence of weather, feedstock and equipment design. The spray drier operational envelope is designed according to the feed type and target efficiency, so some parameters will also be fixed, like chamber size, atomizer technology, chamber insulation, cyclone size and efficiency etc. Air heating technology and feed pre-processing are also fixed according to the application. Inherently, spray drying is an energy-intensive process and actions for preserving energy, energy recycling and reducing energy consumption is also part of the equation of designing a plant.

Sensing the important characteristics of the process should be done by careful digitalization by appropriate sensors and data collection mechanisms. A pilot plant is exemplified by Pote and Sudit [61]. Here, an architecture for controlling the process via PID regulators is established with proposals for further work using first-principle models and grey-box models. Moisture content as a quality parameter could be indirectly controlled by maintaining a specific outlet temperature. This could be controlled by varying either the feed flow or the inlet temperature according to Parastiwi and Ekojono [65]. Other product quality parameters are difficult to control, like product thermal degradation, aroma retention and structure and size of particles. Parastiwi and Ekojono [65] showed that applying a simple PID controller using outlet temperature as the feedback signal, greatly stabilizes the spray drying process, at least when considering the moisture content and outlet temperature relation.

3.3 Modeling the process

Understanding the underlying process is necessary to develop control schemes that target specific optimization criteria, like increased quality, less energy consumption or higher throughput. A mathematical model could be established as a source for finer process control. However, this is a complex task and according to Azadeh et al. [46], claiming that traditional regression methods can not model the process. The reason is explained through the non-linearity of the process as well as the current state of sensor technology. Zbicinski [75] reviews the state of spray drier modelling and simulation, showing several challenges with mathematical models when scaling up a spray drier design. However, advances in computer simulation show a positive correlation with physical experiments and is now considered an established method, involving Computational Fluid Dynamic simulations (CFD).

The question of creating mathematical models as a universal problem-solving method comes to a point of available competence, funding and evaluating return-on-investment within a research area. Oakley [26] describes a layered approach to spray drier modelling where items can be modelled at different levels:

- Level 0 Heat and Mass Balances.
- Level 1 Heat and Mass Balances with solid-vapour equilibrium.
- Level 2A Rate-based with simplifying assumptions about particle motion.
- Level 2B Rate-based with a simulation of gas flow and particle motion (CFD).

Oakley [26] uses the word *fidelity* to describe the model quality according to the level. *High fidelity models* give detailed predictions with the added cost of work and computation.

Modelling heat and mass balances is an established science field used in many situations. The core is understanding the physics and making an informed assumption of the necessary *model fidelity* according to the application. Hence, the layered approach by Oakley [26] is important when designing a modelling strategy.

3.3.1 System boundary

We will create a simple level 1 model for our use case as visualised in Figure 3.2. This model serves several purposes: first, it will be used for creating synthetic process logs for later generative modelling. Second, the model will act as the environment for our RL agent when evaluating its performance.

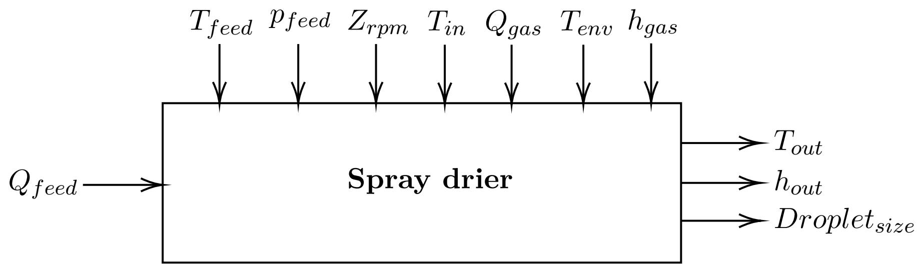


Figure 3.2: System boundary for the model

The model is parameterized by:

- Manipulated variables: Process variables we directly control from the control simulation
- Controlled variables: Process variables we can restrict for the purpose of experiment design
- Responding variables: Process signals describing the response of the system as a function of the manipulation

3.3.2 Level 0 Heat and mass balance model

A level 0 spray drier model is established using the fundamental laws of conservation of mass, stating that the mass of a system remains constant over time and cannot change [124]. Materials can neither be created nor destroyed but can change their form. Further, all inputs and outputs add or subtract to the total mass balance. Our mass balance formula considers three compartments of mass, in the form of solids, liquid and gas, adapted from Oakley [26]:

$$F_{feed,solids} + F_{in-gas,solids} = F_{out-gas,solids} + F_{product,solids} \tag{3.1}$$

$$F_{feed,liquid} + F_{in-gas,liquid} = F_{out-gas,liquid} + F_{product,liquid} \tag{3.2}$$

$$F_{feed,gas} + F_{in-gas,gas} = F_{out-gas,gas} + F_{product,gas} \tag{3.3}$$

where $F_{i,j}$ is the mass flow (kg/min) of component j in compartment i. To model the energy balance, we use the first law of thermodynamics [126]:

$$\Delta U_{system} = Q + W \tag{3.4}$$

stating that the change of energy in a *system* is equal to the difference between the heat supplied to the system (Q) and the work (W) done *on* the system. We model the energy balance of our system as:

$$F_{feed}H_{feed} + F_{in-gas}H_{in-gas} = F_{out-gas}H_{out-gas} + F_{product}H_{product} + Q \tag{3.5}$$

where H_i is the enthalpy (J/kg) of compartment i and Q (Watt) is heat exchange between the system and the environmen, mainly heat loss. The model will ignore the heat exchange parameter.

Model implementation and results

We rewrite this to model the feed and gas streams we can measure by sensors (or at least give a good estimate on) by integrating a rate formula for the mass balances:

$$\frac{dy}{dt} = -F_{feed} + \dot{F}_{feed} \tag{3.6}$$

$$\frac{dy}{dt} = -F_{in-gas} + \dot{F}_{in-gas} \tag{3.7}$$

where F (kg) is the state and $\dot{F}$ (kg/min) is the rate of state change given by the formula:

$$\dot{F} = \frac{\dot{Q} \times \rho}{\tau} \tag{3.8}$$

where $\dot{Q}$ is the stream rate (m^3/h) and ρ the stream density (kg/m^3) and τ the simulation time constant. The stream density can be either measured (for the feed) or estimated (for gas). The density of air varies with temperature [125], but is not modelled here and set constant. The energy balance considers only an instant heat flux between the two streams and we apply the first law of thermodynamics to acquire the heat capacity of the system as:

$$\Delta E = mc\Delta T \tag{3.9}$$

where m is the mass (kg), c is the heat capacity (J/kg°C) and ΔT is the temperature difference between the streams. Our heat flux between gas and feed energy is

$$\dot{T} = \frac{\Delta E}{\dot{F}_{feed}c_{feed}} \tag{3.10}$$

Applied to our differential equations, this gives us

$$\frac{dy}{dt} = -T_{out} + \dot{T} \tag{3.11}$$

where T_{out} is the estimated temperature of the output product feed.

For the sake of the simulation, and learning the RL agent, we will consider the feed rate $\dot{Q}$ as our variable to manipulate, and the T_{out} as the interesting responding variable. Keeping the other variables constant, yields the result according to Figure 3.3 and Figure 3.4 when running the simulation in *Matlab*.

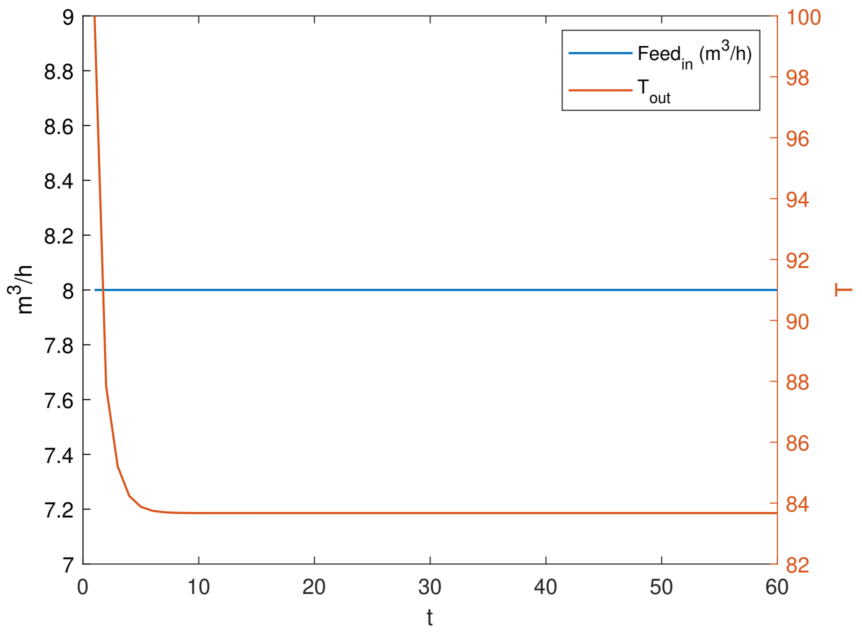


Figure 3.3: System response with a constant feed rate $\dot{Q} = 8$

Intuitively we observe that the temperature decreases as the feed rate is increased, in line with the reported relationship by Santos et al. [88]. As the feed rate increases, more solvent needs to be evaporated and the energy flux from the hot air is increased. If the energy level is kept constant, the temperature drops.

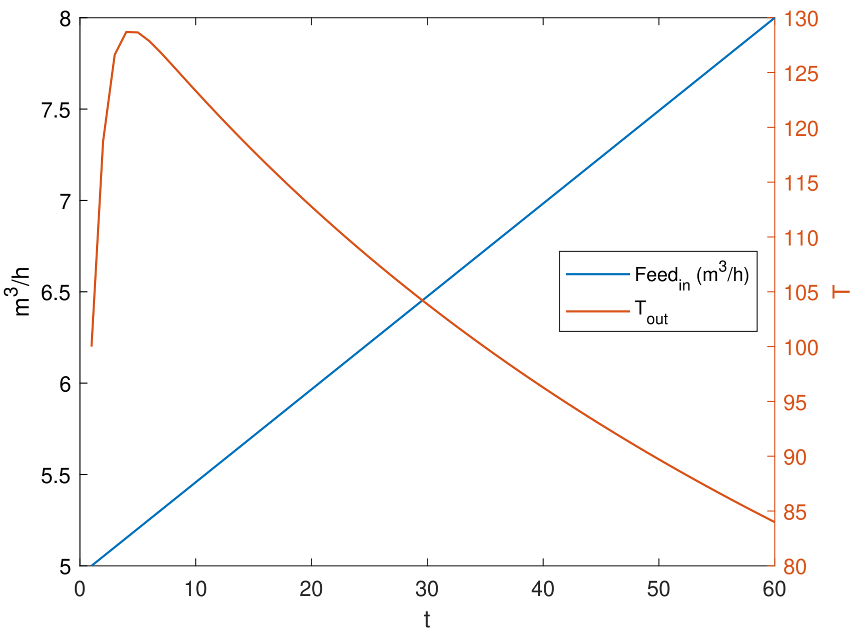


Figure 3.4: System response when ramping up feed rate $\dot{Q}$ from 5 to 8

3.3.3 Level 1 Droplet size and evaporation dynamics

We extend the model to include droplet formation and evaporation dynamics, by using the above mass- and heat balances as input. The formula *Sauter mean droplet size* [35] is an empiric formula that model the droplet size from a rotating disc atomizer:

$$D_{vs} = \frac{1.4 \times 10^4 M^{0.24}}{(Nd)^{0.83} \times (nh)^{0.12}} \tag{3.12}$$

where D_{vs} is the mean droplet diameter (µm), d is atomizer disc diameter (m), h is atomizer disc vane height (m), N=disc rotating speed (rpm) and M is mass feed rate (kg/h). This formula is found and tested by experiments [35], and one must assume that the overall design of a spray drier, and the atomizer in particular, heavily influence the model. Hence, it might not be representative of all spray driers on the market, and one should carefully implement designed experiments to create a model that suits a particular process. For our simulations, we assume that the model is sufficient. Figure 3.5 demonstrates the model output over a feed rate of 5 to 8 m^3/h while keeping the atomizer rotating speed fixed at 7,600 rpm.

Further, modelling the evaporation dynamics can be a daunting task depending on the level of detail wanted. As each particle flow down the chamber, energy between gas and the particles are exchanged to an equilibrium. However, as the moisture evaporates from the particle, the temperature dynamics is altered. The gas flow's ability to store moisture is dependent on the gas' ability to maintain a humidity difference, as a function of the gas temperature profile. We could also model the particle relative velocity, expressed through the *Reynolds number*, to account for the evaporation from the particle surface due to the airstream velocity. Depending on chamber design, one could easily assume that the

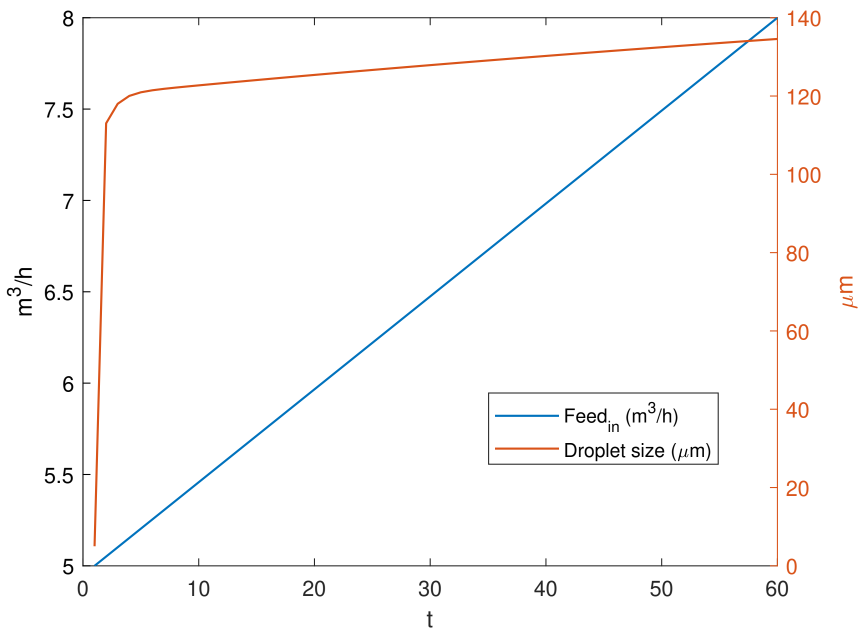


Figure 3.5: Droplet size when ramping up feed rate $\dot{Q}$ from 5 to 8

particle evaporation time is not constant for all fractions, as air turbulence would cause some particles to re-enter the main flow, and also temporary deposit on the walls.

To approximate our model, we make the following assumptions, using the droplet size model as input:

- The droplet is fully heated before evaporation starts
- Evaporation is linear, as a function of droplet size
- All particles are perfectly dried, e.g. powderized
- Droplet mass is homogenous and relates to the volume and density
- The droplet is dried in atmospheric pressure

Therefore, we create a very simplified evaporation model for our use case, adapted from the classical constant droplet temperature model (d^2-law) stating that the squared value of the droplet diameter decreases linearly with respect to time. We state that

$$E_{droplet} = \gamma \frac{T_{droplet}}{A_{droplet} \times \Delta h} \tag{3.13}$$

where E is evaporation, T is the droplet temperature, A is the droplet area and h is the absolute droplet humidity ($grams/mm^3$), empirically adjusted by the γ parameter. The model calculates droplet relative humidity. Our model nowhere justifies the amount of work in the field over the years and is just implemented as a reference for our agent learning.

Another factor is the *glass transition temperature* T_g and the problem of *sitckyness*, where the material becomes sticky and deposits on the chamber surfaces [27]. Our model does not take T_g into account.

The model yields the simulation of relative humidity in the end product as shown in Figure 3.6. The feed rate was ramped from 5 to 8 m^3/h while keeping the controlled variables fixed.

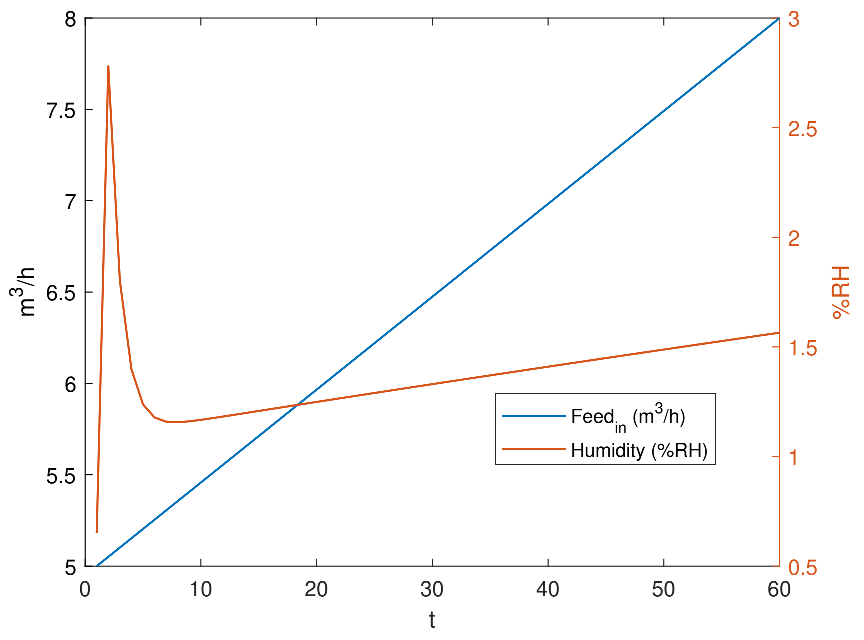


Figure 3.6: Droplet size when ramping up feed rate $\dot{Q}$ from 5 to 8

Our immediate evaluation is that the humidity increases as the feed rate is increased, a relationship described by Santos et al. [88] and well preserved in our model.

The model is a very coarse representation of a spray drier and is made for demonstration of the reinforcement learning concept. It does not capture many of the necessary components of spray drying physics, as briefly discussed above. As a general model that serves our purpose of the reinforcement learning agent, the level of detail should be sufficient.

Complete code listing for the spray drier ODE model is in Appendix A.1.

3.3.4 Historical logs

The effort of making a model needs some explanation. First, in order to explore *Autonomous Learning*, a lot of process data is needed as we enter the field of Sequence-to-Sequence modelling with generative networks. It might be tempting to use data from an existing process plant, but for the sake of explainability of the agent and for developing the concept, synthetic data will be used for training and testing different model scenarios.

Later, real-world applications of the model will need careful data preparation and access to a long history of sequential process operation using a quantifiable method. In such a scenario, the historical dynamics of the process operation will be directly learned by the generative model and locked to one historic operational scheme. A synthetic model enables us to experiment with several process control strategies and vary the setup of manipulated and controlled variables.

The model behaves like the physical operation of the central parts of the spray drier. A rollout framework is made, and the model is encapsulated in a class derived from `matlab.io.Datastore` for convenient usage when training the generative network. The `DataStore` presents `Sequence`s of data to a compatible `DataStore` user, i.e. during training of a memory network in *Matlab*, as an alternative to raw disc storage.

3.4 Autonomous Modelling

The hypothesis of this report is whether logs of historical data can be used to train an RL agent for optimization towards more complex feedback schemes, taking into account other and more *sparse* parameters than just sensor data. The idea is that the recorded historical logs will contain enough description of the dynamics in order to properly make a detailed model. The introduced use case model will serve as a basis for generating such logs as synthetic runs of a fictive spray drier. Later, an agent is trained on the historical logs and evaluated by running a control scheme towards the real model.

The next section will present an implementation of such exemplified through the presented spray drier use case.

Chapter 4

Autonomous Learning of Core Skills

In God we trust, all others bring data.[1]

4.1 Introduction

Central to training an RL agent is establishing *the environment* and a *reward function* for iterative and exhaustive learning. As presented earlier, a traditional approach would be to mathematically model the *environment* using game engines or 3D engines, as presented by OpenAI's Solving Rubik's cube project [99], Dota 2 competition [98] and DQN-based gameplay [60], amongst others. Time series are data, logs or process signals that are related in time and as such fit as input to a *Markov decision* chain concept, which is relevant for our research.

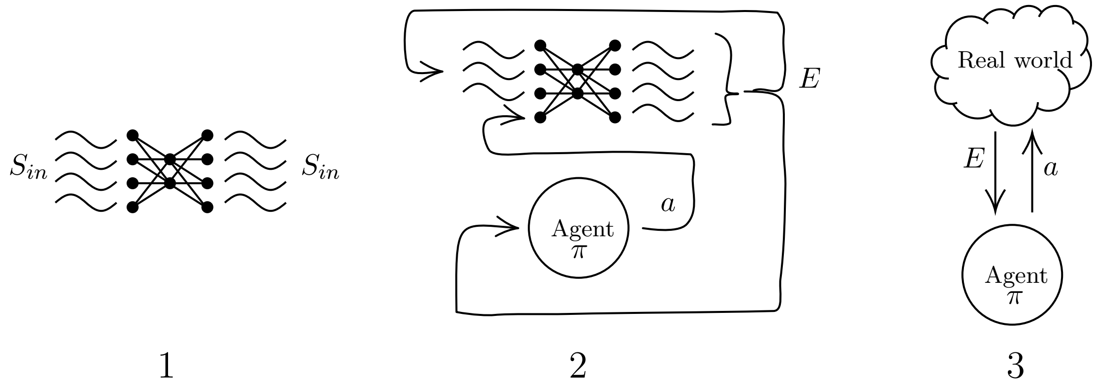


Figure 4.1: Concept for Autonomous Learning of Core Skills
1. Train a Generative Process Dynamics Model on historical data log sequences S_{in}. Skills are stored in latent space of a generative network. 2. Train an RL agent policy π using the generative model as the environment E applying action a with the model E. 3. Execute non-linear control in real world proposing action a based on policy π and observation input E.

[1]American engineer and statistician William Edwards Deming (1900-1993)

Using the ideas of multidimensional latent space models [80], [113], we utilize generative models to passively learn system dynamics. We define a *Generative Process Dynamics Model* that is trained for representing the historical data logs, from this point on called *sequences*. The training is unsupervised, that is, a generative network is presented with many sequences and stores latent space vectors in the network. After training, we can sequentially query the network and change the input parameters according to the action output of the RL agent. We are then able to estimate the next step in the sequence after any given action. The RL agent would apply the proper reward- and explore/exploit-scheme for action proposal. The result is an agent that can be trained on historical logs and execute dynamic non-linear control in a real-time scenario exhibiting the same dynamics as described in the logs, a concept visualised in Figure 4.1. Of course, a prerequisite is that the logs exhibit relevant system dynamics.

For us to explore this idea, the work is divided into the tasks outlined in Table 4.1. The work is implemented using *Matlab* and several add-ons: Statistics and Machine Learning, Deep Learning, Reinforcement Learning and Parallel computing toolboxes. *Matlab* is also used for all data engineering and statistics tasks.

All experiments are executed *in silico* by using *Matlab*-simulation and the *Reinforcement Learning Toolbox*.

Define first principle model	Build a dynamic model of an example process, namely a level of the spray drier process as described in Chapter 3.
Generate synthetic process data logs	Using the dynamic model, create a large amount of fictious run data applying traditional control schemes for univariate and multivariate scenarios.
Establish a predictive baseline	Evaluate the power of traditional control schemes.
Train a Generative Model	Train off-line using the process data logs.
Train a Reinforcement Agent	Train using the Generative Model as the Environment. Actions are applied to the Generative Model and functions as the Observation space. Explore several agent types.
Evaluate Agent towards model	Use the first principle model designed in the Use Case chapter as a real time environment for control evaluation.
Approximate a reward function	By transfer learning or according to optimization. Evaluate performance of sparse feedback in continuous processes presented in the use case.

Table 4.1: Implementation tasks

4.2 Generating synthetic data

The main part of this chapter will focus on generating data from the spray drier first principle model as presented in Chapter 3 with the goal of establishing a baseline for comparison with a reinforcement learning agent. Data could be fetched from disc files as historical logs, but we choose to generate synthetic data in real-time, to save maintenance task time, as well as having a short turn-around-cycle for testing new ideas. This means that we solve the ordinary differential equations when data from the logs are requested, in real-time.

The real-time rollout mechanism is implemented as class `RolloutDatastore` descending from the `matlab.io.Datastore` class, which makes it compatible with the Deep Learning toolboxes in Matlab. The class will roll out data using the control scheme and generate sequences that can be used for neural network training.

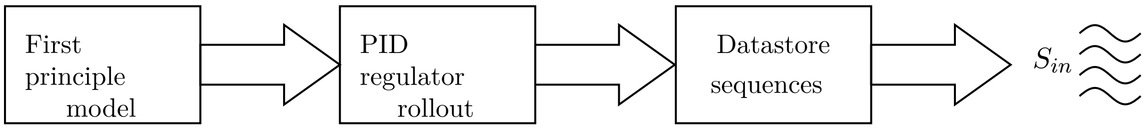


Figure 4.2: Architecture of the `RolloutDatastore` class

The rollout algorithm is made with two scenarios in mind:

- Univare control of output temperature using PID
- Multivariate control of output temperature and particle size

The first scenario is used for evaluating the generative model concept and for training univariate RL agents. The second scenario is used for investigating the performance of an RL agent trained by a multivariate generative model. As such, the system boundary will be adjusted to be multivariate, that is, the manipulative variable set includes more than one signal, as shown in Figure 4.3.

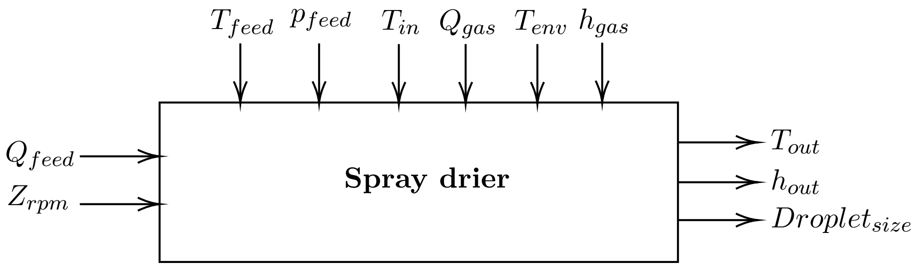


Figure 4.3: System boundary for a multivariate model

We choose the Z_{rpm} signal as the second manipulative variable as this could cater for some interesting challenges. Droplet size calculation is dependent on both Q_{feed} and Z_{rpm} and could be an example of a *conflicting control strategy*, where an optimum in both signal paths is not easily achievable. Balancing adversarial optimums by a trained reward model is of particular interest in fields involving human decision making.

4.2.1 Connecting to the *Environment*

The `RolloutDatastore` is used for training the generative model. To accomplish this, each *rollout* of size n is broken into several *sequences* according to a defined *sequence length* m. Figure 4.4 demonstrates this process and the relations. The produced set of *sequences* is denoted S_{in} as they are used for input to the generative model learning process.

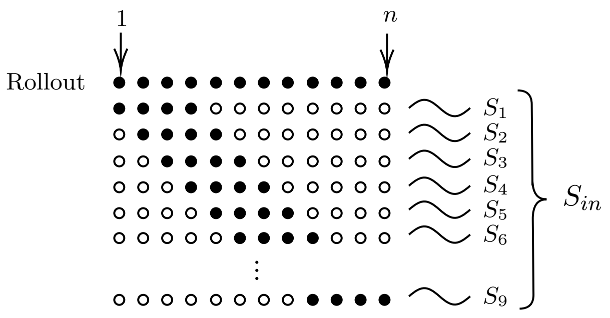


Figure 4.4: Generating *sequences* S_{in} from a *rollout*

For each rollout of size n, a sliding window starting at the first position is glided over the rollout. For each step, a sequence of size m is produced. The process is similar if one reads a historical data log from a disc file, however in this case the synthetic first principle model is used as input instead. In a file-based scenario, a continuous-time series with no time gaps qualifies as one *rollout*. One could also use any historian data collector with querying capability as input.

The synthetic rollout does not take into account all faults that can happen in a real process. Normally, one would need to pre-process any historical data to ensure that all input values are valid, which might not always be the case in a real process. Instrument failures, communication failures and calibration errors are common in industrial processes and will influence the data quality. Further, several types of exceptions occur that do not contribute to explaining the overall process dynamics. The focus is on synthetically produced data where we have full control of the data quality to explore the control concept, and not on engineering a data cleaning strategy. However, to implement these ideas into real life, the focus must turn into data quality.

4.2.2 Predictive baseline

Our Matlab-model for generating data is flexible in terms of control scheme, but we will follow the system boundary as presented in Figure 3.2 for the univariate case. Here, feed flow Q_{feed} is the manipulated variable and we select T_{out} responding variable as the feedback. An overview of the system signals is given in Table 4.2.

By using T_{out} as a error feedback variable, we can define a setpoint T_{sp} and use a simple PID-regulator for control of Q_{feed} by calculating $\dot{Q}$, only using the propotional clause. We define the regulator as:

$$\dot{Q}_t = K_p(T_{sp} - T_{t-1}) \tag{4.1}$$

where the gain coefficient K_p is empirically tuned to 0.1. We can also call this the *damping factor* of the process as responses to system dynamics is smoothed by this factor.

Manipulated variables	Nominal value or range	Unit
Q_{feed}	2-15	m^3/h
Controlled variables		
T_{feed}	95	$°C$
p_{feed}	1200	kg/m^3
Z_{rpm}	7,000-12,000	rpm
T_{in}	180	$°C$
Q_{gas}	100,000	m^3/h
T_{env}	25	$°C$
h_{gas}	50-99	%
Responding variables		
T_{out}	90-100	$°C$
h_{out}	1-5	%
$Droplet_{size}$	90-120	μm

Table 4.2: System signals, nominal values and units

By applying the regulator on the spray drier model, the system response is a clean signal as shown in Figure 4.5.

We observe in Figure 4.5 that the temperature response is a stable signal on $t > 10$ when $K_p = 0.1$. The controlled variables are fixed in this scenario, contributing to an effective regulation by using PID with only the K_p term.

4.2.3 Process noise

However, the controlled variables in a real-world process are not noise-free due to e.g. sensor drifts and deviation, changes in the environment and material, as well as external regulators that affect our system, amongst many other sources. The main noise source we consider is the sensor standard deviations. This noise would be an ideal source for randomness in training the policy of an RL agent. By introducing random disturbance to our controlled variables and our feedback source, the T_{out} responding variable, our model should behave closer to a real process.

We add random disturbance $\pm n\%$ the standard deviation, a method proposed by Del Rio-Chanona et al. [91] and record the mean and standard deviation for some combinations of n and K_p. Table 4.3 lists the results and an example is given in Figure 4.6.

Variables		Results	
n	K_p	μ	σ
1%	0.1	100.0	1.5
1%	0.01	101.2	5.4
3%	0.1	100.0	4.5
3%	0.01	100.9	5.6
5%	0.1	100.0	7.3
5%	0.01	101.3	8.6

Table 4.3: Results of applying different noise levels and tuning of K_p

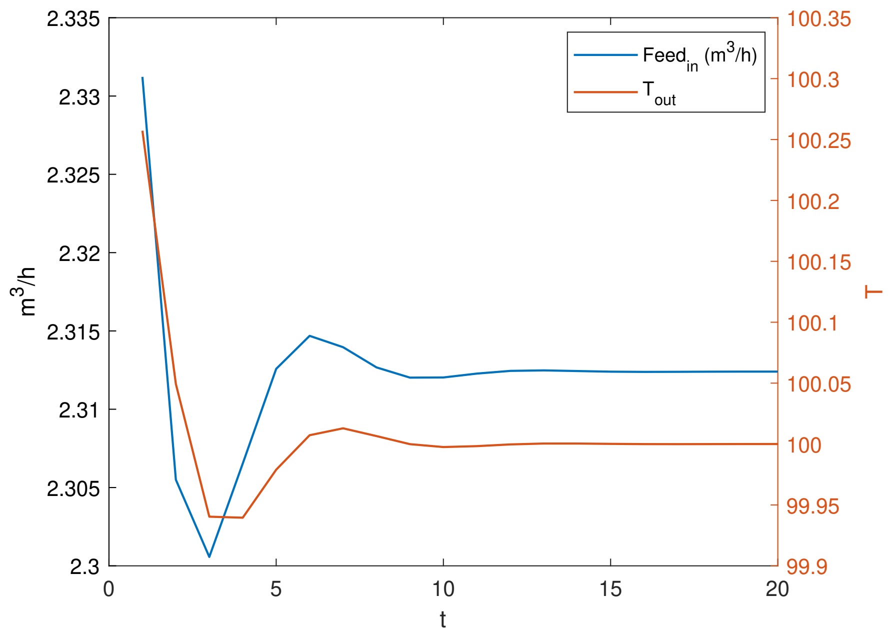


Figure 4.5: System response using a propotional regulator

We can argue that a controlled process may have a noise level at $\pm 1\%$, or even less, but that depends on the instrumentation of the process. The noise level can be estimated or calculated based on available instrument data and documentation. Using neural networks in an RL agent might be challenging if the operational window in the data logs is narrow and the sample space is low. Neural networks do not extrapolate well outside its generalized data space, and are sample inefficient during training. As such, we can argue that the rollout mechanism for generative model training should utilize parameters that uses a higher noise setting than $\pm 1\%$. This is also the reasoning behind utilizing only a proportional regulator, as this might introduce regulation of the system where system dynamics is visible. By adding both integral- and derivative steps to the PID, and tuning the regulator perfectly to the process, one could probably obtain much finer control over the temperature profile. However, this is not the goal of the rollout mechanism - we want to assure that the historical logs exhibit the system dynamics, which is the priority.

Randomizing the noise parameter and selecting different set points for T_{out} could be done for each rollout to capture a larger parameter window. We could also measure the system response according to the same static or variable input.

4.2.4 Randomized input

A randomized Q_{feed} in the range 3-20 m^3/h is input into the model with a varying frequency of every t. As anticipated, the process is not under control via this scheme with a $\mu = 75.3$ and $\sigma = 30.6$, illustrated in Figure 4.7. This gives a coefficient of variance of $C_v = 0.41$, the ratio of the standard deviation to the mean, which we can argue should be below 0.10 or less. Of course, other results are obtained for each simulation run.

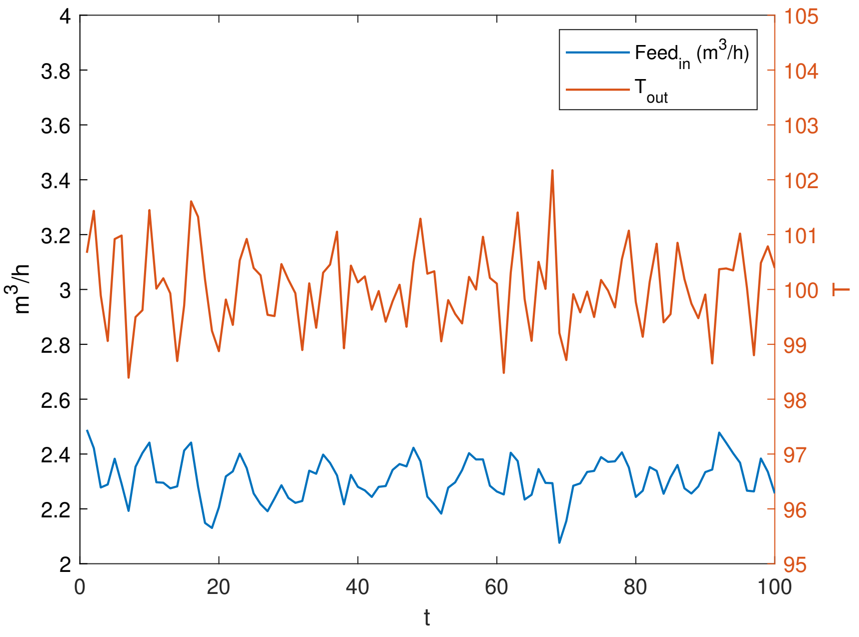


Figure 4.6: System response using a propotional regulator and added system noise, ±1% of σ using $K_p = 0.1$

Result is $\mu = 100.0$ and $\sigma = 1.5$.

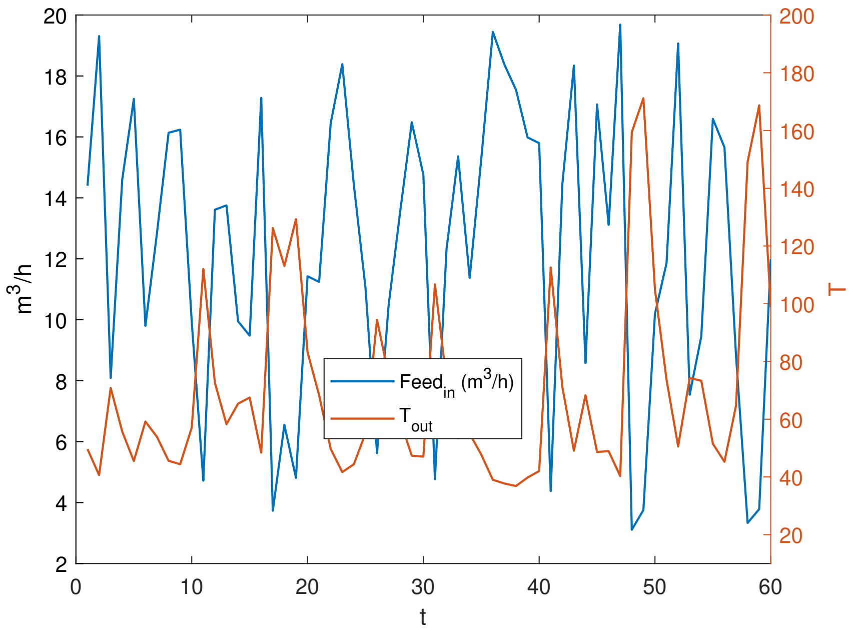


Figure 4.7: System response using a random input if Q_{feed} in the range 3-20

Result is $\mu = 75.3$ and $\sigma = 30.6$ for this run.

It is clear that a randomized control input is not ideal to reveal system dynamics in the form of controlled sequences.

4.2.5 Sinusoidal input

Varying one manipulated variable at a time and recording the system response is a traditional method for identifying a system in control theory. A noisy sinusoidal Q_{feed} is input into the model, and the system response is visualised in Figure 4.8. We see that the response is a bit off-phase from the input with an $R^2 = 0.55$, a property of a cooling effect when increasing the feed of the spray drier, as noted by Santos et al. [88].

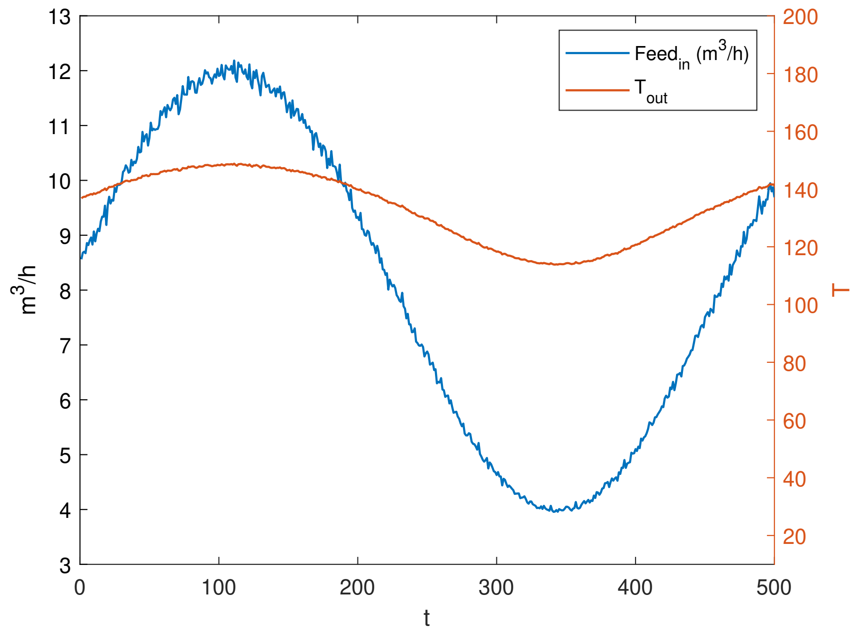


Figure 4.8: System response using a sinusoidal input of Q_{feed}.

Creating rollouts using sinusoidal inputs could be one way of generating data. Converting this to the real world, a proper *Design-of-Experiments* could be conducted to record system response to varying input, in the case, a process is not under control yet and one need to create an initial set of data logs for the system.

4.2.6 Randomized step-wise input

A randomized step-wise input strategy changes each manipulative parameter at random time t, where we observe the dynamics response to the change by recording the responding variables. The change could either be random or a fixed rate from a baseline. The benefit of this explorative strategy is the ease of implementation and clear dynamics response.

4.2.7 Data augmentation

Using the rollout mechanism connected to the first principle model, we can produce many unique runs because of the randomness introduced through process noise that is added to the signals, the data augmentation, and the signal input strategy. We can therefore assume that each sequence being generated holds the property of being more or less unique. Importing data from an existing real-world process could utilize the same method when the sample space is scarce, adding noise to each rollout, producing many unique sequences.

4.3 Design of generative model

The generative model captures the dynamics of the data logs. The idea is that this model can serve as the environment simulator for the RL agent and be able to generate any process response of an action proposed by the agent.

We can extend the problem formulation to finding a function f that minimizes the error of estimating the next step of a sequence as opposed to the actual sequence:

$$\begin{aligned} &\underset{f}{min}\ \mathcal{L}(f) \\ &\mathcal{L}(f) \ = \sum_{t=t_0}^{N} L(\hat{y}_t, y_t) \\ &\text{where} \\ &x_{t+1} = y_t \\ &\hat{y}_t = f(x_t) \end{aligned} \tag{4.2}$$

Depending on the agent's exploration scheme, the generative model can perform unsatisfactory outside its trained data space. When confronted with an action that violates earlier knowledge, as a random action that the network cannot understand the consequence of, the network might confuse the agent more than being an informative colleague. We expect that the choice of the RL agent algorithm must be in line with this prerequisite. A soft actor-critic utilizing maximum entropy as the exploration scheme is probably a good candidate, as this agent type is shown to perform well even on a low number of samples [81], but still explore the environment as randomly as possible. This claim will be challenged as we choose to train both a soft actor-critic (SAC) agent and a selection of other agent types.

A benefit of training a generative model from process data is the ability to produce many unique sample sequences for RL training. RL training is sample intensive and requires many training runs on unique data sets. By encoding the process data in a *memory network* by using long short-term memory (LSTM) blocks in a *shallow autoencoder architecture*, we capture both the dynamics and signal relationships in function f. The architecture is visualised in Figure 4.9. Querying the network takes into account the *state* of the memory network, and as such, producing data would differ for each RL training pass, given the randomness introduced in RL exploration. The generative model has a *dropoutlayer* for the benefit of intensifying this randomness and provide regularization of the model.

Discovering the latent variables is done by unsupervised training of the memory network. Our `RolloutDatastore` produces data for this purpose, and we add another feature to this class for *time series forecasting.* Traditionally, in unsupervised training of an autoencoder,

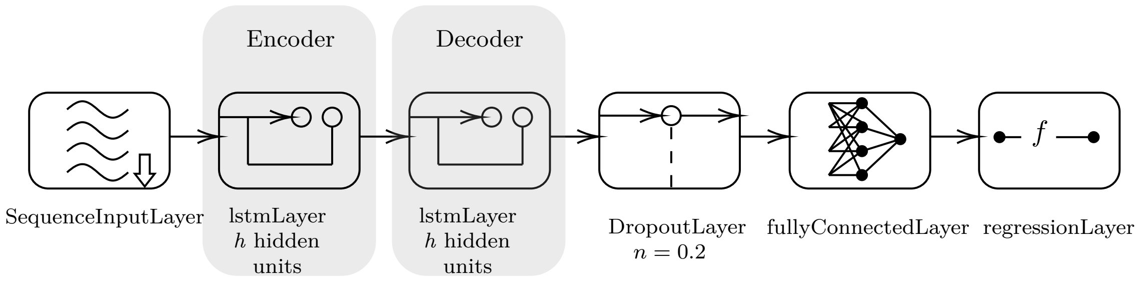


Figure 4.9: Generative model LSTM-AE neural network architecture. h is estimated by Bayesian optimization.

the input sequence X is used as output sequence Y where $X = Y$, to train the latent space. In our case, we shift the output sequence *by one time step* to obtain forecasting abilities:

$$X(x_t^k, x_t^{k+1}, \ \dots\ , \ x_t^{k+m}) = Y(x_{t+1}^k, x_{t+1}^{k+1}, \ \dots\ , \ x_{t+1}^{k+m}) \tag{4.3}$$

where m is the sequence length and x^k is the kth vector of the sequence, where $t+1$ indicates a shift by one time step.

This important addition means that we can input an observation vector into the generative model and receive an *estimate* on the next vector of a similar sequence from history when inferring from the network. This forecasting ability is made possible by the autoencoder mechanism and the memory ability of the recurrent neural network, implemented through a chain of LSTM network blocks. Training is done by shifting the data by one step as described in Equation 4.3. According to the problem formulation in Equation 4.2, the neural network should learn by minimizing training loss. Evaluating the loss and prediction RMSE should explain the forecasting ability.

4.3.1 Training and evaluating the Generative Process Dynamics Model

The class `GenerativeModel` is a tool for model training using the `RolloutDatastore` class and tools from the *Matlab Deep Learning Toolbox*, as well as loading and saving trained models to disk. The class also handles the normalization and de-normalization of data. Normalizing data input is recommended for all machine learning methods, especially when dealing with features with different magnitude of numeric ranges.

The `RolloutDatastore` class will create a baseline for normalisation parameters by creating n number of rollouts when first being initialized. The μ and σ is calculated and stored for global standardisation of all generated datasets, and also for de-standardisation to each feature's numeric range.

To estimate relevant dynamics for all responding variables and create the functional approximation of the problem formulation in Equation 4.2, we design an experiment where the T_{sp} for the regulator is randomized in the range 88-92 $°C$ for each rollout every 60th time step. The generative network is then presented with n number of rollouts. As earlier mentioned, this is the point where historical logs from the process are input to the model, but for experimental efficiency, we use the differential equations as presented in the Use Case chapter implemented in the `RolloutDatastore` class.

Hyperparameter optimization

Selecting the proper hyperparameters may greatly impact the performance of a neural network model. We make educated guesses of many hyperparameters for training the generative model, however parameters for capturing relevant process dynamics need some consideration. We identify the following hyperparameters that are relevant to explore:

- Sequence vector length
- Rollouts per epoch
- Number of hidden units in the generative model
- Minibatch size

Minibatch size is a technical way to iterate on smaller parts of the dataset per epoch before doing backpropagation for all of them. The minibatch size can influence the RMSE but is mostly tuned for computational reasons [47].

Due to the immense computing power needed to test all combinations of hyperparameters, we choose to use *Bayesian optimization* as an alternative to exhaustive hyperparameter search. *Bayesian optimization* selects new parameters based on a selection from a Gaussian distribution of each hyperparameter. The theory is that adaption of the parameters is done based on their calculated impact on the response.

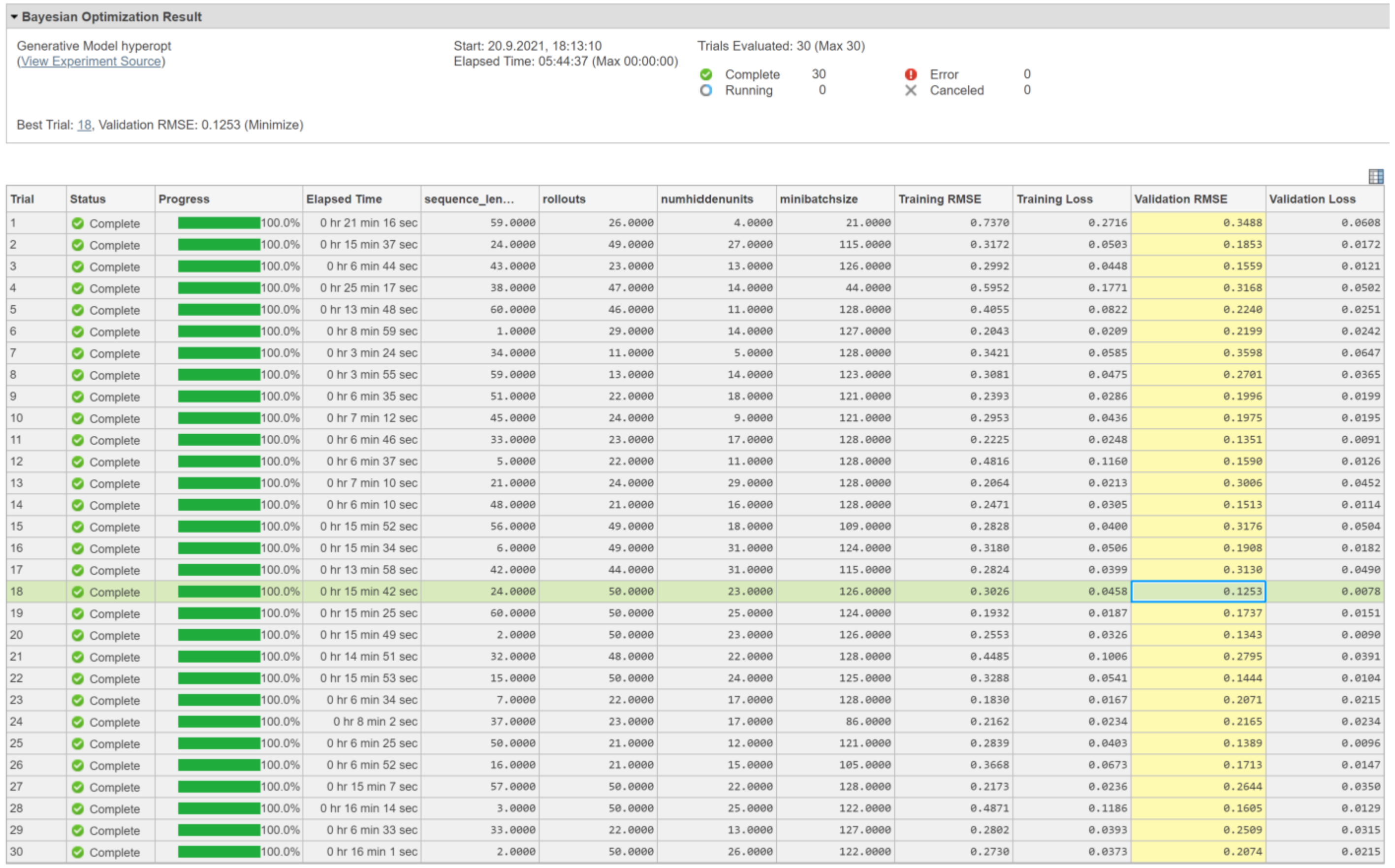
Further, we train the generative model for only 100 epochs per hyperparameter search run. There are some implications to this which need some consideration. First, the number of epochs needed to train a model is greatly influenced by the hyperparameters. One obvious relation is the number of hidden units: a high number might need several magnitudes of more training iterations. We restrict here the number of iterations. Second, due to this restriction, we should not use the found hyperparameters directly. We manually estimate the *direction* of the selection of hyperparameters and make an initial set of efficient parameters to use for *one single final run*. This assumption is based on the fact that Bayesian optimization will iterate on the results and narrow the parameter search based on the Gaussian state. The final run will be set up with more epochs and potentially run for a longer period.

Matlab Experiment Manager is used for establishing 30 hyperparameter search runs using Bayesian optimization. The parameters are constrained to ranges that we believe are relevant to evaluate the parameter trends (see Table 4.4).

Parameter	Range
sequence length	1 - 60
rollouts	10 - 50
numhiddenunits	4 - 32
minibatchsize	8 - 128

Table 4.4: Ranges for hyperparameters for generative model

A screenshot of this session can be seen in Figure 4.10.



▾ Bayesian Optimization Result

Generative Model hyperopt
(View Experiment Source)

Start: 20.9.2021, 18:13:10
Elapsed Time: 05:44:37 (Max 00:00:00)

Trials Evaluated: 30 (Max 30)

Complete	30	Error	0
Running	0	Canceled	0

Best Trial: 18, Validation RMSE: 0.1253 (Minimize)

Trial	Status	Progress	Elapsed Time	sequence_len...	rollouts	numhiddenunits	minibatchsize	Training RMSE	Training Loss	Validation RMSE	Validation Loss
1	Complete	100.0%	0 hr 21 min 16 sec	59.0000	26.0000	4.0000	21.0000	0.7370	0.2716	0.3488	0.0608
2	Complete	100.0%	0 hr 15 min 37 sec	24.0000	49.0000	27.0000	115.0000	0.3172	0.0503	0.1853	0.0172
3	Complete	100.0%	0 hr 6 min 44 sec	43.0000	23.0000	13.0000	126.0000	0.2992	0.0448	0.1559	0.0121
4	Complete	100.0%	0 hr 25 min 17 sec	38.0000	47.0000	14.0000	44.0000	0.5952	0.1771	0.3168	0.0502
5	Complete	100.0%	0 hr 13 min 48 sec	60.0000	46.0000	11.0000	128.0000	0.4055	0.0822	0.2240	0.0251
6	Complete	100.0%	0 hr 8 min 59 sec	1.0000	29.0000	14.0000	127.0000	0.2043	0.0209	0.2199	0.0242
7	Complete	100.0%	0 hr 3 min 24 sec	34.0000	11.0000	5.0000	128.0000	0.3421	0.0585	0.3598	0.0647
8	Complete	100.0%	0 hr 3 min 55 sec	59.0000	13.0000	14.0000	123.0000	0.3081	0.0475	0.2701	0.0365
9	Complete	100.0%	0 hr 6 min 35 sec	51.0000	22.0000	18.0000	121.0000	0.2393	0.0286	0.1996	0.0199
10	Complete	100.0%	0 hr 7 min 12 sec	45.0000	24.0000	9.0000	121.0000	0.2953	0.0436	0.1975	0.0195
11	Complete	100.0%	0 hr 6 min 46 sec	33.0000	23.0000	17.0000	128.0000	0.2225	0.0248	0.1351	0.0091
12	Complete	100.0%	0 hr 6 min 37 sec	5.0000	22.0000	11.0000	128.0000	0.4816	0.1160	0.1590	0.0126
13	Complete	100.0%	0 hr 7 min 10 sec	21.0000	24.0000	29.0000	128.0000	0.2064	0.0213	0.3006	0.0452
14	Complete	100.0%	0 hr 6 min 10 sec	48.0000	21.0000	16.0000	128.0000	0.2471	0.0305	0.1513	0.0114
15	Complete	100.0%	0 hr 15 min 52 sec	56.0000	49.0000	18.0000	109.0000	0.2828	0.0400	0.3176	0.0504
16	Complete	100.0%	0 hr 15 min 34 sec	6.0000	49.0000	31.0000	124.0000	0.3180	0.0506	0.1908	0.0182
17	Complete	100.0%	0 hr 13 min 58 sec	42.0000	44.0000	31.0000	115.0000	0.2824	0.0399	0.3130	0.0490
18	Complete	100.0%	0 hr 15 min 42 sec	24.0000	50.0000	23.0000	126.0000	0.3026	0.0458	0.1253	0.0078
19	Complete	100.0%	0 hr 15 min 25 sec	60.0000	50.0000	25.0000	124.0000	0.1932	0.0187	0.1737	0.0151
20	Complete	100.0%	0 hr 15 min 49 sec	2.0000	50.0000	23.0000	126.0000	0.2553	0.0326	0.1343	0.0090
21	Complete	100.0%	0 hr 14 min 51 sec	32.0000	48.0000	22.0000	128.0000	0.4485	0.1006	0.2795	0.0391
22	Complete	100.0%	0 hr 15 min 53 sec	15.0000	50.0000	24.0000	125.0000	0.3288	0.0541	0.1444	0.0104
23	Complete	100.0%	0 hr 6 min 34 sec	7.0000	22.0000	17.0000	128.0000	0.1830	0.0167	0.2071	0.0215
24	Complete	100.0%	0 hr 8 min 2 sec	37.0000	23.0000	17.0000	86.0000	0.2162	0.0234	0.2165	0.0234
25	Complete	100.0%	0 hr 6 min 25 sec	50.0000	21.0000	12.0000	121.0000	0.2839	0.0403	0.1389	0.0096
26	Complete	100.0%	0 hr 6 min 52 sec	16.0000	21.0000	15.0000	105.0000	0.3668	0.0673	0.1713	0.0147
27	Complete	100.0%	0 hr 15 min 7 sec	57.0000	50.0000	22.0000	128.0000	0.2173	0.0236	0.2644	0.0350
28	Complete	100.0%	0 hr 16 min 14 sec	3.0000	50.0000	25.0000	122.0000	0.4871	0.1186	0.1605	0.0129
29	Complete	100.0%	0 hr 6 min 33 sec	33.0000	22.0000	13.0000	127.0000	0.2802	0.0393	0.2509	0.0315
30	Complete	100.0%	0 hr 16 min 1 sec	2.0000	50.0000	26.0000	122.0000	0.2730	0.0373	0.2074	0.0215

Figure 4.10: Matlab Experiment Manager for Bayesian optimization of generative model hyperparameters

The parameter search results and RMSE results from the *Experiment Manager* has then been analysed to select a final parameter set. First, the results from Figure 4.10 is entered into Matlab and we smooth the data using a 10-point window moving average as a pre-treatment technique. This is done to detect trends in the data and remove some of the stochasticity of the optimization process results. The sequence plot in Figure 4.11 visualises the optimization process and we can see that the RMSE is decreasing as a function of time.

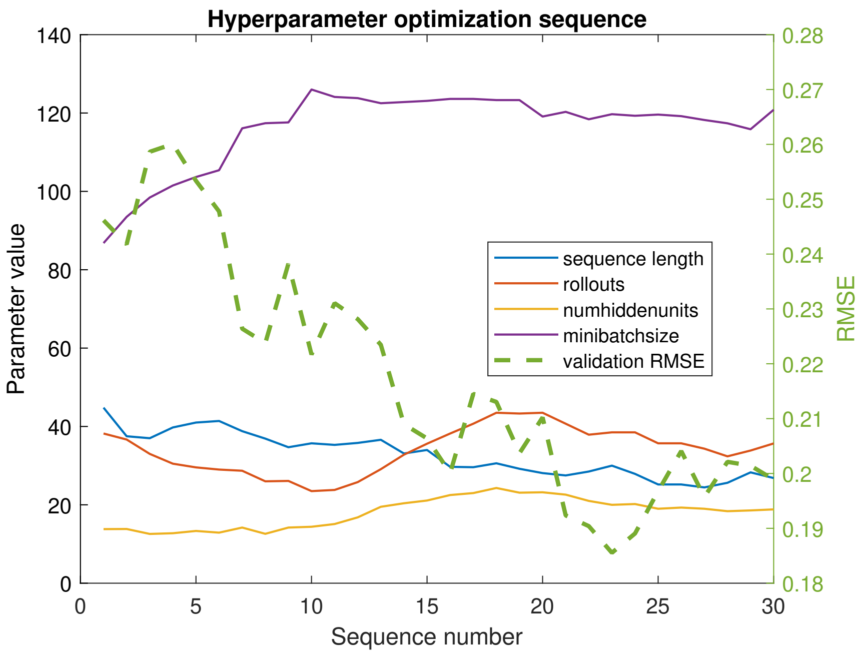


Figure 4.11: Bayesian hyperparameter optimization sequence results from Matlab Experiment Manager

We then run a 2-component *Principal Component Analysis* and obtain the plot in Figure 4.12. We see that the varying components that result in a low RMSE are relatively clustered around a plane that might resemble the optimization sequence. From this, we can infer that the parameter selection is not random and that it should be possible to converge on parameters that affect the RMSE and that we might not need more iterations of the hyperparameter search to conclude on a final set.

For our final selection of parameters, we scatterplot all pre-treated parameters towards the validation RMSE and obtain the results in Figure 4.13.

The trends show us the direction of the parameters, and we deduct the following:

- The rollout parameter has less impact on the RMSE but should be higher than 40.
- The minibatch size should be set high. There is a cluster around high values and should be set larger than 120.
- The number of hidden units in the generative model could be set to more than 20.
- The sequence length should be 25 or less.

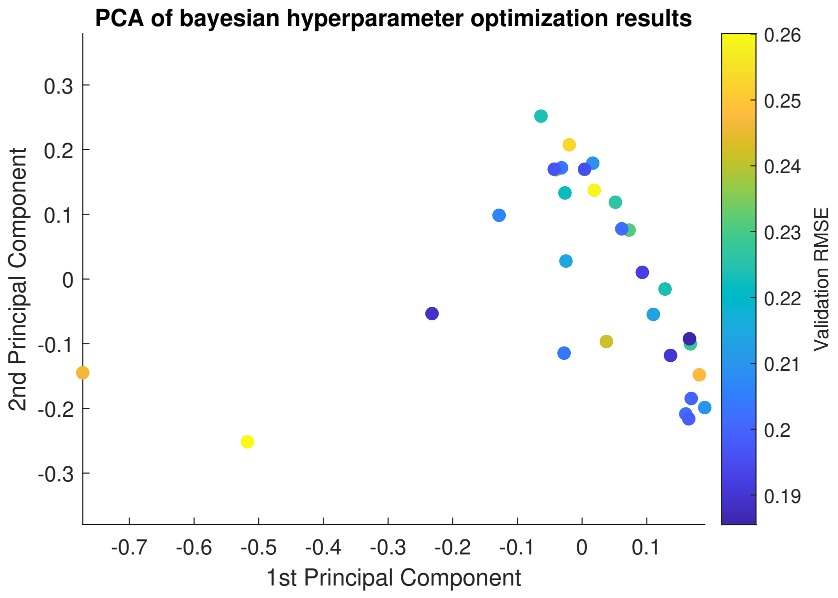


Figure 4.12: PCA of Bayesian hyperparameter optimization sequence results

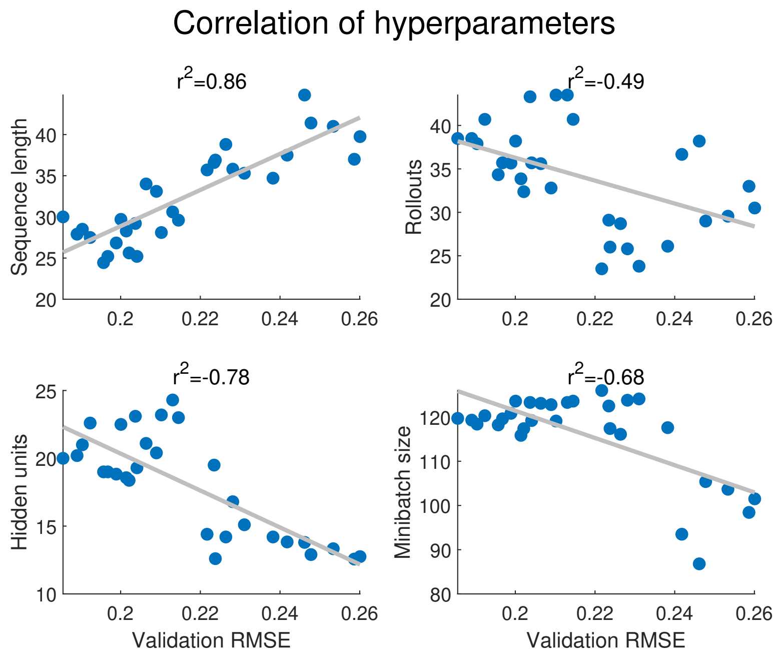


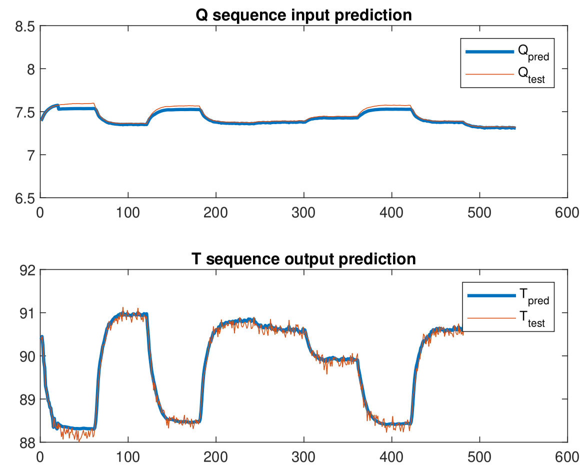
Figure 4.13: Correlation of hyperparameters twoards validation RMSE

As such, we can decide on parameters taking the available dataset and computational resources into account.

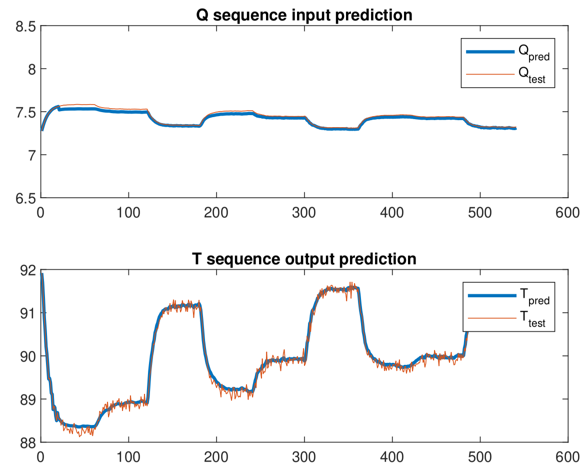
Training a univariate model

The network is initially trained on 300 epochs with the *Adam* optimizer using 60 unique rollouts per epoch, resulting in a training session equivalent to 18,000 unique rollouts. Each rollout is then presented to the network using the skewing mechanism with a sequence length of 20 time steps resulting in a magnitude of increase in data input. The minibatch size is set to 128 and the number of hidden units is set to 32. Further, we step down the learning rate from an initial setting of $lr = 0.05$ by a factor of 0.2 for every 50th epoch. The `RolloutDatastore` is also used for validation using another set of randomized rollouts and the same standardisation settings.

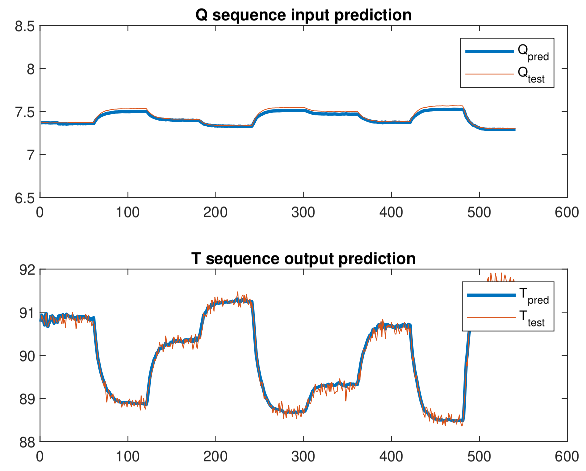
The training session is done using Matlab running the Parallel Computing toolbox on a dedicated NVIDIA Quadro T1000 GPU. We evaluate the generative model by letting the model predict the next step in a sequence, given 20 steps of data from a random rollout. Based on these steps, the next step should be estimated. The RMSE of the training, and the evaluation, should be as low as possible. Low numbers close to 0 means that the model has predictive ability and that the network has converged to the dynamic model. This is an important discovery, as a higher RMSE will not enable predictive power of the generative model. Figure 4.14 show four random runs and their results.



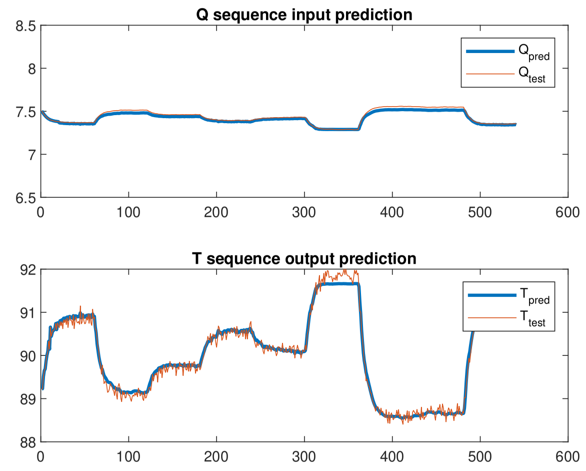
(a) MAE 0.04, RMSE 0.09, r^2=1.00



(b) MAE 0.03, RMSE 0.09, r^2=1.00



(c) MAE 0.02, RMSE 0.09, r^2=1.00



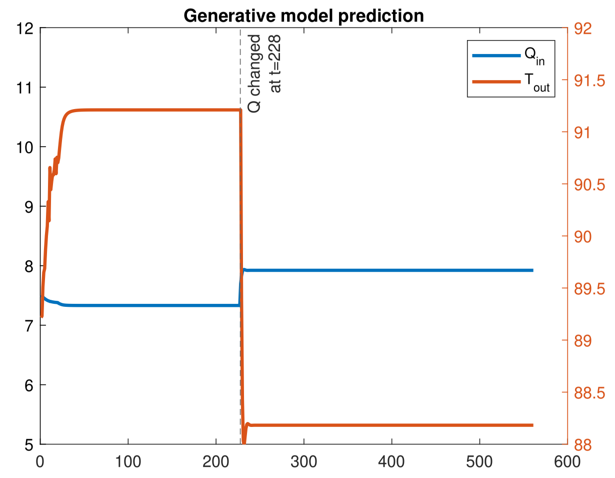
(d) MAE 0.01, RMSE 0.11, r^2=0.99

Figure 4.14: Results of querying the generative model for next steps of random test runs

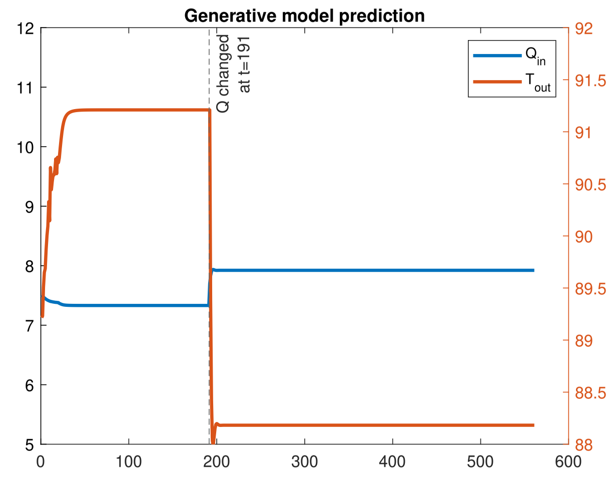
Simulating sequences

The input to the prediction is the previous sequence from the random rollout, so to test the predictive ability, we input *skewed estimated sequences* from the generative model itself instead of the random rollout. We need to start with an initial sequence from the rollout and add the model's output to a historical sequence. These outputs will then be used as an input to the next prediction step. This process will resemble the RL agent training mechanism explained later. We choose a randomly step-wise change of the manipulative variable Q and record the dynamic response from the model. Figure 4.15 show four random runs and their results, where the initial Q is changed by 5% at some random t.

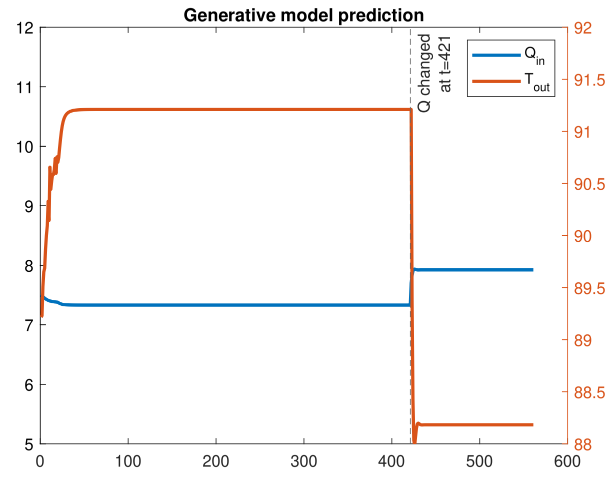
We see that the model responds by decreased temperature when the flow rate is increased, which corresponds to our earlier beliefs of the physical process. This method of replacing values during sequence prediction will be our method of simulation. Simulation is done by *querying the environment* using the current *state* as input, and a proposed *action*.



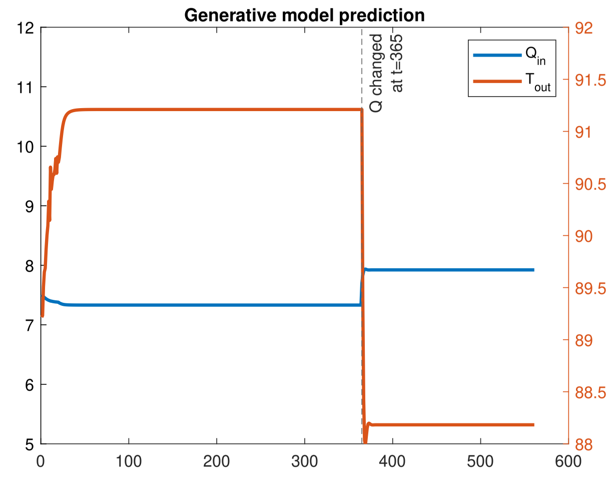
(a) Random run 1, Q changed at $t = 228$



(b) Random run 2, Q changed at $t = 191$



(c) Random run 3, Q changed at $t = 241$



(d) Random run 4, Q changed at $t = 365$

Figure 4.15: Results of querying the generative model for next steps of random test runs

4.3.2 Querying the *Environment*

A feature of our `GenerativeModel` class is to act as input to the *Environment* for RL agents created in Matlab, querying the trained generative model. As such, we also create a class `SyntheticEnvironment` descendant from class `rl.env.MATLABEnvironment`. This class will receive *step*-signals from the RL agent where the Agent's action proposal is a parameter, based on previous calls to the *step*-function. The class responds with an *Observation* from the *Enviroment* based on a query towards the generative model.

The query process needs a little explanation: when we trained the generative model, the Y-sequence is a shifted version of the X resulting in a time series forecasting ability. We now utilize this property by actually replacing the feature for the *Action* with the proposal from the RL agent, and forecast the generative model based on the observation from the last step. The new forecast contains both Action and Observation, but we return only the Observation vector to the environment. However, the complete sequence with historical Action proposals, is stored in the environment class instance. This way, new queries will take into account the earlier states of combined Actions and Observations.

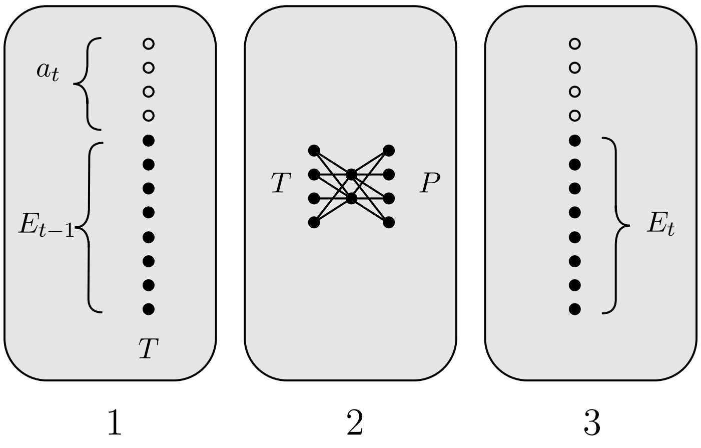


Figure 4.16: Querying the generative model
1. The previous Observation-vector is concatinated with the policy action proposal from the RL agent into vector T. 2. Vector T is predicted by the generative model, which returns P, which is the next step estimate. 3. The subset of the P-vector representing the Observation is stored for next step as well as returned to the RL Agent for processing.

Figure 4.16 explains the process, also outlined in Code 4.1. Figure 4.15 visualises an example of the model response when changing the input.

Within the `SyntheticEnvironment` class, we also define a *reward*-function where the system reward and termination conditions are evaluated.

4.3.3 Process model as RL environment

For us to evaluate the performance of the concept of training an RL agent with a generative model, another class, `ProcessEnvironment` is developed as a connector between the *real process* as implemented in the ordinary differential equations and the Matlab Reinforcement Learning framework. The `ProcessEnvironment` class can be connected to an RL agent for performance evaluation of the learned policy. During such execution, the termination

```
function this = step(this, actions)
    % Predict next step by changing action but keep observation
    % input to the network
    T = [actions this.Observation']';

    % Predict one step
    [this.net, P] = predictAndUpdateState(this.net,{T});
    P = P{1};

    % Update states
    act_size = size(this.datastore.MV, 2);
    this.Action = actions;
    this.Observation = double(P(1+act_size:end));
end
```

Listing 4.1: Querying the generative model (Matlab code)

criterion is disabled since the environment will be used for evaluation and simulation, and not learning. By disabling termination, we can observe the full length of the consequences of a learnt policy. This resembles the situation of executing the policy in real life, where process restarts are not always feasible.

4.3.4 Training a multivariate generative model

Droplet size optimization is added for the multivariate case, as presented in Figure 4.3. The term for controlling the atomizer speed, Z_{rpm}, is promoted from being a controlled variable to a manipulative variable in parallel with Q_{feed}. To train the generative model in the multivariate case, we do the following additions and changes:

- Rollouts are made from a set of four fixed rpm values in the range from 7600 - 9200, in addition to the Q_{feed} rollout scheme
- Huber loss function is introduced

Further, we train a model using 1024 hidden nodes (in two layers), trained for 3,000 epochs and compare performance with a model using 32 hidden nodes trained for 300 epochs. Both versions utilize the same learning rate decay, however, the length of the steps vary according to the number of epochs.

To learn the dynamics from the multivariate case, we present 60 unique rollouts per epoch, resulting in a training session equivalent to 180,000 unique rollouts. Total training time is 54 hours on a dedicated NVIDIA Quadro T1000 GPU for 3,000 epochs. The learning rate decay scheme is adjusted to accommodate for longer training sessions as illustrated in Figure 4.17.

We can see from Figure 4.18 the results of querying the generative model for random rollouts. The input is the previous sequence (20 steps) from a random rollout slid over the time frame. We see a positive correlation of the model compared to the real signal.

Further, we test the model's predictive performance by letting the model *free-run* by sliding the predictions. We do this to test the model's behaviour in a simulation scenario. By randomly step-wise changing the manipulative variables, and recording the sequence predictions, we can evaluate if the model has captured the process dynamics. Figure 4.19(a) show the results of keeping the manipulative variables constant and changing them at specific times t. Q_{feed} is changed (increased) at $t = 192$ and we see a temperature drop.

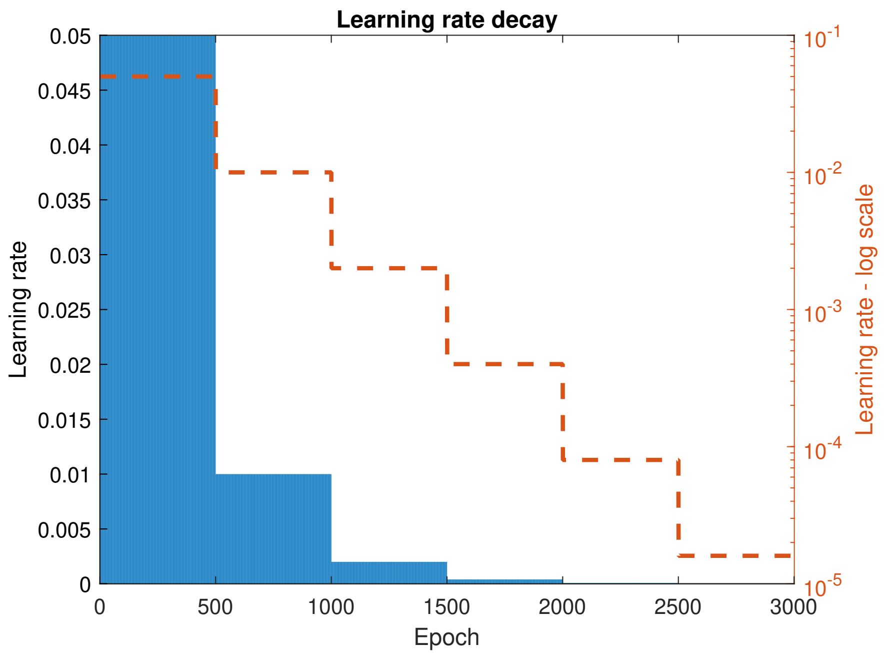


Figure 4.17: Step-wise learning rate decay

Further, at $t = 444$, Z_{rpm} is changed, affecting the particle size. The particle size is affected by both the feed rate and the atomizer speed.

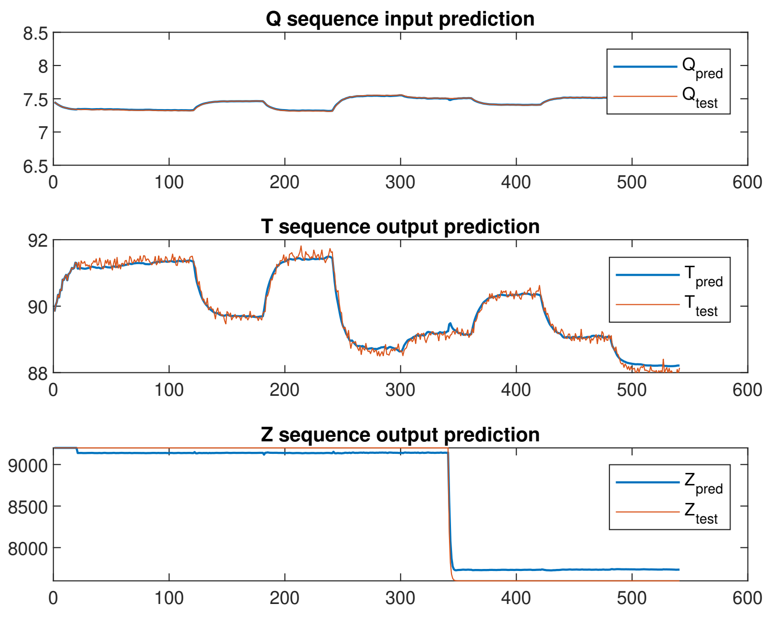
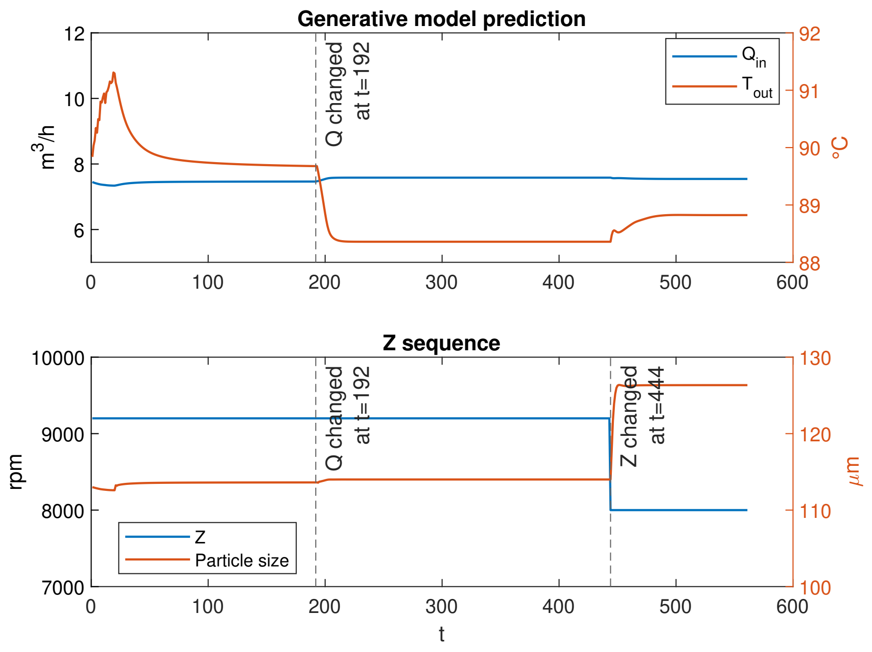


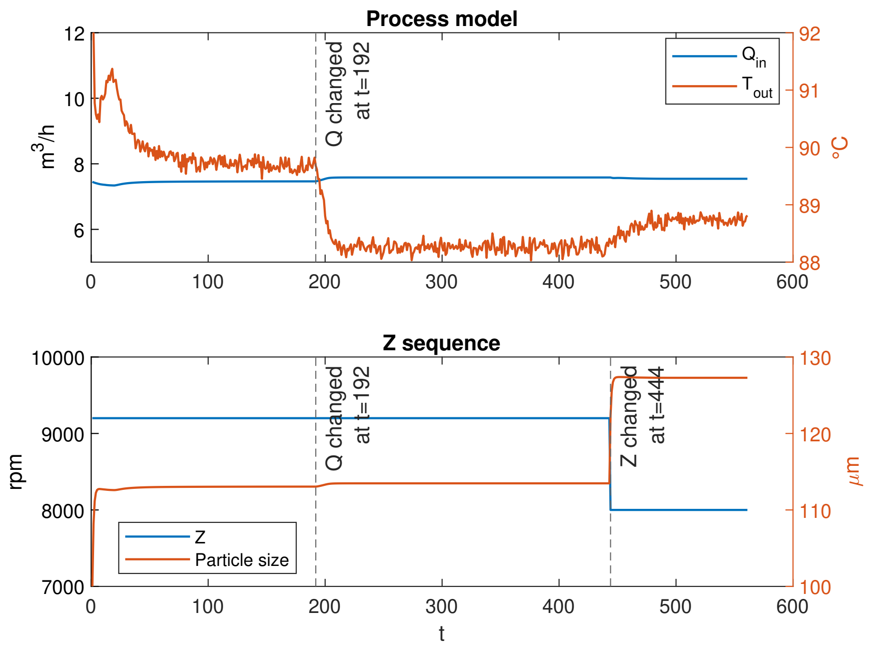
Figure 4.18: Results of querying the generative model, multivariate scneario

To evaluate if the generative model has captured any real process dynamics, we re-run the scenario towards the process model (the ODEs), employing the same input scheme and changes as we did towards the generative model. As we see from Figure 4.19(b), the process model response is in lieu with the generative model.

The scatter plots in Figures 4.20(a) and 4.20(b) visualises the correlation between the models. Some intermittent dynamics is not captured exactly, but we can conclude that the multivariate generative model has been able to replicate most of the dynamics.

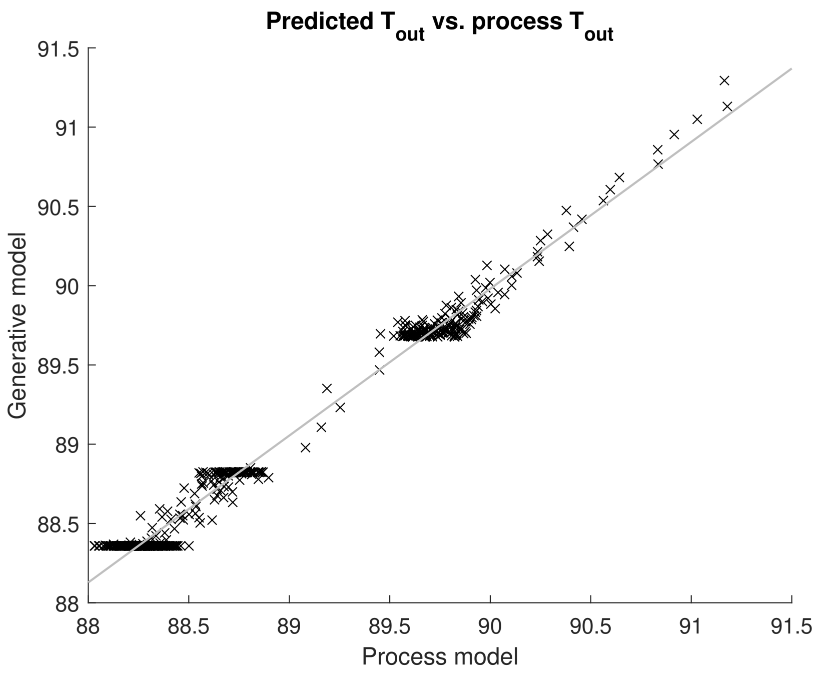


(a) Results of querying the generative model for next steps of random test run, multivariate scenario

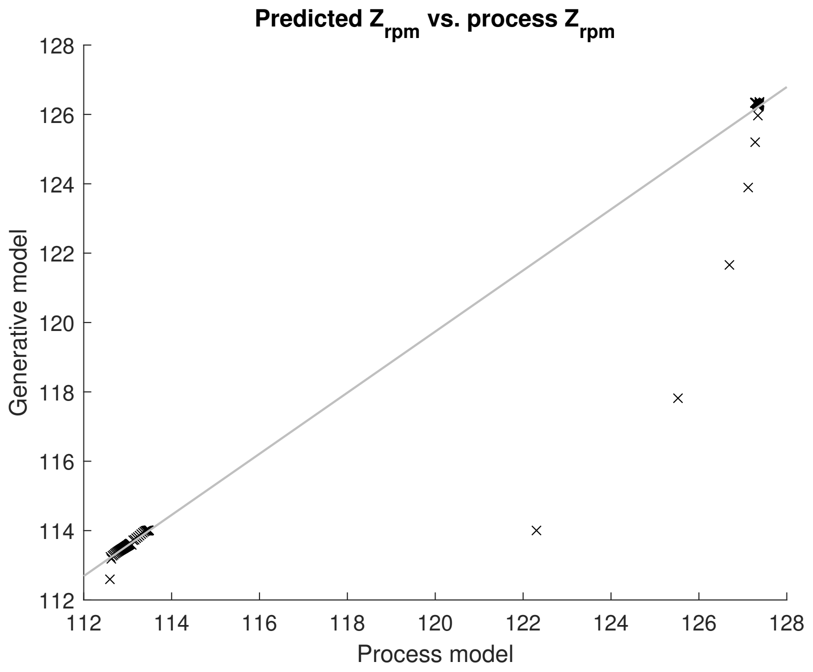


(b) Results of querying the process model for next steps of random test run, multivariate scenario

Figure 4.19: Multivariate query of the generative model vs. process model



(a) Correlation between generative model and process model for T_{out}, first sequence clipped



(b) Results of querying the process model for next steps of random test run, multivariate scenario

Figure 4.20: Comparison of generative model and process model

Huber loss function

As an alternative to *Matlab's* standard loss function based on calculated RMSE, we create a custom loss layer based on Huber loss estimation [9]. Huber loss is less sensitive for outliers in data than RMSE and essentially select either *mean absolute error* (MAE) or RMSE based on whether the residual a is large or small when calculating the loss:

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \le \delta, \\ \delta\left(|a| - \delta/2\right) & \text{otherwise.} \end{cases} \tag{4.4}$$

Code for the custom layer is presented in Listing 4.2. Here, we implement a forward loss function by using Matlab's own Huber loss function implementation.

```
classdef huberRegressionLayer < nnet.layer.RegressionLayer
    % Custom regression layer with Huber loss.

    methods
        function layer = huberRegressionLayer(name)
            layer.Name = name;
            layer.Description = 'Huber loss';
        end

        function loss = forwardLoss(layer, Y, T)
            % Calculate Huber loss of the mini-batch between
            % the predictions Y and the training targets T.
            loss = huber(Y,T,"DataFormat","CB");
        end
    end
end
```

Listing 4.2: Custom Huber loss regression layer (Matlab code)

The benefit of using a built-in `huber`-function, is that we do not need to implement the derivative of the Huber loss due to Matlab's *automatic differentiation technique*. The backward pass error derivative function is automatically determined by the framework during gradient descent.

The custom loss layer replaces the `RegressionLayer` in the neural network. The final generative model supporting both univariate and multivariate cases is listed in Code 4.3.

```
function layers = lstm_network(this, numHiddenUnits)
    layers = [
        sequenceInputLayer(this.datastore.featureDimension)
        lstmLayer(numHiddenUnits)
        dropoutLayer(0.2)
        lstmLayer(numHiddenUnits, "OutputMode", "last")
        dropoutLayer(0.2)
        fullyConnectedLayer(this.datastore.featureDimension)
        huberRegressionLayer('huber')
    ];
end
```

Listing 4.3: Generative model neural network layers (Matlab code)

Effect of changing number of epochs and hidden nodes

Increasing the number of nodes to 1024 and the number of epochs to 3,000 also results in an increased computation time. On an NVIDIA Quadro T1000 GPU, the computation time is

54 hours for 3,000 epochs of the selected rollout configuration. However, by recalculating the model using the original setup of 32 hidden nodes and 300 epochs, we gain performance. Figure 4.21 is the result of running 30 test sessions for each of the models: training a larger model results in a higher mean RMSE than a smaller model.

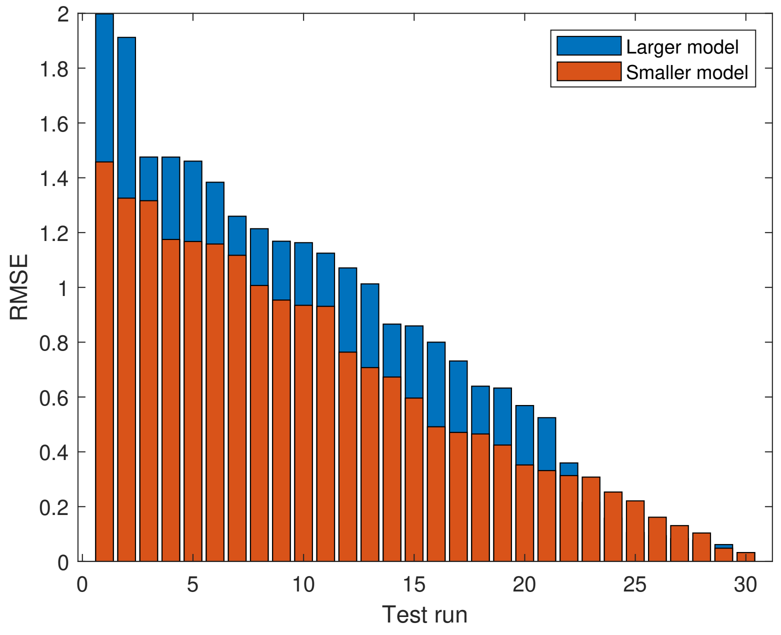


Figure 4.21: RMSE of larger model vs. smaller model
The larger model has a higher mean RMSE and standard deviation than the smaller.

The larger model has an average RMSE of 0.82 and $\sigma = 0.57$, while the smaller model has $\mu = 0.65$ and $\sigma = 0.43$. We can probably explain this difference from the fact that the smaller model, due to its smaller size, has a higher grade of generalisation. The smaller multivariate model will be used for the later experiments.

Effect of learning rate decay

The initial learning rate is according to Bengio [47] the *single most important hyperparameter* in a deep neural network and should be the first parameter to be tuned. The learning rate is a factor describing how much the network weights and other parameters should be changed on each iteration pass by the derivative error of the optimization function. A large initial learning rate avoids entering into spurious local minima and results in faster training. It is also a common belief that stepping down the learning rate avoids *oscillation* around a non-relevant parameter set, mathematically analyzed by Kleinberg and Yuan [85].

There are a few ways to decay the learning rate, most notably and simplest to implement, are *step-wise* and *exponential* learning rate decay schedules. However, other methods exist that can converge faster, on the cost of complexity. Lecun et al. [23] introduces the *stochastic diagonal Levenberg–Marquardt* procedure. Here, a *Hessian* matrix of second derivatives of the errors is used to calculate learning rates for *each individual parameter*.

For our problem, we select a step-wise approach due to its simplicity and path of least resistance.

Using the gradient descent explanation of the effect of the learning rate, we start with a high initial value $lr = 0.05$ and decay by a factor $r = 0.2$ on every n epoch, as illustrated in Figure 4.17. The total number of total decay steps is set to an empirical value $s = 6$. To see the effect of the stepping, training is halted at the end of each decay step and a query towards the generative model is done. This is compared to the test data produced by the process model. Figure 4.22 illustrates the results after the first four decay steps.

We observe the following interesting phenomena:

- Noise is suppressed after the first decay step
- Patterns are identified after the second decay step

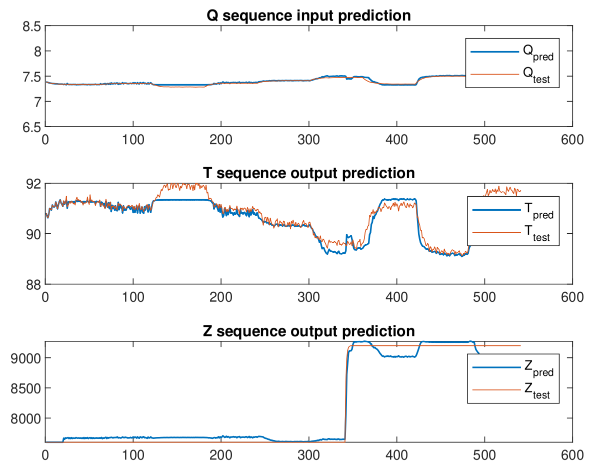
Our training progresses by constantly tweaking the network parameters in the right direction, updating in smaller steps as we decay the learning rate. After breaking the initial *loss wall* during the first iterations, we observe a stable learning process where noise is suppressed and patterns are more precisely recognised. Either, our dataset gently conforms to the gradient descent theory, where few parameter valleys exist, or we see a different effect.

You et al. [102] introduces an alternative explanation than gradient descent that fits the observation from our data. Their view is that;

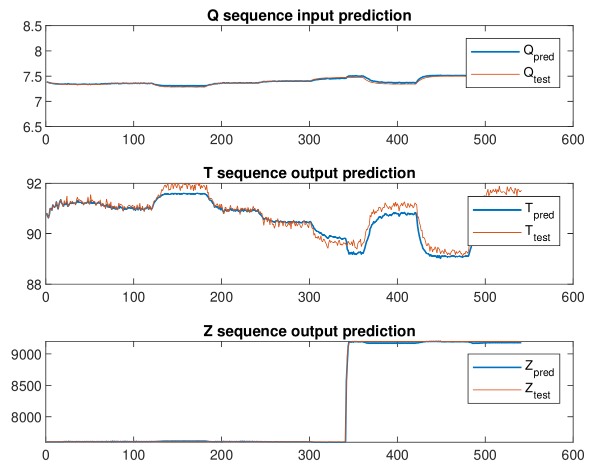
- an initially large learning rate suppresses the memorization of noisy data
- while decaying the learning rate improves the learning of complex patterns

Through experiments, they challenge the original theory of gradient descent and attribute this to the fact that deeper networks need other explanations, than what is developed for simpler neural networks. They also argue that the focus on minimas must be turned towards the data; noisy data is suppressed by a high learning rate, and the learning rate decay step learns complex patterns only after this noise suppression.

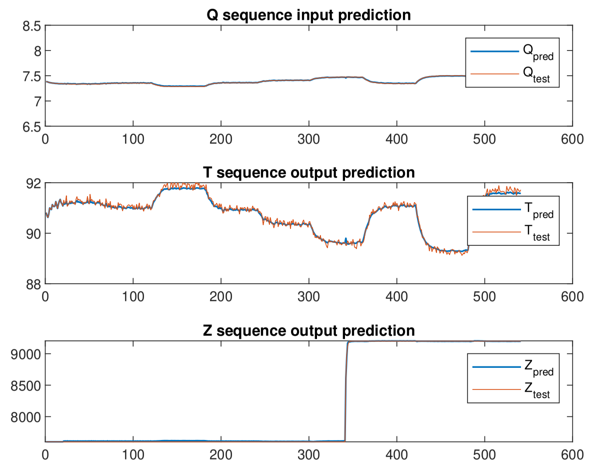
Our generative model's ability to learn is in either case accomplished by utilizing learning rate decay scheduling. Although we can support their idea by data, we cannot falsify the gradient descent theory.



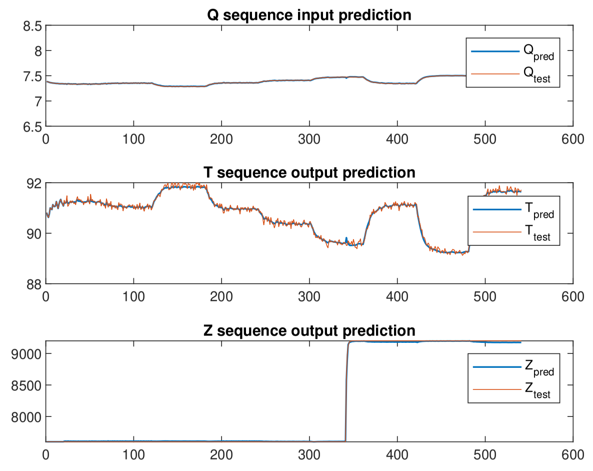
(a) Step 1: $lr = 0.05$ after 50 epochs



(b) Step 2: $lr = 0.01$ after 100 epochs



(c) Step 3: $lr = 0.002$ after 150 epochs



(d) Step 4: $lr = 0.0004$ after 200 epochs

Figure 4.22: Effect of first four learning rate steps in training the generative model

Chapter 5

Autonomous Control

The road to wisdom?—Well, it's plain and simple to express: Err and err and err again but less and less and less[1]

5.1 Design of the Reinforcement Learning agent

This chapter is dedicated to the concept of *learning for control* by using a reinforcement learning agent, trained by the generative model. Several experiments are conducted to reflect the performance of the concept.

Three different agent types are evaluated for learning a control strategy: the Soft Actor-Critic (SAC) agent, Proximal Policy Optimization (PPO) agent and the Deep Deterministic Policy Gradient (DDPG) agent. They all share the common idea of learning from an environment based on rewards. Whereas SAC is sample efficient, PPO excels in stochastic high-dimensional environments. DDPG is the deep learning's equivalent of *Q-learning* in continuous action spaces.

5.1.1 Soft Actor-Critic (SAC)

The soft actor-critic (SAC) algorithm is described as a *model-free, online, off-policy, actor-critic reinforcement learning method* by Haarnoja et al. [83]. The algorithm calculates a policy that maximizes the long-term expected reward, but also the entropy of the policy. We can think of entropy as the measure of uncertainty of a state, and the higher the entropy value, the more exploration of this uncertain state is promoted. Balancing exploration and exploitation of the environment is done by both maximizing the cumulative long term reward and the entropy.

SAC operates stochastically, meaning that we try to act as random as possible while still completing the task. For a process control problem, this might be a good fit: we want to minimize training time and we might have limited data available for off-line training. Maximizing the usage of the available data means that exploration of the state-space is limited to the area surrounding the process *mid-point*, meaning that the agent should learn the dynamics only within the available data window. This is an important pre-requisite, as the data window should describe *possible* process logs only, and not that of greatly exaggerated excitements. Selecting an agent that is compatible with the view of the generative model's inability to extrapolate outside the process window seems important.

[1]Danish poet and scientist Piet Hein (1905-1996)

To illustrate this point, let us denote π^* as the optimal policy that has the highest expected reward for every action:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \underset{\tau \sim \pi}{\mathbb{E}} \left[\sum_{t=0}^{\infty} \gamma^t [r(s_t, a_t)] \right] \tag{5.1}$$

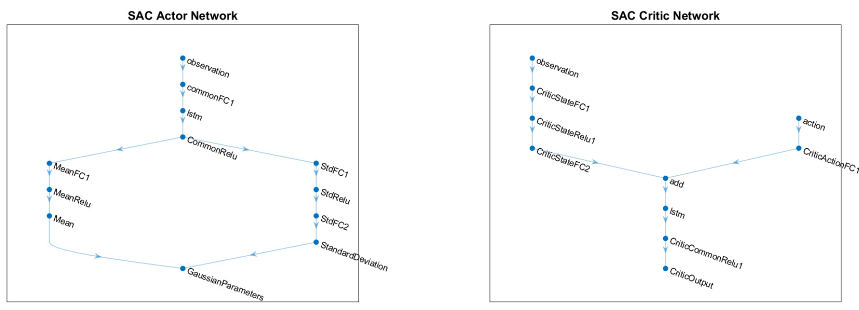
where trajectory τ has been sampled from the probability distribution of the policy π. In this scenario, the cumulative expected reward for the trajectory forces the agent to select actions that it might have visited before, but knowingly leads to a state of high reward. The *SAC* adds the entropy term $\mathcal{H}$:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \underset{\tau \sim \pi}{\mathbb{E}} \left[\sum_{t=0}^{\infty} \gamma^t [r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t))] \right] \tag{5.2}$$

where the *temperature parameter* α controls the relative importance of the entropy term against the reward [83]. This forces the agent to consider the uncertainty, not only the reward associated with the state and selecting actions it has not yet seen. This would mean that the agent stochasticity is somewhat controlled, at least when it comes to the degree of randomness introduced.

Further, gradient update of the policy is done off-policy utilizing the history of available data from a replay buffer. This gradient update means slower learning of the agent than just replacing the target network, as described earlier by Mnih et al. [51]. The soft policy update alternates between policy evaluation and policy improvement as described by Sutton and Barto [112].

Matlab Reinforcement Learning Toolbox includes an implementation of the *Soft Actor-Critic* algorithm and will be used in the experiments. The setup of the actor and critic networks is done by following the example SAC RL agent from *Matlab*.[2] We utilize long short-term recurrent memory blocks for Critic output and Actor input. Figure 5.3 visualises the two network architectures. The *temperature parameter* α was in the first publication of the SAC-algorithm an adjustable hyperparameter, although recent updates of the algorithm involve an automatic optimization of this α. Matlab includes this updated algorithm and is therefore not included in the hyperparameter search.



(a) Actor (b) Critic

Figure 5.1: Network architectures of the Soft Actor Critic (SAC) agent

[2]https://se.mathworks.com/help/reinforcement-learning/ref/rlsacagent.html

Input to the networks is standardised by normalising the data (subtract the mean and divide by the standard deviation). Setting up training involves selecting a few hyperparameters, mainly:

- Discount factor γ
- Target smooth factor τ
- Experience buffer length

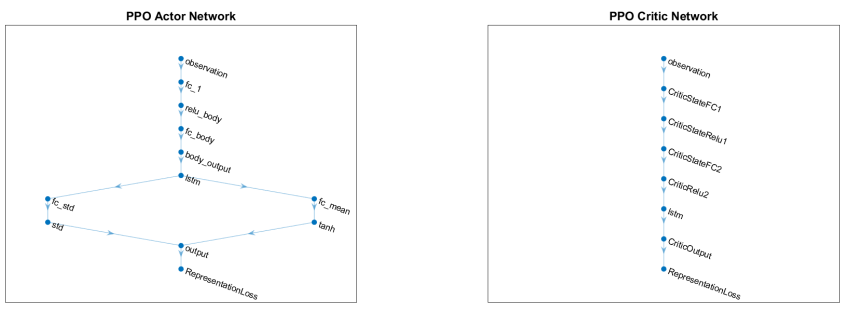
The entropy weight parameter α is automatically optimized by the algorithm.

5.1.2 Proximal Policy Optimization (PPO)

The Proximal Policy Optimization (PPO) algorithm, and its sibling Trust Region Policy Optimization (TRPO), seek to maximize the parameter steps to be taken when evaluating the next policy update. PPO and TRPO try to operate as stochastic as possible and step as long as possible, without stepping too far. PPO does this by evaluating the *Kullback–Leibler (KL) divergence* when optimizing the next policy update through a *clipped surrogate objective function* [73]. The KL-divergence is the *relative entropy* between two probability distributions and is somewhat related to the entropy term of the SAC agent.

PPO is a less complex implementation than TRPO and is included in *Matlab* as a parameterizable RL agent. We adjust the networks of the actor and critics to be compatible with the PPO agent while still keeping a similar configuration, as illustrated in Figure 5.3.

One notable drawback of the PPO algorithm is reduced performance in deterministic environments with few dimensions, which may cause a challenge utilizing our process model based on a few time series.



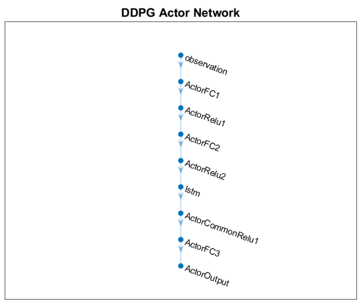
(a) Actor (b) Critic

Figure 5.2: Network architectures of the Proximal Policy Optimization (PPO) agent

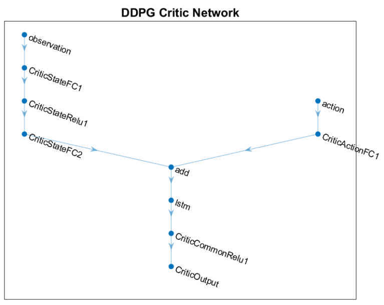
5.1.3 Deep Deterministic Policy Gradient (DDPG)

The Deep Deterministic Policy Gradient (DDPG) algorithm is *deep Q-learning* for continuous action spaces [54], [59] as visited and covered in Chapter 2.3.1. DDPG updates the target network by averaging and adding noise to the proposed actions as an exploration mechanism.

The networks for the DDPG agent has a similar configuration profile as for the SAC and PPO agents, and are visualised in Figure 5.3.



(a) Actor



(b) Critic

Figure 5.3: Network architectures of the Deep Deterministic Policy Gradient (DDPG) agent

5.2 Initial conditions

Whenever we start training an episode, we restart the environment to a known state, our initial condition. This operation places the agent at a known place in the environment and resets the generative model. Any steps taken by the agent will be evaluated solely from the episode actions, and not the accumulative state of the generative model. Depending on the application at hand, we might want to change this condition whenever the RL agent has gained enough learning. This is might be important in situations where we consider that there are significantly different conditions at other positions.

We can compare this to an autonomous driving simulator that provides environment feedback to an agent. At every episode start, we are placed at the same road intersection. But when we have learnt to handle the environment to some degree, we may want to rotate the starting point to a different place in the simulation, and even change the weather- or traffic conditions. In a time series approach, we could restart the environment to any point in time, so that the agent learns how to *escape* from the situation and into the action path that corresponds to the reward policy.

The generative model provides the initial data set at the beginning of each episode, and for the experiments, we choose to always start training at the same time series position for the current training session. A new initial observation is created during training of the generative model and is used for all subsequent RL training sessions. This way, we can compare agent performances at fair grounds. Figure 5.4 show a plot of the stored initial observation for the 300 epoch-version of the generative model we have used for RL agent training. The figure shows the five observation vectors in the same normalized plot, visualising the first 20 steps of a training sequence, to be used as our starting point. Initially, this vector was sampled from a random rollout from our process model. During learning, only the last time step vector is returned to the agent as the environment observation.

However, when simulating towards the process model, we draw a new rollout for each episode. This way, we make sure that the agent can escape any situation and enter the envelope satisfying the reward policy. Figure 5.5 visualises two random episode starts where we rotate the initial conditions on episode start.

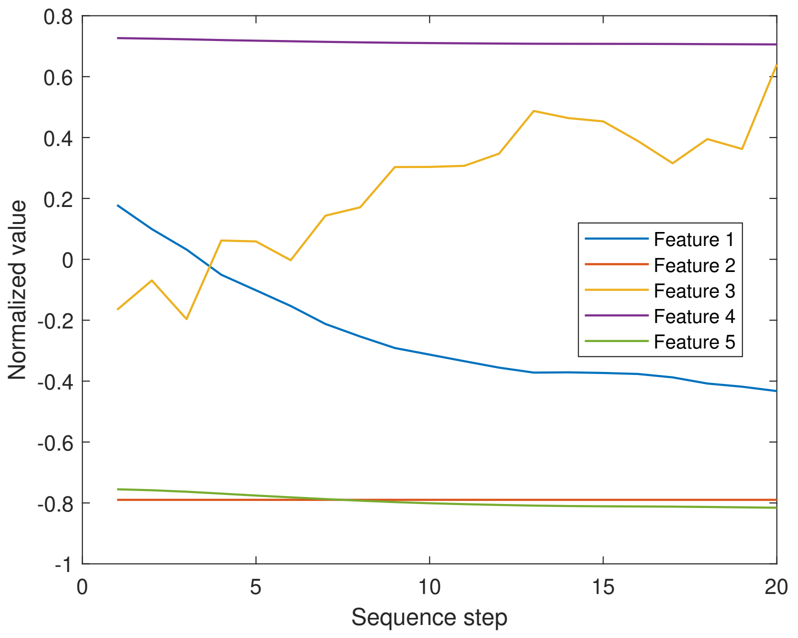


Figure 5.4: Normalized initial observation data for an instance of the generative model

5.3 Reward functions and terminal conditions

The reward function directly affects the behaviour of an RL agent. Ideally, we would like to incorporate several information types into one scalar rewarding signal: is the current state allowed and sound? Are the applied actions within tolerances? Do we have enough information to shape the agent's decision on the best way forward? The work of engineering a proper reward function can be an iterative task involving a lot of experimentation. According to the research questions, the role of a trainable reward function is investigated.

We will implement several strategies when engineering the reward function:

- Direct response reward from equation with $\gamma = 0.99$, univariate scenario
- Sparse reward from result distribution with $\gamma = 0.05$, multivariate scenario
- Trained reward model with $\gamma = 0.99$, multivariate scenario

We prepare the generative model as described above but simplify the datasets by not introducing noise to the manipulative variables, and excluding all controlled variables. Some noise (0.5%) is added to the responding variables, as instrument noise.

The training is done towards the generative model, while the simulation is done on the process model with no termination criteria. The process model environment represents the *real* system, so the agent cannot restart simulation even if the termination criterion is breached. This resembles the situation in the real world where the agent has no second chance for selecting a control strategy. Further, we add 0.5% noise to all controlled variables in the process model for increased realism of the simulator. The controlled variables are not part of the generative model training.

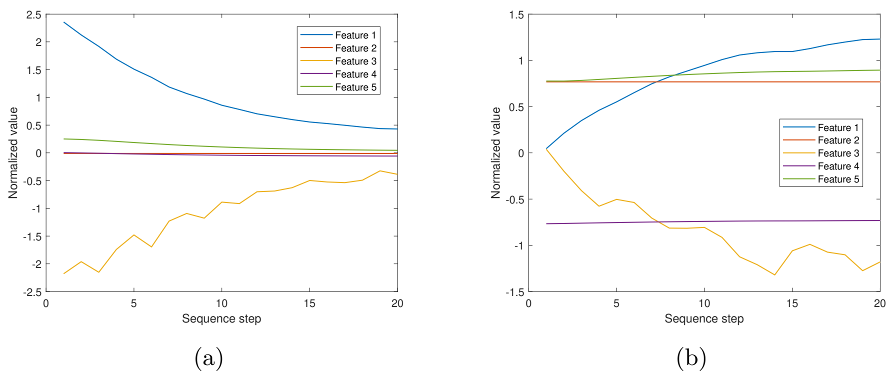


Figure 5.5: Normalized initial observation data for two random instances of the process model

5.3.1 Direct response reward, univariate scenario

The first strategy involves crafting a reward function that resembles the proportional PID-regulator in the process simulator. The aim is that the agent can behave similarly to the PID-regulator by calculating the inverse error between a T_{sp} and T_{out} and using that for maximizing the reward r. We also add a penalty term as a dampening factor for the selected action, taking the previous action into account (Δa):

$$\begin{aligned} r_{observation} &= \frac{1}{|T_{sp} - T_{out}| + \epsilon} \\ r_{action} &= -\beta \cdot |\Delta a| \\ r &= r_{observation} + r_{action} \end{aligned} \tag{5.3}$$

A starting point for the action dampening reward is set to $\beta = 0.05$. The action dampening will reward slowly changing the system over sudden moves in actions. Further, we add a small number $\epsilon = 1 \times 10^{-5}$ to the denominator in $r_{observation}$ to avoid any potential divide-by-zero errors. This reward function is implemented as part of the RL Agent step-function in Matlab:

Since the generative model is trained on rollouts in the range of an output temperature of 88-92 $°C$, we set the T_{sp} to a middle 90 $°C$ and a termination condition where we do not allow the system to continue training on the current rollout if the error is greater than the process window of 2 degrees. These are the process conditions and boundaries we have trained for from the historical logs. Since the reward function is targeted towards nearly immediate feedback, the discount factor γ is set high, typically $\gamma = 0.99$.

The following experiments are run using the univariate generative model trained for 300 epochs.

Experiment 0: Initial baseline

We need a way to identify when to stop agent training, the point where we assume that a suitable policy has been found. An easy method is to stop training when a certain average reward count has been reached. The theoretical total reward can be calculated by

```matlab
function [Observation,Reward,IsDone,LoggedSignals] = step(this,Action)
    epsilon = 1e-5;

    (... clipped ...)

    % Setpoint offset penalty
    setpoint = 90;
    errTout = abs(setpoint - scaled_world(2)) + epsilon; % MV + CV + RV vectors
    rT = (1 / errTout);

    % Penalize control effort
    rA = -0.05*abs(previous_action - action_scaled);

    % Get reward
    Reward = rT + rA;

    % Check terminal condition
    IsDone = errTout > 2;
    notifyEnvUpdated(this);
end
```

Listing 5.1: Direct response reward function (Matlab code)

multiplying the maximum step reward by the number of steps per episode. For the direct response reward function, this would mean that the total reward is

$$\begin{aligned}
r_{observation} &= \frac{1}{\epsilon} \\
r_{action} &= 0 \\
r &= r_{observation} + r_{action} \\
r &= \frac{1}{\epsilon} \\
r &= 1 \times 10^5 \\
r_{total} &= \sum_{i=1}^{600} r_i \\
r_{total} &= 6 \times 10^7
\end{aligned} \tag{5.4}$$

We would never reach this total reward due to the stochasticity of the system, so our first experiment is to find the point where a suitable policy has been found at the minimum training time. Beforehand, we do not know where this point might be, and it might vary with the application, environment parameters and the type of RL agent.

The average reward expected from one episode might dictate the threshold value we use for stopping training. As we see in Figure 5.3.1, a SAC agent has been running for 1323 episodes so that we can identify the step where the agent has identified the policy.

We can see that the agent policy is identified after ~ 700 episodes for the SAC agent and that the average reward threshold for stopping training could be set to 1.2×10^4 for the following experiments, for a value of 600 rollout steps per episode. We see that by training for an extended time, we might accomplish greater policy accuracy. For the experiments, we choose to select a lower number. Further, we set another stopping condition where we abort the training if more than 5000 episodes have been running without reaching the convergence criteria.

A secondary observation is that the SAC RL agent has identified a control policy by using the generative model as a dynamic process simulator. This is an important and

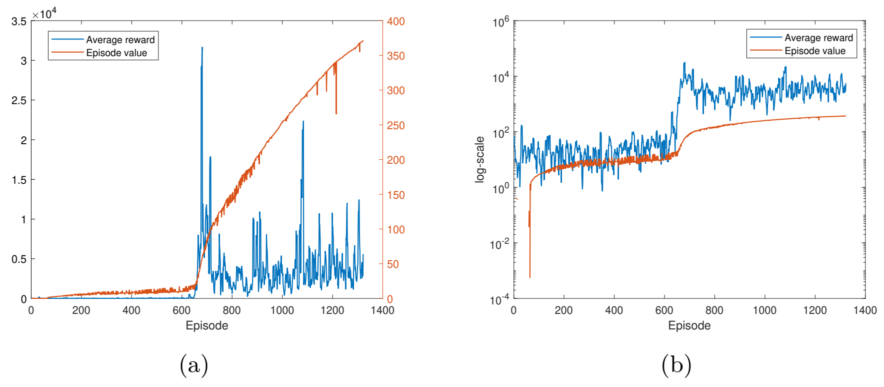


Figure 5.6: Experiment 0: SAC agent training progress (a) and log-scaled (b) We clearly see the step where a policy is identified (around step 700).

positive discovery in the direction of supporting the hypothesis claim. We also see that training performance increases beyond the stopping criteria, which could indicate that the agent is a continuous learner.

Experiment 1: Train & simulate a Soft Actor-Critic (SAC) agent with direct response reward, univariate scenario

A SAC agent with the following hyperparameters is executed until reaching the total reward criteria:

- Discount factor $\gamma = 0.99$
- Target smooth factor $= 1 \times 10^{-3}$
- Experience bufffer length $= 1 \times 10^{6}$

This results in the results visualised in Figure 5.7 and Table 5.1.

Agent type	Episodes	Value μ	Steps	Total Reward	Observation μ	Observation σ
SAC	868	18.76	600	23796.83	90.21	1.11

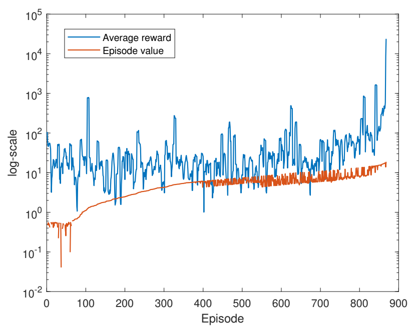
Table 5.1: Experiment 1: SAC training and simulation results

In Figure 5.7, we see that the agent action for the control signal for Q (blue line) successfully controls the observation of output temperature (red line) with an overall $\mu = 90.2$ and $\sigma = 1.11$, not far from the setpoint of the agent training towards the generative model.

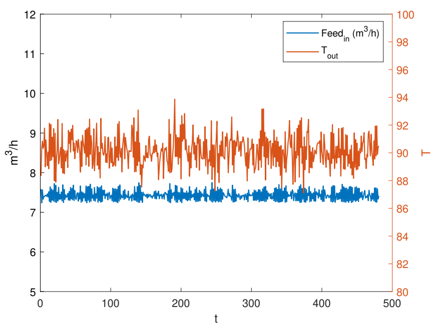
Experiment 2: Train & simulate a Proximal Policy Optimization (PPO) agent with direct response reward, univariate scenario

We exchange the agent type with a PPO agent with the following parameters:

- MiniBatchSize = 20 (the sequence length)



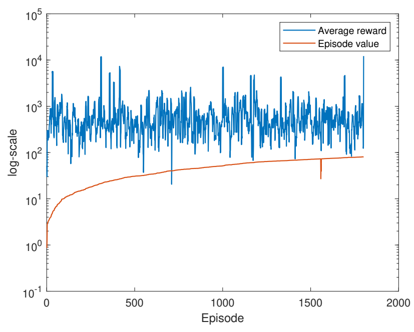
(a)



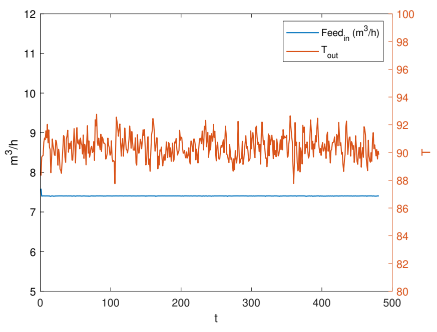
(b)

Figure 5.7: Experiment 1: SAC agent training progress on generative model (a) and simulated process model response (b)

- ExperienceHorizon $= 1 \times 10^6$
- Discount factor $\gamma = 0.99$



(a)



(b)

Figure 5.8: Experiment 2: PPO agent training progress on generative model (a) and simulated process model response (b)

Agent type	Episodes	Value μ	Steps	Total Reward	Observation μ	Observation σ
PPO	1801	80.7	600	12191.0	90.40	0.86

Table 5.2: Experiment 2: PPO training and simulation results

From Figure 5.8, we can see that the PPO reaches a more stable temperature signal, but need more than the double number of episodes compared to SAC for reaching convergence. This observation supports the claim that PPO is sample intensive. However, this might come with a positive side, since the generative model can produce any number of samples for training, given its learnt system dynamics.

Experiment 3: Train & simulate a Deep Deterministic Policy Gradient (DDPG) agent with direct response reward, univariate scenario

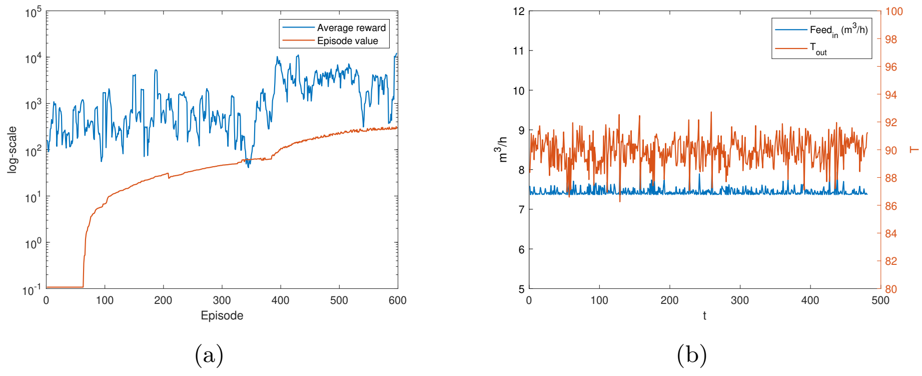


Figure 5.9: Experiment 3: DDPG agent training progress on generative model (a) and simulated process model response (b)

Agent type	Episodes	Value μ	Steps	Total Reward	Observation μ	Observation σ
PPO	598	283.28	600	12319.0	89.92	1.03

Table 5.3: Experiment 3: DDPG training and simulation results

Figure 5.9 show the results from DDPG training. We observe a performance close to that of the SAC agent.

5.3.2 Sparse reward and learning from failure

The second strategy is more involved and is set up to evaluate the performance of using sparse rewards for process control. The idea is that the agent should learn from its failures and only be rewarded when the results fall into an acceptable range. The agent will not get any accumulated reward during a sequence, which means that the control strategy is entirely up to the agent to develop. The reward scheme will not give any hints towards how the process is to be controlled. The agent should learn how to control the process to meet a specific end result that is sparsely presented.

The particle size calculations that are part of the responding variables and calculated by the empirical *Sauter mean droplet size formula* as presented in Equation 3.12 by Huang and Mujumdar [35], is used as sources for the sparse reward scheme. The end result might typically be a laboratory test result which often is a delayed parameter in real processes. In our experiments, we use the particle size distribution from the *Sauter mean droplet size* calculations to resemble this relationship. We define an acceptable normally distributed end result and the desired centre volume, where the reward function is a statistic evaluated to resemble the sparsity of the reward. We use a low discount factor of $\gamma = 0.05$ to promote long-term planning.

To evaluate this strategy, we analyse the particle size records from the multivariate test case above. A histogram of the result from this test case is demonstrated in Figure 5.10.

We also calculate an overlaid normally distributed curve by using the `normpdf`-function, where the distribution μ and σ are parameters. This curve represents our desired particle size distribution.

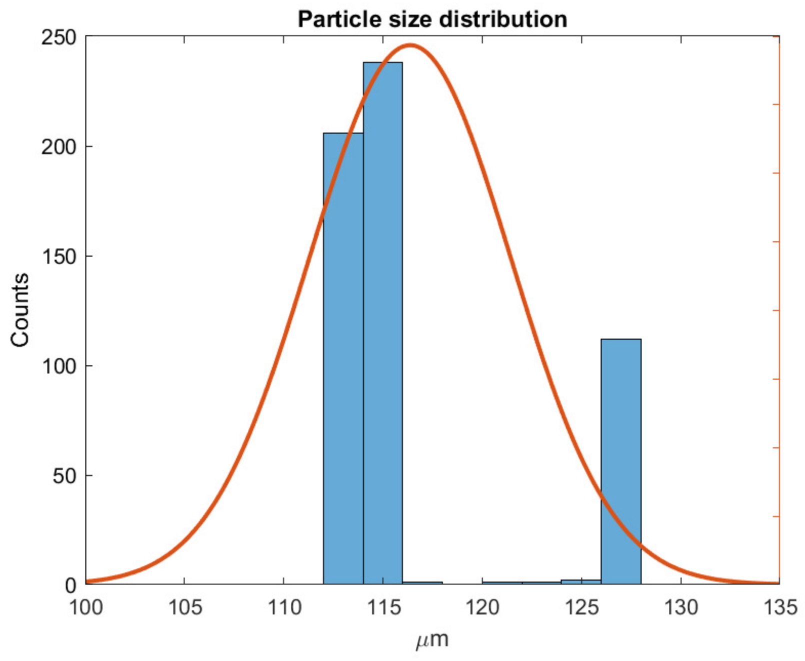


Figure 5.10: Particle size distribution from multivariate test case
$\mu = 116.36$, $\sigma = 5.07$, Anderson-Darling-test failed with $p = 1.07 \times 10^{-6}$

Our sparse reward scheme should evaluate whether a distribution is close to normality or not, and we select Matlab's *Anderson-Darling*-test[3] for this evaluation purpose. The *Anderson-Darling*-test returns a decision for a null hypothesis indicating that the test data is from a population of normal distribution within a 5% significance level. The dataset for Figure 5.10 fails this test, but when we create a randomized demonstration dataset with 351 samples based on the same μ and σ, we pass a re-run of the test, as visualised in Figure 5.11. This would be our desired process result during episode training.

The statistical comparison function results in a sparse reward: 1 if the conditions for normality and volumes are satisfied, otherwise 0. Whenever the conditions are not present during learning, we terminate the episode. We start the reward evaluation after collecting some samples during the start of an episode, so the first few samples will receive 0 reward and termination will be inhibited until we start the evaluation.

To be able to accomplish learning to control by sparse rewards, the agent would need to learn how to change parameters dynamically to meet the distribution criteria, in our case the Q_{feed} parameter and the speed of the atomizer Z_{rpm}, which is part of the formula for mean droplet size. This means that Z_{rpm} will be changed from a *controlled variable* to a *manipulative variable* in those experiments, as presented through the multivariate generative model discussion. Further, we do not employ any control restrictions, meaning

[3]https://se.mathworks.com/help/stats/adtest.html

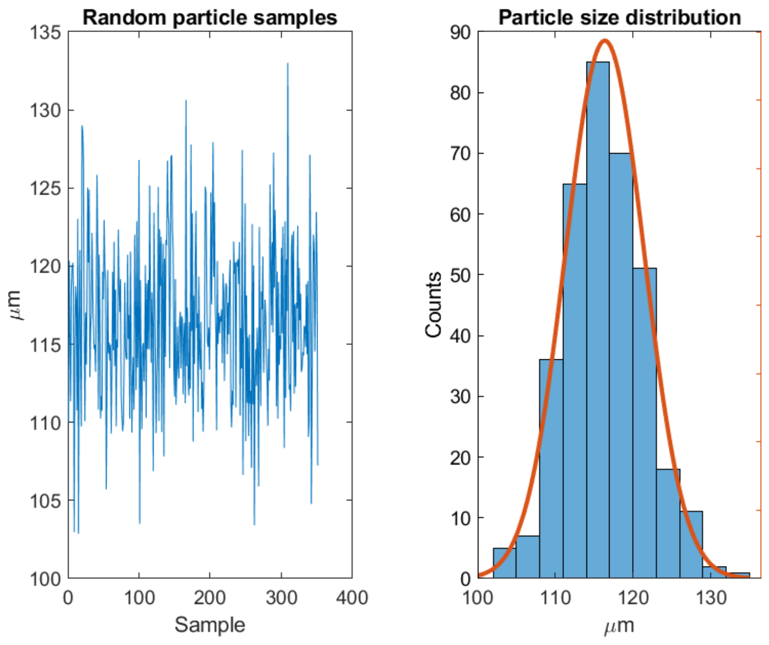


Figure 5.11: Randomized particle samples
Sample size: 351, $\mu = 116.36$, $\sigma = 5.07$, Anderson-Darling-test pass with $p = 0.81$

that the agent is free to control the system to any temperature and speed levels necessary. The designed reward function is listed in Code 5.2.

Experiment 4: Train & simulate a Soft Actor-Critic (SAC) agent with sparse reward, multivariate scenario

A SAC agent is trained using sparse reward for 327 episodes using manual termination. We can see from Figure 5.12 that a policy was identified after roughly 50 episodes.

When simulating the agent towards the generative model, we see from Figure 5.13 an interesting policy. The agent rapidly varies both Q_{feed} and Z_{rpm} in order to create a particle distribution that satisfies the sparse reward scheme. This behaviour is repeatable across several retries of learning and simulation.

Agent type	Episodes	Value μ	Steps	Total Reward	Observation μ	Observation σ
SAC	327	0.17	33	33.8	90.79	5.74

Table 5.4: Experiment 4: SAC training and simulation results

We then test the agent in a free-running process environment (the ODEs), and we see the same pattern of rapid action control, in Figure 5.14 with more exaggerated responding variables.

The policy seems to be a good example of the *Cobra effect* [25], where our control scheme has unintended effects we originally did not design it for: high temperature variation and rapidly exaggeration of control signals in order to reach the goal as quickly as possible.

```
function [Reward, IsDone] = sparse_reward(logged_signals, sequence_length)

    % Sparse reward
    dist = makedist('normal','mu',122,'sigma',5); % target distribution
    rS = 0;
    h = 0;
    if size(logged_signals,1) > sequence_length
         theset=logged_signals(sequence_length:end,4);
         [h,p,adstat,cv] = adtest(theset',"Distribution", dist);
         rS = ~h;
    end

    % Get reward
    Reward = rS;

    % Check terminal condition
    IsDone = h;
```

Listing 5.2: Sparse reward function (Matlab code)

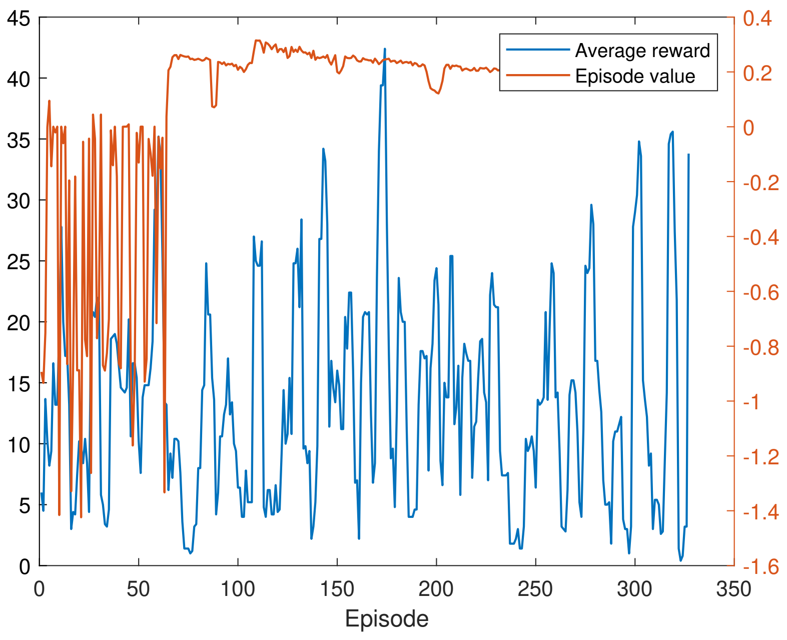
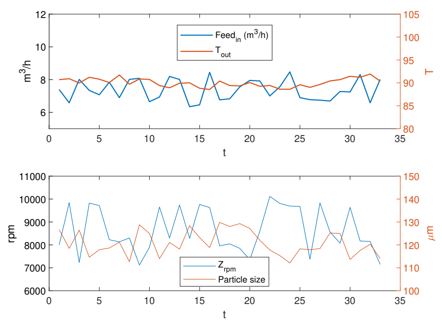


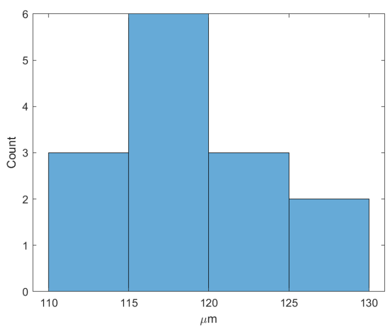
Figure 5.12: Episode data for training an agent based on a sparse reward scheme

A human operator would probably ease the operating envelope for the machinery, as well as distribute the controls over longer intervals for more controlled production. A sparse reward scheme for the multivariate case is probably not suitable for real-life control without adding penalty terms for the control efforts.

Employing a *reward shaping* process, in which we further tweak and redesign the direct response reward in an iterative fashion, can be an alternative. Reward shaping can take other properties such as control dampening, process enveloping and soft ramping into account. This quickly becomes a manual multivariate optimization task, an engineering effort we hope to avoid.

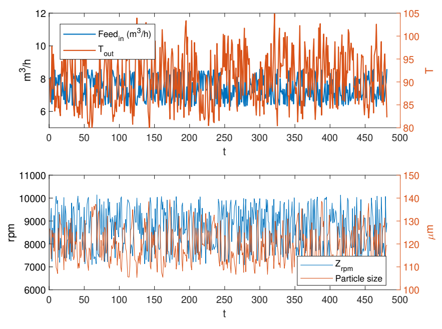


(a) Simulation result sequence, $\mu = 90.06$, $\sigma = 0.93$

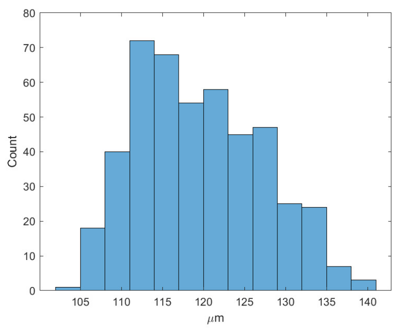


(b) Particle size distribution

Figure 5.13: Simulation towards generative model



(a) Simulation result sequence, $\mu = 90.79$, $\sigma = 5.74$



(b) Particle size distribution

Figure 5.14: Simulation towards process model

5.3.3 Trained reward model, multivariate scenario

The goal of the agent is to control the process within operating limits, the envelope of dynamics, and reach some kind of specified target. We have seen in the former experiments examples of reward functions that can control the process. However, the reward strategy we choose can either involve tedious detailed work as done by the team for the Dota 2 competition [98], or we could introduce a level of *imitation learning* in which the manual efforts could be lower.

In classical imitation learning, the policy is learnt directly and replaces the agent model during simulation. Hence, a reward strategy is not necessary. If we would enable the agent to evolve by continuous training during execution, we need to employ some kind of reward model. As we have visited earlier, *Inverse Reinforcement Learning* is an example of imitation learning where the agent needs to identify a reward model based on demonstration examples. This could be by intervention from an expert, where we assume the demonstration is correct. Or, it could be by sampling from function approximators that are trained towards the desired behaviour.

We could use the learnt generative model as a reward model. By calculating the inverse error ΔE between the normalized versions of *observation estimate* and *actual observation*,

$$\begin{aligned}\Delta E &= E_{estimate} - E_{actual} \\ r &= \frac{1}{|\Delta E|}\end{aligned} \tag{5.5}$$

reward r would continuously update the reward according to visited states. This way, the agent will learn to replicate the behaviour exhibited in the logs and in theory reward any action that conforms to the generative model.

However, there might be a conflict between the recorded logs and the goal of the agent. The logs might include dynamics outside the required process envelope, and data that comes from an uncontrolled process, in that sense it does not follow the goals of our requirements. By introducing a *classification learner*, we can build a reward model where only desired behaviour is recorded. For example, only sequences that contain dynamics that are manually marked as *desired* could be used for learning the reward model. The difference between the environment and the state of the reward model would dictate a lower reward. Hence, an agent exhibiting similar dynamics as the reward model is trained. This way, we do not employ static limits but rather learn from examples of *good processes*.

To set up an experiment for such a reward model, we create a new subclass `RewardModel` which share the functionality of the parent class `GenerativeModel`, although the parameter values to initialize the class instance might be different. This model will then be trained on *good examples* only and queried in the reward function. The inverse error between the model estimate and the actual observation is returned as the action reward. The above equation and reward model query is implemented as a reward function in Code 5.3.

```matlab
function [Reward, IsDone, RewardModel] = learnt_reward(previous_action, ...
action_scaled, scaled_world, logged_signals, sequence_length, ...
generative_model, reward_model)
    % Extract sequence
    trail_length = min(size(logged_signals,1), sequence_length)-1;
    data = logged_signals(end-trail_length:end-1,:);

    % Query the reward model
    data = reward_model.scale(data);
    [reward_model.net, Y] = predictAndUpdateState(reward_model.net,{data'});
    Y = reward_model.inverse_scale(Y);

    % Scale to similar range
    Y = generative_model.scale(Y);
    X = generative_model.scale(scaled_world);
    err = abs(sum(X-Y));

    % Get reward
    Reward = 1/err;

    % Check terminal condition
    IsDone = 0;
    if size(logged_signals,1) > sequence_length*2
        IsDone = err > 2;
    end

    RewardModel = reward_model;
```

Listing 5.3: Reward function for querying the reward model (Matlab code)

When training the reward model, we use the same network parameters as the generative model, but the rollout function is changed to produce *good samples* only, that is, the temperature setpoint T_{out} is restricted to a stable level (90°). Further, the Z_{rpm} is fixed at 7600 rpm with a 0.1% added noise. The controlled variables are fixed and removed in this model. We instantiate, train and save the reward model by:

```
rewardmodel = RewardModel(filename, @spray_rollout_pid_stable, 60, 120, ...
    128, 32, sequence_length, environment_length)
[rewardmodel,~] = rewardmodel.train_model
rewardmodel.save_model
```

The environment is instantiated after training by refering the reward function and reward model:

```
env = SyntheticEnvironment(genmodel, true, @learnt_reward, rewardmodel);
validateEnvironment(env);
```

Experiment 5: Train & simulate a Soft Actor-Critic (SAC) agent with reward model, multivariate scenario

A SAC agent with the following hyperparameters is executed until reaching the total reward criteria (set to 12,000):

- Discount factor $\gamma = 0.99$
- Target smooth factor $= 1 \times 10^{-3}$
- Experience bufffer length $= 1 \times 10^{6}$

The agent reach the termination criteria after 1695 episodes, whereby the deterministic policy was applied to the process model. The results are visualised in Figure 5.15 and Table 5.5.

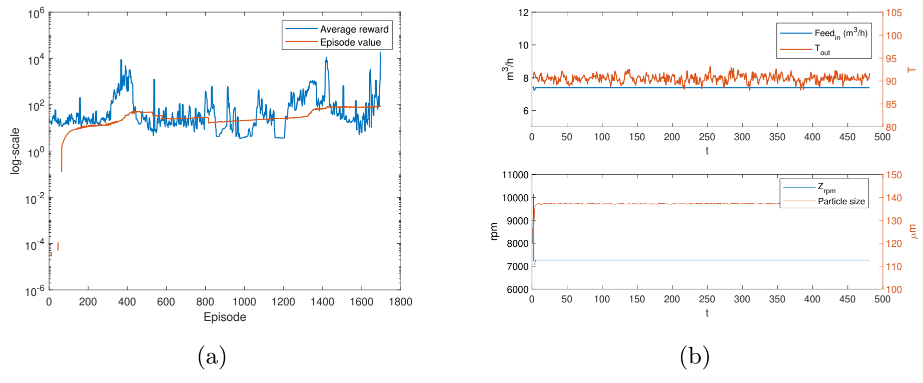


Figure 5.15: Experiment 5: SAC agent training progress on generative model (a) and simulated process model response (b) using trained reward model

In Figure 5.15, we see that the agent action for the control signal for Q_{in} has been identified with a reward model strategy.

We need to evaluate whether the policy is *sane* by setting up new experiments 6-7. Here, we will evolve the agent by switching the training environment from the generative model, to the process model.

Agent type	Episodes	Value μ	Steps	Total Reward	Observation μ	Observation σ
SAC	1695	87.42	480	19158	90.55	0.84

Table 5.5: Experiment 5: SAC training and simulation results

Experiment 6: Evolving the SAC agent from Experiment 5

The trained policy has been shown to control a simulated process environment by its deterministic policy. In this experiment, we evolve the agent by using the process environment directly during training and using the trained reward model for the reward function. The process environment is instantiated with a reference towards the reward model and its query function;

```
envProcess = ProcessEnvironment(genmodel, true, @learnt_reward, rewardmodel);
validateEnvironment(envProcess);
```

We add 0.1% noise to the controlled variables and train the agent for additional number of episodes, reaching termination criteria after 79 episodes:

```
trainStats = train(agent,envProcess,opt);
```

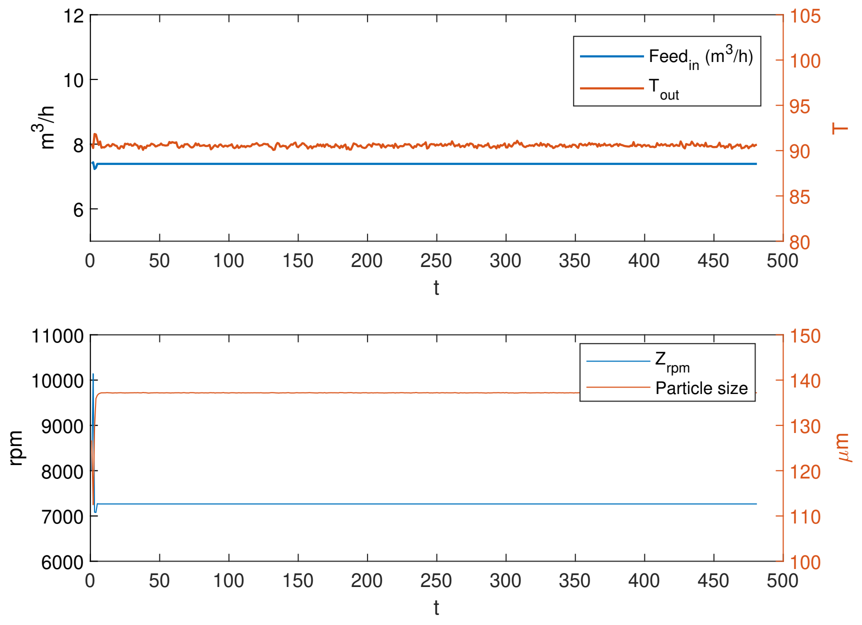


Figure 5.16: Training SAC agent towards process model, $\mu = 90.56$ and $\sigma = 0.19$

We see from Figure 5.16 that the agent keeps the process in control according to the reward model.

Experiment 7: Exaggerated process environment on SAC agent from Experiment 5

This experiment introduces increased process noise (0.1%) on the controlled variables and further trains the SAC agent from experiment 5, within the process environment. Training is terminated after 106 episodes when reaching the termination criteria.

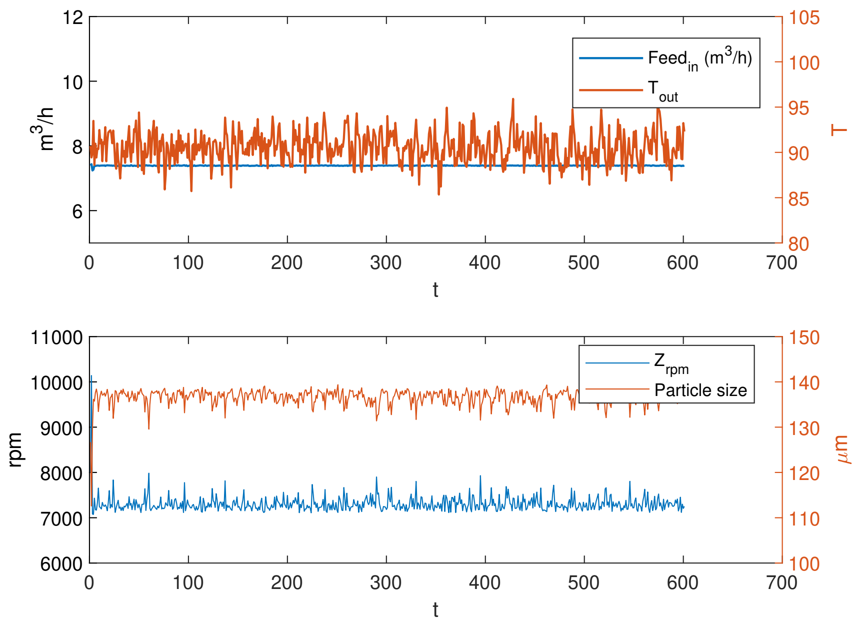


Figure 5.17: Training SAC agent towards process model with 0.1% noise on controlled variables, $\mu = 90.54$ and $\sigma = 1.71$

We can see from Figure 5.17 that the agent does not control the temperature at all, but seeks to reduce the error from the reward model by increasing the atomizer speed and compensating with process noise by rapidly changing the speed. The feed rate Q_{in} is not touched by the agent. According to the synthetic process model, the dynamics for the atomizer has faster response times than regulating the feed rate. This might be the reason for this behaviour.

Experiment 8: Train & simulate a Proximal Policy Optimization (PPO) agent with reward model, multivariate scenario

Agent type	Episodes	Value μ	Steps	Total Reward	Observation μ	Observation σ
PPO	200	15.98	480	17203	90.00	0.28

Table 5.6: Experiment 8: PPO training and simulation results

A PPO agent identifies a policy after a short amount of episodes, and we see that conformance is reached after a stable learning process.

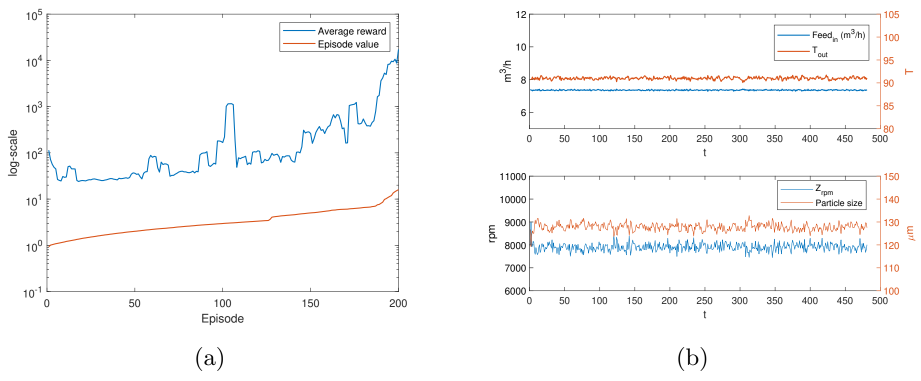


Figure 5.18: Experiment 8: PPO agent training progress on generative model (a) and simulated process model response (b) using trained reward model

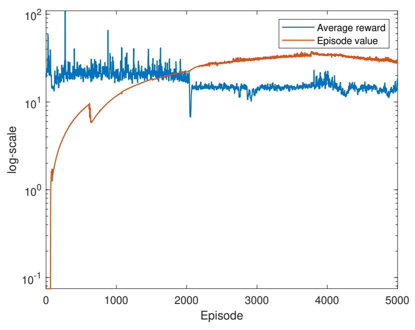
Experiment 9: Train & simulate a Deep Deterministic Policy Gradient (DDPG) agent with reward model, multivariate scenario

A DDPG agent was trained but was automatically terminated after 5000 episodes, on several attempts. Figure 5.19 show that the reward level stays stable and that the agent is not able to control the process. Further, we see that when adding 0.1% noise to the controlled variables, the agent has no policy for a countermeasure.

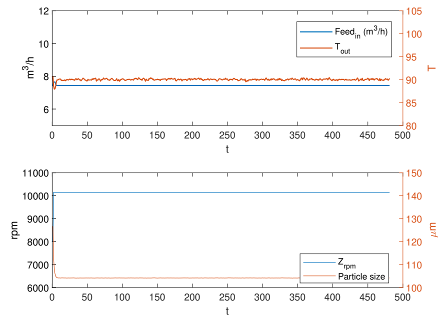
Agent type	Episodes	Value μ	Steps	Total Reward	Observation μ	Observation σ
DDPG	5000	28.04	480	15.72	90.00	0.23

Table 5.7: Experiment 9: DDPG training and simulation results

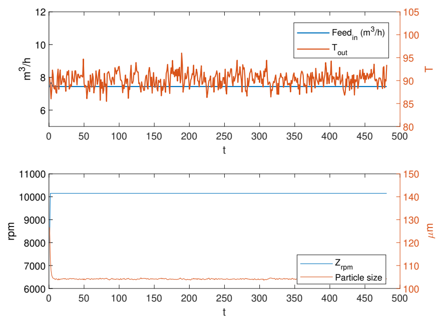
The policy of the agent has not reached convergence within the limit for the experiment training.



(a)



(b)



(c)

Figure 5.19: Experiment 9: DDPG agent training progress on generative model (a) and simulated process model response (b), and introducing 0.1% noise on controlled variables (c), using trained reward model

5.3.4 Trained reward model with an inverse mean squared error as reward

As the reward model seems promising in itself, the reward calculation can be improved by replacing the inverse mean absolute error, with the mean squared error, so that ΔE becomes:

$$\Delta E = (E_{estimate} - E_{actual})^2$$
$$r = \frac{1}{|\Delta E|} \tag{5.6}$$

and the termination critera $T(\Delta E)$ is updated to:

$$T(\Delta E) = \begin{cases} 1 & \text{for } \sqrt{\Delta E} \; > \delta, \\ 0 & \text{otherwise.} \end{cases} \tag{5.7}$$

where $\delta = 2$, to leave room for evolving the episodes. $RMSE = 1.5$ when training the reward model, so the environment could possibly reach this error level. A theoretical maximum reward per episode will in this case be $\frac{1}{\epsilon} = 1 \times 10^5$, but we establish a baseline by first training the agents for several runs. The PPO agent was able to reach an average reward on several trials at $1,800$ which was set as the termination criteria for training SAC, PPO and DDPG agents. This reward corresponds to an RMSE of $1800^{-1} = 5.6 \times 10^{-4}$.

The reward model should penalize larger control efforts by using the squared error and ease the compensation of slow dynamics by faster dynamic control signals.

Experiment 10: Train & simulate a PPO agent with reward model, MSE evaluation, multivariate scenario

A PPO agent was trained using the MSE method of evaluating the result from the reward model. We see from Figure 5.21 that the PPO agent has identified a policy, and according to Table 5.8, the μ and σ are well inside the original data range of $\pm 1°$.

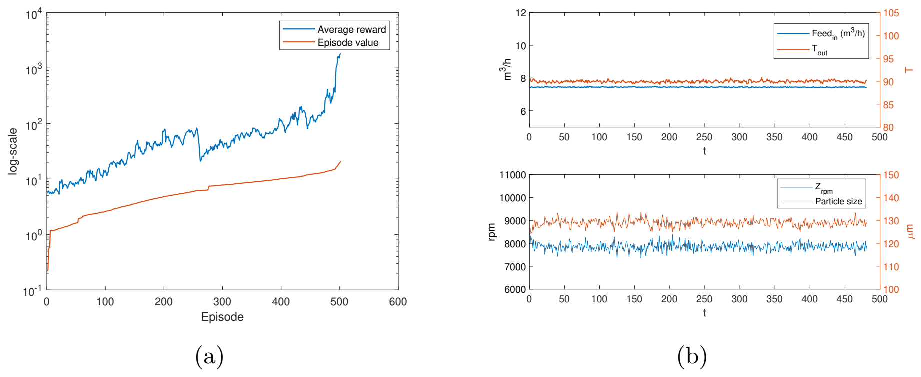


Figure 5.20: Experiment 10: PPO agent training progress on generative model (a) and simulated process model response (b) using trained reward model

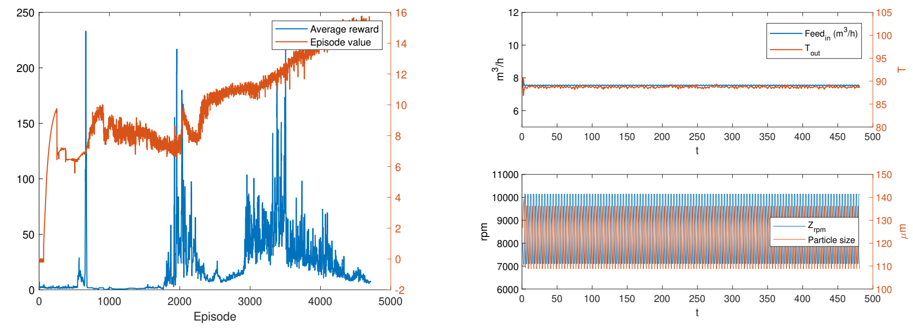
Agent type	Episodes	Value μ	Steps	Total Reward	Observation μ	Observation σ
SAC	502	20.91	480	1804	89.95	0.24

Table 5.8: Experiment 10: PPO training and simulation results

Experiment 11: Train & simulate a SAC agent with reward model, MSE evaluation, multivariate scenario

A SAC agent was attempted trained but never reached the termination goal for average reward. Training was automatically terminated when reaching 5,000 episode counts.

The maximum average reward for the training pass was 233.07, well beyond the termination criteria.



(a) (b)

Figure 5.21: Experiment 11: SAC agent training progress on generative model (a) and simulated process model response (b) using trained reward model

Experiment 12: Train & simulate a DDPG agent with reward model, MSE evaluation, multivariate scenario

A DDPG agent was attempted trained but never reached the goal for the reward. Training was automatically terminated when reaching 5,000 episode counts.

5.3.5 Training towards the process model using the reward model

An alternative approach to train our RL agent is to utilize the process model directly, and the learnt reward model. In such a scenario, the agent is directly connected to a real-time process, in our case the ordinary differential equations. In this particular case, we disconnect the generative model and focus on the reward model only.

Experiment 13: Train & simulate a PPO agent towards process model and synthetic reward model, MSE evaluation, multivariate scenario

A PPO agent was trained reaching the termination criteria of an average reward of 1800. We can see from Figure 5.22 and Table 5.9 that converge was reached after only 194

Agent type	Episodes	Value μ	Steps	Total Reward	Observation μ	Observation σ
SAC	194	26.28	480	1893.7	90.00	0.22

Table 5.9: Experiment 13: PPO training and simulation results

episodes. This experiment demonastrates the RL concept validity and that an agent can learn from a process model.

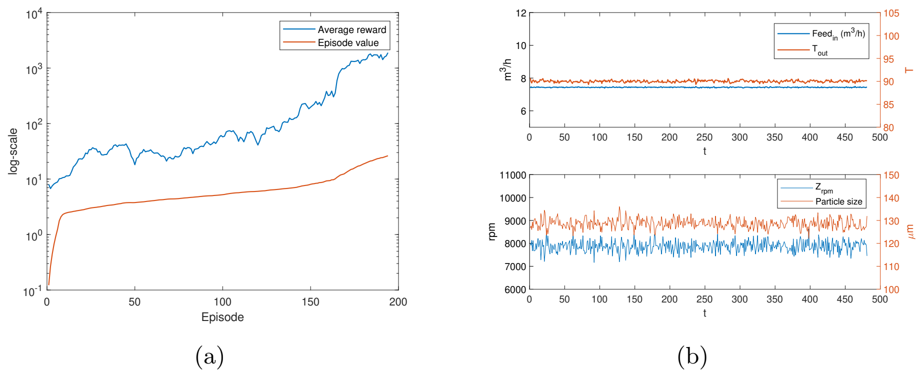


Figure 5.22: Experiment 13: PPO agent training progress on generative model (a) and simulated process model response (b) using trained reward model

5.4 Covering unmodeled dynamics

The experiments show us that the agent quickly learns the system dynamics from the process and that the mean and standard deviation is comparable to when training the agent towards the generative model. An important discovery is that a PPO agent trained with the generative model performs similar to an agent trained on the real process. Further, a reward model trained on time series representing a class of data can be used for agent training towards behaviour that resembles such classes, e.g. *good* data, or other sets of data classes. However, we also see from experiments that the agent has limited knowledge of proper countermeasures when presented with cases representing time series data outside the dynamics envelope it was trained for. We also see that the agent compensates for changes in the environment by selecting fast-moving dynamics Z_{rpm} instead of slowly moving the control signal, which in the real world might be a more costly strategy than regulating with the Q_{in} due to the mass transport involved in changing the rotation speed of the atomizer. These are both examples of the *Cobra effect* discussed earlier and a set of *conflicting objectives* which to the agent only has one solution: the path of least resistance to reach the reward.

The background for this behaviour might descend from the process data's properties:

- There is synthetic instrument noise we add to the responding variables which might be mistaken by the agent, in our human sense, as a selectable action space

- The synthetic model of the atomizer is not taking the cost of speed change into account
- A change of dynamics in the controlled variables has not been trained for in the generative model and the reward model
- There is no difference in cost taking action A over action B

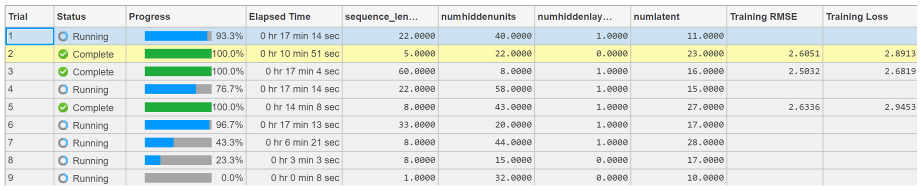
To robustify the model, we increase the process model space by exciting the process parameters slightly. This will involve adding the controlled variables to the process model, which we model as stable with added noise. The process model is already developed for these signals, so adding them means that we clear the suppression of them.

5.4.1 Retraining the generative model

We turn our attention towards training the generative model for an excited process. This means that we expand the trained range for the temperature setpoints to $88 - 142°C$ and add increased normally distributed instrument noise to the controlled variables to 0.5%. This way, we produce data that should lie outside the process operational window so that the generative model can learn policies to escape such situations. We also change the process model for the atomizer to accommodate for slower dynamics, by a factor of 0.01. Exciting the process this way is similar to the model identification steps in classic control theory.

With these changes implemented, we do a new hyperparameter search by Bayesian optimization. This time, the minibatch (60), the number of rollouts (128) and the number of epochs (30) is kept constant. We will optimize the neural network architecture: number of hidden units, number of layers and sequence length. Also, we add the number of hidden layers to the search, to see the effect of stacked LSTM layers.

The `RolloutDatastore` class is extended to support parallel processing by also subclassing `matlab.io.datastore.Partitionable` and implementing the required methods. This way, the parameter search can utilize Matlab's Parallel Computing Toolbox and increase the throughput of our experiments by running several workers in parallel, either locally or by using a cloud service. An example of parallel hyperparameter search is shown in Figure 5.23, which is a screenshot for the Matlab Experiment Manager. Here, we see that three workers have completed their search run, while six other workers are active.



Trial	Status	Progress	Elapsed Time	sequence_len...	numhiddenunits	numhiddenlay...	numlatent	Training RMSE	Training Loss
1	Running	93.3%	0 hr 17 min 14 sec	22.0000	40.0000	1.0000	11.0000		
2	Complete	100.0%	0 hr 10 min 51 sec	5.0000	22.0000	0.0000	23.0000	2.6051	2.8913
3	Complete	100.0%	0 hr 17 min 4 sec	60.0000	8.0000	1.0000	16.0000	2.5032	2.6819
4	Running	76.7%	0 hr 17 min 14 sec	22.0000	58.0000	1.0000	15.0000		
5	Complete	100.0%	0 hr 14 min 8 sec	8.0000	43.0000	1.0000	27.0000	2.6336	2.9453
6	Running	96.7%	0 hr 17 min 13 sec	33.0000	20.0000	1.0000	17.0000		
7	Running	43.3%	0 hr 6 min 21 sec	8.0000	44.0000	1.0000	28.0000		
8	Running	23.3%	0 hr 3 min 3 sec	8.0000	15.0000	0.0000	17.0000		
9	Running	0.0%	0 hr 0 min 8 sec	1.0000	32.0000	0.0000	10.0000		

Figure 5.23: Running Matlab Experiment Manager: parallel bayesian hyperparameter search of the generative model

There is a danger here that every time a worker is finished, the framework calculates a new set of parameters for the next available worker. Since the smaller parameter space is more computationally efficient, we will naturally get an unbalanced result set since more experiment evaluation is done on smaller parameter sets. Fewer workers testing larger parameter sets are able to finish, and could therefore compromise the results since their share in the bayesian calculation is low. We need to evaluate the results with this in mind and distribute our weights taking the unbalanced result set into account. An alternative approach is to do an exhaustive parameter search if we assume it is important to cover the whole parameter space, or discard the parallel search altogether and apply sequential search.

As seen by the screenshot results in Appendix B, the results from the hyperparameter search show us that the validation RMSE for all 30 runs is ~ 2.5. We can infer that the network architecture has less impact on the result, as well as the sequence length.

The sequences could be adjusted to the length we believe captures important start- and ends of dynamics, e.g. a time series containing data from one recipe run, or from a block of residence time in the process. Due to the slowly varying dynamics of the atomizer speed, we therefore select a sequence length of 60 time steps. Clearly, we would need a better method to estimate the sequence length as this directly influences the reinforcement learning performance, more than the generative model.

We also see from the hyperparameter search that the validation loss is lower for a higher number of hidden nodes, using no hidden layers. We select 64 hidden nodes in two layers as our architecture.

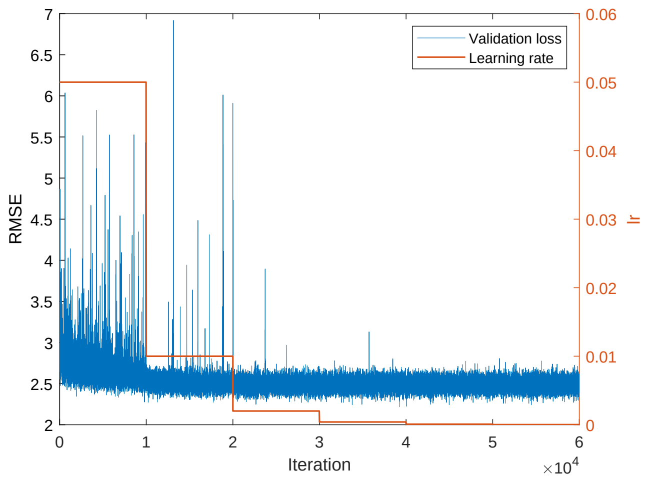


Figure 5.24: Generative model retraining with modifications, RMSE=2.5

With these modifications, the generative model is retrained during 300 epochs. The process is visualised in Figure 5.24 utilizing the learning rate decay schedule as discussed earlier.

5.4.2 Retraining the reward model

We establish a new instance of the reward model by training the network over sequences generated from rollouts created from *good process* data. In this case, we establish 110°C as the setpoint for the PID regulator and present the reward model with these sequences.

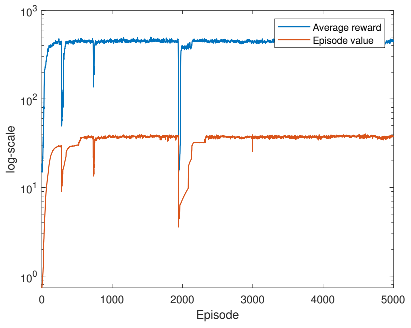
0.1% noise is also added to the controlled variables as well as the temperature sensor output. The resulting RMSE is 2.7 for the dataset.

5.4.3 Retraining the agents

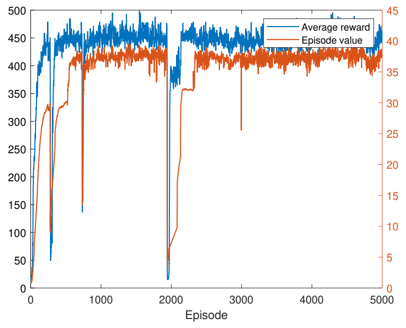
Using the new generative model and the new reward model, we retrain our agents. Further, we complexify the agent training by randomly sampling a new rollout for every initial observation where the added noise is 0.5%. This way, all episodes are provided with an initial observation that is completely new, although taken from the Gaussian distribution of the prior. Further, we set the discount factor $\gamma = 0.98$ to accommodate for a longer sequence length.

Experiment 14: Train & simulate a PPO agent towards noisy generative model and reward model, multivariate scenario with initial observation rotation

The PPO agent did not reach the convergence criteria of an average reward of 1800 and terminated on the episode count of 5000. However, as we can see from the training plot in Figure 5.25, the agent has repeatedly identified several policies, although they are discarded in favour of a new search branch when not terminating.



(a)



(b)

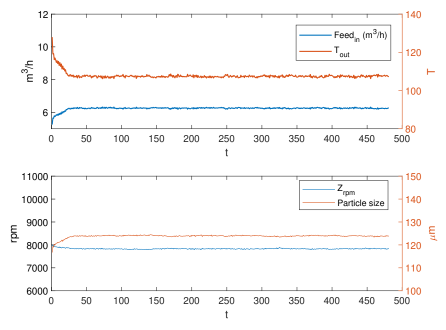
Figure 5.25: Experiment 14: PPO agent training progress on (a) log sccale and (b) normal scale

The agent cycles from *valley to valley* and ends with a policy with an average reward of 460.15 and a value of 38.44, which also is the average maximum value for all attempts. We then simulate the agent both towards the generative environment as well as the process environment.

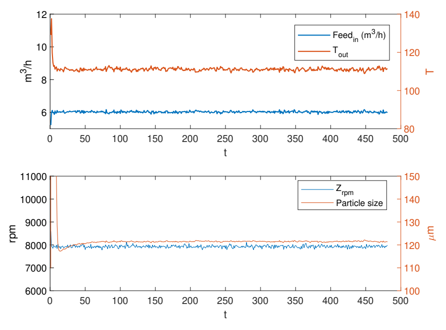
In Figure 5.26, we can see that the agent policy reaches the reward model stable signal at 110°C with a $\sigma > 1$. However, the performance might be a good starting point for evolving the agent on a real process.

Experiment 15: Train & simulate a SAC agent towards noisy generative model and reward model, multivariate scenario with initial observation rotation

A SAC agent was trained towards the complex model and reached convergence after 188 episodes with an average reward of 3564.7 and value of 5.72.

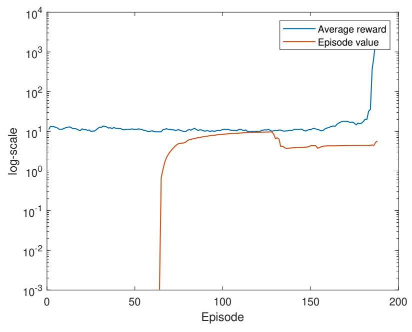


(a)

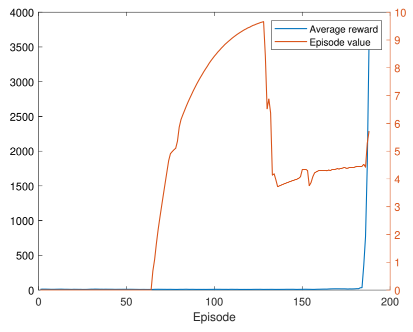


(b)

Figure 5.26: Experiment 14: PPO agent simulation towards (a) the generative model ($\mu = 107.73, \sigma = 1.76$) and (b) the process model ($\mu = 111.20, \sigma = 1.67$)



(a)



(b)

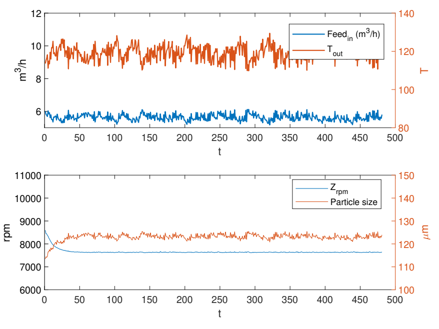
Figure 5.27: Experiment 15: SAC agent training progress on (a) log scale and (b) normal scale

However, the results when simulating are not that impressive, as seen in Figure 5.28. It might seem that the agent has *locked on* to the noise distribution and generated a gaussian policy that resembles the noise.

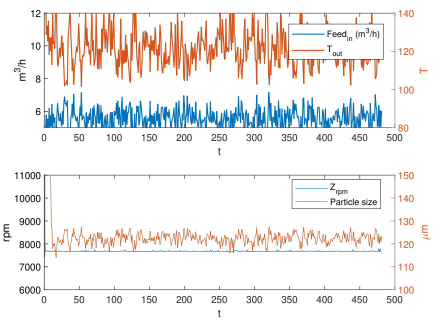
The high reward given might just be that the random policy and random noise is randomly in phase. However, we could anticipate the situation by also evaluating the *discounted future reward*, which is lower than that for the PPO agent policy and setting this as the termination criteria. However, the Matlab framework does not allow for explicitly setting *value* as a termination criterion. Another method could be to increase the average reward termination criteria, to avoid what seems to be a *phase locked loop*.

Experiment 16: Train & simulate a DDPG agent towards noisy generative model and reward model, multivariate scenario with initial observation rotation

A DDPG agent was attempted trained for several sessions but never reached any termination criteria before being manually stopped. The computation resources needed exceeded the quotas multiple times, as only a few hundred episodes were evaluated after > 14 hours with



(a)



(b)

Figure 5.28: Experiment 15: SAC agent simulation towards (a) the generative model ($\mu = 119.4, \sigma = 4.18$) and (b) the process model ($\mu = 121.67, \sigma = 9.67$)

GPU training each time. The operator-observable average reward for these non-evaluated sessions was in the range of $10 - 15$, a low number compared to the other agents.

The bad performance on DDPG for our experiments can be explained by DDPG's exploration strategy being non-compliant with our data augmentation policy and model dynamics. DDPG explore the environment by randomly changing the action on every step, whilst the environment dynamics might not respond in time for this agent to evaluate the policy correctly.

5.5 Estimating uncertainty

In the preceding experiments, we have demonstrated the capability to learn dynamics from recorded data and train a reinforcement learning agent on these data, eliminating the need for environment modelling in such cases. The culprit remains that we still must excite the process, or at least use data from such sessions, in order to be able to learn all facets of the process. When increasing the size of the observation space, such as when adding more sensor observations or incorporating image- or spectroscopic data, the data dimensionality increase becomes a combinatorial problem where the optimum is visitation of all relevant states during training. If the observation space is Gaussian, the policy could have a conformed uncertainty across states. However, in high-dimension non-linear systems with long time-dependent seasonality variations, we must ensure we have balanced datasets from these periods.

Ideally, we should have training data and real-world data that are *exchangable* in terms of sequences, meaning that we capture all relevant dynamics we can foresee in the real world. Let Z denote sequences of state-action-pairs, and we have

$$Z_{history} \sim Z_{future} \tag{5.8}$$

What we have observed during the experiments, is that the agent action proposal does not always lead us to the correct reward situation, if the current situation is not trained for, as demonstrated in Experiment 15. We should have tools to remedy the situation by estimating the uncertainty during operation, indicating situations where the agent is uncertain. As such, we should be able to estimate the uncertainty as a prediction interval:

$$[\hat{y} - \alpha \cdot \sigma, \hat{y} + \alpha \cdot \sigma] \tag{5.9}$$

where α is the desired level of confidence. Neural networks capture non-linear data, so applying a global statistic calculated from training data might not always hold, at least when considering sequence-to-sequence regression. We present here three popular methods, and our own method.

5.5.1 Monte Carlo simulation

One simple method that has shown its importance in light ray-trace rendering optimization, is the use of Monte Carlo simulations [31], [32]. Translated to our problem, we assume that the uncertainty of the non-linearity is already modelled as part of the neural network's latent spaces. By randomly perturbating the input from a distribution of choice, the prediction result of the network varies according to this uncertainty.

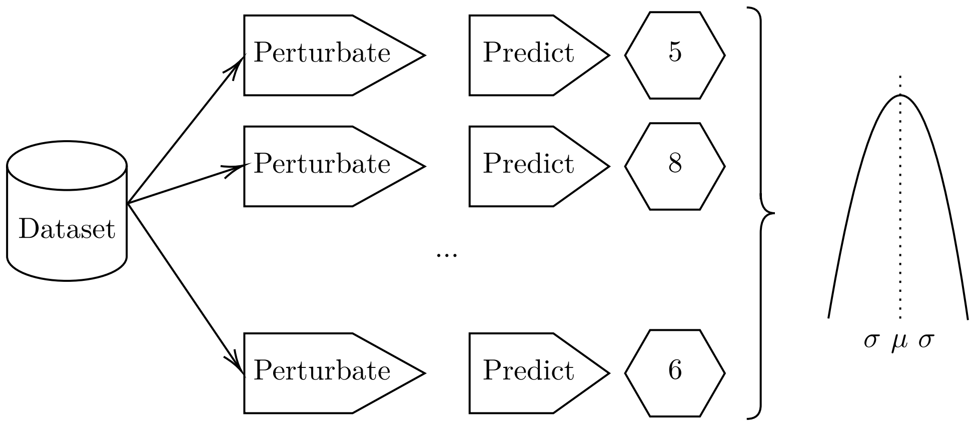


Figure 5.29: Monte Carlo simulation of neural prediction, a dataset is perturbated with a distribution of choice to produce a set of distributed predictions

As visualised in Figure 5.29, by running several perturbations per prediction cycle, we capture a prediction set where we can extract the mean and variance of the prediction. Depending on the sequence length, dimensionality and network architecture, the process might be computationally expensive in the inference phase. It has no impact on the training phase.

5.5.2 Variational inference

Another Monte Carlo simulation method is described by Gal and Ghahramani [56], [57] where we utilize *Dropout layers* during the inference phase. Dropout layers are normally active during training, where outputs are stochastically disconnected by setting their weights to 0. Using Dropouts is a simple and effective way of preventing overfitting in neural networks [55], and is used in our versions of both the generative model and the RL agents.

By changing the behaviour of the Dropout layer to also disconnect outputs during inference, we explore the *uncertainty space* of a model, as described by Gal and Ghahramani [57]. Instead of perturbating the input, the input is static for a repeated number of prediction passes through the network, where the Dropout will randomly disconnect weights according to some probability setting. In a sequence-to-sequence scenario, this would mean fewer iterations since we do not need to change the input data according to its length and structure.

We can implement a forward-pass/inverse Dropout layer in Matlab easily, as demonstrated in Code 5.4. Here, we draw a binary mask from some probability distribution by the `randsrc`-function and apply 0 weight to the outputs on each pass.

```
classdef inverseDropoutLayer < nnet.layer.Layer
    properties
        Probability
     end

    methods
        function layer = inverseDropoutLayer(name, probability)
            layer.Name = name;
            layer.Probability = probability;
            layer.Description = 'Inverse dropout';
        end

        function Z = predict(~, X)
            % generate an inverse binary matrix whos bit value is
            % drawn with a probability
            mask = randsrc(size(X,1), size(X,2), ...
                [1,0; layer.Probability, 1-layer.Probability]);
            mask = logical(mask);

            % set weights to zero and return
            X(mask) = 0;
            Z = X;
        end
    end
end
```

Listing 5.4: Inverse dropout layer (Matlab code)

In an LSTM scenario, we need to ensure that each pass is done on copies of the initial network so that the recurrent state is not altered. The result from multiple passes can be summarized statistically. This method is computationally less intensive than the previous example, although it scales linearly according to the training passes selected (e.g. 30 passes per prediction).

5.5.3 Conformal prediction

The area of *conformal prediction* involves estimating the uncertainty during inference, described by Vovk and Balasubramanian et al. [30], [52]. The concept applies to any type of regression or classification algorithm, but in essence, we add estimation of the statistical parameters to the regression method as long as the datasets for training and testing are *exchangable* as described earlier. In addition to predicting the mean, we add prediction and training towards another distribution metric, like the variance, standard deviation or other quantile descriptors.

We can apply this method in several ways. One method described by Mendels [96] involves building two models: $\boldsymbol{m\mu}$ and $\boldsymbol{m\sigma^2}$, where $\boldsymbol{m\mu}$ is first trained the usual way:

$$E_\mu(x) = \sum_i^N (y_i - m_\mu(x_i))^2 \tag{5.10}$$

This model is trained on one part of our training data set. The other part is used for training $\boldsymbol{m\sigma^2}$ with the squared error as the dependent variables:

$$E_{\sigma^2}(x) = \sum_i^N ((y_i - m_\mu(x_i))^2) - m_{\sigma^2}(x_i)^2 \tag{5.11}$$

This involves two passes our one dataset during training, and as relatively small impact on the inference, although prediction of the confidence interval must be done separately, as shown in Code 5.5.

```
% Prerequisite:
% x_test, x_train, y_train and x_var is given in workspace

% Normalization parameter calculation
x_mean = mean(x_train);
x_std = std(x_train);

% Train the mean network
options = trainingOptions('adam', 'MaxEpochs',1000);
mu_net = trainNetwork(x_train, y_train, model_fc(x_mean, x_std), options)

% Train sigma network based on mean network predictions
y_varpred = predict(mu_net, x_var);
y_se = (y_var - y_varpred).^2; % squared residuals
sigma_net = trainNetwork(x_var,y_se,model_fc(x_mean, x_std),options)

% Predict test network
y_pred_mu = predict(mu_net,x_test);
y_pred_sigma = predict(sigma_net,x_test);
```

Listing 5.5: Our understanding of conformal prediction (Matlab code)

If the framework allows, it is also possible to implement the prediction into one network, as demonstrated by Thorn [90]. Here, the number of outputs from the network is increased to accommodate for the statistics prediction. During a custom training pass, a custom loss function is called on each minibatch to update the dataset statistics.

5.5.4 Value-based uncertainty

In an actor-critic reinforcement learning problem, we have all the information available for our uncertainty estimation, although in a slightly different form. An RL agent draws the action from a Gaussian distribution in the policy, and this distribution is described by the *value* learnt by the *critic* from its action probability distribution. By querying the *critic* by the *observation sequence*, we receive a representation loss value approximated from the critic network trained towards the long-term reward.

The received value cannot be unpacked to represent each signal's contribution to the reward, but by keeping the training statistics from the RL agent training, we can find the maximum value obtained during training. Hence, we can calculate a probability value representing the agent's confidence of being able to reach the goal given the current state observations, given that the agent converged:

$$p_{level} = \frac{v_t}{\max(V_{train})} \tag{5.12}$$

To utilize this concept, we create a new method `sim_confident` that replaces the `sim`-function in Matlab with a custom simulation function, based on the template given from the Matlab documentation.[4] This would give us an indication on the certainty of agent state, and hence the following multivariate action proposals.

There are several ways to utilize this level, for instance in the secure evaluation of the agent, as we will see in the next chapter. We could also use the `Datastore`'s values for normalization to infer that we are p_{level} sure that the actions are within $5 \cdot \sigma$ of the actions. This is a simple calculation that can be used to create the confidence bands we can show to the user:

$$[\hat{y} - 5 \cdot \sigma(1 - p_{level}), \hat{y} + 5 \cdot \sigma(1 - p_{level})] \tag{5.13}$$

Selecting $5 \cdot \sigma$ corresponds to a p-value of $3 \cdot 10^{-7}$ and statistically covers *almost* all real results. We then add the error bars to the plots and simulate with the modified function to extract the value vector. Figure 5.30 show the results from the uncertainty estimation for simulating towards the generative model, and Figure 5.31 show for the process model. Both simulations were done on randomly initial conditions and run for 50 time steps on the PPO agent from Experiment 14.

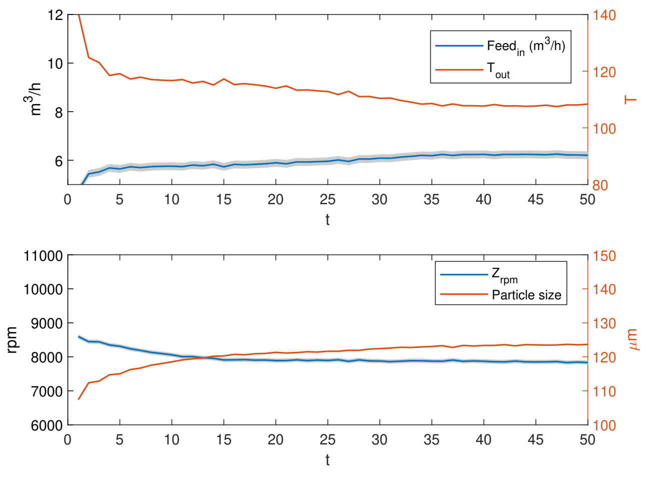


Figure 5.30: Value-based uncertainty simulated on the generative model

We can see that the process model has slightly elevated uncertainty during the initial sequence but quickly gain confidence.

[4]https://se.mathworks.com/help/reinforcement-learning/ug/train-reinforcement-learning-policy-using-custom-training.html

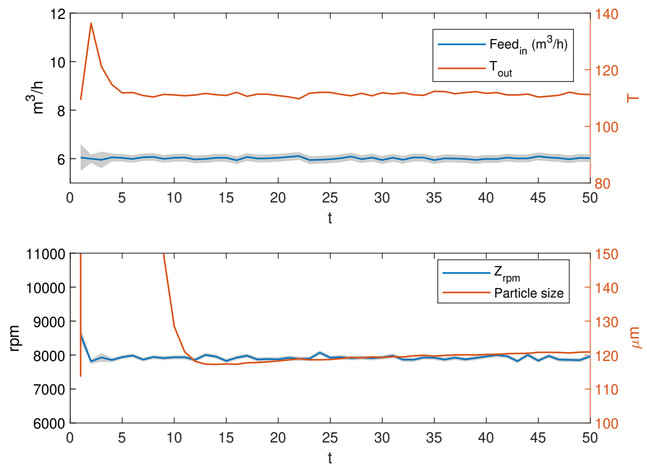


Figure 5.31: Value-based uncertainty simulated on the process model

5.6 Varying conditions

Using the p_{level} metric is a sensible way of evaluating agent performance on varying conditions. We already know that the agents are capable of generalising a policy given different instances of initial conditions. Although not shown earlier, the initial condition we see as the first observation, is in fact a sequence of n observations, depending on the selected sequence length for the training. These are not shown to the agent, which only uses the last observation of the initial sequence, but resides as part of the generative model. The state of the generative model at restart, is an unknown, or undefined, process situation within the operational envelope. The process model is somewhat linked to the generative model, by actually using this randomized initial sequence as its initial conditions.

Experiment 17: Value-based performance of RL agent on varying conditions

We can therefore assume that by restarting an agent several times, and generating random initial condition rollouts, we can evaluate an agent performance by value-based uncertainty. The evaluation is done by running 30 instances of an agent and recording the results for one sequence length period, running towards the real-time process model. However, we select only the PPO agent from the Experiment 15 for this evaluation due to its stable training performance and highest value of the three agents from Experiments 14-16.

We can see from the surface plot in Figure 5.32, that although the agent starts with a randomized initial condition, it will quickly converge in all runs and settle on an uncertainty of $\sim 4.5\%$ of $5 \cdot \sigma$.

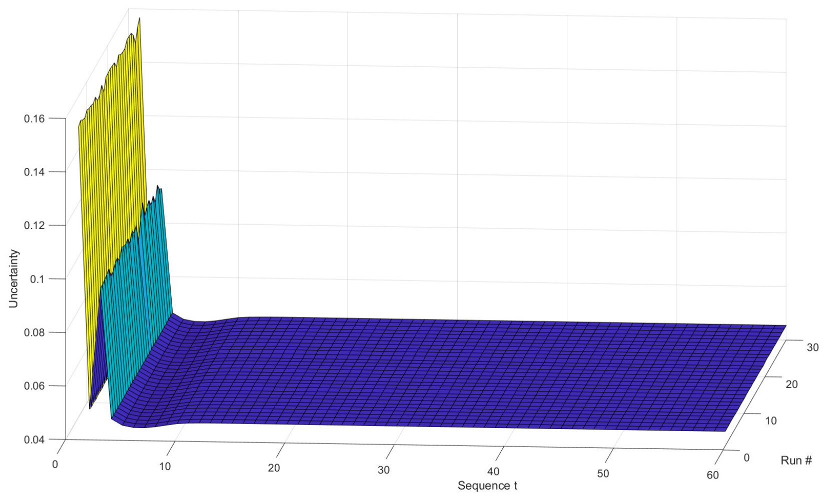


Figure 5.32: Uncertainty estimation of 30 initial sequences of a PPO agent running towards the process model

Chapter 6

Evaluation

There are two kinds of people:
1) Those who cannot extrapolate from incomplete data

6.1 Method

We have explored the area of autonomous learning by asking a set of research questions that have guided the research and experimentation process. The experiments have been conducted exploratory, where results from one experiment have dictated both the content and development of a succeeding set of experiments. As such, the experimentation has been both for evaluation of the methods and for the development of an innovative framework in this field.

Evaluating experiment performances have been done both quantitatively and qualitatively and is summarized here.

6.2 Research questions

To evaluate the research questions, we have set up a framework for compressing historical time-series by a generative model, realized through a shallow recurrent memory neural network (LSTM-based), whose parameters were set by a Bayesian optimization evaluation. This model was then used for generating succeeding time series from an initial set of data sequences. We were also able to demonstrate a technique for generating time-series for the purpose of simulating a response from this compressed model, given a set of either univariate- or multivariate manipulative variables. The generative model was realized as a Matlab Reinforcement Learning Environment, making it suitable for use in training any reinforcement learning agent supporting continuous action- and observation spaces.

To train the generative model, we first established a classic simulation model based on first principle ordinary differential equations of an industrial spray drier process where both energy- and thermal dynamics were modelled. The generative model was trained by querying time step states from these ODEs using a classic PID regulator. The generative model predictions and actual rollouts show a high correlation, as demonstrated in Chapter 4.3.4.

For all experiments, we would evaluate a $\sigma \leq 1$ as an acceptable result. The best results within each research question are written in **bold** text.

RQ 1 Can an RL Agent be efficiently trained on historical data logs?

Experiments 1-3 addressed this question and the results are summarized in Table 6.1.

Experiment	Agent type	Episodes	μ	σ
Experiment 1	SAC	868	**90.21**	1.11
Experiment 2	PPO	1801	90.40	**0.86**
Experiment 3	DDPG	598	89.92	1.03

Table 6.1: RQ 1 Results

The agents were trained on a univariate goal of attaining a process output temperature of 90℃ from a direct response reward function. As we see from the results, all agent types are able to regulate the temperature within approximately ±1℃. These results confirm that we can train all tested agents on historical data logs.

RQ 1.1 Which choice of RL Agent realizations would be a generic solver for the problem?

Experiments 14-16 addressed this question and the results are summarized in Table 6.2.

Experiment	Agent type	Episodes	μ	σ
Experiment 15	SAC	188	121.67	9.67
Experiment 14	PPO	5000*	**111.20**	**1.67**
Experiment 16	DDPG	**		

Table 6.2: RQ 1.1 Results
*Terminated with a high value **Did not converge

The agents were trained on a multivariate goal using a reward model trained for a temperature output of 110℃ and a stable atomizer speed. As we see from the table, only the PPO agent was able to produce results near the acceptable limit. Of the three agents, the increased complexity of the multivariate case was only solvable by the PPO agent.

RQ 2 Can a reward function be approximated and learned without supervision?

Experiments 10-12 addressed this question and the results are summarized in Table 6.3.

Experiment	Agent type	Episodes	μ	σ
Experiment 11	SAC	*		
Experiment 10	PPO	502	**89.95**	**0.24**
Experiment 12	DDPG	*		

Table 6.3: RQ 2 Results
*Did not converge

The agents were trained on a multivariate goal using a reward model trained for a temperature output of 90℃ and a stable atomizer speed. As we see from the table, only the PPO agent was able to produce results, and in range.

RQ 2.1 What are the implications of trained reward-functions: will they behave differently given the state of the system?

Experiment 17 addressed this question using the trained PPO agent from Experiment 14. As we see from Figure 6.1, the PPO agent converged to a policy with a mean stable uncertainty after less than 10 time steps, at an uncertainty of $\sim 4.5\%$ of $5 \cdot \sigma$, when repeating the experiment 30 times. We conclude that using trained reward functions are viable alternatives due to the good performance demonstrated upon changing the system state. The agent is able to bring the system state into equilibrium from any of the random initial conditions.

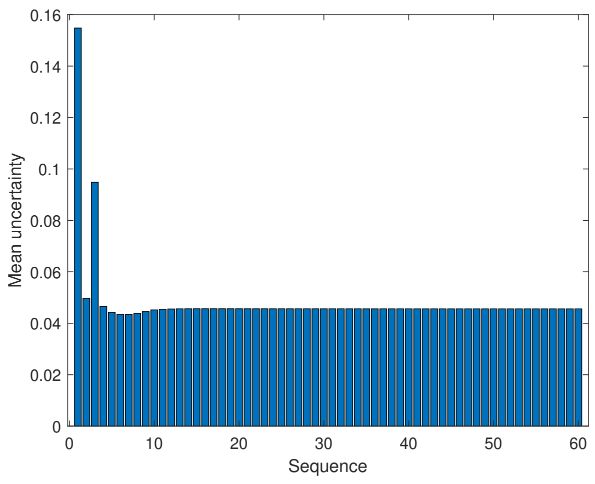


Figure 6.1: Mean uncertainty estimation of 30 initial sequences of a PPO agent running towards the process model

6.2.1 Limitations

The generative model was trained by a synthetic model where stochasticity was added at several levels. One limitation of the experiments is their inability to describe the performance of the agents when used on real-world data. However, we do believe that the synthetic model is a comparable descriptor of real-world processes, based on the available literature. Using data from a real-world process involves data quality engineering and is briefly discussed in the next chapter.

Further, we have only experimented with time-series data. There are no spatial data involved, which could leverage the performance of the agents. However, adding measurements that are *relevant* should positively increase the performance further.

6.2.2 Advantages

The resulting framework used for experimentation has been developed in such a way that fosters further expansion, compatible with the Machine Learning and Reinforcement Learning toolboxes in Matlab. We also see the disruptive power of using reinforcement learning within process control optimization.

6.3 Building experience

All the aforementioned experiments are conducted with specific termination criteria based on the average reward and total number of episodes. However, we see from the results of Experiments 14-16 that the PPO agent has reached a policy that has better performance than the SAC agent. The PPO agent terminated on a total episode count (5000), while the SAC agent terminated based on the average reward criteria after 188 episodes.

From this result, it is evident that training RL agents cannot be solely based on the reward criteria, but should be done by evaluating the *value function* result in addition. The value function is a slower moving target but is our actual target that should be evaluated in training.

This observation makes sense, since training an RL agent is not only about finding *a policy*, but for making a robust agent, we want to find *multiple policies* satisfying the goal for a *high future expected reward* (the value) so that the agent builds experience that can be used on different situations in the real world.

We will use this insight to construct agents in the next chapter that satisfies this condition.

Chapter 7

Applied Use

Many are stubborn in pursuit of the path they have chosen, few in pursuit of the goal.[1]

7.1 Introduction

In Chapters 4 and 5, we constructed a framework for autonomous learning and control and demonstrated that both PPO and SAC agents can gain experience at such a level that enables data-driven control- and optimization operations. The DDPG agent showed less performance in this respect. The key to proper agent training is a generative model that has learnt *enough* dynamics of the system in question, and that forecasting is possible to a necessary degree. We also demonstrated the ability to train for sparse rewards, but with an indication that the agent needs *directional hints* to keep training performance high. We were not able to conclude on a pure sparse reward scheme.

We have demonstrated the concept using synthetic data from a set of ordinary differential equations, simulating an industrial spray drier using mass- and energy balance calculations. In this chapter, we will use a direct response reward scheme and train RL agents on data originating from a commercial spray drier to validate the concept in the real world.

Using real-world data needs careful considerations of the sensors involved, measurement confidence and filtering data from periods that we do not want to model. Equipment can run several products and recipes, as well as maintenance periods, cleaning-in-place periods, inactive downtime etc., and depending on our application, data quality considerations must be made in each specific case. However, as this chapter show, modelling dynamics in a generative model actually relies on inputting *as much dynamics as possible.* In contrast to linear modelling, like OLS and PLS that relies on filtering away unwanted dynamics, the success of the generative model relies on our ability to learn from periods of great interaction between the model variables, e.g during equipment startup- and shutdown periods or where the processing conditions change. Filtering away this data leads to models of less knowledge and predictive ability. We were also able to identify a method for *zero-vector bias estimation* that can be a method to evaluate what actually *enough* dynamics is.

[1]German philosopher Friedrich Nietzsche (1844-1900)

7.2 Architecture

Figure 7.1 show an overview of the complete system architecture. Training the generative model has earlier been done using the `RolloutDatastore` connected to a synthetic ODE model, regulated by PID. This generative model is then the simulator for a `SyntheticEnvironment` class using the rollout mechanism described in Chapter 4. A compatible RL agent uses this environment for training a policy. New in this architecture is that we use the policy to generate future actions and responding variable forecasts, using updated real-time data, that is, purely new data from the process.

Our success criteria are to be able to propose an action that will lead the process from its current state on a path towards a T_{out} of 100℃. We know that the process reacts quickly to manipulations, indicating that the agent should learn a policy with equal reaction time.

There are few changes necessary to enable usage of real data, mainly developing a new class deriving from the `matlab.io.Datastore`, sharing the interface of the datastore. We develop a new class `HistorianDatastore` that enables us to read historical process data directly from a plant, in this case using the commercial *OSISoft PI Web API*.[2] The PI system and API is in operation at an industrial pulping site, enabling historical access to all process data on-site.

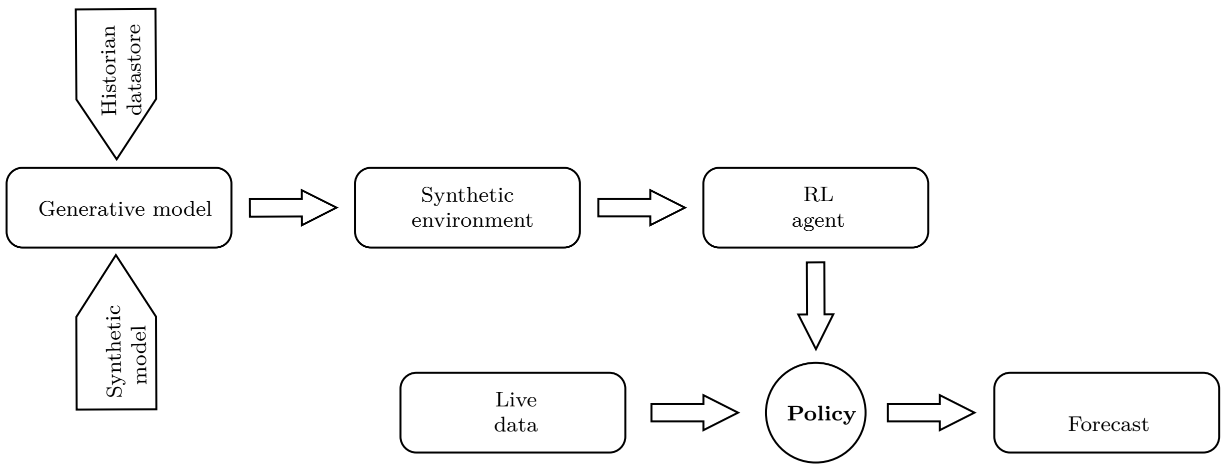


Figure 7.1: System architecture for training and executing agent policy

7.2.1 HistorianDatastore design

A generic class `HistorianDatastore` is developed and can be a framework for connection to any source of time-series data, either flat files or queriable tabular databases. In this particular case, we develop a connector to the *OSISoft PI Web API* which let us query the time-series database for tags, time periods and method of interpolation interval. In the case of *PI*, time-series are stored in a compressed format where each query to the API reconstructs the data to the time-scale in the query.

We also provide a filter function to the datastore, that contains simple rules for data inclusion. The datastore queries all data first, and then filter away the data by the filter

[2]https://docs.osisoft.com/bundle/pi-web-api-reference

function. In a *big data world*, this method is exhaustive and not particularly fine-tuned for use cases with more data than can fit in memory, and should be technically enhanced if the need arise.

Instantiating the datastore is done by:

```
source_interval = '1m';
resample_interval = 1; % minute
pidatastore = HistorianDatastore(sequence_length, @tag_function, ...
    @data_filter_function, start_date, end_date, source_interval, ...
    resample_interval, '00:01:00');
```

where we supply the generative model sequence length, tags- and filter function pointers, as well as the start date and end date. The interval is supplied to the API for data reconstruction. The last two parameters control the internal resample interval and the threshold for identifying gaps in time.

When data is filtered, we have gaps in the time series. To avoid merging incompatible time-series sequences, we identify these gaps as borders for establishing sequences as illustrated by data queried in Figure 7.4. We ensure that each sequence length is twice the requested length for training so that we have space for the *sliding window* mechanism described earlier. Sequences that are too short to satisfy this condition, are discarded. The last parameter to the class constructor is our gap threshold time, namely how long a period of missing data in the source is treated as a border for the gap calculation. Missing data below this threshold are interpolated using the `resample_interval` numerical parameter. By splitting the data into sequences, we can divide the data source into shuffable blocks of training-, test- and validation data without losing the time-dependency of the data.

From this point on, data sequences are provided in a *sequence-to-one* forecasting method, as described and implemented in Chapter 4. Whenever the datastore user issues a `read` request on the datastore, the datastore provides the next *data window* available as a sequence, and the next data vector. Subsequent requests of data will be returned as a slid set of sequences.

As the last processing step, 0.1% noise is added to all signals for each `read` request. During training, all data will be visited for each epoch. By adding noise, we incorporate the knowledge discussed in Chapter 2.3.1, effectively ensuring a data augmentation scheme on every epoch.

7.3 System design

7.3.1 System boundary

When selecting the system boundary, we need to ask ourselves which potential influencers of the observable dynamics a system might have, based on knowledge of the physical system and available literature. Then, we need to identify what can be measured to estimate the state of these dynamics. We have earlier described the relation between Q_{feed} and T_{out}, and also how the temperature of the inflow gas influences the process. We have seen that even the environmental temperature and humidity affect the dynamics.

Figure 7.2 show our defined system boundary for the real-world model. This is an over-simplified version of an actual plant, which involves more equipment, systems and process measurements. However, for our case, this simplification is beneficial due to its simplicity of explanation, and still presumably a high level of dynamics capture.

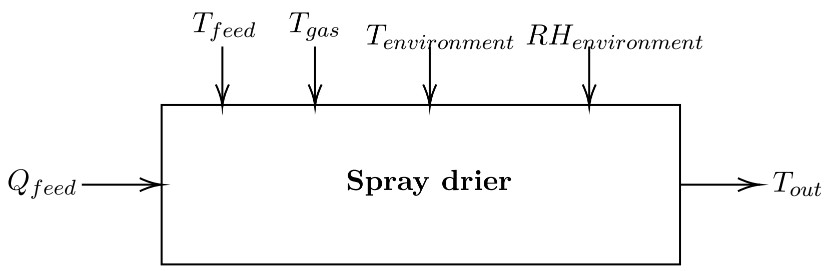


Figure 7.2: System boundary for the real world model

We have included only environmental measurements as controlled variables. This is related to the fact that the process is mostly modelled using energy- and mass balances, dynamics that are observable in these measurements. The dataset used does not reveal any information regarding the products themselves, their recipes or any disclosed IP information.

7.3.2 Data exploration

First, we select process tags that correspond to the system boundary variables and extract data for a period of seven months with 10-minute intervals. In Figure 7.3, we see a cloud plot of the data distributions and correlation between variables. It is not easy to identify any clear correlations from this plot, however, we see that data distributions are *mostly normally distributed* with different levels of skewness and kurtosis. The requirement of normality is important in machine learning. Unbalanced datasets need more treatment, commonly by log-scale transforms or oversampling the data to create a balance between the predictors.

We have included all operational data from the period, making visual identification of any correlations tricky, since we might assume that variables are highly non-linear during startup- and shutdown periods. The only filtering we do is removing all data where the equipment has been at a complete standstill, that is, we filter away data where $Q_{feed} < 1m^3/h$. We have not removed periods of cleaning or when the system runs without product, e.g. only water spray. We see that we have a dataset that describes all areas of the data distributions.

In Figure 7.4, we see the complete dataset in a sequence plot. Using the filtering criteria of $Q_{feed} < 1m^3/h$, some gaps in the data are identified and used as sequence delimiters for the datastore. The plot visualises the input that the `HistorianDatastore` has calculated and uses as input to generative model training. Before training, dataset mean and standard deviation is calculated and stored for normalization of training- and real-time test data, using the same method as for the synthetic model. After the processing steps, we have a dataset of 27,693 sample vectors, distributed on 52 different time-series sequences.

7.3.3 Generative model training

We use the same parameters for the generative model as earlier, but reduce the number of epochs to 60. We are looping all data for each epoch, in contrast to the selectable rollout

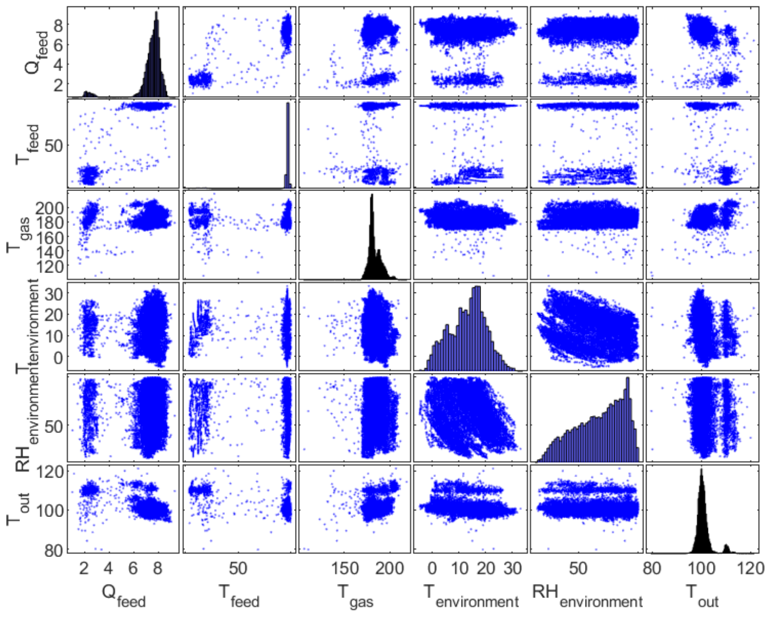


Figure 7.3: Cloud plot of data distributions and variable correlations

count when using the `RolloutDatastore`, and therefore we limit the time used at each learning rate step.

When tweaking the network architecture, the Matlab documentation takes an interesting view on how to select the number of neurons in an LSTM network for dynamics capture:

> *The number of hidden units required for modelling a system is related to how long the dynamics take to damp out. [...] there are two distinct parts to the response: a high-frequency response and a low-frequency response. A higher number of hidden units are required to capture the low-frequency response. If a lower number of units are selected the high-frequency response is still modelled. However, the estimation of the low-frequency response deteriorates.*[3]

This view corresponds with our experienced beliefs during work on the experiments: we try to minimize the number of layers and neurons to such a level that the dynamics are still captured. We do this in order to generalize the model. Increasing the number of neurons results in a network where more parameters must be learned and should be compensated with either more available data, or by increasing the number of epochs. By smoothing input data, the high-frequency dynamics is lost, and part of the lower frequencies might also be filtered. It is advisable to train the network using raw data sequences without any data smoothing contrary to what we would apply to linear system modelling. However, as noted by Thodoroff et al. [89], smoothing reduces variance and helps the learning process. We believe that the amount of smoothing necessary must be tested to suit the data available.

In Figure 7.5 we see the result of a one-step forecast for real-world data, estimated by a trained generative model. The data has been adjusted for bias found using *zero-vector bias*

[3]https://se.mathworks.com/help/ident/ug/use-lstm-for-linear-system-identification.html

estimation, as we will discuss in the next section. Our impression is that the generative model provides estimates in lieu of system dynamics.

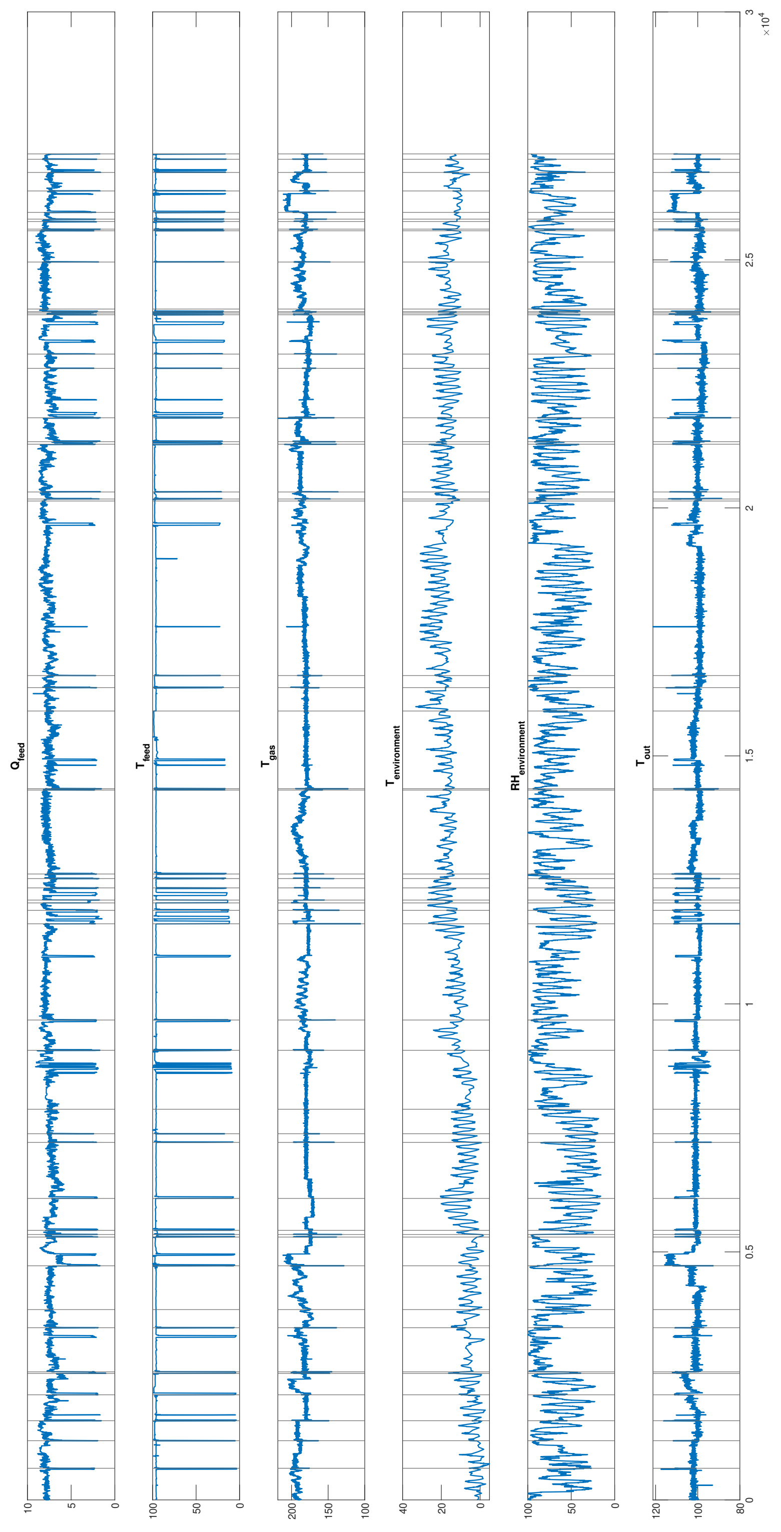


Figure 7.4: Dataset sequences for generative model training

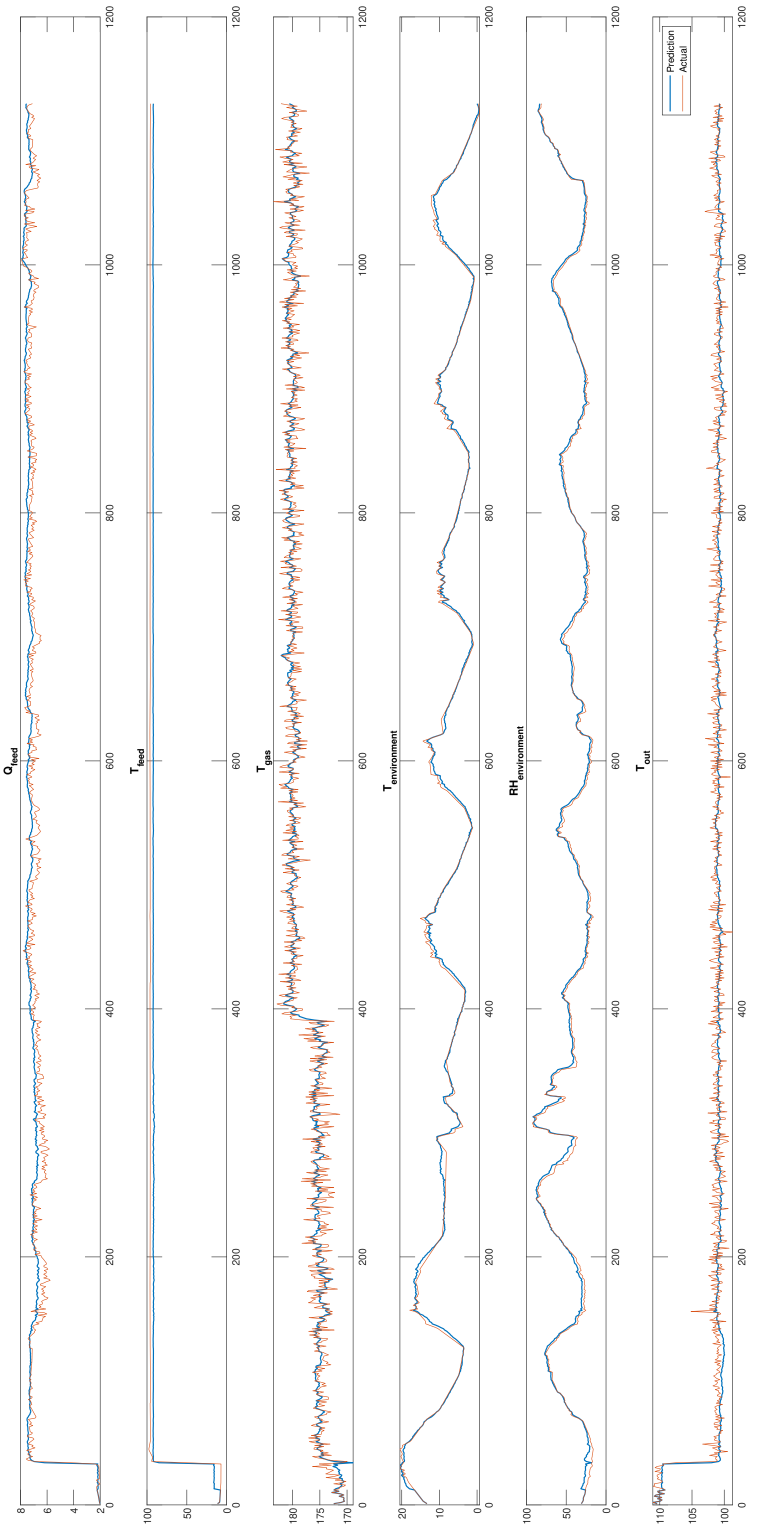


Figure 7.5: One-step forecast from the generative model for real world data

7.3.4 Zero-vector bias estimation

Our intuition tells us that when supplying a vector of zero-values to an autoencoder, we should receive an estimate of zero. Any deviation from zero would be the network error, given that the training has crossed the zero centre vector. By predicting on a network trained by normalized values, any deviation from the population mean and standard deviation is represented as positive or negative scores, indicating that the middle value of a network is zero. This assumption would indicate that a zero-vector estimate is directly in the centre of the neural network.

In a recurrent network scenario, we can use the subsequent estimation of zero-vectors to initialize a model correctly and record the baseline zero-vector estimation error, an idea taken from the Matlab documentation.[4] By non-formal experimentation, there seems to be a two-way-relation between a network's ability to forecast dynamics, and the dynamics of the zero-vector estimation. These relations could be used to automatically evaluate the generative model's predictive power, and correct each prediction before de-normalization:

$$\hat{\boldsymbol{y}} = N(\boldsymbol{x}) - \boldsymbol{b} \tag{7.1}$$

where $\boldsymbol{b}$ is the recorded bias vector, and N is the network prediction function. We assume the following:

- A shorter stabilization time indicates more accurate dynamics capture
- A small zero-vector error indicates more accurate dynamics capture

Whether these assumptions hold, is an interesting future work. However, initial testing shows positive results and we apply this method to evaluate the selection of sequence length of the generative model. In a multivariate scenario, like the autoencoder, the RMSE does not vary much in favour of selecting the sequence length, as seen in our initial experiments and the Bayesian optimization run in Figure 4.10. By evaluating the zero-vector bias, we might have another evaluation criteria for our generative model.

To test these assumptions, we train two generative models using two different sequence lengths: 3 and 12. Each model is then initialized with zero vectors in a loop of 150 steps according to Code 7.1.

```matlab
% Find the zeromapping where the network is stable
results = [];
for steps=1:150
    initializationSignal = zeros(genmodel.datastore.featureDimension, ...
        sequence_length*steps);
    genmodel.net = resetState(genmodel.net);
    genmodel.net = predictAndUpdateState(genmodel.net, initializationSignal);
    zeroMapping_mu = mean(predict(genmodel.net,initializationSignal));
    results = [results; zeroMapping_mu];
end
```

Listing 7.1: Calculating zero-vector bias (Matlab code)

Each loop predicts n number of zero-vectors of length *sequence length*. We then plot the mean of the `results` vector and visually evaluate based on when the mean differentiated signal is approaching infinity, which indicates that the base signal, the bias, is stable. Based

[4]https://se.mathworks.com/help/ident/ug/use-lstm-for-linear-system-identification.html

on empiri, we assume this step number should be as low as possible, i.e. we assume that a shorter stabilization time indicates more accurate dynamics knowledge. Figure 7.6(b) and 7.7(b) show the dynamics of the tests. Figure 7.6(a) and 7.7(a) is the bias vector from the first loop, and we see that the model with the lowest sequence length has one decade more error than the model of sequence length 12.

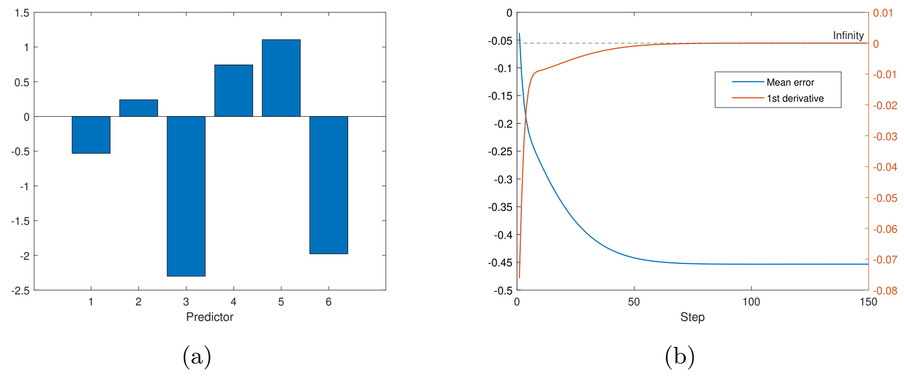


Figure 7.6: Zero-vector bias estimation for sequence length of 3

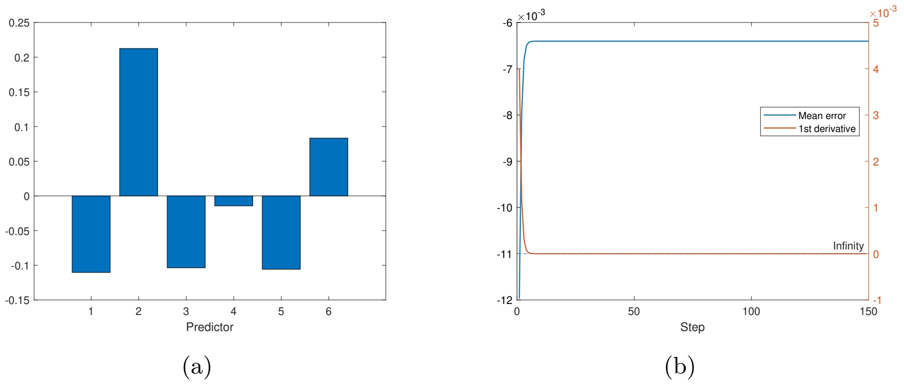


Figure 7.7: Zero-vector bias estimation for sequence length of 12

Our comparison of the two schemes reveals that a generative model with a short sequence length has limited forecasting capability if we follow our assumptions above. Based on our knowledge of the data and system, this observation strengthens our assumptions. We select 12 as the sequence length for the further tests.

7.3.5 Evaluating forecasting capability

We can evaluate the captured dynamics by forecasting beyond a horizon of one time step. Ideally, we would want to be able to simulate all dynamics indefinitely. This would mean that the generative model is an exact replica of the system dynamics. In reality, we must expect to see lower performance when working on real-world stochastic systems with unknown state estimators and non-linear noise and relations.

We believe that the experiments done earlier using the synthetic model has forecasting ability beyond one step, given the positive results from agent testing. Evaluating the generative model's forecasting capability is an important input to the considerating of the reward function and for how long we can expect the agent to calculate the future expected reward in agent training.

To evaluate the dynamics captured by the generative model, we randomly select a sequence from the datastore and test the forecasting for steps 1 to 6. That means we forecast every datapoint 1 step, 2 steps, 3 steps etc. and compare with the original sequence data. The deviation will give us information on what to expect from the generative model using the selected sequence length.

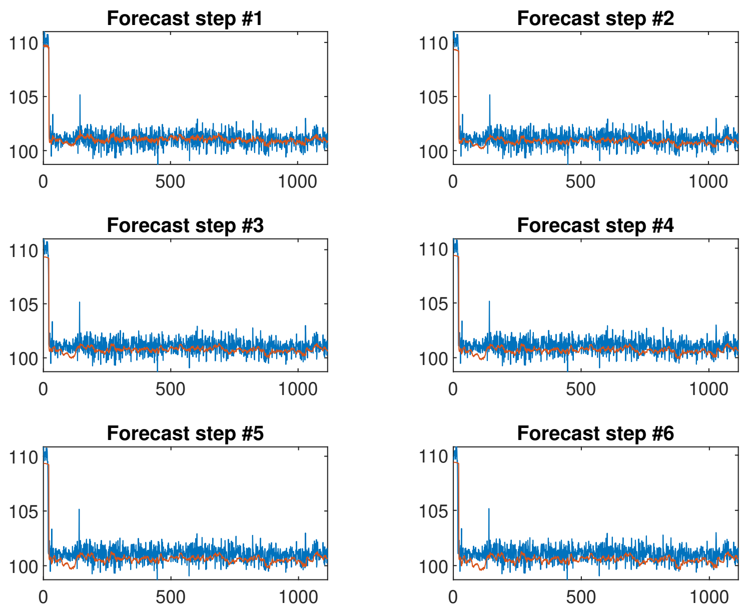


Figure 7.8: Generative model forecasting of steps 1-6, original data (blue) and forecast (red)

In Figure 7.8 we see six forecasts of about 1,100 data points, where the forecast (red) is overlaid the original data. We can see that the generative model starts to deviate from step 3 onwards. For most of the dataset, the forecast is within the expected range, and the deviation could originate from a combination of unknown dynamics, lack of data, lack of training or inability to model. Figure 7.9 summarizes this run, and we see that after 6 steps, the correlation between the test set and the forecast is about $r^2 \sim 0.76$, with a linear descending trend. This means that we enter a negative correlation after approximately 30 steps, but to have any forecasting power, we would like to set a threshold of $r^2 \geq 0.80$. This means that we can expect on average that the model can forecast 4 steps ahead with known correlation. This would imply that the applicable use cases for the spray drier generative model should be limited to reward schemes utilizing four time steps lookahead horizon.

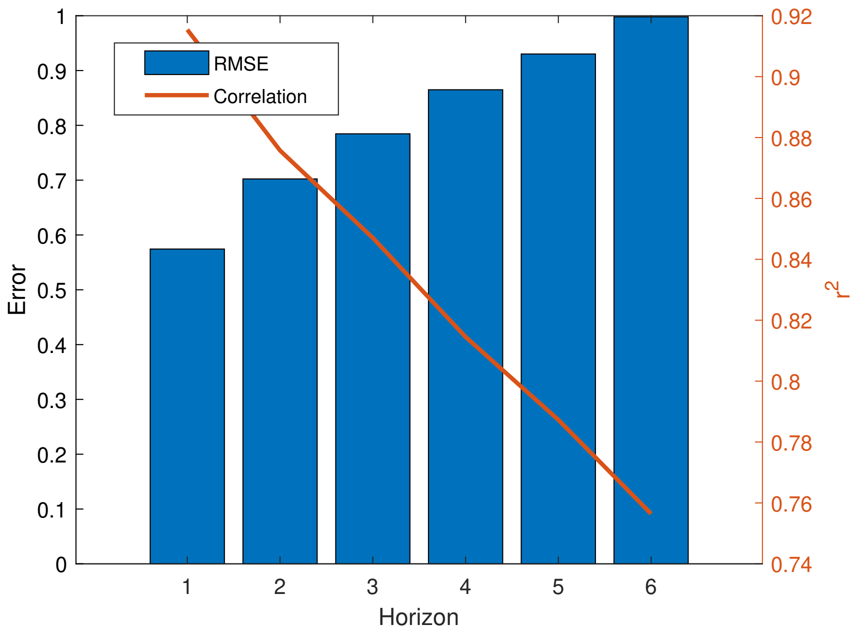


Figure 7.9: Generative model forecasting of steps 1-6, recorded RMSE and correlation

7.4 Training RL agents

Two agents, a SAC and a PPO, are trained with the generative model as the synthetic environment. We apply a direct response reward using the inverse absolute error as reward:

$$\begin{aligned} \Delta E &= E_{estimate} - E_{actual} \\ r &= \frac{1}{|\Delta E|} \end{aligned} \tag{7.2}$$

and a termination critera of $T(\Delta E)$:

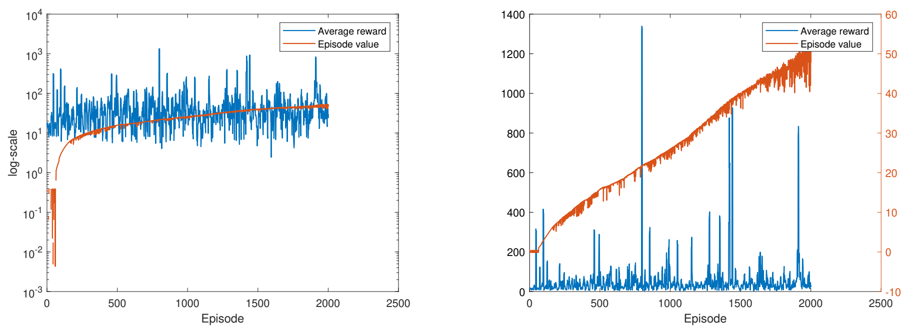
$$T(\Delta E) = \begin{cases} 1 & \text{for } \Delta E \ > \delta, \\ 0 & \text{otherwise.} \end{cases} \tag{7.3}$$

where $\delta = 1$. The termination criteria is evaluated when $i > s$, where i is the current step number and s the sequence length (12). We do this in order for the agent to have time for policy evaluation and building before a strict termination criteria occurs.

7.4.1 Soft Actor-Critic agent

A SAC agent was trained for 2000 episodes using manual termination due to resource quota constraints. We can see from Figure 7.10 that the value growth indicates steady knowledge gain from the generative model. We also see that the value fluctuates when approaching termination, indicating that the agent might have reached a global error minima.

Table 7.1 summarizes the results, and we can see that the policy on average can provide accurate action prediction for 2.1 steps beyond the sequence length of 12. Considering the



(a) (b)

Figure 7.10: SAC agent training progress on (a) log scale and (b) normal scale

above evaluation of the generative model forecasting ability, we would expect the agent to forecast up to 16 steps.

Agent type	Episodes	Max value	Reward μ	Steps μ	Max Steps
SAC	2000	53.62	56.97	14.1	377

Table 7.1: SAC training and simulation results

7.4.2 Proximal Policy Optimization agent

A PPO agent was trained for 5000 episodes using manual termination. In comparison to the SAC agent, there is a substantial difference in computation time between the two agent types, where the PPO requires less resources. However, from our experience, the PPO agent tends to *lock* itself into a minima. Restarting the training is often necessary in cases where the agent cannot identify any policy.

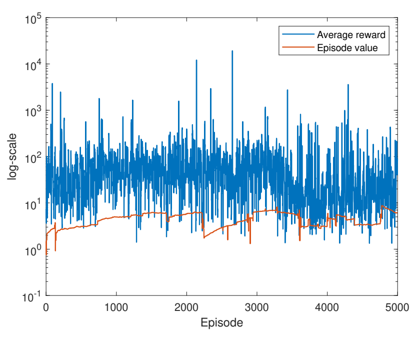
We can see from Figure 7.11 that the value cycles from *valley-to-valley* as described earlier for the PPO agent. In the end, the maximum gained value is less than that for the SAC agent. However, in Table 7.2 we see that the average number of steps has reached 16, indicating that the PPO agent is able to be trained for the maximum forecast of four steps beyond the grace period of 12 steps.

Agent type	Episodes	Max value	Reward μ	Steps μ	Max Steps
PPO	5000	8.76	124.37	16.1	209

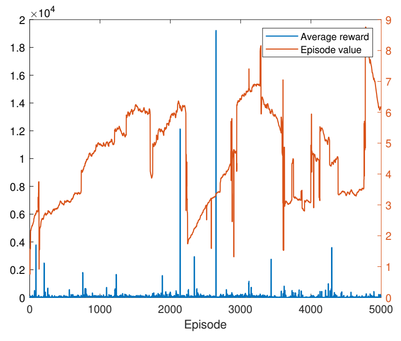
Table 7.2: PPO training and simulation results

7.4.3 Simulating with real-time data

Each policy is simulated with real time data from the process. Another instance of the `HistorianDatastore` class fetches the latest process data and simulate with the `sim_confident` function, utilizing the policy state value as an estimation of uncertainty. We observe from several runs that both agents successfully proposes similar action manipulation



(a)



(b)

Figure 7.11: PPO agent training progress on (a) log scale and (b) normal scale

for Q_{feed} for the process to reach a state toward $T_{out} \sim 100$°C, as exemplified by Figures 7.12 and 7.13.

However, even though the SAC agent completed training with a higher maximum value, the confidence interval for the action manipulation is narrower than that of the PPO. This would indicate that the SAC agent is more confident that the proposed action will lead the process in the correct direction.

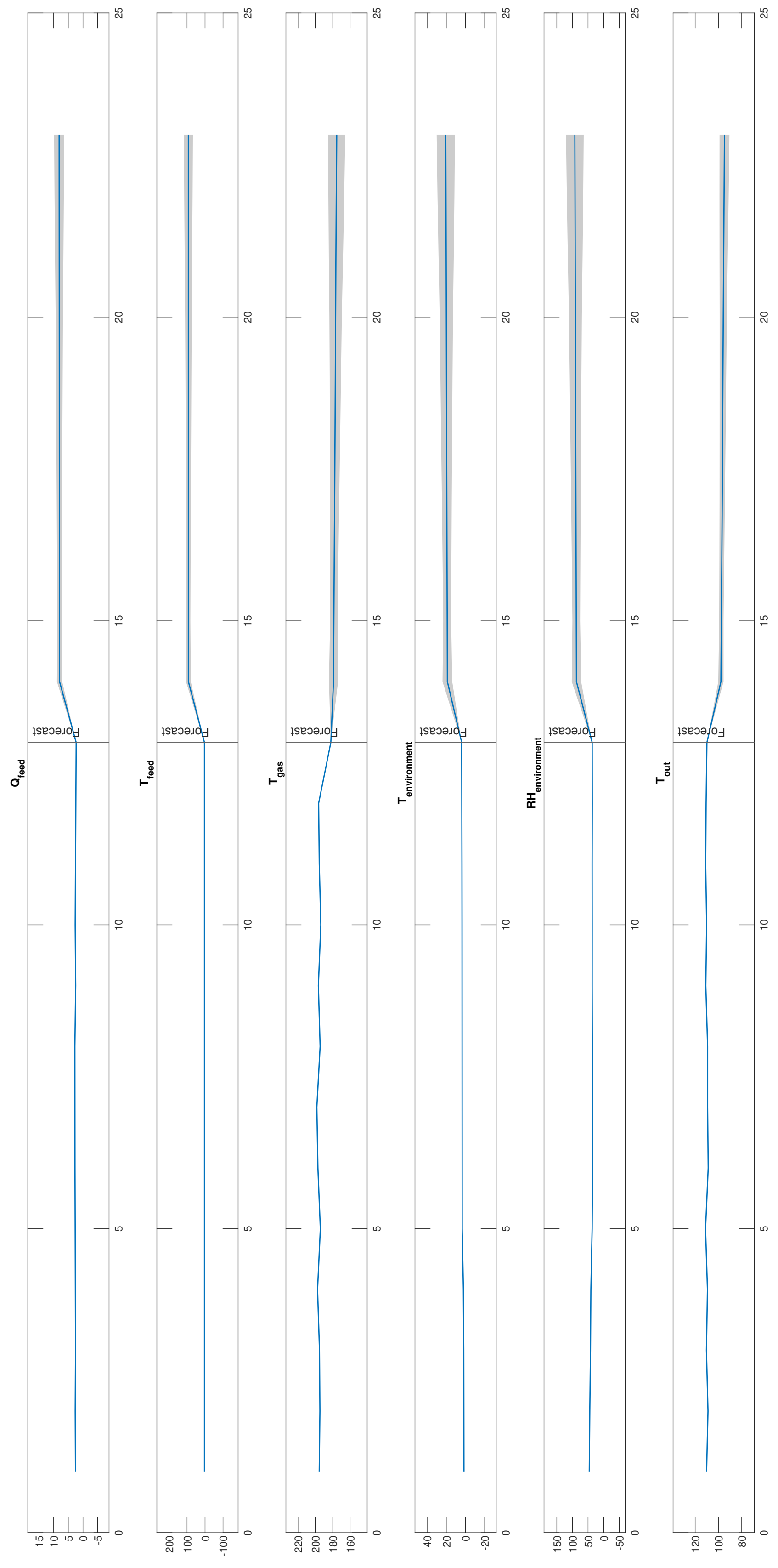


Figure 7.12: Real time simulation with SAC agent

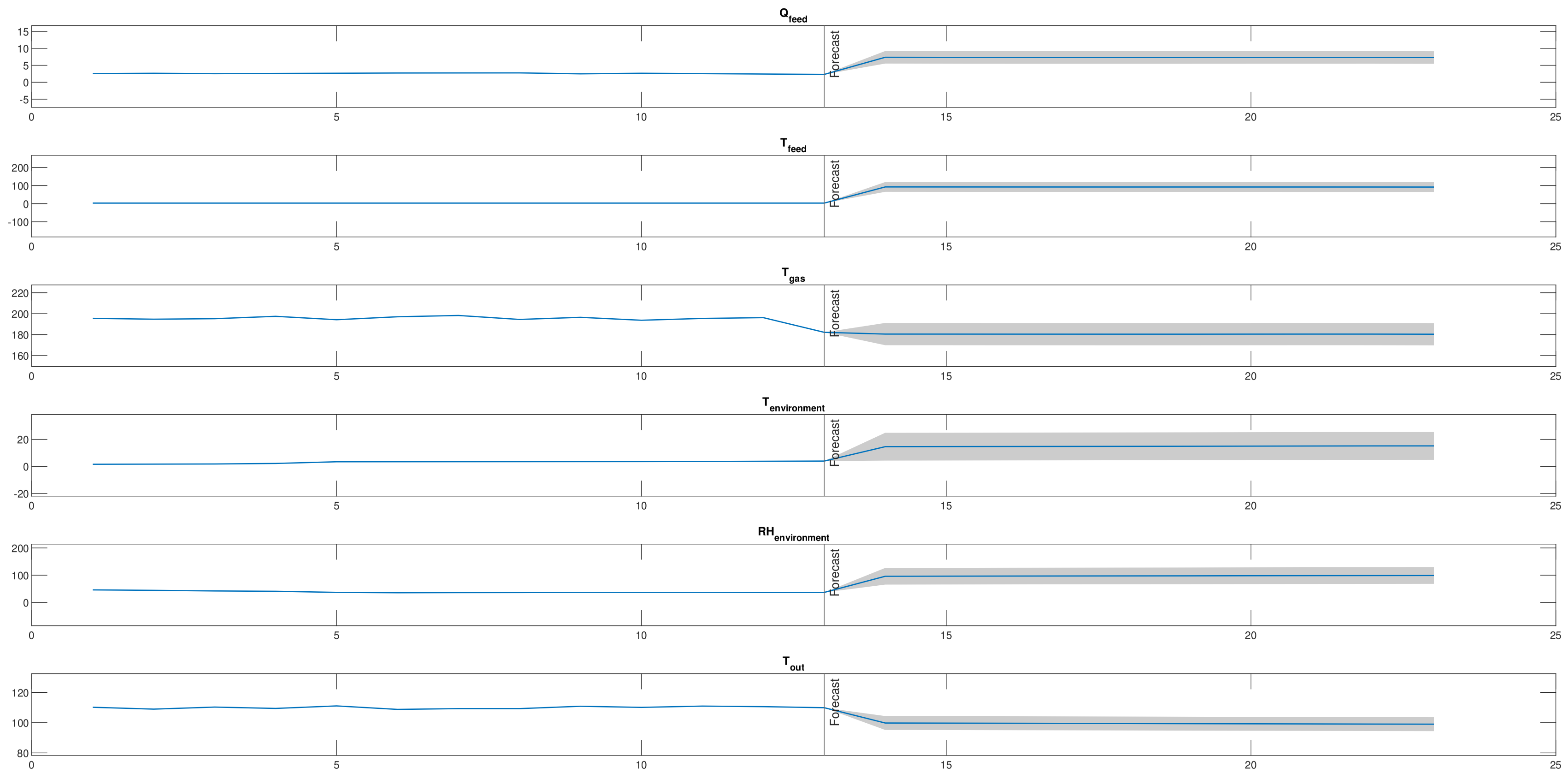


Figure 7.13: Real time simulation with PPO agent

7.5 Evaluating simulation runs

Figures 7.12 and 7.13 visualises one random simulation run with confidence interval for both agents on the latest real-world data. The current state of the agents at this point is that they are only trained by the generative model. A natural extension is to further train the agents by applying them to the real process and update their experience buffers and policies accordingly, a task not covered in this report. Petsagkourakis, Sandoval and Bradford et al. [110] mentions *transfer learning* of the weights of the agents to accommodate for the real system. However, in our case, we have trained the agents not on a stoichiometric model, but by the original process data itself, through the generative model. The agents should behave within the same scale of numbers as the real process, so a particular process of transfer learning should not be necessary.

It is not possible for us to explicitly evaluate the performance of the agents. As we have identified earlier, the real world training set we use has limited forecasting ability, probably due to a more stochastic process and unmodeled dynamics than what we experienced using the process simulator. We must also remember that no pre-processing and data quality engineering of the measurement variables is done. We expect that the agents at this point only can recommend a *direction* of the manipulated variables due to this limitation, but as the agents evolve on a real system, they will behave more in lieu of the reward scheme. Evolving the agents is a task for further work.

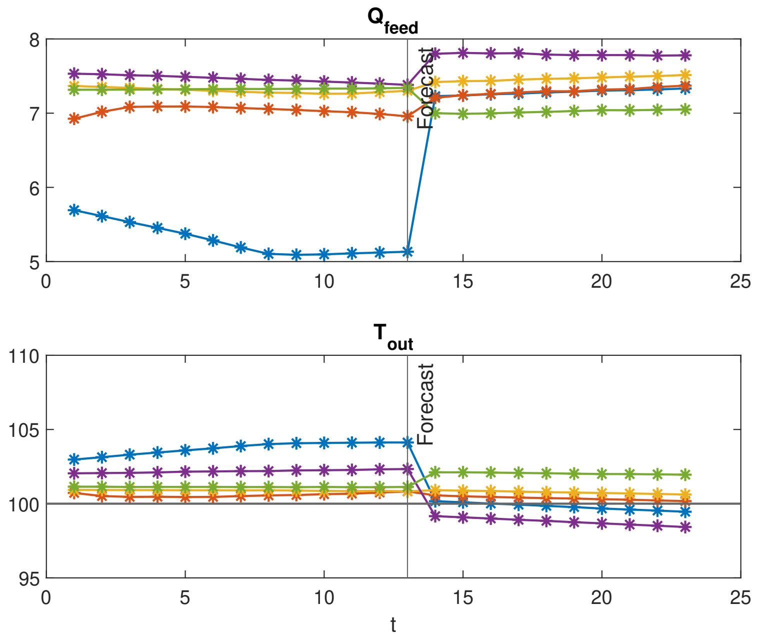


Figure 7.14: Five simulation runs for the PPO agent towards setpoint of 100°C

We can see this effect by running the agents multiple times, where each run is initialized with a new set of initial conditions from the process data. Data from both agents are visualised in Figure 7.14 and 7.15. The SAC agent seems to be more aligned towards the setpoint than the PPO agent, even though the number of training episodes is less than a half (2000 vs. 5000).

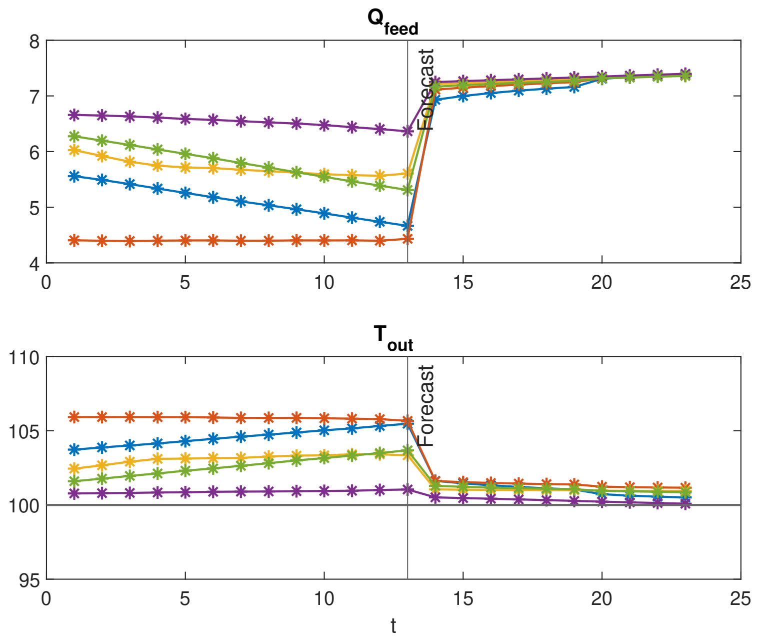


Figure 7.15: Five simulation runs for the SAC agent towards setpoint of 100℃

We know from empiri covered by Santos et al. [88] discussed in Chapter 3.2, that increasing the *feed rate* will lower the outlet temperature. By visual inspection, both agents comply with this evaluation requirement. They are able to propose actions that will guide the process towards the setpoint, although we see signs of policies that are not accurate enough for finer control at this point of agent experience level.

Chapter 8

Discussion

Once there is the slightest suggestion of combinational possibilities on the board, look for unusual moves. Apart from making your play creative and interesting, it will help you get better results.[1]

8.1 Introduction

Training neural networks for dynamics crosses the lines between science, art and philosophy. We have yet to invent appropriate methods to design network architectures with a prescriptive quantitative performance. Much of the selection of choices depends on a *trial-and-error* approach to designing a system, and we have no performance guarantee until we have calculated a set of data on a selected architecture. Still, we cannot be sure on the future outcome. Although we *think and act* mathematically in the context of neural networks, can we be sure that we are operating within the fundamental mathematical domain, or that we, due to the randomness and stochasticity involved, observe and act based on universal randomness? The *anthropic probability principle* [50] can be applied to the problem domain: by setting a lower bound on the statistical probability of the outcome, we can deduct that our observations of the performance are within the pre-described world of neural networks. This statement implies that our *trial-and-error* results are within context and that we select amongst possible outcomes already part of the population. Any choice of architecture will give results within our range of presumable outcomes if we follow this principle.

The analogy is a chess player that acts according to a rule set on a chessboard. In comparison with neural networks, the board and the chess rules are our architectural opportunity space. The player has seemingly endless options of winning the game and must employ either a mathematical view, use earlier training and intuition, or a combination of both. His or her actions have a tint of strategy involved, adjusted by the level of the opponent, or a higher goal in the tournament. Still, the concept of the game entails the universe to explore.

In the case of reinforcement learning agents, assuring the outcome can only be done by exploring the known universe: hence, from a philosophical standpoint, we should be able to select the best solution if we explore the whole environment. In practice, this means that the more we explore, the more experience we capture. Reaching all parts of the

[1]Russian chess player and author Alexander Kotov (1913-1981)

environment is not always achievable, mostly from a practical and computational view and is a founding principle to why our agents implement different strategies to explore vital parts of the environment, as early as possible. Our problem lies in the nature of *perception bias*: at one point we *believe* the environment is fully discovered, whereas we indeed have not covered the parts we do not see. This masking of data heavily influences science and often lead to misconceptions. In the *survivorship dilemma*, our occluded observation of the world also makes us falsely conclude based on the set of data we can see. In a population, weak organisms do not survive, and by observing only the strongest and most prominent remaining samples, we risk falsely making assumptions of the past. In a neural network, weak parts will diminish in the same way, leaving the survivors for us to observe.

8.2 Contribution

Can reinforcement learning be applied as a general optimization concept and trained on historical logs of data? In Chapter 6, we evaluated the research questions and draw conclusions that support our hypothesis. We also explored the necessary techniques to develop a framework that can be implemented in real-world processes based on time-series data. By training a shallow recurrent LSTM autoencoder network on historical time-series data, we can reconstruct the dynamics of a system in a forecasting scenario. We have also tested methods to evaluate the performance of such a model and expanded the typical *RMSE* calculations with the *zero vector estimate* toolset and showcased techniques for forecasting evaluation. With these methods, we have power tools at hand to make informed decisions on the capabilities of a generative model to support a specific optimization scenario.

We call the contribution a *generative model.* This is a compressed model of a system and is an efficient way of exchanging, storing and simulating system dynamics. Being *generative* implies that the model generates data that in lieu with system dynamics, albeit not necessarily a copy of the data it has seen before. We have shown that by manipulating the network, we are able to produce new data that describes the dynamic response of a system given the manipulation. This generative model can sufficiently train a multivariate reinforcement learning agent and respond to action proposals towards a new and higher goal set. The agent reward function has also been shown to be trainable by a static neural network.

Pitfalls have been identified and validated, the most prominent is that data-driven models declines in performance when extrapolating to unknown state-spaces, also reported by Del Rio-Chanona et al. [91] and others. Our experiments support the observation of this phenomenon, and we have learned that the generative model should be exposed to varying system conditions, also those resulting in faulty states or bad outcomes. Learning the generative agent on successful time-series only will lead to a model with less knowledge of the dynamics.

One might further raise the question of why we detour training an RL agent on a trained dynamics model, instead of jumping directly to direct training of the RL agent policy by *representation learing*, as demonstrated by Powell et al. [111]. This is a viable question, raised by the fact that the generative model learning introduces noise and biases, which is then amplified by agent training. However, to deal with the fact that the agent strategy might be different than what we have data available on, we cannot divert to representation learning only. If we have a specific optimization goal where the data logs do not represent this goal, we cannot train the policy directly. Hence, we must develop a reward scheme that satisfies the optimization goal and simulate the environment, demonstrated by our generative model.

8.3 Agent selection

Of the three agents we chose to experiment with, PPO and SAC have shown to be trainable within the problem domain and specified termination. The DDPG agents did not converge to solutions within the criterion we set. The agent might need longer training time than we catered for, and our longer model dynamics might be incompatible with the stochastic action exploration scheme in DDPG. We can neither conclude on which of PPO and SAC are most suitable since they exhibit different performances given different training scenarios. In general, it should be advisable to train both agent types when working on a problem, to compare their performance.

To verify this conclusion, we have trained each of the three agents on a set of known transition functions for a propotional cascade control of two processes. This setup is a typical use of industrial PID control where an outer loop controls the process and a faster inner loop achieve rejection of disturbances, as visualised in Figure 8.1.

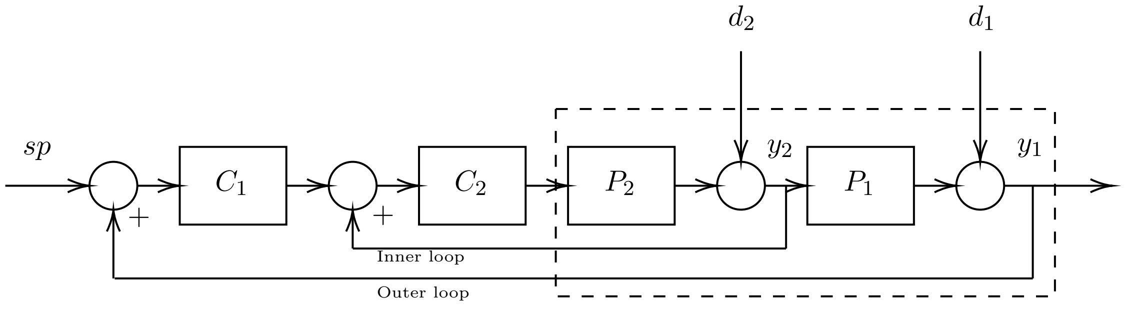


Figure 8.1: Cascade control of two processes with disturbance model

This cascade controlled process is implemented in an environmental class `CascadeEnvironment` that can be trained on any of the three agents. We define two fictual sequential processes P_2 and P_1, where the inner loop process P_2 is

$$P_2(s) = \frac{3}{s+2} \tag{8.1}$$

and P_1

$$P_1(s) = \frac{10}{(10+s)^3} \tag{8.2}$$

Each of the processes are disturbed by d_1 and d_2, which are two phase shifted sine-functions. We utilize the same network configuration for each of the three agents as we have used in all preceding experiments and implement a linear-quadratic reward function for penalizing large control efforts outside the set point of 1. The demonstration code is available in Appendix C.

Figure 8.2 visualises the results. The cascade P-controller in Figure 8.2(a) is manually tuned using the Ziegler–Nichols method [2], and we see an initial ring oscillation before it stabilizes around the setpoint. Of all agents, SAC is the only agent capable of performing similar or better than PID control in this case. The DDPG agent does not converge at all, and the PPO agent policy behaves oscillatory.

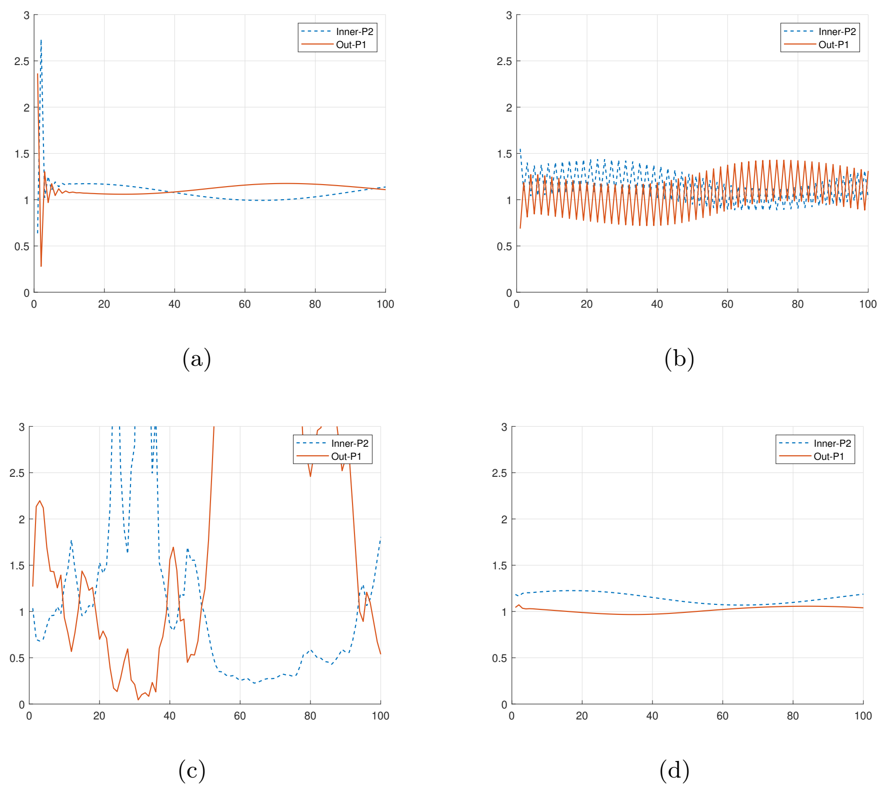


Figure 8.2: RL for control of non-linear functions
(a) cascade control using two P-regulators, (b) PPO agent solution after 4254 episodes, (c) DDPG agent solution after 2802 episodes, (d) SAC agent solution after 1019 episodes

In Figure 8.3 we can see the performance of the agent training by plotting the average reward per episode for the first 1000 episodes tested. Clearly, the SAC agent converges faster than PPO and DDPG, and was terminated after 1019 episodes. PPO exposes a *valley-to-valley* effect which it never escapes from, while the DDPG agent does not converge.

As general rules, we have found from our experiments that

- **PPO agents** have short training times, but do not always converge and might necessarily be restarted. PPO finds the first policy fast but might discard it in favour of others. This is visible in a *valley-to-valley* effect. Restarting the agent means resetting the internal weights and biases to a new randomized set. PPO converged in the multivariate cases using a trained reward model.
- **SAC agents** are stable in their training performance but requires substantially more computing power than PPO. SAC seems to gather more and more experience and steadily increase its value estimation and is not particularly sensitive to restarts. The stochasticity of the policy and value update prevents premature convergence

and encourages exploration due to the maximum entropy objective. SAC did not converge in the multivariate cases, but its performance in univariate action spaces is similar to PPO.

- **DDPG agents** does not converge easily within complex continuous observation- and action spaces, and as noted by Lillicrap et al. [59], the DDPG algorithm requires a large number of episodes to converge. Being patient using DDPG and running for many thousand episodes, is also noted in the Matlab documentation for DDPG.[2]

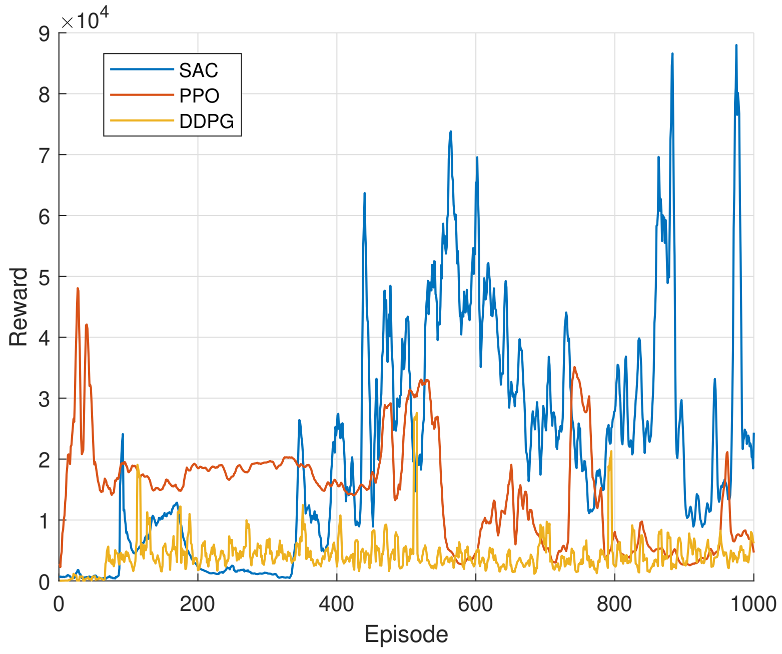


Figure 8.3: Average reward per episode per agent, first 1000 episodes for demonstration case

The case that restarting the PPO agent training increases its likelihood to identify a policy, can be an issue with the weight initialization of the actor and critic neural networks. By restarting, a new set of random weights are issued which might turn out to be especially effective for training. This trait is by Frankle and Carbin [79] described as *winning the initialization lottery*. We might then indicate that SAC is a more stable learner since we have not seen the same behaviour here.

We might criticize our experiments on the fact that the agent runs were terminated after 5000 episodes. However, depending on the horizon of the long-term reward and the complexity of the problem domain, we would probably see results within this training frame for time series like ours. For pure sparse reward scenarios, our generative model must first be able to forecast over the horizon before we can set the agent to train using the generative model as a simulator. In such a case, we expect that the number of episodes to

[2]https://se.mathworks.com/help/reinforcement-learning/ug/create-policy-and-value-function-representations.html

train for must be increased, in order to explore more of the environment and dependency paths.

An actor-critic reinforcement learning agent operates under the assumption that the system is *Markovian* and that any particular state has an expected future reward for taking an action, expressed through the Bellman equation:

$$V(s) = max_a(R(s,a) + \gamma V(s')) \tag{8.3}$$

In systems describable by thermodynamics, mass balances and mechanical physics, any state is expected to be Markovian. The error between the real state and the measured state used for training, is the sensors inability to measure, their errors or that state measurements are not available. Extrapolating for unknown state spaces is crucial, and is why we need to excite the system to extreme values, to be able to generalize the model. Any process not described by the sensors, cannot be dynamically identified.

A central concept to environment exploration is *information entropy* as defined by Shannon [3] and others as:

$$\mathcal{H} = -\sum_i p_i \log_2(p_i) \tag{8.4}$$

used in the value function for a SAC agent [82] as:

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)\sim p_\pi}[r(s_t,a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))] \tag{8.5}$$

The entropy of a learnt policy's state and action pairs $\pi(s_t, a_t)$ affects the *value* of the state, influencing an agent's inclination to select which parts of the environment to explore. The PPO agent uses a variant of the entropy loss and estimation of information gain using the *Kullback–Leibler divergence* [73]. The SAC agent trains a temperature factor α for the entropy term, which is not done in the PPO agent. This might be one of the elements that diverge the results between the two seemingly similar agents.

8.4 Reward engineering

The key to a successful agent policy is the reward scheme, given a balanced dataset for the process data. We have shown that both a manual reward scheme as well as a learnt model utilizing inverse reinforcement learning, can be used for training an agent. Depending on the application, the desired reward function will be the main focus of any practical RL-powered application. The reward function's role is not only to give direct feedback on the current actions but the path to a future reward. Designing the reward function when dealing with future rewards should involve a feedback mechanism on the trajectory towards this reward. Without it, training becomes a computationally intensive problem. We can also combine manual rewards with a learnt reward model. Powell et al. [111] added a *penalty term* to aid the direction of the reward, a term we have included in the reward functions tested. We saw early that an action penalty was crucial for the trainability of the agents. Further, we added a quadratic term to the penalty in order to accelerate learning.

The reward function is one scalar value that aids the agent action selection in the correct direction combined with several parameters. These parameters should be internally weighted so that they are on the same scale, calculated by the reward function R:

$$R(x) = [w_1 r_1 + w_{p1} p_1^2 + w_{p2} p_2^2 + wg] \tag{8.6}$$

where x is the observation vector and

- r_1, w_1 = Reward and weigth of primary observation (on responding variables)
- p_1^2, w_{p1} = Penalty and weigth of action effort (on manipulative variables)
- p_2^2, w_{p2} = Penalty and weigth of secondary observation (on controlled variables)
- g, w = Distance and weigth to future reward / goal

Designing a reward function that transforms to a new function when reaching a specific goal is also possible, for instance, when we have received one goal and need to transit to another goal, as demonstrated by the Dota 2 team by OpenAI [98] and Florensa et al. [69]. This is specifically useful in situations of mode shifts from production to cleaning, or change of recipes, products etc. in a continuous process. The dynamics of the generative model will propose actions to shift from one situation to another given the current state of the system. One important prerequisite is that we have process logs that exhibit these dynamics from earlier runs and that the model has process signals relevant for the task.

Reward engineering and the agent termination criteria are closely related in terms of being the factors that directly affect training. We have used the termination criteria actively to both limit the agent training to a narrow result set, and to indicate when the proper goal has been achieved. Both ways are equal in the sense that they lead the agent on the state-action trajectory to success, faster. In terms of computing efforts, it is advisable to terminate an episode as early as possible, when it is clear that the long-term goal cannot be reached. A proper goal state and reward scheme should reflect this.

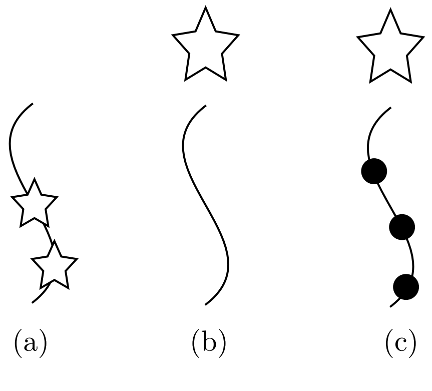


Figure 8.4: Three modes of rewards
(a) continuous, (b) sparse and (c) directed

We have at least three general modes of agent reward types as illustrated in Figure 8.4: a continuous, a long-term based on sparse rewards (episodic), and a directed sparse reward. In the episodic, or long-term mode, we want the agent to accomplish a greater goal, like winning a game or reaching a specific laboratory measurement result as a sparse reward. We have shown that the generative model is less likely to perform in a pure sparse reward scenario and that the reward scheme should lead the agent towards the sparse reward. With this, we mean that the agent should be able to optimize long-term reward trajectories if a consistent way of expressing the future goal in a current reward context exists. This observation is really dependent on the specific trained generative model's ability to forecast over the horizon. If the system state can be estimated accurately in the future, the generative model should be able to support the pure sparse reward scenario. In a pure continuous reward scheme, we have demonstrated that the concept can produce action proposals similar to the performance of the PID-regulator that generated the original training data.

8.5 Autoencoder and recurrency

The foundation for our generative model is the combination of the properties of the autoencoder and network recurrency. An autoencoder stores a latent representation of the learnt world, enabling the regeneration of data. Recurrency is about using earlier states for inference in the current state. Those two methods combined, we can forecast the future based on the past [116]–[118]. By changing the state as described in Chapter 4, we have also demonstrated simulation capabilities from a learnt model.

An important prerequisite for the autoencoder is that the system is generalisable, i.e. that we have sufficiently trained the network for its domain. In a fully connected autoencoder network, we construct the network with a limited number of hidden nodes, with input- and output nodes being of similar dimensions. We call the input layer the *encoder* while the output is the *decoder*, as exemplified in Figure 8.5.

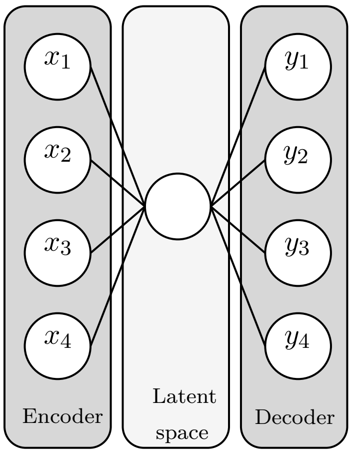


Figure 8.5: A simplified autoencoder

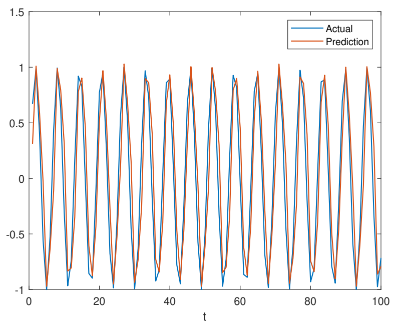
During training, the latent space nodes are gradually adjusted to minimize the error of the output compared to the input. The trained weights are a Gaussian representation of the problem, capable of reproducing data within the same domain. Autoencoders are one popular method for anomaly detection, where the neural network's generalisation properties are exploited. Trained on normal operation, autoencoders can recreate the current situation, if the input is within the latent space of the autoencoder. The autoencoder sum of errors will increase when confronted with data outside its training range, indicating anomalies. This method is becoming a usual method in equipment monitoring and system health surveillance.[3]

A recurrent neural network (RNN) uses past state to infer the next state. At each step in a sequence, we update the state and output according to a set of equations. In a plain RNN as described by Goodfellow et. al [63], forward pass through one node is as expressed in Equation 8.7:

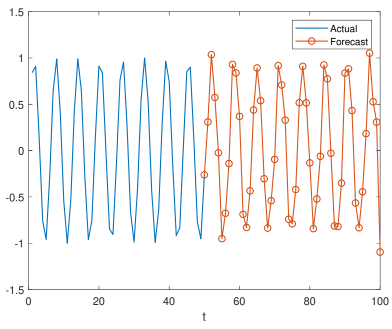
$$\begin{aligned}\mathbf{a}_t &= \mathbf{b} + \mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t \\ \mathbf{h}_t &= \sigma(\mathbf{a}_t) \\ \widehat{\mathbf{y}}_t &= \mathbf{c} + \mathbf{V}\mathbf{h}_t\end{aligned} \tag{8.7}$$

where $\mathbf{U}, \mathbf{V}, \mathbf{W}$ are trainable weight matrices, $\mathbf{b}, \mathbf{c}$ are trainable biases, $\mathbf{x}$ is the input vector and $\widehat{\mathbf{y}}$ the output vector. We can use either *tanh* or *sigmoid* activation function based on whether we have a classification or regression problem. The persistent state across the data is represented by $\mathbf{h}$. Backward passing is done by the derivative of the above functions, used for training through time with backpropagation and gradient descent. We traverse the training sequences backwards for each epoch when updating the error derivative.

We can demonstrate the capabilities of this simple method by training this algorithm to learn the dynamics of a sinus curve. The goal is to recreate and forecast future dynamics based on a sequence input. By using only four RNN nodes each implementing Equation 8.7, we are able to learn the dynamics both for a one-step forecast and a horizon forecast, as illustrated in Figure 8.6.



(a)



(b)

Figure 8.6: RNN for one-step forecast and horizoned forecast

Complete demonstration of the code is given in Appendix D. We see that recurrent memory is a powerful technique for dynamics learning. A limited set of equations and training is able to recreate the dynamics of the problem. Scaling the problem in time and variables, we are able to learn real-world system dynamics in a similar way, and combined with the powers of neural network frameworks, we can mix and combine different architectural components and numerical solvers to suit our problem.

[3]Honeywell Uniformance®Asset Sentinel, Intelecy AS and others

In recurrent training, gradients are backpropagated by walking through the sequences backwards in time (*Backpropagation Through Time*, BPTT). In standard RNN we encounter the problem of *vanishing gradients.* Hence, it is customed to implement a clipping mechanism on the gradients and as a consequence limit the number of steps to walk backwards (*Truncated BPTT*). RNN tends therefore to *forget* the past and is not especially effective for problems with long-term dependencies. One example is text prediction, where the grammar is heavily dependent on earlier events in a sentence.

The Long Short-Term Memory (LSTM) [21] has gradually evolved into a set of equations that is able to *remember* earlier events and by far surpasses the vanilla RNN model. We have earlier shown projects that use LSTM for dynamics in Chapter 2, and our generative model and the actor and critic networks are all made using LSTM. The internals of such a cell is illustrated in Equation 8.8 from Ismail et al. [84].

$$\begin{aligned}
\mathbf{i}_t &= \sigma(\mathbf{W}_{xi}\mathbf{x}_t + \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_t) \\
\mathbf{f}_t &= \sigma(\mathbf{W}_{xf}\mathbf{x}_t + \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f) \\
\mathbf{o}_t &= \sigma(\mathbf{W}_{xo}\mathbf{x}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o) \\
\tilde{\mathbf{c}}_t &= \tanh(\mathbf{W}_{xc}\mathbf{x}_t + \mathbf{W}_{hc}\mathbf{h}_{t-1} + \mathbf{b}_c) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \\
\mathbf{h}_t &= \tanh(\mathbf{o}_t \odot \mathbf{c}_t)
\end{aligned} \tag{8.8}$$

Here, we introduce the concept of *gated cells*, where the recurrent unit control what information is passed through. LSTM contain computational blocks that control the flow of information through many time steps. The LSTM cell consists of four sequential computing steps during the forward pass: the *forget* stage, where we execute what we have learned to forget, *storing* relevant new information to the state, selectively *update* the cell state, and finally the *output gate* that controls the information sent to the next state. There are several versions of the recurrent node with additions of functionality, or simpler implementations for specific purposes, like the *Gated Recurrent Unit* (GRU) or Linear Antisymmetric Recurrent Neural Networks (LARNN) [109], [121].

Per the documentation, the default initialisation of weights in the Matlab implementation of LSTM is done according to the findings of Glorot and Bengio [41]. Our RNN regression example demonstrator above uses the same method when initializing by drawing the initial weights from a normally distributed population.

8.6 Network design

By stacking LSTM layers, as we do in the generative model, we are able to capture more detailed information about the dynamics as we pass through the network if we have enough data coverage. However, by Bayesian optimization in Chapter 4, we have seen that the network need not be deep at all, we can obtain quite accurate results only with one LSTM layer, as demonstrated in Chapter 7.

Even though a single layer LSTM network per definition is not an autoencoder, we predict the same number of outputs as inputs. Per se, the encoder and decoder part of the autoencoder is embedded within the LSTM cells, effectively creating the illusion of an autoencoder.

During the experiments, we have observed some anomalous effects that should be addressed during engineering the generative model:

- **Estimating the mean** When the LSTM network start forecasting the mean, the network has probably no data coverage for the particular region, and assumes the mean is the best option for the lowest loss
- **Shifted estimates** The LSTM network outputs shifted/skewed estimates, usually an indication of estimating the last value only and thus the network has no predicting ability
- **Bias on estimates** The estimates are lower or higher than the real values and may be an indication that the bias values are not trained properly or that the network is not properly initialized
- **Instability** The estimates have a high-frequency component. This instability is visible just before resorting to mean estimation

Addressing these effects are done by adjusting the hyperparameters like the sequence length, the number of layers and the number of nodes per layer, as well as assuring training data is sufficient for modelling. For our model to generalize, we aim to simplify the architecture as much as possible. We have also observed that increasing the number of neurons and layers might lead to models with less predictive powers. We expect this to happen as a combination of not being able to generalize and not having enough training passes and training data.

We have also shown that a learning rate decay scheme is important for network training. By stepping down the learning rate, we see that the network have more accurate dynamics predictions. Increasing the number of epochs must be done according to the decay ratio.

Depending on the amount of training data available, the number of epochs should be selected wisely, and we have demonstrated that increasing the epoch count does not necessarily have a positive effect on the result. During one epoch, all training data are passed through the network once. If we augment the data by adding a noise component, each epoch is presented with slightly altered data to help the error update process, but noise itself could make the network unstable for a high number of epochs due to the integrated component. An alternative to our used function is the Ornstein-Uhlenbeck process [1] that generates noise correlated with previous noise in order to prevent degradation of overall dynamics. In DDPG, this process is used as enviornment exploration method, and should be implemented in our framework as well.

In Matlab, training data is broken into minibatches, mainly for computation efficiency purposes. However, Matlab updates the network weights after each minibatch. Randomizing the block sequence of training data for each epoch should be advisable, but is not tested here and remains a future work.

We can also use other methods and network elements to create models, like the Bi-LSTM [62]. In Bi-LSTM, we divide in forward- and backward layers when presented with an enclosed time series. Since our training of the generative model is based on presenting sequences of data to the network, we have the possibility to traverse the data in both directions. Potentially, this could increase the dynamics description capability of the network and should be tested as further work. Switching LSTM for one-dimensional CNNs is also an architecture that should be tested, as demonstrated by Lang et al. [94].

Entering the real-time domain involves many considerations of data quality and sampling. Ideally, we need to retain the sample time for process data in real-time, as during training. However, we might experience an irregular sampling of data in a real process, which recurrent neural networks are particularly sensitive to due to their dependency on the state matrices. Habiba and Pearlmutter [108] combine neural network training and neural layers based on the training of ODE functions within the network to overcome this issue. From this knowledge, one idea could involve introducing an autoencoder layer between process data and the agent environment interface using ODE neural networks to pertain the sample rate. A positive side effect of this would be inherent data filtering and anomaly detection on the input.

8.7 Use cases

We have used the industrial spray drier as an example system due to its simplicity in explanation, available literature and various possibilities for control as detailed in Chapter 3. However, we should not limit ourselves to controlling industrial processes. Time-series data are everywhere and in all businesses. The current value of a stock is a result of a time series of events, and the current price of electric power is the result of a multivariate chain of events and capacity forecasts. Optimization and planning are everywhere, and the method presented by this thesis could be seen as a *recommender system* for humans to make informed decisions.

The problem of human-computer interaction is a broad area of science in both computational fields and cognitive psychology. In his book *The Design of Everyday Things*, Norman [14] uses the metaphor of the *Gulf of Evaluation and Execution* to frame the problem of operators of a system making errors. He argues that an operator's inability to comprehend a system leads to suboptimal system manipulations, i.e. there is a broader or narrower gulf between what an operator *think* is his or her evaluation of the system state is, as compared to the actual system state. The system state might be more complex than humans can handle, in terms of multivariate parameters and path dependencies. We tend to break the state into manageable pieces, losing the big picture.

Within evolutionary game theory and evolutionary economics, this term is called *bounded rationalty* [43]. Players in a multi-agent simulation game act according to the information and knowledge provided to them, although in a broader essence, the players act wrong according to some global strategy. Bridging the *gulf of evaluation* using recommender systems could make the operator situational awareness increase, as a generative model is able to forecast the consequences of actions. The *gulf of execution* describes the sequence of actions necessary to reach a goal and our reduced understanding of being able to reach this state rationally. In essence, this describes the reasoning behind the reinforcement learning concept and long-term rewards. The distance between a system state and the desired state is expressed as *the gulf*, whereas the action sequence is expressed through *the policy.* If an agent has the policy to move from one state to another, this action sequence is a rational answer to the problem. Being able to identify when the agent is wrong or uncertain, is key to a successful human-computer trust relationship. In Chapter 5.5 we briefly discussed some popular methods for estimating uncertainty and demonstrated some of the principles and uses.

Bridging this *gulf* is the target for many of the applications within optimization and control. However, the method we have presented in this thesis differs when it comes to *how* we model the world when compared to classic control theory. Due to neural networks inherent *black box* property, we cannot state proof on the outcome. There are some initiatives within *Explainable AI*, namely through the calculation and evaluation of

SHapley Additive exPlanations [72] and *Local Interpretable Model-Agnostic Explanations* (LIME) [66]. It is an interesting future work to apply these methods to our framework. However, the methods presented can be used as-is in a secure setting. Below we present some other ideas for exploration:

- **Scenario simulation** Given the current state of a system, we should be able to use the simulation method presented in Chapter 4 to let an operator simulate the consequences of an action on a system.
- **Advisory and recommendation** Given the current state, we could also utilize a trained RL agent to propose the next recommended action on a system, as presented in the example in Chapter 7. This particular method of system-operator communication has also been implemented by Google Deepmind for proper datacenter cooling.[4]
- **Long term optimization** Given a properly trained generative model able to forecast beyond the horizon, we could build a system that optimizes a strategy for long term rewards, for example within energy use and capacity planning, as well as forecasting and alleviating emissions from a multi-step process. This could also be used for balancing a steam energy system, or for adjustments of windmill parameters like blade pitch, relative headings etc. according to weather prognosis and individual features of each windmill.
- **Chemical reactions** Chemical reactions and other mixing problems could use this method to derive the most optimal policy for reactant addition and process control, for example in a pulping process where the acidic bleaching step uses chlorine dioxide (ClO_2) and hydrogen peroxide (H_2O_2) in a mix depending on the material quality, recipe and various other parameters [24], [28], [34], [44]
- **Batch processes** Batch processes possess an interesting problem area where we can expect that each batch has an outcome known upon completion. This would map to the pure sparse reward discussed earlier. In such a problem, we must be able to train a generative model to forecast longer than one step. If we can do so, a batch process contains time series of the process from start till finish. Training on batch time series could for example be used in crystallization processes where we want to predict the best time for process termination.
- **Model for MPC with Kalman filter** We could also use the generative model in a more traditional context where we apply the model within a Kalman filter, for use with Model-Predictive Control (MPC)

As a summary, we believe that this method could be applied to overcome many of the industrial challenges as presented in Chapter 1.2.

8.8 Frontier in research

It is not known to the author a similar method used to train an RL agent for process control. As we introduced earlier, Petsagkourakis et al. [110] demonstrated and proposed a

[4]https://deepmind.com/blog/article/safety-first-ai-autonomous-data-centre-cooling-and-industrial-control

framework for learning the dynamics by hybrid methods: first by defining a mechanistic model based on historical data, and then train the agent using this model and real-time data. We have also seen research on data-driven modelling within process control [86], [91], [122], [128], but not a combination as we have presented. However, we expect that the method eventually will be discovered by many scientists and refined for different applications.

There are many research projects working in the field of machine learning, reinforcement learning and industry. One such research project is *Towards Autonomy in Process Industries (TAPI)*[5] where scientists and industry experts cooperate on several aspects within data-driven modelling. Of relevancy to our project, *compressed sensing* is a technique for modelling the un-measured part of a signal, as demonstrated by Lundby et al. [120] for an aluminium electrolysis process. The method could provide more stable learning due to measurement stabilization.

A successor of the LSTM network node, the Linear Antisymmetric Recurrent Neural Network node (LARNN) [109], [121] adapts an ODE function approximation within the recurrent node for stability reasons. We expect that a LARNN node is particularly suitable within the field of time series for process dynamics, as the mass-, energy- and mechanistic balances have a strong dependency in time and towards a Markovian function.

Handling uncertainty in both data and method is for the moment at least, a human task. In a recent article, Gawlikowski et al. [119] review many of the methods available for uncertainty estimation. One takeaway is to differentiate between epistemic noise, which originates from variability in the real world, and aleatoric noise, where the sensors or measurement variable's inability to reflect the state accurately introduces noise. The combination of these two main factors make up the complete noise picture, but we must handle each of them differently. We cannot throw bad data on a neural network and expect it to deliver top results.

8.9 Further work

Based on the findings in Chapter 7.3.4, an interesting and presumably powerful technique for zero-vector bias estimation is to dynamically compensate for state drift during estimation. This could be done by keeping a copy of the current state of a network and estimating the bias correction for the current state. This estimation could then be used for correcting the output dynamically. We have also observed that erroneous bias estimations are an indication of network instability, which could help in uncertainty estimations.

As an alternative to *compressed sensing* and data augmentation by noise addition, Yang et al. [127] proposes *Feature distribution smoothing* (FDS) for imbalanced datasets. Within classification, it is vital that the training dataset is balanced across the classes to predict, and a similar requirement for regression is to assure as normally distributed input data as possible. However, combinations of input data might not present the whole picture across the numeric range for the prediction.

Deep Imbalanced Regression (DIR) is a method to generate data by *Feature distribution smoothing* (FDS) for missing data areas. Our generative model is trained by datasets of unknown quality in this respect, and to limit the amount of uncontrolled interpolation, we could introduce this method for datasets when training the generative model.

[5]https://www.sintef.no/en/projects/2019/tapi-towards-autonomy-in-process-industries/

Of particular interest within RL research, is to find a viable method to restrict an agent's actions so that a process can be run securely, with respect to the environment, health and safety (EHS). A basic method would be to limit the action range and dampening when the process is under direct control from an agent, or by estimating the unknown dynamics for robust control [129]. Another possibility is to let the agent estimate process *set points* and let the process be controlled by traditional means. Google Deepmind[6] introduced a layered approach with several control barriers between the agent and the process. The integration of an RL agent into real-world control is an obvious further work based on the findings in this work.

Evolving the generative model, and the RL agent needs to be considered when monitoring reveals drift in the system. Drift comes from changed process conditions, unmodelled dynamics and changes in system requirements, amongst others. Introducing new data would either mean a complete recalculation of either model or train each model on new data with a low learning rate. Handling drift identification and compensation is a supportive task to the initial modelling to assure continuous operation over time. Further, we see that when a learnt reward approximation is used as a reward function, the operation monitoring must include this model as well. Handling this area correctly is considered to be one of the key success criteria for user acceptance.

An alternative implementation of the generative model's recurrent network could be a variant of the *Generative Adversarial Network* for time-series forecasting, discovered by Goodfellow et al. [53]. GANs are powerful neural networks for the recreation of highly credible copies of the real world, mainly within image- and audio recreation. However, we have also seen that GANs can be used for heart signal electrocardiogram synthesis [107]. It would be interesting to consider GANs for the generative model, and how to adapt the concept for simulation.

When considering cooperation between multiple RL agents working in isolation within their specific process, devising a scheme for agent cooperation could be beneficial. The ideas of evolutionary game theory have been applied to economical models, surveyed by Safarzynska et al. [43], but recently multiple agent simulation has surfaced using the same ideas. According to Tuyls and Nowé [29], agents working in their own Markovian space cannot converge when set to cooperate, since their joint action spaces are different. Evolutionary game theory is about the strategies employed when conflicting objectives make any action sub-optimal. By mathematical study, these strategies can be evaluated and an optimum can be found.

8.10 Limitations and threats to validity

This thesis has used both synthetic- and real-world data in the evaluation of the research questions. The use case tested is a limited and dynamically simple process. It could be considered a limitation to the conclusions as other systems would cater for more complex dynamics. We believe simplicity is part of the reason for our results.

Further, we have briefly touched on the neural network design of the tested agents, but have not optimized them in any way. Our intention has been to create the network architectures as similar as possible for agent comparison.

[6]https://deepmind.com/blog/article/safety-first-ai-autonomous-data-centre-cooling-and-industrial-control

The research area is in constant motion, and our hope is that the ideas and culprits presented here would be criticised, undermined and improved by others.

Chapter 9

Conclusion

For all the justified concern over automation gone wrong, surveillance, and all the other tech-enabled horrors, it's easy to forget that at its core, technology is about the triumph of humanity.[1]

The initial motivation of this thesis has been to demonstrate the theory and methods that lead us towards a higher stage of autonomy. The fundamental question of how we can formalize all knowledge of the world and utilize the data we have gathered in order to learn more remains a guiding star for this science field.

We have presented the foundation of research within reinforcement learning from the early days of *Q-learning* to the commonly used *actor-critic* methods of today, used for continuous action spaces with recurrent neural networks as function approximators. We have answered research questions that support the claim that reinforcement learning can be applied as a general optimization concept and trained on historical logs of data towards new goal sets. Evolving the agent on the target system seems necessary for finer control.

A generative model has been proposed for learning the compressed dynamics of a system. We have used an industrial spray drier as an example of a *Markovian* system well suitable for modelling. A spray drier can be explained in terms of mass- and energy balances, and we have constructed a synthetic model for training a generative model based on a recurrent autoencoder network. This generative model is able to reproduce the dynamics of a system, and by modification of the model input, we are able to demonstrate simulation and forecasting characteristics.

This generative model is then used as a synthetic environment for training a reinforcement agent. It is demonstrated that a reinforcement learning agent can be sufficiently trained in a multivariate scenario and propose actions towards a new goal set, making it particularly suitable for optimization scenarios as an alternative to traditional control mechanisms. We have also shown that the reward function of an agent can be trained based on a specified class of time-series data enabling efficient reward engineering and human behaviour replication.

Finally, we were able to extract data from a real spray drier and demonstrate the method's applicability in the real world domain. It is not known to the author any current initiatives demonstrating the combination of these features, although all building blocks are known and available.

Initially, we raised the concern on how to build *experience* with an example of crashing a self-driving car. In a simulator, insecure actions lead to negative rewards, in the real

[1]Bionic Eye, Emily Mullin, medium.com

world an agent acts according to a learnt policy and value function. Training the *critic* is crucial, and resembles the human *morale engine.* What constitutes negative rewards in the scenario of agent training, depends on the human engineer. Those who train the critic, copy their morality into the digitized agent. One of the findings of Chapter 5.3.2 demonstrates this dilemma clearly: we need data from *failures* in order to learn.

How can we be sure that the agent acts according to human morality standards? It is a danger that in the world of AI-operated decision-makers, we will cross borders and use autonomous technology to harm humanity. History is full of examples, the grimmest ones can be found in the military drone industry and in the seemingly never-ending cyber-attacks on process plants and other actuators connected to the internet. The challenges for AI and humanity is not necessarily technological but remains to be a question of morality and cultural imbalance.

We will conclude this thesis by stating the three laws of robotics from the *Handbook of Robotics, 56th Edition, 2058 A.D.* transcribed by science fiction writer and professor Isaac Asimov [4] which may serve as a guiding light for agent training:

- **First Law** A robot may not injure a human being or, through inaction, allow a human being to come to harm.
- **Second Law** A robot must obey the orders given it by human beings except where such orders would conflict with the First Law.
- **Third Law** A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

Or, as nicely summoned by Google: *Don't be evil.*

Bibliography

[1] G. E. Uhlenbeck and L. S. Ornstein, "On the theory of the brownian motion," *Phys. Rev.*, vol. 36, no. 5, pp. 823–841, Sep. 1930.

[2] J. G. Ziegler, N. B. Nichols, *et al.*, "Optimum settings for automatic controllers," *Trans. ASME J. Appl. Mech.*, vol. 64, no. 11, 1942.

[3] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, Jul. 1948.

[4] I. Asimov, "Runaround. i, robot (the isaac asimov collection ed.)," *New York City: Doubleday. (first published 1942)*, 1950.

[5] A. M. Turing, "I.—COMPUTING MACHINERY AND INTELLIGENCE," *Mind*, vol. LIX, no. 236, pp. 433–460, 1950.

[6] F. Rosenblatt, *The perceptron: A probabilistic model for information storage and organization in the brain*, 1958.

[7] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM J. Res. Dev.*, vol. 3, no. 3, pp. 210–229, Jul. 1959.

[8] D. H. Hubel and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," en, *J. Physiol.*, vol. 160, pp. 106–154, Jan. 1962.

[9] P. J. Huber, "Robust estimation of a location parameter," en, *aoms*, vol. 35, no. 1, pp. 73–101, Mar. 1964.

[10] K. Fukushima, S. Miyake, and T. Ito, "Neocognitron: A neural network model for a mechanism of visual pattern recognition," *IEEE Trans. Syst. Man Cybern.*, vol. SMC-13, no. 5, pp. 826–834, Sep. 1983.

[11] K. Masters *et al.*, "Spray drying handbook," *Spray drying handbook.*, 1985.

[12] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[13] J. G. Balchen and K. I. Mummé, *Process control: Structures and applications.* Kluwer Academic Pub, 1988.

[14] D. A. Norman, *The design of everyday things.* Currency Doubleday, 1988.

[15] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Mach. Learn.*, vol. 3, no. 1, pp. 9–44, Aug. 1988.

[16] C. J. C. H. Watkins, "Learning from delayed rewards," 1989.

[17] G. Tesauro, “Practical issues in temporal difference learning,” *Mach. Learn.*, vol. 8, no. 3, pp. 257–277, May 1992.

[18] C. J. C. Watkins and P. Dayan, “Q-Learning,” Tech. Rep., 1992, pp. 279–292.

[19] R. J. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning,” *Mach. Learn.*, vol. 8, no. 3, pp. 229–256, May 1992.

[20] G. Tesauro, “TD-Gammon, a self-teaching backgammon program, achieves master-level play,” en, *Neural Comput.*, vol. 6, no. 2, pp. 215–219, Mar. 1994.

[21] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” en, *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[22] T. Fukuda and T. Arakawa, “Intelligent systems: Robotics versus mechatronics,” *Annu. Rev. Control*, vol. 22, pp. 13–22, Jan. 1998.

[23] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[24] J. Gullichsen, C.-J. Fogelholm, and O. Fapet, “Chemical pulping, papermaking science and technology,” *Book 6B*, 2000.

[25] H. Siebert, *Der Kobra-effekt: Wie man irrwege der Wirtschaftspolitik vermeidet.* Dt. Verlag-Anst., 2001.

[26] D. E. Oakley, “Spray dryer modeling in theory and practice,” *Drying Technol.*, vol. 22, no. 6, pp. 1371–1402, Jun. 2004.

[27] B. Adhikari, T. Howes, D. Lecomte, and B. R. Bhandari, “A glass transition temperature approach for the prediction of the surface stickiness of a drying droplet during spray drying,” *Powder Technol.*, vol. 149, no. 2, pp. 168–179, Jan. 2005.

[28] D. Major, M. Perrier, S. Gendron, and B. Lupien, “Pulp bleaching control and optimization,” *IFAC Proceedings Volumes*, vol. 38, no. 1, pp. 466–476, Jan. 2005.

[29] K. Tuyls and A. Nowé, “Evolutionary game theory and multi-agent reinforcement learning,” *Knowl. Eng. Rev.*, vol. 20, pp. 63–90, Mar. 2005.

[30] V. Vovk, A. Gammerman, and G. Shafer, *Algorithmic Learning in a Random World.* Springer, Boston, MA, 2005.

[31] Y.-C. L. A i, S. H. Fan, S. Chenney, and C. Dyer, “Photorealistic image rendering with population monte carlo energy redistribution,” *Eurographics Symposium on Rendering*, 2007.

[32] S. H. Fan, “Population monte carlo samplers for rendering,” Tech. Rep., Sep. 2007.

[33] D. Ramachandran and E. Amir, *Bayesian inverse reinforcement learning*, `https://www.aaai.org/Papers/IJCAI/2007/IJCAI07-416.pdf`, Accessed: 2021-4-24, 2007.

[34] P. W. Hart and D. Connell, “Improving chlorine dioxide bleaching efficiency by selecting the optimum ph targets,” *Tappi J.*, vol. 7, no. 7, pp. 3–11, Jul. 2008.

[35] L. X. Huang and A. S. Mujumdar, “The effect of rotary disk atomizer RPM on particle size distribution in a Semi-Industrial spray dryer,” *Drying Technol.*, vol. 26, no. 11, pp. 1319–1325, Oct. 2008.

[36] B. D. Ziebart, A. Maas, J. Andrew Bagnell, and A. K. Dey, *Maximum entropy inverse reinforcement learning*, `https://www.aaai.org/Papers/AAAI/2008/AAAI08-227.pdf`, Accessed: 2021-2-20, 2008.

[37] S. Bhatnagar, R. S. Sutton, M. Ghavamzadeh, and M. Lee, "Natural actor–critic algorithms," *Automatica*, vol. 45, no. 11, pp. 2471–2482, Nov. 2009.

[38] M. Hoekstra, M. Vogelzang, E. Verbitskiy, and M. W. N. Nijsten, "Health technology assessment review: Computerized glucose regulation in the intensive care unit–how to create artificial control," en, *Crit. Care*, vol. 13, no. 5, p. 223, Oct. 2009.

[39] M. Lopes, F. Melo, and L. Montesano, "Active learning for reward estimation in inverse reinforcement learning," in *Machine Learning and Knowledge Discovery in Databases*, Springer Berlin Heidelberg, 2009, pp. 31–46.

[40] GEA Process Engineering, "Milk powder Technology-Evaporation and spray drying," Tech. Rep. Fifth edition, Feb. 2010.

[41] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," 2010.

[42] J. O'Neill, B. Pleydell-Bouverie, D. Dupret, and J. Csicsvari, "Play it again: Reactivation of waking experience and memory," en, *Trends Neurosci.*, vol. 33, no. 5, pp. 220–229, May 2010.

[43] K. Safarzynska and J. C. van den Bergh, *Evolutionary modelling in economics: a survey of methods and building blocks.* Max Planck Institute of Economics, Jena, 2010.

[44] V. Tarvo *et al.*, *Modeling chlorine dioxide bleaching of chemical pulp.* Aalto-yliopiston teknillinen korkeakoulu, 2010.

[45] A. M. Saxe, P. W. Koh, Z. Chen, M. Bhand, B. Suresh, and A. Y. Ng, "On random weights and unsupervised feature learning," in *ICML*, vol. 2, cl.uni-heidelberg.de, 2011, p. 6.

[46] A. Azadeh, N. Neshat, A. Kazemi, and M. Saberi, "Predictive control of drying process using an adaptive neuro-fuzzy and partial least squares approach," en, *Int. J. Adv. Manuf. Technol.*, vol. 58, no. 5-8, pp. 585–596, Jan. 2012.

[47] Y. Bengio, "Practical recommendations for Gradient-Based training of deep architectures," in *Neural Networks: Tricks of the Trade: Second Edition*, G. Montavon, G. B. Orr, and K.-R. Müller, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 437–478.

[48] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, *Improving neural networks by preventing co-adaptation of feature detectors*, 2012.

[49] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient BackProp," in *Neural Networks: Tricks of the Trade: Second Edition*, G. Montavon, G. B. Orr, and K.-R. Müller, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 9–48.

[50] N. Bostrom, *Anthropic Bias : Observation Selection Effects in Science and Philosophy*, 1st Edition. Routledge, May 2013.

[51] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," Dec. 2013. arXiv: `1312.5602 [cs.LG]`.

[52] V. Balasubramanian, S.-S. Ho, and V. Vovk, *Conformal Prediction for Reliable Machine Learning: Theory, Adaptations and Applications*, en. Newnes, Apr. 2014.

[53] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," Jun. 2014. arXiv: `1406.2661 [stat.ML]`.

[54] D. Silver, "Deterministic policy gradient algorithms," in *Proceedings of the 31st International Conference on MachineLearning*, 2014.

[55] N. Srivastava, G. Hinton, A. Krizhevsky, *et al.*, "Dropout: A simple way to prevent neural networks from overfitting," *The journal of machine*, 2014.

[56] Y. Gal and Z. Ghahramani, "A theoretically grounded application of dropout in recurrent neural networks," Dec. 2015. arXiv: `1512.05287 [stat.ML]`.

[57] ——, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," Jun. 2015. arXiv: `1506.02142 [stat.ML]`.

[58] M. Hausknecht and P. Stone, "Deep recurrent Q-Learning for partially observable MDPs," Jul. 2015. arXiv: `1507.06527 [cs.LG]`.

[59] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," Sep. 2015. arXiv: `1509.02971 [cs.LG]`.

[60] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," en, *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[61] S. Pote and S. Sudit, *Automatic control system for spray drier pilot plant*, `https://www.erpublication.org/published_paper/IJETR032111.pdf`, Accessed: 2021-8-21, May 2015.

[62] S. Cornegruta, R. Bakewell, S. Withey, and G. Montana, "Modelling radiological language with bidirectional long Short-Term memory networks," Sep. 2016. arXiv: `1609.08409 [cs.CL]`.

[63] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, en. MIT Press, Nov. 2016.

[64] J. Ho and S. Ermon, "Generative adversarial imitation learning," Jun. 2016. arXiv: `1606.03476 [cs.LG]`.

[65] A. Parastiwi and Ekojono, "Design of spray dryer process control by maintaining outlet air temperature of spray dryer chamber," in *2016 International Seminar on Intelligent Technology and Its Applications (ISITIA)*, Jul. 2016, pp. 619–622.

[66] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should I trust you?": Explaining the predictions of any classifier," Feb. 2016. arXiv: `1602.04938 [cs.LG]`.

[67] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," en, *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.

[68] A. Antoniou, A. Storkey, and H. Edwards, "Data augmentation generative adversarial networks," Nov. 2017.

[69] C. Florensa, D. Held, X. Geng, and P. Abbeel, "Automatic goal generation for reinforcement learning agents," May 2017. arXiv: `1705.06366 [cs.LG]`.

[70] A. Karpatne, W. Watkins, J. Read, and V. Kumar, "Physics-guided neural networks (PGNN): An application in lake temperature modeling," Oct. 2017. arXiv: `1710.11431 [cs.LG]`.

[71] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.

[72] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, Curran Associates, Inc., 2017, pp. 4765–4774.

[73] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," Jul. 2017. arXiv: `1707.06347 [cs.LG]`.

[74] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," en, *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017.

[75] I. Zbicinski, "Modeling and scaling up of industrial spray dryers: A review," *J. Chem. Eng. Jpn.*, vol. 50, no. 10, pp. 757–767, Oct. 2017.

[76] J. Alexander, *Learning from humans: What is inverse reinforcement learning?* `https://thegradient.pub/learning-from-humans-what-is-inverse-reinforcement-learning/`, Accessed: 2021-4-27, Jun. 2018.

[77] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, "Neural ordinary differential equations," Jun. 2018. arXiv: `1806.07366 [cs.LG]`.

[78] H. I. Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller, "Deep learning for time series classification: A review," Sep. 2018. arXiv: `1809.04356 [cs.LG]`.

[79] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," Mar. 2018. arXiv: `1803.03635 [cs.LG]`.

[80] D. Ha and J. Schmidhuber, "World models," Mar. 2018. arXiv: `1803.10122 [cs.LG]`.

[81] T. Haarnoja, S. Ha, A. Zhou, J. Tan, G. Tucker, and S. Levine, "Learning to walk via deep reinforcement learning," Dec. 2018. arXiv: `1812.11103 [cs.LG]`.

[82] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft Actor-Critic: Off-Policy maximum entropy deep reinforcement learning with a stochastic actor," *arXiv [cs.LG]*, Jan. 2018.

[83] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft Actor-Critic algorithms and applications," Dec. 2018. arXiv: `1812.05905 [cs.LG]`.

[84] A. A. Ismail, T. Wood, and H. C. Bravo, "Improving Long-Horizon forecasts with Expectation-Biased LSTM networks," Apr. 2018.

[85] B. Kleinberg, Y. Li, and Y. Yuan, “An alternative view: When does SGD escape local minima?” In *Proceedings of the 35th International Conference on Machine Learning*, J. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, PMLR, 2018, pp. 2698–2707.

[86] S. Moe, A. M. Rustad, and K. G. Hanssen, “Machine learning in control systems: An overview of the state of the art,” in *Artificial Intelligence XXXV*, Springer International Publishing, 2018, pp. 250–265.

[87] OpenAI, *Part 2: Kinds of RL algorithms — spinning up documentation*, `https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html`, Accessed: 2021-5-2, 2018.

[88] D. Santos, A. C. Maurício, V. Sencadas, J. D. Santos, M. H. Fernandes, and P. S. Gomes, “Spray drying: An overview,” in *Biomaterials*, R. Pignatello and T. Musumeci, Eds., Rijeka: IntechOpen, 2018, ch. 2.

[89] P. Thodoroff, A. Durand, J. Pineau, and D. Precup, “Temporal regularization in markov decision process,” Nov. 2018. arXiv: `1811.00429 [cs.LG]`.

[90] S. Thorn, *Predicting uncertainty with neural networks*, `https://medium.com/@steve_thorn/predicting-uncertainty-with-neural-networks-aec0217eb37d`, Accessed: 2021-10-31, Mar. 2018.

[91] E. A. Del Rio-Chanona, X. Cong, E. Bradford, D. Zhang, and K. Jing, “Review of advanced physical and data-driven models for dynamic bioprocess simulation: Case study of algae–bacteria consortium wastewater treatment,” *Biotechnol. Bioeng.*, vol. 116, no. 2, pp. 342–353, Feb. 2019.

[92] G. Dulac-Arnold, D. Mankowitz, and T. Hester, “Challenges of real-world reinforcement learning,” Apr. 2019. arXiv: `1904.12901 [cs.LG]`.

[93] A. Honchar, *Neural ODEs: Breakdown of another deep learning breakthrough*, `https://towardsdatascience.com/neural-odes-breakdown-of-another-deep-learning-breakthrough-3e78c7213795`, Accessed: 2021-4-20, Jun. 2019.

[94] C. Lang, F. Steinborn, O. Steffens, and E. W. Lang, “Electricity load forecasting – an evaluation of simple 1D-CNN network structures,” Nov. 2019. arXiv: `1911.11536 [cs.LG]`.

[95] A. Look and M. Kandemir, “Differential bayesian neural nets,” Dec. 2019. arXiv: `1912.00796 [cs.LG]`.

[96] G. Mendels, *Estimating uncertainty in machine learning models — part 3*, `https://towardsdatascience.com/estimating-uncertainty-in-machine-learning-models-part-3-22b8c58b07b`, Accessed: 2021-10-31, Oct. 2019.

[97] S. Minaee, E. Azimi, and A. Abdolrashidi, “FingerNet: Pushing the limits of fingerprint recognition using convolutional neural network,” Jul. 2019. arXiv: `1907.12956 [cs.CV]`.

[98] OpenAI, : C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, “Dota 2 with large scale deep reinforcement learning,” Dec. 2019. arXiv: `1912.06680 [cs.LG]`.

[99] OpenAI, I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, J. Schneider, N. Tezak, J. Tworek, P. Welinder, L. Weng, Q. Yuan, W. Zaremba, and L. Zhang, "Solving rubik's cube with a robot hand," Oct. 2019. arXiv: `1910.07113 [cs.LG]`.

[100] A. Sagheer and M. Kotb, "Time series forecasting of petroleum production using deep LSTM recurrent networks," *Neurocomputing*, vol. 323, pp. 203–213, Jan. 2019.

[101] ——, "Unsupervised pre-training of a deep LSTM-based stacked autoencoder for multivariate time series forecasting problems," en, *Sci. Rep.*, vol. 9, no. 1, p. 19 038, Dec. 2019.

[102] K. You, M. Long, J. Wang, and M. I. Jordan, "How does learning rate decay help modern neural networks?," Aug. 2019. arXiv: `1908.01878 [cs.LG]`.

[103] D. Zhang, E. A. Del Rio-Chanona, P. Petsagkourakis, and J. Wagner, "Hybrid physics-based and data-driven modeling for bioprocess online simulation and optimization," *Biotechnol. Bioeng.*, vol. 116, no. 11, pp. 2919–2930, Nov. 2019.

[104] A. Barredo Arrieta, N. Díaz-Rodríguez, J. Del Ser, A. Bennetot, S. Tabik, A. Barbado, S. Garcia, S. Gil-Lopez, D. Molina, R. Benjamins, R. Chatila, and F. Herrera, "Explainable artificial intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI," *Inf. Fusion*, vol. 58, pp. 82–115, Jun. 2020.

[105] W. Fedus, P. Ramachandran, R. Agarwal, Y. Bengio, H. Larochelle, M. Rowland, and W. Dabney, "Revisiting fundamentals of experience replay," Jul. 2020. arXiv: `2007.06700 [cs.LG]`.

[106] T. A. Gaffoor, *AI for industrial process control (part 2): Model predictive control deep dive*, `https://www.innovyze.com/en-us/blog/ai-for-industrial-process-control-part-2-model-predictive-control-deep-dive`, Accessed: 2021-4-17, Nov. 2020.

[107] T. Golany, D. Freedman, and K. Radinsky, "SimGANs: Simulator-Based generative adversarial networks for ECG synthesis to improve deep ECG classification," Jun. 2020. arXiv: `2006.15353 [eess.SP]`.

[108] M. Habiba and B. A. Pearlmutter, "Neural ordinary differential equation based recurrent neural network model," May 2020. arXiv: `2005.09807 [cs.LG]`.

[109] S. Moe, F. Remonato, E. I. Grøtli, and J. T. Gravdahl, "Linear antisymmetric recurrent neural networks," in *Proceedings of the 2nd Conference on Learning for Dynamics and Control*, A. M. Bayen, A. Jadbabaie, G. Pappas, P. A. Parrilo, B. Recht, C. Tomlin, and M. Zeilinger, Eds., ser. Proceedings of Machine Learning Research, vol. 120, PMLR, 2020, pp. 170–178.

[110] P. Petsagkourakis, I. O. Sandoval, E. Bradford, D. Zhang, and E. A. del Rio-Chanona, "Reinforcement learning for batch bioprocess optimization," *Comput. Chem. Eng.*, vol. 133, p. 106 649, Feb. 2020.

[111] B. K. M. Powell, D. Machalek, and T. Quah, "Real-time optimization using reinforcement learning," en, *Comput. Chem. Eng.*, vol. 143, no. 107077, p. 107 077, Dec. 2020.

[112] Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning: An Introduction.* 2020.

[113] J. Robine, T. Uelwer, and S. Harmeling, “Discrete latent space world models for reinforcement learning,” Oct. 2020. arXiv: 2010.05767 [cs.LG].

[114] W. Tang, G. Long, L. Liu, T. Zhou, J. Jiang, and M. Blumenstein, “Rethinking 1D-CNN for time series classification: A stronger baseline,” Feb. 2020. arXiv: 2002.10061 [cs.LG].

[115] A. Tuor, J. Drgona, and D. Vrabie, “Constrained neural ordinary differential equations with stability guarantees,” Apr. 2020. arXiv: 2004.10883 [eess.SY].

[116] P. R. Vlachas, G. Arampatzis, C. Uhler, and P. Koumoutsakos, “Learning the effective dynamics of complex multiscale systems,” Jun. 2020. arXiv: 2006.13431 [physics.comp-ph].

[117] T. Xayasouk, H. Lee, and G. Lee, “Air pollution prediction using long Short-Term memory (LSTM) and deep autoencoder (DAE) models,” en, *Sustain. Sci. Pract. Policy*, vol. 12, no. 6, p. 2570, Mar. 2020.

[118] M. Elsaraiti and A. Merabet, “A comparative analysis of the ARIMA and LSTM predictive models and their effectiveness for predicting wind speed,” Oct. 2021.

[119] J. Gawlikowski, C. R. N. Tassi, M. Ali, J. Lee, M. Humt, J. Feng, A. Kruspe, R. Triebel, P. Jung, R. Roscher, M. Shahzad, W. Yang, R. Bamler, and X. X. Zhu, “A survey of uncertainty in deep neural networks,” Jul. 2021. arXiv: 2107.03342 [cs.LG].

[120] E. T. B. Lundby, A. Rasheed, J. T. Gravdahl, and I. J. Halvorsen, “A novel hybrid analysis and modeling approach applied to aluminum electrolysis process,” *J. Process Control*, vol. 105, pp. 62–77, Sep. 2021.

[121] S. Moe and C. Sterud, “Decoupling dynamics and sampling: RNNs for unevenly sampled data and flexible online predictions,” *Learning for Dynamics and Control*, 2021.

[122] C. Sterud, S. Moe, M. V. Bram, S. Roberts, and J.-P. Calliess, “Recurrent neural network structures for learning control valve behaviour,” *Automation, Robotics & Communications for Industry 4. 0*, p. 20, 2021.

[123] Wikipedia contributors, *Chain rule (probability)*, https://en.wikipedia.org/w/index.php?title=Chain_rule_(probability)&oldid=1017757604, Accessed: 2021-4-27, Apr. 2021.

[124] ——, *Conservation of mass*, https://en.wikipedia.org/w/index.php?title=Conservation_of_mass&oldid=1036600537, Accessed: 2021-8-1, Aug. 2021.

[125] ——, *Density of air*, https://en.wikipedia.org/w/index.php?title=Density_of_air&oldid=1031055600, Accessed: 2021-6-29, Jun. 2021.

[126] ——, *Laws of thermodynamics*, https://en.wikipedia.org/w/index.php?title=Laws_of_thermodynamics&oldid=1039497430, Accessed: 2021-8-19, Aug. 2021.

[127] Y. Yang, K. Zha, Y.-C. Chen, H. Wang, and D. Katabi, “Delving into deep imbalanced regression,” Feb. 2021. arXiv: 2102.09554 [cs.LG].

[128] S. Moe, B. Myhre, A. M. Rustad, H. Helness, and F. Batey, “Neural network to analyze wastewater treatment plant with CEPT,”

[129] C. Sterud, S. Moe, and J. T. Gravdahl, “Stable and robust neural network controllers,”

Appendix A

Matlab Code

A.1 Spray drier ODE function

A function pointer to this function is used for the `ode45`[1] ordinary differential equation solver in Matlab, called by the PID regulator in `spray_rollout_pid` for training the generative model, `spray_rollout_pid_stable` for training the reward model and within `ProcessEnvironment` as a real time process simulator. This ODE function models the spray drier mass- and energy balances.

```matlab
function dydt = spray_f(t,y,Q,Qg,Tfeed,Tin,RHg,pfeed,rpm,tau)

    dydt = zeros(6,1); % Qstate Qgstate Tout Dropletsz Hum bv

    % specific heat capacity
    cfeed = 4000; % Empiric unknown % water 4184, Joules per kilogram per degree Celsius (J/kg°C).
    %pfeed = 1200; % density feed, kg/m3

    cgas = 600; % air, Joules per kilogram per degree Celsius (J/kg°C)
    % density, kg/m3 (table), varies with temp/RH: https://en.wikipedia.org/wiki/Density_of_air
    pgas = 0.7; % Empiric unknown, 1.225 (20*C);

    % Mass balance
    mwrate = (Q * pfeed) / tau; % water rate m3/h to mass kg/min
    mgrate = (Qg * pgas) / tau; % gas rate m3/h to mass kg/min
    dydt(1) = -y(1) + mwrate;
    dydt(2) = -y(2) + mgrate;

    % Atomizer rpm
    dydt(3) = (-y(3) + rpm) * 0.05;

    % Energy balance, instant heat flux
    beta = 0.9;
    deltat_gas = Tin - Tfeed; % gas - feed, diff temp (K)
    energy_gas = mgrate * cgas * deltat_gas * beta;
    dydt(4) = -y(4) + (energy_gas / mwrate / cfeed); % Celsius

    % Droplet size from atomizer
    % Sauter mean droplet size formula, L. X. Huang and A. S. Mujumdar
    M = (Q * pfeed);
    nom = 1.4e4*(M^0.24);
    d = 0.7; % diameter m
    n = 0.1; % height m
    h = 1; % unkown
    denom = ((y(3) * d)^0.83) * ((n * h)^0.12);
    droplet_size = nom / denom;
    dydt(5) = -y(5) + droplet_size; % micrometer (um)

```

[1]https://se.mathworks.com/help/matlab/ref/ode45.html

```
    % Evaporation dynamics, assumptions:
    %   The droplet is heated before evaporation starts
    %   Near-linear evaporation (as a function of droplet size)
    time = 2; % s
    temp = 1/y(4); % C
    area = 1/y(5); % um

    % Convert relative humidity to absolute humidity
    % https://carnotcycle.wordpress.com/2012/08/04...
    % /how-to-convert-relative-humidity-to-absolute-humidity/
    deltaRH = ( 6.112 * exp((17.67 * 25)/(25+243.5)) * RHg * 2.1674) / (273.15+25);
    deltaRH = deltaRH / 1000;
    gamma = 1.8;
    evap = time * temp * area * 1/deltaRH * 100 * gamma;
    dydt(6) = -y(6) + evap;

    % Bulk weight linear model r^2=0.76
    dydt(7) = -y(7) + y(6) * 0.0276 + 0.5088;
end
```

Listing A.1: spray_f.m

A.2 Datastores

The datastores are used for training the generative- or reward models as pluggable classes for training and validation data according to the moving window skewing mechanism described in Chapter 4. There are two datastores implemented; the `RolloutDatastore` and `HistorianDatastore`.

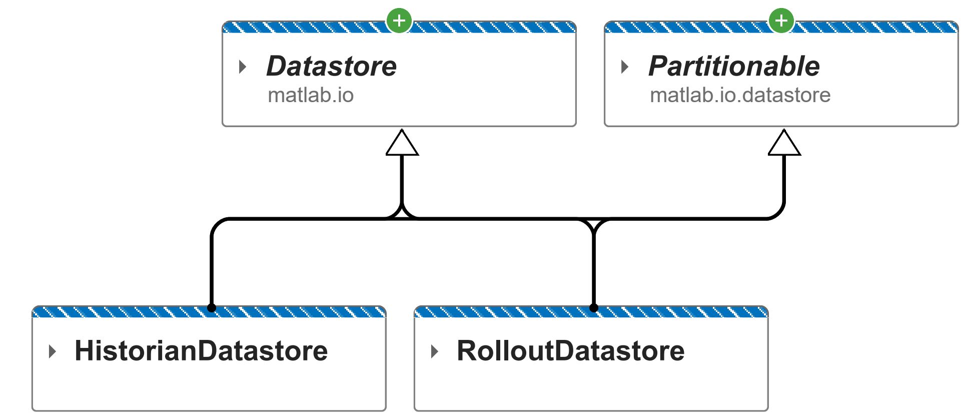


Figure A.1: Class diagram of datastores

The `RolloutDatastore` takes a PID function as input and calls this function to calculate in real time the process data, which otherwise could come from timeseries log files.

Instantiating the class is done by:

```
datastore = RolloutDatastore(@rollout_func, rollouts, sequence_length);
```

where `@rollout_func` is a function pointer to one of the implemented rollout functions:

- `spray_rollout_pid`
- `spray_rollout_pid_stable`

The `HistorianDatastore` connects to a Historian database and queries for the tag data available.

A.2.1 RolloutDatastore.m

```matlab
% A custom datastore for generating data from specified ODE function rollout
% Data is scaled based on an initial dataset and pre-processed accordingly
classdef RolloutDatastore < matlab.io.Datastore & ...
                            matlab.io.datastore.Partitionable

    properties
        featureDimension
        Mean
        Std
        Rollouts
        Sequence_length
        Empty_columns
        MV % manipulated variables
        CV % controlled variables
        RV % responding variables
    end

    properties (Access = private)
        rollout_func
        counts
    end

    methods

        function this = RolloutDatastore(rollout_func, rollouts, sequence_length)
            this.rollout_func = rollout_func;
            this.Rollouts = rollouts; % number of calculated rollouts per session/epoch
            this.Sequence_length = sequence_length;
            reset(this);

            % Establish parameters for scaling by creating random
            % rollout distributions. This will be overwritten if loading a model from
            % disk (GenerativeModel.m)
            Tt = [];
            for i=1:rollouts
                [T, MV, CV, RV] = this.rollout_func();
                Tt = [Tt;T];
            end
            this.Mean = mean(T);
            this.Std = std(T);

            % Identify columns with standard deviation of zero (or some
            % other threshold, need to verify
            this.Empty_columns = ~any(std(T),1);
            this.Mean( :, this.Empty_columns ) = [];
            this.Std( :, this.Empty_columns ) = [];
            this.featureDimension = size(this.Mean,2);

            % Remove those columns from metadata
            mask = mat2cell(this.Empty_columns, 1, [numel(MV) numel(CV) numel(RV)]);
            this.MV = MV(~mask{1});
            this.CV = CV(~mask{2});
            this.RV = RV(~mask{3});
        end

        function [rollout, legend] = inverse_scale(this, rollout)
```

```
            rollout = (rollout .* this.Std) + this.Mean;
            legend = {this.MV this.CV this.RV};
        end

        function [rollout, legend] = scale(this, rollout)
            rollout = (rollout - this.Mean) ./ this.Std;
            legend = {this.MV this.CV this.RV};
        end

        function [mv_std, mv_mean, legend] = inverse_scale_params_mv(this)
            mv_std = this.Std(1:numel(this.MV));
            mv_mean = this.Mean(1:numel(this.MV));
            legend = {this.MV};
        end

        function tf = hasdata(this)
            tf = this.counts <= this.Rollouts;
        end

        function [data,info] = read(this)
            T = this.rollout_func();

            % Remove columns with standard deviation of zero
            T( :, this.Empty_columns ) = [];

            % normalize
            T = (T - this.Mean) ./ this.Std;

            % To forecast the values of future time steps of a sequence,
            % specify the responses to be the training sequences with values
            % shifted by one time step. That is, at each time step of the input sequence,
            % the LSTM network learns to predict the value of the next time step.
            forecast_window = 1;

            % Separate the data into chunks of sequence length (skewed by forecast_window
            sequence_length = this.Sequence_length;

            % The predictors are the training sequences without the final time step.
            % Transform data to be supervised learning
            pointer = 1;
            data = [];
            room = 2*sequence_length-forecast_window;
            while pointer < size(T,1)-room
                % Many to many (OutputMode = sequence)
                %x_end = pointer+sequence_length-forecast_window;
                %y_end = x_end+sequence_length;
                %X = T(pointer:x_end,:)';
                %Y = T(x_end+forecast_window:y_end,:)';

                % Many to one (OutputMode = last)
                x_end = pointer+sequence_length-forecast_window;
                X = T(pointer:x_end,:)';
                Y = T(x_end+forecast_window,:)';

                data = [data; {X Y}];
                pointer = pointer + forecast_window;
            end

            info = [];
            this.counts = this.counts + 1;
        end

        function reset(this)
            this.counts = 0;
        end

        function subds = partition(ds,n,ii)
            subds = copy(ds);
            reset(subds);
        end

```

```
    end

    methods(Access = protected)
        function n = maxpartitions(ds)
            n = 30;
        end
    end

    methods (Hidden = true)
        function frac = progress(this)
            frac = this.counts / this.Rollouts;
        end
    end

end
```

Listing A.2: RolloutDatastore.m

A.2.2 HistorianDatastore.m

```
% A custom datastore for reading and presenting data from Osisoft PI Historian
% Data is scaled based on the largest dataset and pre-processed accordingly
classdef HistorianDatastore < matlab.io.Datastore & ...
                              matlab.io.datastore.Partitionable

    properties
        featureDimension
        Mean
        Std
        Sequence_length
        MV % manipulated variables
        CV % controlled variables
        RV % responding variables
    end

    properties %(Access = private)
        sequence_counter
        T
        currentT
        pointer
        forecast_window
        tag_function
        filter_function
        source_data
        miniBatchSize
        created
        mirrored_output
        ipredictors
        iresponses
        add_noise
    end

    methods

        function this = HistorianDatastore(sequence_length, tag_function, filter_function, ...
                starttime, endtime, interval, resample_interval, gap_threshold, ...
                mirrored_output, add_noise)
            this.Sequence_length = sequence_length;
            this.tag_function = tag_function;
            this.filter_function = filter_function;
            this.created = datetime('now');
            this.mirrored_output = mirrored_output;
            this.add_noise = add_noise;

            if endtime == '*'
                [D, MV, CV, RV] = this.read_data_single(starttime, endtime, ...
                    interval, resample_interval);
            else
                [D, MV, CV, RV] = this.read_data(starttime, endtime, ...
                    interval, resample_interval);
            end
            this.source_data = D;

            % Separate predictors and responses
            if mirrored_output % autoencoder
                this.ipredictors = 1:size(D,2);
                this.iresponses = 1:size(D,2);
            else
                this.ipredictors = 1:numel([MV' CV']);
                this.iresponses = numel(this.ipredictors)+1:numel(this.ipredictors)+numel(RV');
            end

            % Find time gaps in data
            gap = diff(D.Timestamp);
            idx = find(gap > gap_threshold);
            if numel(idx) == 0 % if no gaps, treat as one sequence
                idx = [size(D,1)];
            end

            % Break into sequences
            X={};
            prev = 1;
            for i = 1:length(idx)
                X{i,1} = cell2mat(table2cell(D(prev:idx(i),:)))';
                prev = idx(i)+1;
            end

            % Get the sequence lengths for each observation.
            sequenceLengths = [];
            for i=1:numel(X)
                sequence = X{i};
                sequenceLengths(i) = size(sequence,2);
            end
            [this.miniBatchSize, max_index] = max(sequenceLengths);
            idx = 1:numel(sequenceLengths);

            % Normalization parameter calculation on all data
            this.Mean = cell2mat(table2cell(varfun(@mean,D)));
            this.Std = cell2mat(table2cell(varfun(@std,D)));
            this.featureDimension = size(this.Mean,2);

            % Remove short sequences
            mask = sequenceLengths < (this.Sequence_length*2);
            idx(mask) = [];
            this.T = X(idx);

            % Stop if we have no sequences
            if numel(idx) == 0
                error("No sequences satisfying conditions the found")
            end

            % Set initial pointers
            reset(this);

            this.MV = MV';
            this.CV = CV';
            this.RV = RV';
        end

        function [D, MV, CV, RV] = read_data(this, starttime, endtime, interval, resample_interval)
            % Read data from PI in chunks
            [MV, CV, RV] = this.tag_function();

            D = [];
            currentt = datetime( starttime );
            while currentt < endtime
                stt = datestr(currentt,'yyyy-mm-dd');
                ent = datestr(currentt + calmonths(1) - days(1),'yyyy-mm-dd');
                d = get_data([MV;CV;RV],'startTime', stt, 'endTime', ent, ...
                    'interval', interval, 'resample_interval', resample_interval);
                D = [D;d];
                currentt = currentt + calmonths(1);
            end
            D = this.post_process_data(D);
        end

        function [D, MV, CV, RV] = read_data_single(this, starttime, endtime, interval, resample_interval)
            % Read data from PI
            [MV, CV, RV] = this.tag_function();
            D = get_data([MV;CV;RV],'startTime', starttime, 'endTime', endtime, ...
                'interval', interval, 'resample_interval', resample_interval);
            D = this.post_process_data(D);
        end

        function D = post_process_data(this, data)
            % Filter unwanted data
            [D, smoothing, fill] = this.filter_function(data);

            % Smooth data
            if smoothing
                D = smoothdata(D);
            end

            % Fill missing data
            if fill
                D = fillmissing(D,'constant',0);
            end
        end

        function this = replace_data(this, starttime, endtime, interval, resample_interval)
            [D, ~, ~, ~] = read_data_single(this, starttime, endtime, interval, resample_interval);

            %TODO Calculate starttime
            %This function's intention is to replace sequencelength amount
            %of data back in time, for resetting the generative model with
            %live data

            this.T = [];
            this.T{1,1} = cell2mat(table2cell(D))';

            % Set initial pointers
            reset(this);
        end

        function [rollout, legend] = inverse_scale(this, rollout)
            rollout = (rollout .* this.Std) + this.Mean;
            legend = {this.MV this.CV this.RV};
        end

        function [rollout, legend] = scale(this, rollout)
            rollout = (rollout - this.Mean) ./ this.Std;
            legend = {this.MV this.CV this.RV};
        end

        function [rollout, legend] = scale_predictors(this, rollout)
            rollout = (rollout - this.Mean(this.ipredictors)) ./ this.Std(this.ipredictors);
            legend = {this.MV this.CV};
        end

        function [rollout, legend] = inverse_scale_responses(this, rollout)
            rollout = (rollout .* this.Std(this.iresponses)) + this.Mean(this.iresponses);
            legend = {this.RV};
        end

        function [mv_std, mv_mean, legend] = inverse_scale_params_mv(this)
            mv_std = this.Std(1:numel(this.MV));
            mv_mean = this.Mean(1:numel(this.MV));
            legend = {this.MV};
        end

        function tf = hasdata(this)
            % return true if there are more sequences available
            tf = this.sequence_counter < size(this.T,1);
        end

        function tf = hasdata_in_sequence(this)
            room = this.Sequence_length - this.forecast_window;
            tf = this.pointer < size(this.currentT,1) - room;
        end

        function [data,info] = read(this)
            % Noise to be added to the signals
            sigma = 0.001;

            data = [];

            if ~hasdata_in_sequence(this)
                next_sequence(this);
            end

            if hasdata_in_sequence(this)
                % Many to one (OutputMode = last)
                x_end = this.pointer + this.Sequence_length - this.forecast_window;
                X = this.currentT(this.pointer:x_end,this.ipredictors)';
                Y = this.currentT(x_end + this.forecast_window,this.iresponses)';

                % Add noise to all signal channels
                if this.add_noise
                    for i=1:size(X,1)
                        X(i,:) = addNoise(sigma, X(i,:));
                    end
                    for i=1:size(Y,1)
                        Y(i,:) = addNoise(sigma, Y(i,:));
                    end
                end

                data = [{X Y}];
                this.pointer = this.pointer + this.forecast_window;
            end

            info = [];
        end

        function reset(this)
            this.sequence_counter = 0;
            this.forecast_window = 1;

            this.next_sequence();
        end

        function next_sequence(this)
            this.pointer = 1;
            this.sequence_counter = this.sequence_counter + 1;

            % Randomly select a sequence
            this.currentT = randsample(this.T,1);
            this.currentT = this.currentT{1}';

            % normalize
            this.currentT = this.scale(this.currentT);
        end

        function subds = partition(ds,n,ii)
            subds = copy(ds);
            reset(subds);
        end

    end

    methods(Access = protected)
        function n = maxpartitions(ds)
            n = 30;
        end
    end

    methods (Hidden = true)
        function frac = progress(this)
            frac = this.sequence_counter / size(this.T,1);
        end
    end

 end
```

Listing A.3: HistorianDatastore.m

A.2.3 get_data.m

The `get_data` function is a low level call towards the *OSISoft Web API*[2] used by the class `HistorianDatastore`.

```
function T = get_data(tags, varargin)
    api = <url to pi web api server>;
    options = weboptions('CertificateFilename',<certificate>,'ContentType','json');

    % Parse input arguments
    narginchk(1,inf)
    params = inputParser;
    params.CaseSensitive = false;
    params.addParameter('startTime', '-1h', @(x) ischar(x));
    params.addParameter('endTime', '*', @(x) ischar(x));
    params.addParameter('interval', '1m', @(x) ischar(x));
    params.addParameter('resample_interval', 1, @(x) isnumeric(x));
    params.parse(varargin{:});

    T = [];
    for i = 1:length(tags)
        points = webread([api '/points'],'path',tags{i},options);
        response = webread([api '/streamsets/interpolated'],'webId',points.WebId,...
        'startTime',params.Results.startTime,'endTime',params.Results.endTime,'interval',...
        params.Results.interval,options);

        for j = 1:numel(response.Items.Items);
            if ~isnumeric(response.Items.Items(j).Value)
                response.Items.Items(j).Value = nan;
            end
        end
        D = struct2table(response.Items.Items);

        D.Properties.VariableNames{2} = tags{i};
        D.Timestamp = erase(D.Timestamp, 'Z');
        D.Timestamp = datetime(D.Timestamp);
        D = removevars(D,{'UnitsAbbreviation','Good',...
            'Questionable','Substituted','Annotated'});

        D = table2timetable(D);
        if isempty(T)
            T = D;
        else
            T = synchronize(T,D,'regular','linear','TimeStep',...
                minutes(params.Results.resample_interval));
        end
    end
end
```

A.3 Generative Model

The `GenerativeModel` class is used as environment simulator for the `SyntheticEnvironment` class.

The model uses the choice of datastore during instantiation:

[2]https://docs.osisoft.com/bundle/pi-web-api-reference

```
genmodel = GenerativeModel(name, datastore, epochs, minibatchsize, numHiddenUnits, ...
    numHiddenLayers, numLatent, sequence_length, environment_length)
```

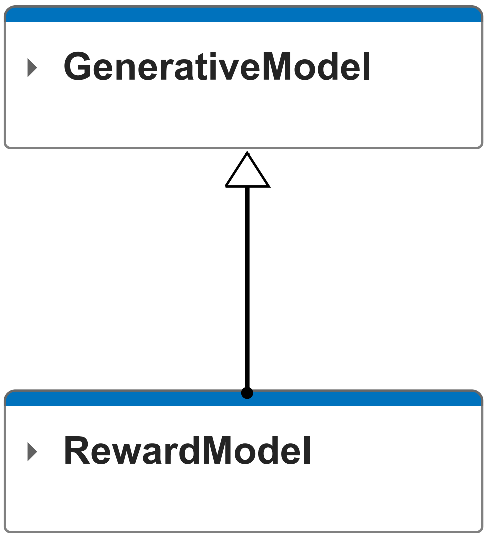


Figure A.2: Class diagram of generative models

A class `RewardModel` inherits all properties and methods from the `GenerativeModel` class.

A.3.1 GenerativeModel.m

```
% Class for handling training of generative model (historical process data)
classdef GenerativeModel

    properties
        Name
        Action
        Observation
        ObservationSequence
    end

    properties %(Access = protected)
        layers
        options
        net
        info
        filename
        datastore
        sequence_length
        environment_length
        initial_observation
        zeroMapping % bias of network
    end

    methods
        % Contructor method
        function this = GenerativeModel(name, datastore, epochs, minibatchsize, ...
                numHiddenUnits, numHiddenLayers, numLatent, sequence_length, environment_length)
            this.Name = name;
            this.datastore = datastore;
            this.sequence_length = sequence_length;
            this.environment_length = environment_length;
            this.initial_observation = [];

            % Define autoencoder network
            this.layers = this.lstm_network(numHiddenUnits, numHiddenLayers, numLatent);

            % Set Training Options
            this.options = this.train_options(epochs, minibatchsize);

            % Load existing file
            this.filename = [name '.mat'];
            if isfile(this.filename)
                load(this.filename, 'net', 'datastore', 'initial_observation', 'info');
                this.net = net;
                this.datastore = datastore; % override the above init
                this.info = info;
                this.initial_observation = initial_observation;
            end

            % read the initial sequence, our starting point for RL
            if isempty(this.initial_observation)
                this.initial_observation = read(this.datastore);
                this.initial_observation{1,1};
            end
        end

        function [this, info] = train_model(this)
            % Train the mean network
            % Assume that the mean response, (y|x), is normally distributed
            [this.net, info] = trainNetwork(this.datastore, this.layers, this.options);
            this.info = info;
        end

        function this = plot_training(this)
            figure
            x = linspace(1,numel(this.info.TrainingLoss),numel(this.info.TrainingLoss));
            plot(x, this.info.TrainingRMSE,"LineWidth",0.1)
            ylabel("RMSE")
            yyaxis right
            plot(x, this.info.BaseLearnRate,"LineWidth",1)
            legend(["Validation loss", "Learning rate"])
            ylabel("lr")
            xlabel("Iteration")
        end

        function [this, info] = continue_train_model(this, initial_lr)
            this.options.InitialLearnRate = initial_lr;
            [this.net, info] = trainNetwork(this.datastore, this.net.Layers, this.options);
            this.info = info;
        end

        function valError = test_model(this, rollout_func)
            [T, MV, CV, RV] = rollout_func();

            T = T(:,[1 2 9 10 11]); % TODO column selection net<-->ode
            [T,~] = this.datastore.scale(T);

            resetState(this.net);
            err=[];
            for i=1:size(T,1)-this.sequence_length
                X = T(i:i+this.sequence_length,:);
                Y = T(this.sequence_length+this.environment_length,:);
                [this.net, pred] = predictAndUpdateState(this.net,{X'});
                err(i) = sum(Y) - sum(pred);
            end
            valError = sqrt(mean(err).^2);
        end

        function save_model(this, info)
```

```
            net = this.net;
            datastore = this.datastore;
            initial_observation = this.initial_observation;
            info = this.info;
            save(this.filename, 'net', 'datastore', 'initial_observation', 'info');
        end

        function [ObservationInfo,ActionInfo] = get_environment(this)
            obs = [this.datastore.MV this.datastore.CV this.datastore.RV];
            obs_size = size(obs, 2);
            act_size = size(this.datastore.MV, 2);

            ObservationInfo = rlNumericSpec([obs_size this.environment_length]);
            ObservationInfo.Name = [this.Name ' Observation'];
            ObservationInfo.Description = strjoin(obs);

            % Initialize Action settings
            ActionInfo = rlNumericSpec([act_size 1],'LowerLimit',-15,'UpperLimit',15);
            ActionInfo.Name = [this.Name ' Action'];
            ActionInfo.Description = strjoin(this.datastore.MV);
        end

        function this = reset(this)
            this = initialize_network(this, 60);

            % Reset initial observation
            this.datastore.reset();
            this.initial_observation = read(this.datastore);

            O = this.initial_observation{1,1};
            A = this.initial_observation{1,2};

            % split into action and observation space
            act_size = size(this.datastore.MV, 2);
            this.Observation = O(:,end-this.environment_length+1:end);
            this.ObservationSequence = O; % towards generative model
            this.Action = A(1:act_size);
        end

        function this = step(this, actions)
            this.Action = actions;

            % Predict next step by changing action but keep observation
            % input to the network
            act_size = size(this.datastore.MV, 2);
            T = this.ObservationSequence;
            T(1:act_size,end) = this.Action; % replace last action vector in observation

            % Predict one step, network accepts sequence input only
            [this.net, pred] = predictAndUpdateState(this.net,{T});
            pred = pred - this.zeroMapping;

            % Add some noise
            sigma = 0.005;
            pred = addNoise(sigma, pred);

            pred = pred'; % for many-to-one models

            % Update states, slide window one step.
            %pred(1:act_size) = this.Action; % uncomment for disabling dynamics prediction
            this.ObservationSequence(:,this.sequence_length+1) = pred;
            this.ObservationSequence(:,1) = [];
            this.Observation = this.ObservationSequence(:,end);
        end

        function [rollout, legend] = inverse_scale(this, rollout)
           [rollout, legend] = this.datastore.inverse_scale(rollout);
        end

        function [rollout, legend] = scale(this, rollout)
           [rollout, legend] = this.datastore.scale(rollout);
        end

        % Return parameters needed for inverse scaling of Manipulated Variables
        function [mv_std, mv_mean, legend] = inverse_scale_params_mv(this)
            [mv_std, mv_mean, legend] = this.datastore.inverse_scale_params_mv();
        end

        function this = initialize_network(this, initialization_period)
            initializationSignal = zeros(this.datastore.featureDimension, ...
                this.sequence_length * initialization_period);
            this.net = resetState(this.net);
            this.net = predictAndUpdateState(this.net, initializationSignal);
            this.zeroMapping = predict(this.net, initializationSignal);
        end
    end

    methods %(Access = protected)
        function layers = lstm_network(this, numHiddenUnits, numHiddenLayers, numLatent)
            % Define input
            layers = [
                sequenceInputLayer(this.datastore.featureDimension)
            ];

            % Define decoder
            for i=1:numHiddenLayers
                layers = [
                    layers
                    lstmLayer(numLatent)
                    dropoutLayer(0.2)
                ];
            end

            % Define output
            layers = [
                layers
                lstmLayer(numHiddenUnits, "OutputMode", "last")
                dropoutLayer(0.2)
                fullyConnectedLayer(this.datastore.featureDimension)
                huberRegressionLayer('huber')
            ];

        end

        function options = train_options(this, epochs, minibatchsize)
            numberOfWorkers = 1;
            minisize = minibatchsize * numberOfWorkers;
            initialLearnRate = 0.05 * minisize/minibatchsize;

            options = trainingOptions('adam', ...
                'Plots', 'training-progress', ...
                'Verbose', true, ...
                'GradientThreshold',1, ...
                'InitialLearnRate',initialLearnRate, ...
                'LearnRateSchedule','piecewise', ...
                'LearnRateDropPeriod',epochs/6, ...
                'LearnRateDropFactor',0.2, ...
                'MaxEpochs',epochs,...
                'Shuffle','never', ...
                'MiniBatchSize',minisize);
        end
    end

end
```

Listing A.4: GenerativeModel.m

A.3.2 RewardModel.m

```
classdef RewardModel < GenerativeModel

    methods

    end

end
```

Listing A.5: RewardModel.m

A.4 Rollout functions

A function pointer to any of the rollout functions is inputted as argument to the instantiation of the `RolloutDatastore`.

A.4.1 spray_rollout_pid.m

```
% Generate process data table T with stoichrometic model
% The process step response is recorded
function [T, MV, CV, RV] = spray_rollout_pid

% Manipulated variables: Q, 10-minute steps
MV = ["Q","rpm"];

t = linspace(1,60*10,60*10)';

a = 2;
b = 20;
r = (b-a).*rand(1,1) + a;
Q = repelem(r, length(t))'; % start value is randomized

% Tsp-scenario, randomize sp every n step
Tsp = repelem(1, length(t))';
n_step_r = 60;
for i=1:n_step_r:length(t)
    r = getrandom(88, 142); % orig: 88-92
    Tsp(i:i+n_step_r-1) = repelem(r, n_step_r);
end

% rpm-scenario, randomize sp
rpm = repelem(7600, length(t))';
n_step_r = floor(length(t)/3);
for i=1:n_step_r:length(t)
    r = randsample([7600, 8000, 8600, 9200],1);
    rpm(i:i+n_step_r-1) = repelem(r, n_step_r);
end

% Controlled variables
CV = ["Tin", "Qg", "Tg", "RHg", "pfeed", "Tfeed"]; %, "tsignal"];
sigma = 0.005;
%rpm = addNoise(sigma, repelem(7600,length(Q)))'; % atomizer rpm
Tin = addNoise(sigma, repelem(180,length(Q)))'; % gas temp in
Qg = addNoise(sigma, repelem(100000,length(Q)))'; % gas flow in m3/h
Tg = addNoise(sigma, repelem(25,length(Q)))'; % outside temp
RHg = addNoise(sigma, repelem(88,length(Q)))'; % gas RH in
pfeed = addNoise(sigma, repelem(1200,length(Q)))'; % density feed
Tfeed = addNoise(sigma, repelem(95,length(Q)))'; % feed temperature
%tsignal = repmat(getsine(1, 1, 0),[1 6])';

% Responding variables
RV = ["Tout", "DropletSZ", "Hum"];
Tout = repelem(nan,length(Q))';
DropSZ = repelem(nan,length(Q))';
Hum = repelem(nan,length(Q))';

T = table(Q, rpm, Tin, Qg, Tg, RHg, pfeed, Tfeed, Tout, DropSZ, Hum); %, tsignal);
T = table2array(T);

tStep = linspace(1,numel(Q),numel(Q));
y0 = [1 1 7600 95 100 5 0.65]; % Qstate Qgstate rpmstate Tout Dropletsz Hum bv
T(1,9) = y0(4); % Tout
```

```
T(1,10) = y0(5); % Dropletsz
T(1,11) = y0(6); % Hum

tau = 60;

for index = 2:numel(tStep)
    % select row
    Q = T(index-1,1); % previous one
    rpm = T(index,2);
    Tin = T(index,3);
    Qg = T(index,4);
    Tg = T(index,5);
    RHg = T(index,6);
    pfeed = T(index,7);
    Tfeed = T(index,8);

    % Regulate Q against Tsp and Tactual
    %MV = Kp * (SP - PV)
    Qdot = -0.01 * (Tsp(index) - T(index-1,9));
    Q = Q + Qdot;

    % Integrate
    af   = @(t,y) spray_f(t,y,Q,Qg,Tfeed,Tin,RHg,pfeed,rpm,tau);
    t    = tStep(index-1:index); % 1 minute
    [t, y] = ode45(af, t, y0);

    % Final value of x is initial value for next step
    y0 = y(end, :);

    % Collect the results
    sigmaR = 0.001; % noise on Responding
    T(index,1) = Q;
    T(index,2) = y(end, 3);
    T(index,9) = addNoise(sigmaR,  y(end, 4) );
    T(index,10) = y(end, 5);
    T(index,11) = y(end, 6);
end
T=T(60:end,:); % clip first frames (warmup)

end
```

Listing A.6: spray_rollout_pid.m

A.4.2 spray_rollou_pid_stable.m

For reward model

```
% Generate process data table T with stoichrometic model
% The process step response is recorded
function [T, MV, CV, RV] = spray_rollout_pid_stable

% Manipulated variables: Q, 10-minute steps
MV = ["Q","rpm"];

t = linspace(1,60*10,60*10)';

a = 2;
b = 20;
r = (b-a).*rand(1,1) + a;
Q = repelem(r, length(t))'; % start value is randomized

% Stable Tsp-scenario
Tsp = repelem(110, length(t))';

% Controlled variables
CV = ["Tin", "Qg", "Tg", "RHg", "pfeed", "Tfeed"]; %, "tsignal"];
sigma = 0.001;
rpm = addNoise(sigma, repelem(7600,length(Q)))'; % atomizer rpm
sigma = 0.001;
Tin = addNoise(sigma, repelem(180,length(Q)))'; % gas temp in
Qg = addNoise(sigma, repelem(100000,length(Q)))'; % gas flow in m3/h
```

```
Tg = addNoise(sigma, repelem(25,length(Q)))'; % outside temp
RHg = addNoise(sigma, repelem(88,length(Q)))'; % gas RH in
pfeed = addNoise(sigma, repelem(1200,length(Q)))'; % density feed
Tfeed = addNoise(sigma, repelem(95,length(Q)))'; % feed temperature
%tsignal = repmat(getsine(1, 1, 0),[1 6])';

% Responding variables
RV = ["Tout", "DropletSZ", "Hum"];
Tout = repelem(nan,length(Q))';
DropSZ = repelem(nan,length(Q))';
Hum = repelem(nan,length(Q))';

T = table(Q, rpm, Tin, Qg, Tg, RHg, pfeed, Tfeed, Tout, DropSZ, Hum); %, tsignal);
T = table2array(T);

tStep = linspace(1,numel(Q),numel(Q));
y0 = [1 1 7600 95 100 5 0.65]; % Qstate Qgstate rpmstate Tout Dropletsz Hum bv
T(1,9) = y0(4); % Tout
T(1,10) = y0(5); % Dropletsz
T(1,11) = y0(6); % Hum

tau = 60;

for index = 2:numel(tStep)
    % select row
    Q = T(index-1,1); % previous one
    rpm = T(index,2);
    Tin = T(index,3);
    Qg = T(index,4);
    Tg = T(index,5);
    RHg = T(index,6);
    pfeed = T(index,7);
    Tfeed = T(index,8);

    % Regulate Q against Tsp and Tactual
    %MV = Kp * (SP - PV)
    Qdot = -0.01 * (Tsp(index) - T(index-1,9));
    Q = Q + Qdot;

    % Integrate
    af   = @(t,y) spray_f(t,y,Q,Qg,Tfeed,Tin,RHg,pfeed,rpm,tau);
    t    = tStep(index-1:index); % 1 minute
    [t, y] = ode45(af, t, y0);

    % Final value of x is initial value for next step
    y0 = y(end, :);

    % Collect the results
    sigmaR = 0.001; % noise on Responding
    T(index,1) = Q;
    T(index,2) = y(end, 3);
    T(index,9) = addNoise(sigmaR,  y(end, 4) );
    T(index,10) = y(end, 5);
    T(index,11) = y(end, 6);
end
T=T(60:end,:); % clip first frames (warmup)

end
```

Listing A.7: spray_rollout_pid_stable.m

A.5 RL Environments

The RL environment classes are used for training an RL agent. The `SyntheticEnvironment` class is a bridge between the `GenerativeModel` and the environment, where action proposals from the agent is comitted through the generative model.

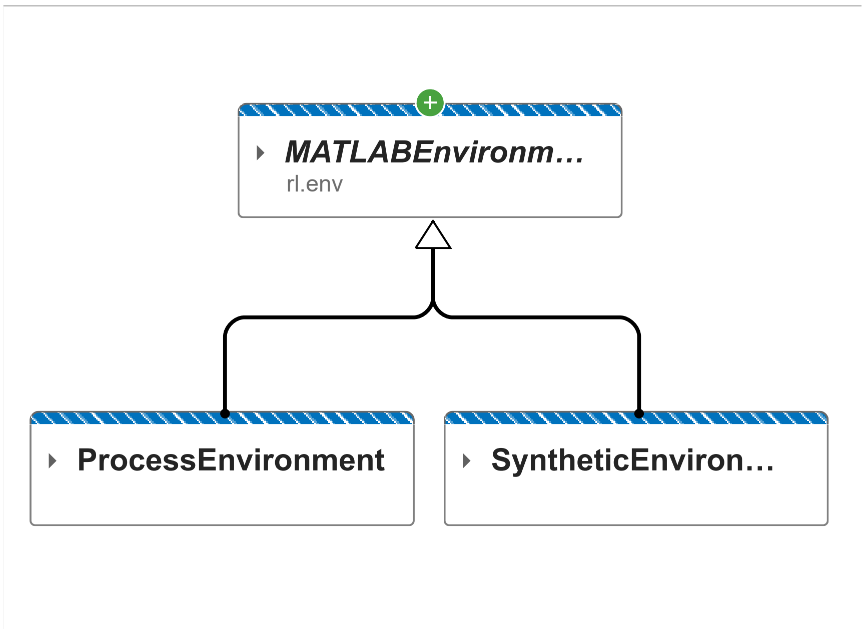


Figure A.3: Class diagram of implemented RL environments

Instantiating the class is done by first specifying the generative model, the reward function and any reward model, if one exists:

```
env = SyntheticEnvironment(genmodel, doplot, @reward_func, reward_model);validateEnvironment(env);
```

where `@reward_func` is a function pointer to one of the implemented reward functions:

- `direct_response_reward`
- `sparse_reward`
- `learnt_reward`

The `ProcessEnvironments` uses the `spray_f` function for real time simulation towards the spray drier ordinary differential equations, and can be used for both training and agent validation:

```
envProcess = ProcessEnvironment(genmodel, false, sigma, @reward_func, reward_model);
validateEnvironment(envProcess);
```

In a case where no process data exists, but a model of the process is implemented as solvable ODE's, the `ProcessEnvironment` can be used as the primary training environment for an agent.

A.5.1 SyntheticEnvironment.m

```
classdef SyntheticEnvironment < rl.env.MATLABEnvironment

    properties
        Rollout
    end

    properties %(Access = protected)
        generative_model
        plot_sequence
        save_sequence
        Figure
        filename
        m
        reward_function
        reward_model
    end

    methods
        % Contructor method creates an instance of the environment
        function this = SyntheticEnvironment(generative_model, plot_sequence, reward_function, reward_mo
            % Query the generative model for the environment
            [ObservationInfo,ActionInfo] = generative_model.get_environment;
            this = this@rl.env.MATLABEnvironment(ObservationInfo,ActionInfo);
            this.generative_model = generative_model;
            this.plot_sequence = plot_sequence;
            this.save_sequence = plot_sequence;
            this.reward_function = reward_function;
            this.reward_model = reward_model;

            %time = datestr(now, 'yyyy_mm_dd_hh_MM_ss');
            %this.filename = sprintf('savedEpisodes\\training_%s.mat',time);
            %m = matfile(filename, 'Writable', true);
        end

        % Apply system dynamics and simulates the environment with the
        % given action for one step.
        function [Observation,Reward,IsDone,LoggedSignals] = step(this,Action)
            LoggedSignals = [];

            % Scale incoming action
            [mv_std, mv_mean, ~] = this.generative_model.inverse_scale_params_mv();
            action_scaled = (Action .* mv_std') + mv_mean';

            % Apply action to the generative model
            previous_action = (this.generative_model.Action .* mv_std') + mv_mean';
            this.generative_model = this.generative_model.step(Action);
            Observation = this.generative_model.Observation;

            scaled_world = [this.generative_model.Observation(:,end)'];
            [scaled_world, ~] = this.generative_model.inverse_scale(scaled_world);
            this.Rollout = [this.Rollout;scaled_world];

            [Reward, IsDone, this.reward_model] = this.reward_function(previous_action, action_scaled, .
                scaled_world, this.Rollout, this.generative_model.sequence_length, ...
                this.generative_model, this.reward_model);

            notifyEnvUpdated(this);
        end

        % Reset environment to initial state and output initial observation
        function InitialObservation = reset(this)
            % plot previous (and complete) episode
            if this.plot_sequence
                plotSequence(this);
            end
            if this.save_sequence
                saveSequence(this);
            end

            % reset model
            this.generative_model = this.generative_model.reset;
            InitialObservation = this.generative_model.Observation;

            scaled_world = [this.generative_model.Observation(:,end)'];
            [scaled_world, ~] = this.generative_model.inverse_scale(scaled_world);
            this.Rollout = scaled_world;

            if ~isempty(this.reward_model)
                this.reward_model = this.reward_model.initialize_network(10); % TODO param!
            end

            notifyEnvUpdated(this);
        end
    end

    methods
        % (optional) Visualization method
        function plot(this)
            % Initiate the visualization

            % Update the visualization
            envUpdatedCallback(this)
        end

        function plotSequence(this)
            if ~isempty(this.Rollout)
                %tiledlayout(2,1);
                %nexttile
                plot(this.Rollout(:,end)) %3
                ylim([80 150]);
                yyaxis right
                plot(this.Rollout(:,1))
                ylim([1 20]);
                yyaxis left
            end
        end

        function saveSequence(this)
            if ~isempty(this.Rollout)
                %m = matfile(this.filename, 'Writable', true);
                %[nrows,ncols] = size(m,'episode');
                %m.episode(end+1,:) = this.Rollout;
            end
        end

        function plotRollout(this)
            if ~isempty(this.Rollout)
                stackedplot(this.Rollout)
            end
        end
    end

    methods (Access = protected)
        % update visualization everytime the environment is updated
        % (notifyEnvUpdated is called)
        function envUpdatedCallback(this)
            %if ~isempty(this.Figure) && isvalid(this.Figure)

            %end
        end
    end
end
```

Listing A.8: SyntheticEnvironment.m

A.5.2 ProcessEnvironment.m

```
classdef ProcessEnvironment < rl.env.MATLABEnvironment

    properties
        Rollout
    end

    properties(Access = protected)
        generative_model
        plot_sequence
        Figure
        %tStep
        y0
        observation
        index
        tau
        previous_action
        reward_function
        reward_model
        sigma
    end

    methods
        % Contructor method creates an instance of the environment
        function this = ProcessEnvironment(generative_model, plot_sequence, sigma, ...
                reward_function, reward_model)
            % Query the generative model for the environment
            [ObservationInfo,ActionInfo] = generative_model.get_environment;
            this = this@rl.env.MATLABEnvironment(ObservationInfo,ActionInfo);
            this.generative_model = generative_model;
            this.plot_sequence = plot_sequence;
            this.tau = 60;
            this.sigma = sigma;
            this.reward_function = reward_function;
            this.reward_model = reward_model;
        end

        % Apply system dynamics and simulates the environment with the
        % given action for one step.
        function [Observation,Reward,IsDone,LoggedSignals] = step(this,Action)
            LoggedSignals = [];
            epsilon = 1e-5;
            this.index = this.index + 1;

            % Scale incoming action (cannot find bug, network should do it)
            [mv_std, mv_mean, ~] = this.generative_model.inverse_scale_params_mv();
            action_scaled = (Action .* mv_std') + mv_mean';
            Q = action_scaled(1);
            rpm = action_scaled(2);

            % Action damping
            %alfa = 0.9;
            %this.Q = (alfa * this.Q) + ((1-alfa) * Qp);
            %this.Q = Qp;

            % Step with Action
            %sigma = 0.001;
            %rpm = addNoise(sigma, 7600); % atomizer rpm
            Tin = addNoise(this.sigma, 180); % gas temp in
            Qg = addNoise(this.sigma, 100000); % gas flow in m3/h
            Tg = addNoise(this.sigma, 25); % outside temp
            RHg = addNoise(this.sigma, 88); % gas RH in
            pfeed = addNoise(this.sigma, 1200); % density feed
            Tfeed = addNoise(this.sigma, 95); % feed temperature

            % Integrate
            af   = @(t,y) spray_f(t,y,Q,Qg,Tfeed,Tin,RHg,pfeed,rpm,this.tau);
            t    = [this.index this.index+1]; %this.tStep(this.index:this.index+1); % 1 minute
            [t, y] = ode45(af, t, this.y0);

            % Final value of x is initial value for next step
            this.y0 = y(end, :);

            % Collect the results
            this.observation(1) = Q;
            this.observation(2) = rpm;
            this.observation(9) = y(end, 4); %addNoise(sigma,  y(end, 3) );
            this.observation(10) = y(end, 5);
            this.observation(11) = y(end, 6);
            Observation = this.generative_model.scale(this.observation)';
            this.Rollout = [this.Rollout;this.observation];

            [Reward, IsDone, this.reward_model] = this.reward_function(this.previous_action, ...
                action_scaled, this.observation, this.Rollout, this.generative_model.sequence_length, ...
                this.generative_model, this.reward_model);
            this.previous_action = action_scaled;
            IsDone = 0; % do not stop the process simulator

            notifyEnvUpdated(this);
        end

        % Reset environment to initial state and output initial observation
        function InitialObservation = reset(this)
            % plot previous (and complete) episode
            if this.plot_sequence
                plotSequence(this);
            end

            % Reset model
            % randomized
            initial_observation = read(this.generative_model.datastore);
            this.observation = (initial_observation{1}(:,end))';
            % static from genmodel
            %this.observation = (this.generative_model.initial_observation{1}(:,end))';
            [this.observation, ~] = this.generative_model.inverse_scale(this.observation);

            act_size = size(this.generative_model.datastore.MV, 2);
            this.previous_action = [this.observation(1:act_size)'];

            Q           = this.observation(1);
            rpmstate    = this.observation(2);
            Tout        = this.observation(3);
            Dropletsz   = this.observation(4);
            hum         = this.observation(5);
            % Qstate Qgstate Tout Dropletsz Hum bv
            this.y0 = [Q 100000 rpmstate Tout Dropletsz hum 0.65];
            InitialObservation = this.observation';
            %this.Q = Q;
            this.index = 0;

            % Make a log
            this.Rollout = [this.observation];
            notifyEnvUpdated(this);
        end
    end

    methods
        % (optional) Visualization method
        function plot(this)
            % Initiate the visualization

            % Update the visualization
            envUpdatedCallback(this)
        end

        function plotSequence(this)
            if ~isempty(this.Rollout)
                tiledlayout(2,1);
                nexttile
                plot(this.Rollout(:,3))
                ylim([80 100]);
                yyaxis right
                plot(this.Rollout(:,1))
                ylim([5 12]);
                yyaxis left

                %multivariate:
                %nexttile
                %histogram(this.Rollout(20:end,4))

                nexttile
                plot(this.Rollout(:,4))
                ylim([100 150]);
                yyaxis right
                plot(this.Rollout(:,2))
                ylim([7000 10000]);
                yyaxis left
            end
        end

        function plotRollout(this)
            if ~isempty(this.Rollout)
                stackedplot(this.Rollout)
            end
        end
    end

    methods (Access = protected)
        % (optional) update visualization everytime the environment is updated
        % (notifyEnvUpdated is called)
        function envUpdatedCallback(this)
            %if ~isempty(this.Figure) && isvalid(this.Figure)

            %end
        end
    end
end
```

Listing A.9: ProcessEnvironment.m

A.6 Reward functions

A choice of reward function is inputed to the instantiation of the RL environment classes.

A.6.1 direct_response_reward.m

```
function [Reward, IsDone, RewardModel] = direct_response_reward(previous_action, action_scaled, ...
    scaled_world, logged_signals, sequence_length, generative_model, reward_model)

    epsilon = 1e-5;

    % Setpoint offset penalty
    setpoint = 90;
    errTout = abs(setpoint - scaled_world(3)) + epsilon; % MV + CV + RV vectors
    rT = (1 / errTout);

    % Penalize control effort
    rA = -0.05*abs(previous_action(1) - action_scaled(1));

    % Get reward
    Reward = rT + rA;

    % Check terminal condition
    IsDone = errTout > 2;

    RewardModel = reward_model;
```

Listing A.10: direct_response_reward.m

A.6.2 sparse_reward.m

```
function [Reward, IsDone, RewardModel] = sparse_reward(previous_action, action_scaled, ...
    scaled_world, logged_signals, sequence_length, generative_model, reward_model)

    epsilon = 1e-5;

    % Shaped Sparse reward
    dist = makedist('normal','mu',122,'sigma',5); % target distribution
    rS = 0;
    h = 0;
    if size(logged_signals,1) > sequence_length+4
```

```
        theset=logged_signals(sequence_length:end,4);
        [h,p,adstat,cv] = adtest(theset',"Distribution", dist);
        rS = (~h);
    end

    % Get reward
    Reward = rS;

    % Check terminal condition
    IsDone = h;

    RewardModel = reward_model;
```

Listing A.11: sparsereward.m

A.6.3 learnt_reward.m

```
function [Reward, IsDone, RewardModel] = learnt_reward(previous_action, action_scaled, scaled_world, logged_si
    % Extract sequence
    trail_length = min(size(logged_signals,1), sequence_length)-1;
    data = logged_signals(end-trail_length:end-1,:);

    % Query the reward model
    data = reward_model.scale(data);
    [reward_model.net, Y] = predictAndUpdateState(reward_model.net,{data'});
    Y = reward_model.inverse_scale(Y);

    % Scale to similar range
    Y = generative_model.scale(Y);
    X = generative_model.scale(scaled_world);
    err = abs(sum(X-Y));
    mse = abs(sum((X-Y).^2));

    % Get reward
    Reward = 1/mse; %err

    % Check terminal condition
    IsDone = 0;
    if size(logged_signals,1) > sequence_length*2
        %IsDone = err > 2;
        IsDone = sqrt(mse) > 5;
        % reward model RMSE is lower
    end

    RewardModel = reward_model;
```

Listing A.12: learnt_reward.m

A.7 Agent neural networks

The SAC, PPO and DDPG agents uses similar network architectures for the actor and critics, adjusted for framework specific requirements. To train the different actors, the neural network functions below are used.

Example of creating a SAC Agent:

```
[actor, critic1, critic2] = agent_sac_v1_norm(genmodel);agentOptions =
    rlSACAgentOptions(..."SequenceLength",
    sequence_length,..."DiscountFactor",0.98,..."TargetSmoothFactor",1e-
    3,..."ExperienceBufferLength", 1e6,..."UseDeterministicExploitation", false,
    ..."ResetExperienceBufferBeforeTraining", false,..."SaveExperienceBufferWithAgent", true);agent
    = rlSACAgent(actor,[critic1 critic2],agentOptions);
```

Example of creating a PPO Agent:

```
[actor, critic] = agent_ppo_v1_norm(genmodel);
agentOpts = rlPPOAgentOptions(...
    "MiniBatchSize", sequence_length,...
    'ExperienceHorizon',1e6,...
    'DiscountFactor',0.98,...
    "UseDeterministicExploitation", false);
agent = rlPPOAgent(actor,critic,agentOpts);
```

Example of creating a DDPG Agent:

```
[actor, critic] = agent_ddpg_v1_norm(genmodel);

agentOpts = rlDDPGAgentOptions(...
    "SequenceLength", sequence_length,...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6,...
    'DiscountFactor',0.98,...
    "ResetExperienceBufferBeforeTraining", false,...
    "SaveExperienceBufferWithAgent", true);
agent = rlDDPGAgent(actor,critic,agentOpts);
```

All agents can then be trained in a similar way:

```
opt = rlTrainingOptions(...
    'MaxEpisodes',5000, ...
    'MaxStepsPerEpisode',600,... % minutes
    'Verbose', false, ...
    'Plots','training-progress',...
    'StopTrainingCriteria',"AverageReward",'StopTrainingValue',1800);

trainStats = train(agent,env,opt)
```

A.7.1 agent_sac_v1_norm.m

```
function [actor, critic1, critic2] = agent_sac_v1_norm(genmodel)

% Datastore sequence: MV, CV, RV
[obsInfo, actInfo] = genmodel.get_environment;
numObs = obsInfo.Dimension(1);
numAct = actInfo.Dimension(1);

action_mean = genmodel.datastore.Mean(1:numAct)';
action_std = genmodel.datastore.Std(1:numAct)';
observation_mean = genmodel.datastore.Mean';
observation_std = genmodel.datastore.Std';

% Critic
statePath1 = [
    sequenceInputLayer(numObs,'Normalization','zscore', 'Mean', observation_mean, ...
        'StandardDeviation', observation_std,'Name','observation')
    fullyConnectedLayer(400,'Name','CriticStateFC1')
    reluLayer('Name','CriticStateRelu1')
    fullyConnectedLayer(300,'Name','CriticStateFC2')
    ];
actionPath1 = [
    sequenceInputLayer(numAct,'Normalization','zscore', 'Mean', action_mean, ...
        'StandardDeviation', action_std,'Name','action')
    fullyConnectedLayer(300,'Name','CriticActionFC1')
    ];
commonPath1 = [
    additionLayer(2,'Name','add')
    lstmLayer(8,'OutputMode','sequence','Name','lstm')
    reluLayer('Name','CriticCommonRelu1')
    fullyConnectedLayer(1,'Name','CriticOutput')
    ];

criticNet = layerGraph(statePath1);
criticNet = addLayers(criticNet,actionPath1);
criticNet = addLayers(criticNet,commonPath1);
criticNet = connectLayers(criticNet,'CriticStateFC2','add/in1');
criticNet = connectLayers(criticNet,'CriticActionFC1','add/in2');

criticOptions = rlRepresentationOptions('Optimizer','adam','LearnRate',1e-3,...
                                        'UseDevice',"gpu",...
                                        'GradientThreshold',1,'L2RegularizationFactor',2e-4);
critic1 = rlQValueRepresentation(criticNet,obsInfo,actInfo,...
    'Observation',{'observation'},'Action',{'action'},criticOptions);
critic2 = rlQValueRepresentation(criticNet,obsInfo,actInfo,...
    'Observation',{'observation'},'Action',{'action'},criticOptions);

% Actor
statePath = [
    sequenceInputLayer(numObs,'Normalization','zscore', 'Mean', observation_mean, ...
        'StandardDeviation', observation_std,'Name','observation')
    fullyConnectedLayer(400, 'Name','commonFC1')
    lstmLayer(8,'OutputMode','sequence','Name','lstm')
    reluLayer('Name','CommonRelu')];
meanPath = [
    fullyConnectedLayer(300,'Name','MeanFC1')
    reluLayer('Name','MeanRelu')
    fullyConnectedLayer(numAct,'Name','Mean')
    ];
stdPath = [
    fullyConnectedLayer(300,'Name','StdFC1')
    reluLayer('Name','StdRelu')
    fullyConnectedLayer(numAct,'Name','StdFC2')
    softplusLayer('Name','StandardDeviation')];

concatPath = concatenationLayer(1,2,'Name','GaussianParameters');

actorNetwork = layerGraph(statePath);
actorNetwork = addLayers(actorNetwork,meanPath);
actorNetwork = addLayers(actorNetwork,stdPath);
actorNetwork = addLayers(actorNetwork,concatPath);
actorNetwork = connectLayers(actorNetwork,'CommonRelu','MeanFC1/in');
actorNetwork = connectLayers(actorNetwork,'CommonRelu','StdFC1/in');
actorNetwork = connectLayers(actorNetwork,'Mean','GaussianParameters/in1');
actorNetwork = connectLayers(actorNetwork,'StandardDeviation','GaussianParameters/in2');

actorOptions = rlRepresentationOptions('Optimizer','adam','LearnRate',1e-3,...
                                       'UseDevice',"gpu",...
                                       'GradientThreshold',1,'L2RegularizationFactor',1e-5);

actor = rlStochasticActorRepresentation(actorNetwork,obsInfo,actInfo, ...
    actorOptions,'Observation',{'observation'});
```

Listing A.13: agent_sac_v1_norm.m

A.7.2 agent_ppo_v1_norm.m

```
function [actor, critic] = agent_ppo_v1_norm(genmodel)

% Datastore sequence: MV, CV, RV
[obsInfo, actInfo] = genmodel.get_environment;
numObs = obsInfo.Dimension(1);
numAct = actInfo.Dimension(1);

action_mean = genmodel.datastore.Mean(1:numAct)';
action_std = genmodel.datastore.Std(1:numAct)';
observation_mean = genmodel.datastore.Mean';
observation_std = genmodel.datastore.Std';

% Critic
criticNet = [
    sequenceInputLayer(numObs,'Normalization','zscore', 'Mean', observation_mean, ...
        'StandardDeviation', observation_std,'Name','observation')
    fullyConnectedLayer(400,'Name','CriticStateFC1')
    reluLayer('Name','CriticStateRelu1')
    fullyConnectedLayer(300,'Name','CriticStateFC2')
    reluLayer('Name','CriticRelu2')
    lstmLayer(8,'OutputMode','sequence','Name','lstm')
    fullyConnectedLayer(1,'Name','CriticOutput')
    regressionLayer('Name','RepresentationLoss')
];
```

```

criticNet = layerGraph(criticNet);

criticOptions = rlRepresentationOptions('Optimizer','adam','LearnRate',1e-3,...
                                        'UseDevice',"cpu",...
                                        'GradientThreshold',1,'L2RegularizationFactor',2e-4);
critic = rlValueRepresentation(criticNet,obsInfo,'Observation',{'observation'},criticOptions);

% Actor
actorNetwork = layerGraph();
tempLayers = [
    sequenceInputLayer(numObs,'Normalization','zscore', 'Mean', observation_mean, ...
        'StandardDeviation', observation_std,'Name','observation')
    fullyConnectedLayer(400,"Name","fc_1")
    reluLayer("Name","relu_body")
    fullyConnectedLayer(300,"Name","fc_body")
    reluLayer("Name","body_output")
    lstmLayer(8,"Name","lstm")];
actorNetwork = addLayers(actorNetwork,tempLayers);

tempLayers = [
    fullyConnectedLayer(numAct,"Name","fc_std")
    softplusLayer("Name","std")];
actorNetwork = addLayers(actorNetwork,tempLayers);

tempLayers = [
    fullyConnectedLayer(numAct,"Name","fc_mean")
    tanhLayer("Name","tanh")
    ];
actorNetwork = addLayers(actorNetwork,tempLayers);

tempLayers = [
    concatenationLayer(1,2,"Name","output")
    regressionLayer("Name","RepresentationLoss")];
actorNetwork = addLayers(actorNetwork,tempLayers);

actorNetwork = connectLayers(actorNetwork,"lstm","fc_std");
actorNetwork = connectLayers(actorNetwork,"lstm","fc_mean");
actorNetwork = connectLayers(actorNetwork,"std","output/in2");
actorNetwork = connectLayers(actorNetwork,"tanh","output/in1"); %scale

actorOptions = rlRepresentationOptions('Optimizer','adam','LearnRate',1e-3,...
                                       'UseDevice',"cpu",...
                                       'GradientThreshold',1,'L2RegularizationFactor',1e-5);

actor = rlStochasticActorRepresentation(actorNetwork,obsInfo,actInfo, ...
    'Observation',{'observation'},actorOptions);
```

Listing A.14: agent_ppo_v1_norm.m

A.7.3 agent_ddpg_v1_norm.m

```
function [actor, critic] = agent_ddpg_v1_norm(genmodel)

% Datastore sequence: MV, CV, RV
[obsInfo, actInfo] = genmodel.get_environment;
numObs = obsInfo.Dimension(1);
numAct = actInfo.Dimension(1);

action_mean = genmodel.datastore.Mean(1:numAct)';
action_std = genmodel.datastore.Std(1:numAct)';
observation_mean = genmodel.datastore.Mean';
observation_std = genmodel.datastore.Std';

% Critic
statePath1 = [
    sequenceInputLayer(numObs,'Normalization','zscore', 'Mean', observation_mean, ...
        'StandardDeviation', observation_std,'Name','observation')
    fullyConnectedLayer(400,'Name','CriticStateFC1')
    reluLayer('Name','CriticStateRelu1')
    fullyConnectedLayer(300,'Name','CriticStateFC2')
    ];
actionPath1 = [
    sequenceInputLayer(numAct,'Normalization','zscore', 'Mean', action_mean, ...
        'StandardDeviation', action_std,'Name','action')
    fullyConnectedLayer(300,'Name','CriticActionFC1')
    ];
commonPath1 = [
    additionLayer(2,'Name','add')
    lstmLayer(8,'OutputMode','sequence','Name','lstm')
    reluLayer('Name','CriticCommonRelu1')
    fullyConnectedLayer(1,'Name','CriticOutput')
    ];

criticNet = layerGraph(statePath1);
criticNet = addLayers(criticNet,actionPath1);
criticNet = addLayers(criticNet,commonPath1);
criticNet = connectLayers(criticNet,'CriticStateFC2','add/in1');
criticNet = connectLayers(criticNet,'CriticActionFC1','add/in2');


criticOptions = rlRepresentationOptions('Optimizer','adam','LearnRate',1e-3,...
                                        'UseDevice',"gpu",...
                                        'GradientThreshold',1,'L2RegularizationFactor',2e-4);
critic = rlQValueRepresentation(criticNet,obsInfo,actInfo,...
    'Observation',{'observation'},'Action',{'action'},criticOptions);

% Actor
actorNetwork = [
    sequenceInputLayer(numObs,'Normalization','zscore', 'Mean', observation_mean, ...
        'StandardDeviation', observation_std,'Name','observation')
    fullyConnectedLayer(400,'Name','ActorFC1')
    reluLayer('Name','ActorRelu1')
    fullyConnectedLayer(300,'Name','ActorFC2')
    reluLayer('Name','ActorRelu2')

    lstmLayer(8,'OutputMode','sequence','Name','lstm')
    reluLayer('Name','ActorCommonRelu1')
    fullyConnectedLayer(numAct,'Name','ActorFC3')
    softplusLayer('Name','ActorOutput')
];

actorNetwork = layerGraph(actorNetwork);
actorOptions = rlRepresentationOptions('Optimizer','adam','LearnRate',1e-3,...
                                       'UseDevice',"gpu",...
                                       'GradientThreshold',1,'L2RegularizationFactor',1e-5);

actor = rlDeterministicActorRepresentation(actorNetwork,obsInfo,actInfo, ...
    'Observation',{'observation'},'Action',{'ActorOutput'},actorOptions);
```

Listing A.15: agent_ddpg_v1_norm.m

A.8 RL Simulation function

When agent training is terminated, we can simulate its performance using the `sim_confident` function:

```
values = sim_confident(agent, envProcess, trainStats, sequence_length);
```

towards a selected environment, e.g. the `ProcessEnvironment` as shown here.

```
function values = sim_confident(agent, env, trainStats, maxStepsPerEpisode)
    obs = reset(env);
    actor = getActor(agent);
    critic = getCritic(agent);
    maxQValue = max(trainStats.EpisodeQ0);
    values = zeros(1,maxStepsPerEpisode);

    for stepCt = 1:maxStepsPerEpisode

        % Select action according to trained policy
        action = getAction(actor,{obs});

        if isa(agent,'rl.agent.rlSACAgent')
            value = getValue(critic(1),{obs},action); % SAC
        end
        if isa(agent,'rl.agent.rlPPOAgent')
            value = getValue(critic,{obs}); % PPO
        end
        values(stepCt) = value / maxQValue;

        % Step the environment
        [nextObs,reward,isdone] = step(env,action{1});

        % Check for terminal condition
%         if isdone
%             break
%         end

        obs = nextObs;
    end
```

Listing A.16: sim_confident.m

A.9 Cascade demonstration environment

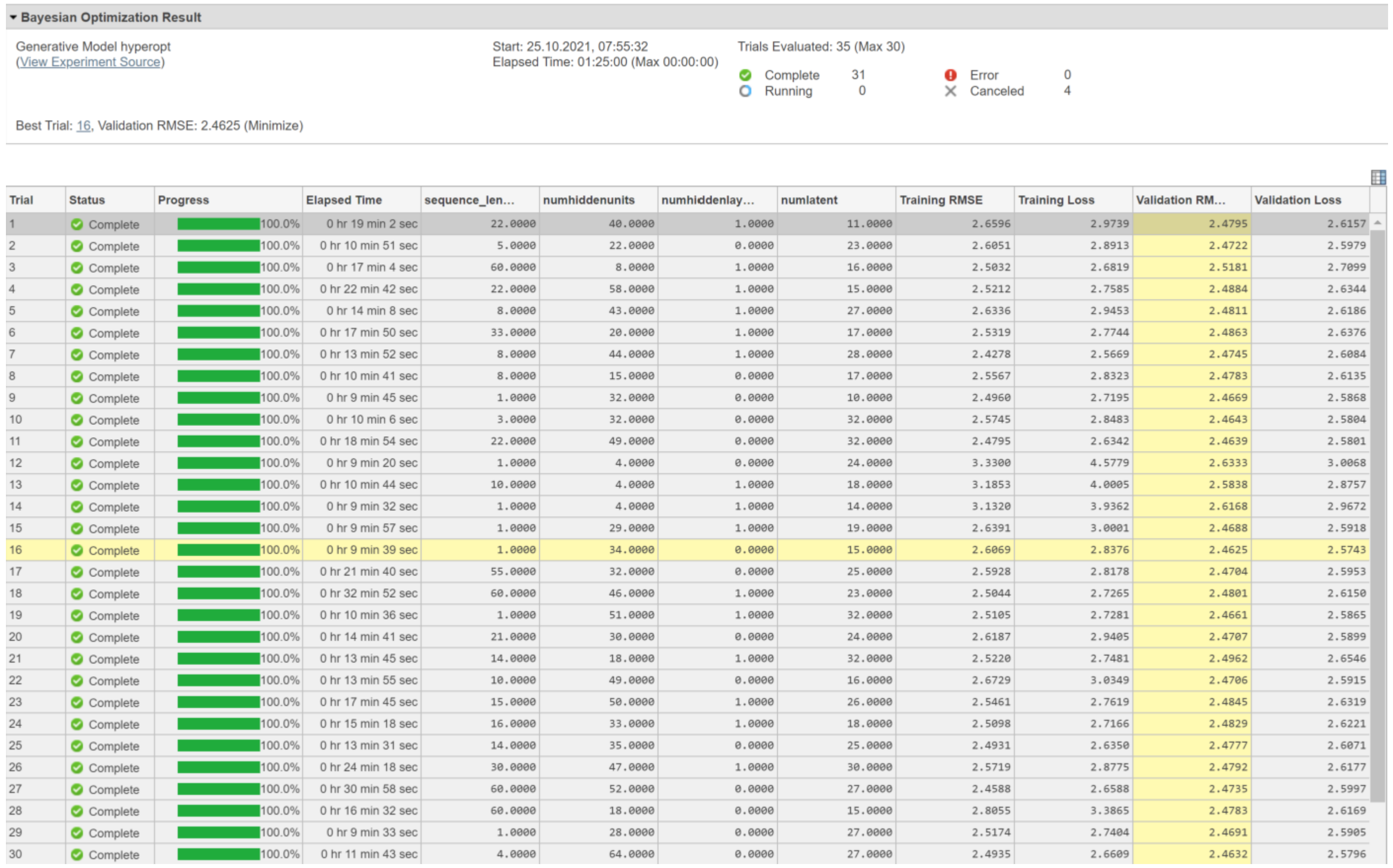
```
classdef CascadeEnvironment < rl.env.MATLABEnvironment

    properties
        sp
        s
        s1
        s2
        s1l
        s2l
        d1
        d2
        u1
        u2
        t
    end

    methods
        % Contructor method creates an instance of the environment
        function this = CascadeEnvironment()
            ObservationInfo = rlNumericSpec([2 1]);
            ActionInfo = rlNumericSpec([2 1],'LowerLimit',-3,'UpperLimit',3);
            this = this@rl.env.MATLABEnvironment(ObservationInfo,ActionInfo);
        end

        function [Observation,Reward,IsDone,LoggedSignals] = step(this,Action)
            LoggedSignals = [];
            this.t = this.t + 1;

            u1 = Action(1); % outer regulator P
            u2 = Action(2); % inner regulator P

            this.s = this.s + u1 + u2;

            this.s = this.P2(this.s) + this.d2(this.t);
            this.s2 = this.s;
            Observation(1) = this.s;
            this.s2l = [this.s2l this.s];

            this.s = this.P1(this.s) + this.d1(this.t);
            this.s1 = this.s;
            Observation(2) = this.s;
            this.s1l = [this.s1l this.s];

```

```matlab
            Reward=1/abs(((this.sp - this.s)^2) + (0.01*(Action(1)^2)) + (0.001*(Action(2)^2)));
            IsDone = 0;

            Observation = Observation';
            notifyEnvUpdated(this);
        end

        % Reset environment to initial state and output initial observation
        function InitialObservation = reset(this)
            if ~isempty(this.s1l)
                clf
                hold on
                plot(this.s2l,'--',"LineWidth",1)
                plot(this.s1l,"LineWidth",1)
                ylim([0 (this.sp*3)])
                legend("Inner-P2","Out-P1")
                grid
            end


            this.sp = 1;
            this.s = 0;
            this.s1l = [];
            this.s2l = [];
            this.s1=0;
            this.s2=0;
            this.d1 = getsine(1,0.1,1);
            this.d2 = getsine(1,0.1,0.5); % phase shift
            this.u1 = 0;
            this.u2 = 0;
            this.t = 0;

            InitialObservation = {[this.s2; this.s1]};

            notifyEnvUpdated(this);
        end

        function y=P2(this, x)
            y=3./(x+2);
        end
        function y=P1(this, x)
            y=10./((x+1).^3);
        end
    end

end
```

Listing A.17: CascadeEnvironment.m

Appendix B

Bayesian optimization run



▾ Bayesian Optimization Result

Generative Model hyperopt
(View Experiment Source)

Start: 25.10.2021, 07:55:32
Elapsed Time: 01:25:00 (Max 00:00:00)

Trials Evaluated: 35 (Max 30)

✓ Complete	31	! Error	0
○ Running	0	✕ Canceled	4

Best Trial: 16, Validation RMSE: 2.4625 (Minimize)

Trial	Status	Progress	Elapsed Time	sequence_len...	numhiddenunits	numhiddenlay...	numlatent	Training RMSE	Training Loss	Validation RM...	Validation Loss
1	✓ Complete	100.0%	0 hr 19 min 2 sec	22.0000	40.0000	1.0000	11.0000	2.6596	2.9739	2.4795	2.6157
2	✓ Complete	100.0%	0 hr 10 min 51 sec	5.0000	22.0000	0.0000	23.0000	2.6051	2.8913	2.4722	2.5979
3	✓ Complete	100.0%	0 hr 17 min 4 sec	60.0000	8.0000	1.0000	16.0000	2.5032	2.6819	2.5181	2.7099
4	✓ Complete	100.0%	0 hr 22 min 42 sec	22.0000	58.0000	1.0000	15.0000	2.5212	2.7585	2.4884	2.6344
5	✓ Complete	100.0%	0 hr 14 min 8 sec	8.0000	43.0000	1.0000	27.0000	2.6336	2.9453	2.4811	2.6186
6	✓ Complete	100.0%	0 hr 17 min 50 sec	33.0000	20.0000	1.0000	17.0000	2.5319	2.7744	2.4863	2.6376
7	✓ Complete	100.0%	0 hr 13 min 52 sec	8.0000	44.0000	1.0000	28.0000	2.4278	2.5669	2.4745	2.6084
8	✓ Complete	100.0%	0 hr 10 min 41 sec	8.0000	15.0000	0.0000	17.0000	2.5567	2.8323	2.4783	2.6135
9	✓ Complete	100.0%	0 hr 9 min 45 sec	1.0000	32.0000	0.0000	10.0000	2.4960	2.7195	2.4669	2.5868
10	✓ Complete	100.0%	0 hr 10 min 6 sec	3.0000	32.0000	0.0000	32.0000	2.5745	2.8483	2.4643	2.5804
11	✓ Complete	100.0%	0 hr 18 min 54 sec	22.0000	49.0000	0.0000	32.0000	2.4795	2.6342	2.4639	2.5801
12	✓ Complete	100.0%	0 hr 9 min 20 sec	1.0000	4.0000	0.0000	24.0000	3.3300	4.5779	2.6333	3.0068
13	✓ Complete	100.0%	0 hr 10 min 44 sec	10.0000	4.0000	1.0000	18.0000	3.1853	4.0005	2.5838	2.8757
14	✓ Complete	100.0%	0 hr 9 min 32 sec	1.0000	4.0000	1.0000	14.0000	3.1320	3.9362	2.6168	2.9672
15	✓ Complete	100.0%	0 hr 9 min 57 sec	1.0000	29.0000	1.0000	19.0000	2.6391	3.0001	2.4688	2.5918
16	✓ Complete	100.0%	0 hr 9 min 39 sec	1.0000	34.0000	0.0000	15.0000	2.6069	2.8376	2.4625	2.5743
17	✓ Complete	100.0%	0 hr 21 min 40 sec	55.0000	32.0000	0.0000	25.0000	2.5928	2.8178	2.4704	2.5953
18	✓ Complete	100.0%	0 hr 32 min 52 sec	60.0000	46.0000	1.0000	23.0000	2.5044	2.7265	2.4801	2.6150
19	✓ Complete	100.0%	0 hr 10 min 36 sec	1.0000	51.0000	1.0000	32.0000	2.5105	2.7281	2.4661	2.5865
20	✓ Complete	100.0%	0 hr 14 min 41 sec	21.0000	30.0000	0.0000	24.0000	2.6187	2.9405	2.4707	2.5899
21	✓ Complete	100.0%	0 hr 13 min 45 sec	14.0000	18.0000	1.0000	32.0000	2.5220	2.7481	2.4962	2.6546
22	✓ Complete	100.0%	0 hr 13 min 55 sec	10.0000	49.0000	0.0000	16.0000	2.6729	3.0349	2.4706	2.5915
23	✓ Complete	100.0%	0 hr 17 min 45 sec	15.0000	50.0000	1.0000	26.0000	2.5461	2.7619	2.4845	2.6319
24	✓ Complete	100.0%	0 hr 15 min 18 sec	16.0000	33.0000	1.0000	18.0000	2.5098	2.7166	2.4829	2.6221
25	✓ Complete	100.0%	0 hr 13 min 31 sec	14.0000	35.0000	0.0000	25.0000	2.4931	2.6350	2.4777	2.6071
26	✓ Complete	100.0%	0 hr 24 min 18 sec	30.0000	47.0000	1.0000	30.0000	2.5719	2.8775	2.4792	2.6177
27	✓ Complete	100.0%	0 hr 30 min 58 sec	60.0000	52.0000	0.0000	27.0000	2.4588	2.6588	2.4735	2.5997
28	✓ Complete	100.0%	0 hr 16 min 32 sec	60.0000	18.0000	0.0000	15.0000	2.8055	3.3865	2.4783	2.6169
29	✓ Complete	100.0%	0 hr 9 min 33 sec	1.0000	28.0000	0.0000	27.0000	2.5174	2.7404	2.4691	2.5905
30	✓ Complete	100.0%	0 hr 11 min 43 sec	4.0000	64.0000	0.0000	27.0000	2.4935	2.6609	2.4632	2.5796

Figure B.1: Matlab Experiment Manager for Bayesian optimization of generative model hyperparameters

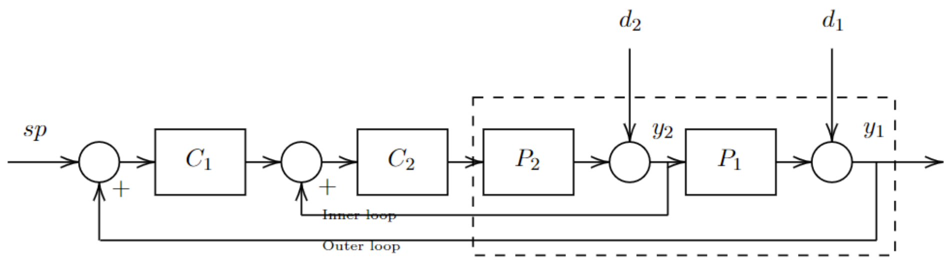
Appendix C

Demonstration of cascade implementation

Cascade control

https://se.mathworks.com/help/control/ug/designing-cascade-control-system-with-pi-controllers.html

https://se.mathworks.com/help/reinforcement-learning/ug/tune-pi-controller-using-td3.html



We define two fictual sequential processes P2 and P1, where the inner loop process P2 is

$$P_2(s) = \frac{3}{s+2}$$

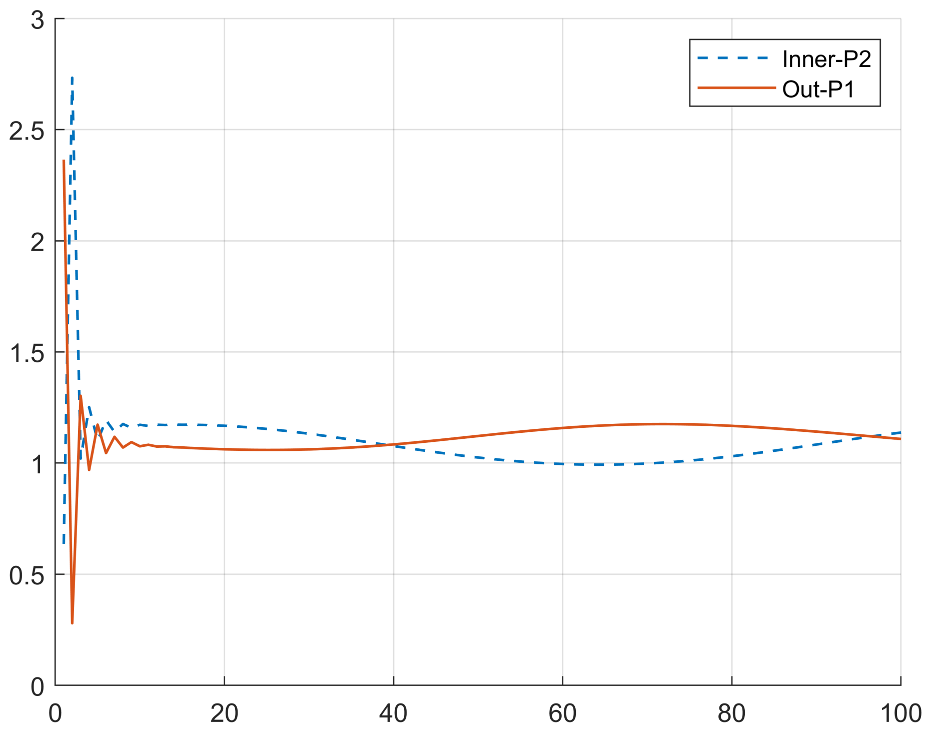
and P1

$$P_1(s) = \frac{10}{(10+s)^3}$$

Each of the processes as disturbed by d1 and d2, which are two phase shifted sine-functions.

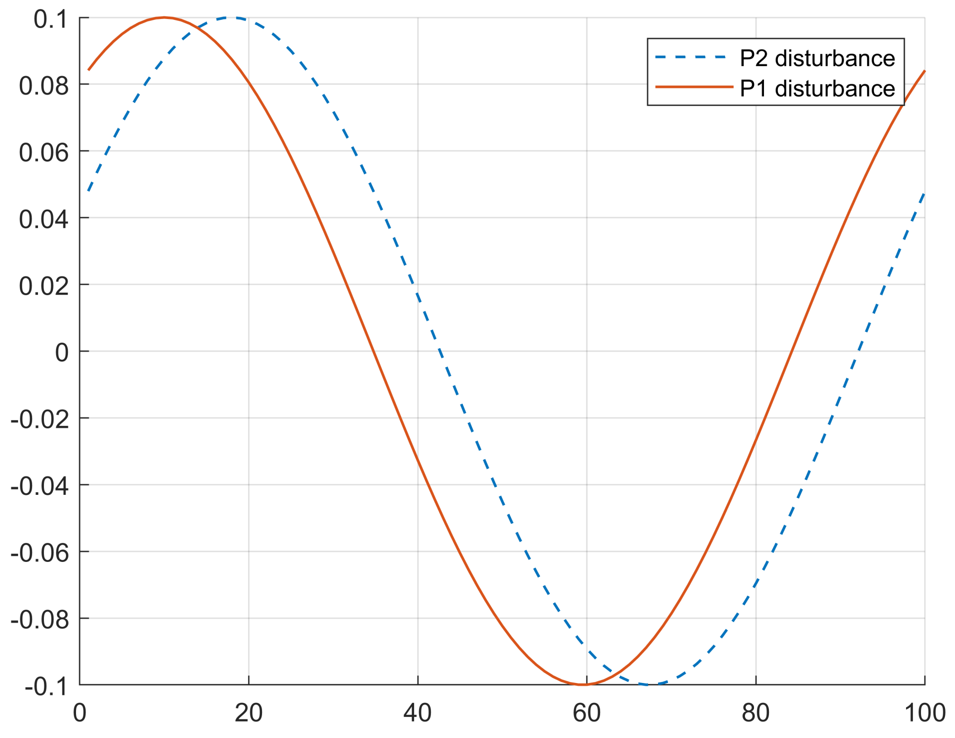
Regulate by P

```
figure, hold on
sp = 1;
s = 0;
s1l = [];
s2l = [];
s1=0;
s2=0;
d1 = getsine(1,0.1,1);
d2 = getsine(1,0.1,0.5); % phase shift
u1 = 0;
u2 = 0;
regulate = true;
for t=1:100

    if regulate
        % manually tuned from unregulated process
        u1 = (2.5286*(sp-s1)); % outer regulator P
        u2 = (0.5666*(sp-s2)); % inner regulator P
    end

    s = s + u1 + u2;

    s = P2(s) + d2(t);
    s2 = s;
    s2l = [s2l s];

    s = P1(s) + d1(t);
    s1 = s;
    s1l = [s1l s];

end

plot(s2l,'--',"LineWidth",1)
plot(s1l,"LineWidth",1)
legend("Inner-P2","Out-P1")
grid
```



Process disturbances

```
figure, hold on
plot(d2,'--',"LineWidth",1)
plot(d1,"LineWidth",1)
legend("P2 disturbance","P1 disturbance")
grid
```



Train agent

```
env = CascadeEnvironment();
validateEnvironment(env);
env.reset;
sequence_length = 10;
```

SAC agent

```
[actor, critic1, critic2] = agent_sac_v1(env);
agentOptions = rlSACAgentOptions(...
    "SequenceLength", sequence_length,...
    "DiscountFactor",0.99,...
    "TargetSmoothFactor",1e-3,...
    "ExperienceBufferLength", 1e6,...
    "UseDeterministicExploitation", true, ...
    "ResetExperienceBufferBeforeTraining", false,...
    "SaveExperienceBufferWithAgent", true);
agent = rlSACAgent(actor,[critic1 critic2],agentOptions);
```

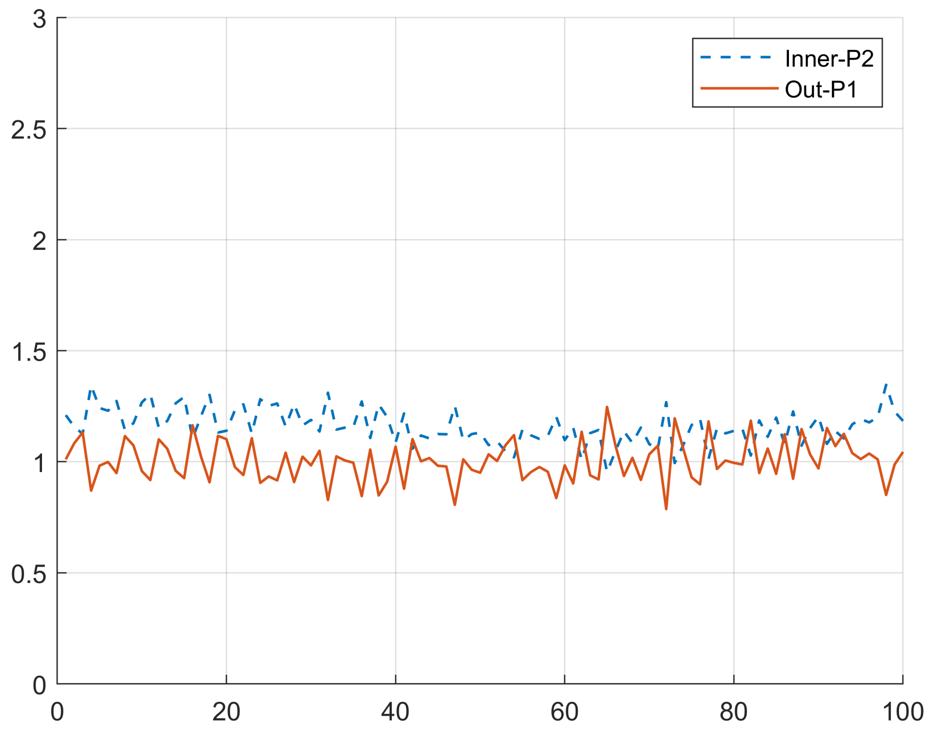
PPO agent

```
[actor, critic] = agent_ppo_v1(env);
agentOpts = rlPPOAgentOptions(...
    "MiniBatchSize", sequence_length,...
    'ExperienceHorizon',1e6,...
    'DiscountFactor',0.99,...
    "UseDeterministicExploitation", true);
agent = rlPPOAgent(actor,critic,agentOpts);
```

DDPG agent

```
[actor, critic] = agent_ddpg_v1(env);
agentOpts = rlDDPGAgentOptions(...
    "SequenceLength", sequence_length,...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6,...
    'DiscountFactor',0.99,...
    "ResetExperienceBufferBeforeTraining", false,...
    "SaveExperienceBufferWithAgent", true);
agent = rlDDPGAgent(actor,critic,agentOpts);
```

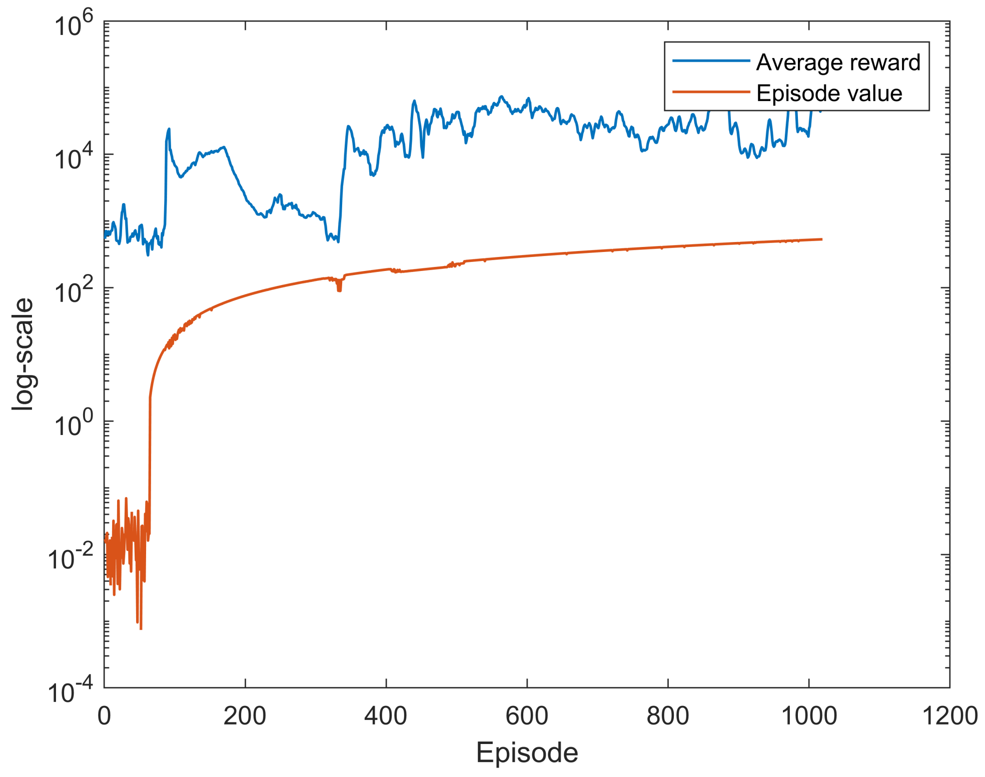
```
opt = rlTrainingOptions(...
    'MaxEpisodes',5000, ...
    'MaxStepsPerEpisode',100);
trainStats = train(agent,env,opt);
```



Episode plots

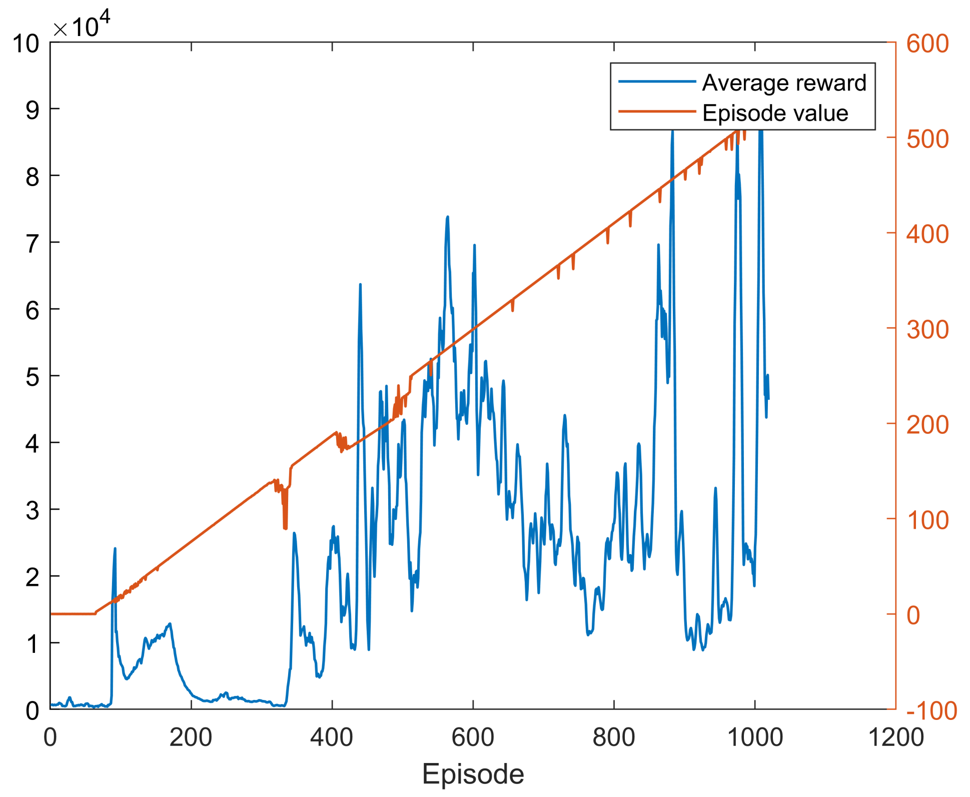
```
figure
plot(trainStats.AverageReward,'LineWidth',1); hold on;
plot(trainStats.EpisodeQ0,'LineWidth',1); hold off;
xlabel("Episode")
```

```
ylabel("log-scale")
set(gca, 'YScale', 'log')
legend(["Average reward","Episode value"])
```



```
Warning: Negative data ignored
```

```
figure
plot(trainStats.AverageReward,'LineWidth',1);
yyaxis right;
plot(trainStats.EpisodeQ0,'LineWidth',1);
xlabel("Episode")
legend(["Average reward","Episode value"])
```

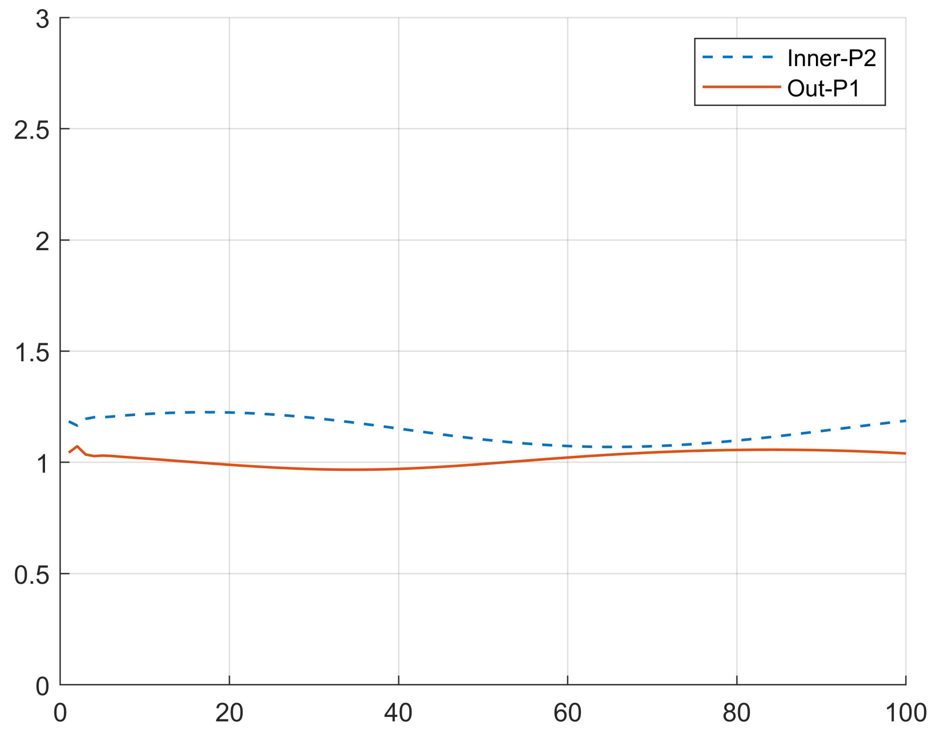


```
numel(trainStats.EpisodeIndex) % episodes
trainStats.EpisodeQ0(end) % value
trainStats.AverageReward(end) % total reward
```

```
%save("savedAgents/2021-12-11-cascade-sac.mat",'agent','trainStats')
%load("savedAgents/2021-12-10-cascade-ppo.mat",'agent','trainStats')
%save("savedAgents/2021-12-10-cascade-ddpg.mat",'agent','trainStats')
```

Simulate

```
env.reset;
simOpts = rlSimulationOptions('MaxSteps',100);
experience = sim(env,agent,simOpts);
env.reset;
```



```
function y=P2(x)
    y=3./(x+2);
end
function y=P1(x)
    y=10./((x+1).^3);
end
```

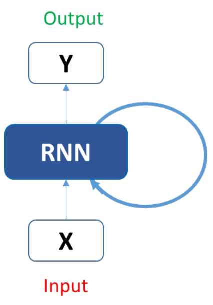
Appendix D

Demonstration of RNN implementation

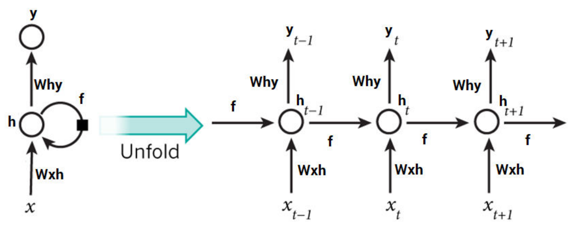
Recurrent Neural Network from Scratch

References:

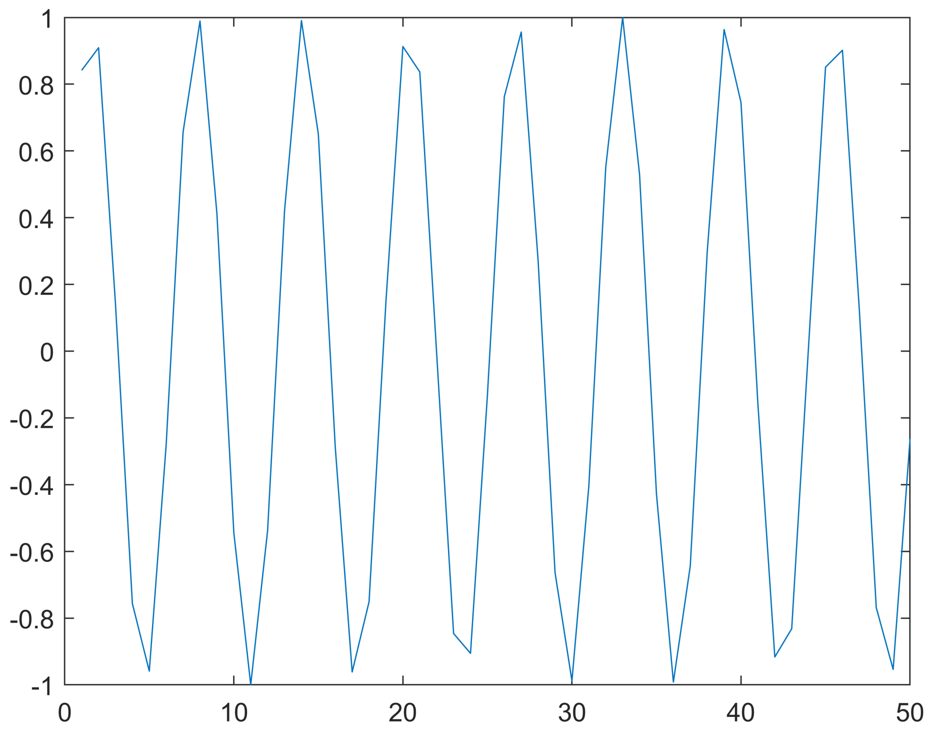
- https://www.analyticsvidhya.com/blog/2019/01/fundamentals-deep-learning-recurrent-neural-networks-scratch-python/
- https://towardsdatascience.com/recurrent-neural-networks-rnns-3f06d7653a85



Unfold:



```
rng(0);
figure
sin_wave = sin([1:200]);
plot(sin_wave(1:50))
```



Prepare dataset

```
X = {};
Y = {};

seq_len = 50;
num_records = numel(sin_wave) - seq_len;

for i=1:num_records - seq_len
    X(i) = {sin_wave(i:i+seq_len-1)};
    Y(i) = {sin_wave(i+seq_len)};
end
```

Create the Architecture for RNN model

```
learning_rate = 0.0001;
nepoch = 25;
T = seq_len;                     % length of sequence
hidden_dim = 4; % neurons
output_dim = 1;

bptt_truncate = 5;
min_clip_value = -10;
max_clip_value = 10;
```

Define the weights of the network

$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}
$$

```
avg = 0;
sigma = 1;

U = normrnd(avg, sigma, hidden_dim, T);
W = normrnd(avg, sigma, hidden_dim, hidden_dim);
V = normrnd(avg, sigma, output_dim, hidden_dim);
```

Train model

```
for epoch=1:nepoch
    loss = 0.0;

    for i=1:size(Y,2)
        x = X{i};
        y = Y{i};
        prev_h = zeros(hidden_dim, 1);
        layers = {};

        dU = zeros(size(U));
        dV = zeros(size(V));
        dW = zeros(size(W));

        dU_t = zeros(size(U));
        dV_t = zeros(size(V));
        dW_t = zeros(size(W));

        dU_i = zeros(size(U));
        dW_i = zeros(size(W));

        % do a forward pass to get prediction
        for t=1:T
            new_input = zeros(size(x))';
            new_input(t) = x(t);
            mulu = (U * new_input);
            mulw = (W * prev_h);
            add = mulw + mulu;
            h = sigmoid2(add);
            o = V * h;
            layers{t} = {h, prev_h};
            prev_h = h;
        end

        % Truncated back propagation through time (TBPTT)

        % derivative of pred
        dmulv = (o - y);

        % backward pass
        for t=1:T
            l = layers{t};
            dV_t = dmulv * l{1}'; % s
            dsv = V' * dmulv;

            ds = dsv;
            dadd = add .* (1 - add) .* ds;

            dmulw = dadd .* ones(size(mulw));

            dprev_s = W' * dmulw;

            for i=t-1:-1:max(-1, t-bptt_truncate-1)
                ds = dsv + dprev_s;
                dadd = add .* (1 - add) .* ds;

                dmulw = dadd .* ones(size(mulw));
                dmulu = dadd .* ones(size(mulu));

                dW_i = W * l{2}; % prev_s
                dprev_s = W' * dmulw;

                new_input = zeros(size(x))';
                new_input(t) = x(t);
                dU_i = U * new_input;
                dx = U' * dmulu;

                dU_t = dU_t + dU_i;
                dW_t = dW_t + dW_i;
            end

            dV = dV + dV_t;
            dU = dU + dU_t;
            dW = dW + dW_t;

            if max(dU) > max_clip_value
                dU(dU > max_clip_value) = max_clip_value;
            end
            if max(dV) > max_clip_value
                dV(dV > max_clip_value) = max_clip_value;
            end
            if max(dW) > max_clip_value
                dW(dW > max_clip_value) = max_clip_value;
            end

            if min(dU) < min_clip_value
                dU(dU < min_clip_value) = min_clip_value;
            end
            if min(dV) < min_clip_value
                dV(dV < min_clip_value) = min_clip_value;
            end
            if min(dW) < min_clip_value
                dW(dW < min_clip_value) = min_clip_value;
            end

        end

        % update
        U = U - (learning_rate * dU);
        V = V - (learning_rate * dV);
        W = W - (learning_rate * dW);

        % calculate error
        loss_per_record = (y - o).^2 / 2;
        loss = loss + sum(loss_per_record);
    end

    loss = loss / size(y,2);
    disp(['Epoch ' num2str(epoch) ' loss ' num2str(loss)])
end
```

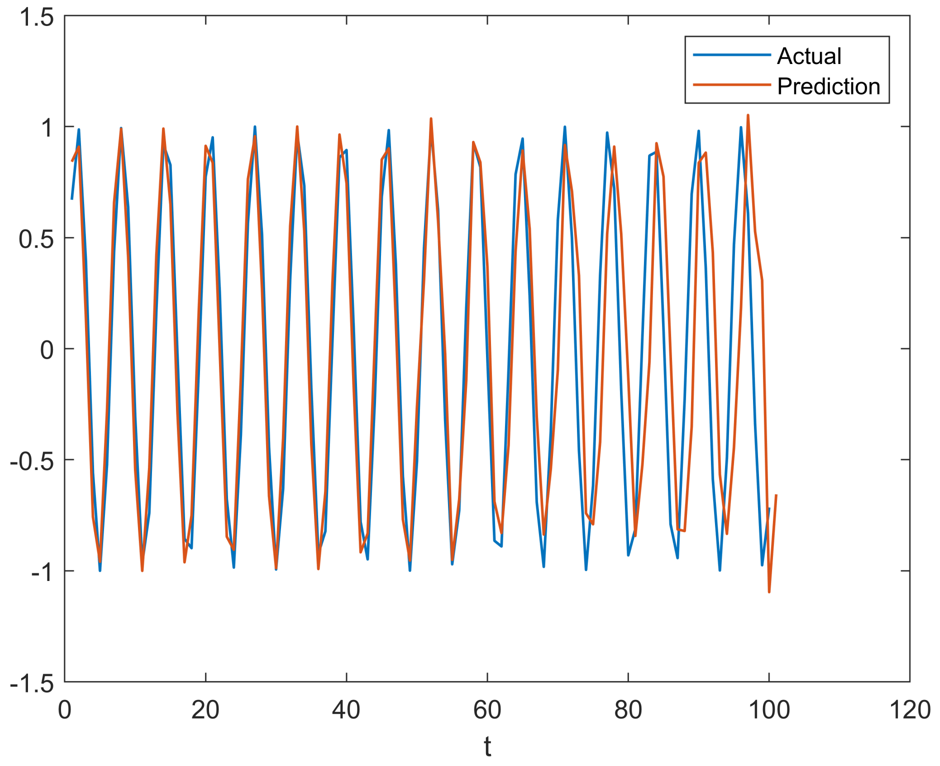
```
Epoch 1 loss 39.4363
Epoch 2 loss 38.2377
Epoch 3 loss 36.5682
Epoch 4 loss 35.6673
Epoch 5 loss 35.2582
Epoch 6 loss 35.0073
Epoch 7 loss 34.8418
Epoch 8 loss 34.7373
Epoch 9 loss 34.6772
Epoch 10 loss 34.6488
Epoch 11 loss 34.6437
Epoch 12 loss 34.6551
Epoch 13 loss 34.678
Epoch 14 loss 34.7085
Epoch 15 loss 34.7437
Epoch 16 loss 34.7814
Epoch 17 loss 34.82
Epoch 18 loss 34.8582
Epoch 19 loss 34.8953
Epoch 20 loss 34.9307
Epoch 21 loss 34.964
Epoch 22 loss 34.9949
Epoch 23 loss 35.0234
Epoch 24 loss 35.0495
Epoch 25 loss 35.0732
```

Predictions

One-step prediction

```
preds = [];
for i=1:size(Y,2)
    x = X{i};
    prev_h = zeros(hidden_dim, 1);

    % Forward pass
    for t=1:T
        mulu = U * x';
        mulw = W * prev_h;
        add = mulw + mulu;
        h = sigmoid2(add);
        mulv = V * h;
        prev_h = h;
    end

    preds(i) = mulv;
end
```

```
figure
ass = cell2mat(Y);
plot(ass,"LineWidth",1); hold on;
plot(preds,"LineWidth",1);
legend("Actual","Prediction")
xlabel("t")
```



```
U
```

```
U = 4×50
     0.5237    0.3048    3.5644    0.7114   -0.1381    0.6575    0.4749    0.2799 ···
     1.8361   -1.3055    2.7716   -0.0608    1.4919   -1.2053    1.0369   -0.7851
    -2.2491   -0.4239   -1.3402    0.7245    1.4187    0.7270    0.7366    0.8981
     0.8622    0.3426    3.0349   -0.2050    1.4172    1.6302   -0.3034   -1.1471
```

```
W
```

```
W = 4×4
```

```
    0.3630    0.3149    0.0174   -0.6960
   -0.7421    0.8030    0.1417   -0.1961
    0.9570    0.2692   -0.5242   -0.7042
    0.3392   -0.9093    1.7142   -1.1421
```

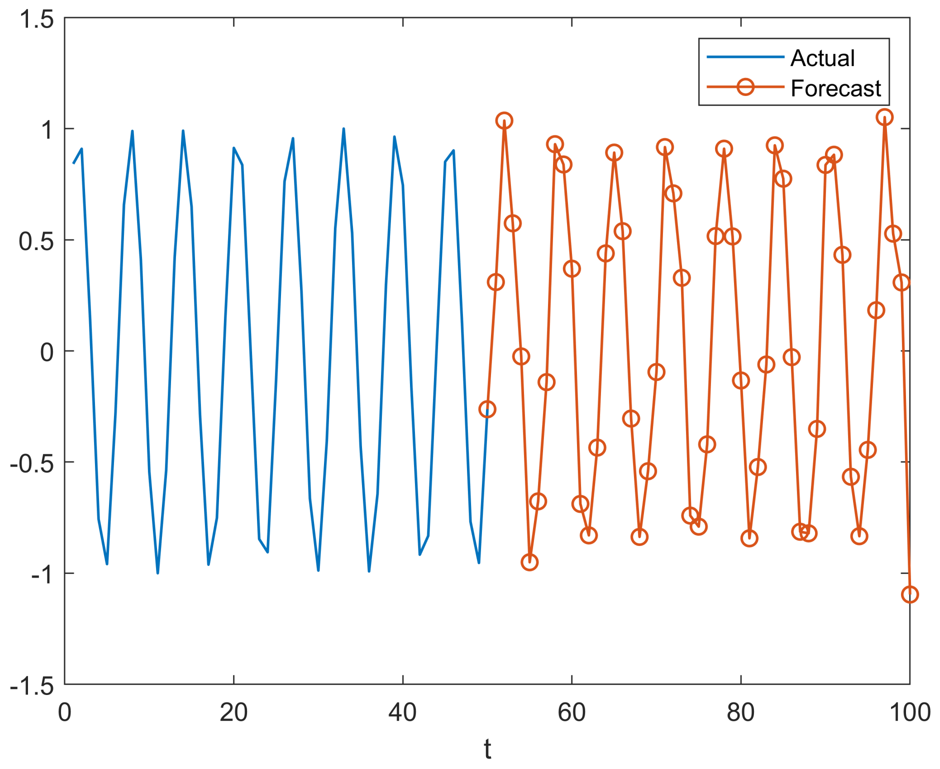
```
V
```

```
V = 1×4
   -0.3851   -0.4618    1.3429   -0.4377
```

Forecast

```
preds = X{1}; % start sequence
length=size(preds,2);
for i=length:length+50
    prev_h = zeros(hidden_dim, 1);
    x = preds(end-length+1:end);

    % Forward pass
    for t=1:T
        mulu = U * x';
        mulw = W * prev_h;
        add = mulw + mulu;
        h = sigmoid2(add);
        mulv = V * h;
        prev_h = h;
    end

    preds(i+1) = mulv;
end
```

```
figure
% ass = cell2mat(Y);
plot(1:50, X{1},"LineWidth",1); hold on;
plot(50:100,preds(50:100),'-o', "LineWidth",1);
legend("Actual","Forecast")
xlabel("t")
```



```
function sig = sigmoid2(x)
    sig = 1 ./ (1 + exp(-x));
end
```